Studies in Computational Intelligence 498

Krzysztof Grąbczewski

Meta-Learning in Decision Tree Induction

Springer

Studies in Computational Intelligence

Volume 498

Series Editor

J. Kacprzyk, Warsaw, Poland

For further volumes:
http://www.springer.com/series/7092

Krzysztof Grąbczewski

Meta-Learning in Decision Tree Induction

Springer

Krzysztof Grąbczewski
Department of Informatics,
Faculty of Physics,
Astronomy and Informatics
Nicolaus Copernicus University
Toruń
Poland

ISSN 1860-949X ISSN 1860-9503 (electronic)
ISBN 978-3-319-00959-9 ISBN 978-3-319-00960-5 (eBook)
DOI 10.1007/978-3-319-00960-5
Springer Cham Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013940294

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To my Family. In other words, to Love, the genuine Love

Love is patient, love is kind. It is not jealous, (love) is not pompous, it is not inflated, it is not rude, it does not seek its own interests, it is not quick-tempered, it does not brood over injury, it does not rejoice over wrongdoing but rejoices with the truth.

1 Cor 13,4–6

God is love, and whoever remains in love remains in God and God in him.

1 J 4,16b

Foreword

The number of different machine learning methods has grown over the past years and so the user faced with the question of which method he/she should use on a given problem. The problem is aggravated by the fact that many machine algorithms require that parameters should be set prior to their application, and besides, given data may be pre-processed in many different ways. The aim of the area of metalearning is to facilitate the task of selecting, adapting or composing machine learning, or data mining solutions. These are referred to in the book as *learning machines*. As these are normally composed of different constituents, these are identified also as *configurations*. These can be compared to what others call *data mining workflows*.

This book represents various interesting contributions to this area. Although it focuses on different variants of decision tree induction, the meta-learning approach described is quite general. It is applicable to other types of machine learning algorithms. The part of the book that discusses different variants of decision tree induction represents a useful source of information in itself to whoever wishes to review some of the techniques used in decision tree learning, as well as different ensemble methods that involve decision trees.

The author rightly argues that the knowledge of different components used within decision tree learning needs to be systematized to enable the system to generate and evaluate different variants of machine learning algorithms with the aim of identifying the top most performers or potentially the best one. A unified view of decision tree learning enables to emulate different decision tree algorithms simply by setting certain parameters. As metalearning requires running many different processes with the aim of obtaining performance results, a detailed description of the experimental methodology and evaluation framework is provided.

Metalearning is discussed in great detail in the second half of the book. The exposition starts by presenting a comprehensive review of many meta-learning approaches explored in the past described in the literature, including for instance approaches that provide a ranking of algorithms. The author distinguishes between so-called passive methods and active ones.

Passive methods rely on the information gathered in a prior phase. Different configurations are obtained with the help of configuration generator and the candidate configurations are accompanied by estimates of time and memory. The configurations that require less time and memory are ranked higher than those that require more resources. The aim of such techniques is to identify the most promising machine configurations with a high probability. The approach described can be related to other work that exploits planning whose aim is to construct data mining workflows. The book stimulates thus interchange of ideas between different, albeit related, approaches.

Active methods maintain a profile of different configurations which is updated as more tests have been carried out. The system uses a sophisticated search process driven by heuristics to identify processes/machine-learning configurations that should be paid attention to and tested. Active methods provide a more practical solution to real problems permitting that profiles be gradually built up as more experience is gained and more results are gathered. It is expected that these will gain even more importance in the future. The method is discussed in detail in the last part of the book and represents an interesting contribution.

Metalearning is an active area of research and many publications exist. The value of the book is in that it presents a unified view of different aspects of metalearning. It systematizes the existing knowledge and presents a distinct conceptual framework that facilitates further advances.

Porto, April 2013 Pavel Brazdil

Preface

Miscellaneous learning machines have been designed and implemented for decades of computational intelligence (CI) research. More and more solutions are being published all the time in all the subdomains like classification, regression, clustering, and others. A natural question arises: how many algorithms to solve the same goals must be proposed, before we regard the available set of tools as satisfactory? One could claim that such time will never come, because there will always be a possibility of improvement in various aspects, various contexts of applications. This is certainly true, but on the other side, it seems quite reasonable to claim something completely opposite: we already have so many tools that, to solve new tasks, we no longer need new methods, but a robust knowledge about how to use the tools we have already got. Probably, each real-world task fitting the domain of computational intelligence can be solved by available tools in a way close to optimal (statistically insignificantly different than optimal), but it does not mean that finding a satisfactory solution is easy. Therefore, more effort of the CI community should be focused on meta-level algorithms, capable of finding attractive solutions with already available tools, than on development of new learning methods that can solve some new, special types of problems.

This book is focused on learning decision tree (DT) models. One of its goals is to show that even the part of CI devoted to DT induction is large enough to make the pursuit of the most successful learning machine a very complex problem. To be successful in assigning adequate learning machines for particular learning problems, we need more systematic research in the field. The number of available solutions is even larger than we usually think, because each learning machine consists of many components and each of them can be easily replaced by many other compatible modules. Only well designed, general architectures of learning machines reveal the real level of complexity of the problem, because in such universal and flexible frameworks, available components may be combined in huge numbers of ways. No human expert can try all these combinations when solving a problem, so even the best experts are likely to miss quite simple and attractive solutions. Therefore, we need automated tools performing reliable search processes in the space of possible learning machines. Because all possible machines cannot be tested, we need trustworthy metaknowledge helpful in recognizing the more and the less interesting algorithms in various contexts. Starting

with the metaknowledge expressed by human experts and gathering new knowledge, extracted from extensive, automated search processes, may easily bring large knowledge bases of great value for automated expert systems. The metaknowledge may be used not only to select probably the most accurate learning machines, but also to construct new complex learning machines. The amount of such knowledge is so large that very soon, no human expert will be able to use it in an optimal way. Therefore, in many fields, the era of human designed models is close to its end—automated learners will soon outperform humans as they already do in playing chess, because in data modeling, also so large databases of metaknowledge must be searched through and analyzed in appropriate order, that it gets unfeasible for humans, but more and more eligible for automated learners.

The goal of the book is to propose some steps in the direction that seems the most adequate: toward easy to explore taxonomy of DT induction methods and automated tools for construction of successful DT learning machines.

Efficient exploitation of available algorithms and even construction of new algorithms are possible only when the nature of the methods is perfectly understood. Therefore, before writing the book got started, plenty of algorithms had been analyzed, redesigned, implemented, and validated. It is not easy to implement faithfully even the most popular algorithms, because some detailed solutions are often kept hidden, but sometimes they are quite significant. Thus, apart from the analysis of available descriptions (not always clear and exhaustive), sometimes, the source codes had to be analyzed to discover some detailed solutions. In some other cases, when the source codes were not available, some elements of algorithm definition had to be guessed or inferred from the analysis of outputs generated by the binary codes.

Chapter 2 is the result of the work in this area done for many years. It presents many algorithms (more and less popular), described from the point of view of a scientist trying to discover all important aspects of the solutions. Therefore, it is not just a review of selected methods, but a survey of the domain with thorough analysis of advantages and drawbacks, possibilities and limitations of many ideas applied to DT induction. The goal of this chapter was to describe the most important solutions shortly but quite exhaustively, intuitively, in common language, but also with formal statements when necessary to make the algorithms unambiguous. This should allow the readers easily understand the algorithms in relatively short time.

After the survey of DT induction research, the book presents a unified view of the algorithms (Chap. 3) that facilitates easy combination of many compatible components into new learning machines. Such a framework is absolutely necessary for advanced meta-learning purposes.

Advanced metalearning requires a robust environment for efficient running machine processes, conducting complex tests, collecting, and analysis of the results. Chapter 4 presents the most important solutions of Intemi system, designed and implemented especially for the purposes of metalearning. It arose from many years of experience with learning machines and very deep analysis of metalearning requirements.

Some aspects of meta-level analysis of DT components in action are presented in Chap. 5, where the problems of reliable testing are discussed, some experiments designed, and their results collected and compared. Adequate methods of result visualization have been worked out to facilitate reliable conclusions. Appropriate visualization of results is very important for humans, but gets quite difficult when tens of thousands of results need to be presented together.

The coping stone of the work described in the book is the meta-learning approach based on search and validation, presented in Chap. 6 with two particular implementations of the idea: One based on machine configuration generators and complexity control, and one dealing with learning results profiles.

Chapter 7 discusses the possible future of meta-learning research and points the directions that seem the most promising and most adequate to follow.

Two appendices contain some descriptions of basic statistical and algebraic methods often used in DT induction. They have been estimated as useful for each inquisitive novice in the field, who may want to get more details without searching outside of the book.

The contents of the book have been prepared thanks to many years of research on DT induction, on learning machines in general, and recently on metalearning. Some fragments of the book review the work of numerous authors, published in miscellaneous articles. The original papers are always pointed as the sources of the information. Some other fragments concern ideas presented by the book author in some publications, but here they are displayed in the context of this monograph—subordinate to the goal of the book, which is presenting the long and complex road from various small and larger algorithms, to a unified approach and the robustness of metalearning.

The book is addressed both to experienced machine learning scientists, interested in the research on DT induction, and to newcomers to the field. A novice should find the review parts (Chap. 2 and Sect. 6.1) especially useful, as they can help to understand many popular methods of DT induction and metalearning. Most experts shall be more attracted by the final chapters, which take up the topic of metalearning, however also many details of DT induction algorithms are likely to interest even the experienced researchers, because the information presented there is a result of a thorough analysis of the methods, performed by a researcher aiming at understanding various computational aspects and practical advantages of the algorithms. Moreover, the review parts may be seen as an interesting repository of knowledge about many algorithms, presented in a succinct but usually exhaustive descriptions, thus very valuable also for experts. The review and analysis of DT induction approaches, presented in the book, is probably the most extensive one and the most in-depth study, published so far.

Both novices to the field and experts may be interested in the unified view of DT induction algorithms (Chap. 3) and the general machine learning framework architecture (Chap. 4). Chapters 5 and 6 are intended to attract especially the researchers focused on metalearning, interested in meta-level analysis of learning algorithms and creating new meta-level algorithms.

Contents

Chapter 1
Introduction

Decision trees (DTs) belong to the most commonly used computational intelligence (CI) models. Even when other algorithms provide more accurate models (better approximating the target), DTs are often regarded as very attractive. One of the most important reasons of their attractiveness is comprehensibility. DTs can be easily expressed in the form of a set of logical rules describing the decision functions. When used for decision support, DTs provide simple explanations of particular decisions, usually in the form of single logical rule (that applies to the case at hand) being a conjunction of several readable premises.

In the pursuit of the best models, many algorithms have been proposed. Only in the realm of DT induction, thousands of various solutions have been published. Their possible combinations are so numerous that even in application to simple problems described by small datasets, there is no possibility to check all possible algorithms. Therefore, the question how to obtain as accurate models as possible has been and will still be, for a long time, the most important concern of CI.

In most contemporary problems, arbitrary model selection performed by human experts is not satisfactory, as humans have many significant limitations and, in particular, no expert can gain detailed knowledge about all the solutions worth a try. Automated systems are more systematic than humans—can explore the space of possible solutions in more thorough way, without missing any important method and without repeating calculations. Human experts need more time for analysis of performed calculations and are not ready for performing the analyses 24 h a day, 7 days a week, as automated tools are. Therefore, automated learning machines can use time more efficiently, so in such practical aspects, they have more potential than humans. Naturally, CI algorithms are still not as intelligent as humans, so they can not draw so deep and versatile conclusions. This is a space for further improvement of CI methods of *meta-learning* (which extract and take advantage of meta-knowledge about learning processes). Continuous progress in the field lets believe that it is just the matter of time that automated discoveries will be more successful than human experts. The research described in this book has been planned as a step in this direction. It does not answer all the questions related to the problem, but the results obtained so far

K. Grąbczewski, *Meta-Learning in Decision Tree Induction*,
Studies in Computational Intelligence 498, DOI: 10.1007/978-3-319-00960-5_1,

confirm that this approach to meta-learning can be successful, so it deserves further development.

1.1 Learning Machines and Meta-Learning

Learning is usually defined as acquiring new or alteration of existing knowledge or behavior. As a result of individual experience, perception of some stimuli, an organism gains new abilities, modifies its behavior, extends its knowledge and so on. Defining *machine learning* can be more precise. In place of miscellaneous stimuli, machines get *data* (of different kind and form) and run *learning processes* to obtain some *models* (artificial counterparts of human abilities, behaviors, knowledge, skills). Models can also have arbitrary forms. Usually, they are functions of classification, clustering, approximation and so on.

Learning Problem and Learning Machines

Formally, the term *learning problem* (or *problem of learning from data* or *learning task*) denotes a pair

$$\mathscr{P} = (D, \mathscr{M}) \tag{1.1}$$

of data $D \in \mathscr{D}$ ($\mathscr{D}$ is a space of *training data*) and *model space* $\mathscr{M}$. Solving a problem $(D, \mathscr{M})$ consists in finding a model for D within $\mathscr{M}$.

In the language of statistics, the set $\mathscr{M}$ can be called the *hypothesis space* and its each member a *hypothesis*. Hence, a learning problem $\mathscr{P}$ can be equivalently called a *hypothesis selection problem* or *model selection problem*.

There are no additional constraints on the shape of the training data space or model space. Usually, the training data space consists of sequences of object-value pairs, and the objective of learning is finding a mapping from the object space to the value space, best reflecting the relation between objects and values.

The procedures solving learning problems are called *learning algorithms* or *learning machines* or just *learners*. From formal point of view, (*learning*) *machines* are processes

$$L \ : \ \mathscr{K}_L \times \mathscr{D} \to \mathscr{M}, \tag{1.2}$$

where $\mathscr{K}_L$ is a space of configuration parameters of L. One could also ignore $\mathscr{K}_L$ and treat a parameterized machine as a family of non-parameterized machines, but the definition of Eq. (1.2) is more intuitive because closer to the natural use of the term.

In the definition above, the word "learning" is parenthesized, because it makes no sense to define precisely which processes really learn something. Distinction between learning and non-learning machines would be very subjective, and its potential profits are doubtful. On the other hand, it is very advantageous to treat all the data analysis processes in a uniform way. As a result, the processes of data transformation, test procedures estimating some measures of learning machines accuracies and even the

functions of data loading and other simple operations can also be seen as machines. Calling them "learning machines" would be exaggerated, but embracing all the processes (learning and non-learning) with the term "machines" is reasonable.

Another formal distinction is also very important and seems adequate at the beginning of this book. It concerns the terms "machine" and "model", which are often mixed up in informal statements. As a result, it is possible to find expressions like "learning process of a model" or "model has learned", but in the formalism proposed here, learning processes are the crucial parts of machines, not models, and models are output by the processes. Such distinction helps formulate clear and succinct descriptions and conclusions. Naturally, models may also be capable of learning, but in such cases it should be clearly specified which learning processes are referred to.

Learning machines may be complex. Practically useful learning machines are almost always composed of several ($k \in N$) other learning machines:

$$L = L_1 \circ L_2 \circ \ldots \circ L_k. \tag{1.3}$$

For example, a composition of a data transformation machine and a classifier machine is just another learning machine. Similarly, each committee is also a complex learning machine. What's more, a committee of complex machines is also a learning machine. Usually complex machines return complex models (models composed of other models), but it is not obligatory. For example, a machine that runs a number of other machines to collect their models and eventually selects one of the models as the result of the whole complex process is a complex machine, but may return simple (non-complex) model. A machine being a part of another machine is called its *submachine* or *child machine* (and the other machine is called a *parent machine*).

Meta-Learning

Whenever a learning process or a model resulting from learning is analyzed, the reasoning is performed at *meta-level*. Any form of learning from information about learning processes and models can be referred to as *meta-learning*, so the term is very general. Formally, meta-learning problems fit the definition of learning problem (1.1). They just represent a specific type of learning (with specific data and model spaces, concerning other learning processes). To distinguish the two levels of learning, the processes analyzed at meta-level are referred to as *object-level* or *base-level* learning machines.

The goal of meta-level analysis is to *learning how to learn* and to apply the gained knowledge in further learning so as to obtain more attractive results. It is a very important step in a complex process, aimed at finding optimal models. The general definitions of learning problem (1.1) and learning machines (1.2) do not touch the aspect of model optimality—any model can be a solution. In practice, we are usually interested in as good models as we can find, and the goodness of the model is formally expressed by a measure of model quality $q : \mathscr{M} \to Q$, where Q is a linearly ordered set of quality values (usually the set of real numbers). Extending

the definition (1.1) of learning problem by a *model quality measure* q, we can state the *optimal-learning problem* (or *model selection problem*) as a triple

$$\mathscr{P}_O = (D, \mathscr{M}, q). \tag{1.4}$$

Solving the problem $\mathscr{P}_O$ consists in finding possibly best model for D within $\mathscr{M}$, that is, a model $M \in \mathscr{M}$ maximizing $q(M)$.

Finding the best solution for $\mathscr{P}_O$ is usually NP-hard, because the set of possible models $\mathscr{M}$ is so large that it is not possible to examine all possible models (usually only a very tiny subset can be explored). Therefore, so many learning algorithms have been created and propose different ways to *suboptimal solutions*. Although the results of particular applications of learning machines are not guaranteed to be globally optimal, they are often quite accurate models obtained within acceptably short time. For many real-life problems, such solutions can be fully satisfactory.

The time of learning is another very important aspect of practical problems of learning, neglected by the formulations of learning problems (1.1) and (1.4). We never have infinite time to spend on searching for satisfactory solutions, even in research enterprises. Therefore, time limits should be included in the definition of learning tasks. According to this idea, extending the definition of an optimal-learning problem $\mathscr{P}_O$ with a time limit t for providing the solution constitutes a *time-limited optimal-learning problem*

$$\mathscr{P}_T = (D, \mathscr{M}, q, t). \tag{1.5}$$

Such definition of the problem of learning from data is much more realistic, and should be preferred also in research approaches. In a natural way, it reflects the reality of data analysis challenges, where deadlines are crucial, but it is also more adequate for real business applications, where solutions must be found in appropriate time and any other applications with explicit or implicit time constraints.

The task of finding possibly best model for given data D can be reformulated as the task of finding possibly most successful learning machine L for the data D. *algorithm selection problem* is the most practical kind of meta-learning problems. It used to be solved manually by machine-learning experts, although more and more often, meta-learning algorithms of different kinds are proposed.

Many different views on meta-learning can be found in scientific publications. Various learning processes, that acquire some knowledge about other algorithms or exploit this sort of knowledge, are referred to as meta-learning methods. More detailed review of meta-learning approaches can be found in Sect. 6.1.

1.2 Basic Definitions and Notations

Some definitions and notations are commonly used throughout the book. To make finding them easier, they have been collected together and are introduced already here. They are grouped into parts referring to the same concepts, for easier orientation.

Less common symbols are introduced just in place where they are first used, so that each definition should be found in close vicinity of the place where it is used or in this section.

Datasets, Data Tables

Learning machines are given the input of data organized into various forms. The most common form is a collection of data objects called a *dataset*. Data objects can be described in arbitrary ways—in general no particular form is assumed. Images, sounds, multidimensional structures or simple numbers are some examples of acceptable forms. The term "dataset" is not strictly used as "a set of data objects". In most cases a dataset is actually a sequence (an ordered sample), as the objects come in some order, with possible repetitions. Some learners are independent of the object order in the training sequence, but some others are not. Despite that the term "dataset" is commonly used. DT induction methods presented in this book are independent of the data order, so here, using the term "dataset" is quite compatible with its natural meaning. The algorithms analyzing data objects in sequence, select the objects at random, so there is no clash with the term, beside the fact of possible repetitions.

Most machine learning algorithms are designed to handle objects as multidimensional vectors of real numbers, and the most common approach to data analysis is to describe objects with such vectors first, and then apply learning machines to such form of data. When collections of objects are described as multidimensional vectors of real numbers ($O \subset R^n$), they are called *data tables*. Then, the dimensions of R^n are called *attributes* or *features*. There have been many discussions about distinction between the terms "attribute" and "feature", but for the purpose of this book, such distinction has no value, so the terms mean the same and are defined as above. Particular values assigned to objects in a given dimension are called *feature* (or *attribute*) values of the objects.

Features may be of different types. The ones with values being real numbers are called *continuous* or *numeric*. Some attributes may be denoted by just several distinct symbols. Such attributes are called *discrete*, *symbolic* or *nominal*. For the purposes of learning machines, usually the most adequate is the distinction to *ordered* and *unordered* features, because discrete features with several ordered values can (and should) be treated as numeric.

Some scientific articles use the language of statistics to describe learning from data. In this language, features in the context of data samples, are treated as random variables and called *covariates*, *explanatory variables*, *independent variables* or *predictors*. The *target variables* (output class, approximated values and so on) are also called *dependent variables* or *response variables*.

Each feature value for subsequent data object is then a value of the random variable. Data objects are often denoted as vectors (bold letters $\mathbf{x}$, $\mathbf{y}$, ...) and collections of data as matrices. Matrix/vector transposition is denoted with uppercase T letter as superscript (for example, $\mathbf{x}^T$, D^T, ...).

When dealing with datasets and data tables, the symbol n usually denotes the number of objects in the dataset. When more than one dataset is regarded in the same analysis, classical mathematical notation of the cardinality of a set is used ($|D|, |D_i|$). For data tables, m is used to denote features count and k—the number of classes (in classification tasks). Some other symbols, for example, Greek letters $\alpha, \beta, \theta, \ldots$, do not have standard meaning for the whole book and follow conventions accepted by original authors (were sensible). In some cases, the original notations are followed, to make comparisons with the original sources easier. A drawback of such decision is that sometimes different letters denote parameters of similar functionality, but it hopefully does not introduce much confusion.

In some learning approaches, unordered features in data tables are converted into so called *binary indicator variables*. A feature with l possible symbols $f_1, \ldots, f_l$ is converted into l binary features with values determined by the *indicator functions* associated with the original symbols:

$$I_{f_i} : \{f_1, \ldots, f_l\} \to \{0, 1\}, \qquad I_{f_i}(f) = \begin{cases} 1 & \text{if } f = f_i \\ 0 & \text{otherwise.} \end{cases} \tag{1.6}$$

The new features, generated in this way are usually called *indicator features* or *indicator variables* associated with the original feature.

Probabilities are denoted with uppercase letter P. Many shortcuts are followed in the field of CI, when probabilities are referred to. For example, the notation $P(C|D)$ is a shorthand notation for $P(x \in C|x \in D)$. In many situations some "natural" assumptions are not explicitly written to make expressions shorter.

Classification

One of the most common tasks of CI is *classification*. In order to define a classification problem one needs to:

- specify a (finite) set of *class labels* (or *categories*) $\mathscr{C}$,
- define a collection O of object descriptions (the *domain of classification task*) and specify a learning dataset as a collection of object-label pairs $D = \{(o_i c_i) : i = 1, \ldots, n\} \subseteq O \times \mathscr{C}$,
- restrict the model space to the set of functions $f : O \to \mathscr{C}$.

Less formally, in classification tasks, a function assigning class labels to objects is searched, and training dataset contains a sample of objects with assigned labels. For example, a task of image categorization may assume that one of two labels ("yes" or "no") should be assigned to each country in the world, depending on whether more than half of its population declare themselves as Christian or not. Naturally, there are many possible ways to define object (country) description: it could be just the name of the country, English Wikipedia text describing the country, and so on. Different definitions provide different classification tasks.

Classes (or categories) can be seen as subsets of the object space, so that intersections like $D \cap c$ of a dataset D and a class $c \in \mathscr{C}$ are formally correct. At the same time the classes can be identified with labels (the elements of $\mathscr{C}$ can be seen as the labels). This does not introduce any confusion, so the book accepts such points of view too.

As defined above, classifiers are functions $f : O \to \mathscr{C}$ assigning a class (label) to each object in O. Sometimes, we are interested in so called *probabilistic classifiers* or *weighted classification*, so that the assignment of a class is not crisp, but described by a probability of belonging to each possible class. Thus a probabilistic classifier can be defined as a function $f : O \to [0, 1]^{|\mathscr{C}|}$ or $f : O \times \mathscr{C} \to [0, 1]$ and provide the probabilities.

Naturally, it is easy to define a probabilistic classifier (with binary probabilities) corresponding to each crisp classifier. Inversely, a crisp classifier corresponding to a probabilistic classifier can be defined as assigning the class label maximizing class probability. The definition is not unique, but is possible, provided the axiom of choice or an order in $\mathscr{C}$.

Approximation

When objects are assigned some real numbers instead of discrete labels, the problem is called an *approximation* or *regression* task. All other aspects of problem definition are the same as in the case of classification. An example of such task may be assigning to each country of the world, its gross domestic product (GDP) per capita, measured in the currency of Polish złoty.

Testing

Quality of classification and approximation models is most often measured by means of accuracy of their predictions for data not used for learning the models (so called *unseen data*). It is not always possible to get another data sample to measure such accuracy, so often a given finite dataset must suffice to train a model and estimate its generalization abilities. One of the possibilities is to split the dataset into two parts and use one of them for training and the other for estimation of the model's real performance. When a classification model is built, its accuracy measured for the training part of the data is called *reclassification accuracy* or *training accuracy* and the accuracy measured on the other part is called *test accuracy* or *validation accuracy*. Errors may be referred to in analogous way to accuracies.

One of the most popular methods of testing classifiers is the technique of *cross-validation* (CV). It splits the dataset into v parts of as similar sizes as possible (v-fold CV), and performs v training and testing processes by selection of each part in sequence as the test part and all the remaining $v - 1$ parts as a single training dataset. After such v training and test stages, all test results are averaged and given as the final CV result. When splitting the dataset into v parts it is often important (especially for small samples) to keep the original proportions between classes. When the split

respects the proportions, the procedure is called an n-fold, *stratified* CV (as opposed to *non-stratified* CV).

DT induction methods often use some validation process to adjust a pruning parameter. They also use stratified CV for this purpose, which is then called an *internal CV* or *CV for training*. The distinction is very important, because often the DT induction algorithms are tested with external CV tests and use internal CV to adjust parameters.

Entropy

Given a data sample D and a discrete feature A with values in $\mathscr{A}$, let p_a denote the proportion of data objects with value $a \in \mathscr{A}$ of feature A. Entropy of the feature A is then defined as

$$H_A = -\sum_{a \in \mathscr{A}} p_a \log_2 p_a. \tag{1.7}$$

Often, entropy is analyzed in the context of two discrete features (usually a feature describing the objects and the column of class labels, being also a discrete random variable; sometimes a partition of the data sample composes the "feature" tested against the class labels). For variables A with values within a set $\mathscr{A}$ and class column C with values within a set $\mathscr{C}$, let p_{ac} denote the proportion of objects of class $c \in \mathscr{C}$ with value $a \in \mathscr{A}$ of feature A, and $p_{a\cdot}$ and $p_{\cdot c}$—the proportions of objects with value $a \in \mathscr{A}$ of feature A and objects of class $c \in \mathscr{C}$ respectively. Then the following formulae define entropies of partitions determined by A and C:

$$H_A = -\sum_{a \in \mathscr{A}} p_{a\cdot} \log_2 p_{a\cdot}, \qquad H_C = -\sum_{c \in \mathscr{C}} p_{\cdot c} \log_2 p_{\cdot c}. \tag{1.8}$$

Joint entropy of A and C, and conditional entropy of C given A are defined as:

$$H_{A,C} = -\sum_{a \in \mathscr{A}} \sum_{c \in \mathscr{C}} p_{ac} \log_2 p_{ac}, \qquad H_{C|A} = H_{A,C} - H_A. \tag{1.9}$$

Decision Trees

Because the book is devoted mainly to DT models, some terms related to DTs are often used. A *decision tree node* is usually denoted as a triple $N = (D, c, Subnodes)$, where D is a data sample accompanying the node (or just *node dataset* or *node data*), $c \in \mathscr{C}$ is a class label assigned to N and $Subnodes$ is a collection of nodes called *subnodes* or *child nodes*.

With such definition of a decision tree node, a *decision tree* does not need a separate definition, as each node determines a tree (a tree is defined by its root). A tree determined by a tree node N is called a *tree rooted at* N and denoted also as T_N (just to emphasize the context, that is, to signal that the focus is on the structure not on a single node; formally N and T_N are the same).

In discussions concerning DT construction, often, a node data sample is split and subsamples created. The analyses use different counts like: n_N—the number of objects in the node N dataset (the same as $|D|$, if D is the node N data sample), $n_{N,c}$—the number of objects in node N belonging to class c (the same as $|D \cap c|$).

The splits are considered in the context of DT nodes or their datasets, so that *splitting a node* and *splitting a node data* are two names for the same operation. Splits may be performed by *n-way split functions* $s^{(n)} : O \to O^n$, such that each dataset $D \in O$ is mapped into a k-tuple of disjoint and complementary subsets of D.

In the context of a split function s, some counts n and datasets D, marked with adequate indices are used. For example:

- n_s denotes the number of nodes resulting from the split s (the n such that s is a n-way split function),
- D_{s_i} denotes the i'th part of the split of D,
- D_c is the set of objects in D of class c,
- $D_{s_i,c}$ is the set of objects of class c in the i'th part of the split of D, and so on.

Given a tree T, the notation $\widetilde{T}$ means the set of all *leaves* (that is, *terminal nodes*) of T (thus $|\widetilde{T}|$ is the number of leaves in T). Each *non-terminal node* (*non-leaf*) can also be called a *split node*.

When a node dataset contains objects from one class only, the node is named *pure* or *clean*. Generalization abilities of trees are obtained with techniques named *pruning*, which regularize the model by converting some non-leaves of the tree to leaves, to eliminate too detailed splits. Pruning can be performed during the process of DT construction (then called *pre-pruning*) or after building full, maximally correct tree (hence called *post-pruning*).

A tree obtained by pruning a tree T is called its subtree, which is denoted with $\prec$ relation, for example, $T_\alpha \prec T$. Sometimes, a tree rooted at one of the nodes of T can also be called its subtree, but usually it is referred to as "a subtree rooted at given node", so that there is no confusion between the two kinds of subtree relation.

Sometimes in the context of DTs, but also in other contexts, a symbol $\perp$ is used. It is a general "null" or "void" symbol, which can mean different things like empty split (no further split of a tree node), empty subnode collection, none attribute selected for split and so on.

Chapter 2
Techniques of Decision Tree Induction

Finding optimal DT for given data is not easy (with exceptions of some trivial cases). The hierarchical structure of DT models could suggest that the optimization process is also nicely reduced with subsequent splits, but it is not so. It is important to realize that optimization of a criterion for tree node is not the same as optimization of the whole tree or even subtree. The difference has been proved formally by de Sá (2001) for decision trees of some specific, simple form—where each class is represented by one leaf. On this assumption, each tree node can be assigned a subset of class labels in such a way that the root node is assigned the full set of class labels, and subsequent splits divide the set of labels assigned to the node being split into disjoint parts. Further assumption of independence of the features used along each path of the tree leads to the conclusion that probability of correct classification to class c_k is given by

$$P_C(c_k) = \prod_{i_k=1}^{n_k-1} P_C(c_k|N_{i_k}), \tag{2.1}$$

where $(N_1, \ldots, N_{n_k})$ is the tree branch ended with the leaf assigned c_k label and $P_C(c_k|N_i)$ is the probability of correct decision at node N_i. Then, the probability of correct classification by the whole tree is

$$P_C(T) = \sum_{c_k \in \mathscr{C}} P(c_k) \prod_{i_k=1}^{n_k-1} P_C(c_k|N_{i_k}). \tag{2.2}$$

On the other hand, the probability of correct classification at node N is a linear combination of probabilities of correct classification of objects belonging to subsequent classes:

$$P_C(N) = \frac{\sum_{c_k \in C(N)} P(k) P_C(c_k|N)}{\sum_{c_k \in C(N)} P(k)}, \tag{2.3}$$

K. Grąbczewski, *Meta-Learning in Decision Tree Induction*,
Studies in Computational Intelligence 498, DOI: 10.1007/978-3-319-00960-5_2,

where $C(N)$ is the set of class labels assigned to node N. As a result, optimization of this probability can not be equivalent to optimization of $P_C(T)$, because the latter depends nonlinearly on subsequent $P_C(c_k|N)$.

With more general definition of DTs (for example, the one presented in Sect. 1.2), the dependencies of tree accuracy on node accuracies is yet less straightforward, so optimization at each tree node can also be quite different from the optimization of the whole tree. It means that to find a global optimum, one would have to examine all possible trees, but even in simple (but not trivial) cases, it is not possible, because the number of possible trees is usually infinite (or at least very large, as it grows exponentially with target tree depth). Therefore, in practical DT induction, the trees are constructed with heuristic search methods. Since classification accuracy at tree nodes does not directly affect the accuracy of the whole tree, instead of the accuracy criterion, other measures of split quality are used as heuristics and turn out to be more valuable.

Practical approaches put some constraints on the domain (assume some form of node split functions) and perform a search based optimization in such limited space of models. Scientists working in the area have come up with large number of different ideas of how to restrict the search and how to optimize model selection. It is not possible to list them all even in a book, but it is possible to present the most interesting contributions to the field, proposed in recent decades. It would not make much sense to present many similar solutions, so this chapter reviews the most popular kinds of decision tree algorithms and some assessed as very interesting or possibly valuable. Of course, the selection is subjective, but it definitely can give a good grasp of the field, even to a novice to machine learning, and good orientation in the area to people who do not have everyday experience in this branch of computational intelligence.

2.1 Recursive Top-Down Splits

The most common approaches to decision tree induction are based on recursive top-down splits of the training dataset. Given a method to split DT nodes, or more precisely, to split the training data corresponding to the node, it is executed at each tree node to find the best splits. Denoting such split method by *BestSplit()*, the recursive DT construction algorithm can be formally written as Algorithm 2.1. Most often, splits are found with an exhaustive search through the collection of all possible splits of the node at hand. In such approaches, the *BestSplit* function analyzes each candidate split with a *split quality measure* (SQM) provided as one of the most significant configuration parameters of the method. Many split quality measures are based on common idea of *impurity reduction* (or *purity gain*). Provided a measure of data sample purity (homogeneity), split quality may be estimated as the increase of the homogeneity between the tree nodes after the split and before the split. *Impurity measures* (or *criteria* or *indices*) should satisfy some conditions to be compatible with the idea: a node containing data objects representing one class only should get

minimum possible value of the index (usually zero). Maximum values should be given to maximally mixed samples (all classes represented by the same number of objects).

Algorithm 2.1 (Common DT induction approach)

Prototype: ***CommonDTRec(D,BestSplit)***
Input: *Training dataset D, node splitting procedure BestSplit.*
Output: *Decision tree (= the root node of the tree).*
The algorithm:

1. $s \leftarrow BestSplit(D)$
2. ***if*** $s \neq \perp$ ***then*** */* a split has been returned */*
   a. $\{D_1, \ldots, D_n\} \leftarrow s(D)$ *(split node N)*
   b. ***for*** $i = 1, \ldots, n$ ***do***
      $N_i \leftarrow CommonDTRec(D_i, BestSplit)$
   c. $Children \leftarrow (N_1, \ldots, N_n)$
   else
   $Children \leftarrow \perp$
3. ***return*** $(D, s, Children)$

Denoting the purity criterion as I, we get the following formula of *purity gain* (*impurity reduction*):

$$\Delta I(s, D) = I(D) - \sum_{i=1}^{k} \frac{|D_i|}{|D|} I(D_i) \tag{2.4}$$

where s is the split, D is the dataset to be split and $s(D) = (D_1, \ldots, D_k)$. Given the index I, the best splits of dataset D are those maximizing $\Delta I(s, D)$.

Exhaustive search for best split guarantees local maximum, however can be costly. To avoid the cost, the exhaustive search is sometimes replaced by statistical tests and/or discrimination methods. Statistical tests can determine the features which seem to maximize the probability of providing attractive data splits. Discrimination methods can calculate split points without the necessity of checking all possible splits. Separating feature selection from split determination may significantly reduce computational complexity of a single split procedure, but it is important to realize that it does not necessarily imply faster tree construction (the resulting trees may be larger, so more splits may compose the final trees).

On the other hand, yet more complex search methods can be run, to examine split advantages more thoroughly. For example, one can estimate split quality on the basis of analysis of potential further splits. More details on DT search procedures can be found in Sect. 3.1.1.

Split criteria and search methods are not all the differences between DT induction algorithms. Some methods use only binary splits, while others accept splits into more than two subnodes. In some trees, only univariate splits are allowed and others

perform multivariate analyses. Many more detailed differences could be enumerated. The following sections present many DT induction algorithms in many interesting aspects. Some alternative collections of decision tree induction techniques can be found for example in Murthy (1998), Rokach and Maimon (2008, 2010), Kotsiantis (2011). More organized analysis and a unified model of DT induction methods is the subject of Chap. 3.

2.2 Univariate Decision Trees

Numerous DT construction algorithms result in models where splits are performed on the basis of simple conditions concerning single features. Decision borders of such models are perpendicular to axes of the space of data object descriptions. From one point of view, it is a serious limitation of the approaches, but from another, decisions can be described with readable formulae making the models 1comprehensible and thanks to that easier acceptable by experts in many fields, for example in medicine, where a responsible clinician can accept support from an artificial decision system only if it provides comprehensible descriptions of its suggestions.

2.2.1 ID3

Iterative Dichotomiser 3 (ID3) (Quinlan 1986; Mitchell 1997) is one of the earliest ideas of DT induction. Its split criterion was founded on information theory. The most serious drawback of ID3 is the requirement that the data description may include only discrete features. When the original data table contains numeric features, they must be first discretized. Success of data mining processes consisting of data discretization and final model creation usually depends on the former part more than on the latter. Therefore, estimation of the efficiency and accuracy of ID3 in application to continuous data does not make much sense, because with one discretization method the results may be very good and with another one—completely wrong.

The method is a typical example of top-down recursive induction presented in Algorithm 2.1. Split qualities are estimated with the purity gain criterion (2.4) using entropy as node impurity measure:

$$I_E(D) = H_C(D) = -\sum_{c \in \mathscr{C}} P(c|D) \log_2 P(c|D). \tag{2.5}$$

Such combination of formulae (entropy reduction) is called *information gain* (IG) criterion:

$$IG(s, D) = \Delta I_E(s, D). \tag{2.6}$$

In practice, the probabilities of classes within the node N are usually estimated by ratios $\frac{n_c}{n}$ of the numbers of objects in node N data representing class c and the numbers of all objects falling into N. When implementing the information gain criterion for the sake of DT induction, one usually simplifies the formulae. Converting expressions according to the equality

$$-\sum_{c\in\mathscr{C}} \frac{n_c}{n} \log \frac{n_c}{n} = \log n - \frac{1}{n} \sum_{c\in\mathscr{C}} n_c \log n_c,$$

the information gain resulting from the split of N into parts N^p can be written as

$$IG = \log n + \frac{1}{n}\left(-\sum_c n_c \log n_c - \sum_p n^p \log n^p + \sum_p \sum_c n_c^p \log n_c^p \right). \quad (2.7)$$

Because in DT induction we are interested in comparison between splits, not in precise calculation of IG, the constant parts of the formula given above can be ignored, and only the second and third components in the big parentheses need to be calculated.

In each step of ID3 algorithm, a node is split into as many subnodes as the number of possible values of the feature used for the split. The exhaustive search for best split, in this case, just estimates the quality of each feature, because only one split is possible per feature. The feature offering maximal entropy reduction is selected, the node split, and the feature used for the split is removed from the data passed down to the subnodes, because it is no longer useful in the tree branch (all objects have the same value of this feature).

An important disadvantage of such split technique is that the features with many possible values are preferred over those with small counts of symbols, even when the former are not too valuable—in an extreme case, if each object has a unique value of a feature, the feature will reduce the entropy to zero, so will be treated as very precious, while in fact, its value is overestimated due to the split into many nodes with single data objects.

Apart from the main ID3 algorithm, Quinlan (1986) has presented some other interesting contributions. One of them is the method for fastening DT induction, when training dataset is very large. Quinlan proposed an iterative framework discussed also by O'Keefe (1983). It is based on using a *window*—a subset of the training dataset instead of all training objects. In such approach, ID3 may need a number of tree induction iterations to provide final classification tree. The process starts with building a tree to classify objects in the window with maximum accuracy. Then, the tree classifies the objects outside the window. If all the objects are classified correctly, the tree is the final result. Otherwise, a selection of incorrectly classified objects is added to the window and next tree is generated. If the window is allowed to grow to the size capable of containing all training data objects, the process is guaranteed to end up with a maximally accurate tree (with respect to the training data). If not, some problems with convergence may also occur.

Quinlan (1986) has also proposed some solutions to avoid overfitting noisy data. Stop criterion defined as zero information gain is too little to guarantee generalization of tree models. Nonzero information gain can very often be observed even in completely random distributions. A threshold for information gain is not recommended either, because finding proper threshold value can be difficult. Therefore Quinlan (1986) proposed a method based on Pearson's χ^2-test for stochastic independence, described in appendix Sect. A.2.2 For example, when an attribute has v possible values $a_1, \ldots, a_v$ and classification problem defines k classes $c_1, \ldots, c_k$, then the independence between the attribute and the classes can be estimated with the Pearson's χ^2 statistic calculated for the contingency table reflecting the joint distribution of the two variables. The statistic may be confronted with the χ^2 distribution with $(v-1)(k-1)$ degrees of freedom and a given confidence level, to verify whether the attribute is irrelevant. When the hypothesis about the independence can not be rejected, the attribute should not define the next split in the tree—a leaf should be generated regardless of its impurity.

Quinlan (1986) mentioned several possible ways of dealing with missing values (mostly proposed earlier by other authors) like imputing the most probable value, using fractional objects, predicting the value with a decision tree trained for this purpose, and others.

2.2.2 CART

Classification and Regression Trees (CART) is one of the most popular and very successful methods of DT induction (Breiman et al. 1984; Michie et al. 1994; Cherkassky and Mulier 1998). The algorithm is nonparametric and creates binary trees from data described by both continuous and discrete features. For continuous features, all possible binary splits into intervals $(-\infty, a]$ and (a, ∞) are considered. For discrete attributes, the analysis concerns all possible splits of the set of symbols into two disjoint and complementary subsets.

Exhaustive search for the best splits estimates split qualities with the impurity reduction criterion (2.4) with impurity defined as so called *Gini (diversity) index*:

$$I_G(D) = 1 - \sum_{c \in \mathscr{C}} P(c|D)^2. \tag{2.8}$$

In place of Gini index, it is also possible to use entropy (2.5) or any other measure of impurity.

Breiman et al. (1984) also proposed a technique named *twoing*, which was created to handle multi-class problems by two-class criteria. Twoing means grouping the classes into two superclasses and performing two-class analysis for the groups, instead of the original classes. Naturally, when the number of classes is large, the number of possible groupings can cause combinatorial explosion, if one tries to check all possibilities. Instead of analyzing all splits for all possible class groupings,

Breiman et al. (1984) proposed an efficient procedure to determine optimal superclasses for each possible split. The procedure is valid for two-class impurity criterion (compatible with Gini index) defined as

$$I(D) = P(c_1|D)P(c_2|D). \tag{2.9}$$

Breiman et al. (1984) proved that for given binary split s, which for a dataset D generates subsets D_L and D_R, maximum decrease of impurity is obtained when one superclass contains all the classes c for which $P(c|D_L) \geq P(c|D_R)$, and the second superclass—the remaining classes. This result lets keep attractive computational complexity of the twoing procedure.

In CART, each tree node is assigned the class label dominating within the node. There is also a possibility to respect misclassification costs in the decisions.

Missing data values are handled with a technique of *surrogate splits*. When a data object is not described with a value necessary for the test of a tree node, it is passed to another test, exploiting another feature to generate a split, as similar to the one of the original test as possible. Several surrogate splits can be found and used for data with missing values in appropriate order.

As indicated in the name of CART, the method is designed to be applicable to both classification and regression problems. Approximation trees are very similar to the classification ones, but instead of class labels, the nodes are assigned some real values. Such tree definition allows the trees to represent piecewise constant functions, so to approximate less trivial functions with low mean squared error, the trees need to be large.

Many improvements and extensions to CART solutions have been proposed later. For example Strobl et al. (2005) proposed using p-values to obtain unbiased feature selection (see Sect. 2.7) and Piccarreta (2008) extended Gini criterion to ordinal response variables.

Cost-Complexity Optimization

Breiman et al. (1984) have also proposed an interesting method for pruning CART after learning, as stop criteria are neither as flexible nor accurate as post-pruning methods. Although CART offers a stop criterion in the form of minimum node size specification (minimum number of training data falling into the node), the main tool for adjusting tree size to given problem is the *cost-complexity optimization*. As suggested by the name, instead of just training error minimization, which usually leads to overfitting the training data, the optimization concerns a risk measure involving both misclassification cost and size (complexity) of the model, defined as the number of leaves of the tree (see Eq. (2.79) and Sect. 2.4.3.2 for detailed explanation). A parameter α controls the trade-off between training data reclassification accuracy and size of the tree. To determine the optimal value of α, validation procedures are proposed to estimate the performance of the candidate values and select the best one. The validation can be performed on the basis of a separate validation dataset or by cross-validation. In the case of CV, in each pass, a DT is built and analyzed

to determine all the threshold αs, that is, all the values of the parameter for which a node obtains the same combined cost as the subtree and reduced to a leaf, so that for α less than the threshold it is advantageous to leave the subtree as it is, and for values greater than the threshold—to replace the subtree by a leaf. Finally, a tree is built for the whole dataset, and its threshold αs are determined. On the basis of the CV, for each threshold, its average risk value is calculated and the one providing minimum risk is chosen as the optimum. The final tree is pruned in the way that minimizes the risk for the selected threshold α. For more formal presentation of the algorithm see Sect. 2.4.3.2.

Pruning Control with Standard Error Margin

CART implementation of the validation procedure introduced a parameter to enforce simpler trees than resulting from normal cost-complexity analysis. As a result the cost-complexity optimization usually comes in two versions: 1SE and 0SE. The acronym SE stands for "standard error". 0SE denotes just the fundamental version of the method (without standard error based correction) while 1SE signifies the modified version, where standard error is estimated and model selection prefers simpler trees with the reservation that the cost can not increase by more than the value of the standard error. More information on the SE parameter and methods of its calculation can be found in Sect. 3.2.4.1.

2.2.3 C4.5

Another very popular DT induction system (next to CART) is C4.5 by Quinlan (1993). It has found numerous applications. The system arose from ID3 and shares many solutions with its ancestor. Main differences introduced in C4.5 are:

- modified node impurity measure,
- support for direct handling continuous attributes (no necessity to discretize them),
- introduction of a pruning method,
- precise methods for handling data with missing values.

Impurity Measure

Modified measure of node impurity aimed at eliminating bias in split feature selection, that is, favoring features with many possible values by the information gain criterion used in ID3. To replace the IG, Quinlan (1993) proposed *information gain ratio* (IGR) defined as

$$\Delta I(s, D) = \frac{\Delta I_E(s, D)}{SI(s, D)}, \tag{2.10}$$

where split information $SI(s, D)$ is the entropy of the split $s(D) = (D_1, \ldots, D_n)$:

$$SI(s, D) = -\sum_i p_i \log_2 p_i, \quad \left(p_i = \frac{|D_i|}{|D|}\right). \tag{2.11}$$

Handling Continuous Attributes

The support for continuous attributes in the data is organized similarly to CART. All sensible binary splits, deduced from the training data, are examined and the one with the best score (here the largest information gain ratio) chosen. Unlike symbolic features, continuous attributes may occur at different levels of the same tree many times (symbolic ones, when used in the tree, are no longer useful and because of that are not considered in further splits in the branch).

DT Pruning

In ID3 a statistical test of independence served as a stop criterion to prevent oversized trees, overfitting the training data. C4.5 offers another technique of generalization control. It builds (almost) maximally accurate trees and then prunes them to get rid of too detailed nodes that have not learned any general classification rule but just adjusted to specific data objects present in the training sample. The word "almost" added in parenthesis reflects what can be found in the source code of C4.5 about the process of tree construction: C4.5 has a parameter MINOBJS, which controls a pre-pruning method. If a node to be split contains too small number of objects or the split would generate too small nodes, further splits are rejected. After the tree is constructed, each node is tested with a statistical tool to estimate the probability that the node split causes error reduction (assuming binomial distribution of erroneous decisions). Each node, for which the probability is below a given threshold, is pruned or the subtree rooted in the node is replaced by its best subtree (the technique was named *grafting*). More details about C4.5 pre-pruning and post-pruning (*Error-Based Pruning*) methods can be found in Sect. 2.4.2.2.

Handling Missing Values

Objects with missing values can also be used in both the process of C4.5 DT construction and in further classification with a ready tree. At the stage of tree construction, in calculation of IGR, the objects with missing values of the feature being analyzed are ignored—the index is computed for a reduced set of objects and the result is scaled by the factor of probability of value accessibility (estimated by the fraction of the number of objects with non-missing value of the feature and the number of all training objects at the node). When the training data sample is split for subnodes creation, weights are introduced to reflect that it is not certain which path should be followed by the training data objects with missing decision feature values. The weights may be interpreted as the probabilities of meeting or not the condition assigned to the node. They are calculated as the proportions reflecting the distribution of other data

(with non-missing value) among the subnodes. When the weights are introduced, they must be considered also in further calculations of the IGR—wherever cardinalities are used (see Eqs. (2.4), (2.10) and (2.11)), sums of the weights are calculated instead of just the numbers of elements (naturally, the default initial weight value for each object is 1). Similarly, at classification stage, if a decision feature value is missing for a data object, all subnodes are tested and decisions obtained from each path are combined by adequate weighting to obtain final probabilities of the classes for the object.

Other Interesting Solutions

Apart from the decision tree algorithm, C4.5 system offers a methodology for building classifiers based on sets of logical rules. The algorithm called "C4.5 rules" starts with building a decision tree and converting it into compatible classification rules, but then the rules are subject to a simplification (pruning) process, which can significantly change the decision function of the model. Rules pruning is performed by removing premises if without them reclassification does not get deteriorated. Each rule is simplified separately, so the resulting rule set based classifier may be significantly different than the original tree (in practice, usually less accurate).

A modified version of C4.5, named *C5.0* or *See5*, is a commercial product and its popularity is very low in comparison to the ubiquitous C4.5. Because of that, also its results are not so commonly known as those of C4.5.

2.2.4 Cal5

Müller and Wysotzki (1994, 1997) have created a decision tree induction algorithm *Cal5* for classification of objects described in spaces of continuous features. Fundamental element of the method is its procedure dividing continuous features into intervals with application of statistical tools to estimate tree node purity.

As most other DT induction algorithms, Cal5 recursively splits nodes into subnodes, starting with the root node containing the whole training data sample. Analysis of each node consists of three main steps:

- selection of the best attribute for the split,
- discretization of the attribute (dividing it into intervals),
- merging adjacent intervals resulting from the discretization.

Algorithm 2.2 sketches the topmost procedure of the method. The parameters mentioned in the input specification section control some details of the three main steps of each node analysis:

- the selection of a measure to estimate attribute eligibility for the split (Q or IG),
- threshold s, defining minimum probability of correct classification by given node, that makes it a leaf,
- confidence level α for statistical tests performed within the discretization process.

Algorithm 2.2 (DT induction by Cal5)

Prototype: Cal5(D)
Input: *Training dataset D, some configuration parameters.*
Output: *Decision tree.*
The algorithm:

1. $A \leftarrow BestAttribute(D)$
2. $Intervals \leftarrow Discretize(A, D)$
3. $Intervals \leftarrow MergeIfReasonable(Intervals)$
4. *If* $|Intervals| > 1$
   a. *For* $i = 1, \ldots, |Intervals|$
      $N_i \leftarrow Cal5(Intervals_i)$
   b. $Children \leftarrow (N_1, \ldots, N_{|Intervals|})$
   else
   $Children = \perp$
5. ***return*** $(D, s_{Intervals}, Children)$

Precise meaning and application of the parameters is explained below, in the descriptions of each of the three steps. To make Algorithm 2.2 more readable, they are not disclosed there.

Nodes of Cal5 trees may have different numbers of subnodes. The counts are automatically determined in the process of attribute discretization and interval merging.

Best Attribute Selection

Decisions, which attribute to select for node split generation, are made in Cal5 on the basis of a statistical approach or with entropy measure.

In the statistical method, each feature eligibility is estimated with the following quotient:

$$Q = \frac{A^2}{A^2 + D^2}, \tag{2.12}$$

Q criterion where D^2 is the mean value of squared variance of the classes with respect to their centroid vector, and A^2 is the mean value of squared distances between the centroids of the classes. An attribute with minimum Q value is selected as the best one.

The method based on entropy measure requires each attribute discretized, so in that case, the order between the steps of feature selection and discretization gets inversed. The discretization procedure, described below, is run for each feature and an index of weighted sum of entropies of the subsets is calculated. In fact, the index is just the information gain defined by Eqs. (2.4) and (2.5) and used in ID3. Naturally, the best attribute is the one with the largest information gain.

Discretization

The process of continuous attribute discretization starts with sorting the training data sample (assigned to the node) in the order of nondecreasing values of the attribute. Next, the intervals starting at $-\infty$ and ending between subsequent two adjacent values of the feature are analyzed. The analysis of an interval x is based on testing two hypotheses:

H1 – there exists a class $c \in \mathscr{C}$, such that $p(c|x) \geq s$,
H2 – for all classes $c \in \mathscr{C}$, $p(c|x) < s$.

The tests are done by calculation of the confidence interval $[d_1, d_2]$ for a specified level α, with the following formula derived from the Chebyshev's inequality with the assumption of Bernoulli distribution of each class:

$$d_{1/2} = \frac{2\alpha n_c + 1}{2\alpha n + 2} \mp \frac{1}{2\alpha n + 2}\sqrt{4\alpha n_c \left(1 - \frac{n_c}{n}\right) + 1}, \tag{2.13}$$

and check whether the whole interval lies on the adequate side of the threshold s. There are three possibilities:

1. Hypothesis H1 is true. Then, the interval is regarded as closed and it corresponds to a leaf assigned with the label of the class c which made H1 condition true. The analysis starts again for intervals beginning at the end of the interval just closed.
2. Hypothesis H2 is true. Then, the interval is also regarded as closed, but it corresponds to a node requiring further splits, because no class sufficiently dominates in the node.
3. Neither H1 nor H2 is true. Then, the interval is exceeded to include the next object from the ordered sample. If no more data objects are available, a leaf labeled with the dominating class is created.

Interval Merging

After discretization, adjacent intervals are merged if they both are leaves with the same class label. Adjacent intervals are also merged if no class dominates in them and they contain the same set of classes represented at least as frequently as in the case of the uniform distribution. The set of classes is determined by elimination of the classes for which $d_2 < \frac{1}{n_I}$, where n_I is the number of class labels occurring in the interval I.

Symbolic Features

Although Cal5 was designed to deal with data descriptions containing continuous features only, it is not very difficult to extend it to accept also symbolic attributes. When attribute selection is made with IG criterion, it can be applied naturally to

symbolic features (discretization is just not necessary). Instead of division to intervals, groups of data objects sharing a value of discrete features can just be examined with hypothesis H1 and H2, to decide whether a node should become a leaf or needs further splits. It is also possible to apply the procedure of merging (designed for intervals) to the groups of data objects sharing a symbolic feature value. When rephrasing the algorithm, one needs to be careful about computational complexity of the resulting method, because in symbolic attributes there is no adjacency as in the case of intervals, and analyzing each pair of values may be costly. Sensible restrictions for the pairs of values considered for merging can be easily introduced in such a way, that computational complexity remains attractive. For example, relations between fractions of groups belonging to particular classes may be used as a substitute for intervals adjacency.

2.2.5 FACT, QUEST and CRUISE

A family of interesting DT algorithms has been created in the group of prof. Wei-Yin Loh. The family includes such algorithms as FACT (*Fast Algorithm for Classification Trees*, Loh and Vanichsetakul 1988), QUEST (*Quick, Unbiased, Efficient, Statistical Tree*, Loh and Shih 1997) and CRUISE (*Classification Rule with Unbiased Interaction Selection and Estimation*, Kim and Loh 2001, 2003). The methods are called "statistical trees" because they strongly base on statistical tools in tree construction and refinement. They have both univariate and multivariate forms, but the univariate algorithms are more often used so their descriptions are included here.

Algorithm 2.3 (DT induction by separate feature selection and split)

Prototype: ***FeatSelThenSplit(D)***
Input: *Training dataset D, some configuration parameters.*
Output: *Node split.*
The algorithm:

1. $A \leftarrow BestAttribute(D)$
2. *If* $A = \perp$ ***return*** $\perp$
3. $s \leftarrow BestAttributeSplit(A, D)$
4. *If* $s = \perp$ ***return*** $\perp$
5. ***return*** s

The algorithms can be seen as classical top-down DT induction algorithms, that is, Algorithm 2.1 with a specific approach to best split selection (*BestSplit* procedure of the algorithm) divided into two parts: first the feature for split is selected and then particular split is searched for the selected feature, as in Algorithm 2.3. Thanks to independent feature selection and split, there is no need to perform exhaustive

search for possible splits of a given node—only the splits of selected feature need to be analyzed. This may fasten the algorithm, if only the feature selection part is accurate. Otherwise the tree may become much larger and it may shatter the gains of the restrictions in split search space.

The authors paid much attention and undertook much effort to make their methods of feature selection unbiased. More detailed discussion of this subject is presented in Sect. 2.7, so here it is not further explored.

2.2.5.1 FACT

FACT (*Fast Algorithm for Classification Trees*, Loh and Vanichsetakul 1988) is designed to split datasets described by numeric features, but the authors provided a solution to convert symbolic attributes to continuous ones before the fundamental algorithm starts.

Conversion of Symbolic Features to Continuous Ones

To convert a discrete attribute X_D into a continuous X_C, FACT first creates a space of $n-1$ binary features, where n is the number of possible values of the symbolic feature. Each dimension of the space is a binary indicator variable corresponding to one feature value (informs which objects originally had this value and which had not). Then, the largest *discriminant coordinate* (crimCoord, see appendix Sect. B.1, Gnanadesikan 1977) is found in this $n-1$-dimensional space. It becomes a new continuous feature (X_C) replacing the original symbolic one (X_D). After the split is determined for the converted feature, it is easy to convert the conditions like $X_C > z$ to more informative form of $X_D \in A$.

Feature Selection

For univariate splits, FACT selects the feature by means of analysis of each attribute (discrete ones are analyzed after the conversion to continuous variables, described above) with the F statistic known from ANOVA (*analysis of variance*) methods, which is the ratio of between to within class variance (for more details see appendix section A.2.1 about *F-test* or (Tadeusiewicz et al. 1993; Brandt 1998)). The feature with the largest F ratio is selected, if only its F statistic exceeds a threshold F_0 (user-specified parameter of the method, with default value of 4). If $F < F_0$ each feature X is transformed to $Z = |X - \overline{X}|$ and F ratios for such transformed features are calculated. If the largest F ratio $F_Z \geq F_0$, than its feature Z is used for the split. Otherwise, the original feature X maximizing F ratio is used to generate a binary split with respect to $\overline{X}$.

FACT also offers an option to search for polar coordinate splits, more effective if there is an angular or radial symmetry in the data. It must be pointed out, however, that the feature constructed in this way makes the splits multivariate. If such option is selected anyway, the original data vectors are first converted into vectors of larger

principal components (as presented below, in the description of the LDA-based split procedure). The resulting new features (linear combinations of the original ones) are subject to a similar analysis of F ratios, as in the case of original features, described above. As a result, either a linear combination feature is used for the split or a possibility of spherical symmetry detected. Since the symmetry may not be present in every variable, the original features are selected on the basis of Levene's homogeneity test of variances (see Levene 1960, appendix Sect. A.2.3), before they are transformed into polar coordinates.

Split Selection

FACT performs splits by means of *linear discriminant analysis* (LDA). The procedure is defined for multidimensional data descriptions. For univariate trees, the calculations get very simple. To avoid nonsingular covariance matrices a *principal component analysis* (PCA) is done at each node before the actual analysis. The components with eigenvalues smaller than β times the largest eigenvalue are rejected (β is a user-specified parameter). The remaining components take part in determination of the linear discriminant functions:

$$d_c(\mathbf{y}) = \mu_c^T \Sigma^{-1} \mathbf{y} - \frac{1}{2} \mu_c^T \Sigma^{-1} \mu_c + \ln p(c|N), \tag{2.14}$$

where $\mathbf{y}$ denotes a vector in the space of selected principal components μ_c is the sample mean vector of class c, Σ is the pooled estimate of the covariance matrix at node N and N is the node being split.

Nodes are split into as many subnodes as the number of classes represented within the node being split. Objects are delegated to the subnodes corresponding to the class minimizing the following formula respecting also a misclassification cost matrix $Cost$:

$$c_{win} = \arg\min_{c \in \mathscr{C}} \sum_{k \in \mathscr{C}} Cost(c|k) e^{d_k(\mathbf{y})}. \tag{2.15}$$

The technique is called *normal theory option*.

Stop Criterion to Improve Generalization

To prevent overfitting the training sample, FACT is equipped with a stop criterion, which is tested at each node. Further splits are not accepted if one of the following two conditions is met:

1. The node contains no more than one class represented by at least MINDAT objects (MINDAT is a user-specified parameter).
2. The split does not decrease predicted error rate. For a node N split into subnodes $N_1, \ldots, N_k$, denoting by c_X the class assigned to node X, the split is not accepted if

$$\sum_{c\in\mathscr{C}} Cost(c_N|c)\,p(c|N) \leq \sum_{i=1}^{k}\sum_{c\in\mathscr{C}} Cost(c_{N_i}|c)\,p(c|N_i). \tag{2.16}$$

FACT does not use any validation based pruning method similar to CART's cost-complexity minimization.

2.2.5.2 QUEST

Quick, Unbiased, Efficient, Statistical Tree (QUEST) algorithm (Loh and Shih 1997; Lim et al. 2000) was created as a significant improvement of FACT. The general idea and organization of the algorithm remained the same: the method realizes algorithm 2.3 separating feature selection from determination of the split, converts symbolic features to numeric ones in a similar way, and uses statistical tests to make some decisions. The main changes concern how the particular goals are obtained:

- split feature is selected on the basis of another approach to estimate feature importance, aimed at unbiased selection,
- the split is made with quadratic discrimination instead of linear,
- the resulting tree is binary, classes are grouped before the split,
- generalization is obtained with cost-complexity minimization, as in the case of CART.

Loh and Shih (1997) claim that the way they convert symbolic feature to continuous ones is also different in QUEST than in FACT, however they mention that FACT's method first converts the feature symbols into binary "dummy" vectors, and then converts them into real numbers with a method that can split the node into more than two subnodes, which is not acceptable in QUEST. They evidently refer to another version of FACT than the one of Loh and Vanichsetakul (1988), because as described above, the latter uses crimCoord transformation (see appendix section B.1) to convert symbols to numeric values, and the same is done in QUEST. Naturally, there is a difference between the two methods in the way they split the features after the conversion. As in FACT, after the split is determined for the continuous counterpart of a symbolic feature, it can be easily rephrased in the language of original symbols, so in the resulting tree, the continuous feature generated during the analysis is not at all visible.

Feature Selection

Estimation of both continuous and symbolic features with F ratio (for symbolic ones calculated for the derived continuous feature) is prone to more frequent selection of symbolic features than continuous ones, also when they are all independent from the target. The idea of QUEST to get rid of the bias (or at least to try to; see Sect. 2.7 for more discussion) is to compare p-values of independence tests most eligible for each type of features, instead of comparing the F ratios. Continuous attributes are

still analyzed with F statistic, but discrete features are subject to the χ^2-test (no conversion to numeric values is performed at the stage of feature selection). The F and χ^2 values are not directly comparable, but comparing their p-values makes sense. The feature with the smallest p-value (the smallest probability of independence with the target variable) is selected as the best one. If none of the p-values exceeds a user-defined threshold parameter, Levene's F-test for unequal variances is computed for each ordered variable. The tests thresholds are Bonferroni corrected (see lines 5 and 5d of algorithm 2.4 and appendix section A.1.4). If the best result exceeds the threshold, the corresponding feature is selected, otherwise the algorithm returns the feature with the smallest p-value calculated in the first stage (with F-test or $)\chi^2$-test.The method is written formally as Algorithm 2.4.

Algorithm 2.4 (QUEST split feature selection)

Prototype: ***QUESTFeatSel(D,*** α***)***

Input: *Training dataset D described by discrete features* $X_1, \ldots, X_d$ *and continuous features* $X_{d+1}, \ldots, X_f$*, confidence threshold* α*.*

Output: *Feature index.*

The algorithm:

1. *If* $d > 0$ *(there are continuous features)*
   a. *for* $i = 1, \ldots, d$
      $F_i \leftarrow$ *the ANOVA F statistic for feature* X_i
   b. $best_1 \leftarrow \arg\max_{i=1,\ldots,d} F_i$
   c. $\alpha_1 \leftarrow$ *p-value of the adequate F distribution for feature* $best_1$
2. *If* $f > d$ *(there are discrete features)*
   a. *for* $i = d+1, \ldots, f$
      $p_i \leftarrow$ *p-value of the* χ^2*-test of independence between feature* X_i *and class labels*
   b. $best_2 \leftarrow \arg\min_{i=d+1,\ldots,f} p_i$
   c. $\alpha_2 \leftarrow p_{best_2}$
3. $\alpha_{12} \leftarrow \min(\alpha_1, \alpha_2)$
4. *If* $\alpha_{12} = \alpha_1$ *then* $best_{12} \leftarrow best_1$ *else* $best_{12} \leftarrow best_2$
5. *If* $\alpha_{12} \geq \frac{\alpha}{f}$ *(no feature is good enough)*
   a. *for each* $i = 1, \ldots, d$
      $F'_i \leftarrow$ *the ANOVA F statistic for feature* $Z_i = |X_i - \overline{X_i}|$
   b. $best_3 \leftarrow \arg\max_{i=1,\ldots,d} F'_i$
   c. $\alpha_3 \leftarrow$ *p-value of the adequate F distribution for feature* $best_3$
   d. *If* $\alpha_3 < \frac{\alpha}{f+d}$ *then* ***return*** $best_3$
6. ***return*** $best_{12}$

Split Selection

QUEST finds the best split points with *quadratic discriminant analysis* (QDA). Because QUEST is assumed to be a binary tree, the splits are done between two classes. If the problem at hand concerns classification to more than two classes,

they are first grouped into two superclasses, with the clustering*two-means clustering* method (Hartigan and Wong 1979) applied to the set of mean vectors calculated for all the classes. The two-means algorithm is initialized with two most distant class means as the cluster centers. If all the class means are identical, then the most populous class composes superclass A and the rest—superclass B.

In the case of continuous features, a procedure of QDA is directly performed, and in the case of symbolic ones the crimCoord based procedure is run to obtain a continuous substitute for the feature, which is then analyzed with QDA.

The QDA estimates the two classes (A, B) distributions with normal densities and determines the split point as the point of intersection of the two Gaussian curves, being a root of the equation

$$P(A|N)\frac{1}{\sqrt{2\pi}s_A}e^{-\frac{(x-\overline{x}_A)^2}{2s_A}} = P(B|N)\frac{1}{\sqrt{2\pi}s_B}e^{-\frac{(x-\overline{x}_B)^2}{2s_B}}, \tag{2.17}$$

where N is the node being split, $\overline{x}_A, \overline{x}_B$ are the means of class A and B respectively and s_A, s_B are standard deviations observed within the classes. The normal densities parameters (means and standard deviations) are calculated from the samples. Equation (2.17) after simple transformations gets the form of a typical quadratic equation:

$$ax^2 + bx + c = 0, \tag{2.18}$$

where

$$\begin{aligned} a &= s_A^2 - s_B^2, \quad b = 2(\overline{x}_A s_B^2 - \overline{x}_B s_A^2), \\ c &= (\overline{x}_B s_A)^2 - (\overline{x}_A s_B)^2 + 2s_A^2 s_B^2 \log\frac{n_A s_B}{n_B s_A}. \end{aligned} \tag{2.19}$$

The split point is one of the two roots that is closer to $\overline{x}_A$, provided this yields two nonempty subsets.

2.2.5.3 CRUISE

Classification Rule with Unbiased Interaction Selection and Estimation (CRUISE, Kim and Loh 2001, 2003) is a descendant of FACT and QUEST. It is the next significant step towards unbiased feature selection. As its predecessors, CRUISE also fits the general strategy of Algorithm 2.3, but differs in many detailed solutions:

- generates multi-split trees,
- introduces new method for split feature selection, named 2D because of analysis of pairs of features (more precisely some contingency tables), but still supports the method of QUEST (here named 1D) based on comparing p-values of proper independence tests for each type of features (ANOVA F-test for continuous ones and χ^2-test for categorical),

- uses Box-Cox power transformation to adjust class distributions in order to improve the accuracy of LDA.

Similarly to FACT and QUEST, categorical features are transformed with CrimCoord at the start of the analysis and Bonferroni corrections (see appendix section A.1.4) are used when performing multiple statistical tests. Missing values in the data can be ignored or imputed. As the main advantages of CRUISE its authors mention its sensitivity to local interactions between variables (thanks to)2D analysis, which results in more intelligent splits and shorter trees, and its speed obtained in parallel with not statistically significant difference in mean misclassification rates, in comparison to the best methods.

CRUISE can be configured to generate linear combination splits as well as univariate splits.

Feature Selection

Two different methods of feature selection are available in CRUISE. They are named "1D" and "2D" respectively. 1D is exactly the same method as the one used in QUEST. The novelty is the second method based on two-dimensional contingency-tables analysis. This method performs statistical tests for five different types of sources (two marginal tests and three interaction tests). For each case, the space is split into a number of regions and a contingency table with classes as rows and the regions as columns is created. The possible sources and the ways of the contingency tables construction are the following:

- for single numeric variable X, four regions correspond to the sample quartiles of X,
- for single categorical variable, the regions correspond to the values of the variable,
- for a pair of numeric variables, the regions correspond to four quadrants resulting from splits at the sample medians,
- for a pair of categorical variables, the regions correspond to pairs of possible values of the variables,
- for a pair of one numeric and one categorical variable, the regions correspond to combinations of two parts of the numeric attribute (split at the median) and all possible categories of the categorical feature.

The procedure of building the contingency table for each context is noted in Algorithm 2.6 as *ContTable()* and its arguments clearly show which of the five versions is called. Testing contingency tables for pairs of variables is aimed at detecting interactions between features, but the possibilities are significantly limited, because splitting numeric features arbitrarily at medians, can correspond to proper decision borders only accidentally.

Each contingency table is analyzed with Algorithm 2.5 to get a corresponding z-value. Maximum of the 5 z-values (with a bootstrap bias correction factor) points the selected feature according to the rules presented in the algorithm 2.6.

The *BootstrapCorrection()* function used in the algorithm, finds a factor $f \in R$, that brings proper balance between discrete and continuous features (eliminates the

Algorithm 2.5 (z-statistic of a contingency table)

Prototype: ZStatistic(T)
Input: *Contingency table T with n rows and m colums.*
Output: *z-statistic.*
The algorithm:

1. $\chi^2 \leftarrow$ *the Pearson* χ^2 *statistic with* $\nu = (n-1)(m-1)$ *degrees of freedom for* T
2. $W \leftarrow \chi^2 - \nu + 1$
3. *if* $z > 1$ *then (Peizer-Pratt transformation)*
   $z \leftarrow \frac{1}{|W|}(W - \frac{1}{3})\sqrt{(\nu - 1)\log\frac{\nu-1}{\chi^2} + W}$
   else
   $z \leftarrow \sqrt{\chi^2}$
4. ***return*** z

Algorithm 2.6 (CRUISE 2D feature selection)

Prototype: CRUISE2DFeatSel(D, α)
Input: *Training dataset D described by discrete features $\mathbf{X}_1, \ldots, \mathbf{X}_d$ and continuous features $\mathbf{X}_{d+1}, \ldots, \mathbf{X}_f$ and targets $\mathbf{Y}$.*
Output: *z-statistic.*
The algorithm:

1. ***if*** $d = 0$ ***then*** $z_d \leftarrow -\infty$
   else $z_d \leftarrow \max_{i=1,\ldots,d} ZStatistic(ContTable(\mathbf{Y}; \mathbf{X}_i))$
2. ***if*** $f = d$ ***then*** $z_c \leftarrow -\infty$
   else $z_c \leftarrow \max_{i=d+1,\ldots,f} ZStatistic(ContTable(\mathbf{Y}; \mathbf{X}_i))$
3. ***if*** $d = 0$ ***then*** $z_{dd} \leftarrow -\infty$
   else $z_{dd} \leftarrow \max_{i,j=1,\ldots,d} ZStatistic(ContTable(\mathbf{Y}; \mathbf{X}_i, \mathbf{X}_j))$
4. ***if*** $f = d$ ***then*** $z_{cc} \leftarrow -\infty$
   else $z_{cc} \leftarrow \max_{i,j=d+1,\ldots,f} ZStatistic(ContTable(\mathbf{Y}; \mathbf{X}_i, \mathbf{X}_j))$
5. ***if*** $d = 0$ *and* $f = d$ ***then*** $z_{dc} \leftarrow -\infty$
   else $z_{dc} \leftarrow \max_{i=1,\ldots,d;\, j=d+1,\ldots,f} ZStatistic(ContTable(\mathbf{Y}; \mathbf{X}_i, \mathbf{X}_j))$
6. $f \leftarrow BootstrapCorrection(D)$
7. $z \leftarrow \max\{z_d, f z_c, z_{dd}, f z_{cc}, z_{dc}\}$
8. ***if*** $z = z_d$ *or* $z = f z_c$ ***then return*** *the feature maximizing z*
9. ***if*** $z = z_{dd}$ *or* $z = f z_{cc}$ ***then return*** *the feature in the interacting pair with larger marginal z*
10. ***return*** *the categorical feature in the interacting pair*

bias). Bootstrap data samples are generated (the data input is copied and the targets are bootstrapped to make them independent from the input) repeatedly and the 5 z-values are calculated for each bootstrap sample as in Algorithm 2.6. A win is scored for continuous features if $f \max\{z_c, z_{cc}\} \geq \max\{z_d, z_{dd}, z_{dc}\}$ and for discrete features otherwise. Several values of f are tested and for each, the proportion between the wins of continuous and discrete features is calculated. Eventually, if necessary, linear interpolation is used to find the final result: such value of f that its proportion of wins

is equal to the proportion of the counts of continuous and discrete features describing the data.

Split Selection

When a continuous feature is selected for a split, Box-Cox power transformation (see appendix section B.4) is run to make the feature more adequate for linear discriminant analysis (LDA) which decides about the split in the same way as it does in FACT.

If the feature selection stage is won by a categorical variable $\mathbf{X} \in \mathscr{X}^n$, where $\mathscr{X} = \{a_1, \ldots, a_r\}$, then it is converted into r binary indicator variables, so that each value a_i is converted into a unit vector $e_{\mathscr{X}}(a_i) = e_i$ with 1 at position i and 0 at all the others. Such vectors are then projected onto the largest discriminant coordinate (CrimCoord) and the new dimension is treated in the same way as numeric attributes: it is passed to the Box-Cox transformation and split with LDA. The final result is translated to the language of the original, discrete feature, to make it comprehensible.

This is the procedure performed by the 2D method. In 1D, there is a difference that the Box-Cox transformation is not run if the feature was selected by Levene's test. Then, instead, LDA is applied to the absolute deviations from the sample mean at the node.

Because LDA happens to generate splits with no data within, such intervals are divided into halves and merged with their neighbors.

2.2.6 CTree

The pursuit of unbiased feature selection in DT construction has gained many different solutions. One of the most interesting results is the approach of Hothorn et al. (2004, 2006a, 2008) and Zeileis et al. (2008) to a conditional inference framework, capable of unbiased recursive partitioning. The authors found the work of Strasser and Weber (1999) on permutation statistics very useful in DT induction.

The main idea behind the framework of *CTree* is the same as in the case of the FACT family, and as depicted by algorithm 2.3, where feature selection and split finding are separate processes. Here, for feature eligibility estimation, and possibly for best split determination, non-parametric permutation tests are used (in place of the F-test and χ^2-test of the FACT line).

Feature Selection and Stop Criterion

The same permutation tests used for feature selection are helpful in deciding when to stop further splits of a node. The null hypothesis of interest is that the unconditional distribution of the target variable $\mathbf{Y}$ and its conditional distributions with respect to each covariate $\mathbf{Y}|\mathbf{X}_i$ are the same. If we are not able to reject such hypothesis at a tree node, the node should be closed into a leaf with no further splits. Otherwise, the input variable $\mathbf{X}_{i^*}$ providing the least independence of the two distributions (the least p-value) is selected as the best feature for the split.

Given a data sample D of n objects described by m features $\mathbf{X}_1 \in \mathscr{X}_1^n, \ldots, \mathbf{X}_m \in \mathscr{X}_m^n$ and the target variable $\mathbf{Y} = (Y_1, \ldots, Y_n) \in \mathscr{Y}^n$, and a case weight vector $\mathbf{w} \in R^n$, Hothorn et al. (2006b) proposed to measure the association between $\mathbf{Y}$ and $\mathbf{X}_j$, $j = 1, \ldots, m$, by linear statistics of the form:

$$\mathbf{T}_j(D, \mathbf{w}) = vec\left(\sum_{i=1}^{n} w_i g_j(X_{ji}) h(Y_i, \mathbf{Y})^T\right) \in R^{p_j q}, \tag{2.20}$$

where:

- $g_j : \mathscr{X}_j \to R^{p_j}$ is a transformation of feature $\mathbf{X}_j$ (for example, to convert symbolic features to more reasonable form like binary vectors),
- $h : \mathscr{Y} \times \mathscr{Y}^n \to R^q$ is the *influence function* dependent on the responses $\mathbf{Y}$ in a permutation symmetric way,
- vec operator converts the $p_j \times q$ matrix it gets as the argument to a $p_j q$-dimensional vector by column-wise concatenation.

This is a general definition that can be used in both classification and regression tasks with miscellaneous definitions of the feature spaces $\mathscr{X}_j$ and $\mathscr{Y}$ and proper substitutions for functions g_j and h. For example, for the sake of univariate classification trees:

- the features are either continuous (real numbers) or symbolic,
- the target variable space is $\mathscr{Y} = \mathscr{C} = \{c_1, \ldots, c_k\}$,
- the function h can be defined as

$$h(y, \mathbf{Y}) = e_{\mathscr{C}}(y), \tag{2.21}$$

  that is, a k-dimensional unit vector with 1 at the position i such that $y = c_i$ (the dimensions are binary indicator variables),
- for $j = 1, \ldots, m$, the function g_j can be defined as:

$$g_j(x) = \begin{cases} x & \text{if } \mathscr{X}_j = R \\ e_{\mathscr{X}_j}(x) & \text{if } \mathscr{X}_j = \{a_1, \ldots, a_r\}, \end{cases} \tag{2.22}$$

  where $e_{\mathscr{X}_j}(x)$ is an r-dimensional unit vector with 1 at index i such that $x = a_i$.

Under the null hypothesis H_0^j, that the distributions of $\mathbf{Y}$ and $\mathbf{Y}|\mathbf{X}_j$ are identical, the distribution of $\mathbf{T}_j(D, \mathbf{w})$ can be analyzed by means of *permutation tests*. In place of the dependency of $\mathbf{T}_j(D, \mathbf{w})$ on usually unknown joint distribution of $\mathbf{Y}$ and $\mathbf{X}_j$, one can fix the covariates and condition on all possible permutations of the responses. Following Strasser and Weber (1999), the conditional expectation $\mu_j \in R^{p_j q}$ and covariance matrix $\Sigma_j \in R^{p_j q \times p_j q}$ of $\mathbf{T}(D, \mathbf{w})$, given all permutations $\sigma \in S(D, \mathbf{w})$ are:

$$\mu_j = E(\mathbf{T}_j(D, \mathbf{w})|S(D, \mathbf{w})) = vec\left(\left(\sum_{i=1}^{n} w_i g_j(X_{ji})\right) E(h|S(D, \mathbf{w}))^T\right), \tag{2.23}$$

$$\begin{aligned}\Sigma_j &= V(\mathbf{T}_j(D, \mathbf{w})|S(D, \mathbf{w})) \\ &= \frac{w_\Sigma}{w_\Sigma - 1} V(h|S(D, \mathbf{w})) \otimes \left(\sum_{i=1}^{n} w_i g_j(X_{ji}) \otimes w_i g_j(X_{ji})^T\right) \\ &\quad - \frac{w_\Sigma}{w_\Sigma - 1} V(h|S(D, \mathbf{w})) \otimes \left(\sum_{i=1}^{n} w_i g_j(X_{ji})\right) \otimes \left(\sum_{i=1}^{n} w_i g_j(X_{ji})^T\right)\end{aligned} \tag{2.24}$$

where $w_\Sigma = \sum_{i=1}^{n} w_i$ and $\otimes$ is the Kronecker product and the conditional expectation and covariance matrix of the influence function are:

$$E(h|S(D, \mathbf{w})) = \frac{1}{w_\Sigma} \sum_{i=1}^{n} w_i h(Y_i, \mathbf{Y}) \in R^q, \tag{2.25}$$

$$\begin{aligned}V(h|S(D, \mathbf{w})) = \frac{1}{w_\Sigma} \sum_{i=1}^{n} w_i (h(Y_i, \mathbf{Y}) - E(h|S(D, \mathbf{w}))) \\ (h(Y_i, \mathbf{Y}) - E(h|S(D, \mathbf{w})))^T.\end{aligned} \tag{2.26}$$

On the basis of the pq-dimensional statistic $\mathbf{T}$, the final test statistic c can be defined in an arbitrary way. The most natural solution for univariate statistic, suggested by Hothorn et al. (2006b) is:

$$c_{\max}(\mathbf{t}, \mu, \Sigma) = \max_{i=1,\ldots,pq} \left| \frac{(\mathbf{t} - \mu)_i}{\sqrt{\Sigma_{ii}}} \right|. \tag{2.27}$$

Another possibility is a more computationally expensive quadratic form:

$$c_{quad}(\mathbf{t}, \mu, \Sigma) = (\mathbf{t} - \mu)\Sigma^+(\mathbf{t} - \mu)^T. \tag{2.28}$$

In any case, one must be aware that the test statistics c may not be directly comparable. In such circumstances, p-values should be calculated, because they allow for unbiased feature selection. Naturally, the way of calculating p-values is closely bound up with the definition of the c statistic. From theorems proved by Strasser and Weber (1999) it can be derived that asymptotic conditional distribution of c_{max} is normal. Quadratic c_{quad} follows asymptotic χ^2 distribution with degrees of freedom given by the rank of Σ.

To get precise stop criterion, the overall null hypothesis being the conjunction of all hypotheses H_0^j needs to be verified. One can construct and analyze an aggregate statistic for this purpose, but in practice it is preferred to use simple techniques like Bonferroni correction for multiple testing. A significance level α must be provided to

control the pre-pruning. Hothorn et al. (2006b) suggest the default value of $\alpha = 0.05$, but the meta-parameter can also be optimized in a validation-based procedure similar to cost-complexity optimization cost-complexity minimization of CART.

Split Selection

When the feature of split is selected, different ways of finding the best split of the feature can be applied including those of CART, FACT, QUEST and many others. The splits can be binary or multi-way, accordingly. After such suggestion, Hothorn et al. (2006b) proposed another application of the permutation test framework to determine optimal binary splits. According to their approach, given a subset A of the sample space $\mathcal{X}_j$, the linear statistic $\mathbf{T}_j^A$ is defined as

$$\mathbf{T}_j^A = vec\left(\sum_{i=1}^{n} w_i 1_A(X_{ji}) h(Y_i, \mathbf{Y})^T\right) \in R^q, \tag{2.29}$$

where 1_A is the indicator function of set A. Given the conditional expectation μ_j^A and its covariance Σ_j^A calculated with Eqs. (2.23) and (2.24), the optimum split is determined by the set $A^\star$, such that

$$A^\star = \arg\max_{A \subset \mathcal{X}_j} c(\mathbf{t}_j^A, \mu_j^A, \Sigma_j^A). \tag{2.30}$$

Although the optimization extends over all subsets of $\mathcal{X}_j$, the number of binary splits is usually significantly limited. In the case of numeric features, only splits into two disjoint and complementary intervals defined by points between values represented in the data are taken into account. Symbolic features with large number of possible values may also need some restrictions in subset analysis, because of computational complexity.

2.2.7 SSV

Separability of Split Value (SSV, Grąbczewski and Duch 1999, 2000) criterion is defined as a split quality measure, but is not based on the purity gain rule (2.4). It reflects the idea that splitting pairs of vectors belonging to different classes is advantageous, while splitting pairs of vectors of the same class should be avoided if possible. Originally, it has got two forms:

$$\mathrm{SSV}(s, D) \stackrel{def}{=} 2 \cdot \mathrm{SSV}_1(s, D) - \mathrm{SSV}_2(s, D), \tag{2.31}$$

$$\mathrm{SSV}_{lex}(s, D) \stackrel{def}{=} \Big(\mathrm{SSV}_1(s, D), -\mathrm{SSV}_3(s, D)\Big), \tag{2.32}$$

where:

$$\mathrm{SSV}_1(s, D) \stackrel{def}{=} \sum_{i=1}^{n_s} \sum_{j=i+1}^{n_s} \sum_{c \in \mathscr{C}} |D_{s_i,c}| \cdot |D_{s_j} \setminus D_{s_j,c}|, \tag{2.33}$$

$$\mathrm{SSV}_2(s, D) \stackrel{def}{=} \sum_{c \in \mathscr{C}} (|D_c| - \max_{i=1,\ldots,n_s} |D_{s_i,c}|), \tag{2.34}$$

$$\mathrm{SSV}_3(s, D) \stackrel{def}{=} \sum_{i=1}^{n_s} \sum_{j=i+1}^{n_s} \sum_{c \in \mathscr{C}} |D_{s_i,c}| \cdot |D_{s_j,c}|. \tag{2.35}$$

The SSV_1 part counts the pairs of separated objects belonging to different classes—it can be called a reward part. SSV_2 and SSV_3 define, in two different ways, some penalties for splitting objects representing the same class. The $\mathrm{SSV}_{\mathrm{lex}}$ version provides pairs of values, which are compared in lexicographic order, so the second value is considered only in the case of equal first elements.

It is not easy to keep proper balance between the reward part and the penalty part of the criteria like (2.31) or (2.32), because repairing some cases may easily spoil the functionality in other cases. Some analyses of special cases, brought an idea to weight the pairs of separated objects when counting the separability index. Weighting can be seen as a heuristic reflecting the fact that separating pairs of objects is more advantageous, when the objects belong to the majority classes within their sides of the split, and less valuable if the objects are still misclassified after the split. Therefore a parameter weight α was introduced (Grąbczewski 2011) as a factor to diminish the contribution of the minority objects in separated pairs. The result is the following definition:

$$\mathrm{SSV}_\alpha(s, D) \stackrel{def}{=} \sum_{i=1}^{n_s} \sum_{j=i+1}^{n_s} \sum_{\substack{a \in \mathscr{C} \\ b \in \mathscr{C} \\ a \neq b}} W_\alpha(D_{s_i}, a) \cdot |D_{s_i,a}| \cdot W_\alpha(D_{s_j}, b) \cdot |D_{s_j,b}|, \tag{2.36}$$

where

$$W_\alpha(D, c) = \begin{cases} 1 & \text{if } c \text{ is the majority class within } D, \\ \alpha & \text{otherwise.} \end{cases} \tag{2.37}$$

Such definition introduces three levels of contribution of the separated pairs (1, α and α^2), dependent on whether the objects represent the majorities or not. If more than one class is represented in a sample with maximum count, one of them is arbitrarily selected as the majority class (in practice, the one with the smallest index).

Search Methods

The basic approach to searching for SSVtrees is the classical top-down induction method presented as Algorithm 2.1 with an almost exhaustive search for each node

split. The term "almost exhaustive" means that all possible splits are examined if only it is acceptable from the point of view of computational costs and does not introduce evident bias in feature selection (favoring symbolic features with many possible values).

In the case of continuous features, all sensible split points are examined and the one maximizing SSVselected. Here, "sensible" means the ones with nonzero probability that they can bring the best result. It is obvious that only the points between the feature values occurring in the node data are worth any interest. It can be easily proved that the points between objects belonging to the same class can be omitted, because some other points are certainly better. So the analysis procedure of a numeric feature starts with sorting all sample objects by the values of the feature and exploring the sorted values one by one and calculating SSVfor the "sensible" split points.

Symbolic features are split not with points but with subsets of symbols, hence the general term "value" in the name SSV. Exhaustive search would test all possible subsets as the split generators. Such algorithm complexity is exponential, so it can be dangerous to run it for larger counts of feature symbols. It would also give symbolic features significantly greater probability of selection, in comparison to continuous ones, when they are similarly informative. To avoid such bias and the danger of combinatorial explosion, SSVuses a *subset pair enumerator* with a possibility to limit the number of enumerated splits. The enumerator provides subsequent pairs of complementary subsets, by generating the first one and setting the other as the complement to the whole set of symbols. Preference is given to smaller subsets (against their complements), so at first, the singletons are handled then pairs and so on. The limit can be set on the size of the first subset. SSVsets the limit to make the number of tested splits as close to the one of continuous features (and the number of objects in the node) as possible.

Apart from the most popular method of tree construction, based on hill climbing, the SSVapproach has been successfully tested with *beam search* and *lookahead search* (see Sect. 2.5). Both solutions are more computationally expensive than hill climbing, but often can find smaller trees. Different explorations of the area of search methods (Quinlan and Cameron-Jones 1995; Segal 1996; Janssen and Fürnkranz 2009) have shown that these approaches can be successful, but also can lead to so called "oversearching", so one must use these methods with caution.

2.2.8 ROC-Based Trees

Receiver Operator Characteristic (ROC, Green and Swets 1966) is an idea that has found especially much interest in domains close to medicine, where it is very important to differentiate between false positive and false negative answers (statistical *Type I error*type Itype I and *type II error*). When the null hypothesis is the diagnosis of "healthy", erroneous rejecting the hypothesis means a false alarm (False positive answer), while failed rejection of the hypothesis, when it is not true (type II error) means ignoring the illness and no medical treatment, when it should be undertaken,

which can be much more serious. One of the solution of dealing with such differences in errors importance is considering misclassification costs (available in many classification learning machines). Clinicians are usually interested in the information about how many errors of each kind are made by the decision support system, they use. Therefore, they find ROC curves very useful, because they are plots visualizing numbers of erroneous answers of both kinds, given by classifiers. In binary classification, the decisions can be divided into four groups:

Predicted class	Original class	
	Positive	Negative
Positive	True positive (TP)	False positive (FP)
Negative	False negative (FN)	True negative (TN)

On the basis of the four groups we can define many important performance indices, for example:

$$\begin{aligned} accuracy &\stackrel{def}{=} \frac{TP+TN}{TP+FP+FN+TN}, \quad sensitivity \stackrel{def}{=} \frac{TP}{TP+FN}, \\ error &\stackrel{def}{=} \frac{FP+FN}{TP+FP+FN+TN}, \quad specificity \stackrel{def}{=} \frac{TN}{FP+TN}. \end{aligned} \tag{2.38}$$

All the values are real in the range $[0, 1]$. Intuitions about the terms of accuracy and error are common and obvious. *Sensitivity* (*Se*, also called *true positive rate*) shows which part of the "positive" class is correctly detected (in medicine: how big part of patients with the disease is correctly diagnosed). *Specificity* (*Sp*, also called *true negative rate*) tells, how accurately the negative cases are recognized. In medicine, even more important is the value of "1-specificity" (*false positive rate*, or *fall-out*), which represents the amount of negative cases classified as positive.

It is easy to increase the sensitivity of a predictor paying the price of lower specificity, and inversely. In utter cases, classification of all objects as positive results in 100 % sensitivity but 0 % specificity, while constant "negative" answers are 100 % specific but 0 % sensitive. The art of learning is to maximize both sensitivity and specificity.

ROC curve is a line plot depicting transition from one extremity to the other through the most valuable solutions. The axes of the plot are: *1-specificity*specificity and *sensitivity*. Assuming two-element set of classes $\mathscr{C} = \{positive, negative\}$, each classifier is a function $\phi : O \to \{positive, negative\}$ and can be visualized as a single point in the ROC plot.

For a probabilistic classifier $\phi : O \to R^2$, we can easily generate a series of crisp classifiers by shifting the decision threshold θ:

$$\phi^{(\theta)} : O \to \mathscr{C}, \quad \phi^{(\theta)}(o) = \begin{cases} positive & \text{if } \phi(o)_1 \geq \theta, \\ negative & \text{otherwise.} \end{cases} \tag{2.39}$$

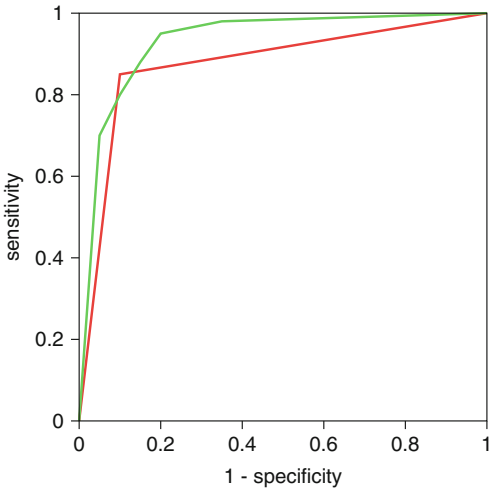


Fig. 2.1 Two example ROC curves: one for crisp and one for probabilistic classifier

Each crisp classifier can be easily converted into a trivial probabilistic classifier (with probabilities 0 or 1 only). Such models can be visualized by an ROC curve based on three points: (0,0), (Se, 1-Sp), (1,1). Each ROC curve starts at point (0,0) and ends at (1,1).

Two example ROC curves are presented in Fig. 2.1. One corresponds to a crisp classifier and one to a probabilistic classifier with several levels of predicted probabilities. The curves are nondecreasing (here, also concave, though not strictly, as they are piecewise linear), because in such families of classifiers, increasing sensitivity is closely bound up with nonincreasing specificity.

In many applications the *area under the ROC curve* (AUC) has been found very attractive index of classifier (family) quality. The larger the AUC, the better the classifier.

AUC Split Criterion

Measuring the area under ROC curve has found applications also in DT construction. Ferri et al. (2002) started their road to *AUCsplit* criterion with an analysis of decision tree models in a similar manner as the transition from crisp to probabilistic classifier described above. They discuss possibilities of different labeling of DT leaves and prove formally an intuitive property that the number of optimal labelings is linearly, not exponentially dependent on the number of leaves. They order the leaves by *local positive accuracy* defined as $\frac{p_N}{p_N+n_N}$, where p_N and n_N are the counts of objects in the node N with labels "positive" and "negative" respectively, and show that optimal labelings are those that give label "positive" to a number of beginning nodes in the sequence and label "negative" to the rest of the nodes. In this way, they obtain a collection of points $P_0, \ldots, P_k$, such that

$$\underset{i=1,\ldots,k}{\forall} P_i = P_{i-1} + \left(\frac{n_i}{n}, \frac{p_i}{p}\right) = \left(\frac{\sum_{j=1}^{i} n_j}{n}, \frac{\sum_{j=1}^{i} p_j}{p}\right), \tag{2.40}$$

where k is the number of leaves in the tree, p_i and n_i are the numbers of objects with labels "positive" and "negative" respectively in $i'th$ DT leaf of the ordered collection, while p and n are the respective sums for all leaves. The points define the ROC curve, which is a piecewise linear function.

The area under such curve can be easily calculated as the sum of the areas of subsequent trapezia.

$$AUC(P_0, \ldots, P_k) = \sum_{i=1}^{k} \frac{y_i + y_{i-1}}{2}(x_i - x_{i-1}), \tag{2.41}$$

where $P_i = (x_i, y_i)$.

This idea is a foundation of the *AUCsplit* criterion (Ferri et al. 2002). Each split s of a node N yields a number of subnodes: $N_1^s, \ldots, N_{n_s}^s$. When the subnodes are sorted by local positive accuracy, they determine ROC points $P_0^s, \ldots, P_{n_s}^s$. The $AUCsplit$ criterion is defined as

$$AUCsplit(s) = AUC(P_0^s, \ldots, P_{n_s}^s). \tag{2.42}$$

It can be used to estimate quality of candidate splits and select the best split of given tree node in the classical top-down DT induction procedure (Algorithm 2.1).

In the case of a crisp classifier, the ROC is determined by three points $(0, 0)$, $(1 - Sp, Se)$, $(1, 1)$, where Sp and Se are the classifier's specificity and sensitivity. The AUC of such ROC is equal to $\frac{1}{2}(Se + Sp)$ which is the same as *balanced accuracy* (in two-class problems). It gives some specific view of the $AUCsplit$ optimization.

Ferri et al. (2002) have tested DTs based on their criterion in combination with the Pessimistic Error Pruning, described in more detail in Sect. 2.4.2.1, but any other pruning method can also be used to increase generalization abilities of the trees.

One of the most serious drawbacks of the *AUCsplit* criterion is that it can be used only for two-class problems. Although it is not difficult to generalize the ideas to arbitrary number of classes, the authors have not proposed such generalizations. Instead they combined the method with the 1-vs-1 strategy (Hand and Till 2001; Ferri et al. 2003) and proposed some improvements to provide more attractive probability estimates from the trees. Doetsch et al. (2009) used the criterion with maximization of the criterion over all class pairs in their Logistic Model Trees.

2.3 Multivariate Decision Trees

Splitting decision tree nodes on the basis of univariate functions is very attractive because of models comprehensibility. A price to pay for the comprehensibility is not

too flexible shape of decision borders. Univariate conditions correspond to separation hyperplanes perpendicular to the axes of the selected features. When the splits are allowed to base on hyperplanes without restrictions, the node conditions may contain linear combinations in place of single features. Resulting trees are usually simpler in the sense of the number of nodes or leaves, but it is important to realize that the nodes are more complex so the overall tree complexity is not necessarily lower. At the same time, it gets much more difficult to interpret the resulting classification functions.

Many DT induction algorithms facilitate building both kinds of DTs by proper parameter settings. For example, some of the algorithms described above as the univariate methods (CART, FACT, QUEST and CRUISE) can also determine linear combinations of original features as new variables to be split. Many others have been especially designed to perform multivariate splits. In subsections below, a subjective selection of algorithms has been presented in more detail, however it must be pointed out that many other interesting approaches can also be found in the literature. It should certainly be recommended to also read the original publications about such methods like SADT (Heath et al. 1993), SPRINT (Shafer et al. 1996), CLOUDS (Alsabti et al. 1998), CMP (Wang and Zaniolo 2000), SODI (Lee and Olafsson 2006), Cline (Amasyali and Ersoy 2008) and many others.

2.3.1 LMDT

One of the first, very popular approaches to building DTs with splits based on linear combinations of features is the LMDT algorithm (Utgoff and Brodley 1991; Brodley and Utgoff 1992a,b). Similarly to the univariate trees described above, the main engine of LMDT is Algorithm 2.1. The main difference is inside the *BestSplit* procedure used in the approaches. LMDT does not select a feature for the split or analyze the dataset feature by feature to find the best split, but builds a *linear machine* for each node to split it. Each linear machine is a set of k linear discriminant functions combined in a single machine to classify data objects to one of the k classes of the problem being solved ($\mathscr{C} = \{c_1, \ldots, c_k\}$). Let $O \subseteq R^m$ be the domain of the classification task. The discriminant functions of the machine are

$$g_i : O \times \{1\} \to R, \quad g_i(\mathbf{y}) = \mathbf{w}_i^T \mathbf{y} \quad i = 1, \ldots, k. \tag{2.43}$$

The input vectors are extended by one dimension with constant value of 1 to facilitate versatility of hyperplanes definition. The linear machine classification function is

$$\phi_{\mathbf{g}} : O \to \mathscr{C}, \qquad \phi_{\mathbf{g}}(\mathbf{x}) = c_i \Leftrightarrow \underset{j \neq i}{\forall} g_j(\mathbf{x}, 1) < g_i(\mathbf{x}, 1). \tag{2.44}$$

In theory, when no unique maximum value of $g_i(\mathbf{x}, 1)$ exists, the value of the linear machine is undefined. Practical implementations usually select arbitrarily one of the joint winning classes if there is a draw, because it is more advantageous in usual

classification tests to select one of the classes and have some chance to guess the result, than to give up because of equal probabilities of two classes.

The training process of an LMDT linear machine, tests randomly selected training vectors, and if the class assigned to the vector by the machine is incorrect (say c_j instead of c_i), then the weight vectors $\mathbf{w}_i$ and $\mathbf{w}_j$ are adjusted appropriately, to correct the classification of the vector. The detailed procedure is presented as Algorithm 2.7.

Algorithm 2.7 (Thermal linear machine training process)

Prototype: ***LMTraining(D,a,b)***

Input: *Training dataset* $D \subseteq O \times \mathscr{C}$ ($O \subseteq R^m$), *thermal parameter adjustment values* a *and* b (*default* $a = 0.995$, $b = 0.0005$).

Output: *Linear machine* $\phi_{\mathbf{g}}$.

The algorithm:

1. $\beta \leftarrow 2$
2. ***for*** $i = 1, \ldots, m$ ***do***
   $\mathbf{w}_i \leftarrow \mathbf{0}$
3. ***while*** $\beta \geq 0.001$ *and* $Accuracy(\phi_{\mathbf{g}}, D) \leq 0.99$ ***do***
   a. *Select an instance* $(\mathbf{x}, c_i) \in D$ *at random*
   b. ***if*** $\phi_{\mathbf{g}}(\mathbf{x}) = c_j$ *and* $j \neq i$ ***then***
      i. $\mathbf{y} \leftarrow [x_1, \ldots, x_m, 1]^T$
      ii. $k \leftarrow \frac{(\mathbf{w}_j - \mathbf{w}_i)^T \mathbf{y}}{2\mathbf{y}^T \mathbf{y}}$
      iii. ***if*** $k < \beta$ ***then***
         A. $c \leftarrow \frac{\beta^2}{\beta + k}$
         B. $\mathbf{w}_i \leftarrow \mathbf{w}_i + c\mathbf{y}$
         C. $\mathbf{w}_j \leftarrow \mathbf{w}_j - c\mathbf{y}$
         D. ***if*** *the magnitude of* $\phi_{\mathbf{g}}$ *decreased but increased in previous adjustment* ***then***
            $\beta \leftarrow a\beta - b$
4. ***return*** $\phi_{\mathbf{g}}$

The term *magnitude of* $\phi_{\mathbf{g}}$, referred to in item 3(b)iiiD of the algorithm, means the sum of magnitudes of the weights $\mathbf{w}_i, i = 1, \ldots, m$.

If the problem is linearly separable, then the procedure will find a solution in a finite time (Duda et al 2001). Otherwise, error corrections would not end. Hence the parameter β has been introduced to realize the idea of *thermal perceptron* (Frean 1990). It is reduced from time to time to simulate the annealing phenomenon, which guarantees convergence of the process also in the case of linearly non-separable data.

Variable k is set to the *absolute error correction* needed to correctly classify the misclassified object. But it is not used directly in weight change formulae. To eliminate deterioration in the convergence process caused by error correction for the cases of misclassified instances located very far or very close to the decision border, the c parameter controlling weight changes is set to $\frac{\beta^2}{\beta + k}$, which guarantees process stability.

To help the thermal linear machine separate the classes, the training data objects should be appropriately prepared. Utgoff and Brodley (1991) proposed to standardize numeric features before the process and convert symbolic features to a number of binary features. If the feature has just two possible symbols, they can be encoded as +1 and −1 respectively. Otherwise, the symbolic attribute is encoded by a number of binary features equal to the number of possible symbols. For a given data object, one of the new features (the one corresponding to the symbol assigned to the object) gets the value of +1 and all the remaining features get −1. Missing values are replaced by 0s, which correspond to the means of the values observed in the training sample.

It may happen that a linear machine does not in fact split the node—all training data objects belong to the same part of the feature space split. In such cases the node gets closed as a leaf, regardless of the fact that it is not pure.

Minimization of arbitrary misclassification cost functions was introduced to the LMDT approach by Draper et al. (1994). They assigned proportions to the classes (all equal to 1 at start) to reflect the misclassification costs and respect them in the thermal learning of the linear machine.

Variable Elimination

To make the models of LMDT as simple as possible, and sometimes more accurate, Utgoff and Brodley (1991) also described a technique to eliminate variables during the learning processes. They proposed to repeat training after elimination of the feature that contributes least to the discrimination. The best solutions, found so far, must be recorded and new ones compared to them so as to estimate if the results do not deteriorate. The comparisons performed by Utgoff and Brodley (1991) used t-test with $\alpha = 0.01$ to estimate if the new results are not significantly worse than the saved result. Moreover, they used an additional threshold parameter defining the size of acceptable decline in accuracy when the feature is eliminated.

After a number of train and eliminate cycles, the best result, saved in appropriate time, is returned as the final linear machine. The scenario described here is the strategy of *sequential backward elimination* (one of the fundamental approaches to feature selection). Brodley and Utgoff (1992a) have also proposed an approach of *sequential forward selection*, where the best single feature is selected first, and then subsequent features are added one by one, so as to maximize the increase of a merit criterion.

Moreover, they have suggested a combination of the two strategies, referred to as *heuristic sequential search*. Its idea is to run the first stages of both approaches (Forward selection and Backward elimination) and select the better of the two. Such initial test may detect, whether there are many noise features that spoil the result or most of the features are important, and select the method accordingly.

2.3.2 OC1

Oblique Classifier 1 (OC1, Murthy et al. 1993; Murthy et al 1994; Murthy 1997) is a method for DT construction by means of a search for optimal hyperplane separating classes of objects. The search uses a heuristics to find local minima and ideas of non-deterministic approaches to get out of the minima in the pursuit of better solutions.

At each DT node a single hyperplane is determined, so the resulting trees are binary. Similarly to the LMDT approach, the hyperplanes are defined by $m + 1$-dimensional vectors, where m is the dimensionality of the object space. The key procedure of OC1 is its, so called, *perturbation method*, adjusting one selected coefficient in the hyperplane to maximize a measure of impurity of the hyperplane split.

Algorithm 2.8 (OC1 hyperplane perturbation algorithm)

Prototype: OC1Perturb(w,d,D,Impurity)

Input: *Initial hyperplane parameters* $\mathbf{w} = (w_1, \ldots, w_{m+1})$*, index of the dimension to be perturbed* d*, training dataset* $D \subseteq O \times \mathscr{C}$ $(O \subseteq R^m)$*,* $D = \{(\mathbf{x}_i, c_i), i = 1, \ldots, n\}$*, hyperplane split impurity measure Impurity.*

Output: *Modified hyperplane parameters* $\mathbf{w}$*.*

The algorithm:

1. ***for*** $i = 1, \ldots, n$ ***do***
   a. $V_i \leftarrow \sum_{j=1}^{m} w_j x_j + w_{m+1}$
   b. $U_i \leftarrow \dfrac{w_d x_{id} - V_i}{x_{id}}$
2. *Sort* $U_1, \ldots, U_n$ *in nondecreasing order*
3. $\mathbf{w}' \leftarrow \mathbf{w}$
4. $w'_d \leftarrow$ *the best 1-D split of the sorted* $U_1, \ldots, U_n$
5. ***if*** $Impurity(\mathbf{w}, D) < Impurity(\mathbf{w}', D)$ ***then***
   a. $w_d \leftarrow w'_d$
   b. $stagnant \leftarrow 0$

   else if $Impurity(\mathbf{w}, D) = Impurity(\mathbf{w}', D)$ ***then***
   a. $w_d \leftarrow w'_d$ *with probability* $e^{-stagnant}$
   b. $stagnant \leftarrow stagnant + 1$
6. ***return*** $\mathbf{w}$

The procedure is presented as Algorithm 2.8. It calculates U_i values for each data object, sorts the objects by nondecreasing values of U_i and performs linear search to find the best 1-D split of the data.

Murthy et al. (1993) performed the hyperplane perturbations in three different ways:

Seq: The coefficients were perturbed one by one in sequence, many times, until none of the coefficient values were modified:

Repeat

1. $\mathbf{v} \leftarrow \mathbf{w}$
2. ***for*** $i = 1, \ldots, m+1$ ***do***
   $\mathbf{w} \leftarrow$ *OC1Perturb(***w***,i,D,Impurity)*

until $\mathbf{v} = \mathbf{w}$

Best: Each coefficient was perturbed independently, to get the one providing maximum impurity reduction. The optimization was run until the same coefficient was returned twice in sequence:

Repeat

1. $j \leftarrow$ *the coefficient providing maximum impurity reduction when perturbed*
2. *OC1Perturb(***w***,j,D,Impurity)*

until j *is the same as in the previous iteration*

R-50: The coefficient to be perturbed was selected randomly 50 times:

for $i = 1, \ldots, 50$ ***do***

1. $j \leftarrow$ *random integer between* 1 *and* $m+1$
2. *OC1Perturb(***w***,j,D,Impurity)*

Escaping from Local Minima

The procedures, described above, optimize hyperplanes in such a way that when a local minimum is reached, no further perturbation reduces the impurity. To increase the probability of finding global minimum, two techniques were applied by Murthy et al. (1993): perturbing coefficients in a random direction and choosing multiple initial hyperplanes. In the first technique, they selected randomly a vector $\mathbf{r} = (r_1, \ldots, r_{m+1})$ and analyzed hyperplanes determined by vectors of the form $\mathbf{w} + \alpha\mathbf{r}$. In a perturbation procedure analogous to the algorithm 2.8, they calculated the best value of α from the point of view of hyperplane impurity. If the new hyperplane impurity was lower than that of $\mathbf{w}$, the perturbation procedure was continued for the new coefficients. Otherwise, the hyperplane of $\mathbf{w}$ was returned as the final result.

The other method was just to start with different initial hyperplane vectors $\mathbf{w}$ and select the best of the local minima found with the OC1 optimization procedure.

Hyperplane Impurity Measures

Algorithm 2.8 is parameterized by the method to calculate impurity of a hyperplane in the context of particular dataset. Murthy et al. (1993) have proposed three methods to measure such impurity:

max minority : $MM(s, D) = \max(\min(|D_{s_1,c_1}|, |D_{s_1,c_2}|), \min(|D_{s_2,c_1}|, |D_{s_2,c_2}|))$,
sum minority : $SM(s, D) = \sum_{i=1}^{2} \min(|D_{s_i,c_1}|, |D_{s_i,c_2}|)$,
sum of impurity : $SI(s, D) = \sum_{i=1}^{2} \sum_{(\mathbf{x},c)\in D_{s_i}} (Bin(c) - avg_i)^2$, where $Bin(c) \in \{0, 1\}$ is a binary coding of the classes and $avg_i = \frac{1}{|D_{s_i}|} \sum_{(\mathbf{x},c)\in D_{s_i}} Bin(c)$.

Although the measures are defined for split s in general, while Algorithm 2.8 calls the *Impurity* method with vectors determining hyperplanes, it is easy to make them compatible also formally, as the hyperplane splits the space into halves. Also, it is not difficult to extend the definitions given by Murthy et al. (1993) to arbitrary numbers of classes.

Other Solutions

Murthy et al (1994) have listed additional measures like information gain, Gini index or twoing criterion. In fact, any measure of split quality may be used here.

Similarly, any pruning method is suitable for OC1 trees, but Murthy et al (1994) have used the cost-complexity optimization cost-complexity pruning of Breiman et al. (1984).

In the original definition, missing values in the data were replaced by mean values of the attribute, before training or testing OC1.

2.3.3 LTree, QTree and LgTree

Probabilistic linear trees LTree Gama(1997) result from a combination of three ideas: *divide and conquer* methodology of decision trees, *linear discriminant analysis* and *constructive induction*.

According to the paradigm of constructive induction, each split of an LTree node (the corresponding training data sample) is performed in two independent steps:

- new attributes construction (linear combinations of existing features),
- best split determination by a technique for univariate DT construction.

Such scheme and other ideas introduced to LTree induction caused important differences between the approach and other methods, introduced earlier:

- the number of features describing data can differ between the nodes of the same classification tree,
- new attributes, once created, are propagated downwards, so that other splits in the branch can also use them,
- the trees estimate probabilities by an analysis of data distributions in the whole path followed when classifying an object: from the root to appropriate leaf.

Attribute Construction

When new features are generated for the sake of a tree node split, it is important that it brings as much information about class discrimination as possible. To determine the class of a given object $\mathbf{x}$, it is sufficient to determine conditional probabilities $P(c|\mathbf{x})$ for each possible class c. The most probable class should be indicated as the class of $\mathbf{x}$. Given the Bayes theorem:

$$P(c|\mathbf{x}) = \frac{P(c)P(\mathbf{x}|c)}{P(\mathbf{x})}. \tag{2.45}$$

To determine the winner class, the denominator (common for all the classes) can be ignored leaving $P(c)P(\mathbf{x}|c)$ as the value to be estimated for each class $c \in \mathscr{C}$.

Linear discriminant analysis assumes that each class is normally distributed and that all classes share the covariance matrix Σ. In such case, maximization of $P(c)P(\mathbf{x}|c)$ is equivalent to maximization of $P(c)f_c(\mathbf{x})$, where $f_c(\mathbf{x})$ is the probability density function for class c. Multidimensional normal density function of mean μ and covariance Σ is given by:

$$f_{\mu,\Sigma}(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^n|\Sigma|}} e^{-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)}. \tag{2.46}$$

After some more simplifications (like comparing logarithms instead of exponential expressions and throwing out the constant components for all classes) one can easily get:

$$P(c|\mathbf{x}) \propto \log(P(c)) + \mathbf{x}^T\Sigma^{-1}\mu_c - \frac{1}{2}\mu_c^T\Sigma^{-1}\mu_c. \tag{2.47}$$

Therefore, Gama(1997) defined the linear discriminant hyperplane as:

$$H_c = \alpha_c + \mathbf{x}^T\beta_c, \tag{2.48}$$

where $\alpha_c = \log(P(c)) - \frac{1}{2}\mu^T\Sigma^{-1}\mathbf{x}$ and $\beta_c = \Sigma^{-1}\mu_c$. The mean μ_c and covariance matrix Σ are calculated as the training sample mean and pooled covariance matrix. Because in some circumstances the polled covariance matrix may be singular, Gama (1999) suggested using SVD (see appendix section B.2) to reduce the features that cause the singularity.

The hyperplane formula (2.48) defines new features added at the tree node. New features are constructed for $k_N - 1$ classes of the k_N represented in the node N by the number of objects greater than the number of attributes (Gama (1999) suggested twice the number of attributes).

When adding the attributes, the α_c is constant for all the objects and could be omitted. In fact, the fragments ignored when deriving (2.47) as not important from the point of view of winner-class selection for an object are not meaningless when comparing different objects and could play significant role here.

Gama (1999) also proposed two other discriminant approaches to be used in analogous ways as the linear one: *quadratic* and *logistic discriminants*, yielding algorithms named *QTree* and *LgTree* respectively. Quadratic discriminants can be inferred similarly to the linear ones from the assumption of normal distributions but without the constraint of equal covariance matrices for all the classes. Similar reasoning as for linear discriminants brings the conclusion that

$$P(c|\mathbf{x}) \propto \log(P(c)) - \frac{1}{2}\log(|\Sigma_c|) - \frac{1}{2}(\mathbf{x}-\mu_c)^T\Sigma_c^{-1}(\mathbf{x}-\mu_c), \tag{2.49}$$

where Σ_c is the covariance matrix for class c. Again, possible problems with covariance matrix singularity can be solved by SVD analysis and feature elimination.

In the LgTree algorithm, generation of new features starts with selecting one of the classes $(c_1, \ldots, c_k)$ as so called *reference class* (say the last one c_k) and $k-1$ vectors $\boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_{k-1}$ to estimate the conditional class probabilities:

$$P(c_k|\mathbf{x}) = \frac{1}{\sum_{j=1}^{k-1} e^{\boldsymbol{\beta}_j \mathbf{x}}}, \quad P(c_i|\mathbf{x}) = \frac{e^{\boldsymbol{\beta}_i \mathbf{x}}}{\sum_{j=1}^{k-1} e^{\boldsymbol{\beta}_j \mathbf{x}}}, \quad i = 1, \ldots, k-1. \tag{2.50}$$

Then the Newton-Raphson iterative procedure is used to find such βs that maximize the likelihood:

$$L(\boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_{k-1}) = \prod_{i=1}^{k} \prod_{\mathbf{x} \in D_{c_i}} P(c_i|\mathbf{x}). \tag{2.51}$$

As in the case of linear discriminants, the subsequent $\boldsymbol{\beta}$s can be used as the projection vectors to create new features discriminating the classes.

Split Criteria

The constructive approach can be combined with any method of univariate DT induction. By default Gama (1997, 1999) used a method very similar to C4.5: the Information gain criterion was used for split quality estimation. Continuous features splits were binary, while using categorical features for splits resulted in multi-way branching (as many subnodes as symbols of the feature).

Decision Making

As mentioned above, the LTree family methods have implemented a special way of class probability estimation, based on data distribution in the whole path from the root to the leaf of interest, instead of the most common solution to base just on the distribution in the leaf. Gama (1997) suggested weighting class proportions in a given node N with its parents probabilities:

$$P(c|N) = \frac{P(c|Parent(N)) + w\frac{n_{N,c}}{n_N}}{1+w}. \tag{2.52}$$

Only for the root node, class probabilities were based directly on class frequencies in the training data sample. The parameter w was set to 1 by default, but its value was up to the user. Such definition of class probabilities may cause that the winner class is not the majority class of the leaf.

Pruning

To obtain well generalizing trees, Gama (1997) suggested using one of two methods: the Error-Based Pruning of C4.5 (but without the option of grafting the best subtree in the place of its parent, when the subtree seems to be better) and another one, strongly related to the way of calculating probabilities, described above. When the sum of errors made by subbranches of a given node, was not less than the error of the (parent) node acting as a leaf, then the node was turned into a leaf. In the most common approach of estimating probabilities by leaves frequencies, it cannot happen that the children nodes make, in total, more errors than their parent, but it is possible with so untypical probability estimation, as described above.

Algorithm 2.9 (Iterative refiltering)

Prototype: ***IterativeRefiltering(D,LM)***
Input: *Training dataset D, a learning machine LM.*
Output: *A model.*
The algorithm:

1. ***repeat***
   a. $M \leftarrow Train(LM, D)$ */* model M is the result of training LM on D */*
   b. $D' \leftarrow Classify(M, D)$ */* D' is D with classes predicted by the ones provided by M */*
   c. $D \leftarrow D \cap D'$
   while $D' \neq D$ ***do***
2. ***return*** *M*

2.3.4 DT-SE, DT-SEP and DT-SEPIR

John (1995b, 1996) has contributed to the field of multivariate DT induction by presenting the ideas of using soft criteria for split quality estimation and iterative refiltering in DT regularization. Soft miltidimensional criteria are not compatible with symbolic features, so the family of DT-SE methods needs some data preprocessing to get rid of the symbols which have no sensible order. In the experiments of John (1996), all unordered categorical attributes were converted into corresponding 0-1 indicator variables (see p.6) to prevent introduction of accidental information or hiding existing information by arbitrary symbols ordering. Therefore, binary splits are most natural in this family of trees, although one could easily apply the same optimization methods for analyzing multi-way splits.

Soft Split Criteria and Soft Entropy

In general, the proposition of John (1996) was softening different criteria evaluating split quality, so that they can be optimized with methods like gradient descent. To make it easier, he described the problem of finding an optimal split of a dataset D as the problem of finding parameters $\theta^\star \in \Theta$ of a split function g_θ that minimize some split quality measure I:

$$\theta^\star = \arg\min_{\theta \in \Theta} I(D|g_\theta). \tag{2.53}$$

The domain Θ can be arbitrarily defined for particular problems: it may contain simple, single-value parameters like split thresholds, but also more sophisticated objects consisting of more important values parameterizing the split function g_θ.

The objective function I may be any split criterion including information gain, Gini index and so on. Although most of the criteria are defined on the basis of cardinalities of datasets $D \in O$, they can be easily softened by using fuzzy membership functions and fuzzy cardinalities: $|D|_w = \sum_{x \in O} w(x)$. An example of a fuzzy membership function is the sigmoid function performing linear discriminant splitting:

$$g_\theta(\mathbf{x}) = \frac{1}{1 + e^{-\theta^T \mathbf{x}}}. \tag{2.54}$$

The composition of the objective function from a split function like (2.54) and split quality measure I helps in using gradient methods for minimization. By the chain rule $\frac{\delta I}{\delta \theta} = \frac{\delta I}{\delta g_\theta} \frac{\delta g_\theta}{\delta \theta}$, so if the quality measure function is differentiable with respect to g_θ and g_θ with respect to θ, performing gradient descent minimization is very easy.

The experiments of John (1996) were performed with the information gain function as the split quality measure and the split function g_θ defined by (2.54). More precisely, the negated sum of entropies (2.5) of the datasets resulting from the split needs to be calculated, as the entropy of the whole dataset is constant for all splits. The result got the name of *soft entropy* (SE) which gave rise to the name DT-SE. A simple steepest descent procedure was used for objective function minimization. After the $\theta^\star$ was found, the dataset was crisply split into two parts corresponding to the conditions $g_{\theta^\star}(\mathbf{x}) \geq \frac{1}{2}$ and $g_{\theta^\star}(\mathbf{x}) < \frac{1}{2}$ which are equivalent to a typical linear discriminant solution: $\theta^{\star T}\mathbf{x} \geq 0$ or $\theta^{\star T}\mathbf{x} < 0$.

Pruning

Tree regularization process prepared by John (1996) for his DT-SE trees used two kinds of methods: pre-pruning and retraining. The former is very simple: tree nodes are not further split if one of the classes has less than 5 representatives. Naturally, the 5 is the value of a parameter, but in John's experiments it was set just to 5. The technique of retraining got the name of *iterative refiltering* and was borrowed from the field of regression methods, where it had been used under the names of *robust* (Huber 1977) or *resistant* (Hastie and Tibshirani 1990) *fitting*. A general definition of

iterative refiltering is presented as algorithm 2.9. It could be made even more general by replacing the call of *Classify()* method by a more general one and generalizing the stop condition of the loop. The idea behind the algorithm is that if an object is classified incorrectly by the model, then it probably spoils not only the leaves of the tree but also the nodes along the whole path from the root, so it should be advantageous to get rid of such object during learning. The procedure just repeats learning, testing the resulting model and removing the data objects incorrectly classified by the model, until the subsequent model classifies correctly all the objects that remain in the training dataset. The algorithm was presented first by (John 1995a) in application to C4.5 creating the method named *Robust C4.5*. The training procedure called within iterative refiltering must be equipped with some generalization mechanisms. Therefore, in application to DT-SE, the simple stop criterion was used. C4.5 has its own pruning strategy, so it was used for tree regularization.

The DT-SE method augmented by the simple stop-criterion for the purpose of pruning was named DT-SEP and the version using also Iterative refiltering is called DT-SEPIR.

2.3.5 LDT

Yildiz and Alpaydin (2000, 2005a) performed an analysis of six aspects of DT induction algorithms and proposed their own algorithm named *Linear Decision Trees* (LDT). The six aspects are:

- node type: univariate or multivariate,
- branching factor: splitting to two or more subnodes,
- grouping classes into two superclasses for binary trees,
- split quality measures,
- minimization methods for finding best splits.

The resulting LDT algorithm creates binary trees, performing Discriminant analysis at each node (with options of univariate or multivariate splits and Linear discriminant analysis linear or)Quadratic discriminant analysis after the classes with representatives in the node, are grouped into two superclasses with one of two algorithms.

The Discriminant Analysis

LDT follows the idea of *Fisher's linear discriminant analysis* (FDA). The goal of FDA is finding the hyperplane providing the best separation of two groups of objects. In other words, the direction $\mathbf{w}^\star$ that maximizes the distances between different class centers while minimizing the average variance within classes:

$$\mathbf{w}^\star = \arg\max_{\mathbf{w}\in R^k} \frac{w^T \mathbf{S}_B w}{w^T \mathbf{S}_W w}, \tag{2.55}$$

where S_B is the between-class covariance matrix and S_W is the within-class covariance matrix:

$$\mathbf{S}_B = (\overline{x}_{c_1} - \overline{x}_{c_2})(\overline{x}_{c_1} - \overline{x}_{c_2})^T, \tag{2.56}$$

$$\mathbf{S}_W = \sum_{c \in \{c_1, c_2\}} \sum_{\mathbf{x} \in c} (\mathbf{x} - \overline{x}_c)(\mathbf{x} - \overline{x}_c)^T. \tag{2.57}$$

The solution of the maximization problem is

$$\mathbf{w}^\star = \mathbf{S}_W^{-1}(\overline{x}_{c_1} - \overline{x}_{c_2}), \tag{2.58}$$

so the projection $\mathbf{w}^{\star T}\mathbf{x}$ provides optimal class discrimination. Assuming normal distributions of the separated groups, the optimal threshold in the new dimension is

$$w_0 = -\frac{1}{2}(\overline{x}_{c_1} + \overline{x}_{c_2})^T \mathbf{S}_W^{-1}(\overline{x}_{c_1} - \overline{x}_{c_2}) - \log \frac{n_{c_1}}{n_{c_2}}. \tag{2.59}$$

Alternatively, one can analyze all possible split positions and select the best one according to a given quality measure like classification accuracy, entropy, Gini index and so on.

The new dimension created as a linear combination of the original variables describing the data can be handled in the same way as the original features in univariate DTs. The split point w_0 of (2.59) is the optimal split point, on the assumption of equal variances of the new feature in the two groups. When resigning from the assumption about the parameters of normal distributions, the QDA analysis can be done to determine the split point in the same way as in QUEST (see Eqs. (2.17), (2.18), and (2.19)). In the case of equal variances the quadratic equation has one root equivalent to w_0 of (2.59). Otherwise, there can be two roots. If so, the one between the means of the groups (if exists) is selected. If not, or both roots are outside the interval designated by the means, then the middle point between the means is used as the split point.

Symbolic features are transformed to the appropriate number of Indicator variables, and the split is found in such space. As described in the case of QUEST, such splits can be easily decoded back to the original feature and verbalized in the language of the symbolic input feature.

Avoiding Problems of Covariance Matrix Singularity

When a linear dependency exists between two variables describing the data, the matrix $\mathbf{S}_W$ is singular and $\mathbf{S}_W^{-1}$ does not exist. Similarly to the approach of FACT, Yildiz and Alpaydin (2000) propose using *principal component analysis* (PCA) to get rid of the undesirable features. The difference between the two is the way the limitation is introduced. Here, provided the Eigenvalues $\lambda_1, \ldots, \lambda_r$ in nonincreasing order, and the associated Eigenvectors $\mathbf{c}_1, \ldots, \mathbf{c}_r$, minimum d is determined, such

that d initial eigenvectors explain more than ε of the variance:

$$\frac{\lambda_1 + \cdots + \lambda_d}{\lambda_1 + \cdots + \lambda_d + \cdots + \lambda_r} > \varepsilon. \tag{2.60}$$

Then, each data vector $\mathbf{x}$ is transformed into the space of selected eigenvectors to

$$\mathbf{z} = (\mathbf{c}_1^T \mathbf{x}, \ldots, \mathbf{c}_d^T \mathbf{x})^T, \tag{2.61}$$

and the LDA is performed in the new space, where the inverse of corresponding $\mathbf{S}_W$ certainly exists. The solution is then transformed back to the original space.

Grouping into Superclasses

When the number k of classes is greater than 2, the LDA procedure described above is preceded by the stage of grouping the classes into two superclasses. Yildiz and Alpaydin (2000) proposed two algorithms for this purpose: one based on *selection* and the other on *exchange*. Unlike the method of QUEST (ClusteringTwo-means clustering), the methods of LDT are supervised. The selection method is presented as Algorithm 2.10. Different random selection at start of the process may generate different superclasses. Yildiz and Alpaydin (2000) suggested to select two most distant classes in place of the random selection.

Algorithm 2.10 (LDT superclasses by selection)

Prototype: ***SuperclassesBySelection(D, SQM)***
Input: *Training dataset D with classes* $\mathscr{C} = \{c_1, \ldots, c_k\}$*, a Split quality measure SQM.*
Output: *Superclasses A and B.*
The algorithm:

1. *Select two classes* c_a *and* c_b *from* $\mathscr{C}$ *at random*
2. $A \leftarrow c_a$
3. $B \leftarrow c_b$
4. $\mathscr{C} \leftarrow \mathscr{C} \backslash \{c_a, c_b\}$
5. ***while*** $\mathscr{C} \neq \emptyset$ ***do***
   a. $s \leftarrow$ *the split provided by the FDA for data D and classes A and B.*
   b. *Select the class* $c \in \mathscr{C}$ *that added to A or B results in the best result returned by SQM*
   c. ***if*** *c maximized SQM when added to A* ***then***
      $A \leftarrow A \cup c$
      else
      $B \leftarrow B \cup c$
   d. $\mathscr{C} \leftarrow \mathscr{C} \backslash \{c\}$
6. ***return*** *A and B*

The second solution offered by LDT is Algorithm 2.11 constructing Superclasses by exchange, proposed by Guo and Gelfand (1992). It divides the classes into two superclasses and then moves the classes from one superclass to the other so as to maximize Information gain of LDA splits. Yildiz and Alpaydin (2000) also suggested a heuristic to replace random initialization: they started with two most distant classes and added remaining ones to appropriate parts, according to the rule of minimum inter-mean distance.

Algorithm 2.11 (LDT superclasses by exchange)

Prototype: ***SuperclassesByExchange(D)***
Input: *Training dataset D with classes* $\mathscr{C} = \{c_1, \ldots, c_k\}$.
Output: *Superclasses A and B.*
The algorithm:

1. $A \leftarrow \bigcup_{i \leq \frac{k}{2}} c_i, \quad B \leftarrow \bigcup_{i > \frac{k}{2}} c_i$
2. ***repeat***
   a. $s \leftarrow$ *the split provided by the FDA for data D and classes A and B*
   b. $IG_0 \leftarrow IG(s, D)$ */* information gain – see equation (2.6) */*
   c. ***for*** $i = 1, \ldots, k$ ***do***
      i. *Construct* A_i *and* B_i *by moving* c_i *from its superclass to the other*
      ii. $s_i \leftarrow$ *the split provided by the FDA for data D and classes* A_i *and* B_i
      iii. $IG_i \leftarrow IG(s_i, D)$
   d. $i^\star \leftarrow \arg\max_{i=1,\ldots,k} IG_i$
   e. ***if*** $IG_{i^\star} > IG_0$ ***then***
      $A \leftarrow A_{i^\star}$, $B \leftarrow B_{i^\star}$

   until $IG^\star \leq IG_0$ */* no improvement */*
3. ***return*** *A and B*

Both superclass generation methods require multiple runs of LDA, so they may be a significant additional cost, as they are run for each DT node, generated during the induction process.

2.3.6 Dipolar Criteria for DT Induction

Decision trees based on *dipolar criteria*, offered by Bobrowski and Krętowski (2000), are based on the same fundamental idea as the SSV criterion (see Sect. 2.2.7): to find splits that separate possibly many pairs of objects belonging to different classes. Bobrowski and Krętowski (2000) defined *dipoles* as pairs of objects and distinguished between *pure dipoles* (containing objects belonging to the same class) and *mixed dipoles* (containing objects from different classes).

To define the objective, dipolar criterion function, two penalty functions need to be defined first, for each input vector $\mathbf{v}$, to control whether it stays on the positive or

negative side of the candidate hyperplane:

$$\varphi_{\mathbf{v}}^{+}(\mathbf{w}) = \begin{cases} \delta - \mathbf{w}^T\mathbf{v} & \text{if } \mathbf{w}^T\mathbf{v} < \delta, \\ 0 & \text{if } \mathbf{w}^T\mathbf{v} \geq \delta, \end{cases} \tag{2.62}$$

$$\varphi_{\mathbf{v}}^{-}(\mathbf{w}) = \begin{cases} \delta + \mathbf{w}^T\mathbf{v} & \text{if } \mathbf{w}^T\mathbf{v} > -\delta, \\ 0 & \text{if } \mathbf{w}^T\mathbf{v} \leq -\delta, \end{cases} \tag{2.63}$$

where δ is a margin parameter (usually set to 1 by the authors).

Then, for each dipole, the functions can be combined to measure the cost related to dividing the dipole or not. For pure dipoles, the cost can be calculated as

$$\varphi_{\mathbf{u},\mathbf{v}}^{p}(\mathbf{w}) = \varphi_{\mathbf{u}}^{+}(\mathbf{w}) + \varphi_{\mathbf{v}}^{+}(\mathbf{w}) \quad \text{or} \quad \varphi_{\mathbf{u},\mathbf{v}}^{p}(\mathbf{w}) = \varphi_{\mathbf{u}}^{-}(\mathbf{w}) + \varphi_{\mathbf{v}}^{-}(\mathbf{w}), \tag{2.64}$$

while for mixed dipoles as:

$$\varphi_{\mathbf{u},\mathbf{v}}^{m}(\mathbf{w}) = \varphi_{\mathbf{u}}^{+}(\mathbf{w}) + \varphi_{\mathbf{v}}^{-}(\mathbf{w}) \quad \text{or} \quad \varphi_{\mathbf{u},\mathbf{v}}^{m}(\mathbf{w}) = \varphi_{\mathbf{u}}^{-}(\mathbf{w}) + \varphi_{\mathbf{v}}^{+}(\mathbf{w}). \tag{2.65}$$

Eventually, appropriately weighted costs of the dipoles compose the dipolar criterion to be optimized:

$$\Psi(\mathbf{w}) = \sum_{(\mathbf{u},\mathbf{v})\in I_p} \alpha_{\mathbf{u},\mathbf{v}} \varphi_{\mathbf{u},\mathbf{v}}^{p}(\mathbf{w}) + \sum_{(\mathbf{u},\mathbf{v})\in I_m} \alpha_{\mathbf{u},\mathbf{v}} \varphi_{\mathbf{u},\mathbf{v}}^{m}(\mathbf{w}). \tag{2.66}$$

Optimization of this kind of objective functions, can be performed with a Basis exchange algorithm (Bobrowski 1991, 2005) similar to the standard methods of Linear programming. An additional difficulty is the orientation of the dipoles, which makes only one of the two alternative forms of each penalty function (2.64) and (2.65) adequate in particular circumstances. For this purpose, the basis exchange algorithm has been equipped with proper search for adequate orientations of the dipoles (Bobrowski 1999, 2005).

Oblique decision trees stand a chance to preserve some comprehensibility if the linear combinations do not include large numbers of features. Therefore, the DT induction algorithm based on the dipolar criterion, implements some Feature selection functionality. In this aspect, the authors followed Brodley and Utgoff (1992a) and used their Heuristic sequential search of the LMDT (see Sect. 2.3.1). The procedure is the most time consuming part of the overall DT induction process.

A step toward cost decline may be early stopping of the splits. Bobrowski and Kretowski (2000) used a simple rule of stopping the splits, when the count of training data objects falling into the node was less than 5. Apart from that, they pruned the trees after construction, according to the principle of Reduced Error Pruning (see Sect. 2.4.3.1). They used 70 % of the training dataset for tree construction and afterwards, decided which nodes to prune by validating the tree on the dataset containing the remaining 30 % of data.

2.4 Generalization Capabilities of Decision Trees

There are many reasons, for which, decision trees generated on the basis of a training data sample are not perfect classifiers for the whole population of data objects. One of them is imperfectness of the training data sample. It often happens that the data objects descriptions are noisy. The noise may come from imprecise measurements, errors made during data collection, loosing some data and so on. On the other side, very often, the data sample is not representative of the problem and does not contain full information about the relations being learned. Sometimes, the information is contained in the data, but it is too difficult to discover it with the learning processes, because there are many local minima, which "conceal" the global minimum or the decision functions of the learning machines are not capable of describing the hidden dependencies. In all these cases and many others, the model resulting from DT induction may overfit the training data in the sense that it perfectly describes the sample, but not exactly the relations of interest.

When Overfitting occurs close to the root node, then the tree model is completely inaccurate and often, nothing can be done to fix it, but when it is due to splits close to the leaves, Pruning some tree branches can be a successful solution. Pruning makes the models simpler, so it is compliant with the idea of the Occam's razor. Fortunately, the root node and other nodes close to it, are usually created on the basis of large data samples, and because of that, generalize well. They should not be affected by pruning techniques, which ought to delete relatively small nodes, responsible for distinction between single data items or very small groups, existent in the training sample but being exceptions rather than representative objects of the population.

Another kind of problems can be observed when the number of features describing data objects is large in relation to the number of objects. Then, it is probable that the data contains features accidentally correlated with the output variable. Such features may be selected by the DT induction algorithms as the most informative ones and may significantly distort the model. In such cases, pruning is less helpful—a better way is to create many models and combine their decisions. In the Ensemble, the influence of accidentally correlated features is likely to be dominated by really informative ones. The price to pay for that is often model comprehensibility, but one can still search for some explanations by exploration of the ensemble members with respect to their compatibility with the ensemble decisions, and so on.

Some researchers, for example Bohanec and Bratko (1994) and Almuallim (1996), have used pruning techniques for tree simplification, which they describe as slightly different task than increasing generalization capabilities. They assume that the full tree is maximally accurate and search for simpler descriptions of the data, consciously accepting some loss in accuracy. So the task is to minimize the loss in accuracy for a given size constraint for the tree.

Most of the commonly used pruning techniques belong to one of two groups:

- *pre-pruning*: the methods acting within the process of DT construction, which can block splitting particular nodes,

- *post-pruning*: the methods that act after complete trees are built and prune them afterwards by removing the nodes estimated as not generalizing well.

Some other techniques, aimed at tree generalization, do not fit either of the groups. An interesting example is the strategy of Iterative refiltering used in DT-SE family of methods (see Sect. 2.3.4), where the final trees are results of multi-stage DT induction with adequate adjustment of the training data sample. Yet another (completely different) approach is the optimization of decision rules, but in fact, they are not proper DT pruning methods, as their final results may not have the form of DTs, even when started with the classification rule sets exactly corresponding to DTs.

Pre-pruning methods are also called *stop criteria*. The most natural condition to stop splitting is when no sensible further splits can be found or when the node is *clean* (*pure*), that is, contains objects belonging to only one class. Some generalizations of these ideas are the criteria of stopping when the node is small enough or contains small number of erroneously classified objects (reaching some purity threshold). They also stop further splitting, so are recognized as pre-pruning techniques.

Usually, it is very hard to estimate whether further splitting the node at hand may bring significant information or not, so apart from the simplest conditions like the ones mentioned above, pre-pruning techniques are not commonly used.

Post-pruning methods simplify trees by replacing subtrees by leaves in a previously constructed DT. Again, the group can be divided into two subgroups:

- *direct pruning* methods, that decide which nodes to prune just on the basis of the tree structure and information about training set distribution throughout the tree,
- *validation* methods, that use additional data sample (separate from the one used for training) in a validation process to determine which nodes do not seem to perform well on data unseen during learning.

Among the latter group, there are methods using single validation dataset (like Reduced Error Pruning) and others, performing multiple tests in a process like Cross-validation (for example, Cost-complexity optimization cost-complexity pruning of CART) to estimate optimal values of some parameters to be used in final tree pruning.

2.4.1 Stop Criteria

Like all recursive algorithms, DT induction methods must define a condition to stop further recursion, to avoid infinite loops.

One of the most natural criteria breaking the recursive splitting process is the condition of nodes purity. A clean node, that is, containing objects of one class only, does not need further splits, as it is 100 % correct (as far as the training data is concerned). Some softened purity conditions may also be defined. It is a popular approach to define the maximum allowable number of errors to be made by a tree node. When a node contains objects from one class and n objects from other classes it is not split when $n < \theta$, where θ is a pre-defined threshold. Another variant of

this idea is to define the threshold as the proportion of objects from classes other than the majority one, and calculate not just the number of errors, but the percentage of errors yielding different allowable error count for nodes of different sizes. Yet another similar methods put size constraints on each node and do not accept splits that generate a subnode smaller than a given threshold.

Another situation, when the recursive process is stopped in a natural way, is when the split criterion being used does not return any splits for the node. For example, when all the object descriptions are the same, but some data objects belong to one class and some to others. Such circumstances may occur when the dataset is noisy or when the features describing data are not sufficient to distinguish the classes.

Some split criteria can return no splits not only when all the data vectors are the same, but because of the setting of their parameters. It is common in the split methods based on statistical tests, that a split is acceptable only if a statistical test rejects the hypothesis of concern, with a specified level of confidence. For example, algorithms like Cal5 or CTree test some hypothesis to determine the split feature and the split of the node. On the basis of the tests they make decisions whether to make further splits or not. They may decide to not split the node even if it is not pure. Naturally, such statistical stop conditions could also be used with algorithms, where the split points are searched with exhaustive methods, but it is not common to do so. It is more popular to use statistical methods to prune the tree after it is fully grown, but such approaches perform post-pruning (the direct part) not pre-pruning.

The authors of DT algorithms usually prefer to build oversized trees and then prune them instead of using advanced stop criteria, because their wrong decisions may significantly affect the quality of the resulting trees. Comparative tests of pre-pruning and post-pruning algorithms (for example, the ones made by Mingers (1989a)) prove that the latter are more efficient, so most often, DT induction algorithms use only the most natural stop criteria. Sometimes, to avoid loosing time for building too big trees, small impurity thresholds are accepted. For example, when a node contains a single object of another class than that of all the others, it is not very reasonable to expect that further splits can generalize well, so ignoring such impurities, in most cases, just saves some time with no negative impact on resulting tree quality.

2.4.2 Direct Pruning Methods

Direct pruning is very time-efficient because it just examines the decision tree and training data distribution throughout the tree. On the other hand, they are provided with information about training data distribution, so they get less information than Validation methods, which are given also the results for some Unseen data. Hence, the task of direct pruning may be regarded as more difficult than the task of validation.

Methods for direct pruning usually estimate misclassification risk of each node and the whole subtree suspended at the node. If the predicted error rate for the node acting as a leaf is not greater than corresponding error for the subtree, than the subtree is replaced by a leaf. The differences between methods of this group are mainly in

the way of the misclassification rate estimation, as the natural estimation on the basis of the training data is overoptimistic and leads to oversized trees.

The following subsections describe some direct pruning methods. To provide a valuable review of this family of algorithms, several methods of diverse nature have been selected (including the most popular ones):

- PEP: *Pessimistic Error Pruning* (Quinlan 1987; Mingers 1989a; Esposito et al. 1997),
- EBP: *Error-Based Pruning* (Quinlan 1987; Esposito et al. 1997),
- MEP and MEP2: *Minimum Error Pruning* (Niblett and Bratko 1986; Mingers 1989a; Cestnik and Bratko 1991),
- MDLP– *Minimum Description Length Pruning* – different approaches have been proposed by Quinlan and Rivest (1989), Wallace and Patrick (1993), Mehta et al. (1995), Oliveira et al. (1996), Kononenko (1998); here only the approach of Kononenko (1998) is presented in detail.

2.4.2.1 Pessimistic Error Pruning

PEP Sometimes it is advantageous to approximate binomial distribution with normal distribution. To avoid some unfavorable consequences of the transfer from discrete values to continuous functions, an idea of *continuity correction* has been successfully used (Snedecor and Cochran 1989). Quinlan (1987) proposed to apply it to estimation of the real misclassification rates of DTs. Given a node N with the information about the number n_N of training data objects falling into it and the number e_N of errors (training data items belonging to classes different than the one with majority representation in N), Quinlan estimated the misclassification risk as

$$\frac{e_N + \frac{1}{2}}{n_N}. \tag{2.67}$$

The rate for the subtree T_N rooted at N can be defined as the weighted sum of the rates for all leaves in T_N ($\widetilde{T_N}$) with weights proportional to leaves sizes, which can be simplified to:

$$\frac{\sum_{L \in \widetilde{T_N}} e_L + \frac{1}{2}|\widetilde{T_N}|}{n_N}. \tag{2.68}$$

Because the continuity correction is often not satisfactory to eliminate overoptimistic estimates, the approach of Pessimistic Error Pruning uses it with the margin of one standard deviation of the error (SE for standard error) (Quinlan 1987; Mingers 1989a; Esposito et al. 1997). Denoting the corrected error of the subtree T_N, that is, the numerator of (2.68), by E_{T_N}, the estimation of the standard error can be noted as:

$$SE(E_{T_N}) = \sqrt{\frac{E_{T_N} \times (n_N - E_{T_N})}{n_N}}. \tag{2.69}$$

The algorithm of PEP replaces a subtree by a leave when the error estimated for the node (2.67) is not greater than the corrected error counted for the subtree (2.68) minus its standard error (2.69).

The procedure is applied in top-down manner, which usually eliminates a part of calculations, because if a node at high level is pruned, all the nodes of its subtree need not be examined.

2.4.2.2 Error-Based Pruning

Another way of pessimistic error evaluation gave rise to the Error-Based Pruning algorithm (Quinlan 1987) used in the popular C4.5 algorithm. Although it is often described as PEP augmented with possibility of grafting maximum child in place of the parent node, the difference is much larger—estimation of the pessimistic error is done in completely different way. Here, confidence intervals are calculated for given probability of misclassification and the upper limits of the error rates are compared (for given node as a leaf and the subtree).

From the source code of C4.5r8 it can be discovered that the Wilson's approach to confidence intervals (Wilson 1927) was applied. Probably, the reason for such a choice was that the Wilson's intervals offer good approximation even for small samples, which are very common in DT induction. Actually, pruning is required almost only for nodes with small data samples. Nodes with large samples usually allow for reliable splits and are not pruned.

Wilson defined the confidence interval at level α for a sample of size n, drawn from binomial distribution with probability p as:

$$\frac{p + \frac{z^2}{2n} \pm z\sqrt{\frac{p(1-p)}{n} + \frac{z^2}{4n^2}}}{1 + \frac{z^2}{n}}, \tag{2.70}$$

where $z = z_{1-\frac{\alpha}{2}}$ is the critical value of the Normal distribution for confidence level α.

To determine whether to prune a node N or to keep the subtree, or to graft maximum child node in place of N, one needs to calculate the upper limit of the confidence interval for misclassification rate of N as a leaf, the subtree T_N and for the maximum child of N. The decision depends on the fact, which of the three limits is the smallest.

To avoid comparing the quality of the node at hand with all possible results of pruning its subtree, one can compare just to the best possible shape of the subtree. However, to obtain the best pruning of the subtree before its root node pruning is considered, the process must be run from bottom to the root of the tree (as opposed to PEP, which is a top-down approach).

EBP has been commented as more pessimistic than PEP. Esposito et al. (1997) argument that although the estimation is more pessimistic, it is so for both the subtree and its root acting as a leave, which makes the method prune less than the PEP. As can be seen in the experiments described in Chap. 5, on average, EBP generates smaller

trees than PEP. The discussion there gives some explanation about the most probable reasons of so different conclusions.

The code of C4.5 contains additional tests, controlling the size of result trees. For example, it checks whether the node to be split contains at least 4 known values (more precisely 2·MINOBJS, where MINOBJS is a parameter with default value of 2) and whether the split would not introduce too small nodes (of size less than MINOBJS). When any of the tests reports the danger of too small DT nodes, the node is not split (gets closed, converted into a leaf).

Existence of missing values in the data is reflected in the method by weights assigned to each training data object. Initially each object gets the weight of 1 and in the case of uncertain splits (splits using feature values not available for the object) the object is passed down to all the subnodes, but with properly decreased weights (equal to the proportions of the training data without missing values, passed down to the subnodes). The numbers of objects in each node are calculated as the sums of weights assigned to objects instead of the crisp counts.

2.4.2.3 Minimum Error Pruning

MEPAs noticed by Niblett and Bratko (1986), misclassification probability estimates of DT leaves can be calculated according to the *Laplace's law of succession* (also called *Laplace correction*). In the case of a classification problem with k classes $c_1, \ldots, c_k$, the class probability distribution may be estimated by:

$$p(c_i) = \frac{n_{N,c_i} + 1}{n_N + k}. \tag{2.71}$$

Cestnik and Bratko (1991) proposed using a more general Bayesian method for estimating probabilities (Good 1965; Berger 1985). According to this method, called *m-probability-estimation*, the estimates (called *m-estimates*) are:

$$p(c_i) = \frac{n_{N,c_i} + p_a(c_i) \cdot m}{n_N + m}, \tag{2.72}$$

where $p_a(c_i)$ is a priori probability of class c_i and m is a parameter of the method.

It is easy to see that the Laplace correction is a special case of m-estimates, where $m = k$ and prior probabilities are all equal to $\frac{1}{k}$. The m parameter serves as a coefficient determining how much the raw training data estimations should be pushed towards a priori probabilities — with $m = 0$ the raw proportions are effective and with $m \to \infty$ the probabilities become the priors.

Given the probability estimation scheme (2.71) or (2.72), the decisions about pruning a node are made according to the results of comparison between the probability of misclassification by the node acting as a leaf, and by the subtree rooted in the node. Such pruning methods are referred to as *MEP* (*Minimum Error Pruning*) and *MEP2* respectively.

To sum up the differences between MEP and MEP2, it must be mentioned that:

- MEP assumes uniform initial distribution of classes, while MEP2 incorporates prior probabilities in error estimation.
- MEP is parameterless and the degree of MEP2 pruning can be controlled with m parameter.
- the m parameter of MEP2 can reduce the influence of the number of classes to the degree of pruning.

It is not obvious what value of m should be used in particular application. Cestnik and Bratko (1991) suggested using domain expert knowledge to define m on the basis of the level of noise in the domain data (the more noise the larger m) or performing validation on a single separate dataset or in the form similar to the one proposed by Breiman et al. (1984) and presented in more detail in Sect. 2.4.3.2. More discussion on validation of the m parameter is given in Sect. 3.2.4.3.

2.4.2.4 Minimum Description Length Pruning

Many different approaches to DT pruning based on the *Minimum Description Length* (MDL) principle have been proposed (Quinlan and Rivest 1989; Wallace and Patrick 1993; Mehta et al. 1995; Oliveira et al. 1996; Kononenko 1998). All of them share the idea that the best classification tree built for a given dataset is the one that offers minimum length of the description of class label assignment for training data objects. Such approaches deal with a trade-off between the size of the tree and the number of exceptions from tree decisions. A nice illustration of the problem is the analogy presented by Kononenko (1995) of the need to transmit data labels from a sender to a receiver with as short message as possible. Naturally, the smaller tree the shorter its encoding, but also the larger part of the training data is misclassified with the tree, so the exceptions require additional bits of code. In other words, if a tree leaf contains objects from one class, it can be very shortly described by the code of the class, and if there are objects from many classes in the leaf, than the class assignment for all the objects must be encoded, resulting in significantly longer description.

The MDL-based DT pruning algorithm implemented and tested in Chap. 5 is the one presented by Kononenko (1998), based on the ideas of using MDL for attribute selection, assuming that the best attribute is the most compressive one (Kononenko 1995).

Given a decision tree node N containing n_N training data objects (belonging to classes $c_1, \ldots, c_k$), the encoding length of the classification of all instances of N can be calculated as:

$$PriorMDL(N) = \log \binom{n_N}{n_{N,c_1}, \ldots, n_{N,c_k}} + \log \binom{n_N + k - 1}{k - 1}. \tag{2.73}$$

The first term represents the encoding length of classes of the n_N instances and the second term represents the encoding length of the class frequency distribution.

The value of $PriorMDL(N)$ suffices for the estimation of the description length of the classification in node N treated as a leaf. The description of the subtree T_N must include the description of the structure of the subtree and classification in all its leaves. After some simplifications, Kononenko (1998) proposed:

$$PostMDL(T_N) = \begin{cases} PriorMDL(N) & \text{if } N \text{ is a leaf} \\ 1 + \sum_{M \in Children(N)} PostMDL(M) & \text{if } N \text{ has children nodes,} \end{cases} \tag{2.74}$$

where $Children(N)$ is the set of children nodes of N.

Eventually, to decide whether to prune at node N or not, it suffices to compare the description lengths of the leaf N and the subtree T_N and prune when

$$PriorMDL(N) < PostMDL(T_N). \tag{2.75}$$

The condition must be tested in bottom-up manner for all nodes of the tree, similarly to EBP and MEP methods.

2.4.2.5 Depth Impurity Pruning

Another interesting approach to decision tree pruning was presented by Fournier and Crémilleux (2002). They defined a measure of DT quality to reflect both purity of DT leaves and DT structure.

An important part of the quality index is *Impurity Quality* of a node N in a tree T:

$$IQN_T(N) = (1 - \varphi(N))\beta^{depth_T(N)-1}, \tag{2.76}$$

where φ is an impurity measure normalized to [0, 1]. Since the node quality (2.76) reflects how deep the node occurs in the tree, *Depth Impurity* of the tree can be defined just as the weighted average of the quality values of all its leaves:

$$DI(T) = \sum_{N \in \tilde{T}} \frac{n_N}{n_{N_0}} IQN_T(N), \tag{2.77}$$

where N_0 is the root node of T.

The definition (2.77) refers to the whole final tree—the depth in the tree must always be calculated for the whole tree, which is often impractical, because when pruning a subtree, it is usually more reasonable to focus just on the subtree vs its root node. Hence, the DI index can be redefined in a recursive form as:

$$DI(T_N) = \begin{cases} 1 - \varphi(N) & \text{if depth of } T_N \text{ is 1,} \\ \beta \sum_{M \in Children(N)} \frac{n_M}{n_N} DI(T_M) & \text{otherwise.} \end{cases} \tag{2.78}$$

The method of *Depth Impurity Pruning* compares $DI(T_N)$ regarding the full subtree T_N and reduced to a single leaf N. It prunes the node if the DI index for the leaf N is lower than the one for the subtree rooted at N.

As usual, proper selection of the β parameter is a nontrivial task. There is no justification for regarding a single value as the best one, so it needs to be determined in special processes, for example in CV based analysis, as described in Sect. 3.2.4.3.

2.4.3 Validation Based Pruning

Validation is a stage of learning, where models being already results of some initial learning processes are adjusted with respect to the results obtained for a data sample different than the one used for the initial learning. It seems a very attractive way of improving DT generalization capabilities, so it has been applied by many authors.

Among numerous DT validation methods, we can distinguish the group of algorithms that perform only a single initial learning followed by a single validation pass and the group of those optimizing parameters in multistage processes.

A typical algorithm belonging to the former group is Reduced Error Pruning (REP), which adjusts the tree to provide minimum classification error for given validation data. The methods based on single training and validation have a disadvantage that the resulting DT model is built on a part of the whole training data (another part is used for validation, so can not be included in tree construction). Therefore, the methods from the latter group have more possibilities in providing accurate trees. Naturally, the methods capable of multi-pass validation are also eligible for single pass validation, but not inversely.

Several validation based pruning methods are presented in the following subsections. Apart from REP, all other methods presented below use Cross-validation to learn how to prune the final tree built on the whole training data.

2.4.3.1 Reduced Error Pruning

The most natural use of a validation dataset to adjust the tree trained on another set is to prune each node, if only it does not increase the classification error calculated for the validation data. Although the method is called *Reduced Error Pruning* (REP), it is advisable to prune nodes also when the error after pruning does not change. According to Occam's razor, a simpler model should be preferred, if it provides the same accuracy as a more complex one. The algorithm passes the validation dataset through the tree to determine numbers of errors made by each node, and analyzes all the splits in the tree, starting from those with leaves as subnodes, up to the root node, by comparing the numbers of errors of the node and the subtree rooted at the node. When the error count of the subtree is not lower than that of its root node, then it is replaced by a leaf.

A theoretical analysis of why the algorithm often fails to prune the tree, although large trees are not significantly more accurate than small ones, was conducted by Oates and Jensen (1999). As a result, they proposed to decide about pruning a node and its subnodes on the basis of different validation samples. It is easy to do so for artificial data, if a new sample can always be generated, but not often feasible in real applications, where data samples are limited, and sometimes so small that even extracting a single validation sample can significantly reduce learning gains.

2.4.3.2 Cost-Complexity Minimization

Accepting Occam's razor in the realm of DT induction implies care for as small trees as possible without decline of models accuracy. This leads to a typical trade-off condition, because pruning branches of trees built for given training data causes deterioration of reclassification scores. Breiman et al. (1984) proposed to control the trade-off with α parameter in a measure of DT misclassification cost involving tree size defined as the number $|\widetilde{T}|$ of leaves of the tree T:

$$R_\alpha(T) = R(T) + \alpha|\widetilde{T}|, \tag{2.79}$$

where $R(T)$ is a standard misclassification cost.

To determine the optimal value of α, Breiman et al. (1984) defined validation procedures which estimate performance of the candidate α values and select the best one. They have proven some important properties of the formula, which revealed that the pursuit of the optimal value of α can be efficient. First of all, they noticed the following property:

Property 2.1 For each value of α there is a unique smallest subtree $T_\alpha \prec T$ minimizing R_α.

It means that if any other subtree $T' \prec T$ also minimizes R_α, then $T_\alpha \prec T'$.

Further reasoning proved a theorem rephrased as the following property, which laid foundation for Cost-complexity optimizationcost-complexity validation methodology.

Property 2.2 There exist: a unique increasing sequence $\alpha_1, \ldots, \alpha_n$ and a decreasing sequence of trees $T_1 \succ \cdots \succ T_n$, such that $\alpha_1 = 0$, $|\widetilde{T}_n| = 1$ and for all $i = 1, \ldots, n$, T_i minimizes R_α for each $\alpha \in [\alpha_i, \alpha_{i+1})$ (to be precise, we need to define additional $\alpha_{n+1} = \infty$).

Less formally: there exists a unique decreasing sequence of αs that explores all possible trees optimizing R_α, and the trees are also precisely ordered. A natural result is that to determine all the αs, one can act sequentially, starting with the whole tree and determining nodes to be pruned, one by one (when it happens that pruning two or more nodes is conditioned by the same value of α, they must be pruned together). Formally, algorithm 2.12 presents the method. It is important from the point of view

of algorithm complexity, that after pruning a node only the nodes on the path to the root need an update of their αs—recalculating all of them would be a serious workload.

Provided the algorithm to determine the sequence of αs and corresponding smallest trees, the optimal α and the optimally pruned tree can be selected, for example, according to Algorithm 2.13.

Algorithm 2.12 (Determining αs and their optimal trees for cost-complexity minimization)

Prototype: ***CCAlphasAndTrees****(D,**Learner**)*
Input: *Training data D, DT induction method Learner.*
Output: *Sequences of αs and trees as in property 2.2.*
The algorithm:

1. $T \leftarrow Learner.LearnFrom(D)$—*induce full DT to be analyzed*
2. *Determine threshold α_N for each node N of T*
3. $i \leftarrow 1$
4. $\alpha \leftarrow 0$
5. ***repeat***
   a. *Prune all nodes N of T with $\alpha_N = \alpha$ (modifies T)*
   b. *Update all α_N for nodes N on the path from the root to just pruned node(s)*
   c. $T_i \leftarrow T$
   d. $\alpha_i \leftarrow \alpha$
   e. $\alpha \leftarrow \min_{N \in T} \alpha_N$
   f. $i \leftarrow i + 1$

   until $|\widetilde{T}| = 1$
6. ***return*** $\alpha_1, \ldots, \alpha_{i-1}$ *and* $T_1, \ldots, T_{i-1}$

The method assumes that two separate datasets are input: one for tree construction and one for validation. After a tree is built, its all sensible (according to cost-complexity minimization rule) pruned trees are determined and their accuracy estimated by classification of the validation data. The smallest tree with minimum classification error is selected as the validated pruned tree.

The situation is slightly more complex, when multiple validation is performed (for example Cross-validation). In each pass of the CV, a DT is built and analyzed to determine all the threshold αs and their smallest trees. Similarly, in the final pass, a DT is trained on the whole dataset and the two sequences are determined. Unfortunately, the series of αs in each pass of CV and in the final pass may be all different. To select a winner α all values of the final pass must be examined and average accuracy estimated for them. Since the sequence contains threshold αs, it is not the most sensible to check how they would behave in CV. In place of the border values α_i, Breiman et al. (1984) proposed using geometrical averages

$$\alpha'_i = \sqrt{\alpha_i \cdot \alpha_{i+1}}. \tag{2.80}$$

Algorithm 2.13 (Cost-complexity minimization with external validation data)

Prototype: ***CCTrnVal(*** D_{trn}***,*** D_{val}***,Learner)***
Input: *Training data* D_{trn}*, validation data* D_{val}*, DT induction method Learner.*
Output: *Optimally pruned DT, optimal* α*.*
The algorithm:

1. $((\alpha_i), (T_i)) \leftarrow CCAlphasAndTrees(D_{trn}, Learner)$
2. ***for*** $i = 1, \ldots, n$ ***do***
   $R_{val}(\alpha_i) \leftarrow$ *misclassification error of* T_i *measured for* D_{val}
3. $opt \leftarrow \arg\min_{i=1,\ldots,n} R_{val}(\alpha_i)$
4. ***return*** T_{opt} *and* α_{opt}

For each α'_i proper validation errors are extracted from CV and averaged. The largest α' with minimum average is the winner. Formal notation of the algorithm is presented as Algorithm 2.14. In line 4 of the algorithm, it should be specified what is meant by α_{j+1} for maximum j. Breiman et al. (1984) do not propose a solution, but since it is the matter of the largest α in the sequence, it is reasonable to assume $\alpha_{j+1} = \infty$, as further increasing α to infinity does not change the corresponding optimal tree—for all values of α in the range $[\alpha_j, \infty)$, the optimal tree is a stub with no split, classifying to the class with maximum prior probability. Such definition is advantageous, because in all the series, always the last $\alpha = \infty$ corresponds to maximally pruned tree, where the root acts as a leaf (sort of baseline, majority classifier).

Algorithm 2.14 (Cost-complexity minimization with CV-based validation)

Prototype: ***CCCV(****D****,Learner,****n****)***
Input: *Training data D, DT induction method Learner, number of CV folds n.*
Output: *Optimally pruned DT, optimal* α*.*
The algorithm:

1. *Prepare training-validation data splits:* $\left(D_1^t, D_1^v\right)$, ..., $\left(D_n^t, D_n^v\right)$
2. ***for*** $i = 1, \ldots, n$ ***do***
   $\left((\alpha_j^i), (T_j^i)\right) \leftarrow CCAlphasAndTrees(D_i^t, Learner)$
3. $\left((\alpha_j), (T_j)\right) \leftarrow CCAlphasAndTrees(D, Learner)$
4. ***for each*** α_j ***do***
   $R_{CV}(\alpha_j) \leftarrow \frac{1}{n} \sum_{i=1}^{n} R_{val}\left(\sqrt{\alpha_j \cdot \alpha_{j+1}}\right)$
5. $opt \leftarrow \arg\min_j R_{CV}(\alpha_j)$
6. ***return*** T_{opt} *and* α_{opt}

2.4.3.3 Degree-Based Tree Validation

The idea of the *degree of pruning* applied in SSV DT (Grąbczewski and Duch 1999, 2000) is based on counting differences between reclassification errors of a node and its descendants. Pruning with given degree (which is an integer) means pruning the splits, for which the difference is not greater than the degree. The rationale behind the definition is that the degree defines the level of details in DT and that for decision trees trained on similar data (in CV, large parts of training data are the same), optimal pruning should require similar level of details. The level of details can be described by the number of leaves in the tree, but then, an additional method is needed for deciding which nodes should be pruned and which should be left. The definition of degree of pruning clearly determines the order in which nodes are pruned. Properties analogous to 1 and 2 are in this case trivial, so are not reformulated here. To analyze all possible degrees of pruning, tree nodes are pruned one by one in the order of increasing differences between node reclassification error and the sum of errors of node children. Such value is immutable for each node, so once determined it does not need recalculation. Therefore the algorithm collecting pairs of degrees and optimal trees is slightly simpler than the corresponding algorithm for Cost-complexity optimizationcost-complexity minimization.

Algorithm 2.15 (Determining degrees of pruning and trees pruned to degrees)

Prototype: ***DegreesAndTrees****(D,****Learner****)*
Input: *Training data D, DT induction method Learner.*
Output: *Sequences of degrees and pruned trees.*
The algorithm:

1. $T \leftarrow Learner.LearnFrom(D)$—*induce full DT to be analyzed*
2. *Determine the degrees of pruning* d_N *for each node* N *of* T
3. $i \leftarrow 1$
4. ***for each*** *degree* d *determined above, in increasing order* ***do***
   a. ***for each*** *node* N *of* T*, with all subnodes being leaves* ***do***
      if $d_N <= d$ *then prune node* N *(change into a leaf)*
   b. $d_i \leftarrow d$
   c. $T_i \leftarrow T$
   d. $i \leftarrow i + 1$
5. ***return*** $(d_1, \ldots, d_{i-1})$ *and* $(T_1, \ldots, T_{i-1})$

Also the main algorithms performing validation and selecting the best pruned trees get simpler. To save space only the one based on CV is presented (Algorithm 2.16). In the case of degrees, no geometrical averages are applied. Average risk is calculated directly from CV tests, where the errors for particular pruning degrees can be easily read out.

Instead of direct optimization of the pruning degree, one can optimize tree size (defined as the number of leaves) and use the concept of pruning degrees, just to

Algorithm 2.16 (Degree-based DT validation based on CV)

Prototype: ***DegCV(D,Learner,n)***
Input: *Training data D, DT induction method Learner, number of CV folds n.*
Output: *Optimally pruned DT, optimal degree d.*
The algorithm:

1. *Prepare training-validation data splits:* $\left(D_1^t, D_1^v\right)$, ..., $\left(D_n^t, D_n^v\right)$
2. ***for*** $i = 1, \ldots, n$ ***do***
   $\left((d_j^i), (T_j^i)\right) \leftarrow DegreesAndTrees(D_i^t, Learner)$
3. $\left((d_j), (T_j)\right) \leftarrow DegreesAndTrees(D, Learner)$
4. *For each* d_j*:*
   $R_{CV}(d_j) \leftarrow \frac{1}{n} \sum_{i=1}^{n} R_{val}(d_j)$
5. $opt \leftarrow \arg\min_j R_{CV}(d_j)$
6. ***return*** T_{opt} *and* d_{opt}

determine the order in which the nodes are pruned. To obtain such methods, it suffices to modify Algorithms 2.15 and 2.16 to handle sequences of leaves counts in place of the sequences of degrees.

This method is much simpler, easier to implement and a bit faster than the cost-complexity optimization of Breiman et al. (1984).

2.4.3.4 Optimal DT Pruning

Bohanec and Bratko (1994) proposed the OPT algorithm for construction of the *optimal pruning sequence* for given DT. By means of Dynamic programming they determined an optimal subtree (maximizing training data reclassification accuracy) for each potential tree size. Then, depending on which final accuracy they were interested in, they pruned the tree to the appropriate size. Algorithm 2.17 presents the main procedure of the method. It generates sequences of error counts (for the training data) and pruned nodes collections for integer arguments denoting the decrease in the number of leaves, corresponding to subsequent sequence positions. The main procedure recursively gets the sequences for each child of the node being examined, and combines the results with the *Combine()* method presented as Algorithm 2.18, in a way compatible with the paradigm of Dynamic programming.

To be precise, there is a little difference between this formulation of the OPT algorithm and the original one: Bohanec and Bratko (1994) did not record the error counts in one of the result sequences (here named **E**), but the difference in error count between the node acting as a leaf and the whole subtree rooted at the node. The difference is null when the full tree is 100 % accurate. It does not matter either, when only one tree is analyzed. Some differences in decisions can appear when multiple validation is performed (for example Cross-validation) to average the scores and draw

Algorithm 2.17 (OPTimal DT pruning sequence)

Prototype: OPT(N)

Input: *Tree node* N *(let* $n = |\widetilde{T^N}|$*).*

Output: *Sequences of reclassification errors* $\mathbf{E} = (E[i])$ *and collections of nodes to be pruned* $\mathbf{P} = (P[i])$ *(for each sensible leaves reduction* $i = 0, \ldots, n-1$*).*

The algorithm:

1. $P[0] \leftarrow \emptyset$
2. $E[0] \leftarrow 0$
3. ***for*** $i = 1, \ldots, n-1$ ***do***
   $E[i] \leftarrow -1$
4. ***for each*** *subnode* M *of* N ***do***
   a. $(\mathbf{E}_M, \mathbf{P}_M) \leftarrow OPT(M)$
   b. $(\mathbf{E}, \mathbf{P}) \leftarrow Combine(\mathbf{E}, \mathbf{P}, \mathbf{E}_M, \mathbf{P}_M)$
5. ***if*** N *has subnodes* ***then***
   $P[n-1] \leftarrow \{N\}$
6. $E[n-1] \leftarrow e_N$ */* e_N is the number of errors made by N for the training data */*
7. ***return*** E *and* P

Algorithm 2.18 (Combining OPT sequences)

Prototype: Combine$(\mathbf{E}_1, \mathbf{P}_1, \mathbf{E}_2, \mathbf{P}_2)$

Input: *Sequences of error counts and collections of nodes to be pruned* $(\mathbf{E}_1, \mathbf{P}_1, \mathbf{E}_2, \mathbf{P}_2$*, indexed by* $0, \ldots, n_1$ *and* $0, \ldots, n_2$ *respectively).*

Output: *Combined sequence of reclassification errors* $\mathbf{E}$ *and collections of nodes to be pruned* $\mathbf{P}$*.*

The algorithm:

1. $P[0] \leftarrow \emptyset$
2. $E[0] \leftarrow 0$
3. ***for*** $i = 1, \ldots, n-1$ ***do***
   $E[i] \leftarrow -1$
4. ***for*** $i = 0, \ldots, n_1$ ***do if*** $E_1[i] \geq 0$ ***then***
   for $i = 0, \ldots, n_2$ ***do if*** $E_2[i] \geq 0$ ***then***
   a. $k \leftarrow i + j$
   b. $e \leftarrow E_1[i] + E_2[j]$
   c. ***if*** $E[k] < 0$ *or* $e < E[k]$ ***then***
      i. $P[k] \leftarrow P_1[i] \cup P_2[j]$
      ii. $E[k] = e$
5. ***return*** E *and* P

some conclusions from the means, but the probability of differences in decisions is not high.

For binary trees, the sequences are full (no -1 value is left in the **E** sequence, because it is possible to get any tree size with pruning. When the pruned tree contains multi-way splits, it may be impossible to get some sizes, so in the final **E** sequence, some elements may stay equal to -1 after the optimization.

The original algorithm was justified from another point of view than improving classification of unseen data (generalization). Instead, the authors assumed that the full tree was maximally accurate and searched for the smallest tree preserving given level of accuracy. Although the motivation for DT validation is different, the same Dynamic programming scheme can be used to determine expected accuracy of trees of each size, inside a Cross-validation. The size maximizing expected accuracy becomes the size of the target tree obtained with optimal pruning of the tree generated for the whole training data.

The idea is quite similar to that of the degree-based pruning, however it is much more detailed, in the sense that it analyzes all possible tree sizes, while in the former algorithm only those resulting from pruning to a given degree (two subsequent trees may have sizes differing by quite large number, because increasing the pruning degree from 1 to 2 may prune many nodes in the tree). Naturally, the cost paid for the increased level of details is an increase of computational complexity of the method—the dynamic programming optimizations cost much more, so in the case of large trees the time of learning may get significantly larger. Almuallim (1996) offered some improvements in the calculations, which under some assumptions reduces the computational costs, but they are not always helpful. They can be applied when we are interested in a particular tree size, but for example, in the tests described in Chap. 5 where many different tree sizes are examined, the simplifications proposed by Almuallim (1996) are not applicable.

2.5 Search Methods for Decision Tree Induction

Search methods and tree structures are inextricably linked with each other. Traces of search procedures have the form of trees, and inversely: tree construction is naturally obtained with search methods. Therefore, also in the area of decision trees, search methods serve as fundamental tools of induction. The simplest and usually the fastest search technique is the greedy one, which at each stage preforms a local minimization to determine the next state and never returns to the previous stages in order to try alternative solutions. Apart from the term *greedy search* such technique can be called many other names. Because of no returns and the goal of class separation (each split may improve the separation and never deteriorate it), the techniques of *depth first search*, *best first search*, *hill climbing* and some others are all equivalent in the sense that they end up with the same tree (just the order of nodes creation may be different, but it does not cause differences in the final models). As a result, they all can be seen as the top-down greedy DT induction method formalized as Algorithm 2.1. Such

technique is used in vast majority of DT induction approaches, but is not the only one applicable to the problem. Some applications of Beam search and Lookahead search, described below, have also been examined.

General Search Aspects

Regardless of the search method used for DT induction it is always important to avoid unnecessary calculations. When a node split is regarded, a set of split candidates must be analyzed and the best one selected. The list of candidates must be determined reasonably, because many of the seeming candidates can be judged in advance as certainly worse than some others and ignored with no change to the final tree.

When the optimal (with respect to a given criterion) binary split of a continuous feature is to be found, it is natural that the sensible split points are only those lying between the values observed in the training data. All split points lying between the same two adjacent values observed in the training data bring the same split result, so in practice, only the points in the middle of the interval are taken into account. Moreover, split quality measures usually have a form of Concave functions or have other properties that justify ignoring the split points lying between objects belonging to the same class, because separating them gives lower scores than putting the neighbors together to one or the other side of the split. Proofs of such properties, with special attention paid to particular methods, have been published for example by Breiman et al. (1984, Gini index), Fayyad and Irani (1992b, Information gain) Grąbczewski (2003, SSV criterion).

In the case of unordered features, the splits are not determined by a point (a real value) but by division of the sets of symbols into disjoint and complementary subsets. A subnode is created for each subset, and the training data objects are distributed to the subnodes, according to the symbols describing them. Some algorithms (for example C4.5) consider only singletons and split into as many subnodes as the number of possible symbols of the split feature. In such approach there is just one way to split on the basis of a symbolic feature. The most serious drawbacks of such solutions are that they can split the training data into many small datasets without significant reason, and that split quality measures may give overoptimistic estimates of the symbolic splits, because of accidental correlation between the (numerous) symbols and assigned classes. Therefore, often, binary splits are preferred also for symbolic features. But the number of possible splits may get huge if the number of possible symbols is large. To check all possible splits one needs to examine $2^{s-1}-1$ candidates, where s is the number of possible symbols of the feature. The number comes from the count of all subsets of an s-element set (2^s) and the fact that neither the empty set nor full set of symbols can be accepted to determine a subnode, and that a subset and its complement determine the same split, so only one of them should be used: $(2^s-2)/2 = 2^{s-1}-1$. When the number s is large, analyzing each possible split may be unfeasible and additional restrictions must be applied to reduce the amount of calculations. A solution based on a subset generator was described in Sect. 2.2.7 on the SSV algorithm.

Another aspect of splitting on the basis of single features is bias in feature selection. When two features are equally informative, they should have the same probability of being selected as the split feature. If the number of analyzed splits is significantly different for ordered and unordered features, then a bias can be observed in favor of one of the types. More thorough discussion of the bias in feature selection for DT induction is presented in Sect. 2.7.

Lookahead Search

One of the possibilities of more thorough search is to use the methods called *lookahead search*. As suggested by the name, such algorithms grow subsequent DT branches not on the basis of direct split quality measurements, but with some forward insight to the potential gains of using particular splits. After hypothetical acceptance of a split, the resulting subnodes are further split, in order to check the quality of the best possible depth-restricted subtree rooted at the node (the depth is given by a parameter). The search with depth n is also called an *n-ply lookahead search*. The algorithms resemble mini-max search used in game playing, but here, the decisions at subsequent levels are made by cooperating parties, not by adversaries (with the same, not competitive, quality measures).

Algorithm 2.19 (Lookahead split selection)

Prototype: ***LookaheadBestSplit(depth,D)***
Input: *Lookahead depth, training dataset D.*
Output: *The split estimated as the best.*
The algorithm:

1. ***for each*** $s \in CandidateSplits(D)$ ***do***
   $N_s \leftarrow SplitAhead(depth, D, s)$
2. $best \leftarrow \arg\max_s Quality(T^{N_s})$
3. ***return*** $best$

Formally, we can see the lookahead search algorithms as ordinary top-down DT induction methods (Algorithm 2.1) with the procedure of best split selection based on some forward insight. So in fact, it is not a new search method, but a special split selection method, although its intuitive perception may be different. A natural instance of such lookahead split selection is presented as Algorithm 2.19. It builds a branch of a pre-defined depth for each candidate split (with the *SplitAhead()* method presented as Algorithm 2.20) and estimates the quality of the split hierarchy. The split providing the highest quality of the generated branch is returned as the result of the lookahead split selection. It is important to realize that the quality measure used in such procedure must be ready for assessment of tree structures, not just single splits. However, each Split quality measure designed to estimate multipart splits is

Algorithm 2.20 (Depth-limited splits)

Prototype: ***SplitAhead(depth, D, s)***
Input: *Lookahead depth, training dataset D, initial split s.*
Output: *The root node of the created tree.*
The algorithm:

1. ***if*** $s \neq \perp$ *and* $depth > 0$ ***then***
   a. $\{D_1, \ldots, D_n\} \leftarrow s(D)$ */* split the node data */*
   b. ***for*** $i = 1, \ldots, n$ ***do***
      i. $best \leftarrow BestSplit(D_i)$
      ii. $N_i \leftarrow SplitAhead(depth - 1, D_i, best)$
   c. $Children \leftarrow (N_1, \ldots, N_n)$
   else
   $Children \leftarrow \perp$
2. ***return*** $(D, s, Children)$

naturally applicable, because each tree can be treated as a single split into the parts corresponding to its leaves.

The lookahead split selection can be called a meta-level split procedure, as it uses external method of node split selection (*BestSplit()*) to build small trees and estimate their quality. Also the methods *CandidateSplits()* and *Quality()* can be arbitrarily chosen to obtain different effects. In fact, the three functions mentioned above are the parameters of *LookaheadBestSplit()*, but are not listed explicitly in the parameter list to keep the code clearer.

The number of splits to be made in a lookahead estimation, grows exponentially with the *depth* parameter, so to avoid large computational overhead, it is not recommended to look deeper than just one level (*depth=1*).

Beam Search

Greedy selection of the best split at each node would be the only sensible technique, if the split quality estimation were perfect. As discussed in the beginning of Chap. 2, optimization of a quality measure at a single node is not the same as optimization of the overall tree, so sometimes it may be more adequate to select a split of lower local quality, but providing better data environment for further branch splits, and in effect, bringing shorter or more accurate trees.

A tool facilitating a number of top-ranked partial solutions to take part in further pursuit of maximum overall quality is the beam search—the process conducted in almost the same way as the breadth-first search, but with a limit on the number of states that can be explored at each level, to prevent combinatorial explosion. In decision tree induction, the beam is a container for a number of top-ranked partial trees, which are developed in parallel. So the main difference from the standard greedy approach is that at each stage of the search, the focus is not only on a single

tree but on all models contained within the beam. The size of the beam container (*beam width*) is a parameter of the method.

In general, the algorithm operates on so called states (here a state is a tree developed so far) and iteratively generates new states from the current ones, with the restriction that only the best ones are placed in the beam and further explored. Algorithm 2.21 presents the scheme formally. It is a general, abstract procedure capable of handling any kind of states, not just decision trees. At the beginning there is a single initial state in the beam. In each iteration, all possible children states of the trees contained in the beam are generated, and the best w of them (the beam width parameter) are placed in the beam. Since the goal is a single final tree, the search is stopped when a final state (a complete tree) is found.

Algorithm 2.21 (Beam search)

Prototype: ***BeamSearch****(S, w)*
Input: *Initial search state S, beam width w.*
Output: *Final search state.*
The algorithm:

1. $beam \leftarrow \{S\}$
2. ***while*** *NoFinalState(beam)* ***do***
   a. $children \leftarrow \emptyset$
   b. ***for each*** $state \in beam$ ***do***
      $children \leftarrow children \cup$ ChildrenStates$(state)$
   c. $beam \leftarrow$ BestStates$(w, children)$
3. ***return*** *the final state in beam*

Apart from the explicit parameters of the code, the functions called within the main procedure also significantly influence the process and its results. The most important of the subroutines is *BestStates()* which selects the best trees to be put into the beam of width w. It may compare the children states (here the trees) according to many different criteria like model accuracy, purity of decision tree leaves, information measures, MDL criteria and so on. The fundamental difference between application of these measures and of those used for single split quality assessment is that they need to compare the whole trees, not single splits. In particular, similarly to the measures used in the lookahead approach, they have to be applicable to multi-way splits, because nontrivial trees may split the space into large number of parts. The other subroutines (*NoFinalState()* and *ChildrenStates()*) can also be implemented in various ways, but their goals are rather straightforward, when the aim of the main search strategy is precisely defined.

The practice of filling the beam with the best states shows that trivial selection of the models maximizing criteria like accuracy, amount of information and others is not the most successful choice. When many children states are generated for each state in the beam, it often happens that after two or three iterations, all trees in the beam are very similar. For example, after the first iteration, the beam contains w single split

trees with different features used for the split, but after the second iteration, the beam contains w children of the same single split tree. As a result, the time consumption is significantly larger, but the time is wasted for exploration of almost the same areas of the model space. A remedy for this is nontrivial beam selection equipped with tools protecting from appropriation of the whole beam by a close family of trees. The tools may be some measures of diversity, for example, the ones proposed in the field of data clustering (Hubert and Arabie 1985; Vinh et al. 2010), as the feature space partition given by a tree ideally fits the assumption of clustering and similarity between trees may be determined by means of similarity between the space partitions they determine.

Broader search means larger complexity of the procedure, so in comparison to simple top-down greedy DT induction, beam search is naturally more expensive. The complexity grows by the factor of w (beam width), because in each iteration, w trees are examined in place of a single one of the greedy approach. Obviously, one can give examples, where beam search ends in a shorter time than the greedy search and inversely, where the relative cost factor of beam search is much greater than w. The former situation may occur when the most attractive split at the root of the tree requires complicated splits in subsequent levels, while some less attractive single split perfectly matches some other splits and yields small and accurate tree. The opposite relation can be observed when beam search rejects the direct solution of the greedy search, because of finding other seemingly more attractive solutions, which turn out to be a blind alley.

Beam search has been found attractive not only in searching for single DTs. Grąbczewski and Duch (2002a, 2002b) have found it useful for generation of a number of trees to act in ensembles (Decision forests). Small modifications of Algorithm 2.21 to stop the search after a specified number of final states (not just the first one) is found, gives an opportunity to build forests without much additional computational effort.

Restrictions on the beam width can be helpful, when combined with the lookahead search described above (Buntine 1993). It can reduce the increase of computation time requirements when the lookahead depth is greater than 1.

The Danger of Oversearching

Some authors have reported that using more thorough search procedures (than the most common, simple hill climbing) is often assisted by a decline in models accuracy (Murthy and Salzberg 1995; Quinlan and Cameron-Jones 1995; Segal 1996; Janssen and Fürnkranz 2009). Their observations mostly concerned classification rules induction, not decision tree algorithms, however the tasks are so similar, that they should be subject to the same effects, if the conclusions are derived from general, reliably prepared experiments.

Murthy and Salzberg (1995) compared greedy DT induction algorithm with a lookahead approach and concluded that both methods are similarly accurate on average and pointed out the pathology, that the lookahead approach can generate larger and less accurate trees than the greedy strategy. On average, the trees obtained with

lookahead turned out to be shallower, although in many cases it was just the matter of balance, lacking in the results of the greedy search. The authors proposed a procedure of balancing the trees based on rotations (without changes to the decision function). Another conclusion was that pruning can provide better results than lookahead, because the trees pruned with the cost-complexity optimizationcost-complexity minimization method (see Sect. 2.4.3.2) to prune the trees on the basis of a validation dataset consisting of 10 % of the training data excluded before tree construction. The observation is reasonable, but should not be treated as an argument against more thorough search procedures, as search and pruning should be perceived rather as two complementary, not alternative techniques. It seems that larger test errors of the lookahead method were not a consequence of oversearching, but the effect of pruning the trees coming from the competing algorithm.

Quinlan and Cameron-Jones (1995) performed some test of beam search applied to a classification-rules-induction method, where model generalization was controlled by Laplace error correction (see Eq. (2.71). They presented some experiments where the rules minimizing the Laplace error estimates turned out to be the less accurate the larger beam width was used (in the range 1–512 with exponential growth). As a proposed solution they introduced the *layered search* method which can be seen as beam search with changing beam width. They started the search with beam width equal to 1 (corresponding to hill climbing) and then, in each iteration doubled it. Such strategy gave more attractive results than both hill climbing and beam search.

The experiments of Quinlan and Cameron-Jones (1995) were repeated and analyzed by Segal (1996), who noticed that the Laplace error estimation of single rule is not much correlated with the true error rate calculated on external test dataset. The "oversearching effect" was a result of poor match between the evaluation function used for training and the test performance measure. Low accuracies were the result of ordinary overfitting. Segal (1996) noticed that Laplace error estimation trades off accuracy for coverage, which is detected by accuracy tests on external data. They suggested a modification of the Laplace correction, named *LaplaceDepth*, calculated for each rule R, and defined as

$$LaplaceDepth(R) = 1 - \frac{n_c + d \cdot p_a(c)}{n + d} \tag{2.81}$$

where c is the decision class of the rule R, $p_a(c)$ is the prior probability of class c, n is the number of examples covered by R, n_c is the number of objects from class c satisfying the rule and d is a parameter. Such evaluation criterion is compatible with the m-probability-estimation (see Eq. (2.72)). Here the m parameter is named d and, in the proposed strategy, should be equal to the length of the rule. The modification of the rule evaluation criterion practically removed the effect of "oversearching". Segal (1996) suggested that other measures of rule quality may do even better than the improved Laplace error.

More recently, Janssen and Fürnkranz (2008, 2009) revisited the problem and, also seeking the causes of the conclusions about oversearching in Laplace error estimation, examined nine different heuristics (including Laplace corrected error) in

experiments with three search strategies applied to the task of decision rule induction. The search methods they examined were: hill climbing, beam search and ordered exhaustive search. The latter is an implementation of exhaustive search, optimized to avoid repeated generation of the same rules. Significant simplification was possible because of the specificity of the definition of the rule induction problem, where the form of rules was limited, for example, in such a way that two premises could not use the same feature, so the features could be analyzed in a strictly defined order. From illustrations in the article, one can infer that the beam search also had a specific form. One of the figures and some text suggest that each new beam is created in a way supporting one child of each state from the previous beam. This is compatible with the heuristics aimed at beam diversity, described above in the description of the beam search approaches.

The experiments of Janssen and Fürnkranz (2009) also show that when the error estimation with Laplace correction is used, the test accuracy tends to deteriorate with increasing beam width. Nevertheless, for other measures, the results are completely different. Evaluating the rules by their precision gives stable accuracy plots, almost constant in the whole range of beam width, while the rules get simpler with larger beam widths. An example utterly opposite to Laplace error is the *odds ratio* for which a significant increase of accuracy was observed with increasing beam width.

To sum up, the conclusions from all the experiments performed to research the aspects of oversearching in DT and rule induction are not consistent. Sometimes, more exhaustive search improves the results and sometimes deteriorates them. Dependence on rule quality measures has been reported, but the notifications require more evidence. The only certain conclusion is that more advanced search procedures should be used with special care, but should not be forgotten. Future meta-learning algorithms will certainly help make decisions also about search method selection.

2.6 Decision Making with Tree Structures

Regardless of the strategy of decision tree induction, a method for final decision making must be selected. The most common approaches are based on the distribution of data objects of different classes falling into proper DT leaves. When a new data object is to be classified with a DT, it traverses the tree to discover the most adequate leaf. Most often, the object is assigned the label of the class dominating within the leaf. If the classification decision is to be presented in the form of probabilities of belonging to particular classes, they are usually estimated on the basis of that leaf (more precisely the training data falling into the leaf), and sometimes on the basis of all the nodes on the path to the leaf.

Probabilities as Proportions

The simplest method to obtain probability estimates is to calculate the proportions of objects within the leaf belonging to particular class and all the objects in the leaf:

$$P(c|x) = \frac{n_{L(x),c}}{n_{L(x)}}, \tag{2.82}$$

where $L(x)$ is the leaf adequate for x. The same notation is used in subsequent formulae.

Laplace Correction

When the counts of objects within the leaf are small, the proportions do not estimate real probabilities accurately. For example, when just three or four objects (all of the same class) fall into a leaf, proportion-based probabilities are binary, claiming that the probability of finding a representative of any other class in the area is zero. Usually, such extreme claims are not true. Therefore, some corrections have been proposed and successfully used in many applications. One of the methods, called *Laplace correction* (or *Laplace's law of succession*), has already been introduced in the context of Minimum Error Pruning (see Sect. 2.4.2.3) and is calculated as:

$$P(c|x) = \frac{n_{L(x),c} + 1}{n_{L(x)} + k}, \tag{2.83}$$

where k in the number of classes in the classification problem.

m-Probability-Estimation

Another way of probability correction was proposed by Cestnik and Bratko (1991) and called *m-probability-estimation*. It has also been introduced in the section on Minimum Error Pruning. The corrected probabilities, referred to as *m-estimates*, are defined by:

$$P(c|x) = \frac{n_{L(x),c} + m \cdot p_a(c)}{n_{L(x)} + m}, \tag{2.84}$$

where $p_a(c)$ is the a priori probability of class c and m is the method parameter. When the a priori probabilities of all the classes are equal, by setting m to the number of classes, we obtain the formula equivalent to the Laplace correction. In general, the parameter m defines the "strength of pushing" the proportions towards a priori probabilities. In a couple of applications, the authors of the idea proposed using $m = 2$. It is important to realize that this method may result in different winner class than the one resulting from pure proportions or those with Laplace correction—when $m \to \infty$, the probabilities converge to the priors, so the winning class may be different than the one dominating the data sample of the leaf. Laplace version of the estimation is compatible with pure proportions when just the winner is to be determined, but of course, the probabilities have different values, so for example, combined with others in an ensemble, may also bring significantly different results.

Path Combined Majority

In LTree family of algorithms Gama (Gama 1997), it is proposed that the decision making strategy respects class distributions in the nodes of the whole path of the tree, followed by the data item to be classified. Gama (1997) suggested to calculate probabilities for each tree node, starting with the root node, down to the leaves, in such a way that only the root node probabilities are estimated as proportions and for each non-root node N, its proportions and the probabilities calculated for the parent node contribute to the final probability estimates:

$$P(c|N) = \frac{P(c|Parent(N)) + w\frac{n_c}{n_N}}{1+w}, \tag{2.85}$$

where w is a method parameter. With $w = 1$ (suggested by the author), if the path from the root node to the leaf $N_L = L(x)$ is $N_0, \ldots, N_L$, then the probability

$$P(c|x) = P(c|N_L) = \sum_{i=0}^{L} \frac{1}{2^{L-i}} \frac{n_{N_i,c}}{n_{N_i}}. \tag{2.86}$$

Similarly to the m-probability-estimation approach, including class proportions of parent nodes may significantly change the decisions in relation to the ones calculated on the basis of the leaf only. Unlike the m-estimates, in this approach, the probabilities are not pushed towards the prior probabilities, but towards probability distributions in larger groups of objects (parent nodes). Although the contribution of the root node plays similar role as m-estimates, because the proportions in the root (that is, in the whole training data sample) are exactly the same as the estimates of priors in m-probability-estimation, the influence of the root node on the final values is very small for longer paths, as its factor is $\frac{1}{2^L}$.

More sophisticated combinations of path nodes proportions were proposed by Buntine (1993) and Kohavi and Kunz (1997) as part of their Option Decision Trees approach. In a Bayesian analysis, they define weights for ensembles of classifiers extracted from trees. The weights can combine decisions, not only of the options (different trees), but also of the nodes of a single path. Indeed, the nodes on a path from the root to a leaf can be treated as separate classifiers combined into an ensemble. Some more information about the approach can be found in Sect. 2.8.1.

2.7 Unbiased Feature Selection

One of the aspects of decision trees interpretation is feature relevance. When splits are performed on the basis of single features, it is reasonable to expect that the feature occurring in a tree node is, in some sense, the most informative one in the context of class discrimination among the data sample of the node. However, the term “the

most informative" is not precise, so there is not a unique measure of such feature relevance. Otherwise, there would exist a single, the most appropriate way to split DT nodes, and one method would be sufficient for all purposes. Therefore, so many different approaches to DT induction have been undertaken, and still none has been announced to outperform all the others.

Despite the fact, that no universal measure exists to estimate feature relevance, many researchers have been struggling for solutions of the problem of *unbiased feature selection* for DT splits. The goal of their research has been to provide methods of DT induction, that would not favor features because of their accidental relationships with the target variable.

So fair selection of split variable would provide very valuable information about the most discriminative features of the data table at hand in the context of particular task. DT techniques have been successfully used to extract information about feature importance for the purpose of feature ranking and selection (Duch et al. 2002, 2003; Grąbczewski 2004; Grąbczewski and Jankowski 2005), but still, conclusions about feature significance on the basis of feature selection made by a DT induction algorithm should be drawn with special care, especially when strong interaction between features can be observed.

The problem of variable selection bias has been observed from the early stages of research on DT induction. In AID (see Sect. 2.9 and Morgan and Sonquist 1963a,b), performing binary splits by means of dividing the set of possible feature symbols into two disjoint subsets, more categories of a feature means more split possibilities, resulting in a bias in favor of the features with many symbols. Kass (1980) proposed CHAID as a modification of the method to perform significance testing, so as to nullify the bias.

Similarly, in the efforts of DT induction based on miscellaneous heuristics instead of statistical tests, such as the one of the pioneer method, ID3, symbolic features with many possible values are favored, because the more symbols, the larger probability of coincidental correlation between the feature and the target variable. An extreme case is a feature with so many values that no value is repeated in the training data. Then, a perfect split can always be done, but it can be fully accidental, providing no sensible information, so that a tree using the split is very unlikely to generalize well the knowledge behind the training data. Probably the most popular enterprise to reduce the bias was introduction of information gain ratio in place of information gain in C4.5 algorithm (see Sect. 2.2.3 and Quinlan 1993).

The problem of bias is not specific only to selecting among unordered features with various counts of possible symbols. It also concerns comparisons between symbolic and numeric variables. Moreover, in different methods the bias may be in favor of different groups of features, for example, some methods are biased in favor of multi-valued symbolic features and some others against them.

Statistical Methods

One of the most popular idea in the approaches to unbiased variable selection for DT splits is separation of the procedures of feature selection and final split definition (feature selection and split-point selection).

In so called statistical trees, it has been often realized by application of statistical tests to estimate feature importance as an opposite of independence between the feature and the target variable.

A turn to such statistical approach was made by White and Liu (1994), who proposed using χ^2 distribution for the purpose of attribute selection instead of measures like information gain, information gain ratio and Mántaras distance (de Mántaras 1991), which are biased in favor of attributes with large numbers of symbols. The approaches utilizing χ^2 distribution are preferable, because they facilitate sensible comparisons of results calculated for different attributes. Compensation of the dependence on the number of cells in the contingency tables being analyzed, was obtained by using χ^2 probability instead of the test statistic, as the probabilities are comparable also in the case of different distribution parameters (degrees of freedom), while the values of test statistics are not. White and Liu (1994) also proposed using G statistic, defined in the language of information theory for the same purpose.

The idea of using statistical test p-values instead of test statistics directly has been applied by many more authors in their approaches to unbiased comparisons. In fact, it is the most commonly used tool of bias elimination efforts.

White and Liu (1994) illustrated their conclusions with simulation results on datasets with three different class distributions, but all with the assumption of independence between the class variable and the predictors.

Because the probabilities coming from the χ^2 and G statistics can be approximated with large error when the expected frequencies are small, White and Liu (1994) suggested using Fisher's exact probability test (see Appendix section A.2.4) instead, in the case of two-class problems. For multi-class tasks, similar approaches can be developed.

Both ideas of separating feature selection from split selection and making decisions on the basis of p-values of statistical tests have been extensively explored and applied to several interesting DT induction algorithms proposed by the group of prof. Loh: FACT, QUEST, CRUISE, GUIDE and others (see Sect. 2.2.5 and Loh and Vanichsetakul 1988; Loh and Shih 1997; Kim and Loh 2001; Loh 2002).

In FACT, symbolic features are converted into numeric ones, and then, for both types of features, F statistic is calculated for each variable to estimate its eligibility for the split. It favors symbolic features, especially those with many possible values.

The solution of QUEST (Loh and Shih 1997) was to reduce the bias by using different statistics (F statistic$F$ and χ^2 statisticχ^2) to estimate continuous and discrete features respectively. Final comparisons were performed for the p-values obtained from adequate distributions. Comparative bias analysis was performed for many pairs of variables of various distributions by drawing samples according to the distributions with class labels generated independently from the predictors. The scores of QUEST were quite close to the value of 0.5, expected in the case of unbiased method.

The results obtained with FACT and exhaustive search were significantly different than 0.5 in many of the tests, confirming their bias.

The feature selection strategy of QUEST was so successful that the CRUISE method of Kim and Loh (2001), published several years later, borrowed this part of the algorithm for its 1D option. Moreover, it introduced the option 2D, capable of detecting some interactions between features with negligible bias in variable selection, similarly to the univariate split analysis (1D). The 2D method, analyzing pairs of features from the point of view of contingency tables of their joint distributions, is presented in detail in Sect. 2.2.5.3.

Apart from the null case of bias analysis (assuming no discriminatory power of the predictors), Kim and Loh (2001) have also examined the problem of bias created by missing values. One of the predictors was deprived of a part of values (20, 40, 60, 80 %) to examine how it affected the split feature selection process (still assuming independence of the target from the predictors). The experiments confirmed the values of CRUISE solutions, in the sense that they selected each of the uninformative features with the same probability.

Another study of attribute selection bias was made by Shih (2004), who focused on the Pearson χ^2 statistic used in many approaches to statistical DTs like CHAID, FIRM (Hawkins 1999) and QUEST family (Loh and Shih 1997; Shih 1999; Kim and Loh 2001). The simulations performed and analyzed by Shih (2004) were composed to test two aspect of bias: the null case of class variable independent of five predictors drawn from different distributions (with some missing observations in one of the variables), and the power studies, testing the abilities to detect an informative predictor among uninformative ones. In the latter tests, different counts of predictors independent from the target were drawn as Gaussian noise. With the total of 5, 10, 15, 20 input variables, the influence of sample size on feature selection has been examined. Conclusions are compatible with those from all other similar experiments, that is, the p-values are more adequate for feature selection than χ^2 statisticχ^2 and ϕ^2 statisticϕ^2 (χ^2 divided by the number of cases) statistics.

Bias in feature selection has also been analyzed in the context of problems with multivariate responses. Some aspects of multi-label classification (where each object can belong to many classes) have been explored by Noh et al (2004), resulting in a DT induction algorithm (M2) splitting nodes with a statistic of Nettleton and Banerjee (2001) for testing equality of distributions of categorical random vectors. In M2, feature selection is also separated from split determination. Similar experiments as in other approaches (null case and power tests) are performed to confirm that the algorithm is unbiased.

An approach to learning multivariate responses by DT models has also been undertaken by Lee and Shih (2006), who generalized the variable selection method of QUEST and CRUISE for multivariate responses. They proposed *CT algorithm* based on conditional independence tests. The feature selection of CT constructs 3-way contingency tables (with dimensions corresponding to feature values, class variables, and response layers respectively) and use χ^2-test extension to estimate features with corresponding p-values. Continuous features are split into quartiles before the 3-way contingency tables are created. Lee and Shih (2006) compared their

new method with an approach of Siciliano and Mola (2000), where weighted sums of Gini index reduction for each response component were used to select variables, and to the M2 algorithm of Noh et al (2004), mentioned above. Both CT and M2 are free of selection bias (in the null case), but CT is shown as providing higher estimated probability of selecting the correct covariate when the response vector depends on that covariate.

Dramiński et al. (2008, 2011) proposed a feature selection method based on a costly Monte Carlo procedure and thousands of DTs generated for different feature sets and training data objects. Feature selection based on relative importance of the features, calculated on the basis of their role in large number of trees, also shows a negligible bias.

When statistical tests are performed to select split features in DTs, there is also a simple possibility of bias reduction by means of Bonferroni corrections, respecting the numbers of possible split candidates available for particular features.

Permutation Tests

Most of the statistical approaches to feature selection for DT splits, discussed above, share the idea of preparing contingency tables and assessing independence between each feature and the target, by means of test statistics like χ^2 statisticχ^2 of G statistic$G$. They are statistically correct and successful, if the data samples, for which they are calculated, are sufficiently large. Then, both χ^2 and G are distributed with chi-squared distribution. Otherwise, the assessment accuracy may be low (Agresti 1990). Unfortunately, in DT induction, dealing with small samples is inevitable, because the data samples are getting smaller and smaller with subsequent splits.

A robust family of methods has been proposed as a result of applying permutation tests to the goal of variable assessment. The fundamental advantage of permutation tests in the context of DT learning is their eligibility for small data samples. Their p-values can be calculated directly, without passing through χ^2 or any other "transient" statistic. By analysis of all possible permutations of the series of values defining targets of the training data, one can estimate the probability of observing such input-target association as in the training data, on the assumption that the input and the target variables are independent.

Frank and Witten (1998) proposed application of approximate permutation tests presented by Good (1994). The tests are based on the multiple hypergeometric distribution. In the approach, appropriate p-values are determined and used for attribute selection and pre-pruning of decision trees.

The permutation tests framework of Strasser and Weber (1999) has been found very useful by Hothorn et al. (2004) and Zeileis et al. (2008) for another successful DT induction approach. The CTree algorithm is described in detail in section 2.2.6.

Bias in Heuristic-Based Methods

Naturally, bias in feature selection has also been a concern of the authors of heuristic based DT induction algorithms. For example, favoring discrete features with

many possible values became the foundation of information gain ratio of C4.5. Also Breiman et al. (1984) noted that, when Gini gain is used as splitting criterion, "variable selection is biased in favor of those variables having more values and thus offering more splits". The analysis of Quinlan and Cameron-Jones (1995) addressed the problem of "Fluke theories" that can be selected by DT algorithms, because they seem accurate, but then, turn out to have low predictive accuracy. Analysis of multi-valued symbolic variable, means testing many theories, among which such "fluke theories" may be selected with the higher probability, the more symbols are possible in the variable. In exhaustive search, the number of tested theories grows exponentially with the number of symbols.

Kononenko (1995) analyzed a number of DT split measures in the context of multi-valued attributes, and concluded that for some of the measures, the probability of selecting a feature increases, and for some other decreases with growing number of possible feature symbols. Using p-values of statistical tests can bring almost unbiased decisions, but suffers from the problem of discriminating more and less informative attributes. When the target is certainly dependent on an attribute, the corresponding p-value gets the value of 1. Two such informative attributes get the same value, although one can still be significantly more informative than the other. When both estimates come from distributions of the same parameters (for example χ^2 with the same degrees of freedom), raw statistic values can be compared to determine which feature is better, but if the distributions are different, no tool to discriminate the two variables is available. As an alternative solution, Kononenko (1995) proposed a criterion based on the MDL principle, which is slightly biased against multi-valued attributes.

An analysis of bias of the Gini criterion used in CART has interested Dobra and Gehrke (2001), who proposed a general method to remove bias of such criteria and showed its application to Gini index. The general method of bias removal consists of two steps: first the value of split criterion is calculated and then its p-value under null hypothesis is determined. Dobra and Gehrke (2001) proposed four ways of computing the p-values for split criteria: exact computation (very expensive), bootstrap estimation (also costly), asymptotic approximations (inaccurate for small samples) and tight approximations (may be hard to find). The authors presented a tight approximation of the Gini gain.

Another p-value based measure derived from Gini index has been proposed by Strobl et al. (2005). Their selection criterion based on the Gini gain was inspired by the theory of maximally selected statistics. To calculate the criterion score, they estimated the distribution function of the maximally selected Gini gain, and calculated appropriate p-value under the null-hypothesis of no association between the target and predictor variables. To derive the exact distribution function of the maximally selected Gini gain, Strobl et al. (2005) used a combinatorial method following the ideas of Koziol (1991) to determine the distribution of the maximally selected χ^2 statistic.

Similarly to other authors studying bias of different DT criteria, Strobl et al. (2005) performed a null case experiment, where five predictors contained no information

about the response variable, and power case studies, assuming that one feature was informative and contained some percentage of missing values.

Concluding Remarks

The pursuit of unbiased feature selection methods for DT induction has brought several very interesting algorithms, but they still are just some among many possible approaches and do not guarantee more accurate models, when solving particular tasks.

It might seem that provided an unbiased method for split-feature selection and equally valuable split selection algorithm, one can create a perfect algorithm, generating optimal DT models for all learning problems. The truth is different because of two reasons:

1. As discussed at the very beginning of the chapter, optimization at node level is not the same as optimization of tree models.
2. The term "unbiased" sounds almost like "perfect", but its definition is not as general as its common sense meaning, and as a result, unbiased methods are not as robust as they might seem from general statements.

To make the efforts viable, unbiased attribute selection is usually defined as preserving equal probability of selection of each feature, independent from the response variable. Theoretical proofs of unbiasedness have nice statistical foundations and are mathematically correct, but do not have so much practical value as might be expected, because good behavior for uninformative (statistically independent) features does not guarantee fair selection between informative features. Models useful in practice make decisions on the basis of informative features, so do not fit the theoretical frameworks. Experiments, confirming that unrelated features are selected equally often, are also correct, but do not explore the actual areas of interest.

As it has been pointed out above, p-values of independence tests run for informative features often are equal to 1, which makes just comparison impossible. When uninformative features are analyzed, their expected p-values are close to 0.5, and the analysis is focused on the area of the most changeable part of cumulative distribution functions, where comparisons may be accurate. When the p-values are equal to 1 and the values of statistics for different attributes are incomparable to each other, fair feature selection can not be done with such tools.

There is no single, commonly accepted definition of a measure of information about response variable, contained within an attribute. That's why so many split criteria have been proposed and no one seems definitely better than all the others. Again, the conclusion about the most reasonable strategy of the search for the best models, directs attention towards meta-learning techniques, capable of

- gathering and drawing conclusions from meta-knowledge about advantages of various algorithms in various applications,
- making predictions of potential gains resulting from running various learning machines for particular learning data,

- making algorithm-selection decisions after validation of models estimated as the most adequate.

Successful model selection is possible only with a bunch of powerful base-level learning algorithms, some meta-knowledge and robust, efficient meta-learner.

2.8 Ensembles of Decision Trees

Many researchers have been attracted by the idea of constructing complex models on the basis of collections of other models. Scientists working in the area of decision tree induction have been especially prolific in this context. The ideas of bagging, boosting and other approaches to combining decisions of sets of models are regarded by some experts as the most significant achievements of computational intelligence research of the 1990s.

Averaging decisions of multiple models can be justified in many theories. One of the most sensible way is the analysis on the ground of Bayesian learning theory. When many models (hypotheses) exist for given training data D, the optimal choice of the class $c \in \mathscr{C}$ for a data object x is defined by the *Bayes Optimal Classifier*:

$$BOC(x|D) = \arg\max_{c \in \mathscr{C}} P(c|x, D) = \arg\max_{c \in \mathscr{C}} \sum_{M \in \mathscr{M}} P(c|x, M) P(M|D). \quad (2.87)$$

Although it is usually not possible to explore the whole space $\mathscr{M}$ of possible models, approximations by some families of probable models are very sensible. Often, a single model is selected on the basis of a criterion like MAP (*maximum a posteriori*) or ML (*maximum likelihood*):

$$M_{MAP} = \arg\max_{M \in \mathscr{M}} P(M|D) = \arg\max_{M \in \mathscr{M}} P(D|M) P(M), \quad (2.88)$$

$$M_{ML} = \arg\max_{M \in \mathscr{M}} P(D|M). \quad (2.89)$$

They can be seen as extreme approximations of the $\mathscr{M}$ family by one-element sets. Informally, one can claim that such approximations make more sense than using random collections of models, because the MAP and ML models are models of confirmed quality.

More detailed analysis and more information on Bayesian learning theory can be found, for example, in the (very good) chapter on the subject by Mitchell (1997).

Approximating Bayes Optimal Classifier by a restriction of usually infinite model space $\mathscr{M}$ to a finite set of models $\mathscr{M}' = \{M_1, \ldots, M_s\} \subseteq \mathscr{M}$ and estimation of $P(M|D)$ by some weights w_M leads to a classifier estimating class probabilities for given object x as:

$$P(c|x) = \sum_{M \in \mathscr{M}'} w_M \cdot P(c|x, M). \quad (2.90)$$

Estimation of the conditional probabilities $P(c|x, M)$ of class c given DT models is quite easy (although not unanimous, so many authors have proposed their solutions). More difficult part of the task is finding proper weights w_m. Some researchers try to go further in Bayesian analysis and determine $w_m = P(M|D)$ as $P(D|M)P(M)$ (Buntine 1993), but here also the definition of priors $P(M)$ is not self-evident. In other approaches, the weights are assumed to be equal (like in bagging and other unweighted voting scenarios) or are determined by miscellaneous algorithms to reflect model competence and the strength of its influence on final ensemble decisions.

By averaging decisions of many models, one can also generalize to new variations, not observed in the training data sample (Bengio et al. 2010). This feature is unavailable to single DT learners, because the splits are determined on the basis of the evidence available in the training data.

Building complex models can bring improvement in approximation error, but a price must be paid for that. Computational costs are the most obvious, but not the only ones. In the realm of decision tree models, the most significant loss accompanying the ensemble gains in modeling accuracy is the loss of model comprehensibility. This cost is especially oppressive, because readable and understandable form is one of the most important reasons of DT induction popularity and appreciation. Interpretation of complex models is much more difficult, so different forms of visualization and explanation of the decision functions are created as some recompense.

Ensembles owe the improvement in accuracy to diversity of their component models. Only a set of committee members specializing in different subareas of the domain of learning can introduce new value, when properly combined. Model diversity may come from different sources (Zenobi and Cunningham 2001; Melville and Mooney 2003; Brown et al. 2005). Two groups of the sources seem the most important:

- different learning algorithms (methods of completely different domains or just changes in parameters of a single learning strategy),
- different training datasets (sampling, transformations and so on).

Analyzing similarity between algorithms does not seem to make much sense, as no dissimilarity guarantees diversity of resulting models. Quite often, completely different DT induction algorithms create very similar (or even exactly the same) decision trees. From formal point of view, it is not justified to distinguish between "completely different algorithms" and "small differences in parameters of the same algorithm", because even the smallest change in parameter settings results in a different algorithm. It is also quite common that a small change in a single parameter results in completely different DT model.

Decision tree induction methods are known to be unstable, which means that small changes of the input data may cause significant changes in the results of learning. Because of that, in the realm of DTs, manipulating the training sample seems much more interesting source of model diversity, and has been explored by many scientists.

In construction of ensembles, two techniques can be distinguished with respect to the fundamental organization of the member learning processes:

- *independent model generation*, also called *perturb and combine* (P&C) approach,
- *dependent subsequent models*, also referred to as *adaptive resample and combine* methods.

The former group facilitates easy parallelization of computations, because each member model is generated independently. In the latter approach, ensemble members must be generated sequentially, one by one, because each next model is constructed on the basis of the results of all the previous models.

Practical learning problems are usually defined by a restricted set of data object descriptions that can be used for learning, so that it is not possible to generate arbitrary many datasets of arbitrary size. Therefore, generation of different training datasets is not a trivial task. The most popular approaches belong to one of two groups: *resampling* and *reweighting*. Resampling methods generate new sets by selection of objects from the original dataset (possibly with repetitions). Reweighting techniques assign weights to each data object from the original dataset and pass the weights to the learning algorithm together with the dataset. Obviously, such operation makes sense only when the learning algorithm accepts and can take advantage of such weights in its learning process.

The most popular approach to independent model generation is *bagging* (Breiman 1996; Quinlan 1996) described in Sect. 2.8.2. Its original definition was a resampling method, but a modification to accept weights (called *wagging*) has also been examined (Bauer and Kohavi 1999).

The family of algorithms generating dependent models is usually referred to as *boosting* and is described in Sect. 2.8.3. The methods are so general that can be used in the manner of both resampling and reweighting.

Breiman (2001) presented a general view of DT ensembles and presented some particular algorithms for random forests generation. More on these methods can be found in a successive subsection.

Many other algorithms have also been proposed to construct ensemble models. Some authors noticed that partitioning methods like cross-validation can be successful in building diverse committees (Parmanto et al. 1995; Domingos 1996; Grąbczewski and Jankowski 2006a; Grąbczewski 2012).

Another idea to collect diverse DT models for ensemble construction is to perform more thorough search in the space of DT models and collect a number of attractive trees instead of just a single tree classifier. Beam search is a perfect tool for such purposes and has been used to induce DT forests based on SSV criterion, also heterogeneous ones, that is, using premises concerning distances from prototypes apart from standard single feature splits (Grąbczewski and Duch 2002a, 2002b).

2.8.1 Option Decision Trees

Option Decision Trees (ODT, Buntine 1993; Kohavi and Kunz 1997) are classification models comprising many DTs in a single complex structure. Actually, ODT structures

are equivalent to a number of separate DTs, but thanks to keeping them together, no common branches are represented in multiple copies. During DT induction process, at each node, several alternative splits are recorded if possible, and the whole alternative branches are constructed.

Nevertheless, not the memory saving is the main focus of ODTs. Thanks to collecting alternative splits at each node of the tree, alternative paths are available and each data object can be classified with respect to many paths, by proper decision averaging. Buntine (1993) has proposed a framework for *Bayesian averaging* of collections of possible decision paths in trees. The framework is applicable also to single trees, because a set of trees resulting from all possible ways of pruning the tree can be seen as an approximation $\mathscr{M}'$ of the space of models of Eq. (2.90). With the procedure named *tree smoothing*, Buntine (1993) assigned probabilities to tree leaves that are reported to estimate class probabilities more accurately.

The most serious drawback of the approach is significant increase of the computation time in comparison to single DT induction techniques. Buntine tested his algorithms also with more thorough search techniques like n-ply lookahead search$n$-ply lookahead with beam width restriction, but naturally, it additionally increased the time of computations.

2.8.2 Bagging and Wagging

The term *bagging* comes from the expression bootstrap*bootstrap aggregating* (Breiman 1996; Quinlan 1996). Diverse learning machines are created by means of preparing *bootstrap samples* for each subsequent learning process. A bootstrap sample drawn from a given dataset D is a collection of items drawn from D at random, independently, with replacement. This shows the inadequacy of the term "dataset", as in the bootstrap sample, an object can occur several times. It has been confirmed that bootstrap samples are successful sources of model diversity.

The technique of bagging is presented by Algorithm 2.22. A predefined number of bootstrap samples of the same size as the original training data is created and a model learned from each. Eventually, the decisions of all the models are combined by ordinary majority voting.

If the learning algorithm applied to generate the ensemble member models can learn with respect to weights assigned to training data objects, instead of drawing bootstrap samples, one can just draw weights and pass them to the learning machine. Such methodology has been named *wagging* (for *weight aggregation*, Bauer and Kohavi 1999) and is presented as Algorithm 2.23.

The definition of the algorithm is very general, as it refers to arbitrary weighting distribution given as a parameter. Bauer and Kohavi (1999) added Gaussian noise to each weight with mean zero and a given standard deviation. Because negative weights do not make sense, each weight value falling below 0 is treated as 0 and the object assigned such weight has no influence on the learning process (the object is treated as nonexistent). Increasing the standard deviation of the noise reduces the

Algorithm 2.22 (Bagging)

Prototype: ***Bagging****(D, s, L)*
Input: *Training data D containing n objects, ensemble size s, learner (machine) L.*
Output: *Voting committee.*
The algorithm:

1. ***for*** *i=1,...,s* ***do***
   a. $D_i \leftarrow$ *bootstrap sample of size n generated from D*
   b. $M_i \leftarrow L(D_i)$ */* train a machine */*
2. ***return*** *the committee* $\{M_1, \ldots, M_s\}$

Algorithm 2.23 (Wagging)

Prototype: ***Wagging****(D, s, L, d)*
Input: *Training data D, ensemble size n, learner (machine) L, weighting distribution d.*
Output: *Voting committee.*
The algorithm:

1. ***for*** $i = 1, \ldots, s$ ***do***
   a. $\mathbf{w} = (w_1, \ldots, w_n) \leftarrow$ *weights drawn randomly from d for each data object in D*
   b. $M_i \leftarrow L(D, \mathbf{w})$ */* train a machine */*
2. ***return*** *the committee* $\{M_1, \ldots, M_s\}$

training dataset, so increases the bias and reduces the variance of learning, facilitating some control on the bias-variance trade-off.

When comparing the algorithms of bagging and wagging, it can be noticed that the learning process is called in different ways. This reflects the difference between resampling and reweighting that bore the wagging algorithm.

Although the original definition of bagging assumed ordinary majority voting as the final decision function of the ensemble, the decision module can be modified in several ways. For example, the combination may reflect probabilities estimation (of belonging to particular classes) returned by probabilistic classifiers as in the experiments presented in Chap. 5.

Another interesting technique to improve bagging and wagging results is *back-fitting* proposed by Bauer and Kohavi (1999). Because the training data samples generated for bagging purposes, contain significantly less objects than the original data (around 63.2 %, see the explanation below), it is advantageous to feed the whole original dataset to the tree and estimate class probabilities at the leaves more accurately. Combined probabilities, estimated in this way, provide better results than simple voting and then probabilities estimated on the bootstrap samples. Similarly, in wagging, the training dataset can be passed through the trees with equal weights for all the objects to obtain better estimation of class probabilities.

Error Estimation with .632 Bootstrap

An analysis of bootstrapping in the context of bagging (Efron 1983; Efron and Tibshirani 1997) brings interesting conclusions about learning possibilities. The probability that a given data object occurs in the bootstrap sample of size n drawn from an input dataset of size n is equal to

$$1 - \left(1 - \frac{1}{n}\right)^n. \tag{2.91}$$

With $n \to \infty$ it converges to $1 - \frac{1}{e} \approx 0.632$. Already for $n = 24$ the value is less than 0.64. The larger n the closer to the limit. Hence it is justified to approximate the part of original dataset occurring in the bootstrap sample as 63.2 %.

Because bootstrap samples contain on average just 63.2 % of objects from the training set, the error estimates are pessimistic. On the other hand, it is obvious that the error estimate calculated for the data used for training can be too optimistic. Therefore, Efron (1983) proposed the *.632 estimator*:

$$\widehat{err}^{(.632)} = .368 \cdot \overline{err} + .632 \cdot \widehat{err}^{(1)}, \tag{2.92}$$

where $\overline{err}$ is the error estimate calculated from training data (biased downwards) and $\widehat{err}^{(1)}$ is the bootstrap estimate (prediction from classification error calculated for points not occurring in the bootstrap sample; biased upwards). Efron and Tibshirani (1997) suggested further improvement of the estimate and proposed a $\widehat{err}^{(.632+)}$ estimate shifting the balance between $\overline{err}$ and $\widehat{err}^{(1)}$ towards the latter, on the basis of a factor named *relative overfitting rate*.

Recent studies by Kim (2009), aimed at fair comparison of three approaches to estimating error rates of classifiers (repeated cross-validation, repeated hold-out and .632 bootstrap), have brought conclusions that different methods provide the best estimations for different tasks, depending on the learning sample size and the classification learner being tested. In particular, repeated CV turned out to be more adequate for "highly adaptive" classifiers, that is, methods like boosting, capable of gaining resubstitution error close to 0.

2.8.3 Boosting

The idea of *boosting classifiers* has been introduced by (Freund and Schapire 1995, 1996, 1997). They have proven some important properties justifying the solutions. In general, *boosting* is a method of converting "weak" learning algorithm to a "strong" algorithm with arbitrarily high accuracy. Because the main means of the method is adaptive resampling (or reweighting) and combining, Breiman has used the name *arcing* for this technique.

Boosting classifiers is a technique of repeated training of a given learning machine on data samples (or weighted data) generated with respect to probability distributions

adjusted to the results of previous learning processes. After a number of models is learnt, they form a committee with properly weighted decisions.

AdaBoost Algorithm

One of the first boosting procedures proposed by Freund and Schapire (1995) was AdaBoost (for adaptive boosting). The algorithm is still the most popular one of this kind. It is presented formally as Algorithm 2.24. The method builds a collection of models on the basis of the training dataset and probabilities p_x assigned to each object x of the training set. The learning stage (item 2a of the algorithm) can be realized in different ways, depending on the preferred (or at all possible) strategy: *weighting* or *sampling*. If the learning process accepts weights assigned to each training data object, then the training dataset and the weights may be passed to it. Otherwise, a data sample is derived from the training set with respect to the probability distribution p and passed to the learning machine.

Algorithm 2.24 (AdaBoost)

Prototype: ***AdaBoost****(D, s, L)*
Input: *Training data* $D = \{(x_1, c_1), \ldots, (x_n, c_n)\}$*, ensemble size* s*, "weak" learner* L*.*
Output: *Weighted committee.*
The algorithm:

1. ***for each*** $(x, c) \in D$ ***do*** */* initialize the probability distribution as uniform */*
   $p_x \leftarrow \frac{1}{n}$
2. ***for*** i*=1,...,*s ***do***
   a. $M_i \leftarrow Learning(L, D_i, p.)$ */* train a machine with respect to* $p.$ **/*
   b. $\varepsilon_i \leftarrow \sum_{\{(x,c)\in D: M_i(x)\neq c\}} p_x$
   c. ***if*** $\varepsilon_i > \frac{1}{2}$ ***then***
      i. $s \leftarrow i - 1$
      ii. ***break*** *the loop*
   d. $\beta_i \leftarrow \dfrac{\varepsilon_i}{1 - \varepsilon_i}$
   e. ***for each*** $(x, c) \in D$ ***do*** */* modify the probability distribution */*
      $p_x \leftarrow p_x \cdot \dfrac{1}{2\varepsilon_i} \cdot \beta_i^{\mathbf{1}_{\{c\}}(M_i(x))}$
3. ***return*** *the weighted committee* $M = \left(\{M_1, \ldots, M_s\}, \{\log \frac{1}{\beta_1}, \ldots, \log \frac{1}{\beta_s}\}\right)$*:*
   $M(x) = \arg\max_{c\in\mathscr{C}} \sum_{i:M_i(x)=c} \log \dfrac{1}{\beta_i}$

The initial distribution is uniform, so D_1 is a bootstrap sample generated in the same way as in the bagging approach. Then, after each subsequent model M_i is created, its error ε_i is calculated and the probability distribution is modified (see item 2e of the algorithm) to focus the subsequent learning processes more on the misclassified data objects than on the ones classified correctly. The idea of distribution

changes is to multiply by $\beta_i < 1$ (diminish) the probabilities of correctly classified objects in such a way that the sum of probabilities of incorrectly classified objects becomes equal to $\frac{1}{2}$ (the other half is the sum of probabilities of correctly classified objects). Hence the factor $\beta_i \leftarrow \frac{\varepsilon_i}{1-\varepsilon_i}$, because the sum of p_x for all correctly classified xs, before the modification is $1 - \varepsilon_i$, so after multiplication by β_i becomes ε_i which by definition is the sum for incorrectly classified objects. Therefore, the factor $\frac{1}{2\varepsilon_i}$ restores the properties of probability distribution.

The final decision of the ensemble is made on the basis of combined decisions of the members with larger weights for more accurate models and smaller for the poorer classifiers. The goal is reached by the weights specified in AdaBoost as $\log \frac{1}{\beta_i}$.

Such formulation of the AdaBoost algorithm has originally been named AdaBoost.M1 as a modification of the first formulation devoted to two-class problems. In multi-class classification, one of the most significant weak points of AdaBoost is the requirement that the errors of weak learners $\varepsilon_i < \frac{1}{2}$. Otherwise, the factors β_i would get larger than 1 and the goal of boosting would get inverted: the probabilities for erroneously classified objects would be decreased instead of increased. To remedy this, Freund and Schapire (1995) proposed an algorithm named AdaBoost.M2, which uses another shape of the models (fuzzy classification by assigning a value in the interval [0, 1] to each class instead of crisp class assignment), another scheme of handling the probability distribution and modified ensemble decision function. The algorithm is less popular, so it is not presented here in detail. It can be found in the articles by Freund and Schapire (1995, 1996, 1997).

When the error of subsequent model gets larger than $\frac{1}{2}$, the AdaBoost algorithm is stopped and the ensemble is composed of the models, found so far, without the last one of too large error (see item 2(c)ii of Algorithm 2.24). Other authors suggest restarting the probabilities (going back to the uniform distribution) in such circumstances, instead of breaking the process or preforming the normal scheme of probability adjustment. After the reset of the probability distribution, the next sample is again a bootstrap sample. If each model results in error greater than $\frac{1}{2}$, then each sample is a bootstrap sample, and the member models are the same as in bagging. The only difference with bagging, in such case, is the weighted decision function of the final model instead of the majority voting used in bagging.

When a model at some stage perfectly classifies the training data ($\varepsilon_i = 0$), the next stage of AdaBoost is not feasible, because the distribution gets degenerate. In such cases, it is also suggested by some authors to reset the probability distribution and build next model on another bootstrap sample.

AdaBoost Modifications

Many different variants of AdaBoost can be found in the literature. Three of them have been used in the experiments described in Chap. 5: *conservative boosting* (Kuncheva and Whitaker 2002), *averaged boosting* (Oza 2003) and *averaged conservative boosting* (Torres-Sospedra et al 2007).

Conservative boosting (Kuncheva and Whitaker 2002), as suggested by its name, uses a more conservative method of distribution adjustment. The conservativeness

is realized by changing the step 2d of Algorithm 2.24 to

$$\beta_i \leftarrow \sqrt{\frac{\varepsilon_i}{1-\varepsilon_i}}, \tag{2.93}$$

and replacing the denominator $2\varepsilon_i$ of the renormalization factor used in step 2e to $\sum_{(x,c)\in D} p_x \cdot \beta_i^{1_{\{c\}}(M_i(x))}$. The introduction of the square root makes the factor β_i larger, so the decrease of the probabilities assigned to correctly classified objects is slower and more correctly classified objects are kept in the next training data. The change of normalization factor is a natural consequence of the modified β_i.

Kuncheva and Whitaker (2002) have also tried a version of boosting called *inversed*, because by changing the factor from $\beta_i^{1_{\{c\}}(M_i(x))}$ to $\beta_i^{1-1_{\{c\}}(M_i(x))}$, the probabilities assigned to correctly classified vectors are increased in this approach, while the incorrectly classified objects get less probable in the next sample. The inversion is a technique similar in idea, to iterative refiltering, used in the DT-SE family of DT induction methods (see section 2.3.4).

The algorithm of *averaged boosting* (Oza 2003) also differs from AdaBoost in the way it modifies the probability distribution after each step. It also slows down the changes in comparison to the standard AdaBoost, as instead of the new value calculated in the AdaBoost manner, it sets new values as the average of all AdaBoost corrected values from the first stage of the process:

$$p_x \leftarrow \frac{i \cdot p_x + p_x \cdot \frac{1}{2\varepsilon_i} \cdot \beta_i^{1_{\{c\}}(M_i(x))}}{i+1}. \tag{2.94}$$

Averaged conservative boosting (Torres-Sospedra et al 2007) is the method combining the ideas of averaged boosting and conservative boosting. The β_i factor of the algorithm is defined by Eq. (2.93), and the probability distribution is corrected according to the formula

$$p_x \leftarrow \frac{1}{Z_i} \cdot \frac{i \cdot p_x + p_x \cdot \beta_i^{1_{\{c\}}(M_i(x))}}{i+1}, \tag{2.95}$$

where Z_i is the renormalization value that guarantees the probability distribution properties of p. This definition may seem slightly inconsequent, because the average is calculated from normalized p_x and not normalized new part $p_x \cdot \beta_i^{1_{\{c\}}(M_i(x))}$. It seems more adequate to normalize the new part first and then calculate the weighted mean, which would not need further normalization as is averaged boosting.

Arc-x4 Algorithm

Breiman (1998) has also undertaken the analysis of adaptive resampling and combining (hence the term *arcing*). He has examined the original approach of Freund and Schapire (1995, 1996, 1997) to boosting in application to creating DT ensembles

and introduced his own ad-hoc algorithm of similar idea. In this work, the original boosting approach was referred to as *arc-fs* and the Breiman's ad-hoc version as *arc-x4*. Arc-fs was modified to reset the uniform distribution in the case of $\varepsilon_i > \frac{1}{2}$ or $\varepsilon_i = 0$.

Algorithm 2.25 presents the Breiman's algorithm formally. The fourth power of $m_{i,x}$ was chosen as the best of three tested values (1, 2 and 4), so it does not come from a significant optimality analysis. Even so simple boosting approach turned out to be quite successful.

Algorithm 2.25 (Arc-x4)

Prototype: ***Arc-x4****(D, s, L)*
Input: *Training data* $D = \{(x_1, c_1), \ldots, (x_n, c_n)\}$*, ensemble size s, "weak" learner L.*
Output: *Voting committee.*
The algorithm:

1. ***for each*** $(x, c) \in D$ ***do*** */* initialize the probability distribution as uniform */*
   $p_x \leftarrow \frac{1}{n}$
2. ***for*** $i = 1, \ldots, s$ ***do***
   a. $M_i \leftarrow Learning(L, D_i, p.)$ */* train a machine with respect to p. */*
   b. ***for each*** $(x, c) \in D$ ***do*** */* modify the probability distribution */*

$$p_x \leftarrow \frac{1 + m_{i,x}^4}{\sum_{(x',c') \in D} 1 + m_{i,x'}^4},$$

   where $m_{i,x}$ *is the number of models in* $\{M_1, \ldots, M_i\}$ *that misclassify x*
3. ***return*** *the voting committee* $M = (M_1, \ldots, M_s)$

Breiman used resampling to generate each training dataset D_i on the bases of the probability distribution p. Apart from the training set, he used another set generated in the same way, from the same probabilities p, as a validation set for DT pruning. It was significantly more efficient than pruning on the basis of cross-validation.

TreeNet

Another boosting approach worth mentioning is a commercial product named *TreeNet*. It arose from the *Multiple Additive Regression Tree* (MART) system dated back to 1999. The fundamental idea of the approaches is the *stochastic gradient boosting* technology of Friedman (1999a,b). Similarly to the boosting approaches described above, also here, the ensembles are additive models combining a set of base models:

$$M(x) = \sum_{i=1}^{s} \beta_i M_i(x). \tag{2.96}$$

Each subsequent model M_i is selected to minimize the value of a loss function, determined on the basis of a training dataset. This technique is a very similar to *cascade correlation* used for training neural networks, where neurons are added in sequence to improve the performance of the network on the training set.

The main difference between the methods of MART (or TreeNet) and AdaBoost is that the gradient boosting methods are devoted to regression, not classification problems. Naturally, classification tasks can be solved by means of regression tools with binomial log-likelihood loss function.

The power of the algorithms is in large numbers of DT models combined. Each single tree learns very little from the data. It can be claimed that high quality of single DTs is not desired in these approaches. Because of that, the trees are never trained on the whole training dataset—usually a random half of the data is used for learning a single tree.

Single trees are not very adequate models for regression goals, because they represent coarse step functions (piecewise constant). Thanks to combining large numbers of trees, quite "smooth" curves can be obtained. Because of large numbers of combined models, the final models do not share the advantages of single tree comprehensibility. Instead, special reports are designed to extract the meaning of the model in the form of feature importance rankings or graphs illustrating the relationship between inputs and outputs.

Alternating Decision Trees

Induction of a generalized decision trees called *alternating DTs* (ADTrees) has been proposed by Freund and Mason (1999).

As a generalization of DTs, ADTrees can represent standard DTs but also significantly more complex structures. They have a form very similar to the one of Option Decision Trees. Similarly to ODT, they can encode many trees in a single structure. Thus, they can also easily represent voting stumps.

ADTrees consist of two types of nodes (*prediction nodes* and *decision nodes*) that occur interchangeably on tree paths. Each prediction node is assigned a real value used in decision making on the basis of the structure: all the real values on the multi-path traversed by an object to be classified are summed and the sign of the sum decides about classification into one of two classes. An object can follow a multi-path, not a single standard path, because in a general alternating tree, at each prediction node, all children are tested and point different further paths.

Decision (splitting) nodes of ADTrees are described with a special form of rules, where *precondition* corresponds to the path from the root to the decision node and *condition* represents the test of the node.

To learn ADTree models, the authors decided to use a modification of Adaboost algorithm proposed by Schapire and Singer (1999), because it is suitable for dealing with real valued predictions of ADTrees.

Freund and Mason (1999) declared that their approach showed results competitive with boosted C5.0 trees, but usually with smaller and easier to interpret trees. As

another important advantage, they pointed out that alternating trees give a natural measure of classification confidence.

2.8.4 Random Forests

Breiman (2001) proposed a general definition of decision tree forests as a collection of tree classifiers built with respect to random vectors. In the framework, given a random vector Θ_i for each $i = 1, \ldots, s$, a tree is grown for the training data and Θ_i. Denoting i'th DT classification model as $M_{\Theta_i} : O \rightarrow \mathscr{C}$, a *random forest* is defined as the majority voting classifier based on the collection of DT models $\{M_{\Theta_i} : i = 1, \ldots, s\}$. Additionally, Breiman (2001) assumed that the random vectors Θ_i were independent and identically distributed.

Such definition of random forests encircles many different schemes of DT ensembles, for example bagging, where the random vectors Θ_i may directly correspond to the n (n is the number of elements in the training dataset) object indices pointing the elements of subsequent bootstrap sample.

Boosted classifiers might also fit the definition of random forests, but the assumption of independent and identically distributed Θ_i is not satisfied. To be precise, we may state that boosting algorithms do not conform to the idea of random forest construction, although each particular boosted model might be obtained with the scheme.

The definition of random forests is constructed in a way suggesting the algorithm for growing forests. Such algorithm, for subsequent values of i, draws the random vector Θ_i and then uses it in the learning process to obtain model M_{Θ_i}.

Randomization of the trees composing forests may come from different sources, for example, training data sampling or different configurations of the DT inducer. Breiman (2001) analyzed two methods of random feature selection for each DT split within the CART algorithm:

- random input selection consisting in drawing a small group of input variables (pre-specified count F) from the set of all features describing data objects and limiting the analysis of possible splits to the F selected variables only,
- random linear combinations of randomly selected inputs, where F linear combinations are generated and analyzed to find the best split; each combination is determined by random selection of features to combine (of pre-specified size L) and drawing L coefficients from uniform distribution on $[-1, 1]$.

The randomization idea of random forests was inspired by the article of Dietterich (2000), where C4.5 algorithm was modified to randomize split selection at each DT node. The idea was to introduce diversity by means of split randomization without counting on inducer instability. For each node, the split was randomly selected from the 20 best splits determined in the normal way. The candidate splits were not pre-selected in any way, so in special cases, all 20 best splits could involve the same (continuous) attribute.

Random feature selection used by Breiman (2001) has additional advantage of reducing the cost of computations needed for DT induction. The approach of Dietterich (2000) required the same calculations as normal C4.5 run plus insignificant time for random selection of the split at each node. In the case of random input selection, calculations may be much cheaper, because not all features are analyzed, but only the selected ones. Therefore random forests can easily handle data with large number of attributes—in practice, the complexity does not depend on the number of features composing the object space. The speed up and improved scalability certainly belong to the most attractive properties of the random forests approach.

2.9 Other Interesting Approaches Related to DT Induction

Many more (than described above) algorithms for DT induction have been proposed by miscellaneous authors. It is not possible to present all the techniques in a single article or even book. Many comparative analyses have been performed and are certainly worth a focus, when searching for interesting solutions. Some reviews and comparisons have been published by Safavian and Landgrebe (1991), Murthy (1998), Provost and Kolluri (1999), Anyanwu and Shiva (2009), Rokach and Maimon (2010), and Kotsiantis (2011).

Miscellaneous Split Quality Measures

The most commonly used and some other interesting split quality measures have been presented above. Exhaustive discussion of all other methods published by CI researchers is not possible in this book, but it would not be right to completely ignore all of them. Therefore, some measures are shortly listed below. Some comparisons of selected criteria are available in the literature. For example, Kononenko (1995) compared eleven measures and introduced the MDL measure presented in Sect. 2.4.2.4. Beside the well known measures of Gini index and information gain ratio, he examined the methods of *J-measure*, *Mántaras distance*, *average absolute weight of evidence*, *relief*, *relevance*, measures based on χ^2 statisticχ^2 and G statistic$G$ statistics and the proposed MDL measure.

J-measure
(Goodman and Smyth 1988b) was defined for discrete random variables X and Y, in the language of information theory:

$$J(X|Y=y) = P(y) \sum_x P(x|y) \cdot \log\left(\frac{P(x|y)}{P(y)}\right). \tag{2.97}$$

It has been used in the ITRULE system (Goodman and Smyth 1988c; Smyth and Goodman 1992) for rule induction from data and then applied also to DT induction

(Goodman and Smyth 1988a). J-measure has also been applied to pre-pruning of DTs, resulting in a method named *J-pruning* Bramer (2002).

Mántaras distance
Similarly to information gain and J-measure, Mántaras distance is also defined in the language of information theory. With the definitions of entropy, joint entropy and conditional entropy as in formulae (1.8) and (1.9) in the introduction, the information gain, resulting from the split according to feature A can be written as

$$IG(A, C) = H_C - H_{C|A} = H_C + H_A - H_{A,C}. \tag{2.98}$$

Mántaras distance between partitions determined by feature A and class C can be expressed as

$$MD(A, C) = 1 - \frac{IG(A, C)}{H_{A,C}}. \tag{2.99}$$

Absolute weight of evidence
The measure of *absolute weight of evidence* has been introduced by Michie (1990) for two-class problems, but it can be easily extended to multi-class problems by averaging:

$$WE(A, C) = \sum_{c \in \mathscr{C}} p_c \sum_{a \in \mathscr{A}} p_{a\cdot} \left| \log \frac{p_{c|a}(1 - p_c)}{(1 - p_{c|a})p_c} \right|, \tag{2.100}$$

where $p_{c|a}$ is the proportion of objects of class c among those with value a of A.

Relief
The *relief* algorithm was designed to solve feature selection problems (Kira and Rendell 1992a,b). It adjusts weights assigned to the features (initially zeros for all features) on the basis of an iterative procedure, which for each data vector finds its closest positive and negative instances (called nearest-hit and nearest-miss adequately to the class of the data vector) and increases the weights of features "responsible" for the distance to nearest-miss and decreases the weights of features participating in the distance to the nearest-hit (the larger the distance the bigger the change). An extension of the algorithm has been proposed, that respects k nearest hits and misses in the analysis (*Relief-A*). Kononenko (1994) analyzed the algorithm and proved that if the k is not restricted (all data vectors are taken into account), then the weight for a discrete feature A (with values in $\mathscr{A}$) is highly correlated with Gini index:

$$Relief(A, C) = \frac{\sum_{a \in \mathscr{A}} p_{a\cdot}^2}{\sum_{c \in \mathscr{C}} p_{\cdot c}^2 (1 - \sum_{c \in \mathscr{C}} p_{\cdot c}^2)} \times Gini'(A, C), \tag{2.101}$$

where

$$Gini'(A, C) = \sum_{a \in \mathscr{A}} \left(\frac{p_{a\cdot}^2}{\sum_{a \in \mathscr{A}} p_{a\cdot}^2} \sum_{c \in \mathscr{C}} p_{c|a}^2 \right) - \sum_{c \in \mathscr{C}} p_{\cdot c}^2, \tag{2.102}$$

which differs from Gini index in that it uses coefficients $\frac{p_{a\cdot}^2}{\sum_a p_{a\cdot}^2}$ instead of Gini's $p_{a\cdot} = \frac{p_{a\cdot}}{\sum_a p_{a\cdot}}$.

Relevance index
Baim (1988) introduced the following relevance measure of a partition A with respect to classes C:

$$Relevance(A, C) = 1 - \frac{1}{|\mathscr{C}| - 1} \sum_{a \in \mathscr{A}} \sum_{\substack{c \in \mathscr{C} \\ c \neq c_m(a)}} \frac{n_{ac}}{n_{\cdot c}}, \tag{2.103}$$

where $c_m(a) = \arg\max_{c \in \mathscr{C}} \frac{n_{ac}}{n_{\cdot c}}$. The purpose of the measure was feature selection, so it perfectly fits the needs of split quality evaluation.

ORT
ORT criterion, introduced by Fayyad and Irani (1992a) measures the quality of binary splits on the basis of orthogonality of class probability vectors calculated for the two data subsets resulting from the split. For a data sample D and a test τ inducing a binary partition on D into D_τ and $D_{\neg\tau}$, having class probability vectors $\mathbf{V}_\tau$ and $\mathbf{V}_{\neg\tau}$, respectively, the orthogonality measure is defined as

$$ORT(\tau, D) = 1 - \cos\phi(\mathbf{V}_\tau, \mathbf{V}_{\neg\tau}) = 1 - \frac{\mathbf{V}_\tau \circ \mathbf{V}_{\neg\tau}}{||\mathbf{V}_\tau|| \cdot ||\mathbf{V}_{\neg\tau}||}, \tag{2.104}$$

where $\circ$ is the inner (dot) product of two vectors. A comparative analysis of the ORT criterion and impurity measures defined as concave-maximum criteria has been presented by Crémilleux et al. (1998).

Kolmogorov-Smirnov distance
Kolmogorov-Smirnov distance between two distributions f_1 and f_2 is defined as the maximum distance between their cumulative distribution functions F_1 and F_2:

$$D(f_1, f_2) = \max_x |F_1(x) - F_2(x)|. \tag{2.105}$$

Although in classification problems, the conditional distributions of the classes with respect to the features describing the data are usually unknown, they can be estimated on the basis of the data and used as decision rules for recursive partitioning (Friedman 1977; Utgoff and Clouse 1996). Kolmogorov-Smirnov criterion for interval valued variables has been studied by Mballo and Diday (2006). They have also compared the criterion to Gini index and entropy-based decisions.

DKM criterion
Dietterich et al. (1996) and Kearns and Mansour (1999)DKM criterion explored the properties of concave functions on [0, 1], symmetric about 0.5 with maximum value 1 for argument 0.5 and minimum value 0 at the borders of the interval. The functions can play the role of impurity measures in impurity-based split criteria

(see Eq. (2.4)). The criterion based on one of the functions, $f(q) = 2\sqrt{q(1-q)}$, has been later named a DKM criterion.

Hellinger distance

Cieslak and Chawla (2008) adapted the Hellinger distance for the purpose of DT induction (HDDT). The Hellinger distance measures the divergence between two continuous distributions P and Q with respect to a parameter λ as

$$d_H(P, Q, \lambda) = \sqrt{\int_\Omega \left(\sqrt{P} - \sqrt{Q}\right)^2 d\lambda}. \tag{2.106}$$

MAP DT model

A Bayesian approach to DT induction has been proposed by Voisine et al. (2009). Their criterion is used in an optimization process performing search for maximum a posteriori (MAP) DT model. They claim that the algorithm offers similar predictive accuracy as the state-of-the-art DT induction methods, with significantly simpler trees.

CCP

After an analysis of some weaknesses of C4.5 and CART, Liu et al. (2010) proposed new measure named *Class Confidence Proportion* (CCP) and CCPDT algorithm for learning DTs on the basis of the new criterion. To provide statistically significant decision rules, they check the significance of tree branches with Fisher's exact test to decide whether to prune them.

AID Family

One of the first DT induction algorithms with statistical foundations is *Automatic Interaction Detection* (AID) proposed by Morgan and Sonquist (1963a,b). In AID trees, at each binary split the between-group-sum-of-squares (the F statistic) is maximized for each predictor, with respect to the groups determined by the dependent variable. In the case of input variables with ordered categories (called monotonic), the splits respect the ordering, while for purely nominal predictors (called free), all possible binary splits are analyzed. This results in a bias in favor of nominal features with large numbers of possible values, as they provide more possible splits to be analyzed.

An attempt to nullify the bias brought the CHAID (*Chi-Squared Automatic Interaction Detection*) algorithm (Kass 1980). Reduction of the bias of AID was achieved by significance testing and using χ^2 statistic.

The original definition of CHAID was applicable to nominal dependent variables. An extension to ordinal target variables has been proposed by Magidson (1993). Moreover, the technique of merging predictor categories with the same prediction of the dependent variable facilitated building smaller and more comprehensible DTs.

ID3 Descendants

Especially in the earliest decades of DT research, various modifications of the ID3 algorithm have been published. An example extension is the system NewID (Niblett

1989; Boswell 1990), using information gain criterion for feature and split selection (as in ID3) and facilitating analysis of continuous attributes (with similar technique as in C4.5). According to the assumptions of Niblett (1989), the NewID system was to operate with a number of split criteria, with the possibility to select the most adequate one for the data at hand. In practice, to the best of my knowledge, it has never been accomplished.

Other interesting examples of ID3 extensions are ID4 (Schlimmer and Fisher 1986) and ID5R (Utgoff 1989, 1994) systems, facilitating incremental induction.

Miscellaneous Software

Many ideas estimated by their authors as valuable have been followed by complete software solutions. Again, it is not possible to address all of them here, but some arbitrary selection of systems, capable of DT induction, is shortly commented below.

IND package
IND is a popular system written in C and C shell languages (Buntine and Caruana 1992) and distributed as opensource. It reimplements such algorithms as ID3, C4, CART and various MML (*Minimum Message Length*) and Bayesian approaches to DT induction.

1R
An idea of very simple classifiers built from a single decision rule (1R) was presented and tested by Holte (1993). The simple rules can be seen as decision trees with just a single split (so called *decision stumps*). It turns out that for many datasets tested there, so simple method offers similar accuracy as much more complex models. Naturally, there are also many datasets for which so simple models as decision stumps are significantly less attractive than more advanced solutions.

TDDT
Top-Down Decision Trees (TDDT) is the name of DT induction algorithm implemented by Kohavi et al. (1996) as a part of MLC++ (*Machine Learning in C++*) library, that became a part of the SGI's MineSet system as *SGI MLC++*. TDDT is very similar to C4.5. The most important difference is a change in the split quality measure, which is the information gain divided by the logarithm of the number of subnodes generated by the split. The goal of the change was to remove the bias in favor of splits into many small subnodes (see Sect. 2.7 for more information on methods dealing with the bias).

SLIQ
The SLIQ algorithm (*Supervised Learning in Quest*, Quest was the name of a Data Mining project developed at IBM Almaden Research Center) was created with special emphasis on its scalability (Mehta et al. 1996). The improvement in learning time was achieved with the techniques of pre-sorting and breadth-first growth of the trees. They proposed to sort the training data items once, at the beginning of the process, according to each ordered feature, to avoid the necessity of sorting at each tree node. The algorithm deals efficiently with large disk-resident

training data, although does not get rid of the limits completely, since some data structures that must be kept in memory grow with the size of the training data. SLIQ uses Gini index for split selection and three possible methods of pruning based on the MDL principle.

SPRINT
Speed and scalability were also the most important objectives of the approach of Shafer et al. (1996), which resulted in the *SPRINT* (*Scalable PaRallelizable INduction of decision Trees*) algorithm. It was designed to remove all the memory restrictions (as compared to SLIQ, created in the same research group) and to facilitate parallelization. Shafer et al. (1996) presented both serial and parallel versions of SPRINT and the results of performance evaluation. SPRINT can be run with any impurity based split criteria.

RPart
The *RPart* package (Therneau and Atkinson 1997) is in fact a reimplementation of the ideas of CART described by Breiman et al. (1984).

RainForest
Gehrke et al. (1998, 2000) created a general framework, that can be used with many specific DT induction methods (C4.5, CART, CHAID and others) to make them fast and scalable. The authors claim that the framework offers performance improvements of over a factor of five over the SPRINT algorithm.

PUBLIC
Rastogi and Shim (2000) proposed to integrate the two stages of standard DT induction approaches, construction and pruning, in a single procedure, to fasten the induction process. The pruning procedure founded on the MDL principle makes it possible to estimate if a tree node would certainly be pruned and save some time thanks to resigning from further analysis of such a node.

BOAT
Bootstrapped Optimistic Algorithm for Tree Construction (BOATGehrke et al. 1999) is another general framework, applicable to a wide range of split selection methods. The main idea of the method is to construct an initial tree using a small subset of the training data and then refine it appropriately. It is guaranteed that the refined tree is exactly the same as the tree that would result from traditional ways of induction. The methodology has an interesting possibility of incremental update to both insertions and deletions over the training dataset.

WhiBo
WhiBo (Delibasic et al. 2011) is an open source platform for white box (component based) machine learning algorithm design. It has been created as an extension of the RapidMiner system (Mierswa et al. 2006). The goal of the framework is to facilitate algorithm design from reusable components that can be extracted from different successful algorithms. WhiBo offers a set of reusable components, including many modules for DT induction.

Miscellaneous Goals and Techniques

Even so well-defined models like decision trees can be analyzed and optimized in many various ways. Some researchers are especially interested in some particular features of the models, so they focus on these aspects and are not interested in penetration of other contexts. This usually results in general techniques, focused on particular subgoals and applicable to many kinds of components, responsible for other subgoals of DT induction.

Probability estimation
It is sometimes very important to obtain good estimates of probabilities of belonging to candidate classes, when trying to classify new data items with a DT classifier. Focus on probabilities made some authors refer to their models as *Probability Estimation Trees* (PETs). In most applications, class probabilities are derived from DTs by calculating appropriate data proportions with possible modifications like Laplace correction or m-probability-estimation (Cestnik 1990; Cestnik and Bratko 1991). Section 2.6 presents some detailed information about the most popular methods. A comparison of several algorithms with respect to probabilistic classification has been conducted by Fierens et al (2005). An approach to dealing with uncertainty during both DT induction phase and in classification with ready trees was published by Jenhani et al. (2008). They introduced a *Non-Specificity based Possibilistic Decision Tree* (NS-PDT) algorithm and its extended version, capable of building option trees. Zhang and Su (2006) analyzed probabilities from the perspective of classifiers yielding large AUC. They proposed a new AUC-based algorithm for learning conditional independence trees (CITrees).

Multivariate responses
multivariate responses In standard classification problems, there is a single discrete response variable. Some applications concern recognition of several aspects in parallel, for example in health-related problems, where several separate but correlated health problems are to be recognized. Because of correlations, it is often more appropriate to learn them together than to treat the problems as completely separate and independent. Zhang (1998) used a generalized entropy criterion and two other measures, to deal with multiple binary responses. Another set of split criteria to grow decision trees with multivariate responses has been proposed by Siciliano and Mola (2000). The new split rules were derived as extensions of criteria used in two-stage binary segmentation. An unbiased method for induction of multi-label classification trees has been presented by Noh et al (2004). As in most statistical approaches interested in unbiased feature selection, they separated the process of variable selection from the search for the best split point. They applied a test statistic to examine the equality of distributions of multi-label target variable. Another interesting generalization is the approach of Lee and Shih (2006), who realized similar ideas as in CRUISE (Kim and Loh 2001), but with the analysis of 3-dimensional contingency tables.

Constrained DT induction
Garofalakis et al. (2000, 2003) grappled with constructing decision trees satisfying additional constraints on tree size and accuracy. With such approaches, one can, for example, define the goal as obtaining properly simple trees of appropriately high accuracy. More general constraints have been analyzed by Nijssen and Fromont (2010), who proposed an adaptation of extensively studied methods from the area of local pattern mining to the field of DT induction.

Three-stage DT induction
Cappelli et al (2002) came up with an interesting idea to introduce third stage into standard DT induction processes. After tree construction and validation, they proposed procedures to guide the search for the parts of tree structure that are statistically significant, and eventually make the final tree statistically reliable.

Hybrid methods
Seewald et al. (2000) presented a hybrid learning algorithm, defined by simple modifications of C4.5 algorithm, aimed at local adjustment of inductive bias by proper selection of leaf-models. In the reduced error post-pruning of C4.5 trees, they were replacing the original leaf nodes by more sophisticated learning models like Naive Bayes or instance-based learners. Their experiments confirmed that such strategies improved performance of created models.
Generalization abilities of a classifier are much larger, when the decision borders are not restricted to perpendicular to feature space axes, as in classical DT induction approaches. Linear combinations of features are more robust, but still can not simply describe nonlinearities. Therefore, Duch and Grąbczewski (2002) proposed heterogeneous trees, that is, trees with split tests referring to distances from some points in the feature space. Naturally, the price for more flexibility is increased computational complexity of the method.
Mining data streams
Wozniak (2011) addressed the problem of learning DTs, when new data streams frequently arrive and dependencies between feature values and classes are continually changing. These are significantly different learning circumstances than in the standard approach with static training data.
Recently, Rutkowski et al. (2012) have proposed to use McDiarmid?s bound instead of the commonly used Hoeffding?s bound, to build DTs for data streams, that are almost identical with the result of standard DT induction procedures. They have examined the bounds in application to the information gain criterion and Gini index.

Uplift modeling
Rzepakowski and Jaroszewicz (2012) have explored the possibilities of using DT induction algorithms for uplift modeling, that is, prediction of changes in class probabilities caused by an action. The main goal of their approach has been to recognize data objects for which an action causes the most significant change, for example, to find out the customers that are most likely to respond to a marketing action. From another point of view, the algorithms can help in deciding which

action (of several options) to take in order to maximize profit. For this purpose, Rzepakowski and Jaroszewicz (2012) have proposed some new splitting criteria and pruning methods.

Large data analysis and induction speedup
Learning from high-dimensional data or large collections of data objects is also a very important aspect of DT induction. Some algorithms mentioned above (like SLIQ, SPRINT and RainForest) address these problems either directly or indirectly. One of the means is parallelization. Srivastava et al. (1999) described two standard methodologies: *Synchronous Tree Construction* and *Partitioned Tree Construction*, to propose a hybrid method joining the advantages of both.
Amado et al (2001) also derived their new parallel implementation of C4.5 from an analysis of different approaches to parallelization.
The CLOUDS algorithm (*Classification for Large or OUt-of-core DataSets*, Alsabti et al. 1998) samples split points of ordered features to reduce complexity of the search, which is then run in a reduced area, not exhaustively.
Beside the methods dealing with large data collections, also other techniques have been proposed to prevent too complex calculations in DT induction processes. For example, Coppersmith et al. (1999) suggested some techniques to avoid analysis of all possible partitions of nominal attributes. Because, in general, the problem of finding optimal partition is very complex, they also proposed a new heuristic search algorithm based on ordering the attribute values according to corresponding class probabilities.
Li et al. (2008) developed a clustering-based classification technique *Automatic Decision Cluster Classifier* (ADCC) for high-dimensional data. In the method, a tree clustering model is generated and Anderson-Darling test is used to automatically determine the adequate size of the resulting model. The test procedures eliminate the necessity of visual cluster validation required in a former approach of Huang et al. (2000).

2.10 Meta-Learning Germs

So far, meta-knowledge analysis has usually been performed by human experts, as it is not easy to define standard automated ways that would regularly bring valuable conclusions.

Each comparison of alternative techniques, analysis of algorithm eligibility for particular kinds of problems, and drawing conclusions about influence of learning parameters on the results can be called meta-learning.

Numerous articles discussing the subjects of various split quality measuresplit quality measures (Mingers 1989b; Buntine and Niblett 1992), different techniques of pruning methods (Quinlan 1987; Mingers 1989a; Mehta et al. 1995; Malerba et al. 1996; Esposito et al. 1997; Breslow and Aha 1997; Kononenko 1998) or advantages of using less and more complex search processes for DT induction (Quinlan and Cameron-Jones 1995; Segal 1996; Janssen and Fürnkranz 2009) are very precious for the area of meta-learning, although they should be rather treated as a prelude

to actual science of meta-learning. The early approaches point the need for meta-learning solutions, as they show that various methods outperform others in various applications, but there are no simple rules that could easily point the most successful methods for a dataset at hand.

Bias analysis approaches (Kononenko 1995; Kim and Loh 2001; Shih 2004; Lee and Shih 2006; Hothorn et al. 2006b) bring some interesting conclusions and new attractive algorithms, but they do not solve the problem of algorithm selection. Although the methods are nicely supported by theoretical reasoning, they have two major disadvantages:

1. They deal with "abstract" definitions, not too compatible with most practical applications, because the bias is defined and verified for features with no information about the target, that is, features that are completely useless for classification. It does not guarantee proper feature selection when information is present in the data. Moreover, "proper feature selection" is not even defined in such circumstances.
2. As discussed in the beginning of Chap. 2, improving feature selection at each single tree node does not guarantee the best quality of the whole resulting tree. Optimization of the whole tree is impossible even for very small structures and not too large sets of features, because the analysis of feature interaction is very complex.

Techniques used by Option Decision Trees (see Sect. 2.8.1 and Buntine 1993; Kohavi and Kunz 1997) can be certainly regarded as meta-learning germs, because the analysis on meta-level, they perform, leads to proper DT ensemble with adequately combined decisions.

Another example of incorporating meta-learning in the process of DT induction is the method of *Omnivariate Decision Trees* (Yildiz and Alpaydin 2001, 2005b; Yildiz 2011), where the kind of decision borders (axis-perpendicular, linear, nonlinear) is tuned automatically on the basis of meta-analysis.

In the area of rule induction, very close to DT induction, interesting meta-learning approaches can also be found. An example is the effort of Janssen and Fürnkranz (2007, 2008, 2010), who investigated the possibility of learning rule induction heuristic from experience. The tools used for this purpose are very similar to those of the METAL project described in more detail in Sect. 6.1.4.

References

Agresti A (1990) Categorical data analysis. John Wiley & Sons, New York

Almuallim H (1996) An efficient algorithm for optimal pruning of decision trees. Artif Intell 83(2):347–362. doi:10.1016/0004-3702(95)00060-7

Alsabti K, Ranka S, Singh V (1998) Clouds: a decision tree classifier for large datasets. Tech. rep, Electrical Engineering and Computer Science, Syracuse

Amado N, Gama J, Silva FMA (2001) Parallel implementation of decision tree learning algorithms. In: Proceedings of the 10th Portuguese conference on artificial intelligence on progress in artifi-

cial intelligence, knowledge extraction, multi-agent systems, logic programming and constraint solving. Springer, London, UK, EPIA '01, pp 6–13. http://dl.acm.org/citation.cfm?id=645378.651223
Amasyali MF, Ersoy OK (2008) Cline: a new decision-tree family. IEEE Trans. Neural Netw. 19(2):356–363
Anyanwu M, Shiva S (2009) Comparative analysis of serial decision tree classification algorithms. Int J Comput Sci Secur 3(3):230–240
Baim P (1988) A method for attribute selection in inductive learning systems. IEEE Trans Pattern Anal Mach Intell 10(6):888–896. doi:10.1109/34.9110
Bauer E, Kohavi R (1999) An empirical comparison of voting classification algorithms: bagging, boosting, and variants. Mach Learn 36:105–139. doi:10.1023/A:1007515423169
Bengio Y, Delalleau O, Simard C (2010) Decision trees do not generalize to new variations. Comput Intell 26(4):449–467. doi:10.1111/j.1467-8640.2010.00366.x
Berger J (1985) Statistical decision theory and bayesian analysis. Springer, New York
Bobrowski L (1991) Design of piecewise linear classifiers from formal neurons by a basis exchange technique. Pattern Recogn 24(9):863–870. http://www.sciencedirect.com/science/article/pii/003132039190005P
Bobrowski L (1999) Data mining procedures related to the dipolar criterion function. In: Applied stochastic models and data analysis-quantitative methods in business and industry society, Lisboa, pp 43–50
Bobrowski L (2005) Eksploracja danych oparta na wypuklych i odcinkowo-liniowych funkcjach kryterialnych. Wydawnictwo Politechniki Bialostockiej, Białystok
Bobrowski L, Krtowski M (2000) Induction of multivariate decision trees by using dipolar criteria. In: Zighed DA, Komorowski J, Zytkow JM (eds) Principles of data mining and knowledge discovery: 5th European Conference: PKDD'2000. Lecture Notes in Computer Science. Springer Verlag, Berlin, pp 331–336
Bohanec M, Bratko I (1994) Trading accuracy for simplicity in decision trees. Mach Learn 15:223–250. doi:10.1007/BF00993345
Boswell R (1990) Manual for NEWID version 2.0. Tech. rep.
Bramer M (2002) Pre-pruning classification trees to reduce overfitting in noisy domains. In: Proceedings of the third international conference on intelligent data engineering and automated learning. Springer, London, UK, IDEAL '02, pp 7–12. http://dl.acm.org/citation.cfm?id=646288.686755
Brandt S (1998) Analiza Danych. Wydawnictwo Naukowe PWN, Warszawa, tum. L. Szymanowski
Breiman L (2001) Random forests. Mach Learn 45:5–32. doi:10.1023/A:1010933404324
Breiman L, Friedman JH, Olshen A, Stone CJ (1984) Classification and regression trees. Wadsworth, Belmont
Breiman L (1996) Bagging predictors. Mach Learn 24(2):123–140
Breiman L (1998) Arcing classifiers. Ann Stat 26(3):801–849
Breslow LA, Aha DW (1997) Simplifying decision trees: a survey. Knowl Eng Rev 12(1):1–40. doi:10.1017/S0269888997000015
Brodley CE, Utgoff PE (1992a) Multivariate decision trees. Tech. Rep. 92–82, Department of Computer Science, University of Massachusetts
Brodley CE, Utgoff PE (1992b) Multivariate versus univariate decision trees. Tech. Rep. 92–8, Department of Computer Science, University of Massachusetts
Brown G, Wyatt J, Harris R, Yao X (2005) Diversity creation methods: a survey and categorisation. J Inform Fusion 6:5–20
Buntine W (1993) Learning classification trees. Stat Comput 2:63–73. doi:10.1007/BF01889584
Buntine W, Caruana R (1992) Introduction to IND version 2.1 and recursive partitioning. Tech. rep., Moffet Field, CA. http://ti.arc.nasa.gov/opensource/projects/ind/
Buntine W, Niblett T (1992) A further comparison of splitting rules for decision-tree induction. Mach Learn 8:75–85. doi:10.1007/BF00994006
Cappelli C, Mola F, Siciliano R (2002) A statistical approach to growing a reliable honest tree. Comput Stat Data Anal 38(3):285–299

Cestnik B (1990) Estimating probabilities: a crucial task in machine learning. In: Proceedings of the ninth european conference on artificial intelligence, pp 147–149

Cestnik B, Bratko I (1991) On estimating probabilities in tree pruning. In: Kodratoff Y (ed) Machine Learning - EWSL-91. Lecture Notes in Computer Science, vol 482. Springer, Berlin, pp 138–150. doi:10.1007/BFb0017010

Cherkassky V, Mulier F (1998) Learning from data. Adaptive and learning systems for signal processing, communications and control. John Wiley & Sons, Inc., New York

Cieslak D, Chawla N (2008) Learning decision trees for unbalanced data. In: Daelemans W, Goethals B, Morik K (eds) Machine learning and knowledge discovery in databases. Lecture Notes in Computer Science, vol 5211. Springer, Berlin, pp 241–256. doi:10.1007/978-3-540-87479-9_34

Coppersmith D, Hong SJ, Hosking JR (1999) Partitioning nominal attributes in decision trees. Data Mining Knowl Discov 3:197–217. doi:10.1023/A:1009869804967

Crémilleux B, Robert C, Gaio M (1998) Uncertain domains and decision trees: Ort versus c.m. criteria. In: International conference on information processing and management of uncertainty in knowledge-based systems, pp 540–546

de Mántaras RL (1991) A distance-based attribute selection measure for decision tree induction. Mach Learn 6:81–92. doi:10.1023/A:1022694001379

de Sá JPM (2001) Pattern recognition. Concepts, methods and applications. Springer, Berlin

Delibasic B, Jovanovic M, Vukicevic M, Suknovic M, Obradovic Z (2011) Component-based decision trees for classification. Intelligent Data Analysis, pp 671–693

Dietterich TG (2000) An experimental comparison of three methods for constructing ensembles of decision trees: bagging, boosting, and randomization. Mach Learn 40:139–157. doi:10.1023/A:1007607513941

Dietterich T, Kearns M, Mansour Y (1996) Applying the weak learning framework to understand and improve C4.5. In: Proceedings of the thirteenth international conference on machine learning. Morgan Kaufmann, pp 96–104

Dobra A, Gehrke J (2001) Bias correction in classification tree construction. In: Brodley CE, Danyluk AP (eds) Proceedings of the eighteenth international conference on machine learning (ICML 2001), Williams College, Williamstown, MA, USA, June 28 - July 1, 2001, Morgan Kaufmann, pp 90–97

Doetsch P, Buck C, Golik P, Hoppe N, Kramp M, Laudenberg J, Oberdörfer C, Steingrube P, Forster J, Mauser A (2009) Logistic model trees with auc split criterion for the kdd cup 2009 small challenge. J Mach Learn Res Proc Track 7:77–88

Domingos P (1996) Using partitioning to speed up specific-to-general rule induction. In: Proceedings of the AAAI-96 workshop on integrating multiple learned models. AAAI Press, pp 29–34

Dramiński M, Rada-Iglesias A, Enroth S, Wadelius C, Koronacki J, Komorowski HJ (2008) Monte carlo feature selection for supervised classification. Bioinformatics 24(1):110–117

Dramiński M, Kierczak M, Nowak-Brzezińska A, Koronacki J, Komorowski J (2011) The Monte Carlo feature selection and interdependency discovery is unbiased. Control Cybernet 40(2):199–211

Draper B, Brodley CE, Utgoff PE (1994) Goal-directed classification using linear machine decision trees. IEEE Trans Pattern Anal Mach Intell 16:888–893

Duch W, Biesiada J, Winiarski T, Grudziński K, Grąbczewski K (2002) Feature selection based on information theory filters. In: Proceedings of the international conference on neural networks and soft computing (ICNNSC 2002) Physica-Verlag (Springer). Zakopane, Advances in Soft Computing, pp 173–176

Duch W, Grąbczewski K (2002) Heterogeneous adaptive systems. In: Proceedings of the world congress of computational intelligence, Honolulu

Duch W, Winiarski T, Biesiada J, Kachel A (2003) Feature selection and ranking filters. In: Artificial neural networks and neural information processing - ICANN/ICONIP 2003, Istanbul, pp 251–254

Duda RO, Hart PE, Stork DG (2001) Pattern classification, 2nd edn. John Wiley and Sons, New York

Efron B (1983) Estimating the error rate of a prediction rule: Improvement on cross-validation. J Am Stat Assoc 78(382):316–331. http://www.jstor.org/stable/2288636
Efron B, Tibshirani R (1997) Improvements on cross-validation: The.632+ bootstrap method. J Am Stat Assoc 92(438):548–560. http://www.jstor.org/stable/2965703
Esposito F, Malerba D, Semeraro G (1997) A comparative analysis of methods for pruning decision trees. IEEE Trans Pattern Anal Mach Intell 19(5):476–491
Fayyad UM, Irani KB (1992a) The attribute selection problem in decision tree generation. In: Proceedings of the tenth national conference on artificial intelligence, AAAI'92. AAAI Press, pp 104–110
Fayyad UM, Irani KB (1992b) On the handling of continuous-valued attributes in decision tree generation. Mach Learn 8:87–102. doi:10.1007/BF00994007
Ferri C, Flach PA, Hernández-Orallo J (2002) Learning decision trees using the area under the ROC curve. In: ICML '02: Proceedings of the nineteenth international conference on machine learning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp 139–146
Ferri C, Flach P, Hernández-Orallo J (2003) Improving the auc of probabilistic estimation trees. In: Lavrac N, Gamberger D, Blockeel H, Todorovski L (eds) Machine learning: ECML 2003, Lecture Notes in Computer Science, vol 2837. Springer, Berlin, pp 121–132. doi:10.1007/978-3-540-39857-8_13
Fierens D, Ramon J, Blockeel H, Bruynooghe M (2005) A comparison of approaches for learning probability trees. In: Gama J, Camacho R, Brazdil P, Jorge AM, Torgo L (eds) Machine learning: ECML 2005. Lecture Notes in Computer Science, vol 3720. Springer, Berlin, pp 556–563. doi:10.1007/11564096_54
Fournier D, Crémilleux B (2002) A quality index for decision tree pruning. Knowl-Based Syst 15(1–2):37–43
Frank E, Witten IH (1998) Using a permutation test for attribute selection in decision trees. In: International conference on machine learning. Morgan Kaufmann, pp 152–160
Frean MR (1990) Small nets and short paths: optimising neural computation. PhD dissertation, University of Edinburgh
Freund Y, Schapire R (1997) A decision-theoretic generalization of on-line learning and an application to boosting. J Comput Syst Sci 55(1):119–139
Freund Y, Mason L (1999) The alternating decision tree learning algorithm. In: Proceedings of ICML 99, Bled, Slovenia, pp 124–133
Freund Y, Schapire R (1995) A decision-theoretic generalization of on-line learning and an application to boosting. In: Vitanyi P (ed) Computational learning theory. Lecture Notes in Computer Science, vol 904. Springer, Berlin, pp 23–37. doi:10.1007/3-540-59119-2_166
Freund Y, Schapire R (1996) Experiments with a new boosting algorithm. In: Proceedings of the thirteenth international conference on machine learning, pp 148–156
Friedman JH (1999a) Greedy function approximation: a gradient boosting machine. Tech. rep., Department of Statistics, Stanford University
Friedman JH (1999b) Stochastic gradient boosting. Comput Stat Data Anal 38:367–378
Friedman JH (1977) A recursive partitioning decision rule for nonparametric classification. IEEE Trans Comput 100(4):404–408
Gama J (1997) Probabilistic linear tree. In: ICML '97: Proceedings of the fourteenth international conference on machine learning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp 134–142
Gama J (1999) Discriminant trees. In: ICML '99: Proceedings of the sixteenth international conference on machine learning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp 134–142
Garofalakis M, Hyun D, Rastogi R, Shim K (2000) Efficient algorithms for constructing decision trees with constraints. In: Proceedings of the sixth ACM SIGKDD international conference on knowledge discovery and data mining. ACM Press, pp 335–339
Garofalakis M, Hyun D, Rastogi R, Shim K (2003) Building decision trees with constraints. Data Mining Knowl Discov 7:187–214. doi:10.1023/A:1022445500761

Gehrke J, Ganti V, Ramakrishnan R, Loh WY (1999) BOAT - optimistic decision tree construction
Gehrke J, Ramakrishnan R, Ganti V (1998) Rainforest: a framework for fast decision tree construction of large datasets. In: VLDB. Morgan Kaufmann, pp 416–427
Gehrke J, Ramakrishnan R, Ganti V (2000) Rainforest-a framework for fast decision tree construction of large datasets. Data Mining Knowl Discov 4:127–162. doi:10.1023/A:1009839829793
Gnanadesikan R (1977) Methods for statistical data analysis of multivariate observations. John Wiley, New York
Good I (1965) The estimation of probabilities. MIT Press, Cambridge
Good P (1994) Permutation tests. Springer, New York
Goodman RM, Smyth P (1988a) Decision tree design from a communication theory standpoint. IEEE Trans Inform Theory 34(5):979–994. doi:10.1109/18.21221
Goodman RM, Smyth P (1988b) An information theoretic model for rule-based expert systems. In: International symposium on information theory, Kobe, Japan
Goodman RM, Smyth P (1988c) Information theoretic rule induction. In: Proceedings of the 1988 conference on AI. Pitman Publishing, London
Grąbczewski K (2003) Zastosowanie kryterium separowalności do generowania regułklasyfikacji na podstawie baz danych. PhD thesis, Systems Research Institute, Polish Academy of Sciences, Warsaw
Grąbczewski K (2004) SSV criterion based discretization for Naive Bayes classifiers. In: Proceedings of the 7th international conference on artificial intelligence and soft computing, Zakopane, Poland
Grąbczewski K (2011) Separability of split value criterion with weighted separation gains. In: Perner P (ed) Machine learning and data mining in pattern recognition, Lecture Notes in Computer Science, vol 6871. Springer, Berlin, pp 88–98. doi:10.1007/978-3-642-23199-5_7
Grąbczewski K (2012) Decision tree cross-validation committees. Data Mining Knowl Discov, submitted. http://www.is.umk.pl/kg/papers/12-DTCVComm.pdf
Grąbczewski K, Duch W (1999) A general purpose separability criterion for classification systems. In: Proceedings of the 4th conference on neural networks and their applications, Zakopane, Poland, pp 203–208
Grąbczewski K, Duch W (2000) The separability of split value criterion. In: Proceedings of the 5th conference on neural networks and their applications, Zakopane, Poland, pp 201–208
Grąbczewski K, Duch W (2002a) Forests of decision trees. In: Proceedings of international conference on neural networks and soft computing, Physica-Verlag (Springer), Advances in Soft Computing, pp 602–607
Grąbczewski K, Duch W (2002b) Heterogeneous forests of decision trees. In: Proceedings of international conference on artificial neural networks. Lecture Notes in Computer Science, vol 2415. Springer, pp 504–509
Grąbczewski K, Jankowski N (2005) Feature selection with decision tree criterion. In: Nedjah N, Mourelle L, Vellasco M, Abraham A, Köppen M (eds) Fifth international conference on hybrid intelligent systems. IEEE, Computer Society, Rio de Janeiro, Brazil, pp 212–217
Grąbczewski K, Jankowski N (2006) Mining for complex models comprising feature selection and classification. In: Guyon I, Gunn S, Nikravesh M, Zadeh L (eds) Feature extraction, foundations and applications. Springer, Berlin, pp 473–489
Green DM, Swets JA (1966) Signal detection theory and psychophysics. John Wiley, New York
Guo H, Gelfand SB (1992) Classification trees with neural network feature extraction. IEEE Trans Neural Netw 3(6):923–933
Hand DJ, Till RJ (2001) A simple generalisation of the area under the ROC curve for multiple class classification problems. Mach Learn 45(2):171–186. doi:10.1023/A:1010920819831
Hartigan JA, Wong MA (1979) Algorithm as 136: a k-means clustering algorithm. J R Stat Soc Ser C (Appl Stat) 28(1):100–108
Hastie T, Tibshirani R (1990) Generalized additive models. Chapman and Hall, London
Hawkins DM (1999) FIRM: formal inference-based recursive modeling. Tech. Rep. 546, School of Statistics, University of Minnesota

Heath D, Kasif S, Salzberg S (1993) Induction of oblique decision trees. J Artif Intell Res 2(2):1–32
Holte RC (1993) Very simple classification rules perform well on most commonly used datasets. Mach Learn 11:63–91
Hothorn T, Hornik K, Wiel MAVD, Zeileis A (2006a) A lego system for conditional inference. Am Stat 60:257–263
Hothorn T, Hornik K, Zeileis A (2006b) Unbiased recursive partitioning: a conditional inference framework. J Comput Graph Stat 15(3):651–674
Hothorn T, Hornik K, van de Wiel MA, Zeileis A (2008) Implementing a class of permutation tests: the coin package. J Stat Softw 28(8):1–23. http://www.jstatsoft.org/v28/i08
Hothorn T, Hornik K, Zeileis A (2004) Unbiased recursive partitioning: a conditional inference framework. Research Report Series 8, Department of Statistics and Mathematics, Institut für Statistik und Mathematik, WU Vienna University of Economics and Business, Vienna
Huang Z, Ng MK, Lin T, Cheung DWL (2000) An interactive approach to building classification models by clustering and cluster validation. In: Proceedings of the second international conference on intelligent data engineering and automated learning, data mining, financial engineering, and intelligent agents. Springer, London, UK, IDEAL '00, pp 23–28. http://dl.acm.org/citation.cfm?id=646287.688767
Huber PJ (1977) Robust statistical procedures. Society for Industrial and Applied Mathematics, Pittsburgh
Hubert L, Arabie P (1985) Comparing partitions. J Classif 2:193–218. doi:10.1007/BF01908075
Janssen F, Fürnkranz J (2007) On meta-learning rule learning heuristics. In: ICDM, pp 529–534
Janssen F, Fürnkranz J (2008) An empirical comparison of hill-climbing and exhaustive search in inductive rule learning
Janssen F, Fürnkranz J (2008) An empirical investigation of the trade-off between consistency and coverage in rule learning heuristics. In: Discovery Science, pp 40–51
Janssen F, Fürnkranz J (2009) A re-evaluation of the over-searching phenomenon in inductive rule learning. In: Proceedings of the SIAM international conference on data mining (SDM-09), pp 329–340
Janssen F, Fürnkranz J (2010)On the quest for optimal rule learning heuristics. Mach Learn 78:343–379. doi:10.1007/s10994-009-5162-2
Jenhani I, Amor NB, Elouedi Z (2008) Decision trees as possibilistic classifiers. Int J Approx Reason 48(3):784–807. doi:10.1016/j.ijar.2007.12.002
John GH (1995a) Robust decision trees: removing outliers in databases. In: First international conference on knowledge discovery and data mining. AAAI Press, Menlo Park, CA, pp 174–179
John GH (1995b) Robust linear discriminant trees. In: AI&Statistics-95 [7]. Springer-Verlag, pp 285–291
John GH (1996) Robust linear discriminant trees. In: Fisher D, Lenz H (eds) Learning from data: artificial intelligence and statistics V. Lecture Notes in Statistics, Springer-Verlag, New York, chap 36:375–385
Kass GV (1980) An exploratory technique for investigating large quantities of categorical data. Appl Stat 29:119–127
Kearns M, Mansour Y (1999) On the boosting ability of top-down decision tree learning algorithms. J Comput Syst Sci 58(1):109–128
Kim JH (2009) Estimating classification error rate: Repeated cross-validation, repeated hold-out and bootstrap. Comput Stat Data Anal 53(11):3735–3745. http://ideas.repec.org/a/eee/csdana/v53y2009i11p3735-3745.html
Kim H, Loh WY (2001) Classification trees with unbiased multiway splits. J Am Stat Assoc 96:589–604. http://www.stat.wisc.edu/loh/treeprogs/cruise/cruise.pdf
Kim H, Loh WY (2003) Classification trees with bivariate linear discriminant node models. J Comput Graph Stat 12:512–530. http://www.stat.wisc.edu/loh/treeprogs/cruise/jcgs.pdf
Kira K, Rendell LA (1992a) The feature selection problem: traditional methods and a new algorithm. In: Proceedings of the national conference on artificial intelligence. John Wiley & Sons Ltd, pp 129–134

Kira K, Rendell LA (1992b) A practical approach to feature selection. In: ML92: Proceedings of the ninth international workshop on machine learning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp 249–256

Kohavi R, Kunz C (1997) Option decision trees with majority votes. In: Proceedings of the fourteenth international conference on machine learning, pp 161–169

Kohavi R, Sommerfield D, Dougherty J (1996) Data mining using MLC++: a machine learning library in C++. In: Tools with artificial intelligence. IEEE Computer Society Press, pp 234–245. http://www.sgi.com/tech/mlc

Kononenko I (1994) Estimating attributes: analysis and extensions of RELIEF. In: European conference on machine learning. Springer, pp 171–182

Kononenko I (1995) On biases in estimating multi-valued attributes. In: Proceedings of the 14th international joint conference on artificial intelligence, IJCAI'95, vol 2. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp 1034–1040. http://dl.acm.org/citation.cfm?id=1643031.1643034

Kononenko I (1998) The minimum description length based decision tree pruning. In: Lee HY, Motoda H (eds) PRICAI'98: topics in artificial intelligence. Lecture Notes in Computer Science, vol 1531. Springer Berlin, pp 228–237

Kotsiantis S (2011) Decision trees: a recent overview. Artif Intell Rev 35:1–23. doi:10.1007/s10462-011-9272-4

Koziol JA (1991) On maximally selected chi-square statistics. Biometrics 47(4):1557–1561. URL http://www.jstor.org/stable/2532406

Kuncheva LI, Whitaker CJ (2002) Using diversity with three variants of boosting: aggressive, conservative, and inverse. In: Roli F, Kittler J (eds) Multiple classifier systems. Lecture Notes in Computer Science, vol 2364. Springer, Berlin, pp 81–90. doi:10.1007/3-540-45428_48

Lee JY, Olafsson S (2006) Multi-attribute decision trees and decision rules. In: Triantaphyllou E, Felici G (eds) Data mining and knowledge discovery approaches based on rule induction techniques, massive computing, vol 6. Springer US, pp 327–358. Doi:10.1007/0-387-34296-6_10

Lee TH, Shih YS (2006) Unbiased variable selection for classification trees with multivariate responses. Comput Stat Data Anal 51(2):659–667

Levene H (1960) Robust tests for equality of variances. In: Olkin I (ed) Contributions to probability and statistics. Stanford University Press, Palo Alto, pp 278–292

Li Y, Hung E, Chung K, Huang J (2008) Building a decision cluster classification model for high dimensional data by a variable weighting k-means method. In: Proceedings of the twenty-first Australasian joint conference on artificial intelligence, Auckland, pp 337–347

Lim TS, Loh WY, Shih YS (2000) A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms. Mach Learn 40:203–228

Liu W, Chawla S, Cieslak DA, Chawla NV (2010) A robust decision tree algorithm for imbalanced data sets. In: SDM, SIAM, pp 766–777. http://dblp.uni-trier.de/rec/bibtex/conf/sdm/LiuCCC10

Loh WY (2002) Regression trees with unbiased variable selection and interaction detection. Stati Sin 12:361–386. http://www3.stat.sinica.edu.tw/statistica/j12n2/j12n21/j12n21.htm

Loh WY, Vanichsetakul N (1988) Tree-structured classification via generalized discriminant analysis (with discussion). J Am Stat Assoc 83:715–728

Loh WY, Shih YS (1997) Split selection methods for classification trees. Stat Sin 7:815–840

Magidson J (1993) The use of the new ordinal algorithm in chaid to target profitable segments. J Database Market 1:29–48

Malerba D, Esposito F, Semeraro G (1996) A further comparison of simplification methods for decision-tree induction. In: In Fisher D, Lenz H (eds) Learning. Springer, Berlin, pp 365–374

Mballo C, Diday E (2006) The criterion of kolmogorov-smirnov for binary decision tree: application to interval valued variables. Intell Data Anal 10(4):325–341

Mehta M, Agrawal R, Rissanen J (1996) SLIQ: a fast scalable classifier for data mining. In: Proceedings of the 5th international conference on extending database technology: advances in database technology. Springer, London, UK, EDBT '96, pp 18–32. URL http://dl.acm.org/citation.cfm?id=645337.650384

Mehta M, Rissanen J, Agraval R (1995) MDL-based decision tree pruning. In: Fayyad U, Uthurusamy R (eds) Proceedings of the first international conference on knowledge discovery and data mining. AAAI Press, Menlo Park, CA, pp 216–221

Melville P, Mooney RJ (2003) Constructing diverse classifier ensembles using artificial training examples. In: Proceedings of the eighteenth international joint conference on artificial intelligence, pp 505–510

Michie D (1990) Personal models of rationality. J Stat Plan Infer 25(3):381–399. http://www.sciencedirect.com/science/article/B6V0M-45SJDGS-F/1/17548ffdb8fe70dfd840185272bdbcdf

Michie D, Spiegelhalter DJ, Taylor CC (1994) Machine learning, neural and statistical classification. Elis Horwood, London

Mierswa I, Wurst M, Klinkenberg R, Scholz M, Euler T (2006) Yale: Rapid prototyping for complex data mining tasks. In: Ungar L, Craven M, Gunopulos D, Eliassi-Rad T (eds) Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, KDD '06. ACM, New York, NY, USA, pp 935–940. URL http://rapid-i.com/component/option,com_docman/task,doc_download/gid,25/Itemid,62

Mingers J (1989a) An empirical comparison of pruning methods for decision tree induction. Mach Learn 4(2):227–243

Mingers J (1989b) An empirical comparison of selection measures for decision-tree induction. Mach Learn 3:319–342

Mitchell T (1997) Machine learning. McGraw Hill, New York

Morgan JN, Sonquist JA (1963a) Problems in the analysis of survey data, and a proposal. J Am Stat Assoc 58(302):415–434. http://www.jstor.org/stable/2283276

Morgan JN, Sonquist JA (1963b) Some results from a non-symmetrical branching process that looks for interaction effects. In: Proceedings of the social statistics section. American Statistical Association, pp 40–53

Müller W, Wysotzki F (1994) Automatic construction of decision trees for classification. Ann Oper Res 52:231–247

Müller W, Wysotzki F (1997) The decision-tree algorithm CAL5 based on a statistical approach to its splitting algorithm. Machine learning and statistics: the interface, pp 45–65

Murthy SK (1997) On growing better decision trees from data. PhD thesis, The Johns Hopkins University, Baltimore, MD

Murthy SK (1998) Automatic construction of decision trees from data: a multi-disciplinary survey. Data Mining Knowl Discov 2:345–389. doi:10.1023/A:1009744630224

Murthy SK, Salzberg S (1995) Lookahead and pathology in decision tree induction. In: Proceedings of the 14th international joint conference on artificial intelligence. Morgan Kaufmann, pp 1025–1031

Murthy SK, Kasif S, Salzberg S (1994) A system for induction of oblique decision trees. J Artif Intell Res 2:1–32

Murthy S, Kasif S, Salzberg S, Beigel R (1993) Oc1: randomized induction of oblique decision trees. In: AAAI'93, pp 322–327

Nettleton D, Banerjee T (2001) Testing the equality of distributions of random vectors with categorical components. Comput Stat Data Anal 37(2):195–208. http://www.sciencedirect.com/science/article/pii/S0167947301000159

Niblett T (1989) Functional specification for realid. Tech. rep.

Niblett T, Bratko I (1986) Learning decision rules in noisy domains. In: Proceedings of expert systems '86, the 6th annual technical conference on research and development in expert systems III. Cambridge University Press, New York, NY, USA, pp 25–34

Nijssen S, Fromont E (2010) Optimal constraint-based decision tree induction from itemset lattices. Data Mining Knowl Discov 21:9–51. doi:10.1007/s10618-010-0174-x

Noh HG, Song MS, Park SH (2004) An unbiased method for constructing multilabel classification trees. Comput Stat Data Anal 47(1):149–164. http://www.sciencedirect.com/science/article/pii/S0167947303002433

Oates T, Jensen D (1999) Toward a theoretical understanding of why and when decision tree pruning algorithms fail. In: Proceedings of the sixteenth national conference on artificial intelligence and the eleventh innovative applications of artificial intelligence conference innovative applications of artificial intelligence. American Association for Artificial Intelligence, Menlo Park, CA, USA, AAAI '99/IAAI '99, pp 372–378. http://dl.acm.org/citation.cfm?id=315149.315327

O'Keefe RA (1983) Concept formation from very large training sets. In: Proceedings of the eighth international joint conference on Artificial intelligence, vol 1. Morgan Kaufmann, San Francisco, CA, USA, IJCAI'83. pp 479–481, http://dl.acm.org/citation.cfm?id=1623373.1623490

Oliveira A, Sangiovanni-Vincentelli A, Shavlik J (1996) Using the minimum description length principle to infer reduced ordered decision graphs. In: Machine Learning, pp 23–50

Oza NC (2003) Boosting with averaged weight vectors. In: Proceedings of the 4th international conference on multiple classifier systems. Springer, Berlin, MCS'03, pp 15–24. http://dl.acm.org/citation.cfm?id=1764295.1764299

Parmanto B, Munro PW, Doyle HR (1995) Improving committee diagnosis with resampling techniques. In: NIPS, pp 882–888

Piccarreta R (2008) Classification trees for ordinal variables. Comput. Stat. 23:407–427. doi:10.1007/s00180-007-0077-5

Provost F, Kolluri V (1999) A survey of methods for scaling up inductive algorithms. Data Mining Knowl Discov 3:131–169. doi:10.1023/A:1009876119989

Quinlan JR, Cameron-Jones RM (1995) Oversearching and layered search in empirical learning. In: IJCAI, pp 1019–1024

Quinlan JR (1987) Simplifying decision trees. Int J Man-Mach Stud 27(3):221–234. doi:10.1016/S0020-7373(87)80053-6

Quinlan JR (1993) C 4.5: programs for machine learning. Morgan Kaufmann, San Mateo

Quinlan JR (1996) Bagging, boosting, and C4.5. In: Proceedings of the thirteenth national conference on artificial intelligence and eighth innovative applications of artificial intelligence conference, AAAI 96, IAAI 96, vol 1. AAAI Press/The MIT Press, Portland, Oregon, pp 725–730

Quinlan JR (1986) Induction of decision trees. Mach Learn 1:81–106

Quinlan JR, Rivest RL (1989) Inferring decision trees using the minimum description length principle. Inform Comput 80(3):227–248

Rastogi R, Shim K (2000) Public: a decision tree classifier that integrates building and pruning. Data Mining Knowl Discov 4:315–344. doi:10.1023/A:1009887311454

Rokach L, Maimon O (2008) Data mining with decision trees: theory and applications. World Scientific, Singapore

Rokach L, Maimon O (2010) Classification trees. In: Maimon O, Rokach L (eds) Data mining and knowledge discovery handbook. Springer US, pp 149–174. doi:10.1007/978-0-387-09823-4-9

Rutkowski L, Pietruczuk L, Duda P, Jaworski M (2012) Decision trees for mining data streams based on the McDiarmid's bound. IEEE Trans Knowl Data Eng PP(99):1–14

Rzepakowski P, Jaroszewicz S (2012) Decision trees for uplift modeling with single and multiple treatments. Knowl Inform Syst 32(2):303–327

Safavian S, Landgrebe D (1991) A survey of decision tree classifier methodology. IEEE Trans Syst Man Cybernet 21(3):660–674. doi:10.1109/21.97458

Schapire RE, Singer Y (1999) Improved boosting algorithms using confidence-rated predictions. Mach Learn 37(3):297–336

Schlimmer JC, Fisher D (1986) A case study of incremental concept induction. In: Proceedings of the fifth national conference on artificial intelligence. Morgan Kaufmann, Philadelphia, PA, pp 496–501

Seewald AK, Petrak J, Widmer G (2000) Hybrid decision tree learners with alternative leaf classifiers: an empirical study. In: Proceedings of the 14th FLAIRS conference. AAAI Press, pp 407–411

Segal R (1996) An analysis of oversearch. Unpublished manuscript

Shafer JC, Agrawal R, Mehta M (1996) SPRINT: a scalable parallel classifier for data mining. In: Proceedings of the 22nd international conference on very large data bases. Morgan Kaufmann

Publishers Inc., San Francisco, CA, USA, VLDB '96, pp 544–555. http://dl.acm.org/citation.cfm?id=645922.673491
Shih YS (1999) Families of splitting criteria for classification trees. Stat Comput 9:309–315. doi:10.1023/A:1008920224518
Shih YS (2004) A note on split selection bias in classification trees. Comput Stat Data Anal 45:457–466
Siciliano R, Mola F (2000) Multivariate data analysis and modeling through classification and regression trees. Comput Stat Data Anal 32(3–4):285–301. http://www.sciencedirect.com/science/article/pii/S0167947399000821
Smyth P, Goodman RM (1992) An information theoretic approach to rule induction from databases. IEEE Trans Knowl Data Eng 4(4):301–316
Snedecor GW, Cochran W (1989) Statistical Methods. No. 276 in Statistical Methods, Iowa State University Press
Srivastava A, Han EH, Kumar V, Singh V (1999) Parallel formulations of decision-tree classification algorithms. Data Mining Knowl Discov 3:237–261. doi:10.1023/A:1009832825273
Strasser H, Weber C (1999) On the asymptotic theory of permutation statistics. Math Meth Stat 2:220–250
Strobl C, Boulesteix AL, Augustin T (2005) Unbiased split selection for classification trees based on the gini index. Ludwig-Maximilian University, Munich, Tech. rep.
Tadeusiewicz R, Izworski A, Majewski J (1993) Biometria. Wydawnictwa AGH, Kraków
Therneau TM, Atkinson EJ (1997) An introduction to recursive partitioning using the RPART routines. Tech. rep., Divsion of Biostatistics 61, Mayo Clinic
Torres-Sospedra J, Hernández-Espinosa C, Fernández-Redondo M (2007) Averaged conservative boosting: introducing a new method to build ensembles of neural networks. In: de Sá J, Alexandre L, Duch W, Mandic D (eds) Artificial neural networks - ICANN 2007. Lecture Notes in Computer Science, vol 4668. Springer, Berlin, pp 309–318
Utgoff PE (1989) Incremental induction of decision trees. Mach Learn 4:161–186. Doi:10.1023/A:1022699900025
Utgoff PE (1994) An improved algorithm for incremental induction of decision trees. In: Proceedings of the eleventh international conference on machine learning. Morgan Kaufmann, pp 318–325
Utgoff PE, Brodley CE (1991) Linear machine decision trees. Tech. Rep. UM-CS-1991-010, Department of Computer Science, University of Massachusetts
Utgoff PE, Clouse JA (1996) A Kolmogorov-Smirnoff metric for decision tree induction. Tech. rep., University of Massachusetts, Amherst, MA, USA
Vinh NX, Epps J, Bailey J (2010) Information theoretic measures for clusterings comparison: variants, properties, normalization and correction for chance. J Mach Learn Res 11:2837–2854. http://dl.acm.org/citation.cfm?id=1756006.1953024
Voisine N, Boullé M, Hue C (2009) A bayes evaluation criterion for decision trees. In: Advances in knowledge discovery and management (AKDM09)
Wallace C, Patrick J (1993) Coding decision trees. Mach Learn 11:7–22. doi:10.1023/A:1022646101185
Wang H, Zaniolo C (2000) CMP: a fast decision tree classifier using multivariate predictions. In: Proceedings of the 16th international conference on data engineering, pp 449–460
White AP, Liu WZ (1994) Bias in information-based measures in decision tree induction. Mach Learn 15:321–329. doi:10.1023/A:1022694010754
Wilson EB (1927) Probable inference, the law of succession, and statistical inference. J Am Stat Assoc 22:209–212
Wozniak M (2011) A hybrid decision tree training method using data streams. Knowl Inform Syst 29:335–347. doi:10.1007/s10115-010-0345-5
Yildiz OT, Alpaydin E (2000) Linear discriminant trees. In: ICML '00: Proceedings of the seventeenth international conference on machine learning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp 1175–1182

Yildiz OT, Alpaydin E (2005a) Linear discriminant trees. Int J Pattern Recogn Artif Intell 19(3):323–353

Yildiz OT, Alpaydin E (2005b) Model selection in omnivariate decision trees. In: 16th European conference on machine learning, Porto, Portugal, pp 473–484

Yildiz OT, Alpaydin E (2001) Omnivariate decision trees. IEEE Trans Neural Netw 12(6):1539–1546

Yildiz OT (2011) Model selection in omnivariate decision trees using structural risk minimization. Inform Sci 181:5214–5226

Zeileis A, Hothorn T, Hornik K (2008) Model-based recursive partitioning. J Comput Graph Stat 17(2):492–514. http://statmath.wu.ac.at/zeileis/papers/Zeileis+Hothorn+Hornik-2008.pdf

Zenobi G, Cunningham P (2001) Using diversity in preparing ensembles of classifiers based on different feature seature subsets to minimize generalization error. In: Lecture Notes in Computer Science. Springer, pp 576–587

Zhang H (1998) Classification trees for multiple binary responses. J Am Stat Assoc 93(441):180–193. http://www.jstor.org/stable/2669615

Zhang H, Su J (2006) Learning probabilistic decision trees for auc. Pattern Recogn Lett 27(8):892–899. ROC Analysis in Pattern Recognition. http://www.sciencedirect.com/science/article/pii/S0167865505003065

Chapter 3
Unified View of Decision Tree Induction Algorithms

A thorough analysis of all the algorithms described in the preceding chapter (and also some other less popular approaches) has brought numerous conclusions on their similarities and differences. The conclusions have resulted in the uniform view of DT induction algorithms described here. This chapter is completely different in form and substance from the previous one. It can be seen like a partial report of object-oriented design and functionality specification of particular modules, created for universal modeling of different DT induction algorithms. Common functionality modules have been extracted from many induction methods and encapsulated to provide maximum possibilities and reduce communication between modules and necessary minimum.

From the topmost point of view, the tasks related to building decision tree models can be split into two separate groups:

- algorithms of *tree construction*,
- methods of *tree refinement* that are applied on top of different tree construction algorithms, including various techniques of post-pruning and approaches like iterative refiltering.

The groups are discussed separately in the following subsections.

3.1 Decision Tree Construction

Following the top-down approach of object-oriented analysis and design, we can determine components of the tree construction algorithms:

- *search method* that composes the tree node by node,
- *node splitter*—a procedure responsible for splitting the nodes,
- *stop criterion*—the rule that stops the search process,
- *split acceptor*—the rule that decides whether to accept the best split of a node or to make it a leaf,

K. Grąbczewski, *Meta-Learning in Decision Tree Induction*,
Studies in Computational Intelligence 498, DOI: 10.1007/978-3-319-00960-5_3,

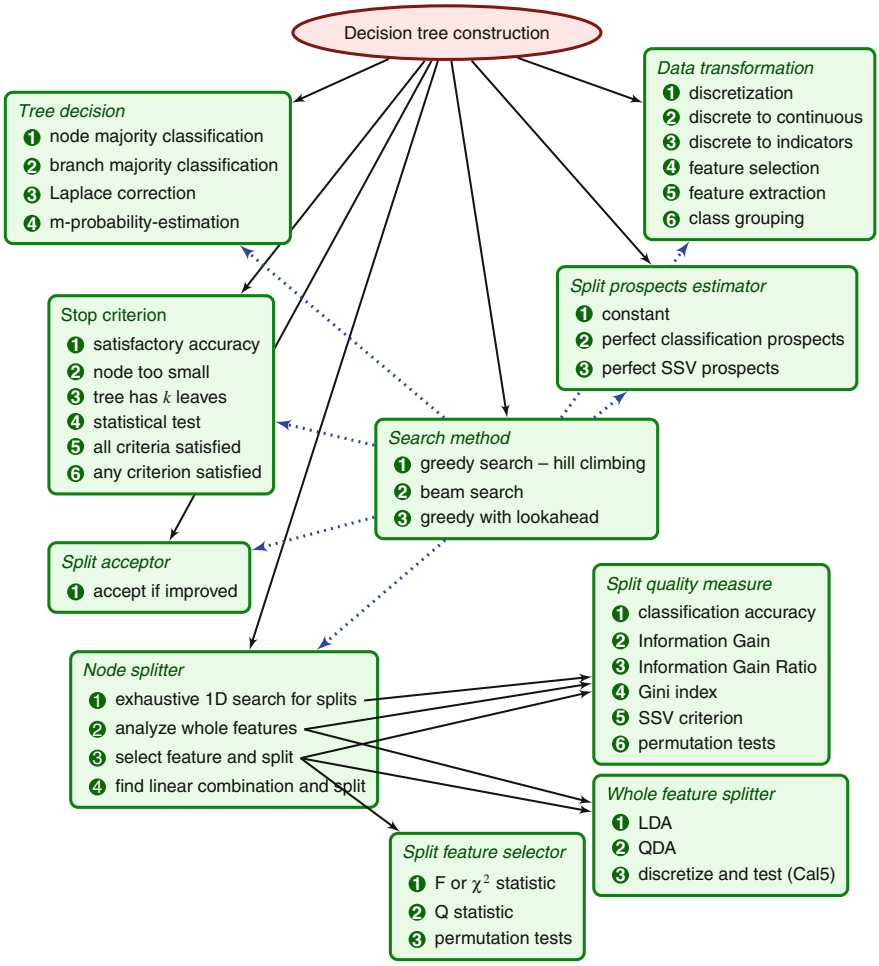


Fig. 3.1 Information flow between DT construction algorithm modules

- *split prospects estimator*—a procedure that defines the order in which the nodes of the tree are split—when using some stop criteria, the order of splitting nodes may be very important, but in most cases it is irrelevant,
- *decision making module* which provides decisions for data items on the basis of the tree,
- optional *data transformations* that prepare the training data at the start of the process or convert the parts of data at particular nodes.

A sketch of dependencies between the modules is presented in Fig. 3.1. It can be viewed in terms of classes and interfaces: each box represents an interface and enumerates some classes implementing the interface. Although one could regard UML diagrams as more appropriate form of presentation for the object oriented

design of the system, this form is not used here, because it would take much more space and would be less readable. In the figure, the solid arrows represent submodule relation while the dotted ones show the modules used by the search process.

Such amount of extracted interfaces and their implementations as depicted in Fig. 3.1 is satisfactory for building quite large number of DT induction algorithms including the most popular ones and many others.

3.1.1 Search Strategies

The search process is the main part of each tree construction algorithm. Its fundamental function is performed by a method *Search()* described in the interface 3.1. It uses the modules passed as parameters to grow the tree on the basis of given dataset *D*. Each parameter has its unique functionality and is responsible for some parts of the DT induction process. The roles of particular modules/parameters are the following:

Interface 3.1 (Search strategies interface)

***Method:* Search(D, NodeT, NS, SPE, SC, SA, DM)**
Input: *Training dataset D, data transformation NodeT, node splitter NS, split prospects estimator SPE, stop criterion SC, split acceptor SA, decision maker DM*
Output: *Decision tree.*

- Data transformation *NodeT* may be used at each node to somehow prepare the node data before the split. It may be a way to implement the technique of LTree family of algorithms, where new features are generated at each node and then analyzed with a split quality measure like information gain. It can also simplify the data at particular node after the split of its parent—for example, a feature can be deleted, when it is useless, because all data vectors share a single value of that feature. Most algorithms do not use this component.
- Node splitter *NS* is called at each node to determine the best splits of subsequent tree nodes.
- Split prospects estimator *SPE* rarely affects the resulting tree. Many algorithms do not use such component at all. If used, it defines the order in which the nodes are split. The order may be important when using a stop criterion that acts globally, for example, sends a stop signal when the tree gets a predefined size.
- Stop criterion *SC* decides when to stop further growth of the tree. Usually, further splits are rejected when the nodes are pure enough or get too small.
- Split acceptor *SA* is a module used by algorithms, which decide a posteriori, whether the split should be accepted or not: first, the split is determined, and then accepted or withdrawn (if withdrawn then the node is converted back into a leaf).

- Decision module *DM* of the tree does not drive the search process, and its presence among the modules required during search processes may surprise, as it is rather related to using a ready tree, not tree construction. The reason of passing it to the search process is that from technical point of view, cooperation between the two modules may be very advantageous. For example, the decision module may prepare some information for further decision making just in time when the information is available, that is, during the search process. When such information is extracted, some data like the training datasets of particular nodes may be discarded from memory, which improves the efficiency of the process.

As described in Sect. 2.5, almost all approaches to DT induction use the same search method: they split the nodes recursively starting with the root node, down to the leaves. For better understanding of the general scheme of cooperation between the search process and other modules, algorithm 3.2 presents the implementation of the *Search()* method of the standard top-down DT induction.The code has been reformatted to get rid of programming language specifics and non-crucial technical stuff. Although not very short, the algorithm should be readable without the need of great programming skills. It shows how particular modules can be arranged to perform hill-climbing search. The implementation is very precise but still very flexible, because separation of particular functionalities facilitates using different implementations of the interfaces by simply passing appropriate objects as the arguments. As a result, the same (configurable) algorithm is the base of all classical top-down splitting DT algorithms.

Recursive nature of greedy search procedure has been replaced by iterative one, based on a queue of nodes awaiting a split. The queue is not a classical LIFO (last-in-first-out) structure, but an ordered collection. Most DT induction approaches do not need the order, but in some cases, we want to make the most improving splits first and to stop before the whole tree is built, hence the order of splitting is important. For this purpose, the (rarely used) split prospects estimator object has been introduced. Its goal is to measure potential separability power of nodes, so that the most promising ones are split first. One could reproach the solution for wasting time on order management, since in most cases it is not necessary, but it needs to be pointed out that properly implemented heap structure allows to add and remove elements in constant time, when all keys are equal.

At start, the root node of the tree is created and added to the queue (item 3). Then, the most promising node is repeatedly popped from the queue, until the queue gets empty. When popped out, each node is analyzed and if the stop criterion is not satisfied (item 4c), the optimal split is determined (item 4e). After optional verification of split acceptability (item 4(f)iii), each child node is pushed to the queue for further analysis (item 4(f)iv).

Getting the best split from the node splitter is performed with a call to *BestSplits()* function (item 4e), which receives, maybe unexpectedly, two parameters (here 1 is passed as the second one). The second parameter tells how many best splits should be returned. In hill climbing, it is set to 1, but other methods may need more than one candidate split, so the interface of node splitters is ready for that.

Algorithm 3.2 (Hill-climbing search for DT)

Prototype: ***Search(D, NodeT, NS, SPE, SC, SA, DM)***
Input: *Training dataset D, data transformation NodeT, node splitter NS, split prospects estimator SPE, stop criterion SC, split acceptor SA, decision maker DM.*
Output: *Decision tree.*
The algorithm:

1. *tree* ← *new tree for D with root only*
2. *queue* ← *empty ordered queue of candidates to split*
3. *queue.Add(SPE.SplitProspects(tree.Root, D), tree.Root)*
4. ***while*** *queue is not empty* ***do***
   a. *node* ← *node from queue of maximum estimated prospects*
   b. *node.trainingInfo* ← *DM.TrainingInfo(node, node data)*
   c. ***if*** *SC.ShouldStop(node, node data, tree)* ***then*** *continue with next iteration*
   d. *D* ← ***if*** *NodeT is given* ***then*** *NodeT.Transform(node data)* ***else*** *node data*
   e. *split* ← *NS.BestSplits(D, 1)*
   f. ***if*** *split* $\neq \perp$ ***then***
      i. *children* ← *perform the split for D*
      ii. ***if*** *children* $= \emptyset$ ***then*** *continue with next iteration*
      iii. ***if*** *SA is given and not SA.SplitAcceptable(node, tree, D, children data)* ***then*** *convert node to a leaf and continue with next iteration*
      iv. ***for each*** *child* $\in$ *children* ***do*** */* enqueue the children */* *queue.Add(SPE.SplitProspects(child, child data), child)*
5. ***return*** *tree*

As mentioned above, the proper action of decision maker takes place after the tree is ready, but during the construction phase, it may be advantageous to prepare some data for further actions and to help in saving memory during the induction process.

A data transformation may be run just before each node is analyzed in the pursuit of the best splits (item 4d). It facilitates adding many interesting possibilities of the general algorithm.

Other search methods, like beam search, are implemented as other implementations of the search method interface. The lookahead search for DTs can be obtained by the hill-climbing method run with proper node splitter, capable of the lookahead functionality. Authors of new search methods should focus on the idea of the search and use the other interfaces to run their functionality, as it is usually done in object oriented programming.

3.1.2 Node Splitters

Node splitting methods are probably the most diverse among the components of DT induction algorithms. The literature offers so many examples that Fig. 3.1 can hold just a very short excerpt of the list of solutions. General specification of their

functionality is quite simple, as presented in the listing of interface 3.3. It contains just one method that, given a data sample D and a number n of splits to be returned, examines the data and returns up to n splits selected as the best ones found. Parameter n defines the maximum number of splits to be returned. The result collection may be smaller when the method is not capable of finding n splits.

Interface 3.3 (Node splitter interface)

Method: BestSplits(D, n)
Input: *Dataset to be split D, the count* n *of best splits to return.*
Output: *A collection of candidate splits.*

To realize the methods described in Chap. 2, four kinds of implementations of the node splitter interface have been defined:

Exhaustive one-dimensional search
The most often used strategy is to test all possible splits (within a specified family) and select the best ones. It has been applied in algorithms like CART, C4.5, SSV, LMDT, OC1 and many others. Continuous features are usually split into two intervals by a single threshold value. Unordered ones can be split into as many subnodes as the number of values they possibly have. In utter approaches, binary splits of unordered features are obtained by checking all possible subsets of the set of feature symbols. Less extreme methods use some heuristics to restrict the number of subsets to analyze.

Select feature and split
Algorithms like Cal5, CTree and the family of FACT, QUEST and CRUISE, separate split selection into two stages: feature selection and split selection. This technique is aimed at minimizing bias in feature selection and reduction of computational complexity of single split determination.

Find features combination and split
Some oblique DT induction methods (for example OC1) generate splits by extracting new features as combinations of the original ones, and using them for the splits.

Whole feature analysis
Yet another possibility is to generate one split for each feature and then compare the results to find out the best one. Such solution is conceptually and computationally intermediate between the exhaustive search and the methods splitting only one feature selected with separate tools. An option of Cal5 performing entropy based feature selection worked in this way, but also each other method of separate feature selection and split procedures can be modified to do so.

Each node splitting method can be further analyzed to extract some separate ideas, define interfaces and their implementations. Three such examples are included in Fig. 3.1. Methods of the split quality measures group can cooperate with a general exhaustive search procedure by taking responsibility for estimation of each single

split quality. Routines of "whole feature analysis" and "separate feature selection and split" have also been designed as general methods using subroutines focused on particular splitting. The feature selection part is also done by separate, specialized modules (different for FACT, QUEST and so on).

3.1.3 Stop Criteria

Before splits are searched for, it can be verified if the efforts should be undertaken. Implementations of stop criteria (interface 3.4) must contain a single *ShouldStop()* method.On the basis of the node, its training data sample and the whole tree, they should determine whether further splitting should be tried or not.

Interface 3.4 (Stop criterion interface)

Method: ShouldStop(Node, Data, Tree)
Input: *Tree node to be split (Node), training data corresponding to the node (Data), the whole tree (Tree).*
Output: *Boolean value indicating whether to stop splitting.*

An obvious stop criterion occurs when the node splitter returns no candidate splits. In fact, such situation is not regarded as a subject to analysis by stop criteria. The code of hill-climbing search (algorithm 3.2) handles this case separately from split criteria (see items 4e vs 4c).

Commonly, splitting is ceased when the node is pure enough. Its purity may be measured in many ways, but the most common (in the context of stop criteria) is node accuracy. Sometimes maximum possible accuracy is required. In other cases, some error is accepted and a threshold is defined as acceptable number of errors or error percentage.

Some authors decide not to split small nodes. Again a size threshold must be provided to precise what "too small" means.

Another possibility is to stop splitting when the tree has reached an assumed complexity (number of leaves). Such strategy may be used when one needs to save time and wants to get trees of some size (not to grow full trees). To prevent growing just a single branch, the technique should be used together with proper split prospects estimator, to guarantee that the nodes with possible large improvement are split before the less promising ones. Another way to prevent unbalanced trees is to use search methods that grow the tree in a balanced way (for example breadth-first search).

Algorithms like Cal5 or CTree join stop criteria with split feature selection, because when it is not possible to reject any hypothesis claiming target independence from the features, it can be suspected that no split can be useful and stop decision should be made. Although such functionality resembles the function of stop

criteria, this is an example of an implicit stop rule, because its result is that an empty collection of split candidates is returned by the node splitter module and a leaf is created not because of a stop criterion, but as a natural consequence of no possibility of further splits.

Stop criteria may also be combined in a manner corresponding to logical operators "and" and "or". Formally, such combinations are also split criteria (hence listed in Fig. 3.1).

3.1.4 Split Acceptors

When the best possible split may still be assessed as not satisfactory, it is rejected and the node it was to split, becomes a leaf. To determine whether to accept a split or not, the functionality of interface 3.5 must be implemented.The method *SplitAcceptable()* gets the information about the node, the tree, and the training data samples corresponding to the node being split and to its daughter nodes. It outputs the decision whether the split should be accepted or not.

Interface 3.5 (Split acceptor interface)

Method: SplitAcceptable(TreeNode, Tree, Data, SplitData)
Input: *Tree node to be split* TreeNode*, the whole tree* Tree*, training data corresponding to the node* Data*, collection of datasets after the split* SplitData*.*
Output: *Boolean value indicating split acceptance.*

Most of the algorithms unconditionally accept the splits and leave the task of split quality verification to pruning methods. Some methods, like FACT, verify whether splits improve tree classification and accept them only when this condition is satisfied.

3.1.5 Split Prospects Estimators

In typical DT induction by recursive splits, the order of splitting is not significant. When tree branches are not grown maximally, the order may become very important. For example, when the target binary tree is to contain maximally seven leaves, after splitting the root node, it would not be right to focus on one arbitrary subnode only and generate a subtree with six leaves while keeping the second subtree as a leaf. To maximize the expected value of tree accuracy, it is sensible to always split the node which seems the most promising. Naturally, there are many possible ways of defining what "promising" means. The heuristics acting as split prospects estimators implement interface 3.6. Its *SplitProspects()* method returns a real value

corresponding to the predicted gain resulting from further splits of the node given as a parameter together with the corresponding training data sample.

Interface 3.6 (Split prospects estimators interface)

Method: SplitProspects(Node, Data)
Input: *Tree node to examine (Node), training data corresponding to the node (Data).*
Output: *Real value indicating prospective gain of further splitting.*

In the search implementation presented as algorithm 3.2, the value returned by *SplitProspects()*, serves as the 3key in a key-value heap structure. In general, adding items to the heap and removing the item of the largest key are operations of complexity $O(\log n)$, where n is the number of elements in the heap. When the order of splits is irrelevant, an estimator returning the same value for all splits may be used. If all items in the heap have the same key, new elements are added in constant time ($O(1)$), because the heap property is always preserved. Also, removing the maximum key element (the root of the heap) is performed in $O(1)$, because after single move of the last element to the root, the heap is still correct.

Another natural solution is to estimate node split perspective as the maximum possible accuracy gain of further splits. Then, the nodes have no larger key values then their parents, so most often, the nodes are added to the heap in decreasing order. When an item is added with key value not greater than the minimum key in the heap, the time complexity of the operation is also $O(1)$. Removing from such heap costs $O(\log n)$ of time, but it must be kept in mind, that the heap size n is the number of nodes waiting for splits, which is close to the length of the path being analyzed, so such solution is also very efficient.

3.1.6 Decision Making Modules

The target functionality of decision tree models is to assign class labels or real values to data objects. Decision making modules have been extracted to focus on this task. Also this aspect of DTs is not unambiguous, so various methods have been proposed. An analysis of possible solutions has resulted in interface 3.7, which must be implemented by decision making modules. The main method of the interface is *Decision()*, which gets the decision tree and a dataset of objects, and returns adequate collection of decisions assigned to subsequent data items, calculated on the basis of the tree. Optionally, when classification DTs are regarded, apart from the most probable class assignment, each object can be described with a number of weights corresponding to particular classes (often interpreted as probabilities of belonging to the classes).

Interface 3.7 (Decision making interface)

***Method:* TrainingInfo(Node, Data)**
Input: *Tree node to examine (*Node*), training data corresponding to the node (*Data*).*
Output: *An object prepared for the purpose of further decisions.*

***Method:* Decision(Tree, Data)**
Input: *Decision tree (*Tree*), data to make decisions about (*Data*).*
Output: *Decisions for all data objects, optional weights for the objects.*

Aside the main functionality, additional method *TrainingInfo()* has been included in the interface. The auxiliary method is called during the tree construction process (as discussed earlier, in the context of algorithm 3.2) to extract the information required for further decision making. It helps reduce memory consumption, because after the necessary information is extracted, the training data sample is no longer used and can be released from memory. So, in fact, the method is introduced for technical purposes, not because of the algorithmic substance. It could also be moved to another interface, specializing in describing DT nodes, but since the main goal of the description is to make decisions based on the trees, it has extended the decision making modules functionality.

In DT classifiers, the decision making module usually determines target values according to the class distribution in appropriate tree leaf, however other solutions have also been proposed. Several selected methods can be found in Sect. 2.6.

3.1.7 Data Transformations

As mentioned in the comments on hill-climbing implementation, the data sample at each node may be transformed, before a node splitter is applied. Data transformations are optional. When supplied, they should just transform one data sample into another. Formally, interface 3.8 must be implemented.

Interface 3.8 (Data transformation interface)

***Method:* Transform(Data)**
Input: *Data to be transformed (*Data*).*
Output: *Data after transformation.*

Data transformation objects are a perfect mean to realize the algorithms of the LTree family (see Sect. 2.3.3), as they add new features at each tree node before the analysis. They are also helpful, when split methods require data in some specific form, for example, discretized or described in the space of continuous features only.

3.1.8 Some Implicit Details

Not all techniques used in DT algorithms have been visualized in Fig. 3.1. For example, there are many different ways of dealing with missing data or reflecting classification error costs in the tree building processes and in final decision making. Such techniques are easily incorporated into data transformations (missing value imputations), split quality measures (error costs), tree decision making (surrogate splits of CART), and so on. Offering some of these solutions, requires special data representation (for example, including misclassification costs or using importance weights assigned to objects). Therefore, compatibility issues must be kept in mind when designing such components, so that they can successfully cooperate with all other modules composing the induction method.

3.2 Decision Tree Refinement

Miscellaneous methods aimed at as good generalization of DT models as possible are described in Sect. 2.4. According to the taxonomy, presented there, three groups of such methods can be distinguished for the purpose of the unified model of DT induction algorithm:

- stop criteria (pre-pruning methods),
- direct pruning methods (post-pruning with no separate validation dataset),
- validation methods (post-pruning with single or many validation passes).

Each of the three approaches must be implemented in different way, in a practical data-mining system. The unified framework, described here, offers adequate solutions for each of them. Specific aspects of each implementation are discussed in subsequent subsections.

3.2.1 Stop Criteria

The concept of stop criteria is intrinsic to the tree induction process, although its aim is very close to tree pruning methods. The stop criterion interface and its most natural implementations have been presented in Sect. 3.1.3.

Simple stop criteria have proven to be far from satisfactory. In many publications they are referred to as significantly worse than numerous post-pruning methods. Therefore, most approaches use one of the two other strategies, that is, build complete trees, possibly overfitting the training data and then try to prune them appropriately.

3.2.2 Direct Pruning

According to the definition, direct DT pruning methods perform their task directly on the basis of the full DT and the information about training data distribution among tree nodes. Most often, the techniques belonging to this group analyze the training process with statistical tests and, on the basis of their results, prune nodes or keep tree splits. Therefore, they can be designed as simple transformations of DTs, which receive a DT and return another DT. No more inputs or outputs are necessary to perform the tasks, just the tree structure and information extracted by decision making module's *TrainingInfo()* method.

3.2.3 DT Validation

Validation methods are more complicated than direct pruning methods. As described in Sect. 2.4.3, this group of algorithms may be further split into two subgroups: performing single-pass validation and based on multi-pass validation.

Single pass-validation is very simple. It is presented as algorithm 3.9. Its input parameters are separate training and validation data samples (D_T and D_V), a DT induction algorithm (*Learner*) and a validation procedure (*Validator*). Given the four parameters, the algorithm learns a tree (with *Learner*) for D_T and validates the resulting tree on D_V with the *Validator*. The validation process is split into two parts: method *Validate()*, which extracts the information about the validation and *PruneOptimally()*, capable of pruning given tree according to the validation results extracted with *Validate()*. The split may seem unnecessary here, but it allows the multi-pass validation methods to act as single-pass validators.

Algorithm 3.9 (DT learning with single-pass validation)

Prototype: ***LearnDTSingle***$(D_T, D_V,$ ***Learner, Validator)***
Input: *Training data (D_T), validation data (D_V), DT learner (*Learner*), DT validator (*Validator*).*
Output: *Decision tree.*
The algorithm:

1. *tree* ← *Learner.LearnFrom(D_T)*
2. *validationInfo* ← *Validator.Validate(tree, D_V)*
3. *tree* ← *Validator.PruneOptimally(tree, validationInfo)*
4. ***return*** *tree*

The general idea of multi-pass validation is presented as algorithm 3.10.Its parameter list is slightly different than the one of single-pass validation. Here, only one data sample *D* is given, as multiple training and validation samples are generated inside the algorithm by means of the parameter named *Distributor*. At the beginning

of the algorithm, the *Distributor* generates a collection of pairs of data samples for n passes of training and validation. Then, for each pair of data samples (D_i^t, D_i^v), the *Learner* is run to induce a DT from D_i^t, which is validated on D_i^v by the *Validator*. The validation results are collected into $\mathbf{V}$, which then serves as the source of information for determining what optimal pruning means in this particular situation. Finally, in item 3, the final tree is induced and, in item 4, pruned optimally by the *Validator* on the basis of validation information $\mathbf{V}$.

Algorithm 3.10 (DT learning with multi-pass validation)

Prototype: ***LearnDTMulti(D, Distributor, Learner, Validator)***
Input: *Training data (D), training and validation datasets distributor (Distributor), DT learner (Learner), DT validator (Validator).*
Output: *Decision tree.*
The algorithm:

1. $(D_1^t, D_1^v), \ldots, (D_n^t, D_n^v) \leftarrow$ *Distributor.PrapareTrnValData(D)*
2. ***for*** $i = 1, \ldots, n$ ***do***
   a. $T_i \leftarrow$ *Learner.LearnFrom*(D_i^t)
   b. $V_i \leftarrow$ *Validator.Validate*(T_i, D_i^v)
3. *tree* $\leftarrow$ *Learner.LearnFrom(D)*
4. *tree* $\leftarrow$ *Validator.PruneOptimally(tree,* $\mathbf{V}$*)*
5. ***return*** *tree*

Thanks to such design of algorithms 3.9 and 3.10, they can be given any compatible validation object, implementing validation interface 3.11,

---

Interface 3.11 (DT validator interface)

Method: ***Validate(Tree, Data)***
Input: *Tree to validate (Tree), validation data sample (Data).*
Output: *An object describing results of validation.*

Method: ***PruneOptimally(Tree, ValidationInfo)***
Input: *Decision tree (Tree) to be pruned, collection of validation results (ValidationInfo).*
Output: *Decisions tree (pruned Tree).*

---

that is, capable of gathering validation information (method *Validate()*) and pruning DTs according to the information gathered in one or more validation passes (method *PruneOptimally()*).

3.2.4 Pruning Methods Parameters

One of the aims of learning-machines unification is to facilitate equitable comparisons of different implementations of the same interfaces by putting them in the same environment, so that the differences in results can unambiguously point the real causes of results improvements and declines.

Each DT pruning or validation method has been published with its own set of parameters and often with some auxiliary solutions, which in a unified framework must be pulled out and separated from the main algorithm. Apart from the parameters closely related to specific functions of the methods, there are some parameters, applicable to most (or even all) of the methods performing similar tasks. Fair comparison should not give advantages of using additional parameters (or techniques) to one method and not to the others. Therefore, all pruning algorithms must be implemented in such a way, that the common parameters may be used, wherever possible. Two common parameters have been extracted in this way and can be used with all adequate algorithms in the same way. These are two factors modifying the way validation error is calculated and applied to model selection: one concerning standard error and the other including training error in the validation process. They are discussed in the succeeding subsections.

3.2.4.1 Standard Error Factor

In most pruning techniques, the main objective is classification error. In the set of algorithms described in Sect. 2.4, there are two exceptions of direct pruning methods based on MDL and DI—they do not directly use misclassification as the main criterion. All the others are eligible for extensions reflecting the idea of standard error, introduced by Breiman et al (1984). According to their suggestions, it is often worth to accept less accurate (as estimated in the validation process) model if only:

- its result is not much worse that the best one, and
- it is structurally simpler than the one with the best validation score.

The latter condition is quite precise, in the case of DTs, where "simpler" may mean "with smaller number of leaves". The former premise is not as precise, and requires more formal definition. Breiman et al (1984) have defined this as the 1SE rule, according to which, they selected smaller tree if its misclassification risk was not larger than the best one observed, by more than the value of estimated standard deviation of the error. Given the estimated error value $e \in [0, 1]$, they estimated its standard deviation by treating the classification results as a series of n independent random variables of the same binomial distribution with probability of success equal to e (or $1 - e$ for more natural interpretation of the term "success"). On such assumption, the standard deviation of the mean of the series of binomial variables is given by the formula:

$$SE = \sqrt{\frac{e(1-e)}{n}}. \tag{3.1}$$

There is no reason, for which the distance of one standard error, should be the only way to define the threshold. A generalization of 0SE (selection of the model with the smallest validation error) and 1SE methods is to use a factor for SE to define the maximum acceptable distance from the minimum error. The tests presented in Chap. 5 deal with the factors of 0, 0.5 and 1 to define the acceptable error increase in simpler models. As a result, 0SE means selection of the parameter which recorded the best result, while 1SE and 0.5SE denote selection of the simplest models with errors not larger than the best one by more than the SE from the formula (3.1), and half of the value of SE respectively.

There is also another possible way of standard error estimation. In the case of multiple validation (based on CV), the standard deviation of the error may be estimated from the sample of CV results. Such algorithms are denoted with additional 's' (for 'sample') in the name: 0.5SEs and 1SEs.

Direct pruning methods can also respect three values of the parameter (0SE, 0.5SE and 1SE), though in slightly different context: here subtree is pruned, if the estimated error of the leaf replacing the subtree is not worse than the result of the whole subtree by more than appropriate part of the estimated standard error.

Although the SE parameters are used in slightly different meaning in different types of pruning methods, the major task of the options is the same—to accept smaller trees, if their results are not significantly worse—and because of that, it makes sense to consider the parameters correspondingly in results comparisons.

3.2.4.2 Training Error Factor

When validating a number of models, in order to select the one of the highest expected accuracy, it is very important to get as good estimates of the accuracy/error as possible. The same concern inspired the work of Efron (1983); Efron and Tibshirani (1997) on 0.632 bootstrap estimator (see Sect. 2.8.2). They noticed that when training on bootstrap samples and testing on the difference between the whole data and its part engaged in the bootstrap sample, the test error is biased upwards, while the training error is biased downwards (which is quite understandable). Because bootstrap samples are expected to contain 63.2 % of the training data objects, Efron (1983) proposed to estimate the error by combining training error and test error linearly with coefficients of 0.368 and 0.632 respectively (see Eq. (2.92) page (91)) .

The phenomenon of biased error estimates is very important also from the point of view of DT validation. For example, in tenfold cross-validation, the training sample contains 90 % of the whole input dataset and the validation data sample is much smaller (just 10 %). When validation maximizes the model performance on the validation data, it may spoil some parts of the model, when the validation sample does not contain objects from some parts of the object space. Since the validation sample is so small, it is very likely to not represent all important areas of the feature space. As a result, the model providing best validation results, may overfit the validation sample.

The idea of *training error factor* is the same as that of the 0.632 estimator: to combine both training sample and validation sample misclassification in final model selection. If model selection is based on the misclassification rate, one can use

$$E = \frac{E_{val} + TEF \times E_{trn}}{1 + TEF}, \tag{3.2}$$

where E_{trn} is the training sample error, E_{val} is the validation sample misclassification rate and TEF is the parameter factor—with $TEF = 0$ only the validation error is in function, and increasing the factor makes pruning weaker and weaker up to some point, above which no pruning is done. The value of $TEF = 0.5$ combines the errors with linear coefficients $\frac{2}{3}$ and $\frac{1}{3}$ respectively (quite close to the .632 estimator) and the value of $TEF = 1$ results in fifty-fifty weights.

In practical implementations, the denominator of (3.2) may be ignored, because it is the same for all candidate trees and the goal is to find the one minimizing the error, not in exact prediction of the error value.

The factor can be successfully used both in model ranking for building cross-validation based committees of decision trees and in single DT induction (see Chap. 5 for more details).

In some sense, the function of *TEF* is opposite to the *SE* factor described above, because increasing *SE* factor forces more pruning, while increasing *TEF* strives in smaller error rate on the training data, which means larger trees. The intuition behind using the two parameters together is that the *SE* factor can offer small trees, while *TEF* can keep the nodes most adequate for the whole data available (training + validation) not just for the validation sample. Therefore, proper balance between the two, may result in small trees, containing the most important nodes.

3.2.4.3 Parameter Search for MEP2 and DI

The algorithms MEP2 and DI, described in Sects. 2.4.2.3 and 2.4.2.5 in the context of direct pruning, use parameters (m and β respectively) to control the strength of pruning. Although the authors of the algorithms mentioned possibilities of CV-based validation of the parameters in Breiman style, they did not propose any particular solutions.

Unfortunately, finding the most appropriate m in MEP or β in DI, is not as easy as determination of the α parameter of cost-complexity pruning or the degree of pruning or the size used by the OPT algorithm. The problem lies in finding the thresholds for the parameters, for which the error of a subtree and a leaf replacing it are the same. In cost-complexity pruning the threshold can be easily calculated as a solution of a linear equation with one parameter (α) regardless of subtree sizes. For finding the threshold m in MEP or β in DI, an equation to solve can also be determined, but its degree is proportional to subtree depth, as the variables modify the objective functions at each node of the path in a cascading way. Since the thresholds are zeros of polynomials of

degrees proportional to proper subtree depth, it gets computationally unattractive for trees of larger sizes, because the problem of finding zeros of polynomials is ill-posed.

Therefore, to create algorithms applicable to all domains, other solutions need to be chosen. A suboptimal, but feasible solution is searching for the best values of the parameters within a predefined set. Because the nature of the parameters is exponential, in the experiments of Chap. 5, the MEP parameter validation has been configured to estimate the values of $m = 2^k$ for $k = -12, \ldots, 12$ and select the best of them for the final model. In the case of the DI algorithm, the validation process has been defined to select the most attractive of the values of $\beta = \sqrt{2}^{-k}$ for $k = 0, 1, 2, \ldots$. In each node, the end of the scope may be determined separately—the calculations may end when the node becomes a leaf.

3.3 Well Known Algorithms as Instances of the Uniform Approach

All the algorithms described in Chap. 2 perfectly fit the unified view presented above.They have been decomposed into suitable modules. Table 3.1 presents general information about the types of modules required to realize the goals of particular algorithms.

It is not possible to show all the details in such compact visualization. Some table cells contain more than one numbered bullet indicating that the methods have several variants. Some options are deliberately omitted to keep the table readable, for example, all algorithms but one are assigned ① in the row of tree decision, while in fact, many algorithms have been tried also with Laplace correction or m-estimates of probabilities. The table just illustrates that algorithm analysis and extraction of separate functional units gave possibility to easy construct many DT induction algorithms, including those described in Chap. 2.

3.4 Framework Facilities

Supplied with many implementations of the interfaces, described above, one can construct thousands of different algorithms by just combining proper components.

It is especially important in meta-learning that algorithms can be easily modified and validated to adjust them to the needs of particular data. The framework has been created because of such needs. It seems inevitable that in future, only such versatile environments, supported by large databases of meta-knowledge extracted from vast amount of experiments, will be used and will provide many interesting models for various purposes. Such architecture can be very beneficial in different kids of analysis on the meta-level.

Table 3.1 Most popular DT algorithms in terms of the unified model

	ID3 (Sect. 2.2.1)	CART (Sect. 2.2.2)	C4.5 (Sect. 2.2.3)	Cal5 (Sect. 2.2.4)	FACT (Sect. 2.2.5.1)	QUEST (Sect. 2.2.5.2)	CRUISE (Sect. 2.2.5.3)	CTree (Sect. 2.2.6)	SSV (Sect. 2.2.7)	LMDT (Sect. 2.3.1)	OC1 (Sect. 2.3.2)	LTree family (Sect. 2.3.3)	DT-SE family (Sect. 2.3.4)	LDT (Sect. 2.3.5)	Dipolar criterion (Sect. 2.3.6
Initial data transf.	①				②							③	③		
Search method	①	①	①	①	①	①	①	①	① ② ③	①	①	①	①	①	①
Split prospects est.	①	①	①	①	①	①	①	①	① ③	①	①	①	①	①	①
Stop criterion	①	① ②	①	①	①	①	①	①	①	①	①	①	①	①	①
Node data transf.					②	⑥						⑤			
Node splitter	①	①	①	② ③	③	③	③	③	①	⑤	④	①	④	④	④
Split quality m.	②	④	③	②				⑥	① ⑤			③			
Whole feature splitter				③	①	②	① ②								
Split feature selector				②	①	①	①	③							
Split acceptor					①										
Tree decision	①	①	①	①	①	①	①	①	①	①	①	②	①	①	①

The framework is being successfully used in research activities concerning miscellaneous aspects of DT induction. It facilitates as fair comparisons of different components in action as possible. For example, to perform a reliable comparative test of a number of split criteria, one can embed each competing component into the same surroundings consisting of a repeated cross-validation of a DT induction process specified by the search method, validation method, decision module and so on. Providing the same environment to all the competing modules guarantees the same training and test data in corresponding passes, even in the case of inner validation, if required by the test scenario. After collecting the results from such test procedures, full information about corresponding scores is available, so statistical tests like paired t test, Wilcoxon test or even McNemar test (which requires the information about correspondence between single classification decisions, not only between the mean accuracies for the whole test datasets) can be applied. In the same way, we may compare other types of components like data transformations, stop criteria, validation methods and others.

Conducting the test is quite easy with such framework at hand, implemented within as flexible machine learning environment like Intemi, presented in Chap. 4.

The information from such tests provides very precious meta-knowledge to be used in further meta-learning approaches and eventually to compose more accurate DT induction methods. The meta-knowledge can be gathered in special repositories, facilitating robust algorithm selection by meta-learners.

References

Breiman L, Friedman JH, Olshen A, Stone CJ (1984) Classification and regression trees. Wadsworth, Belmont

Efron B (1983) Estimating the error rate of a prediction rule: improvement on cross-validation. J Am Stat Assoc 78(382):316–331

Efron B, Tibshirani R (1997) Improvements on cross-validation: the .632+ bootstrap method. J Am Stat Assoc 92(438):548–560

Chapter 4
Intemi: Advanced Meta-Learning Framework

Serious meta-learning applications may require running huge counts of learning processes. The results obtained from the calculations must be reliably analyzed by many comparisons and statistical tests.To gain valuable knowledge about data at hand, it does not suffice to run a couple of methods and show the results. Also in scientific experiments, it should no longer be accepted, that several algorithms are tried and new approaches are claimed to be advantageous, on the basis of several simple tests and comparisons to several other methods. The era of so undemanding data analysis has passed, since available computational power facilitates significantly more thorough experiments. More and more sophisticated, automated meta-learning tools are being developed and will certainly, more and more often, outperform human experts in building accurate models for various kinds of data.

Robust meta-learning analyses, reliable comparisons of numerous methods in complex test scenarios are possible only with a universal and versatile, but also efficient and easy to use framework.It should support learning numerous machines and avoiding multiple calculations of the same tests. It should facilitate robust comparisons between old and new calculations (performed in different runs of the system), due to testing machines on exactly the same data samples.A robust meta-learning framework should provide the following features:

- A unified encapsulation of most aspects of handling CI models like learning-machines creation, running and removal, defining inputs and outputs of adaptive methods and their connections, adaptive processes execution, and so on.
- The same way of handling and operation of simple learning machines and complex, heterogeneous structures. Easy definition, configuration and running of machine hierarchies (submachines creation and management).
- Easy and uniform access to learners' parameters.
- Easy and uniform mechanisms for representation of machine inputs and outputs, and for universal information interchange between machines.
- Efficient and transparent multitasking environment for processes queuing/spooling and running on local and remote CPUs.

K. Grąbczewski, *Meta-Learning in Decision Tree Induction*,
Studies in Computational Intelligence 498, DOI: 10.1007/978-3-319-00960-5_4,

- Versatile time and memory management for optimal usage of the computational resources.
- Easy and uniform access to exhaustive browsing and analysis of the machine learning results.
- Simple and efficient methods of validation of learning processes, conducive to fair validation, that is, not prone to testing embezzlements.
- Tools for estimation of model *relevance*, analysis of reliability, complexity and statistical significance of differences [a useful set has been collected by Lowry (1998–2013)].
- Mechanisms of machine unification and a machine cache system preventing repeated calculations (significant in so large scale calculations like meta-learning).
- Templates for complex method structures with exchangeable parts, instantiated during meta-learning.
- Rich library of fundamental methods providing high functionality, versatility and diversity.
- Simple and highly versatile Software Development Kit (SDK) for programming system extensions.

It is not easy to design an architecture fulfilling so many expectations, especially, when one of the strong requirements for the system is simplicity of use. Fortunately, proper system kernel foundations reconciled all the requirements, despite they seem contradictory.

The key idea of *Intemi* system is to create projects, where:

- *configurable machines* interact by means of their *outputs* and *inputs*,
- each machine can create *submachines* as modules performing appropriate parts of more complex tasks,
- *machine unification* mechanisms prevent multiple runs of the same models,
- *requests* for machine runs are ordered within a task spooler and performed by *multithreaded task running managers*,
- general *results repository* provides uniform mechanisms of access to other machines results,
- universal *query system* lets the user easily collect the results of interest from the repository,
- *results series* can be easily transformed to extract required information.

The following sections describe the crucial solutions of the modules responsible for all these aspects.

4.1 Machines and Models

The discussion of Sect. 1.1 has pointed out that the terms *machine* and *model* are often used without proper definition, which sometimes introduces a confusion. It has proposed a distinction between the two terms.Intemi follows these guidelines, so

that its *machine* is any process that can be configured and run to bring some results, and a *model* is the result of such a process (of a machine run). For example, an MLP network algorithm (the MLP machine) can be configured to use appropriate network structure, initial weights and learning process parameters. It can be run on some training data, and as a result we get a trained network—the MLP model created by the learning process of the MLP machine.

The unified view of machines deliberately avoids the term "learning machine", since a machine can perform any process which, not always, would naturally be called a learning process, such as loading data from a disk file, data standardization or testing a classifier on external data.

A general view of machine is presented in Fig. 4.1. Before a machine may be created, its configuration must be specified and *inputs* (*input ports*) bound to some *outputs* (*output ports*) of other machines. After a machine is created and its process finished, the machine *outputs* and *results* deposited in the repository are available to other machines and to the user. The outputs and repository results compose the model of the machine.

Machine configuration is defined as

$$C = \langle i, o, p, (C_i)_{i=1,\ldots,n} \rangle, \tag{4.1}$$

where

- i and o are specifications of inputs and outputs respectively, in the form of mappings from port names to types of acceptable objects,
- p represents machine process parameters,
- $(C_i)_{i=1,\ldots,n}$ is a sequence (possibly empty) of submachine configurations.

The possibility of defining machine subconfigurations and creating submachines within each machine run is crucial for construction of complex machines, inevitable in nontrivial projects. In fact, machine subconfigurations could be included in machine parameters p, but because of their special function, common usage and a need for tools handling them in a uniform way, they have been made explicit in the definition. The abstract view of machines presented in Fig. 4.1 does not include machine subconfigurations. In further figures, starting with Fig. 4.4, the submachines are visualized as boxes placed within the boxes corresponding to their parent machines and, in the same way, machine subconfigurations are shown inside their parent-machine-

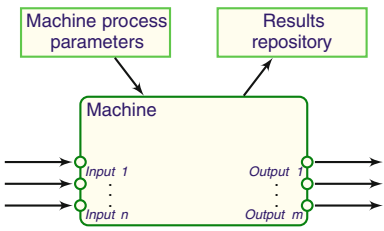


Fig. 4.1 The abstract view of a machine

configuration boxes (with different border color used for machines and machine configurations).

It is important to realize that a single machine configuration may be used to create more that one submachine. For example, a bagging machine must be given a configuration of a classification machine to be repeatedly applied to different bootstrap samples generated from the training data.Implementations created in compliance with the art of object oriented design and programming would also define a separate machine for bootstrap sample generation and generate each subsequent data sample with machines of the same configuration (the outputs can still be different for each machine, because of their randomized behavior—see Sect. 4.2.2.3 for more information on handling random processes in Intemi).

Because of the relations between machines, projects may be seen as directed graphs of machines with edges defined in two different ways:

- by parent–child relation,
- by input–output interconnections.

The former graph is a tree with the role of root played by the project (which can be treated as the main machine), while the latter is a directed acyclic graph of machines (more precisely of their *contexts*, because machine unification mechanisms make the relation between machines more complicated, but not much interesting from the point of view of the project user—see Sect. 4.2).

Machine ports (inputs and outputs) are the main means of information exchange between machines. The difference between machine inputs and parameters is that inputs come from other machines while the parameters are specific to the process and are provided directly by the user or result from parent machine configuration. Similarly, outputs exhibit parts of the model to other machines (to be passed as their inputs, when proper input–output connection is established). Results repository serves as kind of report from machine run and contains an excerpt from the model. It is up to the machine author whether the machine receives any inputs, whether it has some adjustable parameters and whether it has outputs and/or puts results in the repository.

Two examples of simple machines are presented in Fig. 4.2. The Data loader machine receives no inputs and declares a single parameter which is a string containing the file name path from which the data is to be loaded. So Data loader machine configuration has a form of $C = (\perp, o, p, \perp)$, where $o = \{\text{“}Data\text{”} \rightarrow \textit{type of data}\}$ and p is just a string representing a file path. The machine process exposes the data series as the output and deposits no entries into the results repository.

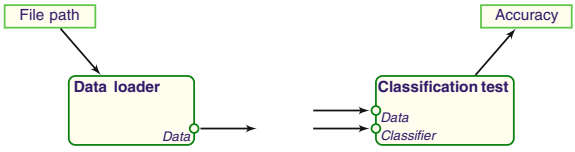


Fig. 4.2 Machine examples: a data loader and a classification test

The machine of Classification test , presented in Fig. 4.2 on the right, takes inputs but not parameters. The inputs introduce the classifier (more exactly an interface with the classification routine) to be tested and the test data series. Formally, its configuration can be written as $C = (i, \perp, \perp, \perp)$, where $i = \{$*"Data"* $\rightarrow$ *type of data*, *"Classifier"* $\rightarrow$ *type of classifier*$\}$. A result of the machine process is the information about accuracy of the classifier (in more complex implementations it could also present such results as confusion matrix). The machine provides no outputs, because the information it gains is not expected as input of any other machine (other points of view could result in outputs representing class labels given by the classifier to each data object, a vector of misclassification indicators or other potentially useful information).

In the following examples, machine parameters and results deposited in the repository are not presented explicitly, as they are not so important from the point of view of presenting machine relations. The information flow between outputs and inputs is usually very important, so the interconnections are thoroughly presented.

The unified concept of machine does not introduce any kind of machine type, so we do not split machines to classifiers, data loaders, data transformers and so on.Instead, the inputs and outputs of a machine define possible contexts of its application, that is, any machine providing an output of type *Classifier* may be called a classifier and used in the context of a classifier, for example, may be tested by the Classification test machine when the latter's *Classifier* input is bound to the classifier output of the former machine. Thus, a single machine may be useful in a number of ways. For example, a decision tree may expose a *Classifier* output and also a *Feature ranking* output (generated on the basis of features occurring in the conditions defining the tree nodes). It lets the decision tree machine occur both in the context of a classifier and of a feature ranking.

4.1.1 Feature Selection and Ranking

The family of feature selection methods based on ranking measures is an interesting illustration of the information flow between machines.

The object oriented encapsulation idea leads to a split of the feature selection process into two stages: generation of feature ranking and selection of the adequate number of the top-ranked features. It results in just one feature selection machine of this kind, and a number of feature ranking machines implementing particular algorithms.

As shown in Fig. 4.3, the feature selection machine declares two inputs (*Data* and *Feature ranking*) and outputs two interfaces (*Data* providing data series obtained by filtering the input data to keep just the selected features and the feature selection *Transformation* in a form applicable to external data). How many features are to be selected is defined within the feature selection machine parameters: arbitrary number of top-level features or the features with the ranking value exceeding given threshold. Naturally, there is no single rule for optimal setting of these parameters for

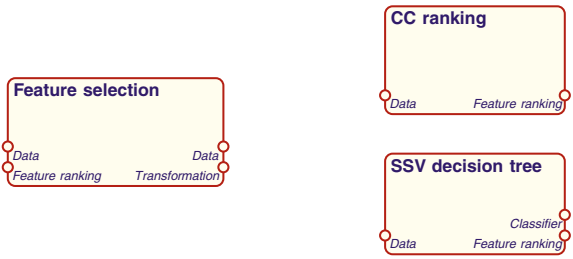


Fig. 4.3 Configurations of *feature selection* and feature ranking machines

all data (all problems), so they must be determined separately for each data.It can be obtained with meta-learning: sometimes so simple methods as meta parameter search presented in Sect. 4.4.1 are satisfactory and sometimes more advanced approaches combining feature selection with other data transformation methods are inevitable (Jankowski and Grąbczewski 2008; Grąbczewski and Jankowski 2008).

The right side of Fig. 4.3 shows two configurations of machines generating feature rankings, eligible for cooperation with the Feature selection machine. The top one (CC ranking) is dedicated to feature ranking, but the bottom one (SSV decision tree) is a DT induction machine. Its main output is Classifier, but because it also builds a ranking on the basis of the features used by the tree, it can also play a role of feature ranking.

4.1.2 Schemes and Configuration Templates

Configuration of complex learning machines often requires providing not only single subconfigurations, but whole scenarios of many machines. To make definitions of such scenarios easy, a machine named Scheme has been created to function as a machine container or machine group.

An example of Scheme machine is presented in Fig. 4.4). Its scenario assumes two data collections coming through the inputs (one for training and one for test). Two learning machines: k Nearest Neighbors (kNN) and Support Vector Machine (SVM) are to be trained on the training data and their classifier outputs tested by separate test machines on the test data.Such scenario is very useful for detailed comparison of the results of the two learning algorithms, because they are trained on exactly the same data series and tested on the same data, so each particular decision can be reliably compared (see Sect. 4.3).

Formally, configuration of the scheme machine presented in Fig. 4.4 is $C = (i, \perp, p, (C_{knn}, C_{svm}, C_{tk}, C_{ts}))$, where i specifies types of the two inputs ("Training data" and "Test data"), the four subconfigurations correspond to kNN, SVM and two classification test machines and p defines all the bindings of the four submachine inputs (bindings are instructions to the scheme, where the submachines will get their inputs from).

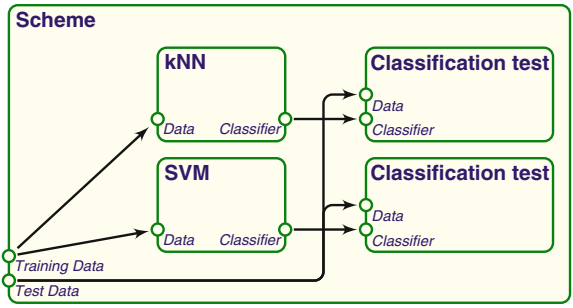


Fig. 4.4 A scheme machine example

The process of a scheme machine simply runs all submachine processes in one-to-one correspondence to the configuration. It is formalized as algorithm 4.1.

Algorithm 4.1 (Scheme machine process)

Prototype: Scheme.Run(C)
Input: *Scheme machine configuration* $C = \left(i, o, p, (C_i)_{i=1,\ldots,n}\right)$.
Output: *Scheme model.*
The algorithm:

1. $(C'_i)_{i=1,\ldots,n} \leftarrow$ *topologically sorted* $(C_i)_{i=1,\ldots,n}$
2. ***for*** $i = 1, \ldots, n$ ***do***
   request submachine configured by C'_i *with its input bindings defined in* p
3. *Wait until all submachines are ready*
4. *Define scheme machine outputs according to the bindings in* p
5. ***return*** *the scheme model (the outputs and submachines)*

First, the subconfigurations must be sorted topologically with respect to the input–output relation, to guarantee that each machine request can provide consistent information about input bindings.Then, all machines are requested in the order just determined. The algorithm reflects multithreaded architecture of the task running module, so the machines are requested asynchronously and after that, the scheme machine process waits until all submachines are ready.Finally, the outputs of the scheme machine (if defined) must be exhibited to the system by informing which submachines outputs are the outputs of the scheme.

Schemes have been created to provide an easy-to-use tool for defining complex machine hierarchies. Each machine can just define a scheme configuration and request a submachine corresponding to it. The scheme configuration can be adjusted to particular needs by adequate definition of the contents, that is: inputs, outputs, submachines and their IO bindings.

Configuration templates

Schemes are also very useful from the point of view of meta-learning. They may play the role of placeholders for machine configurations (of single machines or complex machine scenarios).Such incomplete configurations are called *machine configuration templates*. A template can be easily converted into *fully specified* (*correct*, *feasible*) machine configuration, by inserting machine configurations inside it and binding proper ports. A template scheme configuration becomes feasible when:

- all its outputs are bound to a compatible output of any machine placed inside the scheme,
- all inputs of internal machines are bound to some ports (either outputs of other machines or the scheme inputs).

A valuable template for performing feature selection is presented in Fig. 4.5. The dashed box represents a placeholder for a ranking. In practice, it is represented by a scheme with adequate definitions of its inputs and outputs, so that filling the template can be done by adding subconfigurations to the scheme and specification of ports interconnections.The possibility of fixing scheme outputs facilitates defining machines with universal parts, responsible for precisely defined IO transformations that can be defined by the user, without a danger of changing the ports and loosing compatibility with the parent-machine goals.

After filling the Ranking scheme with a feasible scenario, the whole construct can be created and run or put inside another, more complex configuration.In this way, any complex scenario can play the role of feature ranking machine in Fig. 4.5.

Another substantial configuration template is the raw configuration of Transform and classify (T&C) machine, presented in Sect. 4.1.3 and depicted in Fig. 4.6. T&C is a general machine for combining transformations and classifiers, so its raw configuration does not specify either the transformation part or the classifier to be used. Both combined machines may be defined arbitrarily at configuration time, so raw T&C configuration contains just two empty schemes which can be seen as "placeholders" for subconfigurations performing functions defined by the inputs and outputs of the schemes. As a consequence, the raw T&C configuration is a template for miscellaneous configurations of classification machines requiring some data transformation before the proper learning process can be run.

Similar empty scheme may be used to parameterize configuration of boosting algorithm. The placeholder should be filled, at configuration time, with a classifier to be boosted.It means that the raw configuration of boosting is also a template.

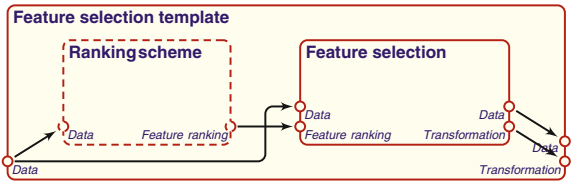


Fig. 4.5 Feature selection template

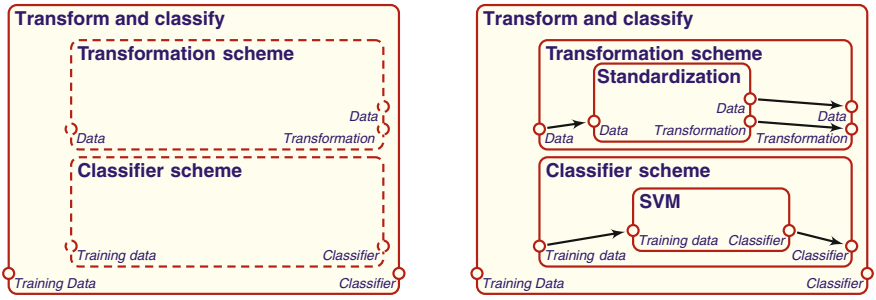


Fig. 4.6 A configuration of the *transform and classify* machine (raw and filled)

Yet another example of a template configuration is the one of the Repeater machine, discussed in Sect. 4.1.4.

Just two of the templates described above (for feature selection and combining data transformations with classifiers), augmented with information about available simple machines for feature ranking, some other data transformations (like standardization) and classification learning machines, comprise quite robust tool for generating machine configurations and, in consequence, for robust meta-learning.

4.1.3 Transform and Classify Machine

Combining data transformations and classification learners is the main goal of the Transform and classify (T&C) machine. It has been created as a general tool eligible for any particular transformations and classifiers. Figure 4.6 illustrates two configurations of the machine: the raw version and a feasible instance. The two templates inside the raw configuration make the T&C a general machine capable of performing any (arbitrarily complex) scenarios of data transformation and classification. It creates unlimited possibilities of applications of the machine. Specific contents of the two schemes may be given by the user at configuration time. For example, one can put a standardization machine inside the Transformation scheme and an SVM machine into the Classifier scheme to perform SVM classification after data standardization (as in the right part of Fig. 4.6). The interconnections within the two schemes (submachines of T&C) define the behavior within the schemes. There are no interconnections between the inputs of the parent machine and submachines or between the outputs of submachines and the outputs of the parent machine, because they are not configurable—the T&C machine will take care of appropriate connections at run time.

After the machine is run, it gets a form depicted in Fig. 4.7. Beside the connections defined at configuration time, the *input bindings* of internal schemes can be seen—they were decided by the parent machine (T&C) according to the aim of the submachines: the transformation is run on the incoming training data and the

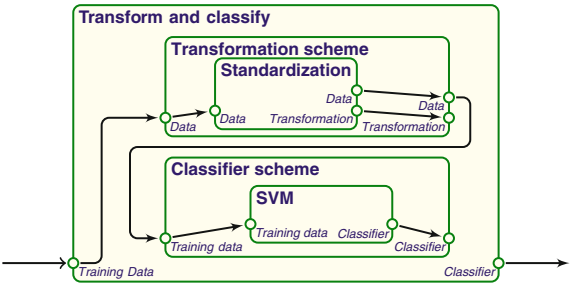


Fig. 4.7 T&C machine at run time

classifier is trained on the transformed training data.When the classifier output of the main machine is questioned to classify a series of data objects, it first transforms the data using the *Transformation* output of the transformation scheme (in fact of the standardization machine) and then classifies the transformed data with the *Classifier* output of the classifier scheme (in fact SVM output). This procedure guarantees that the test data series is standardized (with respect to the statistics of the training data) and then classified. The T&C machine can be used wherever a classifier is expected (in the same contexts as other classifiers, for example SVM), because of its *Classifier* output.

Additional advantage of the Transform and classify machine is that such form of cooperation between machines is also suitable for further optimization and test procedures. For example, classifiers may be easily combined with feature selection methods and the parameters like the number of features can be optimized in a simple and universal way, using machines like meta parameter search (see Sect. 4.4.1).

4.1.4 Repeater Machine

Performing tests like cross-validation, estimating miscellaneous performance indices by averaging, building ensemble models according to strategies like bagging, and many other data mining techniques require repeated calculations of similar scenarios. Therefore, one of the most important machines, constructed in Intemi as one of the first, is a general machine named Repeater. Initial view of the repeater configuration is presented in Fig. 4.8 on the left. It declares two scheme submachines: one for generating inputs for subsequent repetitions (Distributor scheme) and one for a scenario to be repeated (Test scheme). A single cycle of the repeater job is to generate the distributor according to the first subconfiguration, and pass its proper outputs to a number of subsequent instances of the test scheme. The distributor's outputs are so called multi-outputs, that is, collections of output objects. The collections size must be the same for all distributor's outputs, because the elements that occur in multi-output at the same index compose a *suit*, passed to subsequent test schemes. The number of suits defines the number of the test scheme instances that

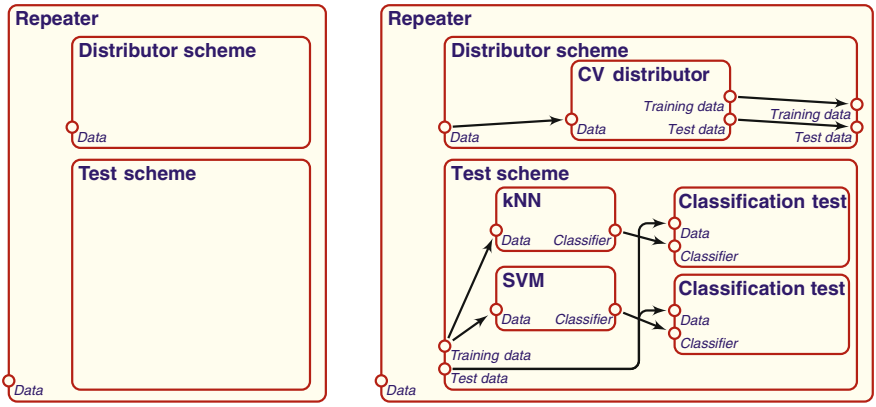


Fig. 4.8 *Repeater* machine configuration (raw and filled)

will be created by the repeater. Moreover, the repeater has a parameter defining how many times the whole scenario is to be repeated. Algorithm 4.2 presents the process of repeater machine more formally.

Algorithm 4.2 (Repeater-machine process)

***Prototype:* Repeater.Run(C)**

Input: *Repeater configuration* $C = (i, \perp, n, (C_D, C_T))$*, where* $i = \{$"Data" $\rightarrow$ *type of data*$\}$*,* n *is the number of repetitions and* C_D, C_T *are configurations of distribution scheme and test scheme respectively.*

Output: *Repeater model.*

The algorithm:

1. ***for*** $i = 1, \ldots, n$ ***do***
   a. *Request the distributor scheme* D_i *with configuration* C_D *and its input bound to the repeater input*
   b. *Wait until* D_i *is ready*
   c. $m \leftarrow$ *count of output suits of* D_i
   d. ***for*** $j = 1, \ldots, m$ ***do***
      Request the test scheme T_i *with configuration* C_T *and its inputs bound 1-1 to the j'th suit of* D_i*'s outputs*
2. *Wait until all submachines are ready*
3. ***return*** *the repeater model (the hierarchy of submachines)*

All instances of the distributor scheme and test scheme are assumed to be independent, so they all may run in parallel after being requested. Synchronization is performed at the end of the repeater process, because all submachine processes must be finished before the parent process may return.

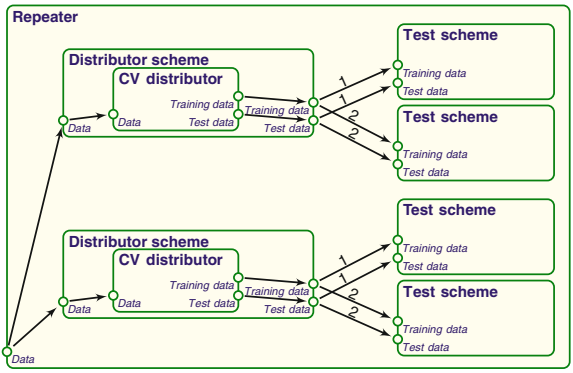


Fig. 4.9 Run time view of Repeater machine configured to perform twice 2-fold CV. Test schemes are simplified for clearer view—in fact each one contains four submachines as in Fig. 4.8

The example presented in Fig. 4.8 on the right is a repeater configured to perform a comparative CV test of kNN and SVM machines. The CV distributor receives the data as input and splits it into a number of training datasets and the same number of test datasets, according to the rule of n-fold cross-validation. During configuration of the repeater, adding the CV distributor to the distributor scheme results in propagation of the outputs from the distributor to the scheme and also to test scheme inputs, so that there is one-to-one correspondence between the distributor scheme outputs and the test scheme inputs. Only when the test scheme inputs are created, the scheme can also be properly configured.

Assuming that the number of repetitions is set to 2 and the number of CV folds in the parameters of CV distributor is also set to 2, a request for repeater machine with this configuration produces machine hierarchy presented in Fig. 4.9. Performing twice (independently) 2-fold CV requires two distributors (one for each CV cycle) and four test schemes (two per CV cycle). The outputs of each CV distributor are two training sets and two test sets—the first elements go to the inputs of the first test scheme and the second elements to the second scheme, as signaled by the numbers placed at the arrows. Although not shown in Fig. 4.9, each test scheme contains four machines as defined in Fig. 4.8, so that for each suit of training and test datasets coming out from the distributor schemes, kNN and SVM machines are trained on the training data and tested with classification test machine on the test data sample.

4.2 Machine Factory

When the system is to create a machine, it must be given more information than just machine configuration. Above, when talking about requests for new machines, usually machine configuration and input bindings were specified. It was sufficient from the point of view of understanding the processes, but formally, more detailed specification is required. Full information necessary for proper handling of machine requests is called a *machine context* and can be written formally as

$$MC = (C, P, I, B)\,, \tag{4.2}$$

where:

- C is the requested machine *configuration*,
- P is the *parent machine* requesting for child,
- I is the *child index*,
- B is the description of *input bindings*.

The information about the parent machine (handled automatically by the system, when a machine requests creation of a submachine) and the child index is necessary, mainly for the purpose of machine unification and management of random processes. Also input bindings can be defined in a more flexible way, when placed in the context together with the specification of parent machine. Input bindings are quite intuitive and naturally depicted as arrows between ports. Their formal introduction is given in Sect. 4.2.1.1.

Thanks to the unified view of machines, their configurations, contexts and related aspects, we can simply state that the fundamental task of a data mining system is to create and run machines in given contexts.

The system serves the request by assigning adequate machine to it and, if necessary, runs the machine process to make the machine ready for further analysis of its outputs and the information deposited in the results repository. Thanks to the concept of machine context, the system may assign the same machine to many contexts (unification). Then, such machine may provide its services and be analyzed in different contexts within arbitrarily complex machine structures. As shown further, sharing machines in different contexts yields significant savings in CPU time and memory.

The possible paths, machine requests go through, are presented in Fig. 4.10. The flowchart presents different states of the requests while the dashed lines encircle the areas corresponding to particular system modules.

When a machine is requested, it is necessary to *determine its inputs* (proper machines outputs, effectively bound to the inputs) first. Given input providers, the request goes to the *unification* system which determines whether the machine needs to be created or has already been run in another context and can be substituted (then, the same machine gets assigned to more than one context). If a new machine must be executed, the request goes to the *task spooler* (sort of advanced queue), from where it is taken by a *task manager thread* in order to be run.

The life of a request may be aborted at any time by the parent machine or by the system user. This may influence the flows of other requests for the same machine. Task spooler must be aware of this and react properly to such events.

The Fig. 4.10 shows a general view of the most important stages of the process and does not contain many details that have been assessed as too technical (for example, the act of registering a newly created, ready machine in the cache system for further unification purposes).

The modules, working at the assembly line of machine production, act independently in a signal-driven manner. Proper signals are sent between them when necessary, and are handled as soon as possible. Such architecture minimizes the time

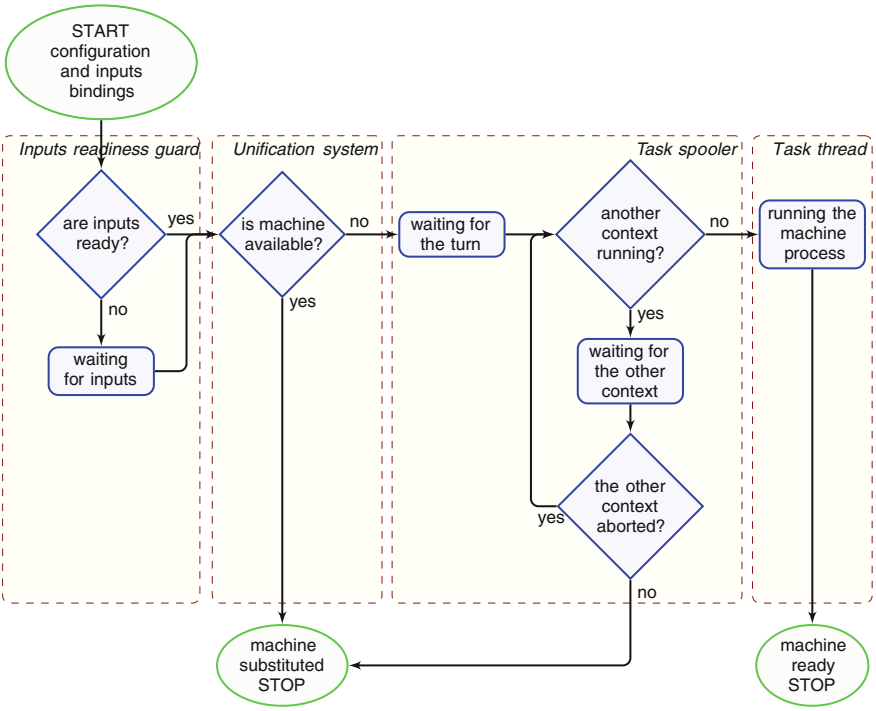


Fig. 4.10 Machine request life cycle

needed by modules to wait for each other—they serve the requests that are ready for the service, and the control of when the requests are ready for next stage takes place just in time, when appropriate events occur.

4.2.1 Inputs Resolution

Input bindings, included in machine context at request time, are high-level abstract information about the bindings, corresponding to arrows connecting ports drawn in the figures. Intemi system kernel must decode it properly to determine:

- first, machine contexts providing the inputs (the machines may not exist yet),
- then, the actual machine outputs providing information for the inputs.

These are two stages of *input resolution* performed by Intemi at appropriate times. More details are presented in the following subsections.

4.2.1.1 Input Bindings

Abstract information about *input bindings*, coming in machine requests, must describe at least one connection for each machine input. Otherwise, the request is incorrect and must be rejected at the very beginning. Therefore, *input bindings* are defined as mappings

$$B = \{input_i \rightarrow B_i | i = 1, \ldots, n\}, \tag{4.3}$$

where $input_1, \ldots, input_n$ are all inputs of the machine and $B_1, \ldots, B_n$ are collections of *single bindings*. Usually the collections contain just 1 element each, but some machines declare multi-inputs, which means that many sources can be bound to a single input. For example, machines acting as committee decision modules may combine a number of members. Then, the member collections are determined by specification of the input bindings. As a result, different instances of the same decision module type may get different counts of members.

A *single binding* can be defined in one of three ways, referring to:

- a parent's input (as it was often the case in the figures presented so far),
- a sibling's output (usually an output of a machine that does not exist at the time of configuration),
- an output of a machine determined by a capsule (which can be seen as a path in the machines subtree, from a machine called the root of the path to its ancestor following the line of direct parent-child dependencies).

In the first case, the binding is determined simply by the name of parent's input. The second type of binding is declared by an identifier of the sibling and its output name. The third option requires specification of the capsule (root machine identifier and a series of child indices of subsequent generations) and the output name.

Such information may be called abstract, because it must be decoded in particular context, to receive outputs specification containing references to proper machine contexts.

After that, a signal is sent to the *input readiness control* module responsible for appropriate notification sent when providers of all the inputs of the request are ready (their processes are finished), which means that full information, determining the requested machine, is available.

4.2.1.2 Inputs Readiness Guard

Optimal machine-running services must perform their operations immediately after adequate conditions are met. As soon as all the inputs of a requested machine are ready, the request should be served and either closed thanks to a substitution (when another identical copy has already been created) or passed to the task spooler for machine creation and run. Therefore, the inputs readiness guard controls, in real time, all the changes in machine readiness and submits runnable requests to the unification module for further services.

When a request occurs, it is handled by the method *Control()* of the inputs-readiness-guard module. Algorithm 4.3 presents the service. At the beginning, the abstract bindings B are converted to a collection of contexts providing desired outputs. All input providers are filtered to determine the set of contexts, MC needs to wait for, before it can be further serviced. If the set is empty, then finding the machines, that actually provide MC inputs, is possible, so the context is passed to the bindings resolver module for the next step. Otherwise, internal information must be updated to include the fact that MC must wait until other contexts are ready (until they are assigned with proper machines).

Algorithm 4.3 (Inputs readiness guard control method)

Prototype: IRG.Control(MC)
Input: *Machine context* $MC = (C, P, I, B)$.
The algorithm:

1. *dependenceContexts* ← *machine contexts providing the inputs for* MC
2. *notReadyContexts* ← *not ready contexts of dependenceContexts*
3. ***if*** *notReadyContexts* = ∅ ***then***
   Pass MC *to the bindings resolver module*
   else
   Add information about MC *waiting for notReadyContexts*

On the other hand, inputs readiness guard must be signaled, when machines get ready, because their contexts may be awaited by some other machine contexts. Such signals are handled with the *MachineFinished()* method sketched by algorithm 4.4.

Algorithm 4.4 (Inputs readiness guard handler of finished machines)

Prototype: IRG.MachineFinished(M)
Input: *Machine* M *after its process finished.*
The algorithm:

1. ***for each*** awaited ∈ *contexts of* M ***do***
   for each context ∈ *contexts waiting for* awaited ***do***
   a. ***if*** context *has all inputs ready* ***then***
   Pass context *to the bindings resolver module*
   else
   Remove information about context *waiting for* awaited
2. *Remove information about contexts of* M

Every context, awaiting for the machine just finished, is checked if this is its last inaccessible input and if it is, then the context is passed to the next stage.

Inputs readiness guard keeps internally all the necessary information in appropriate data structures, including mappings from machines to their contexts and between

waiting and awaited contexts. Statements "add information" of algorithm 4.3 and "remove information" of algorithm 4.4 handle the internal data structures, so that only the information that may be needed is kept.

4.2.1.3 Resolving Input Bindings

Both main methods of inputs readiness guard, under appropriate circumstances, pass a context to the binding resolver module. The module is not displayed in Fig. 4.10, because it is just a simple step on the way from the inputs readiness guard to the unification system.

The goal of *inputs resolution* is to replace machine contexts directly connected to the inputs by references to proper machines (possibly remote), which exhibit the required outputs. The remoteness of the right machines means that the machines, that actually own the output, are not necessarily assigned to the machine contexts bound to the input at start. For example, an output of a scheme is just a "transit" from another machine output (usually a child of the scheme). The same occurs when any other machine exhibits output of a submachine as its own (it is quite common consequence of classical object oriented design of machines).

Resolved input bindings (RIB) contain precise information about the outputs providing input information to the requested machine. They may have one of two possible forms:

- a pair of machine identifier and its output name,
- an output identifier.

The first form is just a specification of the actual output. The second one is a consequence of using output-level unification (see Sect. 4.2.2), which is used to detect situations when two different machines exhibit equal outputs (then, two machines may be unified even if their inputs are bound to distinct but identical outputs). For such purposes, common outputs are given output identifiers and can be accessed independently from the machines they come from.

Resolving input bindings is the operation of transforming input bindings $B = \{input_i \rightarrow B_i | i = 1, \ldots, n\}$ to *resolved input bindings* being a mapping

$$RB = \{input_i \rightarrow RIB_i | i = 1, \ldots, n\}, \tag{4.4}$$

where RIB_i are collections of resolved input bindings B_i, defined as above.

After determining the resolved input bindings RB of a machine context C bindings, the pair (C, RB) is passed as a signal to the machine unification system.

4.2.2 Machine Unification System

Machine unification has been introduced to avoid wasting computational time and memory for two exactly the same machines. In advanced data mining project it is

inevitable that a machine with the same configuration and inputs is requested twice or even more times. It would not be right, if an intelligent data analysis system were running the same adaptive process more than once and kept two equivalent models in memory. Therefore, machine contexts have been introduced as objects separate from proper machines. Different contexts may request for the same machine and may share the machine. The goal of the unification system is to detect the possibilities of machine reuse and substituting the same machine to all the equivalent contexts, requesting the machine.

For the purposes of machine unification, all machines are registered in a *machine cache*. When a new request is analyzed by the unification module, the machine cache is checked, whether exactly the same machine has already been requested (and successfully finished) before, which means that the machine of the requested configuration and assigned the same resolved inputs already exists in the cache and may be reused.

When a request can not be unified with any machine already built, it is passed to the task spooler module, responsible for task distribution to special task running services. It does not mean that the possibilities of unification end here—the spooler must also be aware of unification and control if two tasks contain requests for the same machine. If two (or more) requests are unified before any of them is finalized, then two (or more) tasks of the same machine creation are pushed to the task spooler (with different priorities). Spooler services control the requests to prevent running two requests for the same machine in parallel and producing two identical machines. When the request of the highest priority is finished, all others are just substituted with the same machine. More details about the spooler, also in the context of machine unification, are presented in Sect. 4.2.3.1.

All unification efforts must be very carefully synchronized to guarantee that two identical machines are never created.

4.2.2.1 Unification Methods

Different requests are guaranteed to result in the same machine if:

- configurations of the machines (parameters of the processes) are equal and
- the inputs are bound to the same or equivalent outputs.

Such requests may be *unified*, which means that they may share the machine—the request being served later gets the machine from the cache and next instance of the machine is not run.

Configuration equivalence

The first condition is rather clear and unambiguous, though precise definition of parameter equality function is up to the author of the machine. To supply a properly defined machine configuration class, one has to include a method to determine

equality of two parameter structures of the same type, which is used by the unification system.

The term "machine parameters" includes information about pseudo-random processes performed by the machine (if any) and subconfigurations (if present). Therefore, a strategy to effectively deal with randomness in complex machine hierarchies has been worked out and easy-to-use tools provided for management of random processes within Intemi machines. The idea and its influence on the unification system and on fair results comparisons are described in Sect. 4.2.2.3.

Input equivalencence

Determining equivalenore complicated. First of all, inputs unification possibility may be judged only after the machine inputs are ready (machine contexts are not sufficient, the machines must have finished their processes). Therefore, the resolved inputs (as of Eq. 4.4) must be provided next to pure machine context (Eq. 4.2).

With respect to resolved inputs, natural two levels of unification, corresponding to analysis depth, can be offered:

1. Inputs must be bound to the same output to be regarded as equivalent.
2. Different inputs are compared to determine equivalence.

Naturally, switching unification completely off is a third possibility (zero level), but out of interest here. The first option is much simpler computationally than the second, because on its assumptions, it is sufficient to check if both input bindings being compared are assigned the same output (name) of the same machine.

Level 2 is more expensive computationally, because it must run comparison routines, which for large objects can be costly. Therefore, it is absolutely unreasonable to compare each resolved input of the machine to be created with all outputs of other machines, as it would mean pairwise comparisons of all outputs and would completely paralyze the system. Instead, a repository for outputs is created and outputs are given identifiers at registration in the repository. Equivalent outputs get the same identifier, so once added to the repository, they do not need to be compared with other outputs (from the repository). At registration time, new machine outputs are efficiently checked, whether they need new identifier or are equivalent to some outputs already present in the repository. The operation has two stages:

- first, the hash code is calculated for the output at hand and the repository is checked for other outputs of the same hash code,
- then, for each output registered in the cache and assigned the same hash code, proper comparison routine is called.

Therefore, it is very important that machine code authors provide high-quality routines for hash-code calculation. High quality of hash code generation means close to uniform probability distribution, so that repetitions of hash codes are very unlikely and the number of calls to comparison routines is as small as possible.

There is a number of ways to further speed up verification of input equivalence. For example:

- The outputs can be indexed by their types, so that smaller repository of outputs of the same type is examined. This helps estimate quality of hash code generation routine for each type by analysis of the frequency of repetitions.
- Late output registration, that is, adding outputs to the repository not instantly after their machine process is finished, but later, when the output is really used by another machine context. It can be very advantageous because unused outputs would just unnecessarily increase the repository and slow down the unification procedures.

Naturally, level 1 is also a part of level 2, thus detailed tests are performed only for the bindings that have any chance to be different.

Outputs like classification correctness (binary vectors showing which decisions of a classifier are correct and which are not) are always very similar, so rough methods of hash code calculation may result in distributions, very different than uniform. Since they are very seldom bound to inputs of other machines, they can be easily excluded from the unification analysis described above. Actually, the analysis is off by default, and to switch it on, the programmer must give a special attribute to the output class and implement special interface, containing the methods used by the unification processes.

Memory cache and disc cache

All machines are deposited in machine cache just after they are ready. To make machine search as fast as possible, the cache is built from three hash dictionaries to realize three types of mappings:

- *unificator* dictionary, mapping (C, RB) pairs to unique machine identifiers,
- *unificatorRev* dictionary, providing the mapping inverse to *unificator* (from machine identifiers to (C, RB) pairs),
- *cache* dictionary, mapping machine identifiers to machines.

Given machine configuration C and resolved input bindings RB, the machine cache may provide compatible machine only if the *unificator* dictionary contains appropriate (C, RB) key. Therefore, verification of machine availability is fast.

The three hash dictionaries obviously need fast calculation of hash codes with distribution close to uniform. If this condition is satisfied, they guarantee attractive time complexity of the access to machines, independent of the number of machines in the cache (very important for scalability of data mining systems). This means that each machine configuration has to implement two methods: verifying the equality of two configurations and the hash function of the configuration. Provided high quality of the methods, the maintenance of the unification system costs very little.

Although access to machines in the cache is independent of the cache size, available memory is always limited. It is advantageous to keep all the machines run within the project, but for large projects, containing thousands or even millions machines,

it would result in too much memory consumption. Therefore a *disk cache* cooperating with the memory cache has been introduced. If possible, machines are kept in memory, but to make more memory available, they are swapped to the disc cache.

In fact, each machine is scheduled for saving in the disc cache immediately after its adaptive process is finished. It is kept in memory as long as it is needed by any other machine. The information interchange through inputs and outputs is a subject to open–close management, so that the system receives all information about machines and outputs usage. The requirement to open and close machine inputs facilitates also optimization of machine exchange between the project and computational servers running the project tasks (possibly remote servers).

Unique, specialized structure of the disk cache and its management guarantee that the access to machines (loading/saving) does not depend on the number of machines in the disc cache but just on the length of binary representation of the machine. It keeps the whole unification and cache system as efficient as possible.

A question might be asked here: why to introduce a disc cache instead of relying on the operating system virtual memory mechanisms? The answer is that our internally managed disk cache can be much more effective because it can take advantage of the information about project internal structure and act without any delay that could disable performing the tasks scheduled in the project. Moreover, the operating system cache has no information about which machines are necessary at the moment or will be necessary in further steps, in contrary to the Intemi system, that has full knowledge about that, because each machine usage is registered. The knowledge lets the system decide whether given machine should be kept in the memory cache or just in the disc cache or should be completely discarded from the cache while operating system's virtual memory manager has no such information.

4.2.2.2 Example of Advantages

To better see the advantages of machine unification, imagine a project to test and compare suitability of different feature selection methods for a classification problem. Provided the design of feature selection machines, described in Sect. 4.1.1, the comparison concerns feature ranking methods rather than feature selection methods. To make the test credible, we should, for instance, perform a cross-validation of complex machines consisting of feature ranking, feature selection and classification machines. The complex machine could have the form of Transform and classify machine (see Fig. 4.6) with transformation scheme replaced by the feature selection template (of Fig. 4.3) filled with proper feature ranking. When performing the CV test with different ranking machines, it may turn out that the selection of topmost features of different rankings results in the same set of features, so it makes no sense to train and test the classifier twice for the same data.

Table 4.1 shows feature rankings obtained for Wisconsin breast cancer data from the UCI repository Frank and Asuncion (2010), with eight different methods: three based on indices estimating feature eligibility for target prediction (F-score, correlation coefficient and entropy based mutual information index), one based on internals

Table 4.1 Feature rankings for UCI Wisconsin breast cancer data

Ranking method	Feature ranking								
F-score	6	3	2	7	1	8	4	5	9
Correlation coefficient	3	2	6	7	1	8	4	5	9
Information theory	2	3	6	7	5	8	1	4	9
SVM	6	1	3	7	9	4	8	5	2
DT, Gini	2	6	8	1	5	4	7	3	9
DT, information gain	2	6	1	7	3	4	8	5	9
DT, information gain ratio	2	6	1	5	7	4	3	8	9
DT, SSV	2	6	1	8	7	4	5	3	9

of trained SVM model and four based on decision trees using different split criteria (Gini index, information gain, information gain ratio and SSV). To test a classifier on all sets of top-ranked features for each of the eight rankings, we would need to perform 72 tests, if we did not control subsets identity. An analysis of the 72 subsets brings a conclusion that there are only 37 different sets of top-ranked features, so we can avoid 35 repeated calculations. Very similar relative savings occur, when the rankings are determined inside the CV test (for each training sample, not for the whole dataset as in the case of rankings presented in the table).

Naturally, being aware of saving possibilities, one can design the project in such a way, that different rankings are analyzed first to check such redundancies and avoid them, but it requires programming a special machine to perform the test in proper way. System feature of avoiding repeated calculations eliminates the overload without any special effort from the user and is a general and very efficient solution which may help in many other circumstances.

In advanced meta-learning, it would be highly nontrivial to predict all the possibilities of repeated machines and prevent them in advance (for a more complex example of machine unification advantages see Sect. 4.4.1). To maximize the gains, Intemi solves the unification problem at kernel level. Each machine created and run within the project is registered in machine cache. Each request for a new machine is checked against the machine cache, whether the machine is already available, and if it is, the machine request gets substituted with the ready machine and the request is not sent to the task spooler. Using hash codes when comparing different machine contexts makes the cost of machine cache management very small, so the overall balance is definitely positive.

All the automated solutions do not cause that the authors of machine implementations do not need to care for unification in any way. In the example shown above, almost 50 % of tests could be avoided, because feature rankings could be verified, in advance, as equivalent. When two methods return the same set of top n features, but the feature order is different, then treating them as sequences, not as sets, would prevent the savings. It is up to machine authors, how they organize machine outputs and how they implement equivalence test methods. Keeping in mind the power of machine unification, one can build very efficient machines and projects.

4.2.2.3 Unification of Random Processes

The strategy of random processes management is an aspect, which seems minor, but gets very important when more advanced data analysis is to be performed. It had not been completely solved by any data mining package before Intemi. It is common to include a seed value (controlling the randomness) inside configuration structures of CI algorithms. Then, the seed value can be set arbitrarily or chosen randomly (usually on the basis of current time to provide better randomness), however in the case of complex machines, it is not satisfactory, especially when the mechanisms of machine unification are expected to be maximally functional. Intemi design of seed control aimed to facilitate:

- unification of complex, multi-level machines, possibly with pseudo-random behavior of machines at different levels,
- robust comparisons of different CI methods, based on testing machines on exactly the same data, even when the tests are performed within distinct projects.

These two features are not guaranteed when it is only possible to configure machines to use either fixed or random seed. For example: to perform 10 repetitions of 10-fold cross-validation, just one machine providing cross-validation data is configured and the same configuration is used 10 times. If the seed property is set to "fixed", each repetition results in the same sets of training and test data, and exactly the same 10-fold CV is performed 10 times. If the seed is set to "random" (dependent on time), it is not be possible to repeat the same splits in further tests. This reveals the need for a system that allows the subsequent CVs to run with different seeds and makes it possible to repeat the whole procedure with the same sequence of seeds. The same need occurs, for example, when performing a CV of a neural network initialized by random weights.

For full functionality, the seed control should be done in three ways:

1. The seed is fixed to a given integer value.
2. The seed value is determined by a pseudo-random number generator initialized with the time of machine preparation.
3. The seed is managed automatically by the system to reflect the seed settings of parent machines.

Intemi implements the functionality exactly in this way. The third option of automatic seed management means that the machines can get different seeds in a machine branch, but the seeds depend on the seed of the root machine of the branch and the child machine indices. When a 10-fold CV is to be run 10 times independently, but in such a way that can be repeated at any time, the repeater machine should get a fixed seed and the auto mode should be assigned to the seed of the machine generating data for CV. With such configuration, each repeated CV gets different seed, but dependent on the seed of the repeater. This provides both diversity of different CV repetitions and possibility to repeat the whole procedure. The same 10 times repeated 10-fold CV can be obtained later with the same configuration of the seed for the repeater machine and auto mode for seed control at the CV data generator. Moreover, when

the whole scenario is repeated for exactly the same configuration of a classifier, the whole repeater machine may be unified with the former one and no calculations are necessary because the whole structure is already available.

The possibility of performing tests with exactly the same training and test data, also in completely different projects, opens the gates to the most adequate methods for comparison of different machines results, including paired t-test, Wilcoxon test or even McNemar's test.

4.2.3 Task Management

When all inputs of a requested machine are ready (the processes of all machines providing necessary outputs are finished) and machine can not be substituted by another one, already created and residing in machine cache, the machine context is equipped with proper task information and commissioned to the *task spooler*, where it waits for its turn and for a free processing thread.

The processes of task spooling are quite nontrivial, because different requests may be subject to unification with other requests in the spooler. For example, canceling a task must be properly managed to not cancel other requests for the same machine. Also the structure of the spooler is very important from the point of view of efficiency. Although the order of task running can not affect the resulting models (the tasks are added to the spooler when they are ready to run, and the time when they are started is not significant unless the seeds of random processes are time-dependent), it is important from the point of view of memory and time consumption.

The spooler cooperates with *task managers* which run computational threads, and when a thread is free, request for available tasks and start them.

4.2.3.1 Task Spooler

The task spooler of Intemi is not a simple standard queue. To optimize the efficiency of task running, a system of hierarchical priorities has been introduced. Each parent machine can assign priorities to its children, so that they can be run in proper order. It prevents starting many unrelated tasks in parallel, and as a result, reduces consumption of memory and computation time.

Because of the priorities and machine unification mechanisms, which may result in spooling the same task twice with different priorities, the task spooling system gets quite different than operating systems task schedulers. The number of tasks may be quite large and the spooler can not use the policy of equal CPU time gratification, because it would very often lead to memory exhaustion. As a result, the main part of the spooler has the form of tree, containing nodes with priorities. Apart from the tree, there is a container for machine requests waiting for machines being run for the sake of other contexts—when a machine is running, other requests for the same

machine may not be run, because it would violate the rule of not creating the same machine twice.

When a task gets its turn, it is popped out from the spooler and if the machine ordered within the task is not currently running, it is created and run. When a task related to another context of the same machine is currently running, the new task for the same machine must wait until the other context is fully serviced. If the other context finishes successfully, the machine is assigned to all its contexts, otherwise (that is, when the machine process of the other context gets aborted) the waiting task is started in the same way as when it is not unified with any other task. The functionality described above is succinctly expressed as algorithm 4.5. The main loop, described above, is contained in item 2. Two collections (*WaitingForOthers* and *StoppedWaitingForOthers*) require an additional explanation. When a task is popped from the spooler but can not be run because other instance of the same machine is already running, then the task is moved to the (*WaitingForOthers* collection. When the awaited task finished, a signal is sent and the waiting task is moved to *StoppedWaitingForOthers*) collection. The loop in item 1 handles the *StoppedWaitingForOthers* collection by checking items one by one and changing status to *Substituted* when the awaited task finished correctly or returning the task if the awaited task was aborted.

Algorithm 4.5 (Spooler method to pop a task)

Prototype: ***GetNextTask()***
Input: *None.*
Output: *A task to run or* $\perp$ *if no task available.*
The algorithm:

1. ***while*** *StoppedWaitingForOthers* $\neq \emptyset$ ***do***
   a. *task* $\leftarrow$ *pop task from StoppedWaitingForOthers*
   b. ***if*** *the other task aborted* ***then***
      return *task*
   c. *task.Status* $\leftarrow$ *Substituted /* the machine is ready */*
2. ***while*** *spooler* $\neq \emptyset$ ***do***
   a. *task* $\leftarrow$ *pop task from the spooler*
   b. ***if*** *task's machine is finished in another context* ***then***
      task.Status $\leftarrow$ *Substituted;* ***continue***
   c. ***if*** *task's machine is running in another context* ***then***
      push task to WaitingForOthers
   d. ***return*** *task*
3. ***return*** $\perp$

The method *GetNextTask()* is called whenever a task manager finds a free thread and needs a new task to run.

After the machine process is finished, the machine gets the status "*ready*". It is not the end of its life, but from the point of view of its different states it is the final

state—the machine results and outputs can be exploited, but the machine does not change anymore.

The whole machine life cycle is managed completely automatically. From the user's point of view, only the start and the end of the path, machine goes through, must be taken care of, that is, the user orders a machine providing its configuration and input bindings, and then just waits for the machine (or for a collection of submachines) to be ready for further analysis of the created model(s).

4.2.3.2 Task Running

Task managers (computation servers) pop machine-creation requests from task spoolers and run machines in threads. The idea is depicted in Fig. 4.11. Each project (more precisely, its task spooler) can subscribe to services of any number of *task managers* executed on either local or remote computers. Moreover, subscribing to and unsubscribing from task managers may be performed at project run time, so the CPU power can be assigned dynamically. Each task manager serves the computational power to any number of projects. Task managers run a number of threads in parallel to make all the CPU power available to the projects. Each project and each task manager presented in Fig. 4.11 may be executed on different computer, but it is also possible to integrate projects and task managers even in a single application.

Task threads run machine processes one by one. When one task is finished, the thread queries for another task to run. If a task enters waiting mode (a machine requests some submachines and waits until they are ready) the task manager is informed about it and starts another task thread, to keep the number of truly running processes constant.

Machine tasks may need information from other machines of the project (for example, input providers or submachines). In the case of remote task managers a *project proxy* is created to supply the necessary project machines to the remote computer. To optimize the information flow, only the necessary data is marshaled.

Naturally, all the operations are conducted automatically by the system. The only duty of a project author is to subscribe to and unsubscribe from task man-

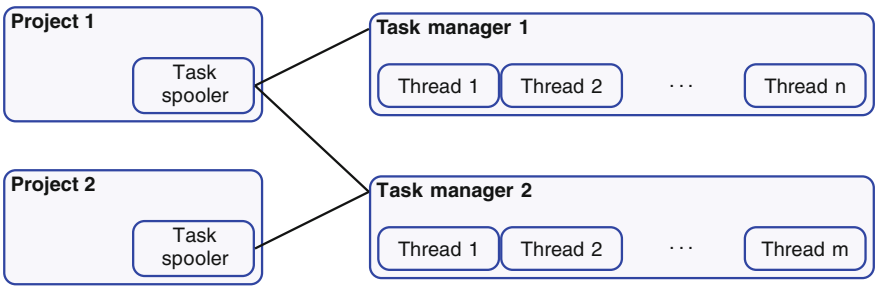


Fig. 4.11 Two projects and two task managers

ager services—each is just a single method call. Similarly, an author of a complex machine needs just to request for submachines and call a *Wait()* method to suspend operation until the ordered machines are ready.

4.2.3.3 Example of Advantages

To observe the advantages of Intemi spooling system in comparison to standard queue, let's analyze the progress of calculating 10 repetitions of 10-fold CV to compare classification accuracy of kNN and SVM algorithms. Such configuration results in a repeater machine as presented in Fig. 4.9, but with 10 distributor schemes instead of 2 and 10 test schemes per distributor scheme in place of 2. The resulting machine hierarchy is sketched in Fig. 4.12. The repeater machine creates 10 distributor schemes (ds_1–ds_{10}) and 100 test schemes (ts_1^1–$ts_{10}^1 \cdots ts_1^{10}$–$ts_{10}^{10}$). Each distributor scheme creates one cross-validation distributor (cv_i) and each test scheme requests 4 child machines: kNN (k_j^i), SVM (s_j^i) and classification tests of kNN (kt_j^i) and SVM (st_j^i).

To avoid nondeterministic behavior of the process, due to parallel calculation, all the tasks were calculated by one task manager with one running thread. The request for the repeater machine pushes the root node of the tree (of Fig. 4.12) into the spooler. When the request is popped out, the repeater process is run and requests all its children (which puts the requests in the queue): the first distribution scheme (ds_1), 10 test schemes (ts_1^1,…,ts_{10}^1) bound to ds_1 outputs, the second distribution scheme (ds_2), 10 test schemes of the second CV and so on. Thus, 110 machine requests go to the spooler. After that, the repeater starts waiting for the children. To avoid a lock, the task manager receives the information that its thread started waiting and fires up a new thread. The new thread starts and requests a task from the spooler (by the agency of the task manager). The *GetNextTask()* method (algorithm 4.5) returns ds_1. The distributor scheme (in its process) requests the CV distributor machine (cv_1) and starts waiting until cv_1 is ready.

When a standard queue is used as the spooler, there are 109 requests in the queue before cv_1, so it is run after all the 109 preceding requests are popped, run and start waiting after pushing all their requests for child machines to the spooler. It means

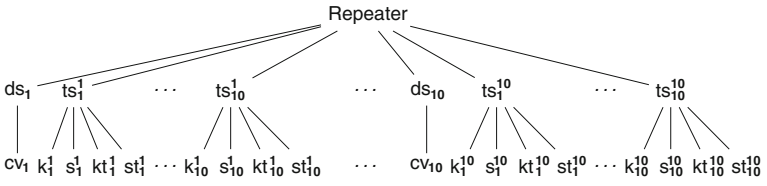


Fig. 4.12 Repeater submachine tree performing 10×10-fold CV of tests defined in Fig. 4.9. *ds* distributor scheme, *ts* test scheme, *cv* CV distributor, *k* kNN, *kt* classification test for kNN, *s* SVM, *st* classification test for SVM

that when cv_1 gets its turn to run, 111 threads are in waiting mode (the repeater machine and all 110 of its children) and all the 410 machines of the third level are in the queue. So, the task manager controls 112 task threads. It costs a lot: the operating system must deal with many waiting threads and all the started machines occupy memory at the same time (usually, machines need more memory during the process than after it—when the model is ready).

With Intemi spooling system based on tree with ordered nodes, the history of machine requests and pops is quite different. Only the begin is similar, because the repeater machine is popped, run and it requests its 110 children. Then, ds_1 is popped out and run. It pushes cv_1 to the spooler and starts waiting. Next pop from the spooler returns not ts_1^1 as in the case of a standard queue, but cv_1, because the branch of ds_1 is favored over all the other children of the repeater. When cv_1 is finished, ds_1 can be finished too and ts_1^1 is run. It requests its 4 children, which are finished before ts_2^1 is started, thanks to the spooling system based on ordered tree. As a result, only two waiting machine processes and one running may be observed at the same time, so the task manager controls only 3 threads. This is because the machines are popped from the spooler in the following order:

$$\mathsf{Repeater}, \mathsf{ds}_1, \mathsf{cv}_1, \mathsf{ts}_1^1, \mathsf{k}_1^1, \mathsf{s}_1^1, \mathsf{kt}_1^1, \mathsf{st}_1^1, \ldots, \mathsf{ts}_{10}^1, \mathsf{k}_{10}^1, \mathsf{s}_{10}^1, \mathsf{kt}_{10}^1, \mathsf{st}_{10}^1, \ldots, \mathsf{ds}_{10}, \mathsf{cv}_{10}, \mathsf{ts}_1^{10}, \mathsf{k}_1^{10}, \mathsf{s}_1^{10}, \mathsf{kt}_1^{10}, \mathsf{st}_1^{10}, \ldots, \mathsf{ts}_{10}^{10}, \mathsf{k}_{10}^{10}, \mathsf{s}_{10}^{10}, \mathsf{kt}_{10}^{10}, \mathsf{st}_{10}^{10},$$

while in the case of a standard FIFO the order is:

$$\mathsf{Repeater}, \mathsf{ds}_1, \mathsf{ts}_1^1, \ldots, \mathsf{ts}_{10}^1, \ldots, \mathsf{ds}_{10}, \mathsf{ts}_1^{10}, \ldots, \mathsf{ts}_{10}^{10}, \mathsf{cv}_1, \mathsf{k}_1^1, \mathsf{s}_1^1, \mathsf{kt}_1^1, \mathsf{st}_1^1, \ldots, \mathsf{k}_{10}^1, \mathsf{s}_{10}^1, \mathsf{kt}_{10}^1, \mathsf{st}_{10}^1, \ldots, \mathsf{cv}_{10}, \mathsf{k}_1^{10}, \mathsf{s}_1^{10}, \mathsf{kt}_1^{10}, \mathsf{st}_1^{10}, \ldots, \mathsf{k}_{10}^{10}, \mathsf{s}_{10}^{10}, \mathsf{kt}_{10}^{10}, \mathsf{st}_{10}^{10}.$$

Since, thanks to the spooling system, Intemi keeps just three running machines at a time, both memory and CPU time are saved significantly. When running this example on the UCI Wisconsin breast cancer data, the peak memory usage was about 30 MB, while with the standard queue it was over 160 MB. Also the overall time used by the project was significantly reduced (around 15 %) although the calculations were exactly the same. The CPU time gain is apparently due to handling the process with smaller number of threads and less memory consumption, which turned out to be so smaller burden for the operating system.

4.3 Results and Query System

Apart from efficient machine creation and running, a successful data mining system must provide convenient tools for handling machine results. From the point of view of meta-learning applications, it is extremely important to manage the results in a unified way, facilitating analysis of results coming from machines completely unknown to the meta-learners.

As described before, machine outputs are handled in a standardized way, independently of particular machine peculiarities. Also the results that are not expected to

have the form of outputs can be deposited in a standard manner. Intemi implements special results repository, where machines and special objects named *commentators* can deposit relevant information.

The information stored in results repository can be accessed directly (it can be called a low level access) or by running a query (definitely recommended) to collect the necessary information from a machine subtree.

Queries return *series* of results in especially designed containers, which can be easily transformed with numerous *series transformations*.

The system is open to extensions with new methods to control queries and transform result series. General mechanisms are offered by the system, together with the most commonly used methods, but specific behavior can always be obtained with user's implementations, usually without much effort.

4.3.1 Results Repositories

Results in the form of name-to-object mappings are stored by Intemi within a repository integrated with the project. Standard form of results collections makes them easily accessible and manageable, due to the general, uniform tools. Putting the results into the repository is advantageous also from the perspective of memory usage. Machines can be discarded from memory when no other machine needs their outputs, while the results and comments repositories stay in memory and are available for further analysis. Therefore, results repository should be used with moderation.

Intemi provides three standard ways of exposing such information:

- depositing to the machine's results repository by the machine itself,
- commenting machines by their parent machines,
- commenting machines by commentators.

The three methods of adding information have been implemented to allow machines to put just the most important information into the repository. This is not a loss of possibilities, because more information can be added later, after machines are finished, by machine commentators. The idea of parent's comments on child machines is slightly different than the two others. Such comments describe the child role, seen from the parent's point of view, not the details learnt by the machine.

Machine-deposited results

Results repository consists of small dictionaries mapping string labels to the results objects, created by the system for particular machines. Machines can put there the information describing the model, the learning procedure and so on. Adding results to the repository is performed by calling the *AddToResultsRepository()* method of *Machine* class (each machine class is obliged to derive from it). For example, the Classification test machine adds calculated accuracy to the repository in the following way:

```
AddToResultsRepository("Accuracy" accuracy);
```

There is no limit for the number of elements that can be put into the repository or their size, however it is advisable to put there only the most important information, in order to save memory. When a machine is swapped to the disc cache, its result repository entries are still kept in memory, so that the most important information about the model is available without the need to restore the whole machine.

Parent's comments

Each machine can comment its submachines to augment further analysis of the submachines structure. For example, the repeater of Fig. 4.9 comments each of its submachines with labels denoting which repetition and which CV fold the subscheme belongs to. Thanks to this, we can run queries filtering appropriate results, for example, we can select all the accuracies of the first CV cycle or all the accuracies of the second folds of each CV cycle and so on.

A machine can comment a child with a call to the *AddChildComments()* method of the *Machine* class. The child index, comment label and comment value must be passed to the method. In the case of the example mentioned above, the repeater comments each subscheme with the code:

```
AddChildComments(submachineId, "Repetition", group + 1);
AddChildComments(submachineId, "Fold", fold + 1);
```

The "+1" shifts of indices convert from 0-based to 1-based index system.

It is important to realize that the comments assigned by parent machines to their children are not attached to machine, but to the parent–child link (edge instead of node in the terms of graph theory). When machine unification occurs, the machine and its results are common for different contexts, but the parent's comments may be different.

Commentators

Queries collect and series transformers analyze data gathered in results repositories. Because, by default, machines should put in the repositories only the most important information, additional tools are provided to facilitate adding machine description items to the repository, when necessary. Such additional information is deposited in the system as a *comment* on a machine.

Each machine can order making comments on its successors. The order can be attached to the request for creation of the submachines, in the form of objects called *commentators*.

To continue the example of the kNN and SVM test, if we need to perform a McNemar's test of statistical significance of difference between performance of kNN and SVM, we need detailed information about correctness of each single answer of each machine. The best way to achieve this is to request comments on both classification tests. The *CorrectnessCommentator* comments a classification test machine with a

binary vector of the length equal to the number of objects in the test data and containing a 1 for each correct answer and 0 for incorrect answer. It would be wasteful to generate and keep such vectors for each classification test machine, so it can be done on demand by means of commentators.

Comments do not need to be requested at the time, the commented machine is requested. It is a very important advantage, that also for machine hierarchies built earlier, one can easily apply a search for machines qualifying for comments and run given commentators for them. The idea of the technique is the same as the one of queries defined in the following section.

Similarly to machine-deposited results and parent comments, comments made by objects like the *CorrectnessCommentator*, can be collected with queries, and transformed with series transformations, as described below.

4.3.2 Query System

To simplify the management of submachines results, which constitute a tree hierarchy, Intemi provides uniform tools for quick and easy results collection and analysis. A *series* of results selected from a machine tree can be obtained by running a *query*. A query is defined by:

- the *root machine* of the query search (root of the searched subtree),
- a *qualifier*, that is, a filtering object—the one that decides whether an item corresponding to a machine in the tree is added to the result series or not,
- a *labeler*, that is, the object collecting the *results objects* that describe a machine qualified to the result series.

Running a query means performing a search through the tree structure of submachines of the root machine, and collecting a dictionary of label-value mappings (the task of the labeler) for each tree node qualified by the qualifier.

For example, consider a repeater machine producing a hierarchy of submachines as in Fig. 4.9. After the repeater is finished, its parent needs to collect all the accuracies of SVM machines, so it runs the following code:

```
Query.Series results = Query(repeaterCapsule,
    new Query.Qualifier.RootSubconfig(1, 3),
    new Query.Labeler.FixedLabelList("Accuracy"));
```

The method *Query* takes three parameters: the first *repeaterCapsule* is the result of the *CreateChild()* method which had to be called to create the repeater, the second defines the qualifier and the third—the labeler. The qualifier *RootSubconfig* selects the submachines which were generated from the subconfiguration of repeater corresponding to path “1, 3”. The two-element path means that the source configuration is the subconfiguration 3 of subconfiguration 1 of the repeater. A look at the repeater configuration in Fig. 4.8 clarifies that subconfiguration 1 of the repeater is the configuration of the test scheme (0-based indices are used) and its subconfiguration 3

is the SVM Classification test. Thus, the qualifier accepts all the machines generated on the basis of the configuration Classification test taking *Classifier* input from SVM machine. These are classification tests, so they put *"Accuracy"* to the results repository. The labeler *FixedLabelList* of the example simply describes each selected machine by the object put into the results repository with label *"Accuracy"*. As a result, we obtain a series of four descriptions containing mappings of the label *"Accuracy"* to the floating point value of the accuracy.

Intemi provides a number of qualifiers and labelers, most likely to be needed by researchers. For example, instead of the *RootSubconfig(1, 3)* qualifier, one could use *ConfigType(**typeof**(ClassTestConfig))*. This would collect the results for all the machine tree nodes for which *ClassTestConfig* is the configuration class, that is, from all the classification test machines. The result series would contain not four but eight elements since both Classification test machines (taking *Classifier* input from kNN and SVM machines) would be qualified by the query.

The labelers are given access to the whole path of machine results: from the query root submachines to the leaves of the machine tree. Thus, the labelers can use labels from any level. For instance, the labeler *AllLabels* collects all the labels commenting the whole path. It lets us easily collect, for example, the accuracies calculated by the classification test machines and fold identifiers given by the repeater machine to its submachines (distribution and test schemes).

Partial results of a $n \times 5$ CV are presented in Table 4.2. The *"Repetition"* and *"Fold"* entries are the comments made by the repeater on its submachines.

4.3.3 Series and Series Transformations

The result of a query is contained within an object of *Series* class. The series is a collection of label-value pairs describing subsequent items. In the case of the series presented in Table 4.2, each item is described by three values assigned to the labels *"Accuracy"*, *"Repetition"* and *"Fold"* respectively.

The series returned by queries are often just intermediate results that undergo more or less sophisticated analysis. Each series can be transformed by specialized objects called *series transformations*. The transformations get a number of series objects and return another series object. One of the basic transformations is *BasicStatistics*

Table 4.2 Partial results obtained with labeler *AllLabels* for a $n \times 5$ CV

Item	1	2	3	4	5	6	7	8	9	10	11	…
Accuracy	0.93	1.0	0.97	0.93	0.97	0.9	0.97	1	1.0	1.0	0.97	…
Repetition	1	1	1	1	1	2	2	2	2	2	3	…
Fold	1	2	3	4	5	1	2	3	4	5	1	…

which transforms a series into a single item series containing the information about minimum, mean, maximum values and standard deviation.

Quite advanced manipulation of series of results can be performed with groups related transformations. As mentioned before, when a query uses a *ConfigType* qualifier pointing to the configuration type of classification test, it collects all the classification test results (regardless which classifier it tests—kNN or SVM in the example shown before). So, a series obtained in this way (say *allResults*) contains descriptions of kNN classification tests and SVM classification tests (by turns). We can easily create two groups by separating kNN results from SVN results with a call to the *GroupModulo* transformation:

```
Series inGroups = allResults.Transform(new GroupModulo(2));
```

It is also easy to group series items containing common values for particular label. For example, when we repeat 7 times a 10-fold CV and collect all available labels from classification test machines, we can group the resulting series by the *"Repetition"* with:

```
Series repetitions = allResults.Transform(new Group("Repetition"));
```

and obtain a series of 7 series containing separated results of each repetition of the whole CV cycle.

A grouped series can be traversed to transform each of its subseries. The *MAP* transformation performs a given transformation on all the subseries. For example, running the code

```
Series stats = repetitions.Transform(new MAP(new BasicStatistics));
```

calculates basic statistics for all the series within *repetitions*. The result remains a series of series (as *repetitions* was), so it must be ungrouped with the *Ungroup* transformation, to obtain a plain series of basic CV statistics for each repetition.

A shorthand notation is available to speed up writing code for such manipulations:

```
Series flatStats =
  allResults.Group("Repetition").MAP(new BasicStatistics()).Ungroup();
```

Several other methods of grouping result items have also been implemented in Intemi. For example, they can split a series into several series of given sizes or select some groups of results while dropping others, also in cycles.

One of the labels contained within the series is selected as the *main label*. Some transformations are focused on the items with main label or treat them in a special way in the transformations. For example, the *BasicStatistics* transformation calculates the statistics for the values assigned to the main label.

There are also basic arithmetic operators available and they also act on the main label values. They can be used in a natural manner as infix operators. For example, the code

```
Series diff = kNNResults − SVNResults;
```

calculates the differences of accuracies (assuming that the *"Accuracy"* is the main label in the series *kNNResults* and *SVNResults*). An example of such operation on results of 10-fold CV (performed once) is shown in Table 4.3.

One of the most important aspects of such results manipulation is easy testing of statistical hypotheses about the results differences. Thanks to universality of proposed ideas, we can easily run statistical tests like t-test, paired t-test, Wilcoxon test, McNemar's test and others. They are implemented as series transformers, so calling them may be as simple as:

```
Series tTest = new TTest().Transform(kNNResults, SVNResults);
Series tTestP = new TTestPaired().Transform(kNNResults, SVNResults);
```

The returned objects are new series containing information about the test results. *TTest* and *TTestPaired* transformations return series with single items labeled with *"t value"* and *"p value"* presenting the value of calculated statistic (t) and the estimated probability of the null hypothesis (about equality of the means) being true. The results calculated for the 10-fold CV accuracies from Table 4.3 are presented in Table 4.4.

In this case, the t-test allows to reject the null hypothesis with 95 % confidence ($\alpha = 0.05$), but not with 99 % confidence ($\alpha = 0.01$). The paired t-test, provided with the information about differences in subsequent passes, confirms that the results are statistically significantly different with more than 99 % confidence.

To perform McNemar's test (for the example scenario testing kNN and SVM with 10-fold CV), binary vectors of classification correctness for each of the two methods are needed. McNemar's test is run for two corresponding samples. Here, we have 10 pairs of data samples, but according to CV, each object from the original data occurs in exactly one test data sample, so merging all CV test samples is very reasonable. On the assumption that the *CorrectnessCommentator* has been run for each classification test machine, an example C# code that performs the McNemar's test can be quite short and readable:

```
Series s = project.Query(repeaterCaps,
```

Table 4.3 5NN and SVN accuracies (in %) for 10-fold CV for UCI image segmentation data

Fold	1	2	3	4	5	6	7	8	9	10	Mean ± St.dev.
5NN	95.2	81.0	90.5	81.0	95.2	81.0	90.5	85.7	90.5	90.5	88.1 ± 5.6
SVM	81.0	76.2	90.5	71.4	81.0	76.2	90.5	76.2	90.5	81.0	81.4 ± 6.9
5NN-SVM	14.2	4.8	0.0	9.6	14.2	4.8	0.0	9.5	0.0	9.5	6.7 ± 5.6

Table 4.4 Statistical significance tests results

Test name	t-test	paired t-test	McNemar
statistic	2.370	3.772	12.25
p-value	0.0292	0.0044	0.0005

```
        new ConfigType(typeof(ClassTestConfig)),
        new FixedLabelList("Correctness"));
    s = s.GroupModulo(2).MAP(new Unpack());
    Series r = new McNemar().Transform(s[0], s[1]);
```

First three lines define the query to collect just the *"Correctness"* results from all classification test machines. The fourth line groups the results of kNN and SVM and maps each of the two groups with *Unpack* transformation which unfolds all given collections of binary results added by the *CorrectnessCommentator* into a single, long series. The fifth line runs McNemar's test on the two long series of correctness binary flags and returns the χ^2 statistic value and the p-value as two values in a single item series. Last column of the Table 4.4 contains the results of the McNemar's test run for the example being discussed.

The mechanisms of query and series facilitate such analyses with very simple means. There are many more predefined series transformations and new transformations can be easily implemented.

4.4 Meta-Learning Support

Intemi framework has been designed with special emphasis on the needs of meta-learning. Therefore, it provides an abundance of tools for efficient and friendly meta-level operation including:

- easy machine configuration manipulation,
- pre-defined machines for testing other machines, including the *Repeater* machine, which can serve as a basis for such techniques like stacking, bagging, boosting and others,
- easy test results collection and queries for selected results,
- series transformations for easy reorganization of the results, comparisons, testing statistical hypotheses and so on,
- efficient machine creation and running with general unification system working at any level of machine hierarchies.

Intelligent machines for learning at meta-level can be easily constructed and tested. In some simple applications, a machine performing meta parameter search, described below, can be very useful, although it can not be regarded as an advanced meta-learning algorithm.

4.4.1 Meta Parameter Search

One of the first steps toward advanced meta-learning is a machine capable of efficient searching in the space of model parameters. Various data mining systems introduce

such tools, but the implementations are either very limited or look like external patches that do not fit the overall inner architecture. Shortages of the engine-level design do not allow for advanced meta-learning in a natural way. When Intemi was being designed, the meta-learning requirements decided about many solutions at the engine level of the system (like those presented in preceding sections), so meta-learning machines are built in the same manner as all base-level machines and are served in ordinary ways. Simple parameter search machine is the first step toward more sophisticated meta-learning and constitutes a good illustration of cooperation of different mechanisms described above.

The aim of the *meta parameter search* (MPS) machine is to repeatedly create a submachine and test different values of parameters included in the submachine configuration. The algorithm of MPS machine allows for testing arbitrarily complex submachines and can search for optimal values of parameters of any part of the submachine configuration hierarchy.

The configuration of the parameter search machine includes:

- a *test configuration* defining arbitrarily complex machine structure,
- a *scenario* of parameter changes,
- a *query specification* which determines the way of estimation of the test results.

The process realized by the meta parameter search machine creates a sequence of submachines for the test configuration with some parameters changed in each iteration, according to the configured scenario. In each pass, it

- gets a candidate configuration from the scenario object,
- creates a submachine to test the configuration,
- runs a query on the submachine branch to collect proper results,
- transforms the series of values resulting from the query by given series transformer to obtain the final measure of the configuration parameters performance,
- passes back the performance result to the parameter changes scenario object, so that it can adjust the process of producing subsequent machine configurations.

The main part of the parameter search is presented as algorithm 4.6. In the main loop (item 2), it repeatedly creates test machines (usually a complex hierarchy) returned by the scenario of parameter changes. When the test is finished, it runs the same query for each iteration, to collect the crucial results obtained with current parameters. The results (usually a collection of accuracy estimations) are then transformed to obtain one real value as the overall estimation of current configuration. After all tests (for all parameter settings returned by the scenario) are finished and their results examined, the best configuration is returned as the main result of the search.

Normally, the test submachine configuration defines the whole test process, for example, multiple training and test of a classifier, so it is convenient to derive it from a test template scheme, introduced in Sect. 4.1.2, by filling placeholders with proper machine configurations, for example, embedding a classifier configuration in a cross-validation test template.

Algorithm 4.6 (Meta parameter search process)

Prototype: ***MPS.Search(C,S,Q,T)***
Input: *Configuration of test machine C, scenario of parameter changes S, query collecting results Q, series transformation calculating the final quality estimation T.*
Output: *Best configuration found and its quality estimation.*
The algorithm:

1. *S.Initialize(C)*
2. ***while*** *(C ← S.GetNext()) ≠⊥* ***do***
   a. *create child with configuration C*
   b. *wait for the child*
   c. *series ← execute query Q*
   d. *series ← T(series)*
   e. *objective ← series[0] as double;*
   f. ***if*** *objective is the best of all seen so far* ***then***
      best ← C
   g. *S.RegisterObjective(C, objective)*
   h. *remove the child*
3. ***return*** *best*

The role of parameter-changes scenarios is to set some parameters to appropriate values in subsequent tests. Different scenarios may be used to perform miscellaneous types of search.

A parameter search scenario may just iterate over a set of possible values of a parameter or perform a sophisticated process exploiting specialized meta-knowledge and the feedback from subsequent tests. Item 2g of algorithm 4.6 is responsible for the feedback passed to the scenario object. The scenario of parameter changes can adapt its behavior according to the results it is informed about. Thanks to that, many intelligent scenarios can be created and used by MPS, resulting in quite sophisticated meta-learning algorithms.

Although it is not possible to define a single scenario, that would guarantee satisfactory results in a short time, machine authors may define default scenarios for machine parameters, that is, the default way of searching for optimal parameter values (suggestions of scenarios that should perform well on average).

After the search process is finished, the MPS machine can create the winner model and exhibit the outputs of the best machines as its own outputs, to provide the functionality of the best model to other machines. For example, a search for the best classification learner for given data, together with pointing the best solution found, can be treated as a machine learning classification itself. The same applies to any other type of models, one would like to search for: regression functions, clustering and so on.

Example experiment with MPS

To summarize many techniques described in this chapter, to put them together, illustrate their common functionality with a practical example, and show that so nontrivial projects can be created in quite readable form, an application of the MPS machine is presented as its original C# source code:

```
DataLoaderConfig dCfg = new DataLoaderConfig();
dCfg.InputFileName = "breast−cancer.dat";
Capsule dCaps = project.CreateMachine(dCfg, null, null);

RepeaterConfig rCfg = new RepeaterConfig();
rCfg.SetCVDistributor();
rCfg.TestScheme.Add(0, new StdConfig(),
    new Bindings().Bind("Data", "Training data"));
rCfg.TestScheme.Add(1, new SVMClassifierConfig(),
    new Bindings().Bind("Data", 0, "Data"));
rCfg.TestScheme.Add(2, new ExtTransformationConfig(),
    new Bindings().Bind("Data", "Test data")
        .Bind("Transformer", 0, "Transformer"));
rCfg.TestScheme.Add(3, new ClassTestConfig(),
    new Bindings().Bind("Data", 2, "Data")
        .Bind("Classifier", 1, "Classifier"));
rCfg.CVParams(5, 2);

ParamSearchConfig psCfg = new ParamSearchConfig();
psCfg.TestingConfiguration = rCfg;

int[] subcfgPath = new int[] { 1, 1 };
StepScenario_D SigmaScenario = new StepScenario_D(subcfgPath,
    new string[] { "Kernel", "Sigma" },
    StepScenario_D.StepTypes.Power2, −12, 2, 8);
StepScenario_D CScenario = new StepScenario_D(subcfgPath,
    new string[] { "C" },
    StepScenario_D.StepTypes.Power2, −1, 2, 7);
psCfg.Scenario = new StackedScenario(
    StackedScenario.StackTypes.Grid,
    new IScenario[] { SigmaScenario, CScenario });

psCfg.QueryDefinition = new QueryDefinition(
    new Intemi.Query.Qualifier.RootSubconfig(1, 1),
    new Intemi.Query.Labeler.FixedLabelList("Accuracy"),
    new Intemi.Query.BasicStatistics());

Capsule psCaps = project.CreateMachine(psCfg,
    new Bindings().Bind("Data", dCaps, "Data"), null);
```

```
40 project.WaitAll();
```

The lines 1–3 define a configuration of a data loading machine and request the machine. The lines 5–17 prepare a configuration of a repeater performing 5 times 2-fold cross-validation of an SVM machine trained on standardized training part of CV data and tested on the test part after standardization performed according to the statistics calculated for the training data. Thus, the machines built for *StdConfig* and *ExtTransformationConfig* are put inside the repeater test scheme. The repeater configuration is then (in the line 20) passed to the configuration of the MPS machine as the configuration of test procedure. It is very advantageous to analyze the code constructing the MPS configuration, together with Fig. 4.13, presenting the configuration graphically.

In the lines 22–31, the scenario of the parameters search process is defined. The search has a form of a grid (see the line 30), so that each declared value of the σ parameter of the Gaussian kernel function is tried against each declared value of the C parameter. Both parameters belong to the SVM machine which is identified by the path *1, 1* (see the lines 22, 23 and 30) of subconfigurations of the MPS test configuration (that is, the repeater configuration defined in the lines 5–17). The indices in the path are 0-based, so the repeater's subconfiguration 1 is its test scheme containing SVM configuration as its subconfiguration 1 (see the line 9). Both parameters are of exponential nature, so the explored sets of values are $\{2^i : i = -12, -10, \ldots, 4\}$ and $\{2^i : i = -1, 1, 3, \ldots, 11\}$ respectively. The sets specification is in the lines 25 and 28, containing the declaration of the exponential type, start exponent, step (the increment of the exponent) and how many values are to be tried. Many other predefined parameter search scenarios are also provided and completely new scenarios can be easily added.

The lines 33–36 specify the query to collect the test results and the transformation of the resulting series to another series containing the final estimate of the configura-

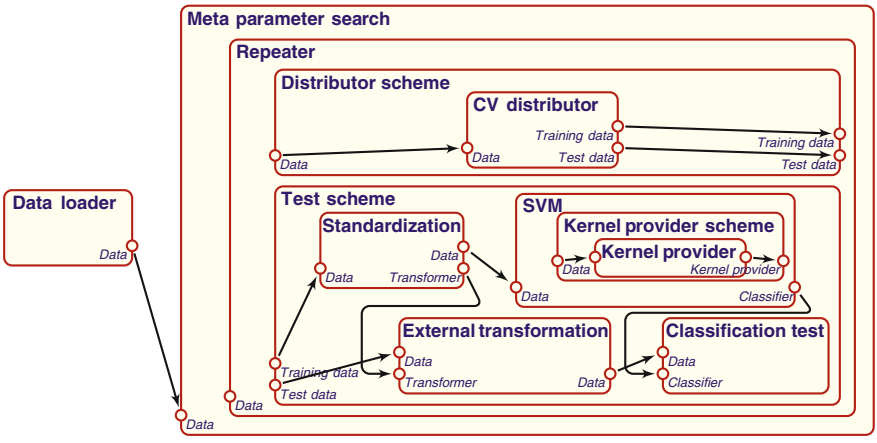


Fig. 4.13 Meta parameter search project configuration corresponding to the source code

Table 4.5 Meta parameters search results for SVM tested by 5×2-fold CV on Wisconsin breast cancer data

C	Gaussian kernel σ							
	0.0010	0.0039	0.02	0.06	0.25	1	4	16
0.5	0.6635	0.9393	0.9642	**0.9671**	**0.9674**	0.9588	0.9419	0.8798
2	0.9396	0.9645	**0.9677**	**0.9665**	**0.9659**	0.9605	0.9508	0.9159
8	0.9645	**0.9677**	**0.9668**	0.9645	0.9591	0.9479	0.9508	0.9159
32	**0.9677**	**0.9668**	0.9634	0.9474	0.9428	0.9451	0.9508	0.9159
128	**0.9674**	0.9617	0.9456	0.9413	0.9322	0.9422	0.9508	0.9159
512	0.9617	0.9474	0.9428	0.9365	0.9293	0.9416	0.9508	0.9159
2,048	0.9476	0.9416	0.9391	0.9339	0.9273	0.9416	0.9508	0.9159

tion performance. The estimate implemented here is the average accuracy obtained within the repeated CV test (transformation *BasicStatistics* calculates mean value, minimum, maximum, standard deviation and some other values, and sets the mean as the main label of the resulting series).

The results obtained with the code are presented in Table 4.5. They are available as an output of the parameter search machine after the whole process is finished.

Finally, the lines 38–40 request the MPS machine from the project and wait for all the machines to finalize their processes.

The system takes care for all the machines that must be built to provide the requested machine. It creates subsequent machines, when it can be determined that such a machine must be created, and uses the unification mechanisms to reuse machines created earlier, wherever possible. In this project, the repeater is performed many times with changed configuration of the SVM machine but without changes to the configuration of the CV processes. Thus, many submachines in this project, for example those responsible for data splits, are reused to save time and memory.

The project configuration is visualized in Fig. 4.13. At run time, a single Data loader and a single Meta parameter search are created. Within the MPS process, a Repeater machine is created for each of the $56 = 8 \times 7$ pairs of parameters. Each repeater performs 5×2 CV, so it creates 5 distributor schemes and 10 test schemes. Thus, the overall number of needed machines is quite large. Thanks to the unification framework (described in Sect. 4.2.2), the number of machines that are really created and run is significantly smaller.The numbers are presented in Table 4.6.

Table 4.6 is not extended with columns describing other data mining systems, because no other system is able to reuse machines, and even if one were 10 % faster in the first learning of a given machine, it would not be faster after next computation of the machine, since Intemi is able to reuse machines, so that serving repeated requests costs almost nothing. Learning processes are sometimes really CPU consuming and this problem can not be trivially neglected.

It can be easily seen in Table 4.6, which machines are reused many times: distributor scheme, CV distributor, standardization, external transformation, kernel provider scheme and kernel provider. Indeed, there is no point in repeating CV data distri-

Table 4.6 Numbers of machines that exist in the project logically and physically

Machine	logical count	physical count
Data loader	1	1
Meta parameter search	1	1
Repeater	56	56
Distributor scheme	280	5
CV distributor	280	5
Test scheme	560	560
Standardization	560	10
External transformation	560	10
SVM	560	560
Kernel provider scheme	560	80
Kernel provider	560	80
Classification test	560	560
Sum	4,538	1,928

bution for different repeater instances, since each time, the data split is the same. Thus, only 5 different CV data pairs are needed, as the repeater performs 5 independent splits. Similarly, the training and test data coming as test scheme inputs are cyclically repeated, so only 10 different standardization machines and 10 different external transformation machines are necessary.

A nice surprise can be the unification of the SVM kernel calculations. Intemi realization of the SVM machine extracted kernel calculations to a separate machine (submachine of SVM) to enable multiple use of the same kernel table. More precisely, as visible in Fig. 4.13, SVM defines the Kernel provider scheme, which can be filled with any machine structure providing kernel calculation interface. In the test being analyzed, the scheme contains a single Kernel provider machine, which outputs Gaussian kernel calculation routine. SVMs are run with different parameters of C and Gaussian kernel σ. Two SVM machines trained on the same data with different C parameter and the same σ may share the kernel table (that is, use the same Kernel provider scheme and Kernel provider). Thus, only 80 different kernel tables are requested in the project (8 different values of σ and 10 different training datasets).

As a result of all the savings, out of 4,538 machines comprising the project there are only 1,928 different machines. Naturally, it means completely different peak memory usage: more than 250 MB and less than 60 MB respectively. Moreover, less machines to be created and run means also time savings: the analysis described above, for Wisconsin breast cancer data, with machine unification takes just 10.5 s, while without unification it takes 16.5 s of a 2 GHz CPU. So different ratios describing savings in the number of machines, memory occupation and CPU time consumption result from the fact, that the machines that were unified in this project need much memory, but little time (the distributors that split data and data standardization

machines). Even the SVM kernel providers have not affected the result significantly, though their complexity is $O(n^2)$ in both space and time.

The unification possibilities are detected automatically. No machine is run twice with the same parameters, which saves both time and memory during the project run. It also saves the time of machine implementers, since they do not need to predict when exactly machines can be reused.

4.4.2 Meta Search Scenarios

The most powerful aspect of MPS (see algorithm 4.6) is the possibility of using arbitrary scenarios of parameter changes. From the algorithm 4.6, it can be easily inferred, that a scenario object must implement at least the three methods listed in interface 4.7. The life of a scenario starts with a call to the *Initialize()* method, which provides the test architecture, in which the scenario is to manipulate some parameters. Then, the *GetNext()* method is called as many times as necessary to get all possible configurations offered by the scenario. After each test is finished and objective value calculated, the scenario is informed about the achievement of its product with a call to the *RegisterObjective()* method. Thanks to that, the scenario can adapt its operation with respect to the results and move to other areas of the parameter space, if the suggested configurations perform poorly in the tests. Many scenarios completely ignore the information received by *RegisterObjective()*. It concerns both scenarios engaged in the MPS experiment discussed in the preceding section: *StepScenario* and *StackedScenario*. The former performs tests of several values of a continuous parameter, step by step, according to given conditions (start value, step, type of changes like linear or exponential, and so on). The latter combines other scenarios in a given way (one after another, combined into a grid and so on). For symbolic parameters, a scenario named *SetScenario* has been prepared, to test the elements of a specified set as the values of selected parameter.

Interface 4.7 (Scenario of parameter changes)

Method: Initialize(C)
Input: *Test configuration to be manipulated (C).*
Output: *None.*

Method: GetNext()
Input: *None.*
Output: *Test configuration with current parameter values.*

Method: RegisterObjective(C, O)
Input: *None.*
Output: *The configuration just tested (C), objective value derived from the test (0).*

Advanced scenarios (respecting the feedback) can be constructed in numerous ways. A modification of *StepScenario* named *StepScenarioWithZoom* facilitates testing continuous parameters in several stages with step changing in each stage. When several values are tested and a border one turns out to provide the best result, the step is increased and larger area is searched in the next stage. When the winner point lies inside the tested interval, the step is decreased to perform more detailed search in the area of the winner. Moreover, *StepScenarioWithZoom* can perform its search for several parameters in parallel, testing points on a line in multidimensional space, not necessarily along a selected axis.

Yet more advanced feedback analysis is performed by *MultiLineSearchScenario*, which uses *StepScenarioWithZoom* to search along different directions and tries to determine new better ones, on the basis of the results obtained with earlier attempts.

Many other search strategies can be constructed, so even with quite simple tool such as meta parameter search machine, but with sophisticated search scenarios, one can create very interesting algorithms, capable of learning from their earlier experience.

References

Frank A, Asuncion A (2010) UCI machine learning repository. http://archive.ics.uci.edu/ml

Grąbczewski K, Jankowski N (2008) Meta-learning with machine generators and complexity controlled exploration. In: Artificial intelligence and soft computing. Lecture notes in computer science. Springer, Berlin, pp 545–555

Jankowski N, Grąbczewski K (2008) Building meta-learning algorithms basing on search controlled by machine's complexity and machines generators. In: IEEE world congress on computational intelligence, IEEE Press, pp 3600–3607

Lowry R (1998–2013) Concepts and applications of inferential statistics. http://vassarstats.net/textbook/

Chapter 5
Meta-Level Analysis of Decision Tree Induction

Object oriented design divides complex algorithms and data structures into smaller and simpler components, specializing in solving extracted subproblems. As a result, also in the approach to a general framework for DT induction, the algorithms can be composed by a number of compatible components. In the framework described in Chap. 3, even the simplest DT induction algorithms are composed of several components responsible for such tasks as performing search process, estimating split quality, pruning and so on.

Functionality of each component can usually be implemented in at least several ways, so when all implementations are regarded of all components, the number of algorithms, that can be constructed within the framework, is very large. Hence, the number of comparative analysis scenarios is so large, that only a tiny sample can be presented in a book like this one. The sections of this chapter present and analyze just two comparative test scenarios, after a discussion on comparison techniques, that should not be used, because they introduce a bias into the comparisons, and the methods for fair, reliable tests. The analyses, presented below, can not go into all details that could be interesting for a reader, since deeper analysis of each test scenario, like the ones presented below, can be illustrated in hundreds of tables and figures, presenting various contexts of the analysis. The selected aspects, discussed below, are:

- an analysis of single tree induction methods, focused especially on tree validation (Sect. 5.3),
- a comparison of components for construction of DT committees (Sect. 5.4).

Such selection lets demonstrate a variety of important DT induction techniques. Because model comprehensibility is usually very important when DT models are preferred, large forests are not analyzed here (though they serve as a reference point for evaluation of small DT committees in Sect. 5.4).

K. Grąbczewski, *Meta-Learning in Decision Tree Induction*,
Studies in Computational Intelligence 498, DOI: 10.1007/978-3-319-00960-5_5,

5.1 Results Comparison Techniques

Reliable conclusions may be drawn only from carefully prepared experiments. Comparisons of algorithms must be performed in such conditions that guarantee the same environment for the compared procedures. Although these remarks sound like truisms, the practice of scientific publications in the field of data mining shows that it is very easy to overlook some important details. Practical algorithms, almost always, consist of many procedures, detailed solutions, specialized data structures, so it is often tempting to simplify analyses by ignoring some of the incorporated concepts, because they seem marginal. Unfortunately, such intuitions sometimes turn out completely wrong, because detailed solutions can cause significant differences in results.

An example of such consequences may be observed when the QUEST algorithm (of Sect. 2.2.5.2) learns multi-class problems by converting them to two-class problems. Two superclasses are generated with two-means clustering of the set of class centers. Such information seems quite precise, but it turns out that such detailed solutions as the way of handling categorical variables may be responsible for large differences between obtained results. Comparing classical Euclidean distance with a symbolic-data-aware version (using Hamming distance for symbols instead of absolute difference between naturals encoding subsequent values), one can observe the difference in classification accuracy of as much as 10 % (flag data, 50.67 vs. 60.48 % accuracy respectively).

Another example of hidden information that significantly changes the achievements of DT induction algorithm is the "trivial" stop criterion of C 4.5 (see Sect. 2.2.3), that is, the control of node size (if there is just a single data object to be split from the others or the whole node dataset contains less than 4 objects, it is not split). Such conditions seem to be not important, because so small nodes are expected to get deleted by further pruning processes, but for some datasets, they introduce quite large, significant differences in the results.

Many more examples can be given, to illustrate that every detailed setting of complex DT induction algorithm can be very important. Therefore, to draw reliable conclusions about influence of particular parameters on final results, one needs to conduct tests of different values of the parameter in accompany with exactly the same other components. Otherwise, one can never be sure about the real reasons of the result differences. Below, some important aspects of fair testing are discussed.

5.1.1 Bad Testing Practices

Data mining tools like Intemi described in Chap. 4 provide very precious mechanisms, supporting conscientious testing of learning algorithms. Uniform representation and management of simple and complex machines, clear distinction between the configuration and runtime lives of machines, results repositories, queries, and many other solutions are very helpful in performing unbiased comparative tests. Naturally,

improper use of the tools is always possible, so there is no guarantee that tests performed with them are fair. Nevertheless, available easy-to-use procedures performing some standard tests are usually very efficient mean to discourage scientists from creating their own testing scenarios from scratch, prone to miscellaneous small and larger errors. Vulnerabilities to testing embezzlements are far larger when no uniform environment is used and many separate applications are combined. Some frauds come in disguise and require deeper analysis to be revealed, so that many have been accepted in numerous scientific articles. The unified view of machines and results handling (collection and transformation), proposed by systems like Intemi, make the embezzlements easier to see and avoid. Of course, it is not possible to completely prevent the frauds with system level protection, because that would mean significant restrictions of the system.

Probably the most common kind of testing mistakes is using data transformation methods in an unfounded way. There are many techniques of data preparation for final adaptive model construction. Many researchers clearly split the whole data mining process into stages including separate *data preprocessing* and final learning or testing. It is dangerous, because it suggests that testing machine generalization may be performed on preprocessed data without unjust consequences. As a result, there are many publications describing such unfair approaches to data mining.

Supervised Preprocessing

The most common mistake is using supervised data transformation methods before testing generalization abilities of learning algorithms, instead of including the supervised transformations in a complex model to be tested.

The set of the most popular fraud techniques of supervised data preprocessing includes:

- supervised data discretization,
- supervised missing values substitutions,
- supervised feature extraction.

In the list of fraud techniques, the word "supervised" has been emphasized deliberately. Naturally, using unsupervised methods (for example, data standardization) as data preprocessing, before testing generalization is not a cheat, although would also be odd in practical applications. For instance, in medicine, the process of gathering data usually takes some time to enable training of adaptive machines. When the models are ready, they are used to classify new data collected for new patients. Standardization of the whole data, which is the union of training and test datasets, would require waiting until the whole data, we want to classify, is collected. After that, the machines would be trained and the decisions for all the new patients determined. Decision support is required at the time each patient is examined, not after years of data collection. This would not make sense either, to retrain the system each time a new patient comes. Instead, it is the most sensible and the most common, that a model created for the data available at some time is used for diagnosing new

patients without retraining. Sensible tests should simulate practical applications, so using strange techniques, because they provide better scores, is unjustifiable.

When testing learning machines, one should always treat all the data transformations performed before training as a part of the training process, and in tests like cross-validation, repeat the whole procedure from the very beginning, for each CV fold. However, assuming that the training data is representative of the whole population, unsupervised data preprocessing methods like different forms of normalizations give very similar results for the whole data and for the training part. Then, the differences are very unlikely to affect further validation results.

An example of a genuine fraud is filling missing values in a supervised manner within the data preprocessing stage. If we allow a data preprocessing method to be supervised, we could create a method to fill the missing values with the class label of the particular data object. It would introduce very precious information and significantly decrease the estimated error rate. In an extreme case, we could delete all the values of a feature and replace them by the class labels. Then, simple identity function would be a perfect classifier, but what would we need to do with new data to be classified? We should do the same we did for the training data, that is, put the class label in proper place, so we would need to now the class label in order to predict it. When the supervised data transformation is included in a complex learning machine, the problem of test data transformation (where no targets are available) gets clearly visible and makes such fraud models infeasible. Using data preprocessing before proper tests does not reveal the problem and accepts such erroneous approaches.

It is important to notice that filling missing values with the mean values observed within the corresponding class is practically equivalent to filling with the class labels, since almost always, the means are unique for particular classes, so they are equally informative as class labels.

Another example of influence of supervised data preprocessing on validation result may be adding new, more informative features, on the basis of an analysis of a model built for the whole dataset. Again, an extreme example of such unfair calculations would be adding the targets to the data. In such case, everybody would have no doubt that it would be a cheat. Unfortunately, less radical examples quite frequently go successfully through the sieve of peer review processes. To not cite erroneous publications, let's regard the following example.[1] SSV decision tree algorithm (Grąbczewski and Duch 2000) generates a very simple and relatively very accurate description of the appendicitis data from the UCI repository:

if HNEA $<$ 7520.5 $\wedge$ MBAP $<$ 12 then class 2 else class 1.

The rule accuracy calculated for the whole set is 91.5 %. Extending the original data by adding a binary feature corresponding to the value of the rule premises introduces the knowledge gained by SSV (by an analysis of the whole dataset) to the data. Therefore, most classifiers can find the information and reach the CV test accuracy of up to 91.5 %, never available in fair CV tests. Of course, the SSV decision

[1] The example has been prepared especially for this illustration. It has not been published in any article as an approach claiming to be right.

tree reaches the maximum accuracy when tested on the modified data, while when trained in a fair way, its average CV accuracy is about 86.1 %. So, the alleged CV test result (after adding the feature) is in fact the reclassification accuracy. It is a less spectacular and less obvious example than adding the target as one of the features, but reveals the same vulnerability.

Data Multiplication

Even unsupervised transformations, when used as *preprocessing*, may be very dangerous. For example, multiplication of instances (with optional addition of noise) has sometimes been treated as a "cure" for small number of instances or class representation imbalance. This lets obtain much better results with learning machines very sensitive to training data size or requiring balanced classes. When this transformation is made before separation of testing data part, it causes that the same instances can be found in the training and test data (when noise is added at multiplication, not the same, but very similar instances can be found). As a result, if large part of the original dataset is multiplied, a miracle is nearly guaranteed, especially with machines like 1NN, which finds original clone-instances as nearest neighbors.

When tests like cross-validation are performed after data multiplication, the probability that a test data item can be found also in the training data part gets quite large, and leads to highly overoptimistic estimates of model accuracy. For example, if the original data describes two-categories classification problem, and the relation of class frequencies is 80–20 %, the second class needs to be included in four copies to restore the balance. In 10-fold CV, the probability that an instance falls into a given test part is equal to 0.1, so the probability that all the remaining 3 copies of a second class instance fall into the same test part (so that none is in the corresponding training part) is just 0.001. Thus, on average, one test data instance of thousand, belonging to the second class, has not been represented in the training data. This completely distorts the test results.

Naturally, data multiplication applied after determination of training and test parts is completely fair and equivalent to using weights assigned to data instances.

Single Test Files

Another abuse in data mining is excessive exploitation of single external test data files, as numerous machine tests on the same data change testing to validation, giving completely different role to the test sample, which in fact, becomes a validation part of the training data.

Although in the case of different contests it seems the best way of estimation of learning-machines generalization abilities, it is not good to isolate a sample as the test data and distribute it with targets alongside the training data. An illustrative, extreme example is a "learner" that does not learn, but simply guesses the target model. It is just the matter of sufficiently large number of trials, to find a model performing

perfectly on the test set. But it would no longer be a fair result, because we would have selected a model on the basis of its behavior on the test data, that is, we would have used the test data for a peculiar type of training.

In the case of the hypothyroid data from the UCI repository, SSV decision trees (with some setting of the parameters) reach 99.73 % accuracy in reclassification of the training data and 99.09–99.36 % when classifying the "test data". The success of the most accurate configuration parameters can not be regarded as fair, since we would never guess that it is so accurate model, if we had not checked it on the test data. Similarly with a kNN machine equipped with mechanisms of feature weighting, one can obtain almost as good results as with SSV decision tree, but running a CV on the union of training and test sets results in definitely lower results, showing that the model with weighting is so accurate, as a result of a coincidence or using the "test data" for model validation. Training and testing machines many times makes finding an accurate model more and more probable. That's why Bonferroni corrections (see Appendix Sect. A.1.4) are used in estimating statistical significance of results differences, when many models are tried.

Adjusting Test Domain

The most difficult to eliminate are the errors made on purpose. One of the easiest ways of results manipulation is to perform many tests on various datasets and select some of them for presentation. Today's large computational possibilities are very advantageous, but they also facilitate obtaining attractive results of particular algorithms by the fraud of selecting a subset of tasks, providing the most favorable quality estimates. In combination with the pressure on the researchers all over the world, to publish numerous articles, this results in many publications of not very valuable algorithms. Such practices impede the valuable research, but no simple solution to the problem exists.

A method to avoid such biased results is blind selection of datasets before starting the experiments. Although it does not guarantee that the set of learning tasks is representative of the whole space of real problems, it does not introduce evident bias. Random selection of the test data and (if feasible) performing tests on as large number of datasets as possible may be very valuable and very important from the point of view of fair results comparison.

5.1.2 Reliable and Just Comparisons

From the comments on bad testing practices, it is easy to infer the correct way of testing learning machines and comparing their results. The most important rule is that testing should always concern the whole complex machine, if the machine consists of a number of components. Series of transformations from raw data, through

all necessary data transformations, to the final decision making module should be treated as a single machine.

Results obtained on external test data samples can be really treated as test results, estimating model accuracy on unseen data, only if no feedback from tests on the data is taken into account before the final model is created. This rule is fundamental in organizing fair contests in data mining. The targets of the test data sample should be kept secret till the end of the contest and competitors should not be given any feedback about the accuracy obtained by their models on the test data. Otherwise, it is possible to learn from the feedback, and the test set stops being a test set. In fact, on the basis of multiple model submission and overall accuracy results it is possible to recover all the outputs expected for the test data items.

When testing different versions of a machine, for own purposes, the authors should also be very careful in defining the tests to avoid different kinds of bias. Extracting a pair of training and test data samples for multiple experiments does not bring reliable estimation of machine capabilities, because accidental correlation of machine results with the true outputs is more and more probable with increasing number of tested machines. Therefore, more trustworthy results are output by cross validation. When comparing results, not only raw accuracy estimates should be taken into account, but also standard deviations of the results.

The best way to compare the results of two methods is to use statistical significance tests, and judge whether the differences can be accidental. The tests estimate the probability that the two samples come from the same distribution—small value of the probability (p-value) is a basis to reject the hypothesis (about the same distribution) and to claim that the differences are not accidental, but statistically significant.

If only possible, it is advantageous to collect the results of two machines to be compared, as pairs of results obtained with the two algorithms after training on the same data sample and test on identical samples. This facilitates using statistical tests specialized in comparing paired samples, which is much more informative, than for example comparing just means and standard deviations. The most popular tests for paired samples are *paired t-test* (a parametric test, see Appendix Sect. A.1.2). and *Wilcoxon test* (nonparametric, see Appendix Sect. A.1.2). Their counterparts for samples not organized in pairs are *t-test* (parametric) and *Mann-Whitney test* (nonparametric).

For faster results comparisons, some simpler ways of results comparison, may also be useful (especially in meta-learning). Quite good measures for the purposes of algorithm selection are simple combinations of results means and their standard deviations (like the difference between the two).

Researchers testing many variants of their methods should always remember that multiple comparisons increase the probability of type I error of statistical tests (rejecting true hypothesis about insignificance of result differences). To avoid such bias in comparisons, one should use techniques like Bonferroni corrections, described in Appendix Sect. A.1.4.

The community of machine learning researchers, still lacks reliable data repositories supporting as reliable comparisons of different methods as possible. Such repositories should specify the datasets for experiments and testing methodologies,

facilitating exactly the same training and test data in each iteration of multi-part tests like cross-validation. Repositories of results obtained with a variety of methods for each training data of each test can be very helpful in thorough comparative analysis of new methods on the background of other algorithms. Such databases will certainly become available in a close future and will prevent publication of fake results. They will also serve as very precious sources of meta-knowledge for advanced meta-learning approaches and will determine the strength of automated learning environments.

Some practice of trustful and successful results comparison can be observed by examples of data mining contests, where methods of testing, parameter search and combining transformations must be joined in the common effort—the pursuit of the most successful models (Grąbczewski and Jankowski 2006; Jankowski and Grąbczewski 2007).

5.2 Test Scenarios for DT Induction Analyses

Many meta-level experiments, presented in this book, were performed for the same selection of datasets. At some time of development of the unified model of decision trees (presented in Chap. 3), 21 datasets from the UCI repository (Frank and Asuncion 2010), were selected as the basis of comparative tests, and then, they have also served for some meta-learning experiments. The datasets define classification problems as summarized in Table 5.1. Such collection of datasets has several important advantages, for example:

- The datasets are fully satisfactory for the sake of simple and more advanced tests. They represent a variety of domains, so they test algorithms in different applications.
- They have different characteristics, so they allow to observe different winners and analyze which methods perform better in which conditions.
- They are very popular and available to anyone, so other authors may easily compare their results with the ones presented here.

Some datasets (of the repository) were not selected, because they would need some preprocessing (for example, to delete classes with very few examples) and that would spoil the clarity of the tests. Some other reasons eliminated other datasets, for example, the popular mushroom data was rejected, because it is too easy for DT learning—many configurations of DT algorithms are 100% accurate with zero variance, so the dataset would add no value to the comparisons. Moreover, in some experiments, where the differences between results are rescaled to the units of standard deviations of the best result observed, the mushroom data would be problematic, because nonzero differences would be converted to infinities and would spoil the analysis.

In the experiments based on the 21 datasets, for each of the datasets, 10-fold cross-validation was repeated 10 times with different data splits, so that 100 accuracies (and

Table 5.1 Datasets used for the experiments

Symbol	Dataset	Classes	Instances	Features		
				Total	Ordered	Unord.
APP	Appendicitis	2	106	7	7	–
AUS	Australian credit	2	690	14	6	8
BRE	Breast cancer (Wisconsin)	2	699	9	9	–
FLA	Flag	8	194	28	10	18
GLA	Glass	6	214	9	9	–
HEA	Heart	2	303	13	13	–
IMA	Image	7	2310	19	19	–
ION	Ionosphere (trn+tst)	2	351	34	34	–
IRI	Iris	3	150	4	4	–
KVK	kr-vs-kp	2	3196	36	–	36
LBR	Ljubjlana breast cancer	2	286	9	1	8
LET	Letter recognition	26	20000	16	16	–
PIM	Pima indians diabetes	2	768	8	8	–
SON	Sonar	2	208	60	60	–
SOY	Soybean large	19	307	35	–	35
SPL	Splice	3	3190	60	–	60
THY	Thyroid (trn+tst)	3	7200	21	6	15
VOT	Vote	2	435	16	–	16
VOW	Vowel	6	871	3	3	–
WAV	Waveform	3	5000	21	21	–
WIN	Wine	3	178	13	13	–

100 other adequate results, for example, tree sizes for DT induction methods) were collected for each of the algorithms being compared. Naturally, each of the algorithms observed the same inputs as the others in all 100 training and test stages. Therefore, after collecting 100 test accuracies (possibly accompanied by other results) for each tested algorithm, the differences could be and have been thoroughly analyzed with statistical tests for paired samples (usually paired t-test, but also Wilcoxon test), because they are very robust in detecting difference significance.

5.3 Single Decision Tree Models

Even in so seemingly narrow area as the family of single DT induction methods, it is possible to perform plenty of comparative tests, by various selection of the methods components. Provided the Intemi DT induction framework, they all can be easily defined and executed. The test presented here is focused on DT pruning methods. Nevertheless, we can not ignore many other algorithmic components which, together with pruning techniques, compose fully-functional learning algorithms. Reliable estimation of pruning advantages is possible only with tests on various kinds of trees, so

different approaches to DT induction have been engaged. Each algorithm has been tested with all possible values of several factors of concern, to learn more about methods parameters and their cooperation. For the sake of conclusions about the most accurate pruning techniques, a broad range of pruning methods of different kinds has been selected.

In compliance with the remarks on fair tests and comparisons of data mining techniques, presented above, all the methods have been compared by training and testing on exactly the same data samples, and collected results have been analyzed with statistical significance tests. Also, when learning processes consist of initial learning and validation, each method has been trained and validated on exactly the same samples. Similarly, multi-pass validation methods have been run on exactly the same validation trees and data, and finally, all the methods have been applied to same tree model.

5.3.1 Algorithms

A selection of the most popular pruning methods representing various approaches, has been done arbitrarily, but most researchers should agree that the collection includes the most interesting techniques. The selected methods can be organized in three groups with respect to their topmost strategies:

- direct pruning methods
  - Pessimistic Error Pruning (PEP),
  - Error-Based Pruning (EBP),
  - Minimum Error Pruning (MEP),
  - Minimum Error Pruning 2 (MEP2),
  - Minimum Description Length Pruning (MDLP),
  - Depth Impurity Pruning (DIP),
- single-pass validation methods
  - Reduced Error Pruning (REP),
  - all multi-pass validation methods listed below, run with single training and validation,
- multi-pass validation methods
  - cost-complexity minimization,
  - degree-based tree validation,
  - OPT,
  - MEP2 with CV-based m parameter estimation,
  - DI with CV-based β parameter estimation.

Raw list of algorithm names does not give complete information about the algorithms details. As emphasized in Sect. 5.1, reliable conclusion can be drawn only

from fair comparison of the methods working in the same environment (with the same list of components composing the fully functional learning machine). Therefore, the list of algorithms to be compared is much longer than the list of pruning methods given above, as each pruning method has been applied in combination with some other parameters.

First of all, to test the performance of pruning methods in application to trees generated with different approaches, each of them has been run in accompany with each of five DT induction methods based on the following split criteria:

- accuracy maximization criterion,
- Gini index,
- information gain (IG) criterion,
- QUEST algorithm,
- Separability of Split Value (SSV) criterion.

Each split criterion has been used with the greedy search algorithm.

Other parameters are not common for all pruning methods, so they need some more explanation. First four of the direct pruning methods and all single-pass validation methods have been run with three values of SE factor (0, .5SE and 1SE). All multi-pass validation methods were run with five values of the SE parameter (0, .5SE, 1SE, .5SEs, 1SEs), because additionally, they facilitate estimation of standard deviations on the basis of results samples. The SE parameter does not apply to MDLP and DIP methods of the direct-pruning group.

Similarly, the training error factor has been applied with three values (0TE, .5TE and 1TE), wherever adequate, that is, to all (both single and multiple) validation based methods.

5.3.2 Experiment

The comparison, discussed here, has been founded on running 10-fold CV repeated 10 times (as announced before) for each combination of the parameters, according to the recipes given above. As a result, the test has been run for $4 * 3 + 2 = 14$ direct pruning methods, $(1 + 5) * 3 * 3 = 54$ single validation methods and $5 * 5 * 3 = 75$ multiple validation methods. These give the total of $14 + 54 + 75 = 143$ pruning methods. A report on the numbers of configurations corresponding to various parameters is presented in Table 5.2.

Many result tables shown below (starting with Table 5.3 on p. 194), contain 145 result cells, because two full DT models (with no pruning) are also included in comparisons: one for the whole training data and one for the part of the training data used for initial learning of single-pass validation methods. The cells corresponding to unfeasible parameter combinations (incompatible parameters), are left blank.

Since each pruning method has been applied in combination with each of the 5 decision tree induction methods, we finally get $5 * 145 = 725$ complete DT algorithms for each examined dataset.

Table 5.2 Summary of configuration counts corresponding to the parameters

Type	Count	SE	TE	Total
Direct (1)	4	3		12
Direct (2)	2			2
Single val.	1 + 5	3	3	54
Multiple val.	5	5	3	75

The same numbers refer to each of the 5 methods of DT construction

Table 5.3 The counts of datasets, for which particular methods obtained results not significantly worse than the best one observed within the tested methods

pruning							single							multiple						
full DT	PEP	EBP	MEP	MEP2	MDLP	DIP	full DT	REP	CC	Degree	OPT	MEP2	DI	CC	Degree	OPT	MEP2	DI		
8	10	9	8	9	5	7	1	5	3	4	7	6	3	12	11	14	7	11	0TE	
								6	4	4	5	6	3	10	8	8	10	9	.5TE	0SE
								6	6	3	7	6	3	10	8	8	9	8	1TE	
	6	10	7	8				4	3	3	3	5	2	8	5	11	5	9	0TE	
								5	6	4	8	6	3	11	7	9	10	9	.5TE	.5SE
								7	5	4	6	4	3	9	9	8	8	8	1TE	
	8	8	5	10				2	2	1	2	4	2	3	2	6	3	7	0TE	
								7	6	4	5	6	2	10	7	13	7	10	.5TE	1SE
								4	7	4	7	8	3	9	7	8	6	9	1TE	
														8	5	11	4	9	0TE	
														10	7	8	7	9	.5TE	.5SEs
														9	8	8	10	8	1TE	
														4	3	6	3	8	0TE	
														9	8	14	7	11	.5TE	1SEs
														8	8	8	7	9	1TE	

All the results come from application of different pruning methods to the trees built with the QUEST algorithm

For deeper analysis, the tests have also included validation algorithms applied to test data (in place of validation data). Such models may be blamed for cheating, because they use targets of the test data. Therefore, they do not appear as competitors in the presentations. They have been run to show the maximum accuracies that could theoretically be obtained with subsequent approaches, and serve as reference, ideal pruning methods defining maximum possible scores, although in practice usually not available. They have been tested to facilitate some special analyses, for example, to compare tree sizes to the optimal ones.

5.3.3 General Results Visualization and Analysis

The results collected for the 21 datasets reported in Table 5.1 are rich enough to facilitate many interesting conclusions. The divergence among the datasets makes

different algorithms win different applications. Focusing on proper selection (or projection) of results can reveal significant properties of the algorithms and confirm or reject the hypotheses about differences between the results obtained with alternative algorithm components.

Classification Accuracy

One of the most important aspects of the models being analyzed is classification accuracy. Because the error level and its variance are completely different among the datasets (for example, the error is less than 0.5 % and its standard deviation less than 0.25 % in the case of the well known thyroid data and close to 38 ± 10 % for the flag data), it does not make much sense to compare the nominal results averaged over the datasets. Therefore, it is more interesting to know the numbers of datasets for which the results obtained by the method at hand are not statistically significantly worse than the best results observed in the tests. In all the comparisons presented below, statistical significance refers to the results of paired t-test with significance level $\alpha = 0.01$. From this point, the methods that obtained results of the largest average accuracy or insignificantly worse than the best ones, are shortly referred to as *winners* of particular tasks, and unless otherwise stated, *the number of wins* means the number of datasets for which the method turned out to be one of the winners.

An example visualization of the numbers of wins is presented in Table 5.3, where the counts are organized in a tabular form to clearly relate the scores with particular values of the parameters. Columns correspond to pruning methods and rows to different configurations of their parameters. The results, shown in the table, refer to the trees built with the QUEST algorithm. Not all cells of the table are filled, because not all configurations of the parameters are feasible.

The first column of the table shows the number of datasets for which full decision trees (with no pruning at all) scored a win. Just the first row is filled, because no pruning parameters apply to full DT (the cell corresponding to 0SE and 0TE is the most natural for this configuration).

Next six columns correspond to direct pruning methods (embraced above the table with "pruning" label). Among them, four can be used with standard error (SE) parameter, but with no sample based estimations (only a single result is available) and no training error factor, because it applies only to validation based approaches. MDL and DI based pruning methods do not deal with error directly, so the SE parameter is not applicable either.

Next, a group of 7 columns is denoted as "single". These are the pruning methods performed with single validation pass. It is important to realize that they prune different trees than the other methods, because for their purpose, a validation sample must be extracted from each training dataset, and only the remaining data subset is used for training. Because of that, there is another column labeled "full DT", which presents the number of wins obtained with full DTs built for the data subsets. The remaining six columns of the group correspond to methods, to which both SE and

training error factors apply, but because of single validation pass, no sample based estimation of SE is possible.

Only the methods performing multiple validation contain full sets of results, because all values of SE and TE factors are compatible with such methods. The last five columns, labeled "multiple", present the results obtained for these methods.

The labels placed on the right of the table, describe the values of TE factor and SE factor, respectively. Rows and columns corresponding to different groups are separated with some lines to make reading and comparing the scores easier.

Visual analysis of Table 5.3 can bring many interesting conclusions from various points of view. For example:

- The largest number of wins has been obtained by the OPT method with two settings of standard error and training error factors:
  - both SE and TE factors equal to 0,
  - SE factor equal to 1 and estimated from sample (1SEs) and including training error with the factor of 0.5.
- When 0SE is used, increasing the training error factor does not (in general) improve the results.
- When SE factor of 0.5 or 1 is used, the training error factor of 0.5 seems the most adequate in most cases.
- Increasing the SE factor, with no training error influence (the TE factor equal to 0) does not improve the results.
- Although Breiman et al. (1984) presented the 1SE rule as an advantageous tool, the experiment, similarly to other publications (Esposito et al. 1997; Lim et al. 2000), has shown that the trees pruned in this way are less accurate.

It must be notified, that the conclusions derived from analysis of Table 5.3 should not be regarded as global rules concerning DT pruning in general. The table summarizes the results of pruning trees generated by the Intemi implementation of the QUEST algorithm, so the conclusions are derived in such context. Nevertheless, all the phenomena but the first one (being a detailed observation) can also be observed (more or less evidently) for the other DT induction algorithms.

The conclusions remain valid also for the overall results (summed for the 5 DT construction algorithms) presented in Table 5.4. The absolute winner (of the first conclusion) differs between the algorithms. The highest score overall[2] has been obtained by the OPT method with 0SE and 0TE. The runner-up is the EBP algorithm with 1SE parameter, and the second runner-up is OPT with 1SE and 0.5TE.

Pure counts of winning results are not fully informative. For example, very high score is obtained by the EBP method, but can it really be regarded as so attractive pruning algorithm? Table 5.5 presents average scores obtained for each of the algorithms, by calculating the accuracy difference between particular method and the best one and dividing the difference by the standard deviation of the best method results

[2] Since the counts in Table 5.4 are summed over 5 algorithms, the maximum possible value is $5 * 21 = 105$. The highest score of 56 means wins in more than half of the tests.

Table 5.4 Win counts summed over all five DT construction algorithms tested

pruning							single							multiple						
full DT	PEP	EBP	MEP	MEP2	MDLP	DIP	full DT	REP	CC	Degree	OPT	MEP2	DI	CC	Degree	OPT	MEP2	DI		
35	36	40	30	39	27	25	11	24	28	30	38	25	24	51	51	56	37	50	0TE	
								27	23	24	27	22	20	32	34	38	38	43	.5TE	0SE
								23	20	16	24	20	16	36	37	33	37	39	1TE	
	35	49	35	36				27	28	29	30	25	23	36	32	42	28	44	0TE	
								29	31	20	36	29	25	38	32	38	41	48	.5TE	.5SE
								31	25	20	31	24	21	37	35	41	38	41	1TE	
	37	54	37	46				19	20	19	17	27	20	24	24	25	19	37	0TE	
								36	33	25	32	29	23	46	38	53	33	50	.5TE	1SE
								36	29	20	32	33	23	39	31	39	32	44	1TE	
														35	34	41	26	42	0TE	
														39	31	42	37	47	.5TE	.5SEs
														36	32	37	37	41	1TE	
														27	28	26	20	38	0TE	
														36	31	46	39	47	.5TE	1SEs
														35	32	39	35	44	1TE	

Table 5.5 Mean differences between different methods and the best possible pruning of the full DT (pruned by REP with the test data as the validation sample) measured in the units of standard deviations of the best possible pruning

full DT	PEP	EBP	MEP	MEP2	MDLP	DIP	full DT	REP	CC	Degree	OPT	MEP2	DI	CC	Degree	OPT	MEP2	DI		
-1.73	-1.51	-1.42	-1.64	-1.48	-1.63	-3.54	-2.02	-1.59	-1.50	-1.55	-1.50	-1.56	-1.61	-1.24	-1.28	-1.24	-1.29	-1.34	0TE	
								-1.60	-1.66	-1.77	-1.63	-1.68	-1.77	-1.43	-1.57	-1.40	-1.38	-1.51	.5TE	0SE
								-1.68	-1.84	-1.92	-1.80	-1.83	-1.92	-1.59	-1.68	-1.56	-1.55	-1.68	1TE	
	-1.47	-1.40	-1.61	-1.46				-1.60	-1.56	-1.60	-1.56	-1.56	-1.63	-1.29	-1.36	-1.32	-1.34	-1.37	0TE	
								-1.48	-1.53	-1.66	-1.51	-1.57	-1.67	-1.31	-1.48	-1.30	-1.32	-1.44	.5TE	.5SE
								-1.53	-1.68	-1.83	-1.63	-1.70	-1.80	-1.44	-1.63	-1.43	-1.46	-1.61	1TE	
	-1.47	-1.58	-1.76	-1.57				-1.76	-1.69	-1.73	-1.71	-1.61	-1.71	-1.39	-1.46	-1.42	-1.42	-1.43	0TE	
								-1.51	-1.51	-1.62	-1.53	-1.54	-1.66	-1.27	-1.42	-1.28	-1.34	-1.41	.5TE	1SE
								-1.47	-1.60	-1.76	-1.57	-1.62	-1.73	-1.38	-1.58	-1.38	-1.41	-1.52	1TE	
														-1.31	-1.37	-1.34	-1.36	-1.39	0TE	
														-1.34	-1.51	-1.33	-1.34	-1.45	.5TE	.5SEs
														-1.49	-1.66	-1.47	-1.50	-1.64	1TE	
														-1.42	-1.49	-1.45	-1.46	-1.46	0TE	
														-1.32	-1.48	-1.33	-1.34	-1.43	.5TE	1SEs
														-1.44	-1.63	-1.44	-1.45	-1.59	1TE	

(the difference with respect to the best, expressed in the units of standard deviations). The table sheds another light on the comparison: the larger SE factor for EBP, the larger average loss. The notice seems contradictory with the win counts, however a closer check confirms that it is possible, because when 1SE rule overprunes the tree, the loss gets large, and such losses significantly influence the mean result. It clearly shows the need for accurate meta-learning approaches capable of recognizing when the resulting trees are very good and when can be very poor, to avoid large costs of misclassification.

Table 5.6 Win-draw-loss counts of the competitions with full DT, summed over all 21 datasets and over all 5 DT algorithms tested

full DT	PEP	EBP	MEP	MEP2	MDLP	DIP	full DT	REP	CC	Degree	OPT	MEP2	DI	CC	Degree	OPT	MEP2	DI	
0-105-0	52-45-8	57-34-14	47-42-16	56-36-13	45-38-22	5-89-11	0-68-37	38-40-27	39-40-26	36-42-27	40-39-26	34-47-24	31-45-29	51-44-10	45-49-11	52-39-14	51-46-8	41-59-5	0TE
								31-52-22	22-60-23	17-60-28	30-53-22	21-59-25	19-56-30	47-55-3	23-81-1	50-51-4	51-51-3	35-70-0	.5TE
								21-60-24	9-69-27	7-66-32	11-69-25	10-68-27	8-65-32	35-67-3	13-92-0	38-64-3	38-65-2	15-90-0	1TE
	54-37-14	53-32-20	49-30-26	52-34-19				39-38-28	34-45-26	33-42-30	38-39-28	35-46-24	30-45-30	46-43-16	40-44-21	47-40-18	46-39-20	38-59-8	0TE
								39-43-23	33-48-24	28-53-24	39-43-23	33-50-22	27-50-28	52-46-7	39-59-7	54-39-12	52-42-11	43-59-3	.5TE
								39-44-22	29-52-24	10-65-30	32-50-23	23-57-25	17-59-29	48-50-7	20-82-3	53-43-9	52-46-7	32-70-3	1TE
	48-38-19	47-28-30	46-23-36	48-23-34				28-39-38	29-40-36	27-37-41	30-41-34	33-46-26	28-40-37	40-41-24	35-46-24	39-41-25	39-42-24	34-54-17	0TE
								39-42-24	37-44-24	28-51-26	36-44-25	35-48-22	29-46-30	52-41-12	47-42-16	54-36-15	48-40-17	44-55-6	.5TE
								38-44-23	32-48-25	19-58-28	37-44-24	32-51-22	25-51-29	55-43-7	32-68-5	57-34-14	53-37-15	39-60-6	1TE
														45-43-17	40-44-21	47-37-21	45-39-21	38-57-10	0TE
														54-41-10	34-64-7	54-38-13	53-42-10	42-60-3	.5TE
														50-48-7	15-88-2	49-45-11	45-53-7	29-75-1	1TE
														42-38-25	35-45-25	40-40-25	39-39-27	35-52-18	0TE
														52-39-14	45-47-13	57-31-17	47-41-17	45-52-8	.5TE
														52-43-10	19-80-6	55-38-12	53-42-10	34-68-3	1TE

0SE .5SE 1SE .5SEs 1SEs

Another example of deceptive appearances is the impression that the score of 35, obtained by full DT, and many results below the value mean that many pruning methods provide, on average, worse models than not pruning at all. Table 5.5 does not confirm such suspicion. Although full DT reached 35 wins, its average loss is −1.73, which is significantly worse than, for example, the results of −1.42 and −1.46 corresponding to the smallest numbers of wins recorded for multi-pass validation methods (the scores of 19 and 20 registered for MEP2 in Table 5.4). Another way to reject the conclusion about no-pruning superiority, is to analyze Table 5.6 presenting the *win-draw-loss counts* of the comparisons of each method against no pruning. Here, a win means statistically significantly higher accuracy, loss–inversely, and draw–no significant differences observed. Full DTs often record higher variance of the results, which does not allow t-test to reject the null hypotheses about no significant differences to the highest score method. As a result, they can be regarded as winners, while other results, although with higher accuracy, may record small variance and be judged as significantly lower than the best scores. This reveals an important danger of drawing misleading conclusions from properly performed statistical tests.

Table 5.6 also facilitates an observation about the influence of training error factor on pruned DT accuracy and stability. In the part of the table devoted to multi-pass validation, it is quite common, that with increasing factor for training data error, the number of losses against full DT decreases and the number of draws increases. In most cases, the number of wins grows with the TE factor of 0.5, but often decreases when it reaches 1. It confirms the intuitions, that respecting training data in validation can prevent overfitting the validation sample. The resulting trees get larger, but it often compensates the overpruning caused by increased SE parameter. In consequence, the trees have similar size to those obtained with 0SE, but are more reliable, because valuable splits are not pruned just because the validation sample does not contain an evidence for their value.

Gains offered by the training data error factor can be perfectly observed in Tables 5.7 and 5.8, presenting results obtained for sonar data. Table 5.7 shows mean accuracies (and their standard deviations) of pruned trees, while Table 5.8 contains

Table 5.7 Mean accuracies and standard deviations for IG DTs built for sonar data

full DT	PEP	EBP	MEP	MEP2	MDLP	DIP	full DT	REP	CC	Degree	OPT	MEP2	DI	CC	Degree	OPT	MEP2	DI	
74.13 ±9.093	74.13 ±9.103	74.13 ±9.103	74.13 ±9.093	74.13 ±9.093	74.27 ±9.021	74.13 ±9.093	74.47 ±7.930	73.59 ±8.305	73.98 ±8.323	73.88 ±9.021	73.21 ±8.690	73.32 ±7.869	73.45 ±8.545	74.45 ±8.978	73.80 ±8.687	74.08 ±8.461	74.46 ±8.848	74.55 ±9.101	0TE
								74.70 ±8.338	74.61 ±7.843	74.32 ±7.926	74.66 ±8.581	74.57 ±7.656	74.90 ±8.139	74.57 ±8.758	74.57 ±9.000	74.42 ±9.241	74.46 ±8.767	74.95 ±9.124	.5TE 0SE
								75.05 ±8.119	74.80 ±8.109	74.70 ±8.120	75.14 ±8.134	74.80 ±7.748	74.90 ±8.083	74.23 ±9.080	74.08 ±9.044	74.27 ±8.962	74.32 ±8.932	74.28 ±9.065	1TE
	74.13 ±9.103	75.04 ±8.724	74.42 ±8.926	74.70 ±8.751				73.40 ±8.235	73.55 ±8.224	73.50 ±8.490	73.26 ±8.702	73.18 ±8.074	73.70 ±8.619	73.25 ±8.836	74.10 ±8.449	73.93 ±8.751	73.09 ±8.457	73.54 ±9.460	0TE
								73.99 ±8.981	74.22 ±8.757	74.02 ±9.552	73.41 ±9.104	74.14 ±8.322	73.85 ±9.046	75.28 ±8.802	74.28 ±8.601	75.00 ±8.712	75.23 ±8.990	76.00 ±9.117	.5TE .5SE
								75.13 ±8.194	74.47 ±8.590	74.17 ±9.300	74.08 ±8.954	74.38 ±8.221	74.62 ±9.144	74.94 ±8.851	74.66 ±8.840	74.75 ±9.322	74.80 ±8.876	75.76 ±9.159	1TE
	74.70 ±8.651	76.29 ±8.824	75.62 ±8.956	75.57 ±9.178				72.88 ±8.592	72.77 ±8.655	73.15 ±8.682	72.59 ±8.315	72.30 ±7.944	72.93 ±8.090	72.97 ±9.037	73.05 ±9.239	73.39 ±9.085	72.24 ±8.422	72.47 ±9.056	0TE
								73.46 ±8.900	73.41 ±8.653	73.88 ±8.944	73.60 ±8.902	73.52 ±8.269	74.08 ±8.573	75.53 ±8.383	75.33 ±8.281	75.61 ±8.181	74.49 ±8.767	75.71 ±8.985	.5TE 1SE
								74.32 ±9.034	74.13 ±8.429	74.22 ±8.556	74.04 ±8.982	74.19 ±8.201	74.09 ±9.205	75.57 ±8.751	74.52 ±8.730	75.43 ±8.784	75.27 ±8.015	76.05 ±8.880	1TE
														73.49 ±8.851	74.25 ±8.614	74.02 ±9.449	73.34 ±8.457	73.68 ±9.536	0TE
														75.04 ±8.626	74.52 ±8.696	75.23 ±8.644	75.23 ±8.910	76.15 ±9.128	.5TE .5SEs
														74.66 ±9.163	74.61 ±8.766	74.37 ±9.194	74.75 ±8.861	75.62 ±9.060	1TE
														73.07 ±8.952	72.90 ±8.978	73.30 ±9.080	71.95 ±8.749	72.61 ±9.104	0TE
														75.76 ±8.462	74.71 ±8.385	75.57 ±8.501	74.98 ±9.120	75.66 ±9.017	.5TE 1SEs
														75.27 ±8.519	74.85 ±8.824	74.99 ±9.108	75.22 ±8.898	76.10 ±9.042	1TE

Table 5.8 Means and standard deviations of leaves counts in the IG trees built for sonar data

full DT	PEP	EBP	MEP	MEP2	MDLP	DIP	full DT	REP	CC	Degree	OPT	MEP2	DI	CC	Degree	OPT	MEP2	DI	
17.71 ±1.479	17.39 ±1.435	17.33 ±1.364	17.71 ±1.479	17.71 ±1.479	16.71 ±1.387	17.71 ±1.479	13.54 ±1.452	5.79 ±2.152	6.15 ±3.517	6.29 ±3.588	5.80 ±3.098	8.71 ±3.791	7.19 ±4.483	8.74 ±3.784	8.63 ±4.009	8.36 ±3.594	13.44 ±3.211	11.47 ±4.478	0TE
								10.95 ±2.086	11.28 ±2.995	11.12 ±3.092	10.94 ±2.954	12.45 ±2.017	12.21 ±2.324	15.01 ±3.410	15.46 ±3.233	14.53 ±2.819	16.01 ±2.181	15.26 ±2.464	.5TE 0SE
								11.93 ±1.748	12.31 ±2.481	12.39 ±2.369	12.12 ±2.266	12.93 ±1.671	12.65 ±1.951	17.09 ±2.156	17.13 ±1.878	16.52 ±1.685	16.88 ±1.945	16.77 ±2.059	1TE
	17.34 ±1.451	12.44 ±1.343	15.56 ±1.754	13.03 ±1.702				5.24 ±2.297	4.71 ±2.965	4.92 ±2.887	4.42 ±2.446	7.45 ±3.494	6.01 ±4.246	6.52 ±2.732	5.40 ±2.902	5.84 ±2.748	11.00 ±4.328	8.99 ±4.489	0TE
								8.32 ±2.335	8.33 ±2.948	8.21 ±2.910	7.58 ±2.559	10.57 ±2.645	9.67 ±2.896	10.33 ±2.941	10.89 ±3.025	9.68 ±1.885	13.86 ±2.179	12.35 ±1.731	.5TE .5SE
								10.19 ±1.768	9.73 ±2.597	9.61 ±2.609	8.85 ±2.105	11.66 ±1.897	10.81 ±2.219	13.12 ±3.016	13.37 ±2.250	11.98 ±1.676	14.49 ±2.096	13.22 ±1.829	1TE
	13.12 ±1.437	9.42 ±1.130	11.49 ±1.617	11.10 ±1.528				3.74 ±1.926	3.50 ±2.338	3.50 ±2.115	3.27 ±1.625	6.45 ±3.236	4.33 ±3.534	4.45 ±2.603	3.63 ±1.988	4.12 ±2.189	8.69 ±4.371	6.57 ±4.887	0TE
								6.59 ±2.919	5.93 ±2.326	5.90 ±2.397	5.48 ±2.037	9.04 ±3.035	7.97 ±3.080	8.13 ±1.637	7.83 ±1.706	8.05 ±1.459	12.77 ±2.546	11.53 ±2.032	.5TE 1SE
								8.56 ±2.166	7.71 ±2.483	7.58 ±2.270	6.95 ±1.956	10.62 ±2.295	9.73 ±2.155	9.76 ±1.634	10.93 ±2.567	9.45 ±1.095	13.64 ±2.153	11.97 ±1.828	1TE
														6.75 ±2.996	5.52 ±2.844	5.79 ±2.653	11.23 ±4.131	9.26 ±4.394	0TE
														11.19 ±3.277	11.74 ±2.994	10.60 ±2.207	14.15 ±2.262	12.87 ±1.983	.5TE .5SEs
														15.16 ±3.142	14.83 ±2.349	13.34 ±1.838	14.83 ±2.094	14.06 ±2.187	1TE
														4.62 ±2.688	3.75 ±2.081	4.55 ±2.315	8.50 ±4.359	6.90 ±4.886	0TE
														9.16 ±2.377	9.32 ±2.748	8.83 ±1.544	13.37 ±2.411	11.73 ±2.131	.5TE 1SEs
														12.11 ±3.055	13.01 ±2.541	11.17 ±1.621	14.27 ±2.004	12.81 ±1.857	1TE

means and deviations of the leaves counts in the corresponding trees. Increasing TE factor results in larger DTs, while greater SE factors decrease the average number of leaves in the trees. Relations between accuracies are not so monotonic, but in general, increasing the SE factor significantly deteriorates classification, though increasing TE factor often elevates the result above the original score (with both SE and TE set to 0). The highest accuracies (above 75.5) are in some cases obtained with not larger trees than those obtained with both parameters equal to 0.

Naturally, the phenomena are not observed always, but very often. For some datasets, full trees are so accurate, that pruning them may only spoil the result.

Table 5.9 Win counts in competition between direct pruning methods

	PEP			EBP			MEP			MEP2		
	0SE	.5SE	1SE	0SE	.5SE	1SE	0SE	.5SE	1SE	0SE	.5SE	1SE
Acc	8	7	9	11	11	7	6	8	6	6	10	8
Gini	9	8	8	8	9	11	5	7	10	8	7	10
IG	8	10	6	10	10	12	7	5	9	9	5	9
QUEST	11	8	8	11	11	9	8	7	6	11	8	11
SSV	8	8	6	9	8	14	7	8	5	9	7	8
Total	44	41	37	49	49	53	33	35	36	43	37	46

5.3.4 Analysis of Results Subgroups

Interesting conclusions can also be drawn from restricted analyses of similar subgroups of algorithms. By focusing on subgroups, one can pay more attention to the influence of particular parameters on the results and gain knowledge about the most appropriate ways of application of each method.

Direct Pruning Methods

To compare the performance of direct pruning methods and their parameters, numbers of wins have been counted with restriction to the four parameterized direct pruning methods: PEP, EBP, MEP and MEP2. The results are presented in Table 5.9. Each row of the table should be analyzed separately, because it contains a set of results from single comparison. For all five DT induction algorithms, some of the largest numbers of wins can be found within the results of EBP. It can not be claimed unambiguously, which values of the SE parameter should be used with particular pruning methods—in some cases 0SE is the most efficient, in others 1SE records the highest score. However it is clear that significant differences between the methods can be observed in vast majority of applications.

Esposito et al. (1997) have claimed, in their comparison of EBP and PEP, that EBP produces larger trees than PEP. The experiment presented here, revealed an opposite relation, which probably shows how details can affect conclusions, since there are small differences between the two implementations of the algorithms. For example, the Intemi implementation of EBP does not use child–parent grafting, performed by the C4.5 implementation. Nevertheless, grafting is performed relatively rarely, so despite the difference, in most cases, the results of the two versions of EBP should be the same. Another small difference between the implementation tested here and the one of C4.5 is that in C4.5, it works with some pre-pruning, as described in Sect. 2.4.2.2. These little differences in the implementation details, might be a reason of some performance differences, however there is much more probable reason—the original definition of PEP includes the 1SE rule by default, so probably, such implementation was tested by (Esposito et al. 1997), while EBP was not equipped

Table 5.10 Mean leaves count differences with respect to the optimum tree in the units of standard deviations

full DT	PEP	EBP	MEP	MEP2	MDLP	DIP	full DT	REP	CC	Degree	OPT	MEP2	DI	CC	Degree	OPT	MEP2	DI		
12.11	6.34	4.96	5.49	6.21	3.11	11.22	8.11	0.41	0.76	0.82	0.60	2.12	2.69	1.67	1.72	1.51	3.73	4.55	0TE	
								4.64	4.60	5.07	4.42	4.77	5.57	6.99	8.37	6.39	7.24	8.41	.5TE	0SE
								5.36	6.09	6.52	5.83	5.86	6.68	9.25	10.10	8.57	8.98	10.25	1TE	
	5.36	1.34	1.32	2.31				0.08	0.00	0.11	-0.21	1.46	2.16	0.53	0.55	0.33	2.63	3.77	0TE	
								1.63	2.26	3.19	1.72	3.14	4.28	4.15	6.10	3.41	5.31	7.10	.5TE	.5SE
								2.81	3.94	5.41	3.22	4.29	5.57	6.41	9.08	5.62	7.24	9.10	1TE	
	2.56	-0.28	-0.38	0.08				-0.67	-0.71	-0.58	-0.86	0.83	1.50	-0.17	-0.19	-0.34	1.88	3.07	0TE	
								0.56	0.84	1.50	0.45	2.14	3.29	2.52	3.90	1.90	4.06	6.08	.5TE	1SE
								1.42	2.44	3.88	1.75	3.15	4.57	4.93	7.58	4.05	5.85	7.90	1TE	
														0.50	0.53	0.32	2.63	3.76	0TE	
														4.64	6.62	3.86	5.67	7.35	.5TE	.5SEs
														7.18	9.49	6.36	7.81	9.48	1TE	
														-0.18	-0.19	-0.34	1.88	3.09	0TE	
														3.18	4.85	2.51	4.61	6.52	.5TE	1SEs
														5.88	8.53	5.04	6.80	8.71	1TE	

with similar correction, as it is not an integral part of the original definition of the algorithm. If we compare the results of such configurations (PEP with 1SE and EBP with 0SE) in the experiment described here, the conclusions are the same as those of (Esposito et al. 1997).

Table 5.10 contains the information about average tree sizes in the overall experiment. As before, to sum over different datasets, the numbers were transformed into proper differences expressed in the units of standard deviation. In this case, the differences are calculated with respect to the size of the trees pruned optimally, that is, to maximize the test data accuracy (as a result of running REP on the test data). Positive numbers mean that the trees were larger than the optimal ones. Very few numbers are negative (small negative), so one may conclude that no algorithm tends to significantly overprune the trees, however such claims should be formulated carefully, because the optimum size of the trees with respect to the test data is not the same as the optimum size for the whole population—we should rather expect that the optimal solution for test data overprunes instead of providing globally optimal pruning. Coming back to the comparison of PEP with 1SE and EBP with 0SE, the scores in the table are 2.56 and 4.96 respectively, so they are compatible with the findings of Esposito et al. (1997), but if we compare the pairs with the same SE parameter, in each of the three cases, we will find EBP to produce smaller trees (on average, of course) than PEP.

Single-Pass Validation Methods

A performance report for pruning methods based on single-pass validation is presented in Table 5.11. Each three rows should be analyzed separately, because they correspond to subsequent DT induction algorithms. Since the algorithms produce different trees, separating the groups is more reasonable for the analysis. Therefore, the pruning methods have been compared separately in the context of each DT induc-

Table 5.11 Win counts in competition between pruning methods based on single-pass validation

		REP			CC			Degree			OPT			MEP2			DI		
		0SE	.5SE	1SE	0SE	.5SE	1SE	0SE	.5SE	1SE	0SE	.5SE	1SE	0SE	.5SE	1SE	0SE	.5SE	1SE
Acc	0TE	12	13	8	17	14	9	14	10	9	17	14	9	11	12	10	11	11	10
	.5TE	12	12	12	11	15	17	7	8	9	11	16	14	11	12	13	7	8	9
	1TE	11	10	16	6	14	13	6	7	10	7	13	13	7	11	12	6	7	7
Gini	0TE	11	10	6	14	8	6	9	9	7	14	10	6	9	8	10	10	7	7
	.5TE	13	13	13	9	17	14	8	9	9	11	16	15	9	9	10	8	13	13
	1TE	13	15	16	8	11	10	8	9	7	8	13	13	8	8	8	8	11	12
IG	0TE	14	11	8	14	13	5	12	9	7	12	14	6	10	9	9	10	9	9
	.5TE	13	17	15	9	15	12	9	8	8	12	12	15	11	11	11	8	9	9
	1TE	13	14	18	9	11	10	7	9	7	12	10	12	8	9	10	8	8	9
QUEST	0TE	12	12	8	15	13	6	8	5	5	15	13	6	10	12	8	11	9	6
	.5TE	14	14	14	7	13	15	7	10	7	11	16	13	9	13	11	7	9	9
	1TE	13	13	16	8	11	12	7	7	8	8	13	15	8	9	12	7	7	9
SSV	0TE	15	13	7	16	11	7	13	9	6	17	14	7	12	11	9	9	9	6
	.5TE	14	15	15	10	15	13	9	9	9	13	15	15	9	12	13	9	11	9
	1TE	12	14	17	9	11	14	9	10	8	10	14	14	8	10	12	7	9	10
Total	0TE	64	59	37	76	59	33	56	42	34	75	65	34	52	52	46	51	45	38
	.5TE	66	71	69	46	75	71	40	44	42	58	75	72	49	57	58	39	50	49
	1TE	62	66	83	40	58	59	37	42	40	45	63	67	39	47	54	36	42	47

tion method, which means that for each node split technique, the winner was selected independently from the others and the scores show wins within the group, not among all the methods. The last three rows, present summed scores for the five groups.

The three subsequent rows of each group, correspond to the three values of TE parameter (0, 0.5 and 1). With rows corresponding to TE and columns to SE, it is easier to analyze the interaction between the parameters, so this is why they have been arranged in this way. In fact the three rows contain the scores of 54 competing configurations of single-pass validation methods.

It can be easily inferred from the darkness of the table cells, that for each node splitting technique, the highest scores can be found in the columns of REP, CC and OPT.

By focusing on squares of 3×3 cells corresponding to particular pruning and node split methods, one can easily observe the effects of interaction between SE and TE. Advantages of the parameters are visible especially in the REP squares. Increasing both parameters gives better overall results in all REP squares for the five split methods, and therefore also for the total summary square. It can be interpreted, that the training data error factor prevents overpruning by REP algorithm, as expected. It is not surprising either, that TE factor works best with 1SE parameter—1TE factor prevents overfitting validation data, while 1SE preserves generalization capabilities of the trees.

The effect observed so consistently in the case of REP is not so common in the other 3×3 boxes, but although most often the winning results are offered by 0SE and 0TE, increasing TE factor to 0.5 and SE to 0.5 or 1 can also be successful. Such knowledge may be very helpful in constructing model search algorithms at meta-level.

It can be claimed with very high confidence, that the two parameters (SE and TE) need each other very much. Increasing just one of the parameters, while keeping the other equal to 0, never provides the highest scores and usually decreases the numbers of wins. It can be seen as the brighter cells in the top right and bottom left parts of each 3×3 box.

All DT induction algorithms can be described with the same conclusions. The patterns of win counts are similar in all five cases, so it is possible to claim that the success of particular pruning technique is not much dependent on the method of tree construction.

On the basis of Table 5.11, it is not possible to draw any conclusions about relations between effects of using particular DT construction methods. Such claims are groundless, because the scores have been calculated for each method independently and ignore the relations between the node split techniques.

Multi-Pass Validation Methods

Table 5.12 is analogous to Table 5.11, but presents the scores obtained by multi-pass validation methods. Therefore, each box corresponding to particular pruning method and node splitting technique contains five instead of three columns corresponding to tested values of SE parameter (additional values of .5SEs and 1SEs using standard error estimated from samples of validation results).

Table 5.12 Win counts in competition between pruning methods based on multi-pass validation

		CC					Degree					OPT					MEP2					DI				
		0SE	.5SE	1SE	.5SEs	1SEs	0SE	.5SE	1SE	.5SEs	1SEs	0SE	.5SE	1SE	.5SEs	1SEs	0SE	.5SE	1SE	.5SEs	1SEs	0SE	.5SE	1SE	.5SEs	1SEs
Acc	0TE	15	9	7	8	8	9	9	6	9	6	13	10	9	10	9	11	9	7	8	8	11	9	8	10	8
	.5TE	6	11	15	10	10	7	7	7	6	6	8	13	14	12	12	8	10	10	11	10	7	7	8	5	6
	1TE	8	8	10	8	7	7	7	8	7	7	7	11	13	9	9	7	10	11	8	9	7	5	7	6	5
Gini	0TE	11	12	8	11	6	15	11	7	11	7	14	13	7	12	9	14	11	6	12	6	12	9	9	9	9
	.5TE	9	10	10	9	9	9	9	10	8	10	11	10	11	9	12	9	11	10	14	9	11	15	14	14	12
	1TE	8	11	10	9	9	10	11	9	9	10	9	10	10	8	10	9	12	8	10	9	10	11	11	10	11
IG	0TE	10	9	7	9	7	11	10	7	11	8	11	7	7	9	5	10	11	8	10	7	11	10	9	9	8
	.5TE	8	12	13	11	11	8	6	10	5	5	10	12	12	12	12	10	10	10	11	10	9	13	13	12	14
	1TE	10	11	10	8	10	9	8	5	8	7	9	9	12	9	10	10	11	10	9	10	9	11	10	11	11
QUEST	0TE	14	10	5	10	5	12	7	3	7	4	16	15	8	14	8	10	6	5	7	5	12	10	8	11	10
	.5TE	11	14	13	12	12	9	8	8	8	9	10	12	15	11	15	10	11	9	9	10	10	10	12	10	12
	1TE	11	11	11	9	10	9	10	8	9	9	9	9	10	9	9	10	9	9	11	10	9	9	9	9	11
SSV	0TE	11	9	7	10	7	12	8	6	8	7	14	10	6	12	7	6	6	5	6	5	11	9	7	8	7
	.5TE	6	10	10	6	11	7	9	8	9	8	10	10	15	9	11	8	7	8	8	7	7	11	9	9	8
	1TE	7	6	10	5	7	8	6	9	7	6	8	8	10	8	10	10	7	7	7	7	7	8	8	8	9
Total	0TE	61	49	34	48	33	59	45	29	46	32	68	55	37	57	38	51	43	31	43	31	57	47	41	47	42
	.5TE	40	57	61	48	53	40	39	43	36	38	49	57	67	53	62	45	49	47	53	46	44	56	56	50	52
	1TE	44	47	51	39	43	43	42	39	40	39	42	47	55	43	48	46	49	45	45	45	42	44	45	44	47

Similarly to the previous analysis, each three rows of the table need to be treated separately, as the winners were determined for proper subsets of algorithms.

It is not possible to definitely point a single overall winner within this group of algorithms. The largest number of wins (68) is recorded for OPT algorithm with parameters of 0SE and 0TE, followed by another settings of OPT (1SE and .5TE) reaching the total of 67 wins. When focused on particular node splitting methods, the OPT pruning is not always the best: for accuracy criterion (Acc) pruning with CC reaches better scores, for Gini criterion, pruning with Degree and DI scores more wins and for IG, CC and DI pruning offer more attractive numbers. Because the counts are similar for many configurations, they could be easily changed by adding or removing a single dataset, so the conclusions must be announced cautiously.

Unlike the previous analysis, here, more difference can be observed between .5TE and 1TE parameters. In corresponding pairs of results, it is rare that the 1TE score is better than the one for .5TE, but quite many opposite relations can be observed.

In competition between the methods of SE estimation, the two approaches provide so similar results, that no winner may be announced. Both methods are quite simple computationally, so neither should be definitely preferred, though the theoretical estimation is $O(1)$ and the sample based estimate is of linear complexity with respect to the sample size (usually very small).

Because of many unsolvable competitions between pruning parameters, there is even stronger need (than in the case of single validation methods) for meta-learning algorithms supporting decision making in the task of algorithm selection.

A comparison between multiple and single validation methods can be performed visually on the basis of Tables 5.4, 5.5 and 5.6, presented before. It can be captured with the naked eye, that multi-pass validation methods are more successful, but it should not be forgotten, that the single validation methods used smaller data samples, as a part had to be left unseen during training for further validation, so they were not given equal possibility of knowledge extraction. It can also be important, in some applications, that the multi-pass validation requires significantly more computational resources.

5.3.5 Summary

Especially when comparing many similar algorithms, capturing significant differences requires much attention to be paid when constructing the test. Only carefully prepared test scenarios may bring reliable results and adequate conclusions. Even, when tests are conducted in a fair way, and the results of tested methods are close to each other, it is easy to manipulate the final conclusions by adequate selection of datasets for the experiment.

The tests, presented above, have been performed without a dedicated dataset selection and cross-validation of all the methods has been performed for the same data samples, so the results are as credible as possible.

As discussed above, there are no common winners for all tested datasets. Some algorithms have performed with insignificant differences to the best solutions for most of the datasets, but none has been among the winners in all tests. In general, the methods of EBP and multi-pass validation techniques are the most successful, but it is not true, that they should always be selected as the best solutions. Usually, we do not face the problem of finding algorithms that provide best average results or are very good in most applications, but given a problem we need to find the best model for this particular case. Moreover, the success can be defined in many ways, so there is no single, commonly accepted method of best model selection. We can draw some interesting conclusions and extract helpful heuristics for algorithm selection, but eventually one should always compare many learning machines in application to the problem at hand.

The experiments, discussed above, have confirmed, that using training error factor and respecting standard error factor bring opposite results with respect to the tree size, but using them together can improve final trees by avoiding overfitting the validation data and preserving generalization capabilities at the same time.

Another expected result, confirmed by the experiments, is that in most cases, multiple validation produces more accurate trees than single validation. On the other side, single validation is computationally cheaper, so it may be valuable under circumstances of strong time restrictions.

Error Based Pruning is a very attractive method, because it offers similar accuracy as the best multiple validation methods, and very low complexity. EBP with 1SE is often very accurate, but sometimes provides large errors, so proper meta-learning is needed to detect whether for a given task, the method is valuable or not.

Heuristics supporting meta-learning, may also be based on elimination of machine configurations that are not too probable to bring attractive solutions. The analysis performed above lets assume, that there is not much sense in testing both alternative methods of standard error estimation, as most probably, it just costs twice the time, but without significant gains. Further time savings can be attained by avoiding nonzero values of just one of the SE and TE parameters.

Apart from the global meta-knowledge (about average gains of particular machine configurations), also local knowledge may be extracted by analyses focused on subsets of possible configurations. The rules extracted in this way may significantly improve meta-level search processes, discussed in Chap. 6.

5.4 Cross-Validation Committees

Model comprehensibility is definitely one of the most appreciated features of decision tree induction methods. Since it usually comes in accompany with fast learning processes, DTs belong to the most popular classification learning schemes. If apart from the two attractive aspects, one observes also the third one, in the form of very good generalization possibilities (high classification accuracy also for unseen data), the solution can be regarded as perfect. Often, DT models are preferred over other

classifiers acting as "black boxes", even when they provide slightly worse classification results. This is because their additional advantages recompense for the shortages.

Unfortunately, it is not an easy task, to find a single DT capable of good generalization of the knowledge encoded in the learning data. Instead of chasing after the impossible, one can use a number of trees together and in this way, recompense for the non-optimality of single trees. When the number of trees is not large, the ensemble model can still be quite comprehensible. Ensembles based on numerous DT models do not provide easy, human-understandable explanations, but have also attracted many researchers and practitioners (Breiman 1996; Gehrke et al. 2000). Such solutions often provide much better classification accuracy than single decision trees, but apart from loosing model comprehensibility, they require much more computational resources to build the models, which may be difficult to accept.

The experiment examined here, tests CV-based committees of decision trees (DTCV committees). Such ensemble classifiers also offer significant improvement in classification accuracy while not resigning from the comprehensibility of the model and low computational cost. When classification is made on the basis of small numbers of trees, several alternative classification rules may be presented to the expert (often some of them are almost identical) providing even more comprehensible decision support.

During the processes of DT validation based on CV, a tree is generated and validated in each fold of the CV. DTCV committees arose from the natural suspicion that these nicely validated DTs should provide a successful collective model. It is important to notice that such collective models do not require any additional calculations in comparison to single tree models validated with the CV. Inversely, building an ensemble model of the CV trees is computationally cheaper, because the final tree trained on the whole data need not be created.

The idea of DTCV committees and their advantages have been initially presented in (Grąbczewski 2011). Here, some of the algorithmic components underlying the DTCV committees have been reformulated and some others have been introduced to increase the flexibility of the algorithm and to examine it in a more exhaustive manner. Interesting meta-knowledge has been extracted from the experiments and is discussed below.

Extensive analysis has been performed within Intemi, described in Chap. 4, with respect of all the rules of fair comparison of learning machines, described in Sect. 5.1.

5.4.1 DTCV Committee Algorithm

There exist many possibilities of creating committees from validated trees. The differences between solutions may concern methods of tree growing, pruning, model selection, combining decisions of ensemble members and other components. Each of the aspects may introduce several parameters influencing the final complex model.

The analysis described below, concerns 13390 different configurations of cross-validation committees and their performance on 21 datasets estimated with 10 inde-

pendent runs of 10-fold cross-validation on each dataset. The test has yielded a number of interesting conclusions about how to build successful DT committees. The result is very precious for human experts building such models manually and for automated meta-learning approaches.

Decision Tree Construction

DVCV committees can be constructed from models created by any DT induction methods. For the purpose of this experiment, to examine advantages of DTCV committees in a broad spectrum of detailed solutions, the committees have been constructed in the context of four algorithms for DT induction following the most popular and successful algorithms like CART (Breiman et al. 1984), C4.5 (Quinlan 1993) or QUEST (Loh and Shih 1997), and the SSV approach (Grąbczewski and Duch 1999, 2000; Grąbczewski 2011a). Details about the algorithms can be found in Chap. 2.

The ideas of CART, C4.5 and SSV are implemented in Intemi as split criteria: Gini index, information gain (IG) and the separability of split value (SSV) criterion, used with a greedy, almost exhaustive search for decision trees with univariate binary splits. The term "almost exhaustive" is explained in Sect. 2.2.7. It denotes a restricted search to prevent much time consumption and uneven contest between features describing learning data. The subject of bias in split feature selection is addressed in Sect. 2.7.

IG criterion is not exactly the solution of C4.5, where information gain ratio is used instead (see Sect. 2.2.3), but the ratio has been introduced to reduce the bias in favor of symbolic features, inherent in the multi-split technique of C4.5. When using the almost exhaustive search for binary splits, the advantages of the IGR disappear. Other experiments have shown, that with this type of search, IG performs better than IGR (Grąbczewski 2011a).

Pruning Strategies

All the algorithms for growing DTs, used in the experiment, build oversized trees in a top-down manner and then prune them for better generalization. For the purpose of DTCV committees, it is also preferable that the member models are pruned and generalize nicely, because the quality of committee members is very important when the members count is not as large as in most applications of bagging, boosting or other types of forests.

The methods of pruning do not need to be especially designed or optimized, to prepare members of DTCV committees. Therefore, the experiment has exploited the most often used pruning techniques based on cross-validation (CV), for example, the cross-complexity pruning of CART, degree based pruning of SSV and dynamic programming based OPT algorithm (Bohanec and Bratko 1994; Almuallim 1996). Also other parameterized methods, like MEP2 or DI, can be embedded in similar CV-based procedure of parameter selection (see Sect. 3.2.4.3). They have also been included in the tests.

Computationally simpler method of Reduced Error Pruning (REP) (Quinlan 1987), has also been engaged as eligible method of another kind than the CV-based algorithms.

No pre-pruning method has been included in the tests, because their applications are, in general, less successful, so despite their small computational requirements they are not as common as post-pruning. Moreover, in CV-based learning, pruning methods based on validation are more adequate then pre-pruning techniques, because they naturally fit the CV scheme and have additional advantage of using more data for learning.

Committee Size

The most obvious method to obtain different committees is to engage different numbers of members, as there is no obligation to use all the models generated during the CV process. It may happen that a training data sample is drawn so unluckily, that the resulting model can not generalize well and can spoil the decisions of the committee.

A way to select members is to order the candidate trees by their validation results and add them one by one to the set of members, creating a series of committees. Such heuristic lets us reduce the number of ensembles to be tested, from $2^N - 1$ to N, where N is the number of validated models. Moreover, it facilitates knowledge transfer between different learning approaches, because the terms like "best five trees" preserve their functionality when applied to another collection of models, while terms like "trees from CV folds: 2, 3, 5, 6, 8" do not (though technically feasible). It also conforms to the intuitions, that the trees performing well in the validation process are better candidates for the ensembles than those of poor results.

Common or Separate Validation Results

Pruning decision trees, in the validation process, can also be done in a number of ways. For example, the pruning methods based on CV, can be applied in two variants: with common optimum parameter determined for all the validated trees (globally, on average) or with independent optimization for each CV fold.

In the following descriptions and result tables these two methods are distinguished by the terms *common* and *separate* validation (or by some shorthand notations of the two words).

Standard Error for Better Validation

Growing trees for the purposes of DTCV committees is subject to the same ways of validation and pruning as in the case of single DTs. A generalization of the standard error (SE) parameter introduced by Breiman et al. (1984) to control the "strength" of pruning is described in Sect. 3.2.4.1.

In the experiment, the values of 0, 0.5 and 1 for the SE parameter have been tested. Apart from the theoretical estimation of standard error proposed by Breiman et al. (1984), the assessment from the sample of validation results has also been examined (where adequate). In this way, additional methods denoted as .5SEs and 1SEs have been introduced.

Training Error Included in Validation

Similarly to the SE factors, the training error factors, described in Sect. 3.2.4.2, have been incorporated into the validation processes, to provide additional control over tree pruning, with special respect to deterioration of training data reclassification accuracy. The TE factor has been examined with values of 0, 0.5 and 1 (also referred to with TE suffix).

Committee Decision Making

Classifiers committees usually make the final decision by counting votes of the members and determining the winner class in the democratic voting (each vote has the same strength). When the member classifiers are capable of providing probabilities of belonging to different classes, instead of simple voting, average probabilities supplied by the members may decide about the committee decision. As shown by Grąbczewski (2011), such decisions are much more reliable then voting.

Another idea of Grąbczewski (2011) was to use weights provided by the committee members in place of probabilities. The weights were the numbers of vectors of different classes falling into the same tree leaf as the examined data item. When summed for all the trees (committee members), they were used to determine the winner class. The suspicion was that it could perform better than the probabilities-based collective decisions, but the idea turned out to be fruitless, because often, when a data object is classified by the trees, it falls into large leaves when it is misclassified, and into small ones when it is classified correctly, so that one erroneous answer happens to dominate (and spoil) the decisions of several correct models. Because of this result, the experiments described below used only the method with probabilities in the committees, however the probabilities were calculated in three different ways, as described in Sect. 2.6.

5.4.2 Experiment

With different values of the parameters of DT induction, validation and committee construction, described above, many configurations of DTCV committees may be created. In the experiments discussed below, the committees were created with the following parameters:

- split strategy (DT induction algorithm): Gini index, IG, SSV, QUEST,
- tree validation method: REP, cost-complexity optimization, degree-based tree validation, OPT, MEP2 and DI,
- validation parameter optimization: separate, common,
- standard error consideration: 0SE, .5SE, 1SE, .5SEs, 1SEs,
- training data error factor: 0TE, .5TE, 1TE,
- committee size (members count): 1,…,10.

Not each combination of the parameters is consistent. For example, REP can not be applied with common parameter optimization, because the error reduction is made for particular validation data, not for a parameter, that could be common for several trees. For the same reason, in REP and other methods using separate validation, there is no possibility of standard error estimation from sample, because no appropriate sample is available. Another exception is the MEP2 validation, which makes sense only with m-estimates of the probabilities at the stage of decision making, because the validation process optimizes the performance of m-estimates—using another technique of probabilities estimation would be completely inconsistent with the learning process. Because of the exceptions, there are less DTCV committee configurations for validation methods of REP and MEP2 than for the others and some of the tables presented below contain empty cells.

The parameterization, listed above, with the exceptions, gives the total of 13560 different committee configurations. The numbers of combinations for each validation method are presented in Table 5.13.

Standard (single tree) validated models were also built for each of the 4 split criteria (Gini, IG, QUEST, SSV), for each of the CV-based validation methods (CC, Deg, OPT, MEP2, DI) and for each value of standard error consideration (0SE, .5SE, 1SE, .5SEs, 1SEs), which makes the total of $4 * 5 * 5 = 100$ single-tree machine configurations.

To analyze all the parameter values, the corresponding committees have been tested and compared in a number of combinations (also against traditional models with single validated trees as the final classifiers). The comparisons have been done on the basis of 10 independent runs of 10-fold CV (100 accuracy results), by means of paired t-test with $\alpha = 0.01$. For the same 100 splits, each of the 100 standard single trees trained with CV have been built and each of 13560 committees created.

Table 5.13 The counts of DTCV committee configurations for all validation methods used

Validator	DT alg	Sep/com	SE factor	TE factor	Decision	Size	Total
REP	4	1	3	3	3	10	1080
CC	4	2	3/5	3	3	10	2880
Deg	4	2	3/5	3	3	10	2880
OPT	4	2	3/5	3	3	10	2880
MEP2	4	2	3/5	3	1	10	960
DI	4	2	3/5	3	3	10	2880
							13560

As in the other experiments presented in this book, the solutions of Intemi system (Grąbczewski and Jankowski 2011) facilitated easy comparisons, conducted in completely fair manner: even the internal CVs (within the training data) used exactly the same data splits for all the committees, and as a result—the same set of induced trees. Apart from the fairness, the experiments have an advantage of fast calculations, because the trees did not have to be induced repeatedly, thanks to Intemi mechanisms for machine unification which saves time and memory without much effort from the designer of the test projects.

5.4.3 Win Counts

The goal of the first analysis was to check each particular configuration, how many times (for the 21 datasets) it obtained results not statistically significantly worse than the results of the best machine (providing the largest average accuracy) of the 13660 configurations (100 single trees and 13560 committees). As before, the fact of obtaining the results not statistically significantly worse than the best ones is called a win, and each method satisfying this condition is called a winner of the task.

It is not easy to show 13660 numbers at once in a readable form, so a form of visualization has been prepared for this reason and the results are split into parts. The configurations have been organized in a way making finding results for particular configurations quite easy.

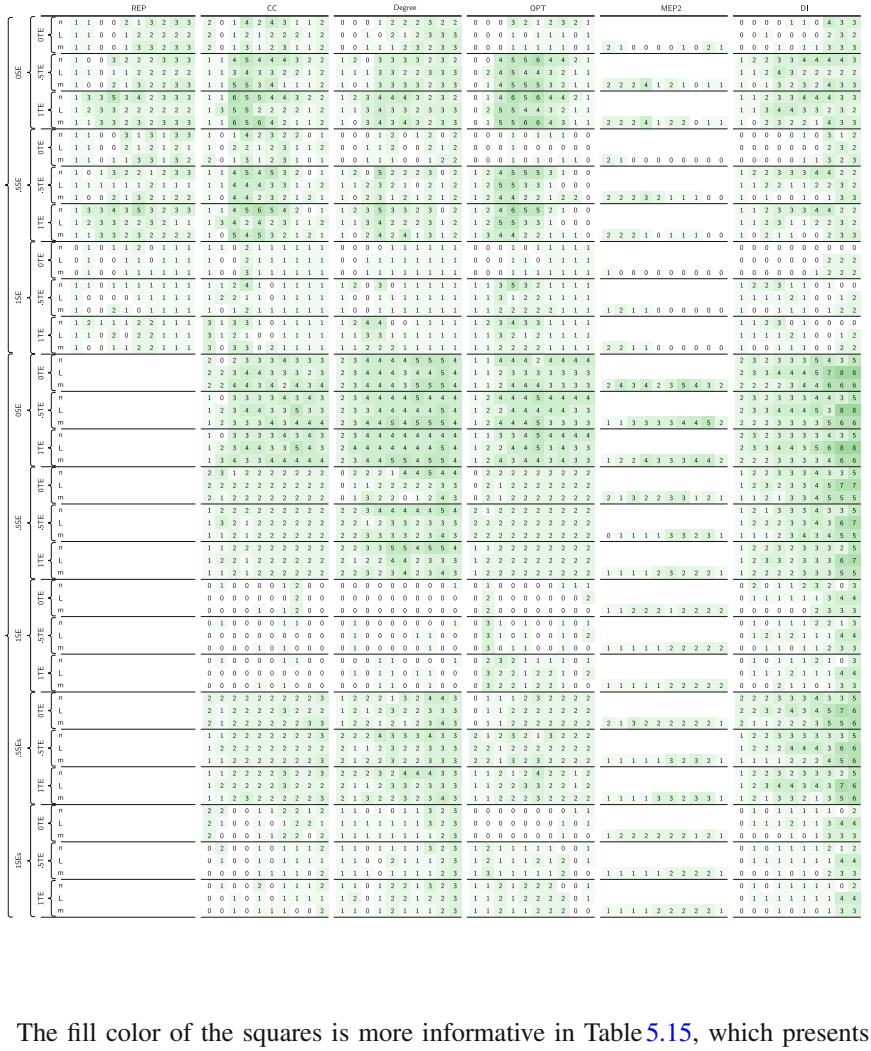
A subset of results is presented in Table 5.14. It contains 100 squares corresponding to the results of 100 single-tree configurations. Each square contains a number of datasets (of the 21) for which the configuration provided results not significantly worse than the best (in term of the mean accuracy) of the 13660 methods. The color, the square is filled with, corresponds to the number of wins—the greater the number the darker the square—to facilitate easy visual analysis. We can see that the maximum number in the table is 3 and most of the squares show 0 or 1, which means that single trees are very seldom comparable to the best results observed for the committees (quite expected result).

Table 5.14 Results for single decision tree models

	Gini					IG					QUEST					SSV				
	0SE	.5SE	1SE	.5SEs	1SEs	0SE	.5SE	1SE	.5SEs	1SEs	0SE	.5SE	1SE	.5SEs	1SEs	0SE	.5SE	1SE	.5SEs	1SEs
CC	1	1	0	1	0	2	1	0	1	0	0	0	0	0	0	2	1	0	1	1
Degree	1	1	0	1	0	3	0	0	1	0	2	0	0	0	0	2	2	0	2	0
OPT	1	1	0	1	0	2	0	0	0	0	2	2	1	2	0	1	2	0	1	1
MEP2	0	1	0	1	0	2	1	0	1	0	0	0	0	1	0	2	1	0	1	0
DI	1	1	1	1	1	2	1	0	1	0	1	1	0	1	0	0	0	0	0	0

Numbers of datasets (of 21) with results not significantly worse than the best ones

Table 5.15 Results for DTCV committees based on IG criterion. Counts of results not worse than the best



The fill color of the squares is more informative in Table 5.15, which presents 3390 results obtained for the committees of trees generated with the information gain criterion ($\frac{1}{4}$ of all the results for the committees).

The table has six "large" columns corresponding to different validation methods. As mentioned before, REP and MEP2 are not consistent with some other combinations of the parameters, so in the places of inconsistent configurations there are no numbers displayed. Each "large" column consists of 10 "small" columns corresponding to subsequent numbers of committee members (1, …, 10). The remaining parameter values are shown on the left. First, two large blocks corresponding to separate and common pruning are marked with proper long braces. Inside each of the two blocks, there are smaller blocks of results corresponding to committees with

the methods of SE consideration, denoted as 0SE, .5SE, 1SE, .5SEs and 1SEs (additional 's' at the end of the identifier means that the SE is estimated "from sample"). For separate validation, there are 3 smaller blocks (for first three options) and for common validation, all 5 settings are compatible, so 5 blocks are presented. Moving deeper, each of the SE-related blocks contains 9 rows of numbers (3 groups of 3 rows). The groups show the results for adequate factors of training data error (0TE, .5TE and 1TE), and each of them contains 3 rows with the results obtained with different approaches to probabilities estimation ('n' means no correction, just proportions, 'L'—Laplace correction, 'm'—the m-estimates).

Provided Table 5.15, we can draw interesting conclusions about the performance of various committee parameters, with the naked eye. For example, everyone can easily notice that the darkest areas can be found in the rows corresponding to the 0SE parameter. The results for .5SE are not so dark, and the rows of 1SE are much lighter.

Another interesting regularity can be easily observed when we focus on the columns corresponding to committee members count:

- combining just one or two models does not offer attractive results,
- the darkest areas in the top part (corresponding to separate pruning) occur in columns from 3 to 6, pointing these values as the most adequate committee sizes,
- in the lower part (of common pruning) the cells are the darker, the closer to the maximum committee size.

Such distribution of the more successful configurations is understandable: separate pruning results in more diverse trees—some are very accurate, some others quite poor, so adding the latter to the committee just spoils the results. When common pruning parameter is determined, it is validated in such a way that all the trees on average act successfully, and it is reflected also in the committees.

It is also worth a notice, that the most successful configurations reach the level of 8 of 21 datasets, where they do not get significantly worse results than the best ones. This means that no single configuration can solve even a half of the 21 problems in a satisfactory way. Although not all the results are presented here, those in Table 5.15 belong to the most successful of the four alternatives, so the conclusions are valid for the overall results. This fact reveals the need for meta-learning approaches capable of recognizing which methods are most likely to provide successful results for the data at hand.

Complete result tables, illustrating many aspects of the analyses of the DTCV committees, can be found in an auxiliary document about this experiment available at http://www.is.umk.pl/~kg/papers/12-DTCVCommRes.pdf.

5.4.4 DTCV Committees Versus Single Validated Trees

One of the most important premises for using DTCV committees can be their advantage over single decision tree models. Therefore, a comparative analysis of classifiers

in the form of committees and single validated trees has also been conducted. Naturally, the comparison may be made from different points of view: classification accuracy, model comprehensibility, model complexity and so on.

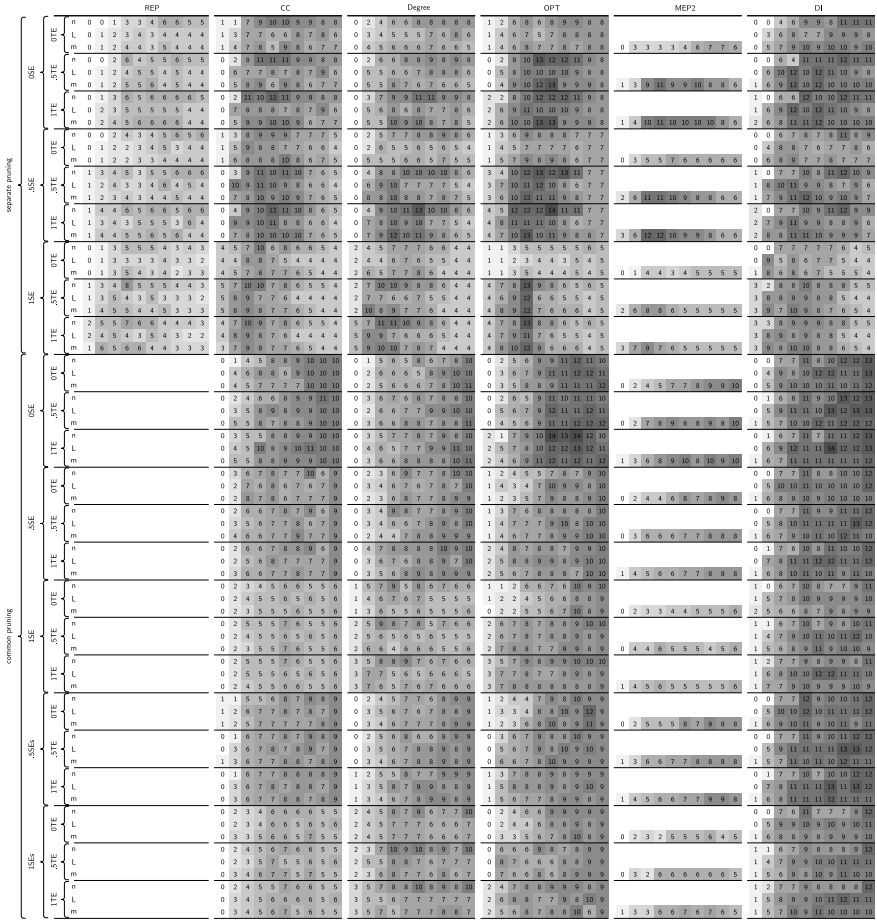
To observe the gains or losses in accuracy, we can compare the results of each particular committee to the results obtained with a single DT model corresponding to the same parameters. The correspondence must respect the DT induction method, the validation method and the option of the standard error factor. Among the 100 single DT models tested aside the committees, one can be naturally assigned to each committee, but those using REP. The natural assignment means that the committee is constructed from exactly the same trees which are created in the process of single DT validation, so the comparison between the two classifiers answers the question if it is better to build a single final model or to use the validated models in an ensemble. Because, for REP-based committees, we can not find a naturally corresponding single tree, they are compared with one of the classical approaches, namely the trees built on the basis of information gain criterion and cross-complexity validation. Again, the results have been prepared as the numbers of datasets for which the particular committee performed significantly better than the corresponding single tree and significantly worse than the single tree. Tables 5.16 and 5.17 visualize the results for committees built with the information gain criterion (the images for the other DT induction methods look very similar and can be found in http://www.is.umk.pl/~kg/papers/12-DTCVCommRes.pdf).

It is easy to see that most of the committees perform much better than single trees. Only the committees pruned with REP are often worse than their counterparts in single trees, but, as mentioned above, they are compared to CC-validated DTs, so the conclusions that can be drawn from these results concern the differences between REP-validated committees and single models validated with CC. It is also very important to realize, that the REP-validated trees are trained on smaller training data samples, because a part of each must be left aside for validation, so the comparison should not be a foundation of wide-ranging conclusions.

Our main interest in this comparison should be focused on the remaining five "large" columns, containing the scores of direct competitions between corresponding committees and single-tree models. In most cases the advantage of committees is overwhelming, up to the score of 14 significant wins (see Table 5.16). The losses can be observed mostly for "committees" made of 1 or 2 trees (see Table 5.17), and there are many areas of the table, where we can see just zeros meaning that for neither of 21 datasets the committees recorded significantly worse results than single tree classifiers.

The conclusion from this analysis is that the DTCV committees of many different settings of their parameters, provide significantly higher accuracy than single validated models. Therefore, instead of using single trees one should consider DT committees, which are very likely to obtain more attractive results.

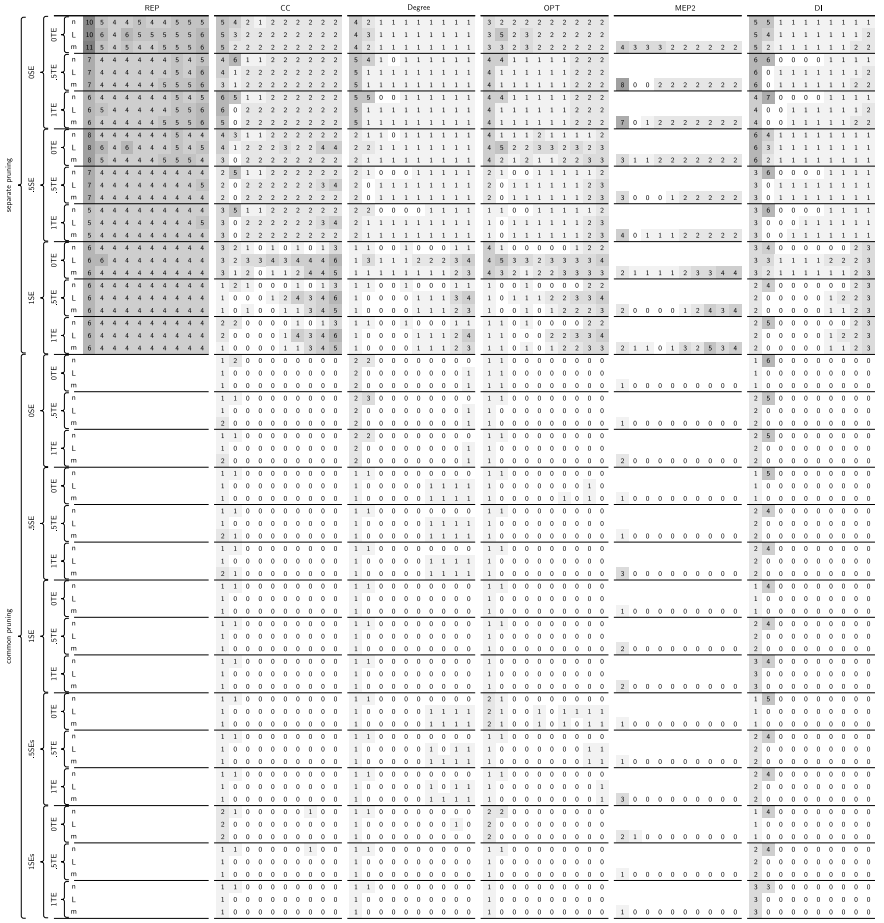
Table 5.16 IG DTCV committees better than single validated trees



Model Comprehensibility

Sometimes, a very important criterion in learning machine selection is model comprehensibility. In such cases, single DTs are very much appreciated, but DTCV committees also offer interesting possibilities. Since the number of trees in the committees is usually in the scope of 3–9, we can easily provide explanations of particular committee members as the rules supporting the decisions. An expert supplied with several alternative sets of rules, can find it even more useful than a single explanation in classical logic. The alternative rule sets can also point to another diagnosis than that of the committee. It may be a precious hint about other possible answers and about more complex nature of the problem being solved. On the other hand, many trees can provide very similar or even identical decision rules, which can be treated

Table 5.17 IG DTCV committees worse than single validated trees



as an additional confirmation of the rule and its stability. As a result, a bit more complex explanation than in the case of a single tree, may be very valuable, and may increase the comprehensibility, not reduce it.

5.4.5 DTCV Committees Versus Bagging and Boosting

As shown above, CV committees of decision trees are more accurate than single DT's. Another interesting aspect is if they have advantages over the state-of-the-art ensemble machines like bagging and boosting (Breiman 1996; Quinlan 1996; Freund and Schapire 1996; Dietterich 2000; Torres-Sospedra et al. 2007).

To check this, the CV committees have been compared to many different bagging and boosting algorithms. Wherever possible, bagged (or boosted) trees were constructed with the same parameters as those used for particular CV committee. It means that bagging and boosting algorithms were applied to four DT induction algorithms (Gini index Gini, IG, QUEST, SSV) to produce ensembles of size from 1 to 10. For each of the settings, seven ways of DT pruning not requiring any validation were followed: no pruning, PEP—Pessimistic Error Pruning (Quinlan 1987; Mingers 1989; Esposito et al. 1997), EBP—Error-Based Pruning (Quinlan 1987; Esposito et al. 1997), MEP and MEP2—Minimum Error Pruning (Niblett and Bratko 1986; Mingers 1989; Cestnik and Bratko 1991), MDLP—Minimum Description Length Pruning (Kononenko 1998) and DIP—Depth Impurity Pruning (Fournier and Crémilleux 2002).

For comparison with bagging-based ensembles, two ways of final decision making have been used: simple voting and making decisions on the basis of average probabilities of belonging to different classes, provided by the ensemble members.

Similarly, to make the comparison reliable, four boosting methods have been applied and compared: adaptive boosting, averaged boosting, conservative boosting and averaged conservative boosting. They differ in the ways they modify the probability distribution used for drawing subsequent data samples in subsequent steps. See Sect. 2.8.3 for more details on the algorithms.

All these conditions make 560 ($4 \times 10 \times 7 \times 2$) different configurations of bagging and 1120 ($4 \times 10 \times 7 \times 4$) configurations of boosting. Each configuration has been tested by 10 independent runs of 10-fold cross-validation (same as in the tests of CV committees).

The comparison between CV committees and bagged or boosted trees has been conducted by counting significant wins and losses between the ensembles of the same member count. Each CV committee can be naturally compared to two corresponding methods of bagging and four of boosting. All the comparisons have been thoroughly performed and their results put in tables available from http://www.is.umk.pl/~kg/papers/12-DTCVCommBagBoost.pdf. They all can not be presented here, because of the amount of tables. To support the final conclusions, Tables 5.18 and 5.19 present the results of the comparisons between DT CV committees of trees validated with cost-complexity approach to the ensembles obtained with bagging and boosting respectively. The results are summed for the two methods of bagging and four of boosting. Therefore, we get 294 competitions in each comparison with bagging (21 datasets, 7 DT validation algorithms, 2 decision making methods) and 588 with boosting (21 datasets, 7 DT validation algorithms, 4 boosting approaches). The numbers presented in each table cell are summed win-draw-loss counts.

CV committees with up to 3-4 members overwhelmingly outperform both bagging and boosting approaches. There are several potential reasons of this:

- validation on unseen data,
- selection of the best models as the first CV committee members,
- more representative training sample (90 % of the training data for each committee member versus approximately 63.2 % for bagging and the first iteration of boosting).

Table 5.18 IG DTCV committees built with CC validator compared to bagged DT ensembles

separate pruning

05E	0TE	n	204-90-0	212-67-15	83-195-16	96-186-12	47-167-80	50-184-60	32-156-106	29-158-107	24-108-162	26-112-156
		L	204-90-0	208-80-6	86-191-17	85-194-15	39-158-97	44-165-85	27-124-143	27-121-146	24-87-183	24-81-189
		m	204-90-0	212-76-6	97-179-18	97-184-13	39-160-95	49-161-84	30-131-133	26-133-135	25-88-181	23-95-176
	.5TE	n	200-94-0	194-88-12	105-173-16	107-176-11	54-187-53	49-193-52	32-172-90	29-169-96	25-109-160	26-112-156
		L	200-94-0	233-54-7	100-177-17	87-195-12	48-197-49	45-190-59	30-131-133	26-126-142	25-89-180	24-81-189
		m	204-90-0	224-63-7	107-167-20	99-183-12	51-177-66	51-191-52	32-140-122	28-138-128	25-90-179	23-95-176
	1TE	n	176-118-0	193-90-11	116-168-10	105-180-9	54-197-43	48-204-42	33-171-90	29-177-88	25-112-157	26-112-156
		L	176-118-0	241-46-7	105-178-11	88-196-10	43-209-42	46-194-54	32-146-116	26-125-143	25-88-181	24-81-189
		m	176-118-0	222-65-7	108-176-10	99-182-13	53-196-45	49-199-46	32-155-107	28-147-119	25-92-177	23-95-176
.5SE	0TE	n	200-94-0	212-67-15	74-204-16	94-180-20	45-162-87	42-171-81	29-149-116	25-155-114	22-97-175	18-106-170
		L	200-94-0	205-83-6	82-193-19	72-188-34	35-148-111	35-154-105	24-121-149	22-123-149	22-84-188	15-88-191
		m	200-94-0	208-80-6	85-191-18	88-184-22	40-141-113	36-163-95	23-134-137	20-132-142	22-92-180	13-96-185
	.5TE	n	208-86-0	205-77-12	100-178-16	105-177-12	51-175-68	49-182-63	30-156-108	25-156-113	21-99-174	18-106-170
		L	208-86-0	230-57-7	98-179-17	86-187-21	40-171-83	37-177-80	27-120-147	21-134-139	22-85-187	15-88-191
		m	210-84-0	220-67-7	107-165-22	96-186-12	43-159-92	41-171-82	27-131-136	22-141-131	22-92-180	13-96-185
	1TE	n	186-108-0	203-80-11	103-179-12	101-184-9	52-174-68	52-179-63	30-157-107	28-153-113	21-101-172	18-106-170
		L	186-108-0	232-55-7	103-180-11	91-184-19	42-175-77	38-168-88	27-121-146	21-134-139	22-85-187	15-88-191
		m	186-108-0	221-66-7	109-175-10	103-178-13	52-161-81	45-168-81	27-131-136	22-142-130	22-93-179	13-96-185
1SE	0TE	n	196-98-0	213-65-16	60-205-29	63-194-37	34-132-128	23-149-122	22-116-156	22-115-157	22-80-192	15-88-191
		L	196-98-0	189-98-7	47-194-53	48-161-85	30-114-150	23-121-150	24-100-170	22-107-165	21-68-205	13-77-204
		m	194-100-0	208-84-2	57-204-33	48-184-62	37-122-135	25-135-134	23-109-162	20-103-171	22-69-203	13-78-203
	.5TE	n	210-84-0	215-65-14	74-199-21	65-199-30	38-129-127	34-135-125	24-113-157	22-116-156	22-80-192	15-88-191
		L	210-84-0	215-72-7	53-203-38	49-181-64	30-118-146	30-118-146	23-100-171	22-108-164	21-68-205	13-77-204
		m	210-84-0	216-73-5	65-205-24	54-192-48	35-127-132	35-127-132	24-102-168	20-106-168	22-69-203	13-78-203
	1TE	n	214-76-4	215-67-12	74-208-12	66-196-32	40-131-123	33-138-123	24-113-157	22-116-156	22-80-192	15-88-191
		L	214-76-4	216-71-7	63-205-26	50-184-60	35-130-129	30-119-145	23-98-173	22-106-166	22-67-205	13-77-204
		m	214-76-4	210-81-3	66-207-21	57-193-44	39-125-130	34-128-132	24-103-167	20-106-168	22-69-203	13-78-203

common pruning

05E	0TE	n	224-68-2	208-80-6	99-190-5	96-191-7	45-180-69	47-199-48	40-164-90	37-181-76	34-133-127	37-126-131
		L	224-68-2	212-76-6	104-186-4	97-189-8	47-175-72	46-201-47	39-159-96	39-178-77	36-143-115	41-123-130
		m	224-68-2	214-71-9	100-168-26	111-171-12	46-166-82	47-192-55	39-150-105	38-166-90	36-134-124	37-128-129
	.5TE	n	224-70-0	216-71-7	105-182-7	97-189-8	46-179-69	47-204-43	40-165-89	36-179-79	34-136-124	37-126-131
		L	224-70-0	217-70-7	111-179-4	99-187-8	48-177-69	52-205-37	39-164-91	41-177-76	36-145-113	41-123-130
		m	224-70-0	217-68-9	110-162-22	111-166-17	46-169-79	50-200-44	39-151-104	37-169-88	37-137-120	37-128-129
	1TE	n	224-70-0	218-71-5	104-184-6	89-197-8	52-185-57	47-205-42	40-168-86	37-178-79	34-136-124	37-126-131
		L	224-70-0	221-67-6	111-179-4	93-200-1	47-188-59	52-211-31	39-162-93	41-177-76	37-144-113	41-123-130
		m	224-70-0	223-63-8	109-165-20	104-178-12	47-170-77	50-194-50	39-153-102	38-173-83	36-136-122	37-128-129
.5SE	0TE	n	212-82-0	205-83-6	82-181-31	76-185-33	42-152-100	50-166-78	37-139-118	38-142-114	33-90-171	36-89-169
		L	212-82-0	205-83-6	75-201-18	64-200-30	40-157-97	50-159-85	33-148-113	36-125-133	28-101-165	31-100-163
		m	214-80-0	205-83-6	81-183-30	71-189-34	43-154-97	49-160-85	36-141-117	33-130-131	26-105-163	29-103-162
	.5TE	n	208-84-2	205-82-7	85-164-45	77-185-32	43-153-98	50-149-95	37-120-137	38-138-118	32-93-169	36-89-169
		L	208-84-2	205-83-6	83-181-30	74-195-25	43-161-90	48-143-103	36-129-129	37-116-141	29-98-167	31-100-163
		m	208-84-2	204-84-6	83-169-42	81-184-29	44-157-93	47-153-94	37-132-125	34-135-125	26-105-163	29-103-162
	1TE	n	204-88-2	205-82-7	88-163-43	81-174-39	44-153-97	51-148-95	37-124-133	37-140-117	32-99-163	36-89-169
		L	204-88-2	206-82-6	84-181-29	77-191-26	48-156-90	49-145-100	36-143-115	37-114-143	29-100-165	31-100-163
		m	204-88-2	203-85-6	84-172-38	85-178-31	50-150-94	50-151-93	39-141-114	33-135-126	26-111-157	29-103-162
1SE	0TE	n	184-100-10	197-87-10	68-152-74	56-175-63	28-129-137	27-133-134	20-116-158	17-122-155	19-96-179	17-97-180
		L	184-100-10	195-89-10	66-161-67	53-175-66	27-133-134	22-132-140	20-108-166	18-125-151	19-91-184	18-98-178
		m	184-100-10	194-89-11	67-149-78	62-169-63	28-138-128	24-135-135	20-117-157	14-124-156	19-95-180	17-96-181
	.5TE	n	182-102-10	197-87-10	69-151-74	56-176-62	28-124-142	27-141-126	21-115-158	18-124-152	19-95-180	17-97-180
		L	182-102-10	196-88-10	66-164-64	54-178-62	26-128-140	24-138-132	21-106-167	18-122-154	19-91-184	18-98-178
		m	182-102-10	197-86-11	68-156-70	65-169-60	28-142-124	25-140-129	21-112-161	16-125-153	19-95-180	17-96-181
	1TE	n	184-100-10	197-87-10	67-155-72	58-175-61	28-127-139	27-137-130	22-114-158	14-126-154	19-95-180	17-97-180
		L	184-100-10	197-87-10	65-165-64	55-176-63	27-129-138	24-141-129	22-105-167	14-126-154	19-91-184	18-98-178
		m	184-100-10	197-87-10	68-156-70	65-168-61	29-141-124	25-141-128	22-112-160	16-125-153	19-95-180	17-96-181
.5SEs	0TE	n	196-98-0	199-86-9	74-186-34	74-190-30	41-148-105	45-160-89	32-146-116	34-123-137	28-87-179	33-94-167
		L	196-98-0	200-87-7	68-202-24	67-195-32	33-156-105	45-150-99	26-148-120	31-128-135	25-98-171	23-107-164
		m	198-96-0	200-88-6	71-187-36	75-188-31	41-149-104	45-153-96	31-139-124	31-126-137	25-100-169	25-105-164
	.5TE	n	196-98-0	200-85-9	77-178-39	78-193-23	43-147-104	46-152-96	32-141-121	34-124-136	28-86-180	33-94-167
		L	196-98-0	200-87-7	75-189-30	75-194-25	37-164-93	45-146-103	30-139-125	32-123-139	25-102-167	23-107-164
		m	196-98-0	202-86-6	74-182-38	82-187-25	42-152-100	44-153-97	33-135-126	31-129-134	25-101-168	25-105-164
	1TE	n	194-100-0	201-84-9	81-175-38	79-188-27	44-150-100	45-152-97	32-140-122	31-127-136	29-107-158	33-94-167
		L	194-100-0	201-86-7	81-183-30	76-193-25	40-162-92	45-147-102	30-139-125	31-128-135	25-104-165	23-107-164
		m	194-100-0	202-86-6	82-174-38	86-181-27	44-152-98	46-151-97	34-135-125	31-130-133	25-107-162	25-105-164
1SEs	0TE	n	176-112-6	193-87-14	58-180-56	52-180-62	22-126-146	24-129-141	20-103-171	20-110-164	18-89-187	19-92-183
		L	176-112-6	194-86-14	57-184-53	49-179-66	21-126-147	21-129-144	16-109-169	19-115-160	18-88-188	18-91-185
		m	176-112-6	196-83-15	58-176-60	59-170-65	24-133-137	21-132-141	18-107-169	19-111-164	18-91-185	19-88-187
	.5TE	n	176-104-14	195-83-16	58-180-56	54-178-62	22-127-145	24-131-139	20-104-170	21-111-162	18-89-187	19-92-183
		L	176-104-14	194-86-14	57-184-53	49-175-70	26-123-145	22-134-138	17-104-173	19-112-163	18-88-188	18-91-185
		m	176-104-14	200-80-14	58-175-61	59-173-62	29-129-136	21-136-137	18-104-172	19-114-161	18-91-185	19-88-187
	1TE	n	170-110-14	196-83-15	58-183-53	54-181-59	22-132-140	22-134-138	20-104-170	21-109-164	18-94-182	19-92-183
		L	170-110-14	196-84-14	58-189-47	49-180-65	26-125-143	21-136-137	17-105-172	19-112-163	18-88-188	18-91-185
		m	170-110-14	198-84-12	58-179-57	59-174-61	29-132-133	20-138-136	18-105-171	19-113-162	18-91-185	19-88-187

Table 5.19 IG DTCV committees built with CC validator compared to boosted DT ensembles

separate pruning

0SE	0TE	n	404-184-0	426-146-16	228-322-38	241-299-48	124-247-217	122-281-185	101-208-279	104-209-275	94-185-309	98-192-298
		L	404-184-0	420-168-0	226-295-67	229-259-100	114-238-236	125-250-213	101-189-298	100-186-302	88-167-333	90-167-331
		m	404-184-0	430-158-0	241-298-49	231-271-86	118-244-226	131-248-209	106-196-286	103-188-297	90-180-318	94-173-321
	.5TE	n	404-184-0	418-154-16	256-292-40	249-290-49	129-278-181	142-283-163	114-209-265	106-217-265	95-192-301	98-192-298
		L	404-184-0	458-130-0	261-283-44	242-275-71	127-269-192	143-260-185	107-199-282	104-186-298	88-173-327	90-167-331
		m	408-180-0	450-138-0	273-260-55	242-274-72	130-263-195	153-246-189	107-206-275	108-193-287	92-179-317	94-173-321
	1TE	n	404-184-0	422-150-16	275-277-36	244-295-49	127-291-170	144-282-162	113-214-261	108-225-255	95-191-302	98-192-298
		L	404-184-0	468-120-0	269-284-35	249-274-65	130-280-178	149-253-186	109-206-273	105-188-295	87-173-328	90-167-331
		m	404-184-0	442-146-0	281-272-35	244-275-69	133-277-178	155-249-184	114-204-270	108-196-284	92-180-316	94-173-321
.5SE	0TE	n	400-188-0	428-144-16	220-328-40	220-312-56	119-232-237	117-262-209	96-202-290	99-197-292	85-181-322	87-194-307
		L	400-188-0	420-168-0	187-321-80	193-296-99	110-218-260	116-236-236	94-185-309	96-169-323	75-166-347	87-155-346
		m	400-188-0	430-158-0	208-329-51	208-281-99	116-226-246	114-245-229	94-197-297	97-176-315	77-180-331	87-169-332
	.5TE	n	404-184-0	424-148-16	252-293-43	247-289-52	122-260-206	128-269-191	101-201-286	101-199-288	89-181-318	87-194-307
		L	404-184-0	458-130-0	246-273-69	224-273-91	114-249-225	117-252-219	102-181-305	98-170-320	83-165-340	87-155-346
		m	408-180-0	452-136-0	260-270-58	230-277-81	124-245-219	119-255-214	103-189-296	99-181-308	83-175-330	87-169-332
	1TE	n	408-180-0	428-144-16	260-290-38	253-284-51	124-262-202	130-271-187	100-201-287	101-203-284	89-178-321	87-194-307
		L	408-180-0	458-130-0	262-270-56	228-275-85	117-260-211	121-259-208	101-186-301	97-171-320	83-163-342	87-155-346
		m	408-180-0	442-146-0	274-275-39	233-279-76	120-259-209	128-253-207	104-189-295	98-179-311	83-176-329	87-169-332
1SE	0TE	n	388-196-4	436-148-4	173-328-87	174-308-106	99-215-274	102-213-273	87-172-329	88-164-336	69-165-354	71-155-362
		L	388-196-4	394-194-0	153-287-148	168-263-157	94-186-308	103-170-315	83-159-346	88-144-356	69-147-372	71-137-380
		m	388-196-4	408-180-0	163-307-118	174-271-143	96-208-284	103-182-303	83-165-340	88-145-355	69-153-366	71-145-372
	.5TE	n	428-156-4	440-142-6	198-323-67	187-321-80	101-220-267	109-218-261	88-173-327	92-160-336	70-165-353	71-155-362
		L	428-156-4	450-138-0	186-292-110	177-285-126	96-205-287	108-177-303	83-157-348	88-143-357	67-148-373	71-137-380
		m	432-152-4	446-142-0	194-307-87	186-278-124	100-219-269	108-202-278	83-162-343	91-142-355	70-151-367	71-145-372
	1TE	n	404-176-8	436-146-6	228-300-60	195-310-83	104-218-266	109-222-257	87-171-330	93-161-334	72-163-353	71-155-362
		L	404-176-8	448-140-0	206-291-91	180-283-125	95-204-289	108-181-299	84-153-351	92-139-357	69-146-373	71-137-380
		m	404-176-8	436-152-0	221-298-69	193-272-123	99-217-272	108-200-280	87-158-343	92-141-355	72-149-367	71-145-372

common pruning

0SE	0TE	n	452-128-8	432-156-0	244-324-20	243-310-35	117-291-180	128-301-159	94-257-237	106-253-229	90-245-253	98-238-252
		L	452-128-8	460-128-0	250-322-16	240-308-40	125-282-181	133-297-158	94-266-228	109-250-229	95-244-249	100-234-254
		m	452-128-8	444-144-0	255-295-38	240-302-46	123-274-191	137-287-164	99-243-246	103-250-235	95-227-266	99-235-254
	.5TE	n	436-152-0	448-140-0	249-313-26	242-308-38	123-286-179	139-292-157	94-257-237	102-258-228	89-247-252	98-238-252
		L	436-152-0	472-116-0	246-322-20	240-308-40	130-277-181	139-292-157	97-254-237	109-251-228	96-247-245	100-234-254
		m	436-148-4	456-132-0	253-294-41	242-298-48	125-272-191	143-282-163	98-245-245	102-250-236	95-230-263	99-235-254
	1TE	n	444-144-0	454-134-0	252-316-20	241-310-37	121-293-174	135-289-164	98-254-236	104-255-229	88-252-248	98-238-252
		L	444-144-0	480-108-0	250-322-16	238-309-41	129-283-176	135-295-158	98-254-236	111-251-226	92-252-244	100-234-254
		m	444-140-4	464-124-0	255-295-38	239-309-40	123-282-183	142-285-161	96-249-243	106-245-237	92-237-259	99-235-254
.5SE	0TE	n	420-168-0	418-170-0	204-314-70	197-310-81	111-249-228	120-260-208	94-227-267	105-216-267	90-205-293	99-195-294
		L	420-168-0	446-142-0	203-336-49	189-323-76	106-260-222	117-256-215	93-223-272	102-210-276	92-202-294	94-201-293
		m	420-168-0	430-158-0	213-315-60	191-318-79	107-253-228	119-255-214	95-221-272	101-210-277	92-199-297	94-197-297
	.5TE	n	412-176-0	412-176-0	211-309-68	203-298-87	115-246-227	121-252-215	95-219-274	103-215-270	90-205-293	99-195-294
		L	412-176-0	432-156-0	203-330-55	192-319-77	118-253-217	114-252-222	93-222-273	102-211-275	91-203-294	94-201-293
		m	412-176-0	428-160-0	216-311-61	190-317-81	117-243-228	116-254-218	96-221-271	102-210-276	93-199-296	94-197-297
	1TE	n	412-176-0	412-176-0	214-305-69	199-294-95	120-242-226	118-255-215	93-225-270	103-218-267	90-211-287	99-195-294
		L	412-176-0	426-162-0	203-325-60	192-319-77	120-250-218	115-254-219	93-226-269	102-210-276	92-203-293	94-201-293
		m	412-176-0	428-160-0	214-312-62	191-311-86	119-247-222	115-257-216	96-221-271	102-212-274	91-204-293	94-197-297
1SE	0TE	n	368-220-0	386-202-0	164-298-126	170-280-138	97-216-275	106-228-254	88-199-301	86-193-309	78-191-319	82-185-321
		L	368-220-0	386-202-0	161-292-135	165-288-135	94-216-278	103-231-254	86-194-308	89-186-313	75-192-321	81-185-322
		m	368-220-0	376-212-0	166-284-138	168-282-138	98-210-280	105-225-258	87-193-308	89-188-311	80-189-319	80-186-322
	.5TE	n	376-204-8	382-206-0	165-309-114	174-277-137	97-217-274	107-230-251	88-196-304	89-188-311	79-192-317	82-185-321
		L	376-204-8	376-212-0	166-309-113	165-289-134	97-222-269	103-238-247	85-193-310	90-185-313	76-189-323	81-185-322
		m	376-204-8	378-210-0	172-299-117	170-284-134	100-212-276	104-229-255	87-188-313	90-184-314	81-188-319	80-186-322
	1TE	n	380-200-8	390-198-0	175-302-111	166-288-134	96-215-277	105-235-248	87-197-304	88-189-311	79-192-317	82-185-321
		L	380-200-8	388-200-0	170-307-111	162-294-132	94-228-266	103-237-248	85-194-309	88-187-313	76-190-322	81-185-322
		m	380-200-8	384-204-0	175-297-116	166-292-130	98-214-276	104-227-257	86-190-312	88-186-314	82-186-320	80-186-322
.5SEs	0TE	n	400-188-0	408-180-0	193-323-72	190-316-82	106-247-235	114-260-214	91-225-272	100-216-272	86-209-293	93-206-289
		L	400-188-0	424-164-0	206-325-57	181-328-79	101-249-238	113-253-222	88-223-277	95-214-279	85-212-291	90-207-291
		m	400-188-0	418-170-0	209-314-65	182-319-87	102-242-244	112-256-220	91-220-277	95-210-283	85-205-298	90-202-296
	.5TE	n	404-184-0	406-182-0	203-317-68	193-317-78	111-245-232	115-256-217	91-221-276	99-221-268	86-211-291	93-206-289
		L	404-184-0	420-168-0	201-328-59	185-326-77	111-253-224	111-255-222	88-222-278	96-219-273	84-212-292	90-207-291
		m	404-184-0	416-172-0	205-325-58	181-328-79	111-243-234	113-252-223	91-220-277	95-220-273	86-208-294	90-202-296
	1TE	n	404-184-0	408-180-0	205-315-68	191-316-81	114-247-227	115-257-216	90-226-272	98-221-269	84-217-287	93-206-289
		L	404-184-0	424-164-0	203-326-59	186-329-73	112-258-218	113-257-218	89-222-277	96-217-275	87-210-291	90-207-291
		m	404-184-0	412-176-0	207-323-58	184-323-81	114-241-233	114-255-219	90-225-273	95-219-274	86-209-293	90-202-296
1SEs	0TE	n	356-224-8	384-200-4	157-308-123	157-292-139	85-219-284	94-230-264	76-200-312	80-186-322	71-189-328	72-200-316
		L	356-224-8	380-200-8	153-314-121	155-286-147	86-210-292	92-220-276	72-199-317	80-184-324	68-191-329	72-185-331
		m	356-224-8	376-208-4	158-305-125	160-285-143	87-208-293	92-223-273	74-192-322	81-186-321	71-189-328	71-190-327
	.5TE	n	360-220-8	394-190-4	162-314-112	164-283-141	85-220-283	96-227-265	76-200-312	83-181-324	71-190-327	72-200-316
		L	360-220-8	392-188-8	158-313-117	157-279-152	86-219-283	92-228-268	72-201-315	83-178-327	68-192-328	72-185-331
		m	360-220-8	384-200-4	160-316-112	161-279-148	87-216-285	93-226-269	73-190-325	84-180-324	71-189-328	71-190-327
	1TE	n	360-220-8	394-190-4	166-311-111	163-292-133	88-212-288	93-233-262	75-203-310	82-183-323	71-190-327	72-200-316
		L	360-220-8	394-190-4	163-312-113	156-283-149	86-221-281	91-233-264	73-200-315	80-183-325	68-193-327	72-185-331
		m	360-220-8	388-196-4	167-313-108	163-285-140	90-208-290	92-229-267	74-192-322	81-184-323	71-190-327	71-190-327

When we focus on the parts of the tables corresponding to 0SE and the training error factors of 0.5 and 1, we can see that the numbers of wins and losses get more or less equal in columns 5 and 6.

Building decision support systems based on several alternative DT models is reasonable, when the number of alternatives is low enough to let a human analyze them and understand the knowledge behind their classification. In this context, three or four sets of rules seem adequate, and DTCV committees provide better solutions of this size than bagging and boosting.

Ensembles containing more DT models are more accurate when built with bagging or boosting approaches, but their comprehensibility gets smaller and smaller with increasing number of combined models. Also for lower number of ensemble members, boosting models are not easy to interpret, because the weights controlling final decisions are quite far from intuitive.

For small numbers of ensemble members, lower accuracy of bagged or boosted trees in relation to CV committees, is accompanied by the fact that single trees are also less accurate. The power of bagging and boosting consists in averaging decisions of many models, which not necessarily are accurate alone. From the point of view of decision support it is quite important that the trees used by the model are as accurate as possible, because it makes explanations most valuable. Therefore, the trees provided by CV committees are more adequate for such applications. Their members are validated, which makes them more reliable or at least provides some estimation of their performance.

5.4.6 Algorithm Parameters Analysis

Large collection of results obtained with many method configurations facilitates more detailed analyses of particular parameters by focusing on properly selected results. Monitoring result changes caused by modification of a single parameter value may bring reliable conclusions on the parameter function.

Committee Sizes

To compare the performance of different committee sizes (member counts), the whole set of 13560 committee results can be split into ten groups (corresponding to the sizes) and the groups compared with paired t-test. It must be pointed out, and it concerns all the following grouped results analyses, that the results within each group are not quite statistically independent, because some committees consist of the same trees, are tested on the same data, and differ only in a single parameter which does not always cause changes in the decisions. Therefore, the assumptions of the t-test are not satisfied, however the tested populations are very large, so the differences we want to observe, are probably well reflected. The interpretation of the

Table 5.20 The results for particular committee member counts

size	1	2	3	4	5	6	7	8	9	10
overall	2	1	2	6	5	6	3	0	2	4
Gini	1	3	3	7	9	6	5	2	2	7
IG	2	2	1	5	2	4	3	2	2	7
QUEST	1	0	5	5	9	6	6	5	4	5
SSV	2	0	3	4	7	8	3	2	3	5
REP	2	0	3	12	7	11	9	7	6	4
CC	0	1	2	7	3	8	4	2	3	7
Degree	2	1	4	7	10	9	7	2	1	6
OPT	3	0	3	6	6	6	3	1	3	6
MEP2	1	2	5	4	12	8	8	9	6	8
DI	1	1	3	3	6	5	5	3	5	4

confidence level is certainly violated in such populations with repetitions, but it is not so important here (at least when we realize the violation of the assumptions).

The comparison of the 10 populations corresponding to subsequent committee sizes is depicted in Table 5.20. The first row below the header shows the numbers of not significantly worse results than the best of all considered sizes, determined on the basis of the results of all configurations. The following rows display similar comparisons for filtered data corresponding to given DT induction method (4 rows) and then for the 6 validation methods used in the test. It can be seen, that with different filters we obtain different images, but with a common pattern of larger values for sizes from 3 to 7 and another increase at 10. The larger numbers in the middle of the scope result from the models built with separate validation, while those at the end of the scope from commonly validated trees. It does not mean that other values are useless—even one dataset for which a method outperforms the others makes it very valuable, if only we have some meta-knowledge about the reasons (or at least the circumstances) of the success. Since it is not obvious, which size should be preferred, in different contexts, one may need to try significantly different member counts. Keeping in mind the violation of the within-sample independence, we must be aware, that real counts of insignificant differences are larger—large samples in the paired t-test strive to incorrectly claim statistical significance, so the committees of the highest scores do not outperform the others so much as one could read out from the table.

Another visualization of the results differences is given in Table 5.21. It shows the effect of pairwise comparisons between the collections of results corresponding to given committee size. In each cell of the table we can see three numbers: the first one is the number of significant wins of the committee size given by the row label over the committee size given by the column label, the last one—inversely, and the middle one is the number of datasets, for which no committee size significantly outperforms the other. The same information is also presented as the colored bars in the background of each cell. The length of each horizontal bar corresponds to the adequate number.

Table 5.21 Pairwise comparisons of committee member counts

	1	2	3	4	5	6	7	8	9	10
1		2-1-18	2-0-19	2-0-19	2-0-19	2-1-18	3-0-18	4-0-17	4-0-17	4-0-17
2	18-1-2		2-0-19	2-0-19	2-0-19	2-0-19	3-0-18	3-0-18	4-0-17	4-0-17
3	19-0-2	19-0-2		3-2-16	4-0-17	4-2-15	5-2-14	7-2-12	8-3-10	10-2-9
4	19-0-2	19-0-2	16-2-3		5-5-11	7-3-11	9-2-10	10-1-10	11-1-9	11-5-5
5	19-0-2	19-0-2	17-0-4	11-5-5		6-5-10	7-8-6	12-4-5	14-1-6	15-0-6
6	18-1-2	19-0-2	15-2-4	11-3-7	10-5-6		13-1-7	13-3-5	14-1-6	15-0-6
7	18-0-3	18-0-3	14-2-5	10-2-9	6-8-7	7-1-13		14-3-4	15-0-6	15-1-5
8	17-0-4	18-0-3	12-2-7	10-1-10	5-4-12	5-3-13	4-3-14		13-3-5	13-3-5
9	17-0-4	17-0-4	10-3-8	9-1-11	6-1-14	6-1-14	6-0-15	5-3-13		11-6-4
10	17-0-4	17-0-4	9-2-10	5-5-11	6-0-15	6-0-15	5-1-15	5-3-13	4-6-11	

From the table, we can conclude that, in general, the most successful CV committees are built from 4 to 7 members. This confirms that it is advantageous to combine decisions of several models, however it is not always advisable to take all 10 trees for the committee, because some of them happen to be sort of degenerate and can spoil the team work.

Probability Estimation

Table 5.22 summarizes the results in the context of probability estimation methods. On the left, we can see the numbers of datasets, for which the result of particular option was not significantly worse than the best of the three. On the right, selected filtered comparisons are presented as win-draw-loss counts of direct competitions between the parameter values.

The overall results seem to show, that both Laplace correction and m-estimates offer slightly better results than no correction (just proportions). However, the scores are so close to each other, that one should suspect more or less equal usefulness of all the variants. Indeed, a closer look at the filtered comparisons gives the information

Table 5.22 Comparison of probability estimation methods

mode	n	L	m
overall	7	9	9
Gini	5	9	10
IG	6	9	10
QUEST	8	7	8
SSV	9	9	8
REP	16	4	5
CC	10	7	9
Degree	11	8	9
OPT	11	7	10
DI	6	8	9

overall

	n	L	m
n		9-1-11	8-0-13
L	11-1-9		8-4-9
m	13-0-8	9-4-8	

IG

	n	L	m
n		8-1-12	6-1-14
L	12-1-8		9-3-9
m	14-1-6	9-3-9	

REP

	n	L	m
n		15-2-4	15-2-4
L	4-2-15		4-5-12
m	4-2-15	12-5-4	

that in some contexts (like REP) the corrections rather spoil than improve, while with Gini or IG criteria, they are quite successful. It should not be surprising that REP trees do not perform better with the corrections. They are pruned to maximize classification based on proportions, so the proportions perform the best. REP modified to respect other classification schemes would certainly bring significantly different numbers in its row.

It is also worth mentioning that the models using REP, which improve the overall result of simple proportions, are usually not too accurate. If we ignore the REP, the comparison of the probability estimates would be worse for proportions. Anyway, no probability estimation method can be pointed as a significant winner.

Training Data Error Factors

The idea of including training data error in the validation process has proven to be very advantageous. The tests have been done for the factors of 0, 0.5 and 1 and the results with respect to the three values are presented in Table 5.23.

Paired t-test with $\alpha = 0.01$ showed no significant difference from the best result in 3, 3 and 19 cases, respectively. So the TE factor of 1 seems to be quite clear winner—only for two datasets it has provided significantly worse results than the best of the three methods. Also, when looking at the scores for selected results, we can easily find out, that the value of 1 is the most successful. The example of REP is the extreme one with respect to the scores of 1TE factor. REP prunes as much as possible without loss to the validation sample classification. It may cause that some valuable parts of the tree are pruned because the validation sample (9 times smaller than the training sample) does not contain examples in this area of the data space. That's why the factor respecting the training data error improves the results so much.

Table 5.23 Comparison of training data error factors

factor	0	.5	1
overall	3	3	19
Gini	1	5	19
IG	1	4	18
QUEST	3	5	18
SSV	2	4	16
REP	2	3	21
CC	6	3	17
Degree	3	2	18
OPT	1	9	17
MEP2	4	11	14
DI	2	12	14

overall

	0TE	.5TE	1TE
0TE		0-3-18	1-2-18
.5TE	18-3-0		2-1-18
1TE	18-2-1	18-1-2	

REP

	0TE	.5TE	1TE
0TE		0-2-19	0-2-19
.5TE	19-2-0		0-3-18
1TE	19-2-0	18-3-0	

MEP2

	0TE	.5TE	1TE
0TE		3-1-17	3-2-16
.5TE	17-1-3		6-8-7
1TE	16-2-3	7-8-6	

Table 5.24 Comparison of standard error factors

factor	0	.5	1
overall	19	1	1
Gini	18	2	2
IG	18	1	2
QUEST	19	0	2
SSV	18	1	2
REP	19	13	3
CC	19	1	2
Degree	19	1	2
OPT	18	1	2
MEP2	19	1	2
DI	18	4	2

REP

	0SE	.5SE	1SE
0SE		7-13-1	18-2-1
.5SE	1-13-7		18-1-2
1SE	1-2-18	2-1-18	

CC

	0SE	.5SE	1SE
0SE		19-0-2	18-2-1
.5SE	2-0-19		19-0-2
1SE	1-2-18	2-0-19	

Standard Error Factors

Very interesting dependencies can be observed in Table 5.24 illustrating differences among methods of SE consideration. It must be pointed out that because of the difference in the sets of possible SE values for separate and common pruning, to calculate the overall summary and the filtered results respecting DT induction methods, only the models validated separately were analyzed.

The overall summary and all the filtered comparisons clearly show that the most accurate parameter is 0SE. It means that the methods do not underprune the trees (at least for the purpose of acting as committee members). Thus, further pruning spoils the committees. The context of committees must be kept in mind in such analysis, because slight underpruning of particular members may not be destructive for the whole committee, as classification of single data objects overfit by one tree may be rescued by other committee members.

Only in the case of REP models, the factor of 0.5 reaches a relatively large score, but we must still remember, that the REP-based committees belong to the least successful ones (of those used in the test).

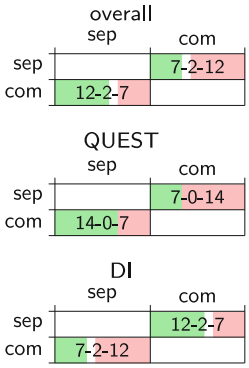
The knowledge about so large differences between the values of the parameters is very precious from the point of view of meta-learning, because it lets us easily restrict the model search space without much loss in the maximum accuracy obtainable. Thanks to focusing on the most promising parts of the model space, we can find attractive solutions in significantly shorter time.

Separate or Common Optimization

In general, committees of trees pruned separately provide lower accuracies than those optimized together for a common global pruning parameter. The results comparison is presented in Table 5.25. Common pruning is not applicable to REP, so the table

Table 5.25 Separate versus common optimization

mode	s	c
overall	9	14
Gini	9	13
IG	12	11
QUEST	7	14
SSV	10	12
CC	12	10
Degree	10	12
OPT	9	14
MEP2	11	10
DI	14	9



ignores the results obtained with this method. The overall result and a majority of the filtered views show the dominance of common validation, however the differences are small—both methods happen to outperform the other. It confirms again that in particular applications, the winners may be different. Even for the same task, the success of a parameter may be dependent on the settings of other parameters. Here, as discussed above, separate validation would be the winner in the case of smaller committees and common validation for larger ones. Such correlations are also a precious meta-knowledge, that can speed up obtaining attractive results by meta-search processes.

There is still a strong need for more meta-knowledge explaining why and when one method is better than another. This will certainly be one of the main subjects of future research in the area of advanced data mining.

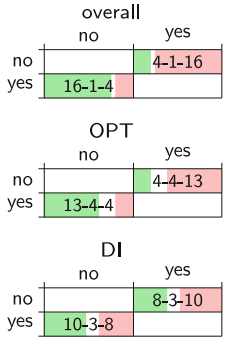
Standard Error Estimation Methods

In the case of common validation, several trees are analyzed in parallel, so a sample of error estimates is available and can be used to empirically evaluate the standard deviation of the error. The results obtained with configurations using sample-based SE estimation and theoretical estimates have been compared, and the scores are presented in Table 5.26. The labels tell whether sample-based estimation was used, so that "no" and "n" denote theoretical estimation and "yes" and "y" mark the results of sample-based calculations.

In the population of test results generated in the experiment, SE estimated from sample has been more successful than the theoretical one in all examined categories, however it must be emphasized, that the comparison concerns the factors of 0.5 and 1 (the way of SE estimation may be significant only if the SE correction is used) which, as discussed above, provide lower accuracies than the factor of 0.

Table 5.26 Comparison of standard error estimation methods (theoretical and sample-based)

mode	n	y
overall	5	17
Gini	8	17
IG	7	16
QUEST	5	19
SSV	8	15
CC	3	18
Degree	7	18
OPT	8	17
MEP2	8	17
DI	11	13



Standard Error Factors with Separate and Common Validation

More informative comparison of SE factors may be done when all variants of separate and common validation are distinguished at the same time. More precisely, we can compare 8 different parameter settings: the values 0SE, .5SE and 1SE used with separate validation, and the values 0SE, .5SE, 1SE, .5SEs and 1SEs working with common validation. The scores corresponding to the eight parameter combinations are presented in Table 5.27. The columns of separate and common validation are described explicitly and the methods of SE estimated from sample are placed after the dashed vertical line.

The table confirms the remarks given above, that the factor of 0 is the most successful, especially when used with common validation. Increasing the factor spoils the results. The decrease in accuracy is slightly less with SE estimated from samples, than with theoretical estimation and the least in the case of separate validation, but still the results are significantly worse than with no SE-based corrections.

Table 5.27 compares the results obtained with all possible values of the other parameters (like committee size, TE factor and so on), so the conclusions are still very general, averaged over many configurations. Some parameters may be successful only with some particular settings of other parameters, for example, in small committees or with Laplace correction used. Averaged, general summary does not show such possibilities. To draw conclusions, that may be more useful in particular cases, one should impose further restrictions on the set of compared results and examine the differences in such contexts. Context-dependent analysis should be performed on demand, when the context of interest is fully specified. The number of possible context is so large, that performing all analyses a priori is impossible. Mechanisms to extract context-dependent knowledge may be incorporated into automated meta-learning processes, dynamically searching for successful machines.

Table 5.27 Comparison of standard error factors, including separate and common validation

	separate			common				
factor	0	.5	1	0	.5	1	.5	1
overall	6	0	0	14	1	0	1	0
Gini	10	2	0	11	1	0	2	1
IG	10	5	0	13	1	1	0	1
QUEST	9	2	1	14	0	0	0	2
SSV	8	4	1	14	0	0	2	0
CC	8	2	1	13	0	0	1	1
Degree	7	1	3	16	0	0	1	1
OPT	6	3	0	15	0	0	0	0
MEP2	12	3	1	12	0	1	0	1
DI	9	2	1	13	3	2	3	1

overall

	s0SE	s.5SE	s1SE	c0SE	c.5SE	c1SE	c.5SEs	c1SEs
s0SE		17-3-1	21-0-0	6-0-15	15-1-5	19-1-1	13-2-6	19-0-2
s.5SE	1-3-17		21-0-0	3-2-16	11-1-9	18-0-3	10-2-9	17-0-4
s1SE	0-0-21	0-0-21		3-0-18	3-1-17	8-0-13	3-1-17	6-1-14
c0SE	15-0-6	16-2-3	18-0-3		19-0-2	19-0-2	19-0-2	20-0-1
c.5SE	5-1-15	9-1-11	17-1-3	2-0-19		19-0-2	3-5-13	19-0-2
c1SE	1-1-19	3-0-18	13-0-8	2-0-19	2-0-19		3-0-18	4-3-14
c.5SEs	6-2-13	9-2-10	17-1-3	2-0-19	13-5-3	18-0-3		19-0-2
c1SEs	2-0-19	4-0-17	14-1-6	1-0-20	2-0-19	14-3-4	2-0-19	

Table 5.28 Comparison of DT validation methods

validation	C	D	O	DI
overall	3	0	5	13
Gini	2	3	9	13
IG	3	5	3	11
QUEST	4	0	7	12
SSV	1	3	4	13

IG

	CC	Degree	OPT	DI
CC		9-1-11	11-3-7	10-0-11
Degree	11-1-9		12-1-8	8-0-13
OPT	7-3-11	8-1-12		8-0-13
DI	11-0-10	13-0-8	13-0-8	

C cost-complexity, *D* degree-based validation, *O* OPT, *DI* depth impurity

Validation Methods

Another subject for interesting observations is the efficiency of tree pruning methods in preparation of DTCV committee members. Of the six methods used in the tests, four (CC, Degree, OPT and DI) have provided complete lists of results (for maximum number of parameters combinations), so they can be reliably compared. The other two (REP and MEP2) are compatible with selected parameters only, and have been ignored in this comparison.

The comparison results are presented in Table 5.28. The clear advantage of the DI method may be a bit surprising and misleading. DI gives attractive results mostly for large committees with common pruning—in the areas where other methods obtain

poorer results, however in the comparison, the advantages in all areas (also, where all the methods record very weak results) are equally important. Although the globally successful machine configurations must never be forgotten, the most attractive parameter combinations must be searched locally, where the relations may be completely different from the global ones. As can be found in the table, the degree-based pruning which recorded the score of 0 in the global comparison, significantly outperformed CC, OPT and DI, respectively in 11, 12 and 8 tests of 21, so the 0 score can not absolutely be interpreted as uselessness of the method.

DT Induction Method

The last comparison within the DTCV committee algorithms experiment, concerns the parameter of DT induction method. The four values (Gini index, information gain, QUEST and SSV) have been compared in a similar manner as the other parameters: their series of results were collected for all the tests and filtered by validation methods. The comparison is presented in Table 5.29.

All the methods reached some significant wins over the others. Although Gini index obtained the lowest scores, it is still capable of outperforming all three remaining methods for some datasets. Information gain has recorded the largest number of successes, 9 datasets with the best (or close) results means 12 datasets with significantly worse results than may be obtained with one of the four methods. Therefore, again, no single method should be regarded as satisfactory. Instead, to determine the most suitable method, one should define the context first, and then analyze properly filtered results.

5.4.7 Summary

The experiment presented above, has been designed to examine miscellaneous aspects of DTCV committees, that in some applications provide an interesting balance between comprehensibility and accuracy. Decision tree models are especially

Table 5.29 Comparison of DT induction methods.

algorithm	G	I	Q	S
overall	3	9	5	7
REP	4	9	6	5
CC	3	9	4	7
Degree	3	7	4	7
OPT	4	8	6	6
MEP2	3	9	7	5
DI	3	9	7	4

overall

	Gini	IG	Quest	SSV
Gini		5-0-16	9-0-12	11-1-9
IG	16-0-5		13-0-8	11-1-9
Quest	12-0-9	8-0-13		10-1-10
SSV	9-1-11	9-1-11	10-1-10	

G Gini index, *I* information gain, *Q* QUEST, *S* SSV

attractive, when comprehensibility and accuracy come together. DTCV committees containing just several member trees can still be quite comprehensible, while offering more accurate decisions than single trees. Many techniques of DT construction, pruning and combining decision functions have been evaluated in a reliable test. Many interesting conclusions have been drawn from the results analysis and visualization.

A general conclusion may be formulated, that DT committees may be highly more accurate than single validated trees with no additional effort at learning time in relation to the approaches employing CV for tree pruning. Because the committees are constructed from the trees prepared during the validation process, building them is computationally cheaper than CV-based pruning, because no final DT needs to be induced. Slightly larger computational cost must be paid when the classification routine is called, but the additional cost is not big—all the members must be tested instead of a single tree, but it is still a constant factor equal to the number of committee members, multiplying the time of classification with a DT model, which is usually tiny, since classification with DT models is very fast.

Combining several decision trees into an ensemble does not necessarily mean the lack of comprehensibility. A decision support system based on a DTCV committee model, may easily present several rules when explaining decisions. Because the trees grown within CV are often very similar, the rules extracted from the trees for a particular data item can be very similar or even identical, so that the number of rules may be significantly less than the number of committee member trees. Alternative classification rules are desirable in the applications, where the most important premises of particular classification decisions are searched for. They point to different aspects of the classification problem, which may shed more light on the decision making process.

DTCV committees containing up to five committee members, are (on average) more accurate than models of the same size obtained with bagging or boosting techniques. So small members count is quite adequate for complex models that are expected to be comprehensible. At the same time, the committees of size greater than 2, significantly outperform single tree models. Although the area of superiority of DTCV committees over both single trees and popular ensembles, seems narrow (3–5 trees in the model), these member counts are very attractive from the point of view of robust and comprehensible decision support systems. Therefore, the CV committees often provide a golden mean between comprehensibility and accuracy (between single trees and large ensembles).

Extensive comparative tests have been quite easily performed, thanks to the versatility and rich test tools of the Intemi system. Thanks to the machine unification mechanism, testing 13560 committees, 100 single tree algorithms, 560 bagging approaches and 1120 boosting techniques, each with 10 runs of 10-fold cross-validation were conducted in relatively short time, because the decision trees underlying the committees were not built repeatedly for each committee, but were unified and reused. The results have been collected in a uniform manner facilitating fair comparisons with statistical tests for paired samples.

The results collected for 21 datasets, show that all the parameters may significantly influence the final performance of the committees. In the pursuit of the best

models, one always needs to try many different parameter combinations. Because no parameters are definitely best in all cases, special meta-level algorithms are necessary to determine which parameters should be used in particular applications. Such meta-search methods can take advantage of different kinds of meta-knowledge, which may speed up the search process. Different estimations of the distribution of winning configurations may be a form of such helpful meta-knowledge.

The results of the experiment include very precious meta-knowledge about the influence of particular learning parameters to the final results. Using the meta-knowledge, advanced search processes may omit testing some parameter settings without significant loss in their final achievements. Meta-learning approaches should take advantage of the meta-knowledge, to save time of further explorations. They can resemble human experts in not loosing time for doubtful approaches, while searching more thoroughly and systematically than humans.

Miscellaneous algorithms and data structures for meta-knowledge representation and gaining, certainly belong to the most important directions for future research in the area of computational intelligence. They will improve machine autonomy in learning from data and will significantly change the future, eventually replacing human experts in many disciplines with autonomous machine learners.

References

Almuallim H (1996) An efficient algorithm for optimal pruning of decision trees. Artif Intell 83(2):347–362. http://dx.doi.org/10.1016/0004-3702(95)00060-7

Bohanec M, Bratko I (1994) Trading accuracy for simplicity in decision trees. Mach Learn 15:223–250. http://dx.doi.org/10.1007/BF00993345, 10.1007/BF00993345

Breiman L, Friedman JH, Olshen A, Stone CJ (1984) Classification and regression trees. Wadsworth, Belmont

Breiman L (1996) Bagging predictors. Mach Learn 24(2):123–140

Cestnik B, Bratko I (1991) On estimating probabilities in tree pruning. In: Kodratoff Y (ed) Machine learning—EWSL-91, Lecture notes in computer science, vol 482. Springer, Berlin, pp 138–150. http://dx.doi.org/10.1007/BFb0017010, 10.1007/BFb0017010

Dietterich TG (2000) An experimental comparison of three methods for constructing ensembles of decision trees: bagging boosting and randomization. Mach Learn 40(2):139–157. doi:10.1023/A:1007607513941, http://dx.doi.org/10.1023/A:1007607513941

Esposito F, Malerba D, Semeraro G (1997) A comparative analysis of methods for pruning decision trees. IEEE Trans Pattern Anal Mach Intell 19(5):476–491

Fournier D, Crémilleux B (2002) A quality index for decision tree pruning. Knowl-Based Syst 15(1–2):37–43

Frank A, Asuncion A (2010) UCI machine learning repository. http://archive.ics.uci.edu/ml

Freund Y, Schapire RE (1996) Experiments with a new boosting algorithm. In: Machine learning: proceedings of the thirteenth international conference

Gehrke J, Ramakrishnan R, Ganti V (2000) Rainforest—a framework for fast decision tree construction of large datasets. Data Min Knowl Discov 4:127–162. http://dx.doi.org/10.1023/A:1009839829793

Grąbczewski K (2011a) Separability of split value criterion with weighted separation gains. In: Perner P (ed) Machine learning and data mining in pattern recognition, Lecture notes in computer science, vol 6871. Springer, Berlin, pp 88–98. http://dx.doi.org/10.1007/978-3-642-23199-5_7

Grąbczewski K (2011b) Validated decision trees versus collective decisions. In: Jedrzejowicz P, Nguyen N, Hoang K (eds) Computational collective intelligence. Technologies and applications, Lecture notes in computer science, vol 6923. Springer, Berlin, pp 342–351. http://dx.doi.org/10.1007/978-3-642-23938-0_35

Grąbczewski K, Jankowski N (2006) Mining for complex models comprising feature selection and classification. In: Guyon I, Gunn S, Nikravesh M, Zadeh L (eds) Feature extraction, foundations and applications. Studies in fuzziness and soft computing. Springer, Heidelberg, pp 473–489

Grąbczewski K, Duch W (1999) A general purpose separability criterion for classification systems. In: Proceedings of the 4th conference on neural networks and their applications, Zakopane, Poland, pp 203–208

Grąbczewski K, Duch W (2000) The separability of split value criterion. In: Proceedings of the 5th conference on neural networks and their applications, Zakopane, Poland, pp 201–208

Grąbczewski K, Jankowski N (2011) Saving time and memory in computational intelligence system with machine unification and task spooling. Knowl-Based Syst 24:570–588. http://dx.doi.org/10.1016/j.knosys.2011.01.003

Jankowski N, Grąbczewski K (2007) Handwritten digit recognition—road to contest victory. In: IEEE symposium series on computational intelligence. IEEE Press, USA, pp 491–498

Kononenko I (1998) The minimum description length based decision tree pruning. In: Lee HY, Motoda H (eds) PRICAI'98: topics in artificial intelligence, Lecture notes in computer science, vol 1531. Springer, Berlin, pp 228–237

Lim TS, Loh WY, Shih YS (2000) A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms. Mach Learn 40:203–228

Loh WY, Shih YS (1997) Split selection methods for classification trees. Stat. Sinica 7:815–840

Mingers J (1989) An empirical comparison of pruning methods for decision tree induction. Mach Learn 4(2):227–243

Niblett T, Bratko I (1986) Learning decision rules in noisy domains. In: Proceedings of expert systems'86, the 6th annual technical conference on research and development in expert systems III. Cambridge University Press, New York, pp 25–34

Quinlan JR (1987) Simplifying decision trees. Int J Man-Mach Stud 27(3):221–234. http://dx.doi.org/10.1016/S0020-7373(87)80053-6

Quinlan JR (1993) C 4.5: programs for machine learning. Morgan Kaufmann, San Mateo

Quinlan JR (1996) Bagging, boosting, and C4.5. In: Proceedings of the thirteenth national conference on artificial intelligence and eighth innovative applications of artificial intelligence conference, AAAI 96, IAAI 96, vol 1. AAAI Press/MIT Press, Portland, Oregon, pp 725–730

Torres-Sospedra J, Hernández-Espinosa C, Fernández-Redondo M (2007) Averaged conservative boosting: introducing a new method to build ensembles of neural networks. In: de Sá J, Alexandre L, Duch W, Mandic D (eds) Artificial neural networks—ICANN 2007, Lecture notes in computer science, vol 4668. Springer, Berlin, pp 309–318

Chapter 6
Meta-Learning

The problems of learning and meta-learning have been introduced formally in Sect. 1.1. Many learning algorithms have been proposed by the CI community to solve miscellaneous problems like classification, approximation, clustering, time series prediction and others. Each method specializes in models of some particular shape (decision trees, neural networks of specific architecture, nearest neighbors classifiers and so on) and tries to maximize some model quality function (or minimize some cost function). None of the algorithms can be regarded as better than all others. Each one has its own *inductive bias* , which can be shortly and informally defined as eligibility for solving some kinds of problems and less potential for learning other tasks, or in other words, the circumstances which make the method succeed. Large amount of algorithms and their different performance in particular applications have naturally risen questions about possibilities to select the most adequate and the most successful learning methods for particular tasks. Unfortunately, it is not easy to describe the inductive bias of learning machines in such a way that would be of much help in selection of the most suitable algorithms for particular task. This fact has born the idea of using CI methods in the pursuit of attractive learning machines, hence the field of meta-learning .

The task of algorithm selection is a natural consequence of using the divide-and-conquer technique for learning—instead of searching directly for the best model, it is often advantageous to select a learning machine to finally use its model. Such scenario shows the ultimate goal of meta-learning—improvement in object-level learning. Miscellaneous meta-learning approaches gather meta-knowledge about learning processes to eventually use it for model selection.

Respecting the ultimate goal of meta-learning and the perspective of real-world applications, the most adequate formulation of learning from data is addressed by the definition of time-limited optimal-learning problem $\mathscr{P} = (D, \mathscr{M}, q, t)$, where not only the training data D and the model space $\mathscr{M}$ are provided, but also a model quality measure q is strictly defined and a time limit t enforces a deadline for all calculations (see Eq. 1.5 on page 4).

K. Grąbczewski, *Meta-Learning in Decision Tree Induction*,
Studies in Computational Intelligence 498, DOI: 10.1007/978-3-319-00960-5_6,

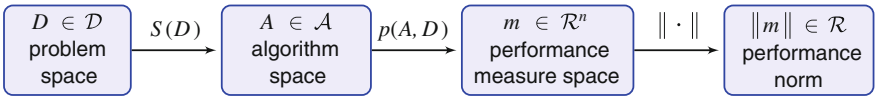


Fig. 6.1 Rice's model of algorithm selection problem

The quality measure is sometimes precisely defined (as predicted classification accuracy, the opposite of approximation error estimate and so on) and sometimes given by some imprecise description (for example: large accuracy accompanied by model comprehensibility). To make crisp optimization possible, imprecise definitions must be clarified. Although fuzzy problem statements are common, when presenting the goals to human experts, they must be defined more precisely for artificial learners.

Definition of time limit is usually very natural, because in practical applications we always need to keep within a deadline (of a data mining contest, of scheduled implementation of the model in a plant and so on). Even in research experiments, one needs to receive results in a reasonable time, so calculations can not run without end. Therefore, the definition of time-limited optimal-learning problem, perfectly fits the concept of algorithm selection.

The algorithm selection problem (ASP ASP) has been discussed already by Rice (1974, 1976), who presented an abstract model of the problem as in Fig. 6.1. The task is to find a mapping $S : \mathscr{D} \rightarrow \mathscr{A}$, such that for given data $D \in \mathscr{D}$, $A = S(D)$ is an algorithm maximizing some norm of performance $||p(A, D)||$. The problem has been addressed by many researchers and undertaken from different points of view (Bensusan 1999; Guyon et al. 2010; Grąbczewski and Jankowski 2007; Hilario 2002; Smith-Miles 2009; Vilalta et al. 2004).

Solving ASP is the ultimate goal of meta-learning, but not its only possible approach. Each task defined in such a way that the data D or model space $\mathscr{M}$ somehow refers to learning processes, should be seen as a task of meta-learning. So general understanding of meta-learning is coherent with the abundance of publications using the term. Various attempts to review and summarize the field have been undertaken (Brazdil et al. 2009; Giraud-Carrier 2008; Vilalta and Drissi 2001, 2002). Some surveys of systems and frameworks devoted to meta-learning have also been prepared (Brazdil et al. 2009; Vanschoren 2011). Because of large amount of methods, no review can be exhaustive and go into details of all approaches.

A survey is also presented below, in Sect. 6.1, and it does not aspire to be full, either. It is included to show the diversity of meta-learning concepts, to sketch the main interests and to present the approaches of further sections against appropriate background. Section 6.2 presents a general algorithm of meta-learning performing a search for attractive algorithms with feedback from validation and the subsequent Sects. 6.3 and 6.4 present two frameworks compatible with the general algorithm: one based on configuration generators and complexity control, and one using learning profiles to steer the search process.

6.1 Meta-Learning Approaches

The definition of meta-learning introduced on page 3 encircles all methods dealing with information about learning processes. So broad definition encompasses the whole spectrum of techniques aiming at gathering meta-knowledge and exploiting it in learning processes. Indeed, abundance of publications devoted to such methods use the term "meta-learning " in many different contexts.

As emphasized before, many different particular goals of meta-learning have been defined, but the ultimate goal is to use meta-knowledge for finding (construction of) more accurate models at object-level and/or to find them with as little resources (time and memory) as possible. Because, the ultimate task is very complex and requires much valuable meta-knowledge, a reasonable way to attractive solutions is to solve many simpler problems (in compliance with the divide-and-conquer principle) and use the results for more and more advanced meta-learning processes. This way of thinking has led many researchers to miscellaneous ideas and algorithms (including complex knowledge-based systems) for meta-level analysis of learning machines.

Some attempts to describe the algorithm selection problem and analyze it theoretically have been undertaken. One of the results is the very famous bunch of *no-free-lunch* theorems, which is very often referred to by CI researchers, although its practical impact is very weak. Because of its popularity, it is the subject of a discussion presented in Sect. 6.1.1.

One of the simplest approaches, claiming to perform meta-learning, combines decisions of object-level models, creating different kinds of ensembles. Section 6.1.2 discusses some methods of this family.

In another group of approaches, regression methods are used to predict learning-machines gains, expressed as numeric scores, so that in future applications, the learning data may be used to anticipate the scores before the methods are run. Such methods are shortly commented in Sect. 6.1.3.

Yet another group of methods strives for accurate ranking of learning algorithms or in other words, prediction which methods will provide best results for a given learning task. Section 6.1.4 is devoted to this kind of algorithms. Many approaches of this family end with unverified rankings, while others rank according to the results of a validation procedure run for each ranked method. The latter approach results from the conviction that building successful tools for automated selection of most suitable learning methods for particular tasks, requires integration of meta-level and object-level learning in a single search process with built-in validation of object-level learning machines and meta-knowledge acquisition and exploitation. Although they still end up with a ranking of algorithms, they are so different from the unvalidated ones, that should be regarded as a separate group of methods. Two algorithms worked out recently, are presented in Sects. 6.3 and 6.4. Both fit the general algorithm for meta-learning based on validation presented in Sect. 6.2. The former validates configurations of learning machines suggested by machine configuration generators and ordered by estimated complexity (Jankowski and Grąbczewski 2011). The latter tests learning machines in the order determined on the basis of learning profiles analysis.

These groups of meta-learning research do not cover all the area of this field. Many other approaches can also be classified as meta-learning. They concern both theoretical analyses and practical applications.

A taxonomy of inductive biases with respect to various properties have been introduced by Bensusan (1999). He presented a meta-learning system named THE ENTRENCHER, using induction to select appropriate generalization procedures.

An attempt to automatically learn inductive bias of learners was also undertaken by Baxter (2000). He presented a model to automatically restrict the hypothesis space for particular learning data.

Several ideas of *meta-learning ontologies* have also been proposed. They are especially interesting and auspicious, because meta-knowledge is usually gained with too large effort, to be extracted each time a meta-learning project is run. Thus, there is a strong need to collect the meta-knowledge is special repositories that can be used by meta-learning processes. Since meta-knowledge may have miscellaneous forms, construction of a versatile and practical knowledge base is extremely difficult. Although several systems based on the ideas of ontologies have already been created, for example CAMLET (Abe and Yamaguchi 2004; Suyama et al. 1998), IDEA (Bernstein et al. 2005) and e-LICO (Hilario et al. 2009, 2011), there is still much to do in this very important area, that will certainly play very important role in the future of machine learning.

Complex ontologies require adequate tools for their exploration. From this point of view, the approaches like the one of Kalousis and Hilario (2003) focused on a relational case-base representation for meta-learning seem very important. They proposed a multi-relational structure of information describing the classification datasets and prepared some similarity measures to facilitate similarity-based relational learning. Relational representation is certainly an interesting idea that needs much further exploration to provide more attractive tools for meta-learning. The information contained within well designed ontologies may bring efficient methods for construction of accurate complex learning machines. Properly organized meta-knowledge may bring attractive descendants of such methods for learning machine design as the approaches of machine configuration generators (Jankowski and Grąbczewski 2011, and Sect. 6.3) and KDD (Knowledge Discovery in Databases) workflows (Kietz et al. 2012).

Certainly, the gains of many fields should be joined in the common effort of providing functional and comprehensive ontologies for meta-learning. For example, fuzzy linguistic data summaries and fuzzy query systems used in decision support systems may also supply a valuable contribution to the goal (Kacprzyk and Zadrożny 2007; Zadrożny and Kacprzyk 2007). The ideas aimed at optimal decision support systems (Kosiński et al. 2010), concerning knowledge acquisition, opinion mining and sentiment analysis are also very close to the concepts of decision support regarding algorithm selection. Because of the necessity to deal with uncertainty and fuzziness in meta-knowledge analysis and meta-level reasoning, modern, flexible neuro-fuzzy systems (Cpałka et al. 2008; Rutkowski and Cpałka 2003) can also be successfully applied to meta-knowledge management.

The research classified to the field of *active learning* (Cohn et al. 1994), can also be treated as a part of the domain of meta-learning. Active learning techniques control the input data (usually by adjusting the training data sample) to optimize the gains of learning processes. As such, they also perform their tasks on the meta-level. The technique of active learning has also been used for meta-learning to result in an *active meta-learning* (Prudêncio and Ludermir 2008), aimed at saving time for collecting meta-examples and improving meta-learning accuracy. An example application of active learning in the realm of DT induction is the algorithm called ALASoft [Active Learning with Automatic SOFt labeling for induction of decision Trees (Su et al. 2009)]. It uses active learning to select high quality unlabeled data to be labeled by human experts and by a random forest, so as to improve the final DT induction process.

Similar comment can be made on the techniques of *deep learning* (Bengio 2009; Hinton et al. 2006), where analysis of deep model structures is very important for efficient learning.

Many applications of various meta-learning techniques to some specialized tasks can also be very precious for the domain, especially, when at some time, their results are placed in meta-learning repositories, enriching the meta-level knowledge about learning algorithms. Some examples of such applications are: feature construction methods proposed by Mierswa and Wurst (2005), the analysis of instance selection algorithms performed by Smith-Miles and Islam (2011), the application of meta-learning to ranking methods of clustering gene expression data and time series forecasting problems, by Prudêncio et al. (2011), choosing the most promising metric for distance based learners, as a compromise between random selection and computationally very expensive exhaustive search (François et al. 2011) or ranking clustering algorithms (de Souto et al. 2008).

More and more often, data mining systems are equipped with some tools for meta-knowledge collection and exploitation. Each of the systems like GhostMiner (Grąbczewski et al. 2004), MiningMart (Morik and Scholz 2004), Data Mining Advisor (arose from the experience gathered within the METAL project; Giraud-Carrier 2005), Intemi (Grąbczewski and Jankowski 2007, 2011) and others, deals with meta-knowledge in its own way. It will certainly take much time till some common representation of meta-knowledge and common formats of meta-learning ontologies appear and get accepted by large part of the CI community, because there is still much research necessary to discover most useful types of meta-knowledge.

6.1.1 No Free Lunch Theorems

In the world of computational intelligence, articles by Wolpert and Macready (1995, 1996) have caused very much reaction. Numerous authors refer to their work, but often cite their theorems in not adequate contexts. This is the main reason for devoting this subsection to that subject, because in practice of machine learning, the theorems are not useful (although theoretically correct).

So popular *no-free-lunch* (NFL) theorems were first published in the context of search processes (Wolpert and Macready 1995) and then of optimization algorithms (Wolpert and Macready 1996), but in fact they are equivalent, because the definitions of the algorithms performing search and optimization are the same. The difference between the two reports consists in vocabulary and some notations used.

The goal of this analysis is not to reject the NFL theorems, because, as mentioned above, they are theoretically correct, but to point their assumptions which unfortunately are easily disregarded by the commentators, leading to misinterpretations of the theses.

A Closer Look at NFL Theorems

Wolpert and Macready (1995, 1996) analyzed the scenario of search/optimization algorithms aimed at finding the optimal points in a space $\mathcal{X}$ with respect to cost functions $f : \mathcal{X} \to \mathcal{Y}$. They assume that both $\mathcal{X}$ and $\mathcal{Y}$ are finite sets. The algorithms work iteratively by selection of next points in $\mathcal{X}$ on the basis of a series of earlier visited points and the costs assigned to them. Formally, algorithms are functions

$$a : \bigcup_{i=0,1,\ldots,|\mathcal{Y}|-1} (\mathcal{X} \times \mathcal{Y})^i \to \mathcal{X}. \tag{6.1}$$

At ith iteration, on the basis of a sequence of pairs $(x_j, y_j) \in \mathcal{X} \times \mathcal{Y}$ for $j = 1, \ldots, i-1$, they determine the next point $x_i \in \mathcal{X} \backslash \{x_j : j = 1, \ldots, i-1\}$. With such definition of the problem, the fundamental NFL theorem is defined as:

> NFL**NFL theorem:** *For any pair of algorithms* a_1 and a_2, *a sample size* $m \leq |\mathcal{Y}|$ *and a series of costs* $\mathbf{y} = (y_1, \ldots, y_m) \in \mathcal{Y}^m$
>
> $$\sum_{f \in \mathcal{Y}^{\mathcal{X}}} P(\mathbf{y}|f, m, a_1) = \sum_{f \in \mathcal{Y}^{\mathcal{X}}} P(\mathbf{y}|f, m, a_2), \tag{6.2}$$
>
> *where* $P(\mathbf{y}|f, m, a)$ *denotes the probability that in* m *iterations of* a, *the costs calculated with* f *for subsequent results returned by* a are $\mathbf{y}$.

A natural conclusion from the theorem is that also the expected values of the outputs $\mathbf{y}$ are the same for all algorithms. Rephrased to informal language: each algorithm brings on average the same quality of its output. The result seems surprising, but such conclusion is possible because of the assumptions which are not congruent with real life observations, with the principles of neither natural nor machine learning (see the explanation below).

Analysis of Theorems' Assumptions

The crucial assumption responsible for the confusion about NFL is averaging over all possible cost functions, which means that all methods of results estimation are equally possible. This contradicts the natural principles and assumptions of learning, where we assume some consistency, for example, that similar decisions should bring

similar assessment and that input data contains some information about the problem being solved. If all cost functions are assumed equally probable, then we deal with completely chaotic environment, where costs of new solutions are completely independent of the solution-cost assignment available in the input. Wolpert and Macready (1996) notice that the uniform prior probabilities of cost functions are not what we usually observe, so it is more reasonable to analyze

$$P(\mathbf{y}|m,a) = \sum_{f \in \mathscr{Y}^{\mathscr{X}}} P(\mathbf{y}|f,m)P(f). \tag{6.3}$$

This makes a significant difference. Arguments that the priors are negligible when not incorporated into the optimization algorithms are not well-founded, because in fact, the non-uniform distribution of f is the foundation of each learning, where the cost estimation feedback is expected to be consistent in some way, and the algorithms adjust to this feedback.

If we accepted the uniform priors of cost functions in real life, it would make all our learning senseless. For example, imagine lessons of mathematics with a teacher using completely inconsistent ways of assessment. That would mean that the pupils could be given any possible mark for a correctly solved maths problem, because all ways of assessment would be equally possible, correct answers would be equally likely rewarded and penalized. It would not make any sense. Education would make no sense. Our children would be happy (at least temporarily), but we would easily spoil the world with such assumptions.

Treating each cost function as equally likely would bring the same expected effects as assignment of completely random costs, which is not what we observe in everyday life. The nature is ruled with some laws, which cause that learning is possible. When we analyze the definitions of "learning", "intelligence", "generalization from experience" and similar concepts, we can see that behind them all, there is the assumption that similar circumstances result in similar effects, which is meant by the term "consistency", used above. Learning is based on the belief that the same actions undertaken in similar situations will bring similar effect which, of course, is not always true (therefore surprises are possible), but in most cases it is. Otherwise there would be no place for learning or intelligence.

Also the principles of machine learning are founded on the assumption that we deal with the class of problems eligible for learning. We expect that there is some knowledge hidden in the data, and try to discover it. In the spaces analyzed by NFL theorems, such assumptions are not valid. In fact, the NFL theorems, rephrased informally, may get the forms like:

- In the space, where no learning is possible from assumption, each algorithm performs the same.
- Random costs can not be predicted.
- From the assumptions that generalization does not make sense we can infer that there is no point in generalizing.

All they are truisms, and also NFL is a truism in the guise of a formal statement.

Wolpert and Macready (1995, 1996) analyzed also some more optimization scenarios, for example, families of cost functions dependent on time, stochastic algorithms, but all they share the fundamental assumptions incongruent with the real life environment.

Learning Binary Logical Functions

The problem of learning arbitrary logical function $g : \{0, 1\}^n \to \{0, 1\}$ for a given natural n is very similar to those of NFL theorems. Here the counterpart of averaging over cost functions is averaging over datasets coming from different logical functions. When a subset of possible mappings is given, is it possible to predict the remaining part of the function? If the assumption is that all functions are equally possible (a priori), than the answer is "no". Each predictor gives the same average number of incorrect predictions.

For simplicity, assume that $n = 2$ and the training dataset consists of three assignments:

$$(1, 1) \mapsto 1,$$
$$(1, 0) \mapsto 1,$$
$$(0, 1) \mapsto 1.$$

Is it possible to predict the output of the logical function for the last possible input $(0, 0)$? Naturally, it is not, because there are two different boolean functions compatible with the training set, and each provides different output for the pair $(0, 0)$. Moreover, each of them can seem attractive, because one of them is the constant function of value 1 and the other is the well-known disjunction. Therefore, the results for each predictor, averaged over all possible functions, are equal.

If we assume equal priors for all possible functions, learning makes no sense, but if we know that there is some "simple" logical function hidden in multidimensional data, then learning is possible and the algorithms that prefer simple solutions are more accurate on average than random predictions.

Unjustifiable "Conclusions" from NFL

Unfortunately, many authors refer to NFL theorems and repeat their theses without the assumptions, leading their analyses astray and formulating incorrect "conclusions". For example, some researchers explain the need for many alternative algorithms with NFL theses, by claiming that because each learner has an area of competence, we need to develop miscellaneous algorithms to cover as broad area in the space of cost functions as possible and to create meta-level algorithms to select proper learning machine. It is true, that the possibility of trying diverse learning machines is usually advantageous, and that meta-learning algorithms are valuable, but it does not imply from the NFL theorems. When focusing on NFL theorems, we can just infer that any

combination of learning algorithms, being just another learning algorithm, has the same burden of NFL, that is, its average result is the same as any other's.

It is often claimed that the meta-level algorithm can recognize the kind of problem and select proper object-level learner, but there is no reason to accept the assumptions of NFL neither at object-level nor at meta-level. When equal priors are assumed at meta-level, any recognition is also equally accurate on average as random guess, so we can not defend the hypothesis that new learning methods can bring any new value.

Justifiable Conclusions from NFL

What one can reliably claim on the basis of NFL theorems is that each algorithm has the same performance on average. Creating more and more complex algorithms brings just other methods equivalent on average to completely random decision making. If one algorithm outperforms another for a collection of cost functions, than it must be outperformed by the other for some other collection of cost functions.

Most of the statements, derivable from NFL theorems, seem very depressing and discouraging, but only when we do not keep in mind the unreal assumptions.

One could also rephrase the pessimistic conclusions in an optimistic way, for example, that for each algorithm one can find data to confirm its advantage over another algorithm. So in fact, interesting results can be prepared for any method, if we know how the method works. Actually, whether this is an optimistic or pessimistic conclusion is the matter of the point of view, so the final opinion is up to the reader.

Conclusions on NFL

To summarize the conclusions on NFL theorems and their applications, it must be emphasized once again that the theorems, although mathematically correct, have no practical value for computational intelligence research. The theorems' assumptions make them unreal, applicable only to the spaces, where no learning makes sense, as no consistency in cost measurement is assumed. The spaces of all possible cost functions with equal priors are not congruent with what we experience in real life, where learning and intelligence are extremely valuable, because of prevailing consistency of phenomena.

Although NFL theorems are true, one should be very careful when drawing conclusions and always keep in mind the theorems assumptions. The authors can not be blamed for providing the theses, because they are true, but their followers often go too far in their interpretations. Also some comments in the original reports were exaggerated and put much stress on theses, but less on assumptions, but unfortunately it is normal (or even necessary) in contemporary science, as articles with no claims about revolutionary results, or at least such perspectives, are difficult to publish. Nevertheless, some comments of the authors are not too accurate. For example, Wolpert and Macready (1995) claim that the NFL theorems are not affected by relaxing the assumption of non-retracing search algorithms, and then explain that a retracing algorithm can be reduced to non-retracing one, for which the NFL is

true. It does not mean that the NFL is valid for retracing algorithms. Inversely, it is easy to prove that an algorithm that checks all possible points in the $\mathscr{X}$ space and then returns always the solution of the lowest cost, outperforms algorithms like random selection. Important assumptions are, unfortunately, so easily neglected. It can sometimes result in a serious confusion.

6.1.2 Ensembles of Decision Models

Simple committees performing majority voting or other simple decision combinations do not learn at meta-level, although contain the decision module dealing with object-level models. More "intelligent" decision modules perform some meta-analysis to decide which decisions of the ensemble members deserve more trust and which should be ignored as probably wrong. Such approaches are often referred to as model *stacking*, because on top of the object-level models (more precisely: their results), a meta-level learner is applied as the decision module. There are plenty of publications about such models, so that all of them can not be mentioned here. A selection of algorithms with interesting activity on meta-level is presented here.

Some of the successful approaches have been proposed by Prodromidis and Chan (2000) and Stolfo et al. (1997) (parallel, distributed Java Agents for Meta-learning). *Meta Decision Trees* (properly adapted C4.5 decision trees) have been used by Todorovski and Dzeroski (2003) and Zenko et al. (2001) to determine which model to use. In NOEMON system (Kalousis and Theoharis 1999; Kalousis and Hilario 2000) stacking has also been referred to as meta-learning.

Another type of methods in the category of model combination is characterized by manipulation of the training data, in order to create specialized models and then appropriately combine their decisions. They perform some meta-analysis to build more stable decision makers. Very popular examples are methods like *bagging* and *boosting* (also called *arcing*) (Breiman 1996, 1998; Dietterich 1999; Freund and Schapire 1996; Quinlan 1996; Torres-Sospedra et al. 2007). These meta-level techniques have been run not only to create ensembles, but also for many other tasks. For example, in DT induction it has been applied to feature construction performed at each DT node, to stabilize the process susceptible to significant changes caused by small variations in the training data sample (Vilalta and Rendell 1997).

Meta-level analysis may lead to estimation of the areas and degrees of competence of each base learner, to provide more reasonable decision of the decision module. As a result we get undemocratic committees , also heterogeneous ones. For example, meta-learning by *arbitration* and *combining* was proposed by Chan and Stolfo (1993, 1996). In their approach, the arbiters were models built on top of two other decision makers and deciding which of the two underlying models should be used for particular decisions. Binary trees of arbiters were constructed in bottom-up fashion, so first, the members were organized in pairs and an arbiter was trained for each pair. After that, the newly created arbiters were organized in pairs, and the next level of arbiters was created. The procedure was finished, when a single arbiter was created on top

of the whole structure. After that, this arbiter was queried, to get the decisions of the whole hierarchy.

Duch and Itert (2003) defined *incompetence functions* to describe (in)competence of each committee member in particular points of the data space. Jankowski and Grąbczewski 2005 introduced solutions of reflecting global and local competence in final ensemble decisions. Both kinds of competence were expressed as factors, used for weighting members decisions in the final decision making process of the ensemble.

Yet another approach to meta-learning by stacking was proposed by Kadlec and Gabrys (2008). In their *Learnt Topology Gating Artificial Neural Networks* each object-level committee member (*local expert*) was accompanied by a *gating network* to learn the performance of the local expert and link it to the positions of samples in the input space.

Learning hierarchical models has also interested Johansson (2007), the author of the Genetic Rule EXtraction (G-Rex) system, who has presented six approaches to meta-level analysis of ensemble members. In one of the approaches, he has examined several methods of combining artificial neural networks (ANNs). In another experiment, he tested his GEMS algorithm (Genetic Ensemble Member Selection) as a mean to combine some of a large number of trained ANNs into hierarchical ensembles. Yet another study of Johansson (2007) concerned potential advantages of restricting ANN training to a subset of the training data in order to use the remaining part for validation and selection of ensemble members.

Cornelson et al. (2003) used the term "meta-learning" to reflect the process of learning from data describing the performance of many information retrieval (IR) algorithms. A classifier trained on the meta-level was successfully used to combine the IR algorithms. In example applications it significantly improved precision while preserving the same level of recall available with base learners.

6.1.3 Meta-Level Regression

It would be very easy to create a ranking of algorithms if we could predict the accuracy of the candidates. The approaches to meta-level regression try to predict the accuracies of different learning machines on the basis of the training data containing dataset descriptions. Bensusan and Kalousis (2001) and Köpf et al. (2000) have applied regression algorithms to predict accuracy of given model (usually a classifier) from the input consisting of series of values derived from information theory and statistics (describing the datasets). To create a ranking of algorithms, they created one regression model for each algorithm, and ordered the learning machines according to decreasing predicted accuracy.

Janssen and Fürnkranz (2007, 2010) have also used regression algorithms for meta-level learning, but with significantly different goal. They investigated possibilities of learning heuristics for rule induction and concluded that the standard rule learning heuristics were outperformed by the meta-learned ones.

Fig. 6.2 The scheme of algorithm ranking approaches

6.1.4 Rankings of Algorithms

The tasks of algorithm selection and algorithm ranking are in fact equivalent, because a ranking may be the basis for selection of its top-ranked methods and inversely a ranking can be created by multiple selection of subsequent items. Therefore, algorithm ranking is even more frequent form of the problem.

The task of algorithm selection defined by Rice (1974) and depicted in Fig. 6.1 on page 234, often gets reduced to the problem of assigning optimal algorithm to a vector of features describing data. Such approaches are certainly easier to handle, but the conclusions they may bring are also significantly limited. Separating meta-learning (ranking) and object-level learning processes simplifies the task, but implies resignation from on-line exploitation of meta-knowledge resulting from object-learners validation.

$$\mu_y$$

This concept lies behind the most popular approach to meta-learning (probably the largest so far), that was initiated by the meta-learning project *METAL*. It was focused on learning rankings of algorithms from simple descriptions of data. The idea followed by the approaches of METAL and its descendants is sketched in Fig. 6.2. Given a dataset representing a problem to be solved, first, the dataset is described by some features (*meta-features*), and on the basis of this description, a ranking of learning algorithms is generated.

In the first enterprises of this kind, *meta-attributes* were basic data characteristics like the number of instances in the dataset, the number of features, the types of features (continuous or discrete, how many of which), data statistics and so on. Features resulting from more advanced statistical analysis and information theory have also been proposed, for example by Engels and Theusinger (1998). A Data Characterization Tool (DCT, Lindner et al. 1999) was prepared to describe datasets with many meta-features, including statistical data, information theory based features, meta-features describing single object-level features and relations between pairs of the object-features.

Provided a description of a dataset, rankings were generated by *meta-learners* in such a way, that for each pair of algorithms to be ranked, a classification algorithm was trained on two-class datasets describing wins and losses of the algorithms on some collection of datasets, and after that, decisions of the meta-classifiers were combined to build the final rankings (Brazdil and Soares 2000a). The same technique was also used by Kalousis (2002), Pfahringer et al. (2000) and Soares et al. (2001) and in many other approaches, as it is a simple, intuitive and quite successful solution.

Another kind of ranking methods, examined within the METAL project, consisted in combining the results of comparisons between gains of candidate learners recorded for a number of datasets available in the knowledge base. The combinations that determined the ranks of the algorithms, were calculated according to one of several measures (Brazdil and Soares 2000a):

- *average ranks* (AR), calculating just the mean rank obtained by the learners for all datasets of the knowledge base,
- *success rate ratios* (SRR), calculating the mean of success ratios between the method of concern and all the others for all datasets of the knowledge base (for methods A and B and data D, the ratio is $SRR^D_{A,B} = \frac{1-Err^D_A}{1-Err^D_B}$, where Err^D_X is the error rate of algorithm X obtained for dataset D),
- *significant wins* (SW), calculating the mean of estimated probabilities of winning between the algorithm of concern and each of the others (the probability that algorithm A wins against algorithm B was estimated by the fraction of the count of datasets with the results of A statistically significantly better than those of method B and the number of all analyzed datasets).

The group working on METAL has also introduced an index to assess relative performance, *adjusted ratio of ratios* (ARR) combining ratios of accuracy and time (Soares 1999; Soares and Brazdil 2000):

$$ARR^D_{A,B} = \frac{\frac{Acc^D_A}{Acc^D_B}}{1 + \log\left(\frac{T^D_A}{T^D_B}\right) * X}. \tag{6.4}$$

In the formula, Acc^D_Y and T^D_Y are accuracy and time of algorithm Y on data D and X is a parameter interpreted as, "the amount of accuracy we are willing to trade for 10-times speed-up". Another index, called *relative landmark* (RL), was proposed for cases involving $n > 2$ algorithms:

$$RL^D_A = \frac{\sum_{B \neq A} ARR^D_{A,B}}{n-1}. \tag{6.5}$$

Ranking quality was estimated by comparison to the ideal one (obtained after collecting the results for the test dataset). As the measures of ranking comparison they used some statistical methods: Spearman's rank correlation coefficient , Friedman's significance test and Dunn's multiple comparison technique.

As a contribution of Brazdil and Soares (2000b) and Brazdil et al. (2003) more interesting meta-attributes were proposed, including advanced statistical measures of datasets and indices derived from histogram analysis or information theory. The authors used k nearest neighbor (kNN) algorithm to choose similar datasets and tested the ranking methods described above: AR, SRR and SW.

6.1.4.1 Landmarking

Dataset descriptions consisting of just statistics about the data and other information about the representation form of the data, are not satisfactory for ranking algorithms, because they ignore much information existing in the data, which makes particular learners succeed or fail. An interesting step forward was proposed by Pfahringer et al. (2000) and called *landmarking*. The idea was to use meta-features measuring the performance of some simple and efficient learning algorithms (*landmarkers*) like linear discriminant learners, naive Bayesian learner or C5.0 decision tree inducer. It has proven to perform better than the methods based on descriptions derived from information theory (Bensusan and Giraud-Carrier 2000).

The results provided by the landmarkers were used as features describing data by meta-learners, aimed at recognizing which algorithms perform better for particular data. The set of algorithms used at meta-level contained C5.0 decision trees and rules, boosted C5.0, some linear discriminant learners and decision tree induction methods with linear discrimination as node splitters, naive Bayesian classifier and nearest neighbors approaches.

The next step forward in landmarking was introduction of meta-attributes describing relations between results instead of accuracies (*relative landmarking*). Fürnkranz and Petrak (2001, 2002) and Fürnkranz et al. (2002) extended the descriptions by ranks of landmarkers, order of landmarkers (inverse of ranks), results of pairwise comparisons between accuracies of landmarkers (+1, -1, ?) and pairwise accuracies ratios. To facilitate landmarking by the target algorithms (usually of larger computational complexity), they proposed *subsampling*—the original datasets were reduced to smaller samples before application of the learning procedures.

Relative landmarking and subsampling were combined together by Soares et al. (2001), who presented some experiments to illustrate advantages of combining the two improvements.

A serious disadvantage of methods like SRR, ARR and SW, is that they perform pairwise comparisons of algorithms, so when the number of candidate algorithms reaches tens of thousands or even millions, they are not applicable. Also, the statistical measures, used to compare rankings to the ideal one, are not too adequate for practical algorithm selection problems, because they measure similarity between rankings with the same attention paid to top-ranked and bottom-ranked methods. This is quite suitable for comparisons of several methods, but not for selection of the best algorithms from large sets of candidates.

Another system advising which classifier to use and performing classification was NOEMON (Kalousis and Hilario 2000; Kalousis and Theoharis 1999). Its model selection was also based on pairwise comparisons of algorithms results. The meta-learning space contained miscellaneous data characteristics resulting from statistics and concentration analysis (histograms), and some performance measures of algorithms like accuracy, training time, execution time, resource demands and so on. As the meta-learners, trained on results of pairs of algorithms, NOEMON used kNN and decision tree algorithms.

Many other landmarking related solutions have also been proposed. An interesting idea was presented by Peng et al. (2002) in DecT system, where data characteristics were derived from the structure of C5.0 decision trees, built on the data. Features related to the shape and size of the trees induced from the dataset, were used to extend the space of data description for the purpose of meta-learning. Similarly to other approaches, they used kNN to select similar datasets, ARR to rank candidate algorithms, and Spearman's rank correlation coefficient to estimate rankings.

More advanced DT analysis was proposed by Bensusan et al. (2000). Their *typed higher-order inductive learning* can be seen as landmarking with some higher-order information derived directly from decision trees, instead of just some characteristics of the resulting trees. They created a special meta-learning framework to facilitate typed higher-order inductive learning.

Providing many features to describe data for the purpose of meta-learning, causes a problem that needs to be solved when the number of meta-features gets large in relation to the number of meta-cases collected for learning. To make meta-learning feasible and prevent meta-learner from building decision models on accidentally correlated features, proper feature selection techniques have been applied at meta-level (Kalousis and Hilario 2001; Todorovski et al. 2000).

Meta-learning has also engaged methods related to clustering . Todorovski et al. (2002) proposed using *Predictive Clustering Trees* (PCT) for ranking and illustrated the method on the problem of ranking learning algorithms according to predicted performance. Predictive Clustering Trees (Blockeel et al. 1998) are multi-split decision trees built with minimization of intra-cluster variance and maximization of inter-cluster variance. In application to meta-learning, the main concern of the induction of PCTs is that clusters should contain data with similar relative performance of algorithms. Meta-data for learning was obtained from statistics, information theory and landmarking methods.

6.1.4.2 A Critique of Passive Ranking Methods

A possibility of ranking algorithms ideally, without running them on the data at hand, would be found with great enthusiasm by many people grappling with data analysis. Unfortunately, the task is very hard. Even with many restrictions imposed on the kind of data and the family of algorithms to be ranked, it is not so easy to provide optimal (or just close to optimal) solutions.

Very naive approaches based on simple data characteristics are doomed to failure, because the information about the methods that can be most successful, is usually encoded deeply in the decision borders and other properties of the data, not in simple information about the form of data. Thus, neither discrimination methods applied to pairs of algorithms nor regression approaches trying to predict the resulting accuracy from simple data descriptions have much chance to gain the goal.

Actually, it can be stated that the task of building rankings of algorithms on the basis of simple data characteristics is ill-posed, because in extreme cases, two datasets with the same descriptions may require completely different methods to be

properly modeled. Moreover, even simple data transformations like discretization or conversion of symbols to continuously valued features, move the dataset representation to very distant point in the feature space of dataset descriptions. As a result, such data transformations may completely change the rankings of methods, even when the information within the dataset remains almost the same.

Of course, transformations which change data significantly, can cause that completely different methods provide attractive results, but this is the result of the changes made inside the data, that usually can not be described by means of simple feature counts, ratios of symbolic and numeric features or simple statistics.

Reliable meta-learning can not just suggest final decision module on the basis of the current form of the data. It should also give hints about the transformations that should be performed to make final learning easier or improve the results.

Landmarking is definitely a step in the right direction, because it tries to take advantage of the knowledge extracted by landmarkers in their learning processes. Nevertheless, methods of landmarking are still passive, in the sense that the selection of landmarkers and of meta-features is done once at the beginning of the process. The resulting ranking can also be called static, because it is not verified—no feedback is expected and no adaptation is performed.

To provide a trustworthy decision support system, the rankings it provides should be validated. No human expert would blindly trust in raw rankings estimating algorithms eligibility for given task. Comparison of some rankings obtained with a method to the ideal ones, is not sufficient to guarantee the quality of other rankings obtained with the method.

Another problem of most ranking approaches is that to estimate the quality of a ranking it is usually compared to the ideal one with the Spearman's rank correlation coefficient (Berrer et al. 2000; Brazdil and Soares 2000a; Soares 1999), which does not reflect what we really expect from meta-learning algorithms. Usually, we do not need the whole ranking to be very close to the ideal one. When the goal is to select the most attractive algorithms from a large set of methods, really important is just the beginning of the ranking, because the positions close to the end will never be tried. Naturally, if there are just several candidate machine configurations examined (as in most applications published so far), the whole ranking can be regarded as the beginning, but a versatile meta-learning algorithm must be capable of algorithm selection in much broader domains. It is not so important whether a learning machine configuration is placed at rank 50000 instead of 100000, but it is very important whether a configuration is at position 1000 while it should be at 1 (though the difference between ranks is much smaller). The measures used in the publications cited above, impose the same penalty to rank differences at the top and at the bottom of the rankings. Giving priority to rankings accurate on average, can bring a winner ranking with very poor methods placed at the top.

In machine learning, it is quite common to separate data analysis process into data preprocessing stage and final learning. As discussed in Sect. 5.1, it sometimes leads to incorrect conclusions from experiments, especially when supervised data transformations are performed. Such split into stages is even less suitable for automated decision support. Often, the data transformation stage is more important in the whole

process of knowledge extraction, than just the final learner which can be as simple as linear discrimination. In serious meta-learning applications, we are not interested in raw rankings of simple methods, but in complex machine combinations that model the data as accurately as possible. Thus, the whole complex machines must be the subject of meta-learning, not just the final decision makers.

The ranking methods, mentioned above, restrict the set of possible solutions to a fixed number of predefined learning machine configurations, so they can not discover new successful combinations of techniques. Advanced meta-learning tools should provide such possibilities, so they need to actively construct complex machine configurations and estimate their attractiveness.

6.1.5 Meta-Learning as Active Search

In general, it is not possible to predict the structure and configuration of the most successful learner on the basis of simple and inexpensive description of data. Providing reliable rankings requires complex processes of model construction including adjustment of training parameters for different parts of the model, construction of hierarchies, combining miscellaneous data transformation methods and other adaptive processes. Candidate machines, whether simple or complex, need to be validated and undergo complexity analysis to select models most required by the user, as we usually prefer simple and comprehensible solutions. In such approaches to meta-learning, complex processes use meta-knowledge provided by experts or gained during automated meta-level analyses, and gather new meta-data to improve pending and further applications. When meta-knowledge is used to control a search for successful learning machines, the process may be called an *active search*.

A meta-search process with validation of machine configurations has been proposed by Duch and Grudziński (2001, 2002). Their method was a search in the space of learning machines parameters. It was illustrated in application to a similarity-based classification learner. Although its use of meta-knowledge was restricted to iterative selection of the learner that gained the largest accuracy estimate, it can be treated as a very simple example of active search.

In the approach of Jankowski and Grąbczewski 2011, meta-learning was conducted as more sophisticated search process driven by heuristics (created and adjusted according to proper meta-knowledge) protecting against spending time on learning processes of poor promise and against the danger of combinatorial explosion.

Meta-learning can be regarded as successful only if it efficiently uses the time it is given. Hence, it must be realized within as efficient CI environment as possible. Such environment must support complex machines development, learning and analysis. During recent years, a robust CI framework *Intemi*, that efficiently manages time and memory, has been developed (Grąbczewski and Jankowski 2011). It has made meta-learning research easier to conduct: thousands of machine configurations can be easily tested, their results analyzed and meta-level algorithms created and tested in a robust way.

The Intemi system supported implementation of a complexity based meta-search that fulfills the requirements of active machine configuration construction, validation and ranking (Jankowski and Grąbczewski 2011). In this approach the key element is the order in which machine configurations are validated. The process can be supplied with appropriate *configuration generators*, capable of machine configuration construction and respecting the feedback coming on-line from validation processes. The candidate configurations are subject to complexity prediction and are ordered in the queue for validation. Rough estimates of the amount of time and memory, try to evaluate machine efficiency in the validation scenario, so as to test candidate configurations from the simplest ones to more and more complex. Exact prediction is impossible, so additional mechanisms prevent large overhead in resources consumption. Such techniques help in testing most promising machine configurations first, so they make optimal usage of the time assigned for calculations. This is the main goal of the approach—to find as attractive and as simple model as possible in the specified amount of time. The algorithm is discussed in detail in Sect. 6.3.

The approach to meta-learning as a complexity-driven search process, also has some shortcomings: when there are many algorithms of similar complexity, the order defined by rough estimators is in fact random, so finding the most attractive ones would require testing all of them. An alternative is to use methods based on *learning profiles*, that is, collections of values describing learning processes. The profiles are very helpful in finding datasets for which similar behavior of the processes has been observed, for example, there are similar relations between results obtained by the processes. On the basis of similar datasets, and more precisely of machine rankings observed for the similar datasets, one can easily build new rankings of candidate machines. Profiles can be actively adjusted to current state of knowledge extracted from the experiments, so that conclusions about dataset similarities and rankings of algorithms to try in next steps can be updated on-line during the search process. The profile based approach is presented and discussed in Sect. 6.4. Similar observations inspired the research done independently by Leite et al. (2012). The method of *active testing*, they proposed, is also a pursuit for the most successful learning machine, that selects the most promising algorithms and tests them with much care for spending as little time for the computations as possible.

6.2 Meta-Learning as Search with Feedback from Validation

Many meta-learning approaches discussed above, create rankings of algorithms on the basis of some heuristics, but most of them do not validate the rankings. This is the most important difference between what they do and what is the usual practice of human experts. We should never forget, that the ultimate goal of meta-learning is to obtain better results on the base-level of learning. Therefore, when a learning problem is given, experts try to use their knowledge to point some algorithms as the most likely to succeed in learning the task, and candidate solutions are subject to verification. Results of the validation are analyzed by the experts to draw conclusions about

adequacy of used methods for this particular task. On the basis of these conclusions, new candidates are picked from the space of machine configurations, to be validated in the same way as the previous candidates. Often, when no significant information about the data to be learned is available, the first choice of candidate methods is almost random or the selection is done with care for diversity of algorithms, because running several learners of different nature may bring rough information about which techniques are most successful. Depending on the time available for analysis, the cycles of algorithm selection, validation and results analysis before the next selection, can be performed many times.

Stating the same more formally, an expert is given a time-limited problem $\mathscr{P} = (D, \mathscr{M}, q, t)$, so the goal is to find possibly best (according to q) solution contained in a search space $\mathscr{M}$, within time t. To provide attractive solutions, the expert needs to pay much attention to the quality measure and defined time restriction. The quality measure is used by the expert for validation of candidate machines, and influences decisions about what meta-knowledge to use when selecting next candidates and how to collect new meta-knowledge, that can be helpful in further search process. The time restriction must also be actively monitored. It may force decisions about not trying some complex machines that, although suspected of success, could not be validated within the time. Instead of selecting unvalidated machines, it is more reasonable, to validate some less complex machines and select among them. Pursuing the goal, the experts (although not in a formal, systematic way) order testing tasks during the progress of the search and build meta-knowledge based on experience from passed tests. They try to avoid test tasks, that could consume too large amount of time, since it would decrease the probability of finding the most interesting solutions.

The same rationale should be the foundation of automated meta-learning machines. It lies behind the tools presented in this book: the meta parameter search (MPS) machine presented in Sect. 4.4.1, and its generalization, presented below. Search, validation and learning from feedback are three pillars of the approach. Good selection of algorithms for the three substantial roles is the key to really successful model selection. The quality measure should be reflected by the validation procedure. To maximize the expected quality of the best model found, with respect to time restrictions, candidate machines must be tested in adequate order. During the search process, meta-knowledge of different kinds may be extracted and may help in choosing further steps. The meta-knowledge may come from the analysis of correlations between machine configurations and obtained results, between subparts of machines and others, in the context of their performance, complexities of machines and so on.

Automated procedures can be even more successful than humans, because of more regular search (not susceptible to omitting interesting solutions), better memory of already tested configurations (helpful in avoiding repeated calculations) and significantly greater possibilities in exploring large databases of results obtained for other problems.

6.2.1 The Algorithm

Because, many meta-learning algorithms may share the idea of iterated generation of candidate learning machines and observation of their behavior in test scenarios, it is advantageous to have a general algorithm, performing common functionality and using appropriate components to realize the parts responsible for particular approaches.

The algorithms differ in the strategy of selecting learning machines for observation in tests and in the objectives estimated from tests and optimized during the search process. In practice, the processes must respect some limitations of memory and time. Memory limitations may be ignored, because usually learning algorithms are constructed in a way facilitating running them on available machines and in the worst case, one can run tests in sequence or find a computer with larger amount of RAM to avoid running out of available memory. As a result, the assumption that memory limitations do not affect the final result is not a serious misdemeanor. Time limits are more important, because they can have crucial influence on attainability of high quality models (see the discussion at the beginning of the chapter).

A general view of the functionality of *general meta-learning* (GML) by search with validation and feedback is presented in Fig. 6.3 in the form of a flowchart. It is also presented formally as algorithm 6.1, since this form has been chosen for most procedures discussed in the book.

The algorithm is called a scheme, because it shows the topmost shape of many possible algorithms and does not perform any learning itself, but prepares the workspace for proper meta-learner given as a parameter to do the meta-learning parts of the job. Many existing meta-learning algorithms fulfill this scheme, including bagging, boosting, the algorithm based on machine configuration generators and complexity analysis, presented in Sect. 6.3 and the profile-based meta-learning algorithm of Sect. 6.4.

Fundamental common idea of the algorithms fitting this scheme is that they iteratively run learning machine and adapt the process to results obtained from each learning. As a result, the general algorithm performs a single loop, in which learning/testing tasks are run and when any task is finished, it is analyzed and new tasks are created with respect to the conclusions drawn from the analysis.

Apart from a time deadline, the algorithm has two parameters, responsible for tasks generation and analysis (the meta-learner) and for task running. Actually, these are two modules that determine the whole system possibilities and efficiency.

There are three ways to get out of the loop:

- when the proper meta-learner decides that the process should stop,
- when all candidate tasks are finished and there is nothing more to do in the process,
- when the time for calculation is passed and the algorithm must finish with what it has managed to gain so far.

All three loop exits are clearly seen in both flowchart and meta-code: in the flowchart, the conditions are represented by the three decision boxes, while in the meta-code, the first exit is encoded in the loop condition and the two others are the *break* calls.

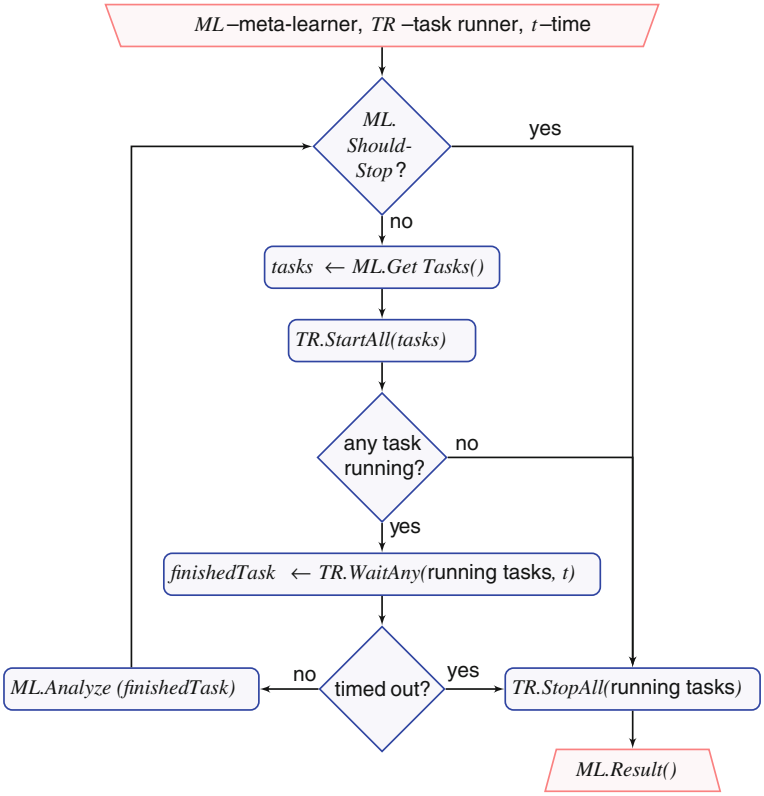


Fig. 6.3 General meta-learning algorithm scheme

In each iteration of the loop, the meta-learner can provide any number of tasks to be executed. For example, bagging can generate all its submachines in the first cycle, as the subsequent machines are not dependent on the results of their sibling machines, generated earlier. On the other hand, most approaches to boosting need to generate submachines one by one and wait until each process ends before the next machine is determined.

After starting all the tasks scheduled by the meta-learner, the GML algorithm starts waiting until a task is finished. Then, immediately, the proper meta-learner is informed about the task which is finished and can analyze the results for the purpose of adequate generation of further tests.

When one of the three stop conditions gets satisfied and the loop is broken, all running tasks (if any) are killed and the meta-learner is given an opportunity to conclude its learning and return the final model (with the *Result()* method), which becomes the result of the whole procedure.

Algorithm 6.1 (General meta-learning scheme)

Prototype: ***GML(ML,TR,t)***
Input: *Proper meta-learner (ML), task running service (TR), a time deadline (t).*
Output: *Winner machine configuration and model.*
The algorithm:

1. *runningTasks* $\leftarrow \emptyset$
2. ***while*** $\neg$*ML.ShouldStop* ***do***
   a. *Tasks* $\leftarrow$ *ML.GetTasks()*
   b. *TR.StartAll(Tasks)*
   c. *runningTasks.Add(Tasks)*
   d. ***if*** *runningTasks* $= \emptyset$ ***then break*** */* nothing more to do */*
   e. *finishedTask* $\leftarrow$ *WaitAny(runningTasks, t)*
   f. ***if*** *finishedTask* $= \perp$ ***then break*** */* out of time */*
   g. *runningTasks.Remove(finishedTask)*
   h. *ML.Analyze(finishedTask)*
3. *TR.StopAll(runningTasks)*
4. ***return*** *ML.Result()*

6.2.2 Proper Meta-Learners

The flowchart in Fig. 6.3 and the Algorithm 6.1 allow to realize the importance and the interface of proper meta-learners which give shape to the GML algorithm. Interface 6.2 is a summary of the functionality.

Interface 6.2 (Proper meta-learner)

Method: ***GetTasks()***
Input: *None.*
Output: *Collection of machine configurations to be run.*

Method: ***Analyze(finishedTask)***
Input: *Reference to a task just finished.*
Output: *None.*

Method: ***Result()***
Input: *None.*
Output: *An adequate form of results (up to particular instance).*

Property: ***ShouldStop***
Output: *Informs that the process is finished.*

The method *GetTasks()* returns the most important information, deciding about the crucial parts of the meta-learning process, that is, the collections of machine

configurations to be run. Within this method, the meta-learner decides what tests should be performed.

On the other end of each subprocess life, the method *Analyze()* is called to inform the meta-learner about just-finished submachine run. This method is called for each task separately and immediately after the task is finished. The meta-learner should analyze the results obtained by the task and adjust its internal knowledge to this information. For example, it may save information about best machines, build meta-knowledge about influence of some parameter values on the results or about different machines cooperation. Such knowledge may be very useful in improving further meta-learning.

At the beginning of each iteration (or from another point of view: instantly after each call to *Analyze()*), the meta-learner is queried if the process should be stopped or continued. When the main goal of the search is to find a satisfactory solution, it may be undesirable to continue the search after a task satisfied the criteria. To minimize interaction between the meta-learner and task running system, the meta-learner does not need to request for cancellation of all its subtasks—it suffices that the *ShouldStop* property returns *true*, and the main loop gets broken and the remaining tasks are aborted by the GML .

Although the familiarity of meta-learners with task running subsystem is not required, some cooperation may be useful. For example, a meta-learner, after scheduling a set of tasks for running, may find out from results of some tasks, that some others are very improbable to give interesting results and would like to cancel some subtasks without finishing the whole process. Then, it may be very useful that meta-learners can cancel their subtasks. Actually the same functionality is advantageous also for other machines which generate submachines, not necessarily the meta-learning ones.

When the main loop of the GML algorithm is finished, the proper meta-learner prepares and returns the final result (a call to the *Result()* method). It may be the configuration of the winner learning machine, a ranking of learning machines (ordered by a degree of goal satisfaction), comments on chosen learning machines and their interactions, and so on. It is up to particular implementations to decide what is regarded as the result of the ML process and how it is represented.

As mentioned above, a bagging machine may be realized as an instance of the GML algorithm, though it does not perform any learning, but just creates children and combines their decisions. All the submachines can be requested in the first call to *GetTasks()* and *Analyze()* can be empty in this case. The task of *Result()* in bagging, is to prepare the final ensemble and return its functionality.

Implementation of boosting must be slightly different. First of all, it should request (with *GetTasks()*) its children one by one, because each next submachine depends on the results from the previous ones. During the subtask results analysis (the *Analyze()* method), it should calculate proper weights for the next training data sample, and in fact, determine the configuration to be returned by the next call to *GetTasks()*. Finally, the call to *Result()* should prepare the final decision module of the ensemble and return the boosted model.

More advanced meta-learners can be defined in many different ways. This framework has facilitated creating such algorithms as the one based on configuration

generators and complexity control (see Sect. 6.3), and profile-based meta-learning algorithm described in Sect. 6.4. Also, the ranking validation algorithm presented in Sect. 6.4.3 has been built according to this general framework.

Validation Scenarios

ML search processes usually aim at finding the most successful learning machine of some pre-specified kind, for example, the best classifier or approximator. The tasks returned by meta-learners in subsequent iterations of GML algorithm should perform sensible tests of the configurations being the proper subject of the search. The same circumstances have been observed for the MPS machine presented in Sect. 4.4.1). In both cases, a natural solution is to generate configurations of learning machines of interest and embed them in the same validation scenario. For such purposes, Intemi system provides its templates (see Sect. 4.1.2). An example of a template eligible for testing classification models is given in Fig. 6.4. It is a template representing the same scenario as the one analyzed in the context of MPS experiment discussed in Sect. 4.4.1. The only difference is that here, it is a template—it contains the Classifier placeholder, which in the case of MPS was substituted by an SVM configuration.

In the process of meta-learning, the Classifier template scheme can be replaced by (or filled with) any configuration of a machine providing *Classifier* output. Meta-learners can just generate miscellaneous classifiers and put them into the template to obtain a feasible test configuration that can be requested from the task-running system.

The scenario illustrated in Fig. 6.4 performs a number of repetitions (a parameter of the Repeater machine) of n-fold cross-validation tests (n is a parameter of the CV distributor), where in each fold, the training data is standardized (by the Standardization machine), the test (validation) data transformed according to the training

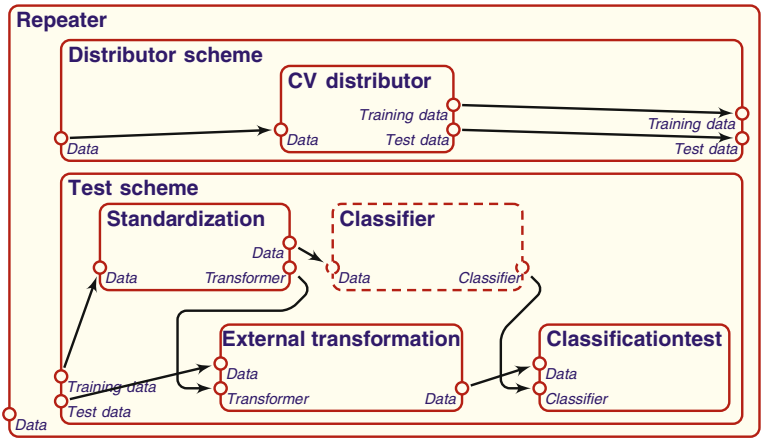


Fig. 6.4 Example validation scenario for classifiers

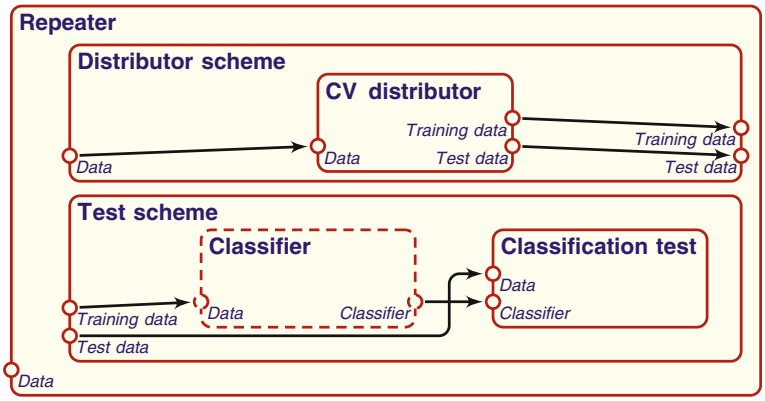


Fig. 6.5 Typical validation scenario for classifiers

data standardization parameters (with the External transformation machine), the classifier is trained and its accuracy on the transformed test data is checked (with the Classification test machine).

Classifiers replacing the Classifier scheme do not need to be simple machines. They can be any complex hierarchies of machines, from which a compatible classifier output is exhibited through the Classifier output of the scheme. In particular, the classifier can be composed of a data transformation machine and proper classifier (for example, in the form of the Transform and classify machine presented in Sect. 4.1.3), performing the same function as the composition of Standardization , Classifier and External transformation machines from Fig. 6.4.

Obligatory data standardization may sometimes be harmful, so in most applications, raw training and test scenario, presented in Fig. 6.5 is more adequate.

Using the same test template for each configuration proposed within the search, is very advantageous from the point of view of reliable results comparison. It tests each machine in the same environment for exactly the same training and test data.

6.2.3 Task Requests and Task Running

The task runner interface used in the GML algorithm, is quite simple. It is summarized as interface 6.3. Naturally, it is just the interface to a complex task-running system implementing a solution to the nontrivial problem of efficient task running. The solution of Intemi is sketched in Sect. 4.2.3.

It is important to realize that when *StartAll()* is called, the task runner is not obliged to start processes immediately, but just to schedule the tasks for running in appropriate time. Optimization of task running is an internal matter of the task management module and should not require meta-learners to know its assumptions and specificity.

Interface 6.3 (Task runner)

Method: StartAll(tasks)
Input: *Collection of machine configurations to run.*
Output: *None.*

Method: WaitAny(tasks, t)
Input: *Collection of tasks to monitor (tasks), maximum time to wait.*
Output: *One of the tasks, that have finished.*

Method: StopAll(tasks)
Input: *Collection of machine configurations to cancel.*
Output: *None.*

Waiting until a requested task ends it process is realized by the *WaitAny()* method. It must be provided with a collection of tasks to be monitored and the time deadline for waiting. When a task gets finished before the time elapses, it is returned as the result. If the time elapses before any task is finished, it returns null, so that it is easy to recognize whether any task has finished or the time deadline has come.

Additional time constraints may also be contained within a task. For example, the meta-learning algorithm using complexity control, described in Sect. 6.3, can assign a maximum run time for a task, so that it is broken when it consumes more time than assigned. It is possible to assign an amount of CPU time, an absolute time deadline, or both at the same time or any other time-related constraint, compatible with the task manager. Thanks to respecting the limits, long-lasting processes do not lock computing resources. Such constraints and related services are not visible in the GML, because they are internal to the task management system.

A disadvantage of machines designed to use time constraints is that unification of such machines is impossible due to their nondeterminism. Even consumed CPU time measurements can give different results for the same process run twice, so it is impossible to predict if two runs of such machine with the same parameters end up with the same model or not. What is more, it is never known whether a machine process will finish in the specified time or will be aborted, even if another instance with the same parameters has already been run and the results are known.

Time limits, assigned to the tasks, cause that the task returned by *WaitAny()* may be finished abnormally, due to such time limit, so it is not guaranteed that the model of returned task is ready. Nevertheless, meta-learners imposing time limits, can draw important, valuable conclusions also from such interrupted machine runs.

The *StopAll()* method is called when the GML algorithm is about to exit and needs to clean up the remainders of its activity. The tasks contained within the collection given to the method as parameter, are not necessarily running—they can still be waiting in the spooler. The task runner should handle them correctly regardless of whether they have already been started or not.

6.3 Meta-Learning with Configuration Generators and Complexity Control

An example of meta-learning algorithm working in a manner compatible with the general meta-learning algorithm scheme is the meta-search based on *machine configuration generators* and *complexity control*, published recently (Jankowski and Grąbczewski 2011). Its general idea is exactly the same as that of GML framework discussed above: candidate machine configurations are generated, embedded in a validation scenario, run and estimated by means of multiple cross-validation test analysis.

Apart from the common assumptions, the goal of this approach is to test as many, most promising machine configurations, as possible in given time t. Although the main criterion of model selection is based on an estimate of classification accuracy (so far, the algorithm has been tested on classification problems only), other very important premises are model size and time necessary for learning models and testing them.

Because the key idea of the method is complexity based control of the search process, the algorithm can be given the name of Complexity-Driven Meta-Learning (CDML) algorithm.

The second important idea of the CDML approach, is that machine configuration generators can produce arbitrarily complex machine structures, so that not only simple learning machines are tested but also such machines as compositions of (several) transformations and classifiers (or other final decision making machines), committees of different types of machines (including complex ones in the role of members, for example, compositions of data transformers and classifiers).

6.3.1 CDML as an Instance of GML

Implementation of the CDML algorithm has been done within Intemi framework, so the matters of task management have been solved by the engine of the system. The only component of the approach that needs to be specified is the proper meta-learning module of the GML algorithm. Its implementation of the interface 6.2 is sketched below.

GetTasks() The idea of constructing machine tests by embedding learning machines in validation scenarios allows to focus just on proper learning machines here, though remembering about the embedding. Anyways, two main aspects of providing subsequent test tasks have the most important influence on the performance of CDML algorithm:

- formulation of the search space of learning machine configurations—the definition of the set of learning machines that are regarded for testing,

- definition of the order in which the test tasks are run—since normally time limits do not allow to perform all the tests, the order strongly affects the availability of attractive results.

For the purpose of the CDML approach, a family of machine configuration generators has been created. Their configuration determines the space of machine configurations regarded by the algorithm. More detailed discussion on generators follows in Sect. 6.3.2. Task complexity issues, that determine the order of task submission are discussed in Sect. 6.3.3

Analyze() Notifications about finished tasks may trigger analysis of several aspects. First of all, because of complexity control, the tasks may finish normally (with a model generated) or because of timeout. One of the aspects is to analyze the execution times and rerun aborted tasks when appropriate. Other aspects analyze the model accuracy to rank tested methods, to adjust further treatment of the configurations supplied by generators, to modify operation of generators and so on.

ShouldStop CDML algorithm does not use any other stop criterion than just the time available for calculations. Therefore its *ShouldStop* property is always false. In practice, the timeout is usually the only stop criterion, because finishing because of testing all candidate configurations is possible only when the number of possible configurations is so limited that can be called degenerate. The main idea of CDML is to construct machine configuration generators in such way that many more configurations are generated than can be tested, and complexity control helps in performing tests in a sensible order.

Result() Depending on the needs, the final result of the CDML algorithm may be a single best machine configuration determined by the process or a ranking of some number of the best configurations. Naturally, the winner configurations can be easily converted into models and CDML may exhibit the best models as its own outputs to play the role of the model built with the best configuration found.

6.3.2 Machine Configuration Generators

In the CDML algorithm, the search space is defined in the functional form of *generators flow*. It is a special type of directed graph, where *machine configuration generators* (MCGs) are vertices and the edges are used to send machine configurations provided by one generator to another. For a quick grasp of the mechanism, it is recommended to start with a look at Fig. 6.6, where a simple example of generators flow is presented. A generator may have a number of inputs and must provide a single output. The ports serve as transfer media for configuration collections. Each MCG generates collections of machine configurations and exhibits them through the output to other MCGs or to the *flow output*, which is the final port providing the overall collection of generated configurations. Because MCGs may have many inputs, each

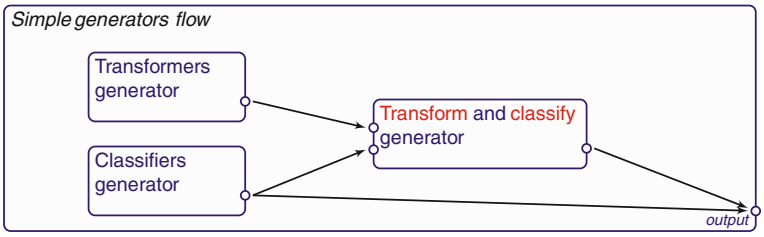


Fig. 6.6 Example of a simple generators flow

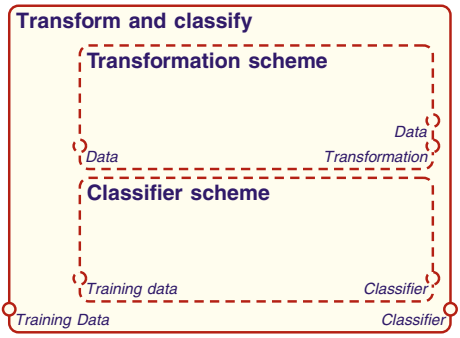


Fig. 6.7 Configuration template of transform-and-classify machines

connection between MCGs is labeled with the input name it connects to. The simple generators flow presented in Fig. 6.6 has three MCGs inside, two of which are connected to the flow output. It means that the collection output by the flow is the union of collections obtained from Classifiers generator and Transform and classify generator. Before the latter generates its output, it is given the configurations output by the two other generators.

Each MCG may act in its own way, may base on different meta-knowledge and may change its behavior in time (during the progress of meta-learning). The generators discussed here may be divided into two groups:

- *set-based generators*,
- *template-based generators*.

Two generators of the flow in Fig. 6.6 (Transformers generator and Classifiers generator) belong to the former group and one (Transform and classify generator) to the latter.

Set-based generators have no inputs and just generate a collection of pre-defined configurations. In fact, they just exhibit the sets of configurations they are given a priori. In the example of Fig. 6.6, one of the set-based generators outputs a collection of machine configurations for data transformation and the other for classification.

Transform and classify generator is a *template-based generator*. It means that it is parameterized by a template machine configuration with placeholder(s) for other machines configurations. The transform-and-classify machine has been described

in Sect. 4.1.3. Its raw configuration is actually a template presented in Fig. 6.7. It contains two placeholders for subconfigurations, so the generator based on this template has two inputs—each input provides configurations to be put in adequate placeholder. Therefore, the name of the generator, shown in Fig. 6.6, has two words printed in red—they inform about the placeholders contained within the generator's template. The role of template based generators is to produce configurations from their templates by replacing the placeholders with configurations obtained through corresponding inputs. When filling the placeholders, template-based generators may use different strategies to combine the input collections. The most natural methods to combine two lists into a list of pairs are:

- sequential pairs construction (the lists must be of equal size, and elements at the same positions are joined in pairs),
- combinatorial pairs construction (each element of the first list is combined with each element of the second list).

For full clarity, let's consider the simple generators flow with 5 data transformations and 7 classifier configurations generated by the set-based generators and the template-based generator configured to combine the inputs combinatorially. Then, the flow outputs 42 configurations: 7 classifier configurations from the Classifiers generator (passed directly to the generator flow), and $5 \times 7 = 35$ transform-and-classify configurations from the template based generator.

So simple generator flows are good illustrations, but are not very useful in practice. In serious search for models solving a real world problem, it is necessary to use much more complex flows. An example of a flow that can be useful in some real meta-learning problems is presented in Fig. 6.8. It contains all the generators of the simple flow (with the same connections) and also several other generators. Additional set-based generator (Rankings) provides a collection of configurations of feature ranking machines to the template-based generator named Feature selection from

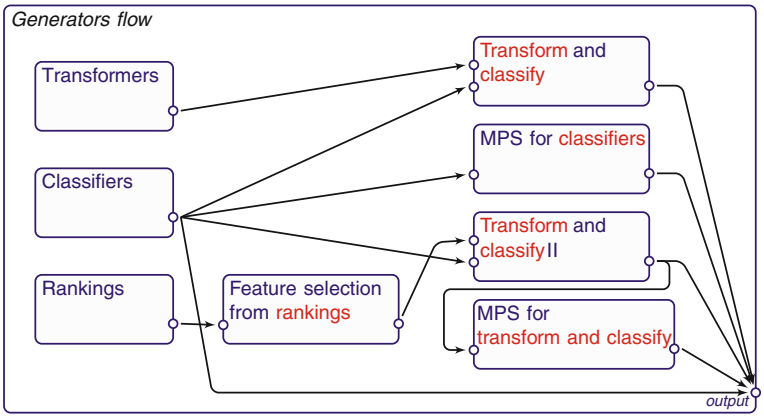


Fig. 6.8 Example of a practical generators flow

rankings using the template presented in Fig. 4.5. The template contains just one placeholder, so normally, it generates a collection of configurations containing the same number of elements as the collection it gets as the input, but later, at the corresponding MPS generator, more configurations may be generated because of the possibility to select different numbers of top-ranked features from each ranking.

The meta-parameter search machine is described in Sect. 4.4.1. It can optimize parameters of any learning machine (either simple or complex) basing on declarations of the suggested optimization strategies for given learning machines. It means that the MPS for transform and classify generator can manipulate all parameters of machines occurring in the input configurations, namely both feature-ranking machines and feature-selection machine (the configuration of transform-and-classify machine can not be changed by MPS because it has no parameters). In its name, the whole transform-and-classify part is printed in red in the figure, because the whole name refers to a placeholder, filled with this type of machines, not two placeholders as in the previous example.

The Transform and classify II generator combines feature selection methods with classifiers in the same way as its twin generator does with transformation methods obtained from another source.

The output of the whole graph collects configurations from 5 generators, so that the final collection contains:

- pure classifiers from a set-based generator,
- classifiers combined with transformations,
- classifiers with parameters manipulated with MPS,
- classifiers combined with feature selection based on rankings,
- MPS-manipulated configurations of classifiers combined with feature selection based on rankings.

Provided that the set-based generators output rich sets of configurations, such graph of generators can bring a variety of configurations (see Sect. 6.3.7 for the generators flow in action). Slight and simple changes in settings of the generators, facilitate flexible control of the search space definition resulting in powerful meta-learning algorithms.

Generator flows may consist of any number and any kind of generators. There are no a priori limits on the number of connections between the generators, either. Often, the number of configurations available from a generators flow grows exponentially with the number of connections between generators. Sometimes, it is even preferable to separate groups of configurations (like classifiers or data transformations) into independent set-based generators to simplify the connection paths in the generators flow graph, and to avoid "strange" combinations of machine configurations that are very unlikely to discover valuable knowledge. Grouping data transformations may provide more flexibility, for example, when some machines have to be preceded by a transformation to discretize data or should be used with (or without) proper filter transformations. The possibilities introduced by generators flow, are very large and can bring much profits to meta-learning.

6.3.3 Complexity Control

Not all configurations coming out from a generator flow are instantly submitted to task runner. The most important aspect of the Complexity-Driven Meta-Learningz algorithm is the control over the order in which candidate configurations are tested. CDML keeps the control by means of complexity estimation: each machine configuration is subject to multi-aspect analysis aimed at estimation of its overall complexity. The configurations of the lowest estimated complexity are transformed into tasks and run first.

There is no single, perfect complexity measure that could be accepted by everybody as the best solution, however there are some unarguable values that should be taken into account when creating such measure:

- time consumed by machine process,
- model simplicity,
- predicted attractiveness of the model.

Only the first item is quite precise, the other two are ambiguous, general terms that can be formalized in many different ways. Since the complexity measurement should be done before a machine is run (provided just its configuration and uncertain information about input data), also the aspect of time consumption is undecidable. When combination of the three aspects comes into play, new ambiguous factors are introduced, so prediction of the most promising configuration is extremely difficult.

One of the easily accepted ideas of meta-learning algorithms is that simple solutions should be favored and given precedence over more complex ones. Therefore, it might seem that ML algorithms based on search and validation should start with single learning machines, then test simple compositions of machines (for example, a single transformation and a classifier), and then proceed to more and more complex structures of learning machines (complex committees, multiple transformations and so on).

Unfortunately, the problem is not that simple and the structural complexity of machines does not correspond to real time and space complexity of the tasks. Sometimes, a composition of a transformation and a classifier may be indeed of smaller complexity than the classifier without transformation. When using a transformation, the data passed to the learning process of the classifier may be of lower complexity and, as a consequence, classifier's learning is simpler and the difference between the classifier learning complexities with and without transformation may be larger than the cost of the transformation.

Moreover, even when time complexities of two algorithms are known, it can still be impossible to point one of them as less time consuming. For example, consider two learning machines M_1 and M_2 of computational complexities $O(nf^2)$ and $O(n^2 f)$ respectively, where n is the number of vectors in the training data and f is the number of features describing objects (the training dataset is assumed to have a form of data table). In such case, it is not possible to compare time consumption of M_1 and M_2 until the final values n and f are known. If the machines occur inside a more complex

structure, the crucial information may be available not earlier than after part of the structure is created and run. The CDML algorithm must cope with the complexities before the whole task is started, so it must find some ways to predict such values like n and f or directly the complexity estimate.

Fortunately, very rough estimation and additional mechanisms preventing single machines from blocking the CPU with "never-ending" calculations can be quite satisfactory. Some particular measures are described in Sect. 6.3.5. Here, the discussion focuses on the methods of using the estimates to control the meta-level search process.

Generator flows provide machine configurations, which are embedded in validation scenarios to facilitate

- estimation of machine quality in a reliable test,
- analyze both training and testing procedures (some machines may learn very fast or need no learning at all, and have very large CPU time requirements at the stage of testing, for example, most nearest-neighbors algorithms).

The test procedures should be as similar to the whole life cycle of a machine as possible (and of course as trustful as possible). The whole validation tasks are passed to the task runner in appropriate order, respecting the estimate of task complexity.

During meta-learning, new information may arrive from the analysis of finished tasks. It can bring new knowledge, which affects the order of other tasks. Therefore, MCGs are given opportunity to learn from experience—they receive feedback from the tests and can modify the way they produce new configurations.

Problems of Complexity Estimation for Complex Machines

To understand the needs and problems of complexity computing, we need to keep in mind the specifics of machines and their learning processes.To provide a learning machine, regardless of whether it is a simple machine or a complex structure, whether it is an important, target machine or a machine constructed to help in the process of analysis of other machines, its configuration and inputs must be specified (machine process is a function mapping the inputs to the outputs with respect to given parameters). An input is *specified*, when it is bound to an output of another machine (*resolved* in the vocabulary of Intemi and Chap. 4). Complexity computation must reflect the information from machine configuration and inputs. The recursive nature of configurations, together with input–output connections, may compose quite complex information flow. Sometimes, the inputs of submachines become known just before they are started, that is, after learning of some other machines (providing necessary outputs) is finished.This is one of the most important reasons why determination of complexity, in contrary to actual learning processes, must base on *meta-inputs*, not on exact inputs (which often remain unknown). Assume a simple scene, in which a classifier TC is built from two parts: a data transformer T and a classifier C (for example, a transform-and-classify machine). It is impossible to compute complexity of the classifier C basing on its inputs, because one of the inputs is taken from the output of the transformer T, which is not known before the learning process of T is

finished. Computing complexity of the TC machine may not be limited to a part of the machine or wait until some machines are ready.

To make complexity computation possible in any case, the concept of *meta-inputs* has been introduced. Meta-inputs are counterparts of inputs in the "meta-world". They contain (as informative as possible) descriptions of inputs, which "explain" or "comment" every useful aspect of each input that can be helpful in determination of complexity of various procedures dealing with the input. Because machine inputs are outputs of other machines, the space of meta-inputs and the space of meta-outputs are the same.

For complex machine configurations, complexity is determined recursively. This is natural, because of the recurrent nature of the definition of machine configuration and the recurrent structure of real machines. Therefore, the functions which compute complexity must also provide meta-outputs, because they are crucial in complexity computation for machines which read the outputs through their inputs.

Respecting all the specifics of machines and their processes, a function computing the complexity for machine class $\mathscr{L}$ should be a transformation

$$\mathscr{D}_{\mathscr{L}} : \mathscr{K}_{\mathscr{L}} \times \mathscr{M}_{+} \to R^2 \times \mathscr{M}_{+}, \tag{6.6}$$

where the domain is composed by the configuration space $\mathscr{K}_{\mathscr{L}}$ and the space of meta-inputs $\mathscr{M}_{+}$, and the results are: time complexity , memory complexity and appropriate meta-outputs.

Although the type of the function in Eq. (6.6) is quite simple, the problem of finding a good mapping is not easy. Influence of some configuration elements and of some inputs (meta-inputs) to the machine complexity is sometimes unpredictable, so in general, there is no hope for finding real dependencies and describing them as a function. Configuration elements are not always as simple as single scalar values. Some configuration elements may be represented by functions or submachine configurations. Similar problem concerns meta-inputs, because they may have quite complex functional form. Often, they need their own complexity estimation tool to reflect their features. For example, a committee of machines, which plays a role of a classifier, uses other classifiers as "slave" machines. It means that the committee will use classifiers' outputs, and the complexity of using the outputs depends on members' outputs, not on the committee itself. This shows that the behavior of meta-inputs/outputs is not trivial and proper complexity determination requires multi-level encapsulation.

6.3.4 Analysis of Finished Tasks and the Quarantine

The GML algorithm of Sect. 6.2.1 assumes that a task comes from a call to *ML.GetTasks()* method of the proper meta-learner, then goes to task runner, and is analyzed with *ML.Analyze()* when the process is finished. Slightly more detailed flow, showing the stages, a task goes through in the CDML algorithm, is presented in

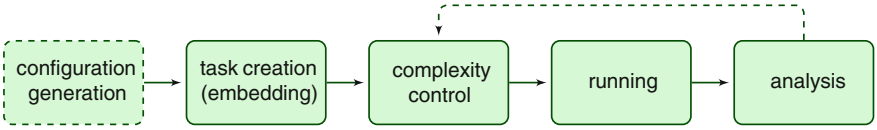


Fig. 6.9 CDML task life stages

Fig. 6.9. Task life starts, when a machine configuration (coming out from generators flow) is embedded in a validation scenario. Then, it undergoes the complexity control procedure (complexity estimation and putting the task in the task spooler for running). From there, the tasks go out to the GML algorithm with a call to *ML.GetTasks()* (after all tasks with more attractive complexity estimates have already been passed to the GML in the same way). Then, they are run and, after the process is finished, the results are analyzed.

The analysis is not necessarily the last stage of such task life. Because the complexity is only an approximation, the meta-learning algorithm must be ready for situations of inaccurate approximations or even of the test tasks that are not going to finish (for example, due to some problems with convergence of learning).

To bypass the halting problem and the problem of inaccurate approximations, complexity control is still active during the process run, and each task is assigned a time limit for running:

$$task.timeLimit \leftarrow \tau \cdot task.cmplx.time/task.cmplx.r \tag{6.7}$$

In this assignment, *task.cmplx.time* is the predicted time consumption of the machine process, *task.cmplx.r* is a task reliability factor (at start set to 1 for all tasks) and τ is a factor of forbearance shown to the tasks when they consume more time than estimated (suggested default value of 2 lets the tasks run twice as long as estimated, to prevent breaking the tasks too early and too often).

The time is calculated in universal seconds, to make time measurements independent of the computer that computes the task. The time in universal seconds is obtained by multiplication of the real CPU time by a factor reflecting the comparison of the CPU power with a reference computer. It is especially important when a cluster of different computers is used for computations or when learning processes of complexity estimators are run on another machine than the meta-learning project.

When the task gets timed out, it is stopped and, although the machine process is formally finished, the machine is not ready. During the analysis process it must be checked, whether the model is ready or not. If not, then the task goes back to the complexity control module, where it is *quarantined*. The quarantine is realized by reassignment of complexity estimation to the task and putting it back in the queue. The position in the queue reflects an update of complexity estimate by a pre-defined factor (default of 4) and the task can still get its chance, when all less complex tasks are finished (or aborted and also delegated to the quarantine). Technically, increasing the complexity estimate is performed by dividing the quality factor *cmplx.r* assigned

to the task. It keeps the information about originally estimated complexity and adds information about the broken run, encoded as *cmplx.r*. When the task is given the next try, it will get significantly more time for running, according to Eq. (6.7).

The task analysis procedure of the CDML algorithm is presented as Algorithm 6.4.

Algorithm 6.4 (Task run analysis in complexity-driven meta-learning)

Prototype: CDML.Analyze(task)
Input: *Task, that has just finished its run (task).*
The algorithm:

1. ***if*** *task was broken due to timeout* ***then***
   a. *task.cmplx.r* ← *task.cmplx.r / 4*
   b. *Pass task to complexity control module for quarantine*
2. ***else***
   a. *quality* ← *quality calculated from the test*
   b. *Add pair* $(quality, task.machineConfig)$ *to the final ranking*
   c. *Send feedback about quality to the machine generators flow*
   d. ***if*** *attractiveness module in use* ***then***
      analyze quality attractiveness

Depending on whether the task has been successfully completed or broken due to a timeout, different steps are undertaken. As described above, if the task has been aborted, the reliability of the task is updated and the task goes back to the complexity control module to wait for the next chance to be run.

When the task is finished normally, the quality of underlying machine configuration is determined on the basis of the test task results. Usually, the quality is calculated by means of a query and series transformations (see Sect. 4.3).The information is added to the machine ranking, and feedback sent to the flow of machine generators and to the attractiveness module (if used) to improve their functions in subsequent iterations. The attractiveness modules are tools for yet another control on the order in which the test tasks are submitted for running. They may learn and organize meta-knowledge, for example, to avoid running many similar tasks of low complexity, but not providing as attractive results as expected, to care for diversity of machine configurations tested and so on. For more information on attractiveness see Sect. 6.3.5.2 and especially Eq. (6.16).

Costs of the Quarantine

It has been already mentioned, that when a task is aborted because of exceeding the time-limit (assigned with respect to the complexity approximation), the task is moved to the *quarantine* for a period not counted in time directly, but determined by the complexities.Instead of constructing a separate structure responsible for the functionality of a quarantine, the quarantine is realized by two naturally cooperating elements: the ordered queue of test tasks and the reliability term of the complexity

formula [see Eq. (6.16)]. First, the reliability of the test task is updated, and then, the task is sent back to the complexity control module.

The role of the quarantine is very important and the costs of using the quarantine are, fortunately, not too large. The CDML algorithm restarts only those test tasks, for which the complexity was badly approximated. To better see the costs, assume that a test task was completely badly approximated and visited quarantine maximum number of times. If the real universal time used by this task is t, then, in the above scheme of the quarantine, the ML algorithm may spend for this task, a time not greater than

$$t + t + \frac{1}{4}t + \frac{1}{16}t + \cdots = \frac{7}{3}t. \tag{6.8}$$

Thus, the maximum overhead is $\frac{4}{3}t$, however it is the worst case—the case where we halt the process just before it would be finished (hence the two whole t's in the sum). The best case gives only $\frac{1}{3}t$ overhead which is insignificant. The overhead is not a serious hamper, especially, when we take to the account that the CDML with the quarantine is not affected by the halting-problem of test-task.

Moreover, the cost estimation is pessimistic also from another point of view: when a task is run again, all its calculations are not necessarily repeated. Thanks to the modular structure of Intemi machines and to the unification engine embedded in the system (see Sect. 4.2.2), cooperating with the specialized *machine cache*, the submachines that have been successfully created during the previous run(s) can be reused—do not need to be recalculated. Under such circumstances, when the cache is large enough to maintain the finished submachines of the aborted task, the next run of the same task can be seen as continued from the point it was aborted at, so the recalculation overhead is rudimentary.

6.3.5 Machine Complexity Evaluation

To obtain the right order in the ML queue of test tasks, a complexity measure applicable to machine structures of any structural complexity, should be used. It reveals a natural analogy with the algorithmic information theory, where object complexity is measured on the basis of the amount of computational resources necessary to specify the object (for example, to print some output). In meta-learning, we search for models, which can be analyzed in the same manner, because machines are programs that need some CPU time and memory to generate their models.

6.3.5.1 Complexity Measures by Kolmogorov and Levin

Kolmogorov complexity (Kolmogorov 1965; Li and Vitányi 1993) is very well known and has been analyzed from theoretical point of view. It measures the complexity of a given output O as

$$C_K(O) = \min_p \{l_p \ : \text{program } p \text{ prints } O\}, \tag{6.9}$$

where l_p is the length of program p. Unfortunately, this definition is inadequate for the CDML approach, because the program p may work for unacceptably long time. The Kolmogorov complexity is not useful in real tasks (in particular in computational intelligence problems), also because the problem of finding a minimal program is undecidable—the search space of programs is unlimited and the time of program execution is unlimited. Levin's complexity definition (Li and Vitányi 1993) introduced a term responsible for time consumption:

$$C_L(O) = \min_p \{C_L(p) \ : \text{program } p \text{ prints } O \text{ in time } t_p\} \tag{6.10}$$

where

$$C_L(p) = l_p + \log t_p. \tag{6.11}$$

The Levin's definition facilitates finding the complexities with the Levin Universal Search (LUS, Jankowski 1995; Li and Vitányi 1993), but the problem is that this algorithm is NP-hard. This means that, in practice, it is impossible to find an exact solution to the optimization problem. Anyways, the strategy of meta-learning is different than the one of LUS. The goal of ML is not to find the optimal program, but as good program as possible in some limited time, so it is acceptable to apply some algorithms providing approximate solutions in reasonable time.

It is also important to notice that the CDML complexity must be computed for machine configurations and descriptions of the inputs, not for ready learning machines, because the information about complexity is needed before the machine is ready. In most cases, there is no direct analytical way of computing the machine complexity on the basis of its configuration. Therefore, an approximation framework for automated complexity estimation has been introduced (Jankowski and Grąbczewski 2011).

6.3.5.2 Complexity in the Context of Machines

An assumption of the CDML approach is that machine generators flows output large numbers of machine configurations to be tested. The tests need to be performed in the most advantageous order because of a time limit and impossibility to perform all the tests in time (in most real applications just a little fraction of all the calculations can be done).

Given a learning problem $\mathscr{P}$ with time limit t and a machine generators flow that generates learning machine configurations $m_1, m_2, \ldots, m_q$, assume that their testing times are $t_1, t_2, \ldots, t_q$ respectively. When not all tests can be done within the time limit ($\sum_{i=1}^{q} t_i > t$) and if we assume that no machine configuration is preferred to another (each one is equally promising), then to maximize the expected value of the best test result obtained within the time, we need to run as many tests as possible.

Therefore, the optimal order of the tests is the order $i_1, i_2, \ldots, i_q$ of nondecreasing testing times:

$$t_{i_1} \leq t_{i_2} \leq \ldots \leq t_{i_q}, \tag{6.12}$$

and the choice of first m shortest tests such that

$$m = \max \left\{ k \in \{1, \ldots, q\} : t_{i_1} + \cdots + t_{i_k} \leq t \right\}, \tag{6.13}$$

is the optimal selection of tests to be done in time t. The proof of this property is trivial.

This property clearly confirms that the CDML approach is the most reasonable solution to the problem. We can claim that CDML is optimal if the strategy of estimating complexities is accurate.

Even if we do not assume that learning machines are equally promising, but some meta-knowledge allows to estimate the attractiveness of each candidate machine, the order may be accurately determined by proper adjustment of the complexity measure to maximize the expected value of test results (see Eq. (6.16) and comments about it).

The halting problem directly implies that, in general, it is not possible to accurately calculate time and memory usage of a learning process.

Strengthening Time Influence

The property of optimal ordering of tests means that ordering programs (machines) only on the basis of their length [as it was defined in Kolmogorov Eq. (6.9)] is not rational in the context of learning machines. The problem of using Levin's additional term of time in real applications is that it is not rigorous enough in respecting time. For example, a program running 1024 times longer than another one may have just a little bigger complexity (just +10) when compared to the program length.

On the other side, total rejection of the length l_p is also not recommended because always machines have to fit in memory, which is also limited. Besides, the length control is in compliance with the commonly accepted rule of Occam's razor.

In learning machines, typically time complexity exceeds significantly memory complexity. Therefore, a measure of complexity combining time and memory complexity has been proposed:

$$C_a(p) = l_p + \frac{t_p}{\log t_p}. \tag{6.14}$$

Although the time part may seem to be respected less than the memory part, it is not really so, because, as mentioned above, time complexity almost always significantly exceeds memory complexity. Thus, in fact, memory part dominates anyway, so the order of performing tests is very sensitive to time requirements. The length component l_p just "keeps in mind" the limited resources of memory and the whole C_a defines a sensible balance between the two complexities.

Complexity Measure and Quarantine

Naturally, we can use just an approximation of the complexity of a machine, because the actual resource consumption is not known before the real test task is finished. Because of that and because of the halting problem (we never know whether given test task will finish) an additional penalty term is added to the above definition:

$$C_b(p) = \frac{1}{r_p}\left(l_p + \frac{t_p}{\log t_p}\right), \tag{6.15}$$

where r_p^{-1} is a factor reflecting reliability of $C_a(p)$ estimate, modified when it turns out that the estimate is too optimistic.

At first run of a task p, CDML sets $r_p = 1$ (generally $r_p \leq 1$), but when the estimated time is not sufficient to finish the program p, the program p is aborted and the reliability is decreased. The aborted test task is quarantined according to the new value of complexity reflecting the change of the reliability term. This mechanism prevents running test tasks for unpredictably long time (or even infinite time). Otherwise, the meta-learning algorithm would be very brittle and susceptible to running tasks consuming unlimited CPU resources. More details on the mechanism of r_p are presented in Sect. 6.3.4.

Complexity Measure and Machine Attractiveness

Another extension of the complexity measure is possible thanks to the fact that CDML algorithms are able to collect meta-knowledge during learning. The meta-knowledge may be useful in assessment of configuration generators, complexity estimators, machine results quality and so on. It may be used to modify the order of test tasks waiting in the machine heap and machine configurations which will be provided during next iterations of configuration generation. A comfortable way of doing this is adding a new term to the $C_b(p)$ to shift the start time of given test in appropriate direction. The new measure

$$C_m(p) = \frac{1}{r_p \cdot a_p}\left(l_p + \frac{t_p}{\log t_p}\right), \tag{6.16}$$

obtained in this way respects additional a_p factor interpreted as the attractiveness of the test task p.

$C_a(p)$ is a measure aimed at such order of the tasks that maximizes the expected value of results obtained in a limited time, assuming equal a priori probabilities of each task p. Introducing the factor a_p^{-1} is a correction to the initial assumption of equal priors, so on the assumption that the attractiveness factors can be estimated with a reasonable accuracy, the new measure $C_m(p)$ may be much more adequate tool for proper task ordering.

6.3.5.3 Meta-evaluators

Measuring complexity with Eq. (6.16) requires approximation of all values occurring on the right-hand side. r_p is controlled by precise rules, described above and a_p is an optional factor for advanced control of the meta-learning process, so at least l_p and t_p require estimation to make the complexity control practical.

Computing complexity-related values for machine of class $\mathscr{L}$ has been announced as a function $\mathscr{D}_{\mathscr{L}}$, of type defined in Eq. (6.6). It has also been mentioned that precise definition of the function can be difficult or even impossible. Therefore, a framework has been created to learn functions that can provide successful approximations of the values necessary for calculation of complexity measures.

Equation (6.6) defines the domain in terms of meta-outputs describing real machine inputs. Sometimes, it is necessary to estimate complexity on the basis of real inputs, so we would need also another type of machine evaluators:

$$\mathscr{D}'_{\mathscr{L}} \ : \ \mathscr{K}_{\mathscr{L}} \times \mathscr{I}_{+} \to R^2 \times \mathscr{M}_{+}, \tag{6.17}$$

where $\mathscr{I}_{+}$ is the space of inputs for machine class $\mathscr{L}$. To avoid learning evaluators of both forms, we can replace the evaluators based on real outputs by compositions of mappings from real to meta-outputs (output evaluators) and evaluators for meta outputs:

$$\mathscr{D}'_{\mathscr{L}} = (\mathscr{D}_{o_1}, \ldots, \mathscr{D}_{o_n}) \circ \mathscr{D}_{\mathscr{L}}, \qquad (\mathscr{D}_{o_1}, \ldots, \mathscr{D}_{o_n}) : \ \mathscr{I}_{+} \to \mathscr{M}_{+}, \tag{6.18}$$

In this way, instead of learning dual evaluators for each machine, just the one based on meta-inputs can be learned and applied together with output evaluators.

Sometimes, machine complexity depends on nontrivial machine components, which should not be regarded as integral machine parts, but rather external (often exchangable) modules. For example, configurations of machines like kNN or SVM are parameterized by metric. The complexity of metric (the time needed to calculate single distance between two instances) does not depend on the kNN or SVM machine, but on the metric function. Separate evaluators for such nontrivial objects/components simplify creation of machine evaluators. Every evaluator may order creation of any number of such nested evaluators.

Very general framework has been developed for these purposes, because there are many different kinds of machines, machine outputs and miscellaneous components used by machines at learning time or later, when resulting model performs its services. The concept of *meta-evaluators* has been introduced to

- evaluate and exhibit some values representing different aspects of complexity inferred from meta-descriptions like meta-inputs or configuration (for example, amounts of time and memory needed by the machines),
- exhibit functional descriptions of complexity aspects useful for further use by other meta evaluators (for example, meta-outputs are exhibited to be used by evaluators of machines bound to these outputs through their inputs).

Because of the recurrent nature of machines (and machine configurations) and the complex, object-oriented, hierarchical structure of computational intelligence projects realized in Intemi, meta-evaluators are created for different types of entities:

- machines (for time and memory estimates),
- outputs (to describe different aspects of their functionality),
- components used by machines, because of object-oriented, hierarchical design.

Each evaluator needs an *adaptation* process, which can be seen as initialization and can be compared to the learning processes of machines. The adaptation process is the major functionality of each evaluator and depends on the type of the evaluator. Because of this dependence, adaptation of different evaluators may be parameterized with completely different types of values. Some evaluators may need just several numerical values and some others may require functional descriptions of some nested objects. For example:

- if an evaluator is defined for a machine, then the parameters may provide a real machine or a machine configuration and meta-inputs,
- evaluators constructed for outputs, get references to real outputs as parameters,
- in other cases, the parameters depend on the needs of particular evaluators.

Despite the differences, the goal of each adaptation process is to use the data given as parameters, as the "source of information" for building evaluation functions.

When evaluators may be defined analytically (quite rare cases), their preparation requires just the adaptation process (they are called *plain evaluators*). In other cases, the *approximation framework* is used to construct *learnable evaluators*. Figure 6.10 depicts the differences between the two processes. Details about the process of evaluators creation and training approximators to perform proper functions of evaluators are described in (Jankowski and Grąbczewski 2011).

Precise functionality of meta evaluators depends on their types. Some contexts are shortly addressed below.

Machine Evaluator

In the case of any machine evaluator, the additional functionality consists of:

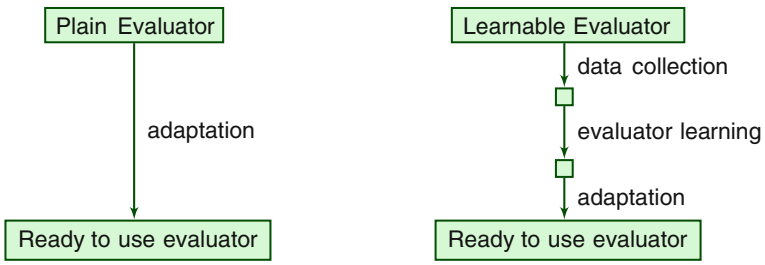


Fig. 6.10 Creation of ready to use evaluator for plain evaluator and evaluator constructed with approximation framework

Declarations of output descriptions: Machines that exhibit outputs, must be accompanied by output evaluators, devoted to the machine type, providing meta-descriptions of the outputs. The descriptions of outputs are meta evaluators of appropriate kind (for example: meta-classifier, meta-transformer, meta-data and so on). Output description may be the machine evaluator itself or a nested evaluator produced by the machine evaluator or the evaluator provided by one of submachine evaluators constructed by the machine evaluator (machine may create submachines, evaluator may create evaluators of submachines basing on their configurations and meta-inputs).

Time and memory: Complexities defined by Eqs. (6.10), (6.14)–(6.16) make use of program length and time (machine size and learning time). The two quantities must be provided by each machine evaluator to enable proper computation of time and memory components of the complexities.

Child evaluators: For advanced analysis of complex machines complexities, it is useful to have access to meta evaluators of submachines. Child evaluators are designed to provide this functionality.

Classifier Evaluator

An evaluator of a classifier output has to provide the time complexity of classification of an instance:

```
real ClassifyCmplx(DataEvaluator dtm);
```

Apart from the learning time of given classifier, the time consumed by the classification routine is also very important in calculation of complexities. To estimate time requirements of a classifier test machine, one needs to estimate time requirements of the calls to the machine classification function. The final time estimation depends on the classifier and on the data being classified. The responsibility to compute the time complexity of the classification function belongs to the meta classifier (the evaluator of the classifier). Consider a classification committee: to classify data, it needs to call a sequence of classifiers to get the classification decisions of the committee members. The complexity of such classification, in most natural way, is a sum of the costs of classification using the sequence of classifiers, plus a (small) overhead which reflects the scrutiny of the committee members results to make the final decision. Again, the time complexity of data classification is crucial to estimate the complexity and must be computable.

Approximator Evaluator

An evaluator of an approximation machine has exactly the same functionality as the one of a classifier, except that approximation time is considered in place of classification time:

```
real ApproximationCmplx(DataEvaluator dtm);
```

Data Transformer Evaluator

An evaluator of a data transformer has to provide two estimation aspects. The first one is similar to the functionality of the evaluators described above. Here, it represents the time complexity of transformation of data instances. The second requirement is to provide a meta-description of data after transformation: a data evaluator. It is very important because the quality of this meta-transformation of data-evaluator affects the quality of further complexity calculations.

Metric Evaluator

The machines that use metrics, usually allow to set the metric at the configuration stage (for example, kNN or SVM). As parameters of machine configurations, metrics have nontrivial influence on the complexity of the machine, while not being separate learning machines. The most reasonable way to enable complexity computation in such cases is to reflect the metric-dependence inside the evaluators (one evaluator per one metric). The meta-evaluators for metrics provide the functionality of time complexity of distance computation and are used by the evaluators of proper machines or outputs:

```
real DistanceTimeCmplx();
```

Data Evaluator

Another examples of evaluators of crucial meaning are data evaluators. Their goal is to provide information about data structure and statistics. Data evaluator has to be as informative as possible, to facilitate accurate complexity determination by other evaluators. In the context of data tables, the data evaluators should provide information like the number of instances, the number of features, descriptions of features (ordered/unordered, the count of symbols used and so on), descriptions of targets, statistical information per feature, statistical information per data and others that may provide useful information to compute complexities of different machines learning from the data.

Other Evaluators

The number of different types of meta evaluators is not determined. Above, only a few examples are mentioned. Many other instances are also available in Intemi. When the system is extended with new types of machines, it should also be extended with appropriate evaluators to facilitate including the new machine types in meta-search processes.

6.3.6 Learning Evaluators

In the examples of evaluators listed above, many kinds of values can be found including time and memory consumption of machine processes, classification related data, metric-calculation estimates, time and result characteristics of data transformation and so on. Manual definition of procedures calculating all the values would be very hard or even impossible, so adequate solutions to simplify this task had to be introduced. Fortunately, in meta-learning based on search and validation, it is not necessary to predict time and memory consumption accurately. Very rough estimates are fully satisfactory because the differences of orders of magnitude need to be found out, not just differences of several or tens percent. When a meta-learning process is limited by a total time given, it is important which methods are tested within the time, but not in which order. Even with very rough estimates, it is very rare to invert the order between very small/fast processes and very large/time-consuming ones. Thus, order disturbance usually affects meta-learning results only in the scope of similarly complex processes which happen to be tested at the end of assigned time—then an inversion may cause that a more complex task is run and less complex one is not because of the time limit. When the time limit is not too restrictive (that is, not tens but at least thousands of tests fit in time), even estimation errors of 100 % are not too significant.

Since the estimation of complexity is expected to reflect the orders of magnitude rather than the exact values, it is fully satisfactory to train some learning machines on specially prepared data about relations between experimental configurations and their results. Human experts preparing environments for such learning processes can use their meta-knowledge about the processes to point some values, that are known (or expected), to relate with time and memory consumption of the machine of concern. Then, approximation machines may be used to learn the final relationships.

For the purpose of the task of learning complexity estimation from examples, a special framework has been developed. It facilitates building advanced meta-evaluators, that learn their estimation functions from example applications. The major rules governing creation of evaluators are that

- a single evaluator may contain a number of *approximators* organized in *levels* and *layers*,
- training is performed on the basis of specially prepared data tables,
- training dataset is collected from experiments generated from an *observation configuration* and *scenario* for changes,
- each evaluator is trained once, and then can be used arbitrarily many times.

To realize all these goals, the meta-evaluator must implement the functionality defined by interface 6.5.

The main function of each meta-evaluator is *Adaptation()*, responsible for:

- performing the (final) adaptation of the values needed for complexity analysis (in both plain and learnable evaluators),

Interface 6.5 (Advanced meta-evaluator functionality)

Method: Adaptation(objectCollection)
Input: *Collection of objects with training data (objectCollection).*
Output: *Ready evaluator.*

Method: GetMethodsForApprox()
Input: *None.*
Output: *Collection of additional methods used for complexity estimation.*

Method: ApproximatorDataIn(level)
Input: *Integer defining the level for which the input data vector is requested.*
Output: *Input vector for approximators of given level (collection of real numbers).*

Method: ApproximatorDataOut(level)
Input: *Integer defining the level for which the output data vector is requested.*
Output: *Output vector for approximators of given level (collection of real numbers).*

- preparation of the experiment environment to collect learning data for approximators.

The approximators trained within an evaluator, learn real-valued functions on the basis of multidimensional real data. The training data sequences are collected from experiments performed in such a way, that an *environment for observations* (a test configuration) is created and its modifications obtained with a scenario *changes scenario* are analyzed to extract data items. Input and output data of each item are generated with calls to *ApproximatorDataIn()* and *ApproximatorDataOut()* methods of the evaluator. The observations environment and changes scenarios are the same solutions as those for generating test configurations in meta-parameter search machine (see Sect. 4.4.1).

Approximators can be created to estimate miscellaneous quantities. The most important ones are those responsible for assessment of time and memory consumption. Each method declared for analysis with *GetMethodsForApprox()* is also a subject to approximator creation and training. Yet another group of approximators may be created to estimate arbitrary values, that the author of the learning evaluator regards as helpful in learning other assessed quantities (for example, some information about predicted neural network structure, when automatically determined by the algorithm). The three groups are called *layers* and differentiate the way, the approximators are handled. The most important discrepancy between them is the way the output part of training data is prepared. Only the group corresponding to arbitrary values needs output specification by *ApproximatorDataOut()*. The other groups approximate functions of well-defined outputs, so it is known how they should be prepared—time and memory consumption must be measured for machines and the methods declared with *GetMethodsForApprox()* (memory occupation can be measured, because they return collections of created objects for the purpose of memory analysis).

The approximators are created and run one by one in appropriate order: first these belonging to the layer of arbitrary quantities, then the ones corresponding to declared methods and finally the one estimating time and memory consumption of the machine. Jankowski and Grąbczewski 2011 presented more detailed description of the approximation framework.

6.3.7 Example Experiment

The complexity-driven meta-learning algorithm, presented above, can be realized in different ways. Although it is an instance of the GML approach, it is still a general framework, not a single algorithm. Some general assumptions about CDML implementations of the functionality of the interface of proper meta-learner are presented in Sect. 6.3.1. The experiment described below has been performed with the following specification of particular functions.

GetTasks() As in all CDML implementations, two substantial modules have been defined: configurations generator and complexity estimation strategy.
The design of complexity estimation framework including meta-evaluators, presented above, is eligible for any kind of learning machines used in meta-learning projects, so it can be used without changes in many applications, and so it was, in this experiment.
The flow of machine configuration generators, used for search space definition, is presented in detail just below the items describing proper meta-learner functionality.
Because the experiment concerns classification problems, the machine configurations output by the flow have been embedded in the typical test validation scenario based on multiple cross-validation test presented in Fig. 6.5 on page 257.

Analyze() Each machine configuration, tested with cross-validation, was assessed by a measure of average accuracy obtained in the CV test.The results analysis performed below uses this measure as the estimator of machine configuration quality.

Result() From the point of view of this experimental analysis, it is not important, what is formally regarded as the result of the meta-learning algorithm. Here, the most interesting is the ranking of algorithms generated and all the details about the course of the ML process. It is the subject of detailed analysis presented below.

ShouldStop Because we consider an experiment to present the mechanisms of CDML and their advantages, the stop criterion is not typical for real-life approaches. To maximize analysis possibilities, the number of generated configurations was kept small and no special stop criterion was used (apart from the standard ones of the GML), so that all configurations could be successfully tested and CPU time management analyzed.

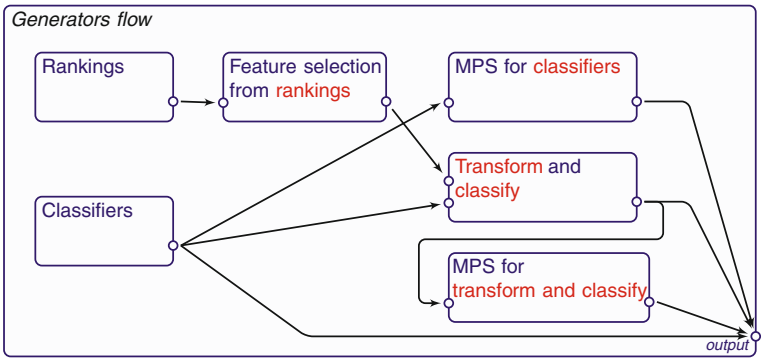


Fig. 6.11 Generators flow used in the experiment

Generators Flow

For the sake of readable analysis, facilitating observation of different mechanisms in action, the generators flow used for this CDML experiment is rather simple. It is not the best choice for solving classification problems in general, but lets us better see very interesting details of cooperation between configuration generators and complexity control mechanism including the quarantine. The generators flow used in the experiment is presented in Fig. 6.11.

To know what exactly will be generated by the generators flow, the configuration of classifiers generator and rankings generator must be specified. Here, the configuration includes the following sets:

Classifiers:

1. *kNN (Euclidean)*—k Nearest Neighbors with Euclidean metric,
2. *kNN [MetricM (EuclideanOUO)]*—kNN with Euclidean metric for ordered features and Hamming metric for unordered ones,
3. *kNN [MetricM (Mahalanobis)]*—kNN with Mahalanobis metric,
4. *NBC*—Naive Bayes Classifier
5. *SVMClassifier*—Support Vector Machine with Gaussian kernel
6. *LinearSVMClassifier*—SVM with linear kernel
7. *[ExpectedClass, kNN [MetricM (EuclideanOUO)]]*—first, the ExpectedClass[1] machine transforms the original dataset, and then, the transformed dataset becomes the learning data for kNN,
8. *[LVQ, kNN (Euclidean)]*—first, Learning Vector Quantization algorithm (Kohonen 1986) is used to select prototypes, then kNN uses them as its training data (neighbor candidates),
9. *Boosting (10x) [NBC]*—boosting algorithm with 10 NBCs.

[1] ExpectedClass is a transformation machine, which outputs a dataset consisting of one "super-prototype" per class. The super-prototype for each class is calculated as the vector of means (for ordered features) or expected values (for unordered features) for given class. Followed by a kNN machine, it composes a very simple classifier, though sometimes quite successful.

Rankings:

1 *RankingCC*—correlation coefficient based feature ranking,
2 *RankingFScore*—Fisher-score based feature ranking.

The sets of classifiers and ranking algorithms, together with the generators flow presented in Fig. 6.11, produce 54 configurations, that are nested (one by one) within the meta-learning test-scheme (Fig. 6.5 on page 257) and sent to the complexity control module. The 54 configurations come from the four generators connected to the flow output: 9 from pure Classifiers, 9 from MPS for classifiers, $2 \times 9 = 18$ from Transform and classifygenerator (2 feature selections combined with 9 classifiers) and another 18 from the MPS for transform and classify generator.

All the configurations provided by the generators flow are presented in Table 6.1. For the sake of further analysis, the order in the table corresponds to the complexities estimated for a particular dataset. The square brackets, used in the descriptions of the machines, denote submachine relation. A machine name standing before the brackets is the name of the parent machine, and the machines in the brackets are the submachines. When more than one name is embraced with the brackets (comma-separated names), the submachines are placed within a scheme machine. Parentheses embrace significant parts of machine configurations.

To make the notation easier to read, some entries of the table are explained below. The notation does not present the input–output interconnections, so it does not allow to reconstruct the full scenario in detail, but it shows the machine structure, which is sufficient here and significantly reduces the occupied space. With the information about the generators flow and particular machines, that can be found in the preceding chapters and sections, it is not difficult to faithfully reconstruct the whole machine test scenarios.

The following notation:

TnC [[[RankingCC], FeatureSel], [kNN (Euclidean)]]

means that a feature selection machine selects features from the top of a correlation coefficient based ranking, and next, the dataset composed of the selected features is an input for a kNN with Euclidean metric—the combination of feature selection and kNN classifier is controlled by a transform-and-classify machine. The sequence

TnC [[[RankingCC], FeatureSel], [LVQ, kNN (Euclidean)]]

means nearly the same as the previous example, except the fact that an LVQ machine for instance selection functions between the feature selection machine and the kNN.

The following notation represents the meta-parameter search machine which optimizes parameters of a kNN machine:

MPS [kNN (Euclidean)]

Table 6.1 Complexities of the tasks produced by the generators flow for mushroom data

22	TnC [[[RankingFScore], FeatureSel], [NBC]]	5.89E + 006
13	TnC [[[RankingCC], FeatureSel], [NBC]]	5.91E + 006
4	NBC	6.50E + 006
31	MPS [NBC]	6.72E + 006
7	[ExpectedClass, kNN [MetricM(EuclideanOUO)]]	8.79E + 006
25	TnC[[[RankingFScore], FeatureSel], [ExpectedClass, kNN [MetricM(EuclideanOUO)]]]	9.21E + 006
16	TnC [[[RankingCC],FeatureSel], [ExpectedClass, kNN [MetricM (EuclideanOUO)]]]	9.22E + 006
19	TnC [[[RankingFScore], FeatureSel], [kNN(Euclidean)]]	1.12E + 007
10	TnC [[[RankingCC],FeatureSel], [kNN (Euclidean)]]	1.13E + 007
26	TnC[[[RankingFScore], FeatureSel], [LVQ, kNN (Euclidean)]]	2.11E + 007
17	TnC [[[RankingCC], FeatureSel], [LVQ, kNN(Euclidean)]]	2.11E + 007
27	TnC [[[RankingFScore],FeatureSel], [Boosting (10×) [NBC]]]	5.41E + 007
18	TnC [[[RankingCC], FeatureSel], [Boosting (10×) [NBC]]]	5.41E + 007
34	MPS [ExpectedClass, kNN [MetricM (EuclideanOUO)]]	8.91E + 007
8	[LVQ, kNN (Euclidean)]	9.49E + 007
9	Boosting (10×) [NBC]	1.04E + 008
36	MPS[Boosting (10×) [NBC]]	1.04E + 008
20	TnC[[[RankingFScore], FeatureSel], [kNN [MetricM (EuclideanOUO)]]]	1.79E + 008
11	TnC [[[RankingCC], FeatureSel], [kNN[MetricM (EuclideanOUO)]]]	1.79E + 008
49	MPS [TnC[[[RankingFScore], FeatureSel], [NBC]]]	1.81E + 008
40	MPS [TnC [[[RankingCC], FeatureSel], [NBC]]]	1.81E + 008
52	MPS [TnC [[[RankingFScore], FeatureSel], [ExpectedClass, kNN[MetricM (EuclideanOUO)]]]]	2.65E + 008
43	MPS [TnC[[[RankingCC], FeatureSel], [ExpectedClass, kNN [MetricM(EuclideanOUO)]]]]	2.65E + 008
21	TnC[[[RankingFScore], FeatureSel], [kNN [MetricM (Mahalanobis)]]]	2.91E + 008
12	TnC [[[RankingCC], FeatureSel], [kNN[MetricM (Mahalanobis)]]]	2.91E + 008
46	MPS [TnC[[[RankingFScore], FeatureSel], [kNN (Euclidean)]]]	3.17E + 008

Continued.

Table 6.1 Continued

37	MPS [TnC [[[RankingCC], FeatureSel], [kNN(Euclidean)]]]	3.18E + 008
1	kNN (Euclidean)	3.56E + 008
2	kNN [MetricM (EuclideanOUO)]	3.61E + 008
53	MPS [TnC [[[RankingFScore], FeatureSel], [LVQ, kNN(Euclidean)]]]	5.64E + 008
44	MPS [TnC[[[RankingCC], FeatureSel], [LVQ, kNN (Euclidean)]]]	5.64E + 008
24	TnC [[[RankingFScore], FeatureSel],[LinearSVMClassifier [LinearKernelProvider]]]	1.01E + 009
15	TnC [[[RankingCC], FeatureSel], [LinearSVMClassifier[LinearKernelProvider]]]	1.01E + 009
	TnC[[[RankingFScore], FeatureSel], [SVMClassifier [KernelProvider]]]	1.08E + 009
14	TnC [[[RankingCC], FeatureSel],[SVMClassifier [KernelProvider]]]	1.08E + 009
54	MPS[TnC [[[RankingFScore], FeatureSel], [Boosting (10×) [NBC]]]]	1.60E + 009
45	MPS [TnC [[[RankingCC], FeatureSel],[Boosting (10×) [NBC]]]]	1.60E + 009
6	LinearSVMClassifier [LinearKernelProvider]	1.65E + 009
33	MPS [LinearSVMClassifier [LinearKernelProvider]]	1.65E + 009
3	kNN [MetricM (Mahalanobis)]	1.97E + 009
5	SVMClassifier [KernelProvider]	2.40E + 009
29	MPS[kNN [MetricM (EuclideanOUO)]]	2.48E + 009
47	MPS[TnC [[[RankingFScore], FeatureSel], [kNN [MetricM(EuclideanOUO)]]]]	3.75E + 009
38	MPS [TnC[[[RankingCC], FeatureSel], [kNN [MetricM (EuclideanOUO)]]]]	3.75E + 009
48	MPS [TnC [[[RankingFScore],FeatureSel], [kNN [MetricM (Mahalanobis)]]]]	6.06E + 009
39	MPS [TnC [[[RankingCC], FeatureSel], [kNN [MetricM(Mahalanobis)]]]]	6.06E + 009
35	MPS [LVQ, kNN(Euclidean)]	8.19E + 009
30	MPS [kNN [MetricM(Mahalanobis)]]	1.35E + 010
28	MPS [kNN (Euclidean)]	1.87E + 010
51	MPS [TnC [[[RankingFScore],FeatureSel], [LinearSVMClassifier [LinearKernelProvider]]]]	2.18E + 010
42	MPS [TnC [[[RankingCC], FeatureSel],[LinearSVMClassifier [LinearKernelProvider]]]]	2.18E + 010
50	MPS [TnC [[[RankingFScore], FeatureSel], [SVMClassifier[KernelProvider]]]]	2.41E + 010
41	MPS [TnC[[[RankingCC], FeatureSel], [SVMClassifier [KernelProvider]]]]	2.41E + 010
32	MPS [SVMClassifier [KernelProvider]]	9.24E + 010

In the case of

MPS [LV, kNN (Euclidean)]

both LVQ and kNN parameters are optimized by the MPS machine. However, in the case of notation

MPS [TnC [[[RankingCC], FeatureSel], [kNN (Euclidean)]]]

only he number of chosen features is optimized. This configuration is provided by the MPS for transform and classify generator (see Fig. 6.11), where the MPS configuration is set up to optimize only the parameters of the feature selection machine. Of course, in general, the MPS machine can optimize all the parameters of all submachines, but this is not the goal of the example and, moreover, the optimization of too many parameters may provide very complex machines (sometimes uncomputable in a rational time).

Given the notation, sketched above, it is quite easy to recognize the source generator of each item in Table 6.1. The four kinds of entries start in characteristic way: some with *MPS* followed by *TnC*, others with *MPS* but not followed by *TnC*, some start with *TnC* and the remaining nine represent pure machines from the set-based generator of classifiers (start with neither *MPS* nor *TnC*).

Data Benchmarks

Table 6.2 summarizes the properties of five data benchmarks from the UCI repository, selected for the presentation of CDML in action. All the benchmarks are classification problems. The goal is not to gain as attractive classification results as possible, but to monitor the mechanisms of meta-learning by a couple of examples. Therefore, the analysis is not focused on accuracies obtained by the models, but on the correctness of complexity based task ordering.

Table 6.1 presents condensed descriptions of all the classifier configurations that came out from the configuration generators flow, ordered by the complexities, calculated according to Eq. (6.15) (for full test scenarios, not just the machine configuration) for the mushroom dataset. The table has three columns: the first one contains the task id which reflects the order in which the configurations are provided by the

Table 6.2 Benchmark data for the CDML experiment

Dataset	# Classes	# Instances	# Features	# Ordered f.
Mushroom	2	8124	22	0
German-numeric	2	1000	24	24
Glass	6	214	9	9
Splice	3	3190	60	0
Thyroid-all	3	7200	21	6

generators flow, the second column is the task configuration description, and the third one shows the task complexity (the ordering key). It is important to realize that the numbers are not directly interpretable as amounts of time or memory consumption, because they were calculated as proper combinations of the two.

CDML Results Diagrams

The CDML algorithm results obtained for the benchmarks are presented in the form of diagrams. Each diagram is very reach in specific CDML information, presenting many properties of the meta-learning algorithm in application to one of the datasets. The diagrams present information about the relative times of starting, stopping and breaking of each task, about complexities (global, time and memory) of each test task, about the order of the test tasks (according to their complexities, compare Table 6.1) and about the accuracy of each tested machine.

In the middle of the diagram—see the first diagram in Fig. 6.12—there is a column with task ids (the same ids as in Table 6.1). Row ordering reflects the complexities of test tasks, but is inversed in relation to that of Table 6.1: the most complex tasks are placed at the top and the tasks of the smallest estimated complexity (run by the system as the first ones) at the bottom.

The beginning of Table 6.1 illustrates the possibility, signaled before, that more complex machine structures may in fact bring lower complexities. It is fully understandable, that for example, single NBC machine can run longer than a machine with the same configuration run for a dataset obtained after some feature selection. Therefore, such order between the first two rows and the third one.

On the right side of the "Task id" column, there is a plot presenting starting, stopping and breaking times of each test task. As it was presented above, the tasks are started in the order of the estimates of their complexities, and with time limits defined on the basis of the estimates. When the task does not end within the limit, it is broken and restarted according to the modified complexity (see Sect. 6.3.4). For examples of restarted tasks please look at Fig. 6.12, at the topmost and the third topmost tasks (id 32 and 50)—in each of the two rows, there are two horizontal bars corresponding to two periods of the task run. The break means that the task was started, broken because of exceeded allocated time and restarted when the tasks of larger complexities got their turn. In both rows referred above, we can observe the results of Intemi system efficiency due to the cache system and unification. The second bar in both examples is shorter than the first bar for the same task. If Intemi had to recalculate the whole machine again, each next bar for the same machine would be longer than the previous one. The fact that the second bars are shorter means that the unification system reused parts of the machine from the previous run and continued the process instead of running it from the beginning. The task with id 50 is a perfect example of this: when the task was restarted it reused all the subtasks finished during the first pass so that the second bar is very thin, meaning that very little time was consumed after the restart.

A survey of different diagrams (in Figs. 6.12, 6.13, 6.14, 6.15 and 6.16) easily brings the conclusion that the amount of inaccurately predicted time complexity

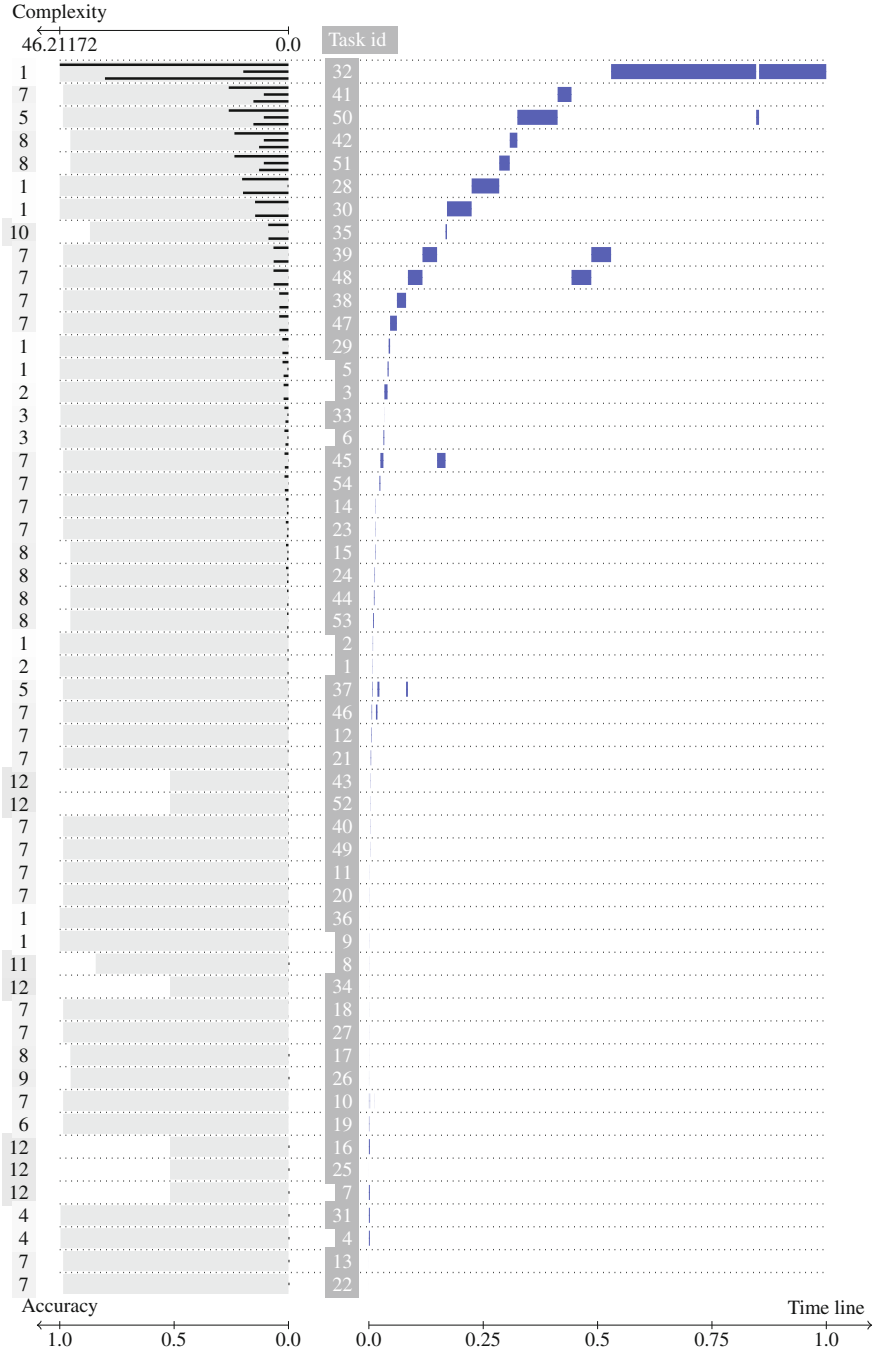


Fig. 6.12 Results for mushroom data

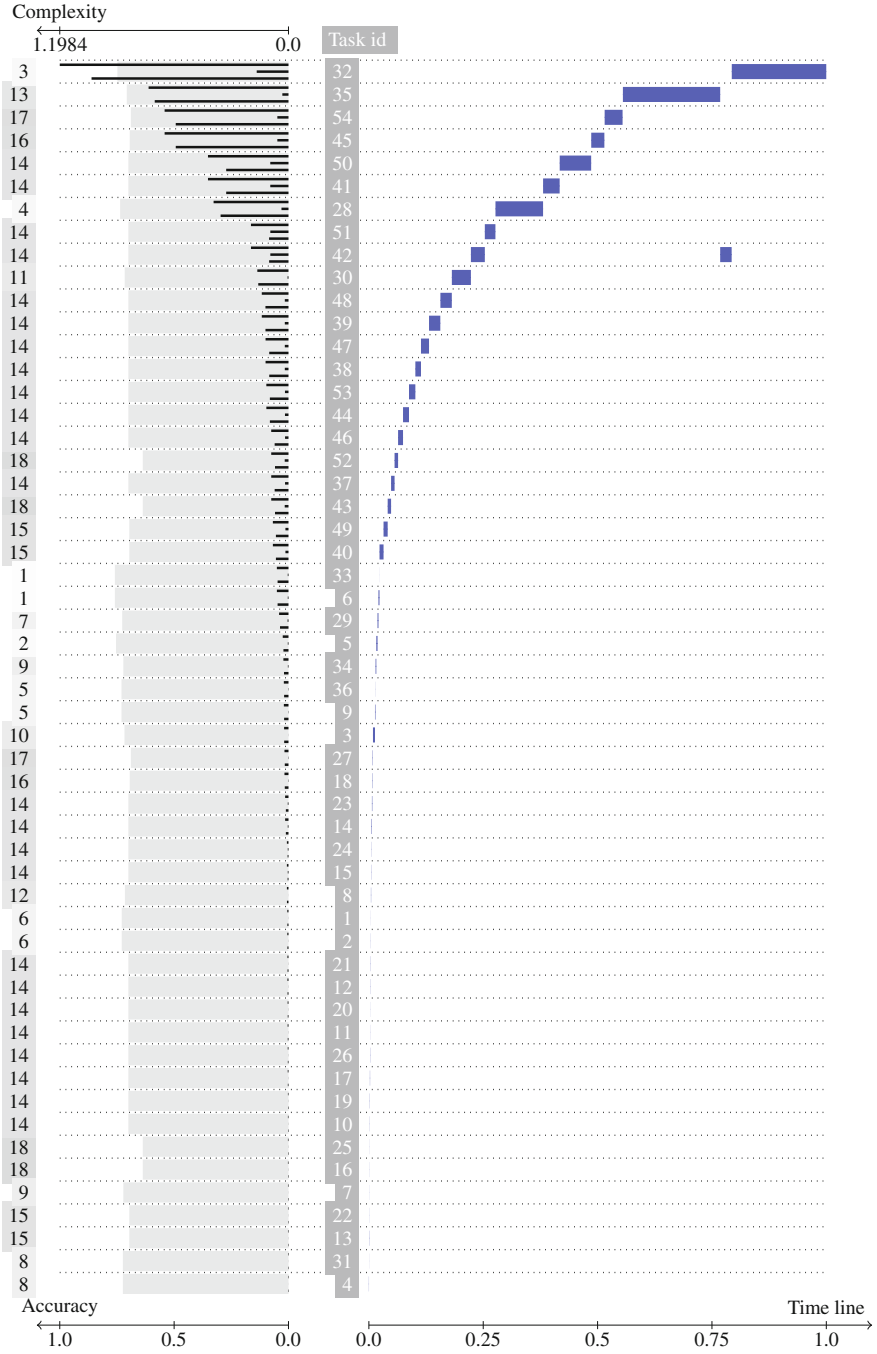


Fig. 6.13 Results for german-numeric data

is quite small (there are quite few broken bars). It confirms that the prediction of relations between learning machine complexities can be quite accurate.

At the bottom of the figure, the "Time line" axis can be seen. The scope of the time is the interval [0, 1] to show the times relative to the start and the end of all the computations for the dataset. To make the diagrams clearer, the tests were performed on a single CPU, so only one task was running at a time. Thus, there are no bars overlapping in time. If the projects were run on more than one CPU, a number of bars would be "active" at almost each time point, which would make reading the plots more difficult.

The tasks estimated as the simplest, are started first. They can be seen at the bottom of the plot. Their bars are very short because they required relatively short time to be calculated. It confirms that the tasks started at the beginning are really the simplest. The higher in the diagram (that is, the larger predicted complexity), the longer bars can be observed. Again, it confirms the adequacy of the complexity estimation framework, because the relations between the predictions correspond very good to the relations between real time consumed by the tasks. When browsing other diagrams, similar behavior can be observed—the simplest tasks are started at the beginning and then, more and more complex ones are executed.

The leftmost column of the diagram presents ranks of the test tasks (the ranking of the accuracies). In the case of the mushroom data, it is not difficult to obtain 100 % accuracy, so there are many machines with rank 1.

Between the columns of task id and accuracy ranks, on the left side of each diagram, the accuracies of classification test tasks and their approximated complexities are presented. At the bottom, there is the "Accuracy" axis with interval from 0 (on the right) to 1 (on the left side). Each test task is illustrated with a gray bar, starting at 0 and finished at the point corresponding to the accuracy, so that the accuracies of all the tasks are easily visible and comparable (the longer bar, the higher accuracy). Although the best accuracies obtained among the 54 classification methods seem not bad, it is not justified to draw far-reaching conclusions from the experiments, as they were not tuned to obtain the best accuracies possible, but to illustrate the behavior of the complexity driven meta-learning and the generators flow.

On top of the gray bars corresponding to the accuracies, some thin solid lines can be seen (at least in the upper part of the diagram). The lines start at the right side as the accuracy bars and go to the right according to proper magnitudes involved in calculation of complexity. Three lines are drawn for each task. They correspond to the total complexity (Eq. 6.15), the memory complexity (l_p) and the time complexity ($t / \log t$). All three complexities are assessed with the approximation framework. Approximated complexities presented on the left side of the diagram can be easily compared visually to the time-schedule obtained in the real time on the right side of the diagram. Longer lines mean higher complexities. The longest line is spread to maximum width and the others are proportionally shorter. Therefore, the complexity lines at the top of the diagram are long while the lines at the bottom are almost invisible. It can be seen that sometimes the time complexity of a task is smaller while the total complexity is larger and vice versa. For an example see the tasks 28 and 51 in Fig. 6.12. Memory complexity is related to the kind of machine, for example, top

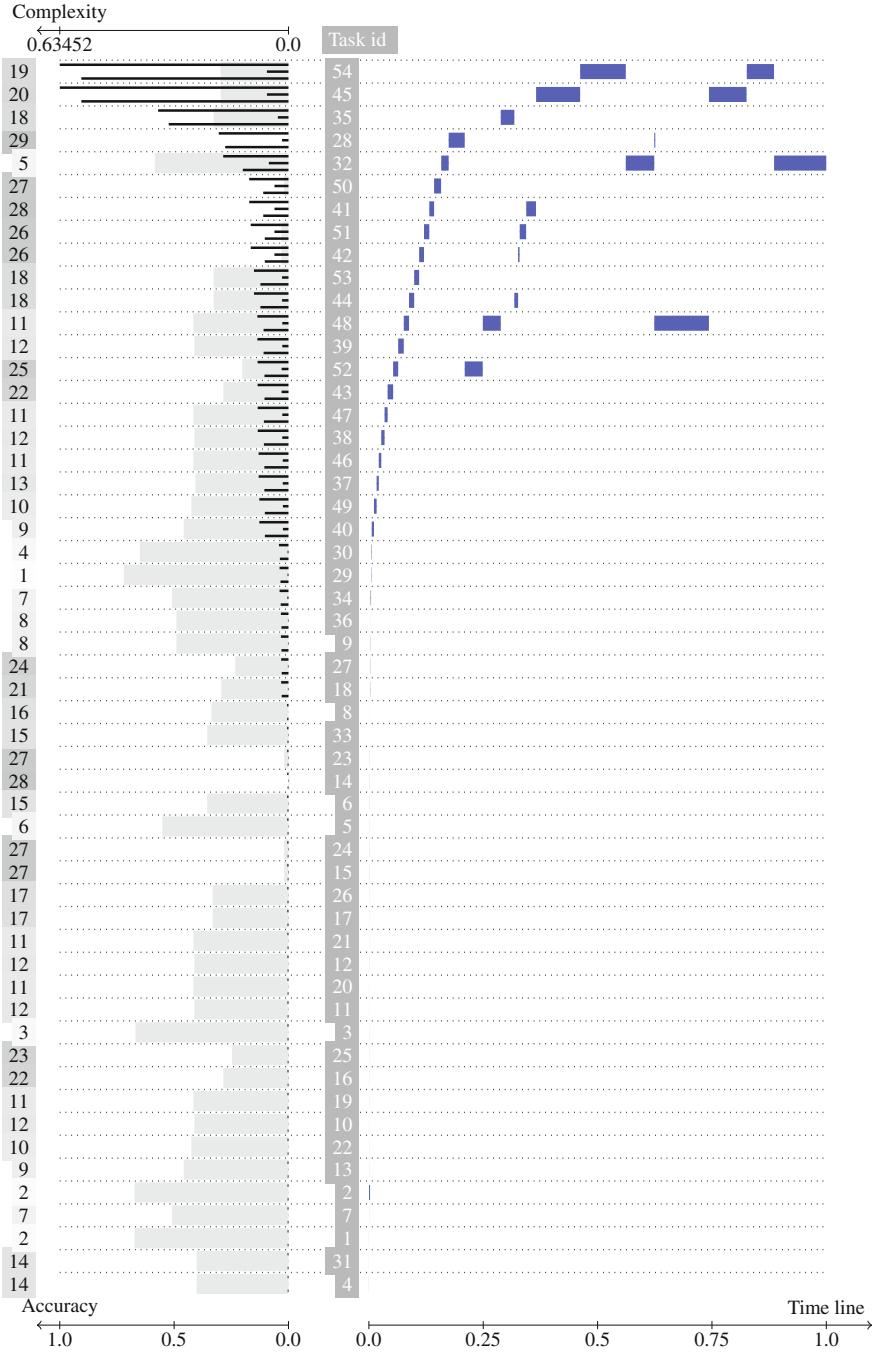


Fig. 6.14 Results for glass data

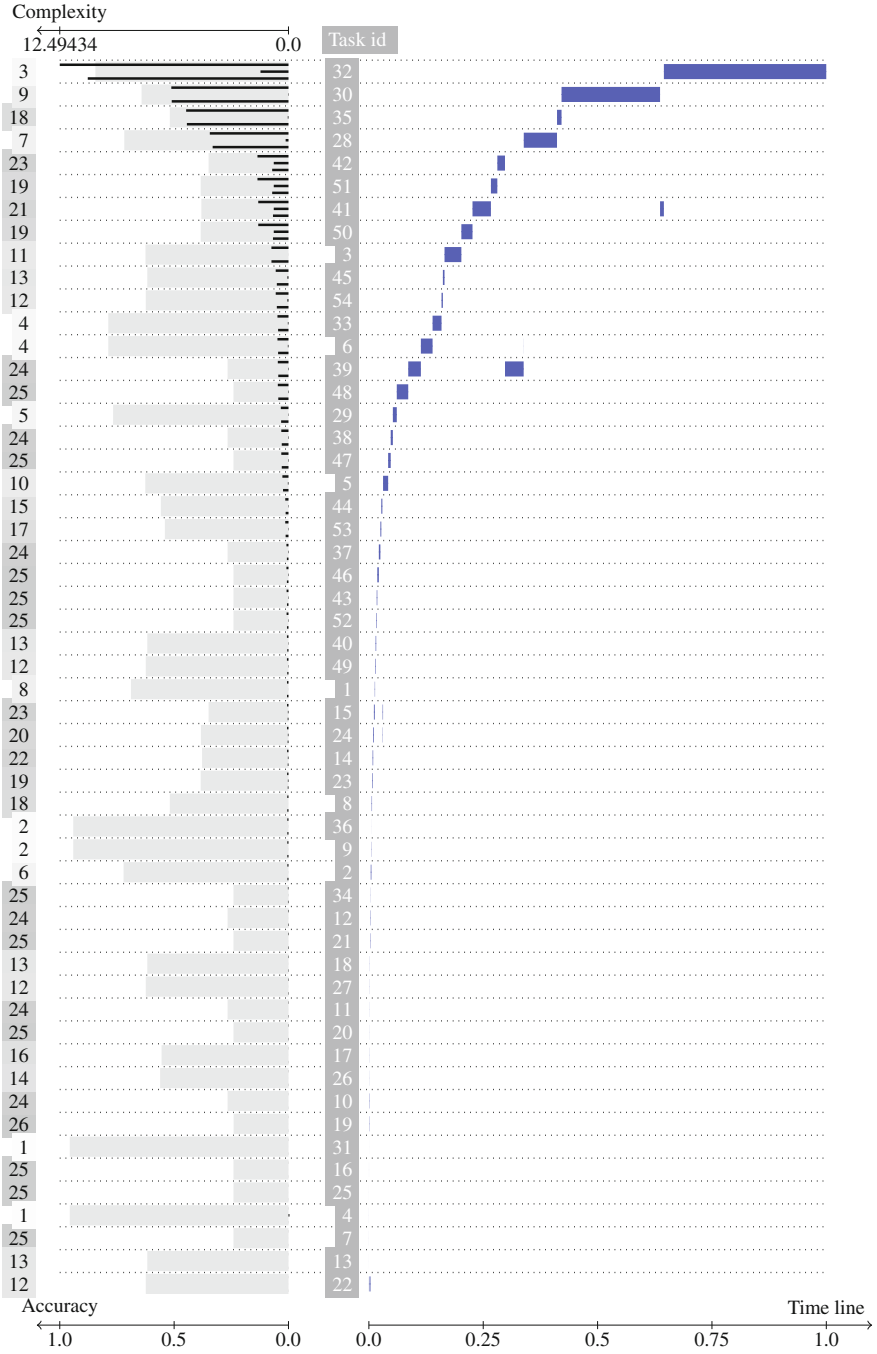


Fig. 6.15 Results for splice data

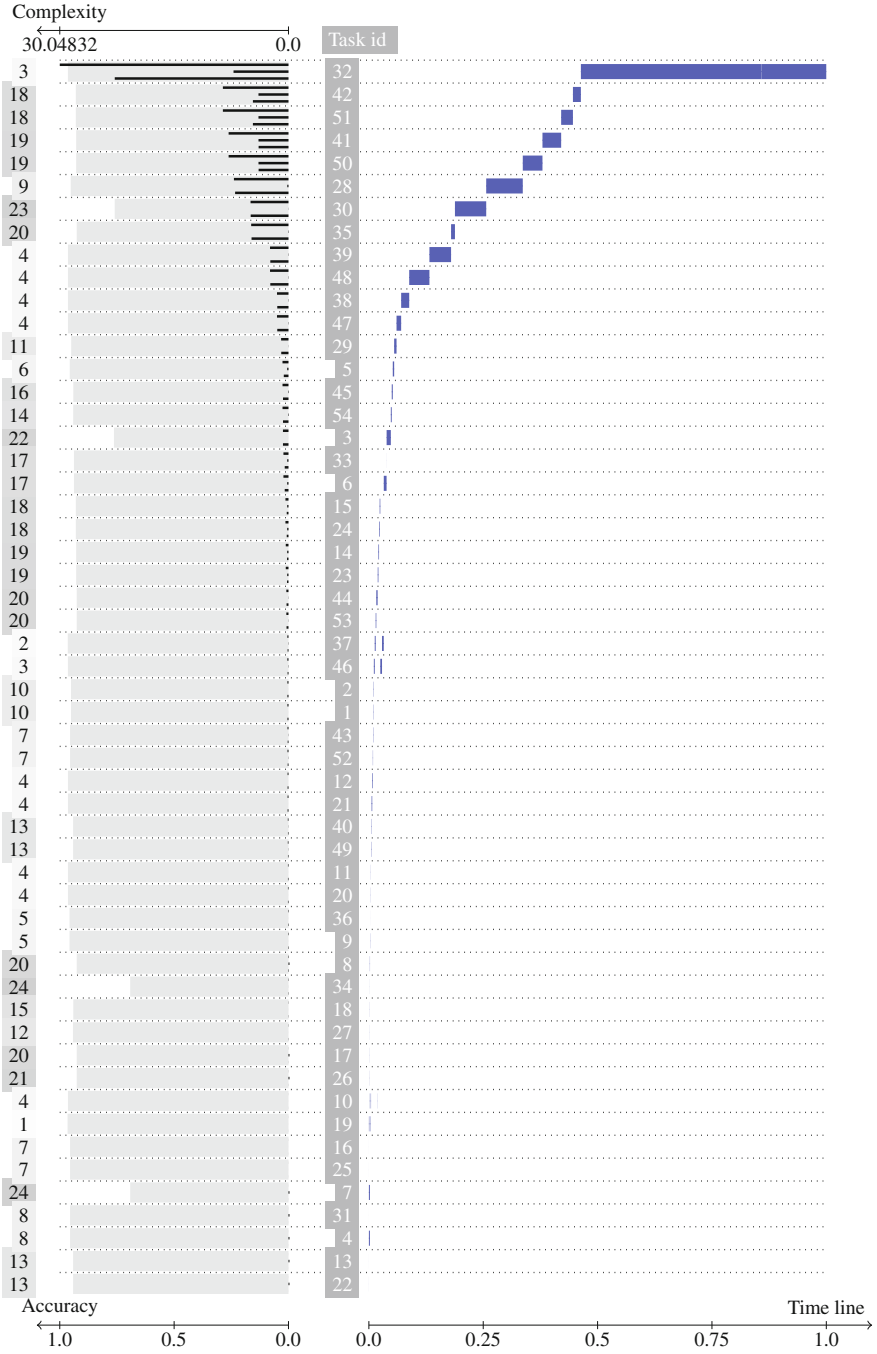


Fig. 6.16 Results for thyroid-all data

5 rows of the diagram (5 machines estimated as the most complex) have memory complexity part, significantly different than 0 (hence, at all visible). They correspond to machines involving SVM , so memory consumption is mainly due to the kernel tables. The following rows, have almost invisible memory-complexity lines. They correspond to the machines engaging kNN , so the models do not use much memory.

Results Analysis

The diagrams illustrating CDML operation for different datasets (Figs. 6.12, 6.13, 6.14, 6.15 and 6.16) clearly show that the behavior of machines changes between benchmarks. When looking at accuracies within some test, groups of machines of similar accuracy may be seen, however for other benchmark, within the same group of machines, the accuracies are quite diverse. Of course, the complexity of a test task for given configuration may change significantly from benchmark to benchmark. However, it can be seen that in the case of benchmarks of similar properties (like similar feature counts), the permutations of task ids in the diagrams are partially similar (for example, see the bottoms of Figs. 6.12 and 6.16).

The most important feature of the CDML algorithm is that it facilitates finding accurate solutions in the order of increasing complexity. Simple solutions are started before the complex ones, to maximize the probability that a simple and accurate solution is found as soon as possible. It is confirmed by the diagrams in Figs. 6.12, 6.13, 6.14, 6.15 and 6.16. Thanks to this property, in the case of a strong stop-condition (significant restriction on the running time) we are able to find really good solution(-s) because of starting test tasks in proper order. Even if some tasks get broken and restarted, it is not a serious hindrance to the main goal of the algorithm.

For some benchmarks, very simple and accurate models were found just at the beginning of the meta-learning process. For an example see task ids 4 and 31 in Fig. 6.15 or task id 19 in Fig. 6.16. But "simple" does not refer to the structural simplicity, but to the complexity measure of Eq. (6.15). The most accurate machine (of the 54 machines being analyzed) for thyroid data is the combination of feature selection based on F-score with kNN machine (task id 19). In the case of very huge datasets (with huge numbers of instances and features) almost no single algorithm works successfully in rational time. However, the same algorithms preceded by not too complex data transformation methods (like feature selection or instance selection) may be calculated in a very short time. The transformations may reduce the costs of learning and testing of the machines applied afterwards, resulting in significant decrease of the overall time/memory consumption.

In some of the benchmarks (see Figs. 6.12 and 6.13) the most accurate machine configurations were not of so small complexity as in the cases mentioned above. The CDML algorithm running on the mushroom data has found several alternative configurations of very good performance: the simplest is a boosting of naive bayes classifier (task id 9), the second simplest is the *kNN [MetricM (EuclideanOUO)]*, followed by SVM (with Gaussian kernel). Some of the most complex machines applying *MPS* to *kNN* and *SVM* have also finished with 100 % accurate models. For the german-numeric benchmark, the best machines are SVMs with linear and

Gaussian kernels (task ids 6, 33 and 5). The winner machines, for this benchmark, are of average complexity and are placed in the middle of the diagram.

Naturally, in most cases, more optimal machine configurations may be found when using more sophisticated configuration generators and larger sets of classifiers and data transformations (for example: adding decision trees, instance selection methods, feature aggregation and other methods) and performing deeper parameter search.

The approximated complexity time is not in perfect correlation with the real time consumed, presented by the bars at the right side of the diagrams. The differences are due to not only the approximation inaccuracy, but also the machine unifications and natural deviations in real CPU time consumption observed also for two runs of the same task (probably it is caused by the .Net kernel, for example, by garbage collection which, from time to time, must use the CPU to perform its tasks).

Without repeating the experiments, one can think of the results obtained with the stop criterion set to a time-limit constraint. For example, assume the time limit set to 20 % of the time really used by the whole analysis of 54 machines. In such case, some of the solutions described above would not be reached, but still, for several datasets the optimal solutions would be obtained and for other benchmarks, some slightly worse solutions would be the winners. A very important observation is that thanks to the complexity control solutions of CDML, in the 20 % of time, about 85 % of candidates would have been examined and only several most complex methods would not be tried because of the time limits.

These properties of CDML are very advantageous because in real life problems the time is always limited, and to find as good solutions as possible within the time, it is necessary to test as many candidate machines (suspected of being successful) as possible.

6.4 Profile-Based Meta-Learning

Driving meta-search with complexity control is very attractive and adequate when the set of machines to be tested is diverse with respect to memory and time consumption. When the task of meta-learning is to search for the most successful learners among large groups of similarly complex machines, complexity control can not offer a reasonable solution, because complexity prediction errors may by larger than real differences between amounts of resources occupied by the algorithms. Even if accurate complexity prediction were possible, it would be no point in testing learners in the order of their complexity, because time savings would be marginal and probability of finding more accurate machines before less accurate ones would be fifty-fifty (exactly as when running the tests in completely random order), because of likely independence between complexity and accuracy.

The problems of handling many machine configurations of similar complexity are inherent in the realm of decision trees. Many algorithms can be constructed by combining various components in appropriate ways, so two methods may differ only in a single component like split quality measure , stop criterion, validation method

and so on. Such difference in configuration may result in none or tiny difference in time and memory consumption of the algorithms. At the same time, the difference in accuracy may be significant. Therefore, we can not base on complexity estimates (which are equally successful as random guess in this case), but need other knowledge about the machines abilities, to judge which machines should be validated first and which ones are not worth testing at all.

A rationale behind the approach of *profile-based meta-learning* (PBML) is that the relative differences between the results obtained by different machines may point the directions to more attractive machine configurations. The idea is similar to relative landmarking, but in contrast to it, is realized in an active way—there is no a priori definition of landmarkers, but instead, a results *profile* is created as a collection of results obtained in validation processes for selected machine configurations. The selection of profile members is not fixed a priori as it is in landmarking approaches. Here, any machine can become a landmarker, if its results are assessed to be useful in prediction of attractive solutions (not necessarily must be attractive itself). On the basis of profile similarity between learning tasks, the candidates for best results providers are determined as the ones that turned out to be the most successful in solving the most similar problems.

Profiles are called active, because they can change on-line, when the feedback shows that the current profile predictions are not as good as they could. Because of the similarity to relative landmarking, the technique could also be called an *adaptive (or active) relative landmarking* .

Active ranking validation is a procedure resembling what human experts do when they want to find as accurate model as possible, within some limited time period. Since they usually have some suspicions (resulting from their expertise) about what learning machines may be adequate for the task, they test different candidates and propose the models which passed the test with the highest scores. After each test, they analyze the results and possibly modify their bets about the methods of most probable success. Naturally there are significant differences between operation of human experts and automated processes. Some of them are in favor of humans and some in favor of machines. The automated search procedures conduct the experiments in more systematic ways, so they are not susceptible to forgetting about simple solutions that should be tested first, they better use the time available and so on. The fundamental difference in humans' favor is that humans have still more abilities to learn from the experiments and use their expertise to efficiently modify the sequence of candidate configurations when they read from the experiments results that their original assumptions were not fully correct. It can be said that human experts perform active search, as they modify their plans of the search according to the results of performed tests. This has been one of the most significant inspiration for the automated active search in the form of PBML.

The idea of profile-based meta-learning is not an alternative to complexity-driven ML. The two approaches should be seen as complementary, since CDML can successfully deal with collections of algorithms of diverse complexity, and needs special attractiveness modules to adjust to estimated success possibilities, while PBML can deal with families of methods of similar complexity and is focused on discovery

of successful machines. Proper combination of the two approaches can join their forces. One of the possibilities is to use a profile-based analysis as an attractiveness estimation module of CDML.

6.4.1 The Algorithm

Profile-based meta-learning can be presented as an instance of the general meta-learning (GML) approach defined by Algorithm 6.1 (on page 254). It means that PBML is an iterative procedure, testing properly selected machine configurations. The selection is based on prediction of the best algorithms made by means of an analysis of gathered results.

Profiles of results are created and maintained during the ML process to encode the meta-information contained within the learning machines test results, and use the knowledge to rank other methods. Their applications may take advantage of relations between results to determine the most promising directions of machine parameter changes.

Method Parameters

Although PBML is an instantiation of GML, it is still a very general algorithm and can be configured to search in different machine configuration spaces, with different detailed strategies.

Apart from the general parameters like the time deadline and task runner, defined and handled by the GML algorithm, there is a number of parameters pertaining to the subject matter of PBML. The parameters specific to PBML are:

- the *set of machine configurations* $\mathscr{C}$, the search is performed within,
- *validation scenario* VS, defining how candidate configurations are tested,
- *result calculator* RC, that is, a tool to extract a result assessing the machine quality on the basis of the test just performed,
- *profile manager* PM, defining the rules of profile management and its use for subsequent rankings generation,
- *number* cpi *of configurations* tested in each iteration.

All they can get miscellaneous forms and values, so that PBML is still quite large family of algorithms.

In contrary to CDML, machine configurations are not generated gradually, but all possible configurations are in the scope of interest from the very beginning.

The validation scenario defines what needs to be done to facilitate estimation of the quality of a given machine configuration. The implementation inside the Intemi framework uses the most natural solution for such purposes, namely template schemes. A template playing the role of VS must contain a single placeholder which is filled with the configuration to be tested. Such instantiated template becomes a feasible configuration and the corresponding complex test machine may be requested

from the task runner. For testing classification learners, the most commonly used *VS* is a template for cross-validation such as the one presented in Fig. 6.5 (on page 257).

When the machine corresponding to the *VS* is ready, the result calculator *RC* is asked to assess the quality of the machine just tested. In general, we can just assume, that the quality result is an element of a set $\mathscr{R}$ (specified for particular purposes). It may be just the set of real numbers or a set of collections of results (from multiple test) or any other form of results—the only requirement is that each two results can be compared (and thanks to that, used as keys for ordering). In Intemi, the result estimating machine quality is also calculated in the most natural way supported by the system, that is, with a query augmented by a series transformation. The query extracts a series of results from the test machine hierarchy and the series is transformed accordingly. When searching for classification machines, the most natural solution is a query collecting classification accuracies and a simple series transformer calculating the mean value (sometimes it is preferred to estimate the quality by the mean accuracy decreased by the standard deviation observed in the sample).

The profile manager (*PM*) is directly responsible for profile maintenance and its usage for measurements of learning problems similarity. More information on this subject can be found below, in Sect. 6.4.2.

Natural number *cpi* defines the count of machine configurations, tested in parallel, before profile changes take effect. The strategy of PBML is to return a collection of this size with a call to *GetTasks()* method, and then withhold returning next tests until the ones already submitted are finished. According to the GML algorithm, methods *Analyze()* and *GetTasks()* are called each time a task gets finished, but it is up to the meta-learner, when *GetTasks()* returns nonempty collection. PBML updates the profile if necessary, but returns next package of tests for running only after the previous package is completely served. Setting *cpi* to be greater than 1 has three major roles:

- lets the runners-up of the rankings also be validated,
- speeds up the process because of less frequent ranking generation,
- facilitates more parallelization of the tests.

In the experiments of Sect. 6.4.5 *cpi* was set to 5, so that 5 top-ranked configurations of each ranking were validated and new rankings were created after validation of each 5 configurations.

Internal Data Structures

Meta-learners steer the GML algorithm with subsequent calls of their methods. To keep track of the learning process, they need to create proper internal representation of the process and reflect it in their responses to calls from the GML algorithm.

Main PBML internals are three collections of machine configurations equipped with some additional information:

- C_R—a collection of machine configurations c validated so far (in the search process) assisted by their validation results r (formally, collection of pairs $cr \in \mathscr{C} \times \mathscr{R}$),
- $C_P \subseteq C_R$—a collection of specially selected results (the *profile*),
- C_Q—a sequence of candidate configurations (the queue) ordered with respect to the estimated qualities and step numbers at which they were added.

An auxiliary ordered collection C_B is used temporarily in the algorithm to represent the current ranking of candidate machine configurations.

To avoid long but fully formal statements, some shortcuts are taken below. For example, the expression "configuration in the profile" should be understood as appropriate pair of the configuration and its result. Although slightly informal, they will not introduce ambiguities and will significantly simplify descriptions.

In the Algorithms 6.6 and 6.7, two control counters are used: *step* and *tasksRunning*. The former iterates calls to *GetTasks()* and the latter informs about the number of requests that have been submitted and have not generated a feedback yet, to indicate when the next package of tasks should be passed to the task runner.

Proper Meta-learning Methods

Full functionality of a meta-learner within GML algorithm is obtained by proper implementation of the four members composing interface 6.2 (page 21). Before standard search cycles start, internal PBML structures need initialization. It could be done in the first call to *GetTasks()*, but object-oriented programming knows better locations of such initialization. Regardless of the placement, the initialization needs the following instructions:

1. $C_R \leftarrow \emptyset$
2. $C_Q \leftarrow \emptyset$
3. *step* $\leftarrow 0$
4. *tasksRunning* $\leftarrow 0$
5. Initialize *PM* (with $\mathscr{C}$)

After proper initialization, PBML algorithm is ready for the main loop of GML, which exploits its implementation of the meta-learner interface.

GetTasks() The procedure generates new ranking only when all the tasks submitted for execution have already been finished (resulting in *taskRunning = 0*) and either the process has just started or the profile has changed since the last time the ranking was built. Configurations listed in the new ranking are pushed to the ordered collection C_Q (called queue) with keys combining the *step* number and the ranks. The value of *step* is introduced to guarantee that the newest ranking, as the most current, is popped first from the queue. When no changes have been done to the profile, since providing the last tasks, it makes no sense to create new ranking as it would be the same as before—the tasks are popped from the queue without new ranking preparation. If all configurations from the newest ranking are popped out, the next newest ranking is served and so on. Finally, each popped

configuration is embedded in the validation scenario and such tasks are returned. The operations are written formally as Algorithm 6.6

Algorithm 6.6 (PBML tasks preparation)

Prototype: PBML.GetTasks()

Input: *None passed directly, but the general PBML parameters (profile manager PM, validation scenario VS, natural number cpi) and internal data structures of PBML are used.*

Output: *Collection of tasks to be run.*

The algorithm:

1. ***if*** *tasksRunning = 0* ***then***
   return $\emptyset$
2. ***if*** *step = 0 or PM.ProfileChanged* ***then***
   a. $C_B \leftarrow$ *PM.CurrentRanking()*
   b. ***for each*** $c \in C_B$ ***do***
      if *c does not occur in* C_R ***then*** *add* $\left(c, step + \frac{rank\ of\ c\ in\ C_B}{length(C_B)}\right)$ *to* C_Q
3. ***if*** $C_Q = \emptyset$ ***then*** *break the loop*
4. $(c_1, \ldots, c_k) \leftarrow$ *try to pop from* C_Q*, up to cpi items with maximum ranks*
5. $tasksRunning \leftarrow k$
6. $step \leftarrow step + 1$
7. ***return*** $\{VS(c_1), \ldots, VS(c_k)\}$

Analyze() The analysis method of PBML is very simple, as it just delegates adequate parts of the functionality to the results calculator (the call *RC(task)* determines the result of the test) and to the profile manager (*PM.RegisterResult(c,r)* adjusts its internal structures according to the result of the test). Algorithm 6.7 presents the method in a more formal way.

Algorithm 6.7 (PBML analysis of finished task)

Prototype: PBML.Analyze(task)

Input: *Test task, just finished, general PBML parameters (profile manager PM, results calculator RC), internal PBML data structures.*

Output: *None.*

The algorithm:

1. $tasksRunning \leftarrow tasksRunning - 1$
2. $r \leftarrow$ *RC(task)*
3. *Add* (c, r) *to* C_R
4. *PM.RegisterResult(c,r)*

ShouldStop The goal of PBML is to search the space of candidate configurations during a pre-specified time period, so as to determine the most accurate machines. There is no premise that would let us regard the search as accomplished and stop it with a fully satisfactory solution. Thus, PBML defines no additional stop criterion and just relies on the main GML loop control.

Result() As usual in meta-learning, the final result may be defined in a couple of ways, but the fundamental objective is to find as accurate learner as possible, so the most natural output is a single learning machine estimated as the most successful (according to the validation result). A natural alternative is to return a ranking of the best scorers and let the addressee of the result message decide what to do next.

6.4.2 Profile Management

It is easy to notice in Algorithms 6.6 and 6.7 that the substantial functionality of the profile manager is exhibited by three members specified in interface 6.8.

Interface 6.8 (Profile manager)

***Method:* RegisterResult(c,r)**
Input: *Machine configuration (c) and test results (r).*
Output: *None.*

***Method:* CurrentRanking()**
Input: *None passed directly, set of machine configurations $\mathscr{C}$ from internal structures.*
Output: *Ranking of machine configurations wrt predicted eligibility for solving the problem.*

***Property:* ProfileChanged**
Output: *Informs whether the profile has changed since last ranking generation.*

Maintenance of the profile is the task of the *RegisterResult()* method. On the basis of the configuration–results pair provided as arguments, profile manager decides about the shape of the profile, that is, when the profile should be modified and which machine configurations should be added to or removed from the profile.

The current machine configuration ranking, built with respect to the eligibility for solving the problem at hand, estimated on the basis of the profile, is returned by the *CurrentRanking()* method. Each time a ranking is generated, a flag is set to notify that the profile has been used to generate a ranking and there is no point in generating next ranking until a call to the *RegisterResult()* method modifies the profile and resets the flag. The information about the state of the flag can be retrieved with the property *ProfileChanged*.

Profile Maintenance

It is not obvious how to create a profile, what should be its size and which configurations should be kept in it. It would be naive to believe that there exists a globally optimal solution to these problems. Moreover, simple strategies like keeping the most successful configurations in the profile are not successful at all. Experiments show that it is more advantageous to care for diversity among the profile configurations. The most successful configurations are often similar to each other and obtain similar results. When the differences between result pairs are small, it is more probable that they are statistically insignificant, so it can be very brittle to derive conclusions from such relations. When the differences between result pairs are large, they are more stable and can carry significant information. Thus, results diversity in profiles is highly recommended.

Usually, when there is no single best method of solving a problem it is advantageous to comply with the rule of divide and conquer. As a result, profile manager can be defined as a general solution, which cedes the crucial decisions to submodules. For example, one of the most intuitive choices is to additionally parameterize PM by:

- the number of configuration–result pairs to keep in the profile,
- the strategy of determining the configurations to remove from the profile, when a new configuration is provided and the profile has already reached its destination size.

Profile adjustment done by such PM may be quite simple: the configuration is added to the profile and if the profile size gets larger than the maximum allowed size, one of the elements is removed from C_P.

According to the remarks made above, that the profile results should be diverse, removing a result from the profile should be done with respect to a diversity measure. When the profile size is not too large, greedy maximization procedures may be acceptable.

Profile-Based Ranking Provision

Providing rankings for a given profile can also be done in a variety of manners. The most natural one is to collect the results obtained with the profile configurations for other data files, determine the most similar profile, or more precisely, the data for which the profile is the most similar, and generate a ranking corresponding to the most successful configurations for that data. Instead of focusing on a single dataset, one can use nearest neighbors methods to select a number of the most similar datasets and then combine the rankings for all these datasets into one final sequence. The similarity measure can also be used to calculate weights defining the strength of influence of particular dataset results on the final scores used for ranking. So, additional fundamental parameters of the PM are:

- profile similarity measure,

- configuration selection strategy (deciding how many and which configurations are included in the ranking),
- knowledge base used for calculations: the collection of datasets $D_1, \ldots, D_k$ and results obtained for these dataset with the methods of interest, that is, a set of functions $f_{D_i} : \mathscr{C} \to \mathscr{R}$.

In practice, it may be very advantageous if the methods of profile adjustment and ranking generation are designed together to interact in their tasks. Therefore, they are enclosed in a single PM object. It can also be very profitable to use the feedback sent to PM not only for profile adjustment, but also for learning how to generate rankings on the basis of the profiles (to exploit the information about how successful the previous rankings were).

Profile Similarity Measures

Comparisons of profiles can be performed with standard similarity/distance measures. Numerous metrics have been tried with the abundance of nearest neighbors methods. They all are perfectly eligible also in this application, so it is possible to exploit the meta-knowledge gathered by many authors during decades of research.

A very important aspect of similarity measurements, in the context of learning results comparisons, is normalization of the compared values. For example, when the results are single real numbers corresponding to measures like classification/approximation error, it would not be sensible to compare raw values of the errors, as the optimal error values for different datasets may be quite distant although relations between the scores can be very informative. We need to compare the shapes of the profiles (when plotted as lines in the style of parallel coordinates), not the values themselves. Therefore, measures like the Pearson's linear correlation coefficient or cosine similarity measure seem very suitable for this purpose. However, any vector similarity measure can also perform well, if applied after a method of data normalization.

In the domain of similarity-based methods, many techniques of feature selection and feature weighting have been profitable in numerous applications. The experience of the field can also be helpful in profile similarity measurement. Moreover, the techniques of feature selection (or weighting) can bring much help to profile maintenance, as the selection (or weights) nicely suggest which scores in the profiles are of little benefit or maybe even spoil the information exhibited by other features. A lot of research may still be done in this area.

6.4.3 Ranking-Based Meta-Search

Advantages of a meta-learning approach like PBML can only be appreciated when analyzed on the background of other similar techniques. Since the PBML approach is an active ranking method, some baseline approaches to compare with, are *passive*

rankings, that is, simple rankings generated according to some analysis of the results obtained by the same methods for other datasets, with no on-line adjustment to the feedback from tests performed for the data currently being learned.

In the case of passive ranking methods, meta-learning based on the rankings is in fact a step-by-step validation of machine configurations with subsequent ranks. After collecting validation results, the meta-search can return machine configurations in new order, corresponding to the validation results.

Validation of passive rankings is much simpler then advanced, active meta-learning methods, but it fits the general scheme of the GML algorithm. Therefore, it can be implemented in the same framework, by definition of the four members of the meta-learner interface 6.2. From the perspective of GML, passive ranking validation (PRV) is a lean example, as it submits all the tasks in the first call of *GetTasks()* and performs no meta-level analysis apart from simple results collection and ordering.

Specification of all the details of the PRV machine, requires providing the following additional parameters:

- *validation scenario VS* defining how tests are performed,
- *result calculator RC* that determines the final results on the basis of subsequent tests.

Validated ranking is constructed during the whole ML process, when feedback informs about each finished test separately, so the PRV machine uses an internal data structure C_R to collect the final ranking. It must be initialized with the instruction $C_R \leftarrow \emptyset$, before the main algorithm starts.

Precise information about the main functionality of PRV as a meta-learner is provided in the following members descriptions.

GetTasks() The method is presented as Algorithm 6.9. It is very simple, as it just generates the ranking and returns tests of all ranked configurations, embedded in the validation scenario *VS*.

Algorithm 6.9 (Passive ranking tasks preparation)

Prototype: ***PassiveRanking.GetTasks()***

Input: *None passed directly, PassiveRankingMethod and VS given as a part of machine configuration.*

Output: *Collection of tasks to be run.*

The algorithm:

1. ***if*** $C_R \neq \emptyset$ ***then*** */* check if not the first call */*
   return $\emptyset$
2. $(c_1, \ldots, c_k) \leftarrow$ *PassiveRankingMethod()*
3. ***return*** $\{VS(c_1), \ldots, VS(c_k)\}$

The operations are performed only in the first call to the method, which is controlled by the test performed at the beginning: second call to the method is made after a call

to *Analyze()*, where an item is added to C_R, so the condition $C_R \neq \emptyset$ means that it is not the first run.

Analyze() By definition, passive rankings do not use the feedback coming from the tests performed for the sake of ranking validation. Therefore, no analysis of the results is necessary, beside calculating the results (with *RC*) and registering them in the final, validated ranking C_R, as in algorithm 6.10.

Algorithm 6.10 (Passive ranking analysis of finished task)

Prototype: PBML.Analyze(task)
Input: *Test task, just finished, results calculator RC as machine parameter, internal data.*
Output: *None.*
The algorithm:

1. $r \leftarrow RC(task)$
2. *Add* (c, r) *to* C_R

ShouldStop Similarly to PBML, passive-ranking validation does not use any additional stop criterion. It relies on the time-limit control performed within the GML algorithm.

Result() Depending on the needs, the final result may get various forms, but usually it is the validated ranking C_R or just the winner machine configuration, if the goal is to find the best single method.

Because of the time limit, it is important in what order the candidate machine configurations are tested. The better the ranking returned by the *PassiveRankingMethod()*, the higher result can be obtained within given time.

The scheme of passive ranking validation can also be used to get the absolute baseline algorithm performing validations completely at random, which can be seen as validation of a randomly generated passive ranking.

6.4.3.1 Ranking Criteria

Many ranking algorithms have been proposed and can be found in the literature. Some examples of the approaches are presented in Sect. 6.1.4. Unfortunately, many of them are not adequate for advanced ML applications, because of their computational complexity, making them feasible only for small numbers of machine configurations. The algorithms performing pairwise comparisons, like ARR with RL , SRR or SW must be regarded as unfeasible, when the number of candidate machine configurations is tens of thousands and more than several datasets are considered. They are practical only for analysis of several algorithms. For larger sets of machine configurations, even if they are still feasible, they are not suitable for meta-learning

by ranking validation, because calculating input ranking, could consume most of the time allotted to the process. In comparisons to other methods, it would bring conclusions, highly disadvantageous for these methods—in fact, others could finish all validations before these ones could start the first one. If we ignored this time, then the comparisons (done with respect to time) would not be fair. Because of that, the passive rankings included in the plots depicting the experiments of Sect. 6.4.5, are based on less computationally complex formulae, presented below.

Let the knowledge base used for meta-learning contain n datasets. For each candidate configuration c, let's define $Acc_c(i)$ as the accuracy of the machine c tested on i'th dataset and $\sigma_c(i)$ as the standard deviation of this accuracy estimated in the test.

The index of *average test accuracy* is given simply by:

$$\mathrm{AA}(c) = \frac{1}{n}\sum_{i=1}^{n} Acc_c(i). \tag{6.19}$$

Since the accuracies for different datasets may have quite different scopes, it seems more sensible to calculate *average difference* between the test accuracies of the method of interest, and the best method in the examined population, expressed in the units of standard deviations of the best method:

$$\mathrm{AD}(c) = \frac{1}{n}\sum_{i=1}^{n} \frac{Acc_c(i) - Acc_{Best(i)}(i)}{\sigma_{Best(i)}(i)}. \tag{6.20}$$

Notation $Best(i)$, used above, stands for a configuration that has reached the highest accuracy for the ith dataset. Such definition is not fully precise, because there may be more than one configuration with maximum accuracy, but in the tests described below, when such situations happened, all the machines of maximum accuracy, gave the same variance of the results, so the ambiguity did not appear.

Another possibility to make the results of different scopes comparable is to calculate average ranks of the methods:

$$\mathrm{AR}(c) = \frac{1}{n}\sum_{i=1}^{n} Rank(c, i), \tag{6.21}$$

where $Rank(c, i)$ is the rank of configuration c in the ordered results for the ith dataset.

Yet another sensible ranking can be made on the basis of average p-values $p(c, i)$ of the paired t-test comparing the results of the method of interest and those of the best method in the population:

$$\mathrm{AP}(c) = \frac{1}{n}\sum_{i=1}^{n} p(c, i). \tag{6.22}$$

Experiments of meta-learning are usually performed for not large numbers of datasets. Under such circumstances, usually the technique of leave-one-out is selected, which, on one hand, is quite natural or even seems the only sensible method, but on the other hand, suffers from serious disadvantages: as argued by Kohavi (1995), in classification tasks, when classes are represented by equal numbers of objects and we take one object out of the population, the class of the object becomes a minority in the population, and classification on the basis of majorities gets completely wrong. In meta-learning, we observe the same phenomenon: the datasets "similar" to the one taken out, become a minority and get dominated by the datasets for which other methods are more successful, so the rankings get less appropriate. To solve this problem, weighting or nearest neighbors methods may be used to select "similar" datasets before averaging particular indices.

6.4.4 Comparing Rankings of Algorithms

Meta-learning based on search for the most adequate algorithms for given problem results in a ranking of algorithms. Even if the formal output of a method is a single learning machine, selected as the winner, the learning process has assessed a set of solutions and compared them with a criterion to select the best one, so in fact, it has created a ranking. This is a property common to all algorithms compatible with the GML approach, so in particular, also PBML, passive ranking validation and random machine validation methods.

Comparisons of the rankings output by the ML algorithms, should reflect what we expect from the rankings, but the problem is that the expectations from meta-learning search processes are not so easy to express. In general we want the procedures to find as attractive learning machines as possible in as short time as possible, but this formulation is imprecise. The rankings should include the best solutions at the beginning and then alternative models with as high accuracy as possible, in the decreasing order of their quality.

In the literature, reviewed in Sect. 6.1.4, one can find many approaches to rankings, and rankings evaluation usually made by comparison to the ideal rankings with some statistical tools like correlation coefficients. Although the strategy suits experiments involving several learning algorithms, it is completely inadequate for comparing long rankings of algorithms. When several algorithms are in focus, it is important to preserve each relation between the pairs of algorithms. When thousands of algorithms are available, and an ML algorithm is to find the best ones in a limited time, full rankings should not be compared item by item, from top to bottom. The focus should be on the top-ranked items, to estimate how attractive model(s) can be found in different time periods. In most cases, we are not interested in the similarity of the whole rankings but in obtaining one or several models of as high quality as possible. For example, when we search for an accurate classifier, the meta-search should find a single classifier of maximum accuracy and it is not important how accurate are

the remaining models tested by the algorithm and at which position they have been ranked.

Sometimes, we can get n top-ranked machines provided by the meta-search and create a committee, or just be supplied with n alternative learners. Then, it gets important how accurate are the n top-ranked machines, but not if other machines in the ranking are accurate. In fact, when n machines are selected to create a committee or to build other kind of complex model, the task should also be commended to the meta-search process, so that the ensembles could also be validated and possibly returned by the algorithm, if better than other machines. Thus, even the restriction of $n = 1$ is quite acceptable.

In all above-mentioned applications, it should be regarded as much more costly to put the best algorithm on position 100 of the ranking, than to switch positions 10000 and 20000. The most popular, also in the domain of meta-learning, ranking comparison techniques do not address such requirements and pay the same attention to each part of the ranking.

Meta-learning conceived as search for the best solutions, requires results analysis to answer the question, how accurate model (the best one) can be found in a given time. Complying with this idea, the comparisons presented below are focused most on plots of maximum accuracy (estimated in the validation process) obtained till given time.

6.4.5 Experiments, Results and Analyses

Reliable test of the PBML framework required a nontrivial knowledge base of learning-machines results. Comparative experiments in the realm of cross-validation committees of DTs, presented in Sect. 5.4, resulted in collecting a database of 10×10-fold CV test results for 13660 different machine configurations and 21 datasets. It is perfectly suitable for miscellaneous meta-learning experiments, so it was chosen as the knowledge base for evaluation of the PBML approach.

Datasets

The 21 datasets come from the UCI repository (Frank and Asuncion 2010) and are summarized in Table 5.1.

The count of 21 datasets is not large—the probability that almost each of them has different nature and requires different methods is quite large. It makes meta-learning difficult, but turned out to be sufficient to show some relations between meta-learners results.

Learning Machines

The 13660 different machine configurations, mentioned above, are 100 parameter settings of single decision tree induction methods and 13560 different settings

of cross-validation committees. The committees were constructed from validated decision trees generated in 10-fold cross-validation processes. Hence, the number of cooperating models could be set in the range from 1 to 10. Moreover, different configurations were obtained by application of four different methods for decision tree induction: two methods based on purity gain criterion with Gini index and entropy (information gain), QUEST and the algorithm based on Separability of Split Value value criterion. Several methods of DT validation have also been tested: Reduced Error Pruning (REP REP), cost-complexity optimization cost-complexity minimization (CC), degree-based tree validation , OPTimal pruning, Minimum Error Pruning 2 and Depth Impurity . Some other parameters like respecting standard error in pruning strength determination, combining validation and training error in pruning criteria, common or separate parameter optimization in subsequent folds of CV, were also examined, resulting in the final number of 13560 CV committee configurations. More information about the parameters and the overall experiment settings, can be found in Sect. 5.4.

Each configuration had been tested with 10×10-fold CV using the same training-test data splits. As a consequence, a nice database of results had been collected and is perfectly suitable for meta-learning experiments.

6.4.5.1 PBML Versus Random and Passive Rankings

To test the possibilities of the PBML framework, the exchangeable general modules like VS and PM had to be precisely defined and implemented.

Because the task concerned classification, the validation scenario was defined to perform 10×10-fold cross-validation and estimate average accuracy of the tests. Such definition was compatible with the experiments that generated the knowledge base, so the CV tests could be just reused instead of recalculating them.

Ranking Methods

In the experiments presented here, profile management has been specified to maintain profiles of variable size. At the beginning of the search process, the profile was empty and grew up during the process. Each time a new configuration was validated, its result was calculated and added to the profile. For readability, the experiments were restricted to 100 configurations. Because of that, it was not a problem to keep and handle full profiles, especially that the algorithms of profile management used here were very efficient.

In Algorithm 6.6, the first ranking is created when the profile is empty, so ranking generation procedure must be ready for that. Also, when the profile contains just one configuration, it is not very useful, because relative differences are the base of machine quality predictions. Therefore, profiles of size less than 2, are not helpful in ranking generation. In the experiments described here, the initial PBML ranking was calculated on the basis of average p-values (6.22). The *cpi* parameter was set to 5, so the second call to ranking generation was given a profile of 5 results, and each

next time the profile was larger. Rankings based on profiles, were calculated with respect to weighted p-values for candidate configurations:

$$WPV(c) \leftarrow \sum_{D \in KB} Max(0, CC(P, D)) * PV(c, D), \tag{6.23}$$

where:

- $D \in KB$ means "dataset D in the knowledge base",
- $CC(P, D)$ is the Pearson linear correlation coefficient calculated for the results in the profile and corresponding results extracted from the knowledge base for dataset D,
- $PV(c, D)$ is the p-value obtained in paired t-test of difference significance, between the results obtained by c and the best configuration recorded for dataset D.

Pearson linear correlation coefficient serves here as a measure of similarity between the profile and corresponding results obtained for other datasets. The similarity factor is replaced by 0, when the correlation is negative (hence Max in the formula), to ignore the results of utterly dissimilar datasets (instead of accepting penalties for obtaining high accuracy on such data).

Results and Analysis

In the first comparison, the results obtained with the PBML algorithm have been confronted with the results of 5 passive ranking methods: completely random one and 4 implementing formulae (6.19)–(6.22).

The test has been conducted as the leave-one-out procedure: for each of the 21 datasets, the dataset was removed from the knowledge base to be used for testing, while the remaining 20 datasets served for rankings generation.

The most important aspect of the comparison is to check what maximum validation accuracy can be obtained by each method in given time. Since the time consumed by the competing methods was more or less equal for all configurations, we can use more readable time unit than seconds: the number of configurations validated so far. Each method was run to generate a ranking of 100 configurations (for each of the 21 datasets), and the averages of the results are presented in Fig. 6.17. Detailed results for each step of the leave-one-out procedure can be found in an external document available from http://www.is.umk.pl/~kg/papers/12-PBMLRes.pdf.

The most important plot is placed at the top. For each dataset, for each ranking, for each argument t in the scope from 1 to 100, the maximum accuracy obtained by one of the t top-ranked machines was determined. The results for all 21 passes were first converted to the unit of standard deviations of the best result recorded for the dataset and then averaged. The means for $t = 1, \ldots, 100$ became subsequent points in the line corresponding to given ranking. Function values shown on the left of the vertical axis show maximum and minimum of all the points presented in the plot. The legend presents which line corresponds to which ranking. The same line colors have been used in all plots.

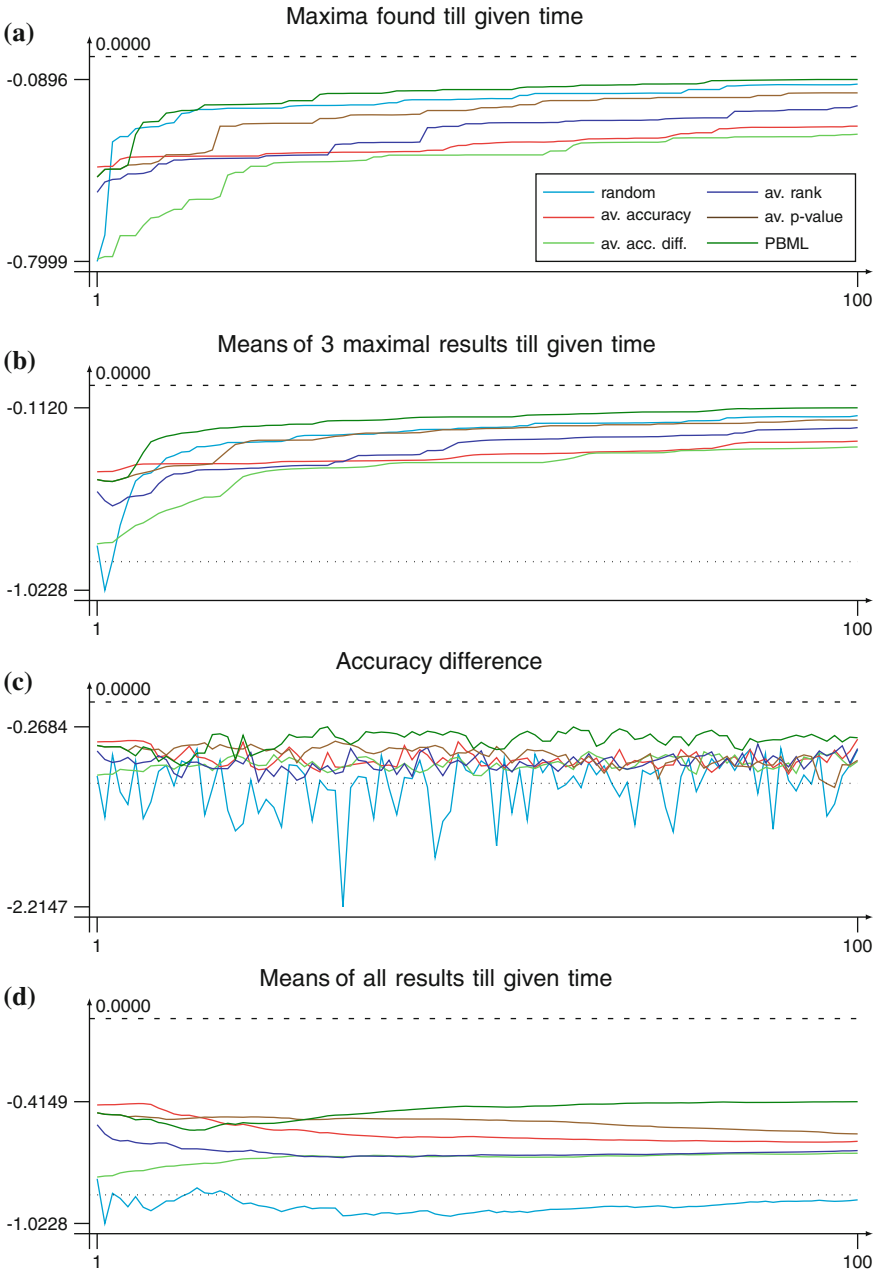


Fig. 6.17 Results obtained in the experiment, averaged over the 21 datasets. Arguments—time (the number of configurations tested), values—average accuracy difference in the units of standard deviations of the best results observed for particular datasets. **a** Maxima found till given time. **b** Means of 3 maximal results till given time. **c** Accuracy difference. **d** Means of all results till given time

What can be observed in the topmost plot is a bit surprising: the place of the curve corresponding to completely random ranking is unexpectedly attractive in relation to other rankings. Only the line representing PBML ranking occurs higher than the one for random ranking (after the initial 5 steps in which the p-value-based ranking was used and no PBML mechanisms had yet a chance to act). The random ranking picks quite attractive solutions from time to time, so it managed to outperform (in the topmost, maxima plot) all other rankings but PBML. As can be noticed in the third plot (counting from the top), presenting just average accuracies at each step (instead of "maxima till the step" of the first plot), the random ranking is characterized by the largest variance and significantly lower mean than any other ranking. Averaged means of all the results recorded up to given step t are presented in the bottom plot of the figure. It clearly shows that the random ranking obtains the lowest mean accuracy, which according to the law of large numbers, goes to the overall mean, marked in the plots as the dotted horizontal line. Another advantage of the PBML approach can also be seen in the bottom plot: the method keeps high average accuracy with increasing number of configurations validated. In fact, the curve is growing to the end of the scope, so it confirms that for the whole range of the plot, active profiles bring new information, helpful in finding yet more attractive solutions.

If we expected three attractive models to be output by the ranking algorithms, the second plot from top, would be more adequate. It shows means of three best results obtained till subsequent steps. In this plot, the lines of random and p-value-based ranking almost overlap, showing that the attractiveness of the random selection decreases. At the same time, the advantage of PBML over the random ranking gets even larger. Actually, the relations between all other lines are preserved, just the random ranking result is less attractive, which is not surprising.

As an explanation of why averaging over all 20 training datasets produces rankings loosing with the random one, it can be stressed again, that in the group of 21 datasets, the datasets are, in general, quite variant, so the collections of most accurate machines are also quite different. Averaging over 20 datasets yields rankings of machines, where top places are taken by machines that perform well on average, but not perfectly on any of the datasets. This property transfers to the data left out, so that we can clearly see in the two bottom plots in Fig. 6.17, that the configurations selected with ranking methods provide quite good results in the tests: all the lines of non-random rankings are above the average at almost whole range of the plots. There are very few exceptions, showing that with safe strategies we can find quite stable, solid results, but to find the best machine configurations (or better prediction), more advanced means must be undertaken.

6.4.5.2 Passive Rankings, Nearest Neighbors Selection and Active Profiles

Passive rankings based on averaging results globally, can be significantly improved to win competitions with the random ranking, also in the case of registering single maxima. One of the solutions is to engage a nearest neighbors technique to select datasets within the knowledge base, being the most similar to the test data, and then calculate

the averages for the selected subpopulation of datasets. A reasonable way to describe the datasets for nearest neighbor analysis is landmarking with selected machines of the examined population. It has been shown that landmarking performs better than describing the data by statistical and information theory measures. Machines from the population are the most representative of the population, so there is no point in searching for other candidates for landmarkers.

In the approach presented here, simple k nearest neighbors algorithm was used with Euclidean distance measure to calculate distances between profiles (more precisely: squares of the distances, to simplify calculations while introducing no changes to the NN results). The distances determined five closest datasets (5NN) and the results obtained for these datasets were averaged to get the quality estimation for candidate machine configurations. Four quality measures were applied: classification accuracy, accuracy differences expressed in the units of standard deviations of the best result, ranks resulting from ordering by test accuracy and p-values from t-test of statistical significance of differences.

The GML framework is so general, that made it easy to implement the algorithms of passive landmarking with kNN selection, too. The active parts of the general algorithm got deactivated by dummy implementation of adequate methods of the profile manager (as in the case of ordinary passive rankings, no changes to the profile are necessary and no new rankings need to be generated). Common implementation of all the methods within the same framework in Intemi helped in preparation of fair conditions for the comparisons.

The overall results of the comparison between the passive methods, their modifications by kNN-based selection and algorithms with active profiles, are presented in Fig. 6.18. Detailed results for each dataset can be found in a document available from http://www.is.umk.pl/~kg/papers/12-PBMLRes.pdf. The active profile methods started with randomly selected machine configurations in the role of the profile, to calculate the first distances and select neighbors for the first ranking (as mentioned above, when first ranking is generated the profile is still empty). Subsequent distance calculations and rankings generation were performed normally, with actively extended profiles according to the general PBML algorithm.

To make visual analysis of Fig. 6.18 easier, all methods using accuracy as the criterion, got red color, those using accuracy difference measured in standard deviations—green, the three lines based on average ranks—blue, and the three representing methods that average out p-values—brown. Whether the line corresponds to a passive ranking, to a passive ranking augmented by nearest neighbors (5NN) selection or to an active profile based method, can be recognized by line pattern, adequately: dotted, dashed and solid. Additionally, the lines shown in cyan, represent random ranking results.

All four aspects of the analysis clearly show that the PBML methods, based on active profiles, provide the most attractive results. In the topmost plot, each solid line lies above the cyan one within almost whole scope, proving that active profile management can significantly fasten finding attractive results. At the same time, they keep high results all over the scope, providing large mean accuracies, so in contrary

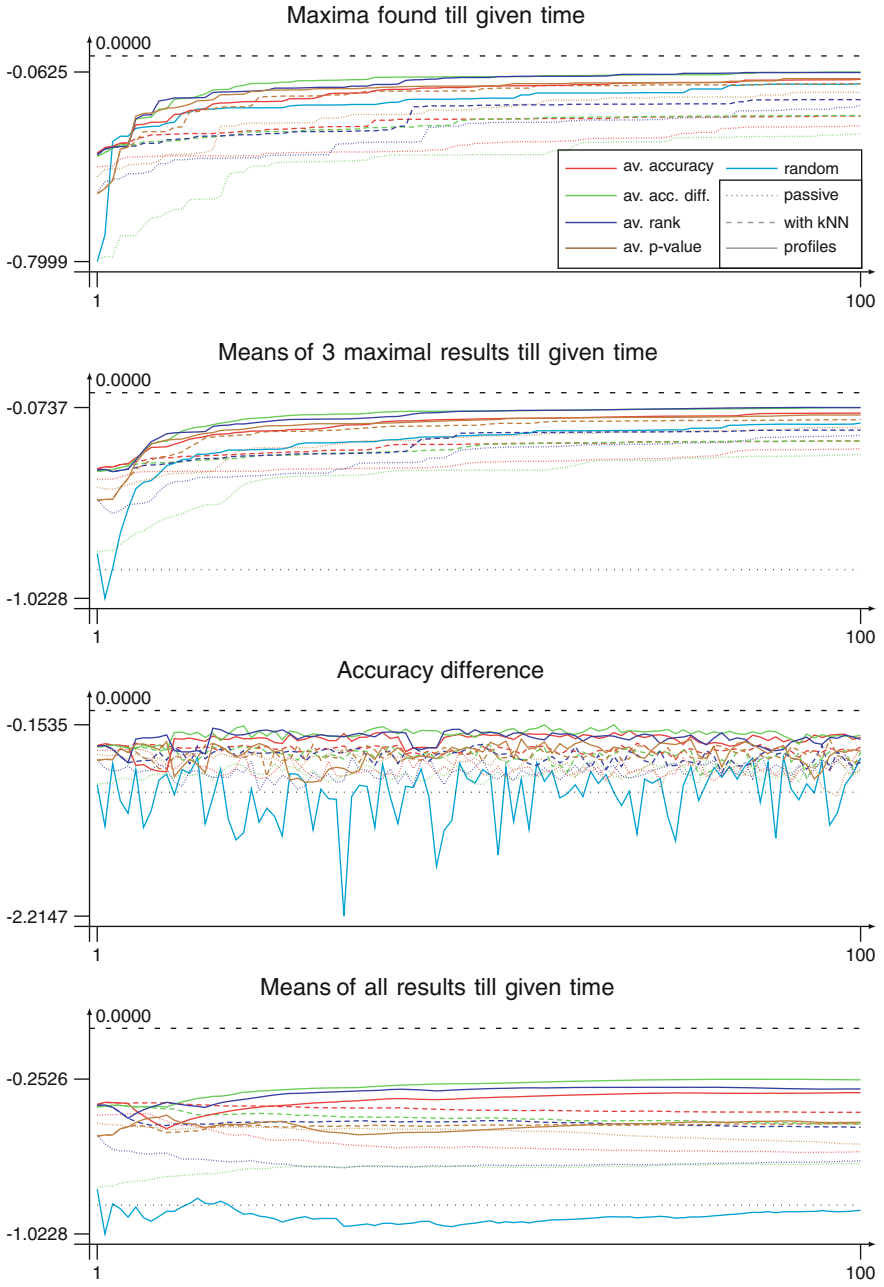


Fig. 6.18 Comparison of passive ranking methods (*dotted lines*), passive ranking with nearest neighbors selection (*dashed lines*) and active profile based methods (*solid lines*). **a** Maxima found till given time. **b** Means of 3 maximal results till given time. **c** Accuracy difference. **d** Means of all results till given time

to random ranking they detect appropriate machine configurations and find accurate models not by chance, but by meta-level analysis.

Calculating average results for datasets selected by 5NN also significantly outperforms the classical passive methods, but does not provide as attractive results as using active profiles. The line of NN-supported p-values on the maxima plot is as attractive as some active methods. The plots of accuracy difference and means clearly show that the improvement introduced by 5NN is significant.

References

Abe H, Yamaguchi T (2004) Constructive meta-learning with machine learning method repositories. In: Proceedings of the innovations in applied artificial intelligence, pp 502–511

Baxter J (2000) A model of inductive bias learning. J Artif Intell Res 12:149–198

Bengio Y (2009) Learning deep architectures for AI. Found Trends Mach Learn 2(1):1–127 (Also published as a book in Now Publishers, 2009)

Bensusan H (1999) Automatic bias learning: an inquiry into the inductive basis of induction. PhD thesis, School of Cognitive and Computing Sciences, University of Sussex. http://www.cs.bris.ac.uk/Publications/Papers/1000410.pdf

Bensusan H, Giraud-Carrier C (2000) Casa batlo is in passeig de gracia or landmarking the expertise space. In: Proceedings of the ECML'2000 workshop on meta-learning: building automatic advice strategies for model selection and method combination, ECML'2000, pp 29–47. http://www.cs.bris.ac.uk/Publications/Papers/1000470.pdf

Bensusan H, Giraud-Carrier C, Kennedy CJ (2000) A higher-order approach to meta-learning. In: Cussens J, Frisch A (eds) Proceedings of the work-in-progress track at the 10th international conference on inductive logic programming, pp 33–42

Bensusan H, Kalousis A (2001) Estimating the predictive accuracy of a classifier. In: Lecture notes in computer science, vol 2167, pp 25–31

Bernstein A, Provost F, Hill S (2005) Toward intelligent assistance for a data mining process: an ontology-based approach for cost-sensitive classification. IEEE Trans Knowl Data Eng 17(4):503–518

Berrer H, Paterson I, Keller J (2000) Evaluation of machine-learning algorithm ranking advisors. In: Brazdil P, Jorge A (eds) Proceedings of the PKDD-00 workshop on data mining, decision support, meta-learning and ILP: forum for practical problem presentation and prospective solutions. Springer, Lyon

Blockeel H, Raedt LD, Ramon J (1998) Top-down induction of clustering trees. In: Proceedings of the 15th international conference on machine learning. Morgan Kaufmann, pp 55–63

Brazdil P, Giraud-Carrier CG, Soares C, Vilalta R (2009) Cognitive technologies, metalearning: applications to data mining. Springer, New York

Brazdil P, Soares C (2000a) A comparison of ranking methods for classification algorithm selection. In: López de Mántaras R, Plaza E (eds) Machine learning: ECML 2000. Lecture notes in computer science, vol 1810. Springer, Berlin, pp 63–75. http://dx.doi.org/10.1007/3-540-45164-1_8

Brazdil P, Soares C (2000b) Ranking classification algorithms based on relevant performance information. In: Proceedings of the European conference on machine learning and principles and practice of knowledge discovery in databases

Brazdil P, Soares C, Costa JPD (2003) Ranking learning algorithms: using IBL and meta-learning on accuracy and time results. Mach Learn 50(3):251–277. http://dx.doi.org/10.1023/A:1021713901879

Breiman L (1996) Bagging predictors. Mach Learn 24(2):123–140

Breiman L (1998) Arcing classifiers. Ann Stat 26(3):801–849

Chan P, Stolfo SJ (1993) Toward parallel and distributed learning by meta-learning. In: Proceedings of the AAAI workshop in knowledge discovery in databases, pp 227–240
Chan P, Stolfo SJ (1996) On the accuracy of meta-learning for scalable data mining. J Intell Inf Syst 8:5–28
Cohn D, Atlas L, Ladner R (1994) Improving generalization with active learning. Mach Learn 15(2):201–221. http://www.springerlink.com/index/10.1007/BF00993277
Cornelson M, Grossman RL, Greengrass E, Karidi R, Shnidman D (2003) Combining families of information retrieval algorithms using metalearning. In: Berry MW (ed) Survey of text mining: clustering, classification, and retrieval. Springer, New York
Cpałka K, Er MJ, Rutkowski L (2008) New methods for designing neuro-fuzzy systems. In: Proceedings of the 12th WSEAS international conference on systems, pp 575–580
de Souto M, Prudêncio R, Soares R, de Araujo D, Costa I, Ludermir T, Schliep A (2008) Ranking and selecting clustering algorithms using a meta-learning approach. In: Proceedings of the IEEE international joint conference on neural networks (IJCNN), part of world IEEE congress on computational intelligence, pp 3729–3735
Dietterich TG (1999) An experimental comparison of three methods for constructing ensembles of decision trees: bagging boosting and randomization. Mach Learn 40:1–22
Duch W, Grudziński K (2001) Meta-learning: searching in the model space. In: Proceedings of the international conference on neural information processing (ICONIP), Shanghai, pp 235–240
Duch W, Grudziński K (2002) Meta-learning via search combined with parameter optimization. In: Rutkowski L, Kacprzyk J (eds) Advances in soft computing. Physica/Springer, New York, pp 13–22
Duch W, Itert L (2003) Committees of undemocratic competent models. In: Proceedings of the joint international conference on artificial neural networks (ICANN) and international conference on neural information processing (ICONIP), Istanbul, Turkey, pp 33–36
Engels R, Theusinger C (1998) Using a data metric for preprocessing advice for data mining applications. In: Proceedings of the European conference on artificial intelligence (ECAI-98). Wiley, Chichester, pp 430–434
François D, Wertz V, Verleysen M (2011) Choosing the metric: a simple model approach. In: Jankowski N, Duch W, Grąbczewski K (eds) Meta-learning in computational intelligence. Studies in computational intelligence, vol 358. Springer, Berlin, pp 97–115. http://dx.doi.org/10.1007/978-3-642-20980-2_3
Frank A, Asuncion A (2010) UCI machine learning repository. http://archive.ics.uci.edu/ml
Freund Y, Schapire RE (1996) Experiments with a new boosting algorithm. In: Proceedings of the 13th international conference on machine learning
Fürnkranz J, Petrak J (2001) An evaluation of landmarking variants. In: Lavra N, Moyle S, Kavsek B, Giraud-Carrier C (eds) Proceedings of the ECML/PKDD workshop on integrating aspects of data mining, decision support and meta-learning
Fürnkranz J, Petrak J (2002) Extended data characteristics. Technical report, METAL-consortium
Fürnkranz J, Petrak J, Brazdil P, Soares C (2002) On the use of fast subsampling estimates for algorithm recommendation. Technical report, Österreichisches Forschungsinstitut für Artificial Intelligence
Giraud-Carrier C (2008) Metalearning: a tutorial. In: Proceedings of the 7th international conference on machine learning and applications (ICMLA'08)
Giraud-Carrier CG (2005) The data mining advisor: meta-learning at the service of practitioners. In: Kurgan LA, Reformat M, Hafeez K, Wani MA, Milanova MG (eds) Proceedings of the ICMLA. IEEE Computer Society
Grąbczewski K, Jankowski N (2007) Meta-learning architecture for knowledge representation and management in computational intelligence. Int J Inf Technol Intell Comput 2(2):27
Grąbczewski K, Jankowski N (2011) Saving time and memory in computational intelligence system with machine unification and task spooling. Knowl.-Based Syst 24:570–588. http://dx.doi.org/10.1016/j.knosys.2011.01.003
Grąbczewski K, Jankowski N, Duch W (2004) GhostMiner 3.0. FQS Poland, Kraków, Poland

Guyon I, Saffari A, Dror G, Cawley G (2010) Model selection: beyond the Bayesian/frequentist divide. J Mach Learn Res 11:61–87. http://dl.acm.org/citation.cfm?id=1756006.1756009

Hilario M (2002) Model complexity and algorithm selection in classification. In: Lange S, Satoh K, Smith C (eds) Discovery science. Lecture notes in computer science, vol 2534. Springer, Berlin, pp 113–126. http://dx.doi.org/10.1007/3-540-36182-0_12

Hilario M, Kalousis A, Nguyen P, Woznica A (2009) A data mining ontology for algorithm selection and meta-learning. In: Proceedings of the ECML/PKDD09 workshop on third generation data mining: towards service-oriented knowledge discovery (SoKD-09), pp 76–87

Hilario M, Nguyen P, Do H, Woznica A, Kalousis A (2011) Ontology-based meta-mining of knowledge discovery workflows. In: Jankowski N, Duch W, Grąbczewski K (eds) Meta-learning in computational intelligence. Studies in computational intelligence, vol 358. Springer, Berlin, pp 273–315. http://dx.doi.org/10.1007/978-3-642-20980-2_9

Hinton GE, Osindero S, Teh YW (2006) A fast learning algorithm for deep belief nets. Neural Comput 18(7):1527–1554. http://dx.doi.org/10.1162/neco.2006.18.7.1527

Jankowski N (1995) Applications of Levin's universal optimal search algorithm. In: Kącki E (ed) System modeling control'95, vol 3. Polish Society of Medical Informatics, Łódź, Poland, pp 34–40

Jankowski N, Grąbczewski K (2005) Heterogenous committees with competence analysis. In: Nedjah N, Mourelle L, Vellasco M, Abraham A, Köppen M (eds) Proceedings of the 5th international conference on hybrid intelligent systems. IEEE Computer Society, Rio de Janeiro, pp 417–422

Jankowski N, Grąbczewski K (2011) Universal meta-learning architecture and algorithms. In: Jankowski N, Duch W, Grąbczewski K (eds) Meta-learning in computational intelligence. Studies in computational intelligence, vol 358. Springer, Berlin, pp 1–76

Janssen F, Fürnkranz J (2007) On meta-learning rule learning heuristics. In: Proceedings of the ICDM, pp 529–534

Janssen F, Fürnkranz J (2010) On the quest for optimal rule learning heuristics. Mach Learn 78:343–379. http://dx.doi.org/10.1007/s10994-009-5162-2

Johansson U (2007) Obtaining accurate and comprehensible data mining models: an evolutionary approach. Doctoral thesis, Department of Computer and Information Science, Linkoping University, SE-581 83, Linkoping, Sweden. http://hdl.handle.net/2320/2136

Kacprzyk J, Zadrożny S (2007) Towards human-consistent data-driven decision support systems via fuzzy linguistic data summaries. In: Batyrshin I, Kacprzyk J, Sheremetov L, Zadeh L (eds) Perception-based data mining and decision making in economics and finance. Studies in computational intelligence, vol 36. Springer, Berlin, pp 37–54. http://dx.doi.org/10.1007/978-3-540-36247-0_1

Kadlec P, Gabrys B (2008) Learnt topology gating artificial neural networks. In: Proceedings of the IEEE world congress on computational intelligence. IEEE Press, pp 2605–2612

Kalousis A (2002) Algorithm selection via meta-learning. PhD thesis, University of Geneve

Kalousis A, Hilario M (2000) Model selection via meta-learning: a comparative study. In: Proceedings of the 12th IEEE international conference on tools with artificial intelligence (ICTAI'00). IEEE Computer Society, Vancouver, pp 406–413

Kalousis A, Hilario M (2001) Feature selection for meta-learning. In: Advances in knowledge discovery and data mining, pp 222–233

Kalousis A, Hilario M (2003) Representational issues in meta-learning. In: Proceedings of the 20th international conference on machine learning (ICML-2003), Washington DC, vol 20, pp 313–320

Kalousis A, Theoharis T (1999) NOEMON: an intelligent assistant for classifier selection

Kietz JU, Serban F, Bernstein A, Fischer S (2012) Designing kdd-workflows via htn-planning for intelligent discovery assistance. In: Vanschoren J, Kietz JU, Brazdil P (eds) Proceedings of the CEUR workshop on planning to learn 2012, Workshop at ECAI 2012

Kohavi R (1995) Wrappers for performance enhancement and oblivious decision graphs. PhD thesis, Stanford University

Kohonen T (1986) Learning vector quantization for pattern recognition. Technical report TKK-F-A601, Helsinki University of Technology, Espoo, Finland

Kolmogorov AN (1965) Three approaches to the quantitative definition of information. Prob Inf Trans 1:1–7

Köpf C, Taylor CC, Keller J (2000) Meta-analysis: from data characterisation for meta-learning to meta-regression. In: Proceedings of the PKDD-00 workshop on data mining, decision support, meta-learning and ILP: forum for practical problem presentation and prospective solutions, Lyon, France

Kosiński W, Dziczkowski G, Golénia B, Węgrzyn-Wolska K (2010) Towards an optimal decision support system. In: Devlin G (ed) Advances in decision support systems, pp 299–324. http://www.intechopen.com/books/decision-support-systems-advances-in/towards-an-optimal-decision-support-system

Leite R, Brazdil P, Vanschoren J (2012) Selecting classification algorithms with active testing. In: Perner P (ed) Machine learning and data mining in pattern recognition. Lecture notes in computer science, vol 7376. Springer, Berlin, pp 117–131. http://dx.doi.org/10.1007/978-3-642-31537-4_10

Li M, Vitányi P (1993) An introduction to Kolmogorov complexity and its applications: text and monographs in computer science. Springer, New York

Lindner G, Ag D, Studer R (1999) Ast: support for algorithm selection with a cbr approach. In: Recent advances in meta-learning and future work, pp 418–423

Mierswa I, Wurst M (2005) Efficient feature construction by meta learning: guiding the search in meta hypothesis space. In: Proceedings of the international conference on machine learning, workshop on meta-learning, pp 84–92

Morik K, Scholz M (2004) The miningmart approach to knowledge discovery in databases. In: Intelligent technologies for information analysis. Springer, Heidelberg, pp 47–65

Peng Y, Flach PA, Soares C, Brazdil P (2002) Improved dataset characterisation for meta-learning. In: DS '02: proceedings of the 5th international conference on discovery science. Springer, London, pp 141–152

Pfahringer B, Bensusan H, Giraud-Carrier C (2000) Meta-learning by landmarking various learning algorithms. In: Proceedings of the 17th international conference on machine learning. Morgan Kaufmann, pp 743–750

Prodromidis A, Chan P (2000) Meta-learning in distributed data mining systems: issues and approaches. In: Kargupta H, Chan P (eds) Book on advances of distributed data mining. AAAI, Cambridge

Prudêncio R, Ludermir TB (2008) Active meta-learning with uncertainty sampling and outlier detection. In: Proceedings of the IEEE international joint conference on neural networks (IJCNN), part of world IEEE congress on computational intelligence, pp 346–351

Prudêncio RBC, Souto MCP, Ludermir TB (2011) Selecting machine learning algorithms using the ranking meta-learning approach. In: Jankowski N, Duch W, Grąbczewski K (eds) Meta-learning in computational intelligence. Studies in computational intelligence, vol 358. Springer, Berlin, pp 225–243. http://dx.doi.org/10.1007/978-3-642-20980-2_7

Quinlan JR (1996) Bagging, boosting, and C4.5. In: Proceedings of the 13th national conference on artificial intelligence and 8th innovative applications of artificial intelligence conference, AAAI 96, IAAI 96, vol 1. AAAI Press/The MIT Press, Portland, pp 725–730

Rice JR (1974) The algorithm selection problem: abstract models. Technical report cSD-TR 116, Computer Science Department, Purdue University, West Lafayette, Indiana

Rice JR (1976) The algorithm selection problem. Adv Comput 15:65–118

Rutkowski L, Cpałka K (2003) Flexible neuro-fuzzy systems. IEEE Trans Neural Networks 14(3):554–574

Smith-Miles KA (2009) Cross-disciplinary perspectives on meta-learning for algorithm selection. ACM Comput Surv 41(1):6:1–6:25. http://doi.acm.org/10.1145/1456650.1456656

Smith-Miles KA, Islam RMD (2011) Meta-learning of instance selection for data summarization. In: Jankowski N, Duch W, Grąbczewski K (eds) Meta-learning in computational intelligence. Studies in computational intelligence, vol 358. Springer, Berlin, pp 77–95. http://dx.doi.org/10.1007/978-3-642-20980-2_2

Soares C (1999) Ranking classification algorithms on past performance. Master's thesis, Faculdade de Economia, Universidade do Porto

Soares C, Brazdil P (2000) Zoomed ranking: selection of classification algorithms based on relevant performance information. In: Proceedings of the 4th European conference on principles of data mining and knowledge discovery (PKDD-2000). Springer, pp 126–135

Soares C, Petrak J, Brazdil P (2001) Sampling-based relative landmarks: systematically test-driving algorithms before choosing. In: EPIA '01: proceedings of the 10th Portuguese conference on artificial intelligence on progress in artificial intelligence, knowledge extraction, multi-agent systems, logic programming and constraint solving. Springer, London, pp 88–95

Stolfo S, Prodromidis A, Tselepis S, Lee W, Fan D, Chan P (1997) JAM: Java agents for meta-learning over distributed databases. In: Proceedings of the 3rd international conference on knowledge discovery and data mining, pp 74–81

Su J, Jelber S, Matwin S, Huang J (2009) Active learning with automatic soft labeling for induction of decision trees. In: Gao Y, Japkowicz N (eds) Advances in artificial intelligence. Lecture notes in computer science, vol 5549. Springer, Berlin, pp 241–244. http://dx.doi.org/10.1007/978-3-642-01818-3_33

Suyama A, Negishi N, Yamaguchi T (1998) CAMLET: a platform for automatic composition of inductive learning systems using ontologies. In: Proceedings of Pacific rim international conference on artificial intelligence, pp 205–215. http://citeseer.ist.psu.edu/42442.html

Todorovski L, Dzeroski S (2003) Combining classifiers with meta decision trees. Mach Learn J 50(3):223–249

Todorovski L, Blockeel H, Dzeroski S (2002) Ranking with predictive clustering trees. In: ECML '02: proceedings of the 13th European conference on machine learning. Springer, London, pp 444–455

Todorovski L, Brazdil P, Soares C (2000) Report on the experiments with feature selection in meta-level learning. In: Proceedings of the PKDD-00 workshop on data mining, decision support, meta-learning and ILP: forum for practical problem presentation and prospective solutions, pp 27–39

Torres-Sospedra J, Hernández-Espinosa C, Fernández-Redondo M (2007) Lecture notes in computer science, vol 4668. Springer, Berlin

Vanschoren J (2011) Meta-learning architectures: Collecting, organizing and exploiting meta-knowledge. In: Jankowski N, Duch W, Grąbczewski K (eds) Meta-learning in computational intelligence. Studies in computational intelligence, vol 358. Springer, Berlin, pp 117–155. http://dx.doi.org/10.1007/978-3-642-20980-2_4

Vilalta R, Giraud-Carrier C, Brazdil P, Soares C (2004) Using meta-learning to support data mining. Int J Comput Sci Appl 1(1):31–45

Vilalta R, Drissi Y (2001) Research directions in meta-learning. In: Proceedings of the international conference on artificial intelligence, Las Vegas, Nevada

Vilalta R, Drissi Y (2002) A perspective view and survey of meta-learning. Artif Intell Rev 18:77–95. http://dx.doi.org/10.1023/A:1019956318069

Vilalta R, Rendell L (1997) Integrating feature construction with multiple classifiers in decision tree induction. In: Proceedings of the 14th international conference on machine learning. Morgan Kaufman, pp 394–402

Wolpert DH, Macready WG (1995) No free lunch theorems for search

Wolpert DH, Macready WG (1996) No free lunch theorems for optimization

Zadrożny S, Kacprzyk J (2007) Bipolar queries using various interpretations of logical connectives. In: Melin P, Castillo O, Aguilar LT, Kacprzyk J, Pedrycz W (eds) Foundations of fuzzy logic and soft computing. Lecture notes in computer science, vol 4529. Springer, Berlin, pp 181–190. http://dx.doi.org/10.1007/978-3-540-72950-1_19

Zenko B, Todorovski L, Dzeroski S (2001) A comparison of stacking with meta decision trees to other combining methods. In: Proceedings A of the 4th international multi-conference information society IS'2001, Jozef Stefan Institute, Ljubljana, Slovenia, pp 144–147

Chapter 7
Future Perspectives of Meta-Learning

The preceding chapter shows various approaches to learning at meta-level. It emphasizes that although the goals may be defined in many different ways, the ultimate goal should always be an improvement in learning at base-level. Even the most attractive form of meta-knowledge is not a value for itself, but only if it can help improve learning processes, so that learning at object-level gets faster or more accurate.

Regardless of the particular goal definition of a meta-learning approach, its meta-level perspective is characterized by answers to the following three fundamental questions:

1. What kind of meta-knowledge can be useful in the approach and in what form it should be kept in repositories?
2. How to extract the meta-knowledge from learning experiments?
3. How to use the collected meta-knowledge in further learning processes?

The future success of the field will definitely depend of how the repositories are prepared and whether they can be shared by various approaches. Since collecting valuable meta-knowledge is very costly, it can not be done from scratch each time a meta-learning approach is undertaken.

Building advanced meta-learning systems is not very popular, yet. The reason of that is not insufficient attractiveness of the field for the researchers. The difficulty of the problem of answering the three questions, enumerated above, is not the cause either. It seems that the main reason is the lack of convenient and powerful tools for easy construction of so complex systems. It can be changed with systems like Intemi, presented in Chap. 4—an environment facilitating multi-aspect analysis of learning processes, with easy-to-use tools. Such frameworks may significantly change the meta-learning research thanks to fastening the development of advanced projects. In preparation of standard and substantial comparisons, the researchers must be freed from the necessity of laborious organization of sophisticated tests, collecting results and thorough, attentive analysis. Easy to run, high-level tools should take care for fair comparisons with statistical methods, leaving for researchers just the most advanced task at higher level of abstraction, requiring non-standard reasoning and intelligence.

K. Grąbczewski, *Meta-Learning in Decision Tree Induction*,
Studies in Computational Intelligence 498, DOI: 10.1007/978-3-319-00960-5_7,

The mechanisms of Intemi made it possible to easily prepare and explore the possibilities of two meta-learning algorithms presented in Sects. 6.3 and 6.4 (the former based on machine configuration generators and complexity control, and the latter driving the search with analysis of learning profiles), and will certainly help create many more attractive meta-learning systems.

Meta-learning as search with validation
The general meta-learning algorithm based on search with feedback from validation, described in Sect. 6.2, integrates meta-learning with the ultimate goal of algorithm selection, that is, solving the object-level tasks. It can be used to solve any shapes of model selection problems, however it does not mean that everything has already been done and no further research in the area is desired. Inversely, there is still a lot of research to do in the realm of meta-learning, because the success of the GML algorithm relies on the proper meta-learners it employs for meta-knowledge extraction and exploitation.

Creating such general algorithm has not given a global answer to the three questions about meta-knowledge, so they can still be answered arbitrarily by specific implementations of the method. Two possible answers have been given in the two instantiations of the GML algorithm discussed in the previous chapter.

Complexity-Driven Meta-Learning
CDML algorithm has been described and tested in Sect. 6.3. One of the most important aspects of this approach concerns the order in which candidate machine configurations are tested. The order is defined by estimates of machine complexities. A general framework of learning evaluators for complexity estimation has also been proposed. An important feature of such evaluators is that they approximate complexity of simple and complex machines, including huge hierarchies performing complicated tests. Complexity control is very advantageous in meta-search, because it helps find the most interesting simple solutions at the beginning of the process and proceed to testing more complex solutions after the simplest ones are verified. Such control prevents spending much time on testing complex machines, while simpler and more accurate models are not checked because of timeout. The results obtained with complexity control are very successful and confirm the advantages of the approach.

The questions about handling meta-knowledge are answered by CDML in a manner specific to this algorithm:

1. CDML uses meta-knowledge of several kinds:
    - usually experts knowledge is very useful in preparing adequate machine generators flow, so that valuable configurations are output by the generators, however the knowledge gained during automated machine generation can also be very helpful and opens new possibilities, because machines may search the areas, that humans would never try,
    - meta evaluators collect meta-knowledge about time and memory consumption of learning machines,
    - optimization scenarios encode meta-knowledge about machine parameter spaces,

- the attractiveness modules remember which machine configuration generators produced attractive solutions and which have not—they can help avoid loosing time for time-consuming calculations, that are not very likely to bring attractive results.

2. Meta-knowledge supporting the CDML algorithm is collected partly from human experts (during the configuration of the meta-learning search) and partly during the search process by analysis of test results and during learning complexity estimation by the evaluators.
3. Exploiting the meta-knowledge is intrinsic to the search process—the factors resulting from complexity analysis or attractiveness estimates represent the meta-knowledge and significantly influence the ML process.

Although more precise than GML, the CDML algorithm is still a general scheme of methods, not a single, strictly defined and closed one. With configuration of subsequent modules, one can obtain a variety of effects, so there is still much space for improvement and new experiments that can bring important meta-knowledge.

One of the future possibilities in this area is certainly a research on new, more and more sophisticated machine generators. Another possibility is to develop intelligent attractiveness modules, significantly more sophisticated than just recording accuracies assigned to machine configurations and their generators. The feedback from experiments can be thoroughly analyzed to draw conclusions about correlations between the configurations and their successfulness.

It is also very important to learn how to transfer the knowledge gained in one ML process to another, that is, how to extract universal information from particular experiments.

Profile-Based Meta-Learning

PBML is also an open framework that facilitates implementation of many meta-learning algorithms integrating meta-knowledge extraction and exploitation with object-level learning. It fits the scheme of the GML algorithm, and specifies many details of dealing with the meta-knowledge, but still, different kinds of problems may be solved with appropriate implementation of the framework modules (the validation scenario and the profile manager). The framework facilitates active management of learning-results profiles, leading to more adequately adapted meta-learning algorithms.

Profile-Based Meta-Learning specifies its own answers to the three fundamental questions about meta-knowledge:

1. The meta-knowledge is encoded in the profiles and their performance. Properly selected profiles can distinguish between data of different internal type, data for which different learning methods are successful. Such information may be a very efficient lodestar in the search for the most suitable learners.
2. Profiles should be constructed with respect to the information that distinguishes between data for which different learners are successful. This kind of profile optimization is at the same time a method for meta-knowledge extraction.

3. As in CDML, the meta-knowledge (here, contained in the profiles and their management) is used internally by the PBML algorithm for proper ordering of candidate learning machines. No special tools for exploitation of this knowledge are necessary, because using the profiles does the job.

A lot of research can still be done to provide successful implementations of the PBML framework. In the previous chapter, only some very simple methods of profile management were used. There is still much space for development of more intelligent methods, that would make the profiles yet more informative. Some experiments with simple methods of profile size reduction have shown, that it is very easy to spoil the results by inaccurate profile maintenance. For example, the idea of handling the profile by keeping the results with as high accuracy as possible, turned out to degenerate the profiles. This is because such technique may, at some point, generate a poor ranking, which is kept and followed to the end of the time scheduled for learning, because once machines of poor accuracy appear at the top, subsequent validations end up with weak results, not eligible for the profile, so the profile does not change and the poor ranking is kept as still valid and is followed on and on. The lack of changes in the profile implies no changes in ranking and inversely. A sort of dead lock appears. Therefore, intelligent profile management must be aware of such possibilities and eliminate them in advance. The profiles must be diverse, and continuously controlled, whether they result in attractive rankings. Otherwise, it can be more successful to return to an old profile and check another way of its modification, instead of loosing time for validation of many machines from a degenerate ranking.

The task of profile management encircles also adaptive methods of dataset similarity measurement. The shape of the profile and the methods of profile similarity measurement are strictly bound with each other in the context of PBML. They must be analyzed and adjusted together to provide successful rankings.

It is not certain if a single profile is the most reasonable way of data similarity measurement. Large knowledge-bases may contain several groups of learning tasks, with completely different sets of characteristic methods, that should be placed within the profile. In such cases it may be much more reasonable to implement multi-profile approaches.

During the processes of profile analysis, some control over knowledge base properties should also be performed to detect irregularities that can spoil the results. As a result, one can prepare knowledge bases in the form, most eligible for meta-learning.

Meta-knowledge repositories

Meta-learning approaches, published so far, are usually completely separate algorithms worked out from scratch. The domain is so huge and so complicated, that significant progress can be made, when new methods "do not look back" all the time and do not start everything from the beginning again and again, but learn to take advantage of many previous experiments.

Each meta-learning experiment is quite costly, and we should not waste any gain of such effort. It is very important, to collect the meta-knowledge within specialized repositories and make it available to other learners through adequate, easy-to-use interfaces. The repositories should not be just databases but ontologies for

more comprehensible meta-learning. As mentioned in the review of Sect. 6.1, several ontologies for meta-learning have already been proposed. These steps certainly move in right direction, however new quality of meta-learning will require creation of very general, easily accessible and easily extensible ontologies, that will attract and will be intensively used by many researchers.

Naturally, with growing amount of meta-knowledge contained within such ontologies, more and more specialized services will be necessary to provide access to the knowledge at appropriate level of abstraction.

Apart from the large repositories and their services, we must still use versatile data mining systems capable of meta-learning, supporting fair validation and drawing reliable conclusions. Otherwise, the knowledge bases may get infected with unreliable information, that could spoil the whole enterprise, so the problem is not so easy as collection of some results.

From specialized search to more and more general solutions

The two meta-learning algorithms described in the previous chapter (CDML and PBML) have been created for significantly different circumstances. CDML can successfully perform validation of learning machines of different kinds (various time and memory complexity) and is not very useful when we need to select between tens of thousands of similarly complex learners. On the other side, the PBML specializes in searching for successful learners among large numbers of similarly complex algorithms, but it does not measure any complexities or adjusts to them.

As in most algorithmic design problems, we will certainly need many more tools specialized in particular applications and mechanisms that will integrate them appropriately, to take advantage of all their skills. Such general and versatile solutions can occur in quite short time, as computational resources available today are not a barrier. Adequate general forms of meta-knowledge representation and methods of meta-knowledge acquisition and exploitation will definitely bring many successful systems, that will more and more often replace human experts in miscellaneous areas.

Appendix A
Some Statistical Methods

The goal of this appendix is not to give a thorough lecture on statistical methods, but to present some details about several methods used by the DT induction algorithms described in the book. Therefore, no complete elementary material is presented here, but only precise information on particular methods.

All algorithms presented here are closely related to *hypothesis testing*. When looking for statistically significant differences between results of two algorithms, or testing if a sample comes from a random variable of some particular parameters, the general idea is to define the *null hypothesis*, prepare a statistic, and estimate, how probable it is to observe such value of the statistic, assuming the null hypothesis. If the probability (the p-value) is small enough, one can reject the null hypothesis with a certain level of confidence.

A.1 Results Comparison

The most robust comparisons of results series are performed on the basis of paired collections. When preparing a test scenario, one should always remember, that availability of results assigned to exactly the same learning tasks but obtained with different learners will facilitate the most reliable assessment of the learners possibilities.

Comparative studies should apply learning and test procedures with the same training data and test data for each algorithm being compared. The result should be a collection of result series of the same length n for each algorithm, $\mathbf{A} = (A_1, \ldots, A_n)$, $\mathbf{B} = (B_1, \ldots, B_n)$ and so on, such that all results with the same index should come from experiments on the same training data and test data samples. For simplicity, the analyses below concern just two samples $\mathbf{A}$ and $\mathbf{B}$.

More detailed explanations of the concepts presented below, together with informative examples, can be found in the materials prepared and exhibited by Lowry (1998–2013).

K. Grąbczewski, *Meta-Learning in Decision Tree Induction*,
Studies in Computational Intelligence 498, DOI: 10.1007/978-3-319-00960-5,

A.1.1 t-Test and Paired t-Test

The t-test for the significance of the difference between the means of two correlated samples (paired t-test) assumes normal distribution of the samples. The procedure starts with calculations of the differences $D_i = A_i - B_i (i = 1, \ldots, n)$ between the correlated results. Then, it suffices to estimate the mean value μ_D and its standard deviation σ_{μ_D}. The mean estimated from the sample is given by:

$$m_D = \frac{1}{n} \sum_{i=1}^{n} D_i. \tag{A.1}$$

The estimate of the variance of the population is

$$s^2 = \frac{1}{n-1} \left(\sum_{i=1}^{n} D_i^2 - \left(\sum_{i=1}^{n} D_i \right)^2 \right). \tag{A.2}$$

Since we are interested in the standard deviation of the mean μ_D, we estimate it as

$$s_{\mu_D} = \sqrt{\frac{s^2}{n}}. \tag{A.3}$$

Finally, we calculate the t-statistic as

$$t = \frac{m_D}{s_{\mu_D}}. \tag{A.4}$$

To decide about significance of the difference, the critical value of t distribution with $n-1$ degrees of freedom must be calculated or read from appropriate table (keeping in mind whether a one-tailed or two-tailed test is conducted).

Sometimes, preparation of paired results for comparison is not possible. If we have two independent samples of results, they can be compared, for example, with the plain version of the t-test (for two independent samples). Starting with samples $\mathbf{A} = (A_1, \ldots, A_n)$ and $\mathbf{B} = (B_1, \ldots, B_k)$, the mean difference may be estimated as

$$m_{A-B} = \frac{1}{n} \sum_{i=1}^{n} A_i - \frac{1}{k} \sum_{i=1}^{k} B_i. \tag{A.5}$$

The variance of the source population is then estimated as

$$s^2 = \frac{\left(\sum_{i=1}^{n} A_i^2 - \left(\sum_{i=1}^{n} A_i \right)^2 \right) + \left(\sum_{i=1}^{k} B_i^2 - \left(\sum_{i=1}^{k} B_i \right)^2 \right)}{n - 1 + k - 1}. \tag{A.6}$$

Hence, the standard deviation of the mean difference can be estimated as

$$s_{\mu_{A-B}} = \sqrt{\frac{s^2}{n} + \frac{s^2}{k}}. \tag{A.7}$$

Finally, the t-statistic is calculated as

$$t = \frac{m_{A_B}}{s_{\mu_{A-B}}}, \tag{A.8}$$

and critical values of t distribution with $n - 1 + k - 1$ degrees of freedom are used.

A.1.2 Wilcoxon Test

Wilcoxon signed-rank test is a *non-parametric* counterpart of the parametric paired t-test. Similarly, the Mann-Whitney test is a non-parametric counterpart of the t-test for independent samples. Here, only the Wilcoxon test is presented. It also starts with calculations of the differences $D_i = A_i - B_i (i = 1, \ldots, n)$ between the correlated results. Then, each difference D_i gets split into the sign (+ or −) and the absolute difference $|D_i|$. Next the results are ordered by the absolute difference $|D_i|$ and assigned ranks R_i. Before the ranks are assigned, the results with zero difference are usually removed. The ranks are granted from smallest to the largest $|D_i|$ in such a way, that if several scores happen to be the same, all they get the same rank equal to the mean of the ranks applicable to their positions, for example, if the same scores are placed at position 2 and 3, they both get the rank of 2.5. Given the ranks, they are multiplied by the corresponding signs to get the *signed ranks* S_i. All the signed ranks are summed and the sum denoted as W:

$$W = \sum_{i=0}^{n} S_i. \tag{A.9}$$

Under the null hypothesis (of insignificant differences between the results), the value of W is expected to follow a distribution with

$$\mu_W = 0, \qquad \sigma_W = \sqrt{\frac{n(n+1)(2n+1)}{6}}. \tag{A.10}$$

With increasing n, the distribution of W gets closer to normal. In practice, when $n \geq 10$ the z-ratio is calculated:

$$z = \frac{(W - \mu_W) \pm 0.5}{\sigma_W} = \frac{W \pm 0.5}{\sigma_W}, \tag{A.11}$$

and the standard normal distribution is used for making decisions about null hypothesis rejection. The ± 0.5 is added for continuity correction (positive when $W < 0$ and negative otherwise).

For samples smaller than 10-elements, individual analysis of the exact sampling distribution can be performed.

A.1.3 McNemar's Test

McNemar's test verifies significance of the difference between two correlated proportions. It is applied to 2×2 contingency tables with matched pairs of binary values, to determine whether the row and column marginal frequencies are equal. It perfectly fits the task of comparing the correctness and errors of decisions made by two classifiers. After checking the correctness of n decisions of each classifier (A and B), denote by n_{11} the number of cases with correct answers of both classifiers, n_{00}—the number of cases with errors observed in both A's and B's decisions, n_{01}—the number of cases for which A is wrong and B is correct, and n_{10}—how many times A was right and B was wrong. The contingency table can be written as:

	B:1	B:0
A:1	n_{11}	n_{10}
A:0	n_{01}	n_{00}

The null hypothesis is that the two marginal distributions (row sums and column sums) are the same (*marginal homogeneity*), which means that $n_{10} = n_{01}$.

Thus, the McNemar's statistic is calculated as

$$\chi^2 = \frac{(n_{10} - n_{01})^2}{n_{10} + n_{01}}, \tag{A.12}$$

and is distributed as χ^2 with 1 degree of freedom. In practice, this path is followed if $n_{10} + n_{01} > 25$. Also, a continuity correction is often applied in the form:

$$\chi^2 = \frac{(|n_{10} - n_{01}| - c)^2}{n_{10} + n_{01}}, \tag{A.13}$$

with $c = 0.5$ or $c = 1$.

When $n_{10} + n_{01} \leq 25$, then assuming binomial distribution of successes with $p = 0.5$, the exact probability of $k = \min(n_{10}, n_{01})$ or fewer successes in $n_{10} + n_{01}$ trials can be calculated:

$$P(X \leq k) = \sum_{i=0}^{k} \binom{n_{10} + n_{01}}{i} 0.5^{n_{10}+n_{01}}. \tag{A.14}$$

A.1.4 Bonferroni Corrections

Hypothesis testing can be a very precious source of information, if only it is performed in a fair, reliable manner. So formal procedures can also be used improperly and falsify conclusions. For example, when a researcher creates a new parameterized classification algorithm and compares its results with some state-of-the-art methods, a temptation can occur to test a number of possible values of the parameters to explore the space of capabilities of the new approach. Comparing each parameter settings separately, against the state-of-the-art methods, with statistical significance tests at confidence level α, does not authorize to a conclusion that the best setting is significantly better at level α, even if the test confirms so. When a number of learning machines is tested, the probability of obtaining erroneous confirmation of the significance grows, so interpretation of the α should be changed.

Bonferroni proposed to use appropriately smaller confidence level in such multiple tests. Without the assumption of statistical independence of the tests, it can be easily inferred from the Boole's inequality, that when n tests are performed and the final conclusions is expected at the confidence level of α, than it suffices to run each of the n tests with the level of $\frac{\alpha}{n}$.

Similar analysis of type I errors in multiple tests, but with the assumption of statistical independence between the tests, leads to the conclusion, that to guarantee the overall type I error of n tests to be smaller than α, each particular test should be performed with the confidence level of $1 - \sqrt[n]{1-\alpha}$.

A.2 Distribution Comparison

The statistical tests presented below are often used in DT induction to determine whether a random variable (usually a feature or a split) contains some statistically important information about the classification of interest.

A.2.1 F-Test of ANOVA

Some statistical DT induction algorithms use the F-test of ANOVA (analysis of variance) to measure how different the groups of objects representing different classes are.

Given k groups of values $X_{i1}, \ldots, X_{in_i}$ $(i = 1, \ldots, k)$, the ANOVA F statistic is defined as the ratio between the variability observed between groups by the within-group variability:

$$F = \frac{\frac{1}{k-1}\sum_{i=1}^{k} n_i(\overline{X}_{i\cdot} - \overline{X})^2}{\frac{1}{n-k}\sum_{i=1}^{k}\sum_{j=1}^{n_i}(X_{ij} - \overline{X}_{i\cdot})^2}, \tag{A.15}$$

where $\overline{X}$ is the overall mean of the data, and $\overline{X}_{i\cdot}$ is the sample mean within i'th group. Under the null hypotheses that all groups are normally distributed with the same variance, the definition of F follows the F distribution with $k-1, n-k$ degrees of freedom.

When the null hypotheses is rejected, the test still does not answer the question which group is significantly different than the others, but it has been successfully exploited to detect discrimination potential of variables describing data.

A.2.2 Pearson's χ^2 Test

In classification DT induction the Pearson's χ^2-test is often used to test independence between a data feature and the class variable. Generally, given two discrete variables V and C with possible values $v_1, \ldots, v_m$ and $c_1, \ldots, c_k$ respectively, the counts of observed objects with appropriate combinations of the values may be written as the following contingency table:

	c_1	$\ldots$	c_k
v_1	O_{11}	$\cdots$	O_{1k}
$\vdots$	$\vdots$		$\vdots$
v_m	O_{m1}	$\cdots$	O_{mk}

Under the null hypothesis, about independence of variables V and C, the expected values of observed combinations are proportional to the marginal frequencies:

$$E_{ij} = \frac{1}{n} \sum_{u=1}^{k} O_{iu} \cdot \sum_{w=1}^{m} O_{wj}, \tag{A.16}$$

where n is the total sample size (the sum of all cells). The χ^2 statistic can be defined as:

$$\chi^2 = \sum_{i=1}^{m} \sum_{j=1}^{k} \frac{(O_{ij} - E_{ij})^2}{E_{ij}}. \tag{A.17}$$

Assuming the independence (null hypothesis), the statistic follows the χ^2 distribution with $(k-1)(m-1)$ degrees of freedom.

In some approaches to variable selection, a statistic $\phi^2 = \frac{\chi^2}{n}$ is used in place of χ^2, to reduce the bias in favor of variables with many values.

A.2.3 Levene's Test of Homegeneity of Variances

The null hypothesis in *Levene's test* claims that the variances of k groups of values $X_{i1}, \ldots, X_{in_i}$ $(i = 1, \ldots, k)$ are equal. The test statistic is defined as

$$W = \frac{\frac{1}{k-1}\sum_{i=1}^{k} n_i(\overline{Z}_{i\cdot} - \overline{Z})^2}{\frac{1}{n-k}\sum_{i=1}^{k}\sum_{j=1}^{n_i}(\overline{Z}_{ij} - \overline{Z}_{i\cdot})^2}, \tag{A.18}$$

where $Z_{ij} = |X_{ij} - \overline{X}_{i\cdot}|$, and $\overline{X}_{i\cdot}$ is the mean of the ith group. Naturally, $\overline{Z}_{i\cdot}$ and $\overline{Z}$ are respectively the mean of the i'th group and the overall mean of Z variable. Sometimes, in the definition of Z_{ij}, the median or a trimmed mean is used instead of the mean $\overline{X}_{i\cdot}$.

Assuming homogeneity of variances, W follows F distribution with $k-1, n-k$ degrees of freedom.

A.2.4 Fisher's Exact Test

Fisher's exact probability test is a nonparametric technique for analyzing 2×2 contingency tables, representing a small sample. It determines whether the groups differ in the proportion with which they fall into the two classifications. Therefore, it perfectly fits many situations in DT induction, where analysis of a small sample is necessary (for example, in the context of deciding whether to prune a branch or not).

Given a 2×2 contingency table

	1	2	Total
Group 1	a	b	$a+b$
Group 2	c	d	$c+d$
Total	$a+c$	$b+d$	$n = a+b+c+d$

the exact probability of observing the particular frequencies, regarding the marginal totals as fixed, can be calculated as

$$p = \frac{\binom{a+c}{a}\binom{b+d}{b}}{\binom{n}{a+b}} = \frac{(a+b)!(c+d)!(a+c)!(b+d)!}{n!a!b!c!d!}. \tag{A.19}$$

When the p-value is appropriately small, one can reject the null hypothesis that the two groups do not differ significantly in the proportions shown in columns.

Appendix B
Some Algebraic Methods

The four algebraic procedures presented in this appendix are a selection of the most important algebraic transformations referred to in the book. No systematic introduction to algebra is provided here. Included procedures are not discussed exhaustively, but just shortly described, to give the reader orientation about the goals of the transformations and a general view of the techniques.

B.1 CrimCoord Transformation

CrimCoord is a short name of a procedure called *discriminant coordinate* or *canonical variate* transformation. This transformation is used, for example, in QUEST and CRUISE methods of the FACT family (see Sect. 2.2.5) to convert symbolic features to numeric ones. The CrimCoord transformation detects the most discriminant direction, analogously to the Principal Component Analysis (PCA) which detects directions of the largest variance. Discrimination means that the objects are grouped in a way, so the algorithm is supervised, in opposition to PCA which is unsupervised. The most discriminant direction means the projection $\mathbf{a}^T$ that maximizes the ratio of between-classes to within-classes sum-of-squares

$$\frac{\mathbf{a}^T \mathbf{B} \mathbf{a}}{\mathbf{a}^T \mathbf{W} \mathbf{a}}, \tag{B.1}$$

where:

$$\mathbf{B} = \sum_{C \in \mathscr{C}} n_C (\overline{\mathbf{v}}^C - \overline{\mathbf{v}})(\overline{\mathbf{v}}^C - \overline{\mathbf{v}})^T, \tag{B.2}$$

$$\mathbf{W} = \sum_{i=1}^{n} (\mathbf{v}_i - \overline{\mathbf{v}}^{c_i})(\mathbf{v}_i - \overline{\mathbf{v}}^{c_i})^T. \tag{B.3}$$

K. Grąbczewski, *Meta-Learning in Decision Tree Induction*,
Studies in Computational Intelligence 498, DOI: 10.1007/978-3-319-00960-5,

The vector $\mathbf{v}_i$ is the i'th data vector, c_i is its class label, $\overline{\mathbf{v}}$ is the mean vector of the whole set of vectors, $\overline{\mathbf{v}}^C$ is the mean of the vectors assigned class $C \in \mathscr{C}$ and n_C is the number of vectors assigned class label C.

Algorithm B.1 (CrimCoord transformation)

Prototype: ***CrimCoord(V,c)***
Input: *$n \times m$ matrix* $\mathbf{V}$*, group labels vector* $\mathbf{c} \in \mathscr{C}^n$, $\mathscr{C} = \{C_1, \ldots, C_k\}$.
Output: *m-dimensional vector.*
The algorithm:

1. $\mathbf{H} \leftarrow I - \frac{1}{n}\mathbf{1}\mathbf{1}^T$ *(centering matrix)*
2. $\mathbf{PDQ}^T \leftarrow SVD(\mathbf{HV})$ *(*$\mathbf{D} = diag(d_1, \ldots, d_m), d_1 \geq \cdots \geq d_m \geq 0$*)*
3. $r \leftarrow \max\{i = 1, \ldots, m : d_i > max(m, n)d_1\epsilon\}$ *(ϵ is the machine epsilon: the smallest machine number such that* $1 + \epsilon > 1$*)*
4. $\mathbf{F} \leftarrow (\mathbf{Q}_1, \ldots, \mathbf{Q}_r)$
5. $\mathbf{U} \leftarrow diag(\frac{1}{d_1}, \ldots, \frac{1}{d_r})$
6. $\mathbf{G} \leftarrow (\mathbf{L}_1, \ldots, \mathbf{L}_k)^T$*, where* $\mathbf{L}_i = (\overline{\mathbf{v}}^{C_i} - \overline{\mathbf{v}}, \ldots, \overline{\mathbf{v}}^{C_i} - \overline{\mathbf{v}})$ *($n \times n_{C_i}$ matrix, needs group labels* $\mathbf{c}$*)*
7. $\mathbf{a} \leftarrow$ *eigenvector associated with the largest eigenvalue of* $\mathbf{GFU}$ *(another SVD call)*
8. ***return*** $\mathbf{a}^T\mathbf{UF}^T$

Algorithm B.1 presents the procedure formally. It runs *Singular Value Decomposition* twice, to eventually determine a vector transforming each vector of the original space into a real number, being the discriminant coordinate.

In QUEST and CRUISE the method is applied to the binary indicator variables, encoding symbolic features, so as to convert each symbolic feature to a continuous one.

B.2 Singular Value Decomposition

Singular Value Decomposition (SVD) is a decomposition of a matrix into a product of three matrices as described by the following theorem:

Theorem 2.1 *For each $m \times n$ matrix* $\mathbf{A}$ *of rank r, there exist:*

- *a $m \times m$ orthonormal matrix* $\mathbf{U}$*,*
- *a $n \times n$ orthonormal matrix* $\mathbf{V}$*,*
- *and $m \times n$ matrix* $\Sigma = \begin{bmatrix} \mathbf{D} & 0 \\ 0 & 0 \end{bmatrix}$, $\mathbf{D} = diag(\sigma_1, \ldots, \sigma_r)$, $\sigma_1 \geq \cdots \geq \sigma_r > 0$ *are singular values of* $\mathbf{A}$*,*

such that

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T. \tag{B.4}$$

SVD has a number of useful consequences:

- Columns of **U**are eigenvectors of $\mathbf{AA}^T$ for eigenvalues $\sigma_1^2, \ldots, \sigma_r^2$ (because $\mathbf{AA}^T = \mathbf{U}\Sigma^2\mathbf{U}^T$).
- Columns of **V**are eigenvectors of $\mathbf{A}^T\mathbf{A}$ for eigenvalues $\sigma_1^2, \ldots, \sigma_r^2$ (because $\mathbf{A}^T\mathbf{A} = \mathbf{V}\Sigma^2\mathbf{V}^T$).
- If **A**is a nonsingular square matrix ($n \times n$), than its inverse is $\mathbf{A}^{-1} = \mathbf{V}\Sigma^{-1}\mathbf{U}^T$, where $\Sigma^{-1} = diag(\frac{1}{\sigma_1}, \ldots, \frac{1}{\sigma_n})$.

B.3 Principal Component Analysis

Principal Component Analysis (PCA) is a statistical method of data analysis, often used for data reduction, visualization of high-dimensional data and other purposes. It is a linear transformation that changes the basis from the original, possibly correlated coordinates into linearly uncorrelated variables in such a way that the first coordinate (first principal component) of the destination system corresponds to the greatest variance, second coordinate contains the second greatest variance and so on. The transformation is often used for dimensionality reduction because several first principal components usually contain almost all information about the variance in the data, so that for example, by looking at just two first principal components one can see all the dependencies in the original multidimensional space.

PCA is usually performed by means of *eigenvalue decomposition* or *Singular Value Decomposition*. A nice compilation of the two methods has been presented by Shlens (2005).

Formally, PCA transforms a given matrix **X** into $\mathbf{Y} = \mathbf{PX}$ with an orthonormal matrix **P**such that the covariance matrix $\mathbf{C_Y} = \frac{1}{n-1}\mathbf{YY}^T$ is diagonalized. Then, the rows of **P**are the principal components of **X**.

Performing PCA by means of SVD is based on the property that the principal components of **X**are eigenvectors of the covariance matrix $\mathbf{C_X} = \frac{1}{n-1}\mathbf{XX}^T$. Given this fact, it suffices to perform the SVD of matrix $\mathbf{Z} = \frac{1}{\sqrt{n-1}}\mathbf{X}^T$, because when $\mathbf{Z} = \mathbf{U}\Sigma\mathbf{V}^T$, then the columns of **V**are eigenvectors of $\mathbf{Z}^T\mathbf{Z} = \mathbf{C_X}$ so they are the principal components of **X**.

B.4 Box–Cox Transformation

Best results in data modeling can be obtained when data distribution is compatible with the assumptions of the modeling method. Logarithmic or exponential transformations may be valuable tools in proper data preparation. This idea lies behind the *Box-Cox power transformation* (Box and Cox 1964; Sakia 1992; Qu and Loh 1992) defined as the following function parameterized by λ:

$$x^{(\lambda)} = \begin{cases} \frac{x^{\lambda}-1}{\lambda} & \text{when } \lambda \neq 0, \\ \log(x) & \text{when } \lambda = 0. \end{cases} \tag{B.5}$$

CRUISE uses the transformation before linear discriminant analysis is applied to split a tree node. Before the transformation, all data points are shifted to make the sample contain positive values only. The shift is defined as $\theta = x_{(2)} - 2x_{(1)}$, where $x_{(1)}$ and $x_{(2)}$ are the first and the second order statistics of the sample (that is the first and the second smallest values occurring in the sample). Transformation parameter λ is selected to minimize the following formula:

$$\sum_{j=1}^{k} \sum_{i=1}^{n_{C_j}} \left(x_{ji}^{(\lambda)} - \overline{x_j^{(\lambda)}} \right)^2 exp\left(-\frac{2}{n} \lambda \sum_{j=1}^{k} \sum_{i=1}^{n_{C_j}} \log x_{ji} \right), \tag{B.6}$$

where n_{C_j} is the count of objects in the sample representing class C_j, x_{ji} is the value of i'th object from class C_j, and $\overline{x_j^{(\lambda)}}$ is the mean of transformed values of objects from class C_j.

References

Box GEP, Cox DR (1964) An analysis of transformations. J Roy Stat Soc Ser B (Methodol) 26(2):211–252, http://www.jstor.org/stable/2984418

Lowry R (1998–2013) Concepts and applications of inferential statistics. http://vassarstats.net/textbook/

Qu P, Loh WY (1992) Application of Box-Cox transformations to discrimination for the two-class problem. Commun Stat Theor Methods 21:2757–2774

Sakia R (1992) The box-cox transformation technique: a review. Statistician 41:169–178

Shlens J (2005) A tutorial on principal component analysis. Systems Neurobiology Laboratory, University of California, San Diego

Index

K. Grąbczewski, *Meta-Learning in Decision Tree Induction*,
Studies in Computational Intelligence 498, DOI: 10.1007/978-3-319-00960-5,