O'REILLY®

Data Science with Java

PRACTICAL METHODS FOR SCIENTISTS AND ENGINEERS

Michael R. Brzustowicz, PhD

Data Science with Java

Michael R. Brzustowicz, PhD

Data Science with Java

by Michael R. Brzustowicz, PhD

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com/safari). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Editors: Nan Barber and Brian Foster
- Production Editor: Kristen Brown
- Copyeditor: Sharon Wilkey
- Proofreader: Jasmine Kwityn
- Indexer: Lucie Haskins
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest
- June 2017: First Edition

Revision History for the First Edition

- 2017-05-30: First Release

978-1-491-93411-1

[LSI]

Dedication

This book is for my cofounder and our two startups.

Preface

Data science is a diverse and growing field encompassing many subfields of both mathematics and computer science. Statistics, linear algebra, databases, machine intelligence, and data visualization are just a few of the topics that merge together in the realm of a data scientist. Technology abounds and the tools to practice data science are evolving rapidly. This book focuses on core, fundamental principles backed by clear, object-oriented code in Java. And while this book will inspire you to get busy right away practicing the craft of data science, it is my hope that you will take the lead in building the next generation of data science technology.

Who Should Read This Book

This book is for scientists and engineers already familiar with the concepts of application development who want to jump headfirst into data science. The topics covered here will walk you through the data science pipeline, explaining mathematical theory and giving code examples along the way. This book is the perfect jumping-off point into much deeper waters.

Why I Wrote This Book

I wrote this book to start a movement. As data science skyrockets to stardom, fueled by R and Python, very few practitioners venture into the world of Java. Clearly, the tools for data exploration lend themselves to the interpretive languages. But there is another realm of the engineering–science hybrid where scale, robustness, and convenience must merge. Java is perhaps the one language that can do it all. If this book inspires you, I hope that you will contribute code to one of the many open source Java projects that support data science.

A Word on Data Science Today

Data science is continually changing, not only in scope but also in those practicing it. Technology moves very fast, with top algorithms moving in and out of favor in a matter of years or even months. Long-time standardized practices are discarded for practical solutions. And the barrier to success is regularly hurdled by those in fields previously untouched by quantitative science. Already, data science is an undergraduate curriculum. There is only one way to be successful in the future: know the math, know the code, and know the subject matter.

Navigating This Book

This book is a logical journey through a data science pipeline. In Chapter 1, the many methods for getting, cleaning, and arranging data into its purest form are examined, as are basic data output to files and plotting. Chapter 2 addresses the important concept of viewing our data as a matrix. An exhaustive review of matrix operations is presented. Now that we have data and know what data structure it should take, Chapter 3 introduces the basic concepts that allow us to test the origin and validity of our data. In Chapter 4, we directly use the concepts from Chapters 2 and 3 to transform our data into stable and usable numerical values. Chapter 5 contains a few useful supervised and unsupervised learning algorithms, as well as methods for evaluating their success. Chapter 6 provides a quick guide to getting up and running with MapReduce by using customized components suitable for data science algorithms. A few useful datasets are described in Appendix A.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **CAUTION**
>
> This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/oreillymedia/Data_Science_with_Java*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Data Science with Java* by Michael Brzustowicz (O'Reilly). Copyright 2017 Michael Brzustowicz, 978-1-491-93411-1."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O’Reilly Safari

NOTE

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O’Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

- O’Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- 800-998-9938 (in the United States or Canada)
- 707-829-0515 (international or local)
- 707-829-0104 (fax)

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I would like to thank the book's editors at O'Reilly, Nan Barber and Brian Foster, for their continual encouragement and guidance throughout this process.

I am also grateful for the staff at O'Reilly: Melanie Yarbrough, Kristen Brown, Sharon Wilkey, Jennie Kimmel, Allison Gillespie, Laurel Ruma, Seana McInerney, Rita Scordamalgia, Chris Olson, and Michelle Gilliland, all of whom contributed to getting this book in print.

This book benefited from the many technical comments and affirmations of colleagues Dustin Garvey, Jamil Abou-Saleh, David Uminsky, and Terence Parr. I am truly thankful for all of your help.

Chapter 1. Data I/O

Events happen all around us, continuously. Occasionally, we make a record of a discrete event at a certain point in time and space. We can then define *data* as a collection of records that someone (or something) took the time to write down or present in any format imaginable. As data scientists, we work with data in files, databases, web services, and more. Usually, someone has gone through a lot of trouble to define a schema or data model that precisely denotes the names, types, tolerances, and inter-relationships of all the variables. However, it is not always possible to enforce a schema during data acquisition. Real data (even in well-designed databases) often has missing values, misspellings, incorrectly formatted types, duplicate representations for the same value, and the worst: several variables concatenated into one. Although you are probably excited to implement machine-learning algorithms and create stunning graphics, the most important and time-consuming aspect of data science is preparing the data and ensuring its integrity.

What Is Data, Anyway?

Your ultimate goal is to retrieve data from its source, reduce the data via statistical analysis or learning, and then present some kind of knowledge about what was learned, usually in the form of a graph. However, even if your result is a single value such as the total revenue, most engaged user, or a quality factor, you still follow the same protocol: *input data → reductive analysis → output data*.

Considering that practical data science is driven by business questions, it will be to your advantage to examine this protocol from right to left. First, formalize the question you are trying to answer. For example, do you require a list of top users by region, a prediction of daily revenue for the next week, or a plot of the distribution of similarities between items in inventory? Next, explore the chain of analyses that can answer your questions. Finally, now that you have decided on your approach, exactly what data will you need to accomplish this goal? You may be surprised to find that you do not have the data required. Often you will discover that a much simpler set of analysis tools (than you originally envisioned) will be adequate to achieve the desired output.

In this chapter, you will explore the finer details of reading and writing data from a variety of sources. It is important to ask yourself what data model is required for any subsequent steps. Perhaps it will suffice to build a series of numerical array types (e.g., `double[][]`, `int[]`, `String[]`) to contain the data. On the other hand, you may benefit from creating a container class to hold each data record, and then populating a `List` or `Map` with those objects. Still another useful data model is to formulate each record as a set of key-value pairs in a JavaScript Object Notation (JSON) document. The decision of what data model to choose rests largely on the input requirements of the subsequent data-consuming processes.

Data Models

What form is the data in, and what form do you need to transform it to so you can move forward? Suppose *somefile.txt* contained rows of `id`, `year`, and `city` data.

Univariate Arrays

The simplest data model for this particular example is to create a series of arrays for the three variables `id`, `year`, and `city`:

```
int[] id = new int[1024];
int[] year = new int[1024];
String[] city = new String[1024];
```

As the `BufferedReader` loops through the lines of the file, values are added to each position of the arrays with the aid of an incrementing counter. This data model is probably adequate for clean data of known dimensions, where all the code ends up in one executable class. It would be fairly straightforward to feed this data into any number of statistical analysis or learning algorithms. However, you will probably want to modularize your code and build classes and subsequent methods suited for each combination of data source and data model. In that case, shuttling around arrays will become painful when you have to alter the signatures of existing methods to accommodate new arguments.

Multivariate Arrays

Here you want each row to hold all the data for a record, but they must be the same type! So in our case, this would work only if you assigned cities a numerical, integer value:

```
int[] row1 = {1, 2014, 1};
int[] row2 = {2, 2015, 1};
int[] row3 = {3, 2014, 2};
```

You could also make this a 2D array:

```
int[][] data = {{1, 2014, 1}, {2, 2015, 1}, {3, 2014, 2}};
```

For your first pass through a dataset, there may be a complicated data model already, or just a mixture of text, integers, doubles, and date times. Ideally, after you have worked out what will go into a statistical analysis or learning algorithm, this data is transformed into a two-dimensional array of doubles. However, it takes quite a bit of work to get to that point. On the one hand, it's convenient to be handed a matrix of data from which you can forge ahead with machine learning. On the other, you may not know what compromises were made or what mistakes have been propagated, undetected.

Data Objects

Another option is to create a container class and then populate a collection such as `List` or `Map` with those containers. The advantages are that it keeps all the values of a particular record together, and adding new members to a class will not break any methods that take the class as an argument. The data in the file *somefile.txt* can be represented by the following class:

```
class Record {
    int id;
    int year;
    String city;
}
```

Keep the class as lightweight as possible, because a collection (`List` or `Map`) of these objects will add up for a large dataset! Any methods acting on `Record` could be static methods ideally in their own class titled something like `RecordUtils`.

The collection's structure, `List`, is used to hold all the `Record` objects:

```
List<Record> listOfRecords = new ArrayList<>();
```

Looping though the data file with a `BufferReader`, each line can then be parsed and its contents stored in a new `Record` instance. Each new `Record` instance is then added to `List<Record> listOfRecords`. Should you require a key to quickly look up and retrieve an individual `Record` instance, use a `Map`:

```
Map<String, Record> mapOfRecords = new HashMap<>();
```

The key to each record should be a unique identifier for that particular record, such as a record ID or URL.

Matrices and Vectors

Matrices and *vectors* are higher-level data structures composed of, respectively, two- and one-dimensional arrays. Usually, a dataset contains multiple columns and rows, and we can say that these variables form a two-dimensional array (or *matrix*) **X** in which there are m rows and n columns. We choose i to be the row index, and j to be the column index, such that each element of the $m \times n$ matrix is $x_{i,j}$

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix}$$

When we put values into a data structure like a matrix, we can gain convenience. In many situations, we will be performing mathematical operations on our data. A matrix instance can have abstract methods for performing these operations, with implementation details that are suited for the task at hand. We will explore matrices and vectors in detail in Chapter 2.

JSON

JavaScript Object Notation (JSON) has become a prevalent form of representing data. In general, JSON data is represented by simple rules at *json.org*: double quotes! No trailing commas! A JSON object has outer curly braces and can have any valid set of key-value pairs separated by commas (the order of contents is not guaranteed, so treat it as a `HashMap` type):

```
{"city":"San Francisco", "year": 2020, "id": 2, "event_codes":[20, 22, 34, 19]}
```

A JSON array has outer square brackets with valid JSON contents separated by commas (the order of array contents is guaranteed, so treat it as an `ArrayList` type):

```
[40, 50, 70, "text", {"city":"San Francisco"}]
```

There are two main categories you will find. Some data files contain complete JSON objects or arrays. These are usually configuration files. However, another type of data structure that is common is a text file of independent JSON objects, one per line. Note that this type of data structure (list of JSONs) is technically not a JSON object or array because there are no closing braces or commas between lines, and as such, trying to parse the whole data structure as one JSON object (or array) will fail.

Dealing with Real Data

Real data is messy, incomplete, incorrect, and sometimes incoherent. If you are working with a “perfect” dataset, it’s because someone else spent a great deal of time and effort in getting it that way. It is also possible that your data is, in fact, not perfect, and you are unwittingly performing analyses on junk data. The only way to be sure is to get data from the source and process it yourself. This way, if there is a mistake, you know who to blame.

Nulls

Null values appear in a variety of forms. If the data is being passed around inside Java, it's entirely possible to have a `null`. If you are parsing strings from a text file, a `null` value may be represented by a variety of the literal string `"null"`, `"NULL"`, or other string such as `"na"`, or even a dot. In either case (a null type or null literal), we want to keep track of these:

```
private boolean checkNull(String value) {
    return value == null || "null".equalsIgnoreCase(value);
}
```

Often a null value has been recorded as a blank space or series of blank spaces. Although this is sometimes a nuisance, it may serve a purpose, because encoding a 0 is not always appropriate to represent the concept that the data point does not exist. For example, if we were tracking binary variables, 0 and 1, and came across an item for which we did not know the value, then wrongly assigning 0 to the value (and writing it to the file) would incorrectly assign a true negative value. When writing a null value to a text file, my preference is for a zero-length string.

Blank Spaces

Blank spaces abound in real data. It is straightforward to check for an empty string by using the `String.isEmpty()` method. However, keep in mind that a string of blank spaces (even one blank space) is not empty! First, we use the `String.trim()` method to remove any leading or trailing spaces around the input value and then check its length. `String.isEmpty()` returns `true` only if the string has zero length:

```
private boolean checkBlank(String value) {
    return value.trim().isEmpty();
}
```

Parse Errors

Once we know the string value is neither null nor blank, we parse it into the type we require. We'll leave the parsing of strings to strings out of this, because there is nothing to parse!

When dealing with numeric types, it is unwise to cast strings to a primitive type such as `double`, `int`, or `long`. It is recommended to use the object wrapper classes such as `Double`, `Integer`, and `Long`, which have a string-parsing method that throws a `NumberFormatException` should something go wrong. We can catch that exception and update a parsing error counter. You can also print or log the error:

```
try {
    double d = Double.parseDouble(value);
    // handle d
} catch (NumberFormatException e) {
    // increment parse error counter etc.
}
```

Similarly, date times formatted as a string can be parsed by the `OffsetDateTime.parse()` method; the `DateTimeParseException` can be caught and logged should something be wrong with the input string:

```
try {
    OffsetDateTime odt = OffsetDateTime.parse(value);
    // handle odt
} catch (DateTimeParseException e) {
    // increment parse error counter etc.
}
```

Outliers

Now that our data is cleaned and parsed, we can check whether the value is acceptable given our requirements. If we were expecting a value of either 0 or 1 and we get a 2, the value is clearly out of range and we can designate this data point as an outlier. As in the case with nulls and blanks, we can perform a Boolean test on the value to determine whether it is within an acceptable range of values. This is good for numeric types as well as strings and date times.

In the case of checking ranges with numeric types, we need to know the minimum and maximum acceptable values and whether they are inclusive or exclusive. For example, if we set `minValue = 1.0` and `minValueInclusive = true`, all values greater than or equal to 1.0 will pass the test. If we set `minValueInclusive = false`, only values greater than 1.0 will pass the test. Here is the code:

```
public boolean checkRange(double value) {
    boolean minBit = (minValueInclusive) ? value >= minValue : value > minValue;
    boolean maxBit = (maxValueInclusive) ? value <= maxValue : value < maxValue;
    return minBit && maxBit;
}
```

Similar methods can be written for integer types.

We can also check whether a string value is in an acceptable range by setting an enumeration of valid strings. This can be done by creating a `Set` instance of valid strings called, for example, `validItems`, where the `Set.contains()` method can be used to test the validity of an input value:

```
private boolean checkRange(String value) {
    return validItems.contains(value);
}
```

For `DateTime` objects, we can check whether a date is after a minimum date and before a maximum date. In this case, we define the min and max as `OffsetDateTime` objects and then test whether the input date time is between the min and max. Note that `OffsetDateTime.isBefore()` and `OffsetDateTime.isAfter()` are exclusive. If the input date time is equal to either the min or max, the test will fail. Here is the code:

```
private boolean checkRange(OffsetDateTime odt) {
    return odt.isAfter(minDate) && odt.isBefore(maxDate);
}
```

Managing Data Files

This is where the art of data science begins! How you choose to build a dataset is not only a matter of efficiency, but also one of flexibility. There are many options for reading and writing files. As a bare minimum, the entire contents of the file can be read into a `String` type by using a `FileReader` instance, and then the `String` can be parsed into the data model. For large files, I/O errors are avoided by using a `BufferedReader` to read each line of the file separately. The strategy presented here is to parse each line as it is read, keeping only the values that are required and populating a data structure with those records. If there are 1,000 variables per line, and only three are required, there is no need to keep all of them. Likewise, if the data in a particular line does not meet certain criteria, there is also no need to keep it. For large datasets, this conserves resources compared to reading all the lines into a string array (`String[]`) and parsing it later. The more consideration you put into this step of managing data files, the better off you will be. Every step you take afterward, whether it's statistics, learning, or plotting, will rely on your decisions when building a dataset. The old adage of "garbage in, garbage out" definitely applies.

Understanding File Contents First

Data files come in a bewildering array of configurations, with some undesirable *features* as a result. Recall that ASCII files are just a collection of ASCII characters printed to each line. There is no guarantee on the format or precision of a number, the use of single or double quotes, or the inclusion (or exclusion) of numerous space, null, and newline characters. In short, despite your assumptions as to the contents of the file, there can be almost anything on each line. Before reading in the file with Java, take a look at it in a text editor or with the command line. Note the number, position, and type of each item in a line. Pay close attention to how missing or null values are represented. Also note the type of delimiter and any headers describing the data. If the file is small enough, you can scan it visually for missing or incorrectly formatted lines. For example, say we look at the file *somefile.txt* with the Unix command `less` in a bash shell:

```
bash$ less somefile.txt

"id","year","city"
1,2015,"San Francisco"
2,2014,"New York"
3,2012,"Los Angeles"
...
```

We see a comma-separated values (CSV) dataset with the columns `id`, `year`, and `city`. We can quickly check the number of lines in the file:

```
bash$ wc -l somefile.txt
1025
```

This indicates that there are 1,024 lines of data plus one line more for the header. Other formats are possible, such as tab-separated values (TSV), a “big string” format in which all the values are concatenated together, and JSON. For large files, you may want to take the first 100 or so lines and redirect them to an abridged file for purposes of developing your application:

```
bash$ head -100 filename > new_filename
```

In some cases, the data file is just too big for a pair of eyes to scan it for structure or errors. Clearly, you would have trouble examining a data file with 1,000 columns of data! Likewise, you are unlikely to find an error in formatting by scrolling through one million lines of data. In this case it is essential that you have an existing data dictionary that describes the format of the columns and the data types (e.g., integer, float, text) that are expected for each column. You can programmatically check each line of data as you parse the file via Java; exceptions can be thrown, and, perhaps, the entire contents of the offending line printed out so you can examine what went wrong.

Reading from a Text File

The general approach for reading a text file is to create a `FileReader` instance surrounded by a `BufferedReader` that enables reading each line. Here, `FileReader` takes the argument of `String` *`filename`*, but `FileReader` can also take a `File` object as its argument. The `File` object is useful when filenames and paths are dependent on the operating system. This is the generic form for reading files line by line with a `BufferedReader`:

```
try(BufferedReader br = new BufferedReader(new FileReader("somefile.txt")) ) {
    String columnNames = br.readline(); // ONLY do this if it exists
    String line;
    while ((line = br.readLine()) != null) {
        /* parse each line */
        // TODO
    }
} catch (Exception e) {
    System.err.println(e.getMessage()); // or log error
}
```

We can do the exact same thing if the file exists somewhere remotely:

```
URL url = new URL("http://storage.example.com/public-data/somefile.txt");
try(BufferedReader br = new BufferedReader(
    new InputStreamReader(url.openStream())) ) {
    String columnNames = br.readline(); // ONLY do this if it exists
    String line;
    while ((line = br.readLine()) != null) {
        // TODO parse each line
    }
} catch (Exception e) {
    System.err.println(e.getMessage()); // or log error
}
```

We just have to worry about how to parse each line.

Parsing big strings

Consider a file in which each row is a “big string” of concatenated values, and any substring with starting and stopping positions encodes a particular variable:

```
0001201503
0002201401
0003201202
```

The first four digits are the `id` number, the second four are the `year`, and the last two are the `city` code. Keep in mind that each line can be thousands of characters long, and the position of character substrings is critical. It is typical that numbers will be padded with zeros, and empty spaces may be present for null values. Note that periods occurring inside a float (e.g., 32.456) count as a space, as will any other “strange” character! Usually, text strings are encoded as values. For example, in this case, `New York = 01`, `Los Angeles = 02`, and `San Francisco = 03`.

In this case, the values from each line can be accessed with the method `String.substring(int beginIndex, int endIndex)`. Note that the substring starts at `beginIndex` and goes up to (but not including) `endIndex`:

```
/* parse each line */
int id = Integer.parseInt(line.substring(0, 4));
int year = Integer.parseInt(line.substring(4, 8));
int city = Integer.parseInt(line.substring(8, 10));
```

Parsing delimited strings

Considering the popularity of spreadsheets and database dumps, it is highly likely you will be given a CSV dataset at some point. Parsing this kind of file could not be easier! Consider the data in our example formatted as a CSV file:

```
1,2015,"San Francisco"
2,2014,"New York"
3,2012,"Los Angeles"
```

Then all we need to do is parse with `String.split(",")` and utilize `String.trim()` to remove any pesky leading or trailing whitespaces. It also will be necessary to remove any quotes around strings with `String.replace("\"", "")`:

```
/* parse each line */
String[] s = line.split(",");
int id = Integer.parseInt(s[0].trim());
int year = Integer.parseInt(s[1].trim());
String city = s[2].trim().replace("\"", "");
```

In the next example, the data in *somefile.txt* has been separated by tabs:

```
1	2015	"San Francisco"
2	2014	"New York"
3	2012	"Los Angeles"
```

Splitting tab-delimited data is achieved by replacing code for `String.split(",")` in the preceding example with this:

```
String[] s = line.split("\t");
```

At some point, you will undoubtedly come across CSV files with fields that contain commas. One example is text taken from a user blog. Yet another example occurs when denormalized data is put into a column — for example, “San Francisco, CA” instead of having separate columns for city and state. This is quite tricky to parse and requires regex. Instead, why not use the Apache Commons CSV parser library?

```
/* parse each line */
CSVParser parser = CSVParser.parse(line, CSVFormat.RFC4180);
for(CSVRecord cr : parser) {
    int id = cr.get(1); // columns start at 1 not 0 !!!
    int year = cr.get(2);
    String city = cr.get(3);
}
```

The Apache Commons CSV library also handles common formats including *CSVFormat.EXCEL*, *CSVFormat.MYSQL*, and *CSVFormat.TDF*.

Parsing JSON strings

JSON is a protocol for serializing JavaScript objects and can be extended to data of all types. This compact, easy-to-read format is ubiquitous in Internet data APIs (in particular, RESTful services) and is the standard format for many NoSQL solutions such as MongoDB and CouchDB. As of version 9.3, the PostgreSQL database offers a JSON data type and can query native JSON fields. The clear advantage is human readability; the structure of the data is readily visible, and with "pretty print," even more so. In terms of Java, JSON is nothing more than a collection of `HashMaps` and `ArrayLists`, in any nested configuration imaginable. Each line of the data from the prior examples can be formatted as a JSON string by placing the values into key-value pairs; strings are in double quotes (*not* single quotes), and *no* trailing commas are allowed:

```
{"id":1, "year":2015, "city":"San Francisco"}
{"id":2, "year":2014, "city":"New York"}
{"id":3, "year":2012, "city":"Los Angeles"}
```

Note that the entire file itself is not technically a JSON object, and parsing the whole file as such will fail. To be valid JSON format, each line would need to be separated by a comma and then the entire group enclosed with square brackets. This would comprise a JSON array. However, writing this kind of structure would be inefficient and not useful. It is much more convenient and usable as is: a line-by-line stack of JSON objects in string representation. Note that the JSON parser does not know the type of the values in the key-value pairs. So get the `String` representation and then parse it to its primitive type by using the boxed methods. It is straightforward to build our dataset now, using `org.simple.json`:

```
/* create JSON parser outside while loop */
JSONParser parser = new JSONParser();
...

/* create an object by casting the parsed string */
JSONObject obj = (JSONObject) parser.parse(line);
int id = Integer.parseInt(j.get("id").toString());
int year = Integer.parseInt(j.get("year").toString());
String city = j.get("city").toString();
```

Reading from a JSON File

This section covers files that are stringified JSON objects or arrays. You have to know beforehand whether the file is a JSON object or an array. If you look at the file with, for example, `ls` on the command line, you can tell if it has curly braces (object) or square braces (array):

```
{{"id":1, "year":2015, "city":"San Francisco"},
 {"id":2, "year":2014, "city":"New York"},
 {"id":3, "year":2012, "city":"Los Angeles"}}
```

Then you use the Simple JSON library:

```
JSONParser parser = new JSONParser();
try{
    JSONObject jObj = (JSONObject) parser.parse(new FileReader("data.json"));
    // TODO do something with jObj
} catch (IOException|ParseException e) {
    System.err.println(e.getMessage());
}
```

And if it's an array,

```
[{"id":1, "year":2015, "city":"San Francisco"},
 {"id":2, "year":2014, "city":"New York"},
 {"id":3, "year":2012, "city":"Los Angeles"}]
```

then you can parse the entire JSON array:

```
JSONParser parser = new JSONParser();
try{
    JSONArray jArr = (JSONArray) parser.parse(new FileReader("data.json"));
    // TODO do something with jObj
} catch (IOException|ParseException e) {
    System.err.println(e.getMessage());
}
```

> **WARNING**
>
> If you really have a file with one JSON object per line, the file is not technically a qualified JSON data structure. Refer back to "Reading from a Text File" where we read text files, parsing JSON objects one line at a time.

Reading from an Image File

When using images as input for learning, we need to convert from the image format (e.g., PNG) to a data structure that is appropriate, such as a matrix or vector. There are several points to consider here. First, an image is a two-dimensional array with coordinates, $\{x_1, x_2\}$, and a set of associated color or intensity values, $\{y_1 \ldots\}$, that may be stored as a single, integer value. If all we want is the raw value stored in a 2D integer array (labeled `data` here), we read in the buffered image with this:

```
BufferedImage img = null;
try {
    img = ImageIO.read(new File("Image.png"));
    int height = img.getHeight();
    int width = img.getWidth();
    int[][] data = new int[height][width];
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            int rgb = img.getRGB(i, j); // negative integers
            data[i][j] = rgb;
        }
    }
} catch (IOException e) {
    // handle exception
}
```

We may want to convert the integer into its RGB (red, green, blue) components by bit shifting the integer:

```
int blue = 0x0000ff & rgb;
int green = 0x0000ff & (rgb >> 8);
int red = 0x0000ff & (rgb >> 16);
int alpha = 0x0000ff & (rgb >> 24);
```

However, we can get this information natively from the raster with this:

```
byte[] pixels = ((DataBufferByte) img.getRaster().getDataBuffer()).getData();
for (int i = 0; i < pixels.length / 3 ; i++) {
    int blue = Byte.toUnsignedInt(pixels[3*i]);
    int green = Byte.toUnsignedInt(pixels[3*i+1]);
    int red = Byte.toUnsignedInt(pixels[3*i+2]);
}
```

Color may not be important. Perhaps grayscale is really all that's needed:

```
//convert rgb to grayscale (0 to 1) where colors are on a scale of 0 to 255
double gray = (0.2126 * red + 0.7152 * green + 0.0722 * blue) / 255.0
```

Also, in some cases the 2D representation is not necessary. We convert the matrix to a vector by concatenating each row of the matrix onto the new vector such that $\mathbf{x}_n = \mathbf{x}_1, \mathbf{x}_2, \ldots$, where the length n of the vector is $m \times p$ of the matrix, the number of rows times the number of columns. In the well-known MNIST dataset of handwritten images, the data has already been corrected (centered and cropped) and then converted into a binary format. So reading in that data requires a special format (see Appendix A), but it is already in vector (1D) as opposed to matrix (2D) format. Learning techniques on the MNIST dataset usually involve this vectorized format.

Writing to a Text File

Writing data to files has a general form of using the `FileWriter` class, but once again the recommended practice is to use the `BufferedWriter` to avoid any I/O errors. The general concept is to format all the data you want to write to file as a single string. For the three variables in our example, we can do this manually with a delimiter of choice (either a comma or `\t`):

```
/* for each instance Record record */
String output = Integer.toString(record.id) + "," +
Integer.toString(record.year) + "," + record.city;
```

When using Java 8, the method `String.join(delimiter, elements)` is convenient!

```
/* in Java 8 */
String newString = String.join(",", {"a", "b", "c"});

/* or feed in an Iterator */
String newString = String.join(",", myList);
```

Otherwise, you can instead use the Apache Commons Lang `StringUtils.join(elements, delimiter)` or the native `StringBuilder` class in a loop:

```
/* in Java 7 */
String[] strings = {"a", "b", "c"};

/* create a StringBuilder and add the first member */
StringBuilder sb;
sb.append(strings[0]);

/* skip the first string since we already have it */
for(int i = 1; i < strings.length, i++){
    /* choose a delimiter here ... could also be a \t for tabs */
    sb.append(",");
    sb.append(strings[i]);
}

String newString = sb.toString();
```

Note that successively using `myString += myString_part` calls the `StringBuilder` class, so you might as well use `StringBuilder` anyway (or not). In any case, the strings are written line by line. Keep in mind that the method `BufferedWriter.write(String)` does not write a new line! You will have to include a call to `BufferedWriter.newLine()` if you would like each data record to be on its own line:

```
try(BufferedWriter bw = new BufferedWriter(new FileWriter("somefile.txt")) ) {
    for(String s : myStringList){
        bw.write(s);
        /* don't forget to append a new line! */
        bw.newLine();
    }
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

The preceding code overwrites all existing data in the file designated by filename. In some situations, you will want to append data to an existing file. The `FileWriter` class takes an optional Boolean field `append` that defaults to `false` if it is excluded. To open a file for appending to the next available line, use

this:

```
/* setting FileWriter append bit keeps existing data and appends new data */
try(BufferedWriter bw = new BufferedWriter(
    new FileWriter("somefile.txt", true))) {
    for(String s : myStringList){
        bw.write(s);
        /* don't forget to append a new line! */
        bw.newLine();
    }
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

Still another option is to use the `PrintWriter` class, which wraps around the `BufferedWriter`. `PrintWriter` and has a method `println()` that uses the native newline character of whatever operating system you are on. So the `\n` can be excluded in the code. This has the advantage that you don't have to worry about adding those pesky newline characters. This could also be useful if you are generating text files on your own computer (and therefore OS) and will be consuming these files yourself. Here is an example using `PrintWriter`:

```
try(PrintWriter pw = new PrintWriter(new BufferedWriter(
    new FileWriter("somefile.txt"))) ) {
    for(String s : myStringList){
        /* adds a new line for you! */
        pw.println(s);
    }
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

Any of these methods work just fine with JSON data. Convert each JSON object to a `String` with the `JSONObject.toString()` method and write the `String`. If you are writing one JSON object, such as a configuration file, then it is as simple as this:

```
JSONObject obj = ...

try(BufferedWriter bw = new BufferedWriter(new FileWriter("somefile.txt")) ) {
    bw.write(obj.toString());
}
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

When creating a JSON data file (a stack of JSON objects), loop through your collection of `JSONObjects`:

```
List<JSONObject> dataList = ...

try(BufferedWriter bw = new BufferedWriter(new FileWriter("somefile.txt")) ) {
    for(JSONObject obj : dataList){
        bw.write(obj.toString());
        /* don't forget to append a new line! */
        bw.newLine();
    }
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

Don't forget to set the append-bit in `FileWriter` if this file is accumulative! You can add more JSON

records to the end of this file simply by setting the append-bit in the `FileWriter`:

```
try(BufferedWriter bw = new BufferedWriter(
    new FileWriter("somefile.txt", true)) ) {
...
}
```

Mastering Database Operations

The robustness and flexibility of relational databases such as MySQL make them the ideal technology for a wide range of use cases. As a data scientist, you will most likely interact with relational databases in connection to a larger application, or perhaps you will generate tables of condensed and organized data specific to the tasks of the data science group. In either case, mastering the command line, Structured Query Language (SQL), and Java Database Connectivity (JDBC) are critical skills.

Command-Line Clients

The command line is a great environment for managing the database as well as performing queries. As an interactive shell, the client enables rapid iteration of commands useful for exploring the data. After you work out queries on the command line, you can later transfer the SQL to your Java program, where the query can be parameterized for more flexible use. All of the popular databases such as MySQL, PostgreSQL, and SQLite have command-line clients. On systems where MySQL has been installed for development purposes (e.g., your personal computer), you should be able to connect with an anonymous login with an optional database name:

```
bash$ mysql <database>
```

However, you might not be able to create a new database. You can log in as the database administrator:

```
bash$ mysql -u root <database>
```

Then you can have full access and privileges. In all other cases (e.g., you are connecting to a production machine, remote instance, or cloud-based instance), you will need the following:

```
bash$ mysql -h host -P port -u user -p password <database>
```

Upon connecting, you will be greeted with the MySQL shell, where you can make queries for showing all the databases you have access to, the name of the database you are connected to, and the username:

```
mysql> SHOW DATABASES;
```

To switch databases to a new database, the command is `USE` *`dbname`*:

```
mysql> USE myDB;
```

You can create tables now:

```
mysql> CREATE TABLE my_table(id INT PRIMARY KEY, stuff VARCHAR(256));
```

Even better, if you have those table creation scripts stored away as files, the following will read in and execute the file:

```
mysql> SOURCE <filename>;
```

Of course, you may want to know what tables are in your database:

```
mysql> SHOW TABLES;
```

You may also want to get a detailed description of a table, including column names, data types, and constraints:

```
mysql> DESCRIBE <tablename>;
```

Structured Query Language

Structured Query Language (SQL) is a powerful tool for exploring data. While object-relational mapping (ORM) frameworks have a place in enterprise software applications, you may find them too restrictive for the kinds of tasks you will face as a data scientist. It is a good idea to brush up on your SQL skills and be comfortable with the basics presented here.

Create

To create databases and tables, use the following SQL:

```
CREATE DATABASE <databasename>;

CREATE TABLE <tablename> ( col1 type, col2 type, ...);
```

Select

A generalized bare-bones `SELECT` statement will have this form:

```
SELECT
    [DISTINCT]
    col_name, col_name, ... col_name
    FROM table_name
    [WHERE where_condition]
    [GROUP BY col_name [ASC | DESC]]
    [HAVING where_condition]
    [ORDER BY col_name [ASC | DESC]]
    [LIMIT row_count OFFSET offset]
    [INTO OUTFILE 'file_name']
```

A few tricks may come in handy. Suppose your dataset contains millions of points, and you just want to get a general idea of the shape. You can return a random sample by using `ORDER BY`:

```
ORDER BY RAND();
```

And you can set `LIMIT` to the sample size you would like back:

```
ORDER BY RAND() LIMIT 1000;
```

Insert

Inserting data into a new row is implemented via the following:

```
INSERT INTO tablename(col1, col2, ...) VALUES(val1, val2, ...);
```

Note that you can drop the column name entirely if the values account for all the columns and not just a subset:

```
INSERT INTO tablename VALUES(val1, val2, ...);
```

You can also insert multiple records at once:

```
INSERT INTO tablename(col1, col2, ...) VALUES(val1, val2, ...),(val1, val2, ...),
(val1, val2, ...);
```

Update

On some occasions, you will need to alter an existing record. A lot of times this occurs quickly, on the command line, when you need to patch a mistake or correct a simple typo. Although you will undoubtedly access databases in production, analytics, and testing, you may also find yourself in an ad hoc DBA position. Updating records is common when dealing with real users and real data:

```
UPDATE table_name SET col_name = 'value' WHERE other_col_name = 'other_val';
```

In the realm of data science, it is hard to envision a situation where you will be programmatically updating data. There will be exceptions, of course, such as the aforementioned typo corrections or when building a table piecemeal, but for the most part, updating important data sounds like a recipe for disaster. This is particularly true if multiple users are relying on the same data and have already written code, and subsequent analyses depend on a static dataset.

Delete

Deleting data is probably unnecessary in these days of cheap storage, but just like `UPDATE`, deleting will come in handy when you've made an error and don't want to rebuild your whole database. Typically, you will be deleting records based on certain criteria, such as a `user_id` or `record_id`, or before a certain date:

```
DELETE FROM <tablename> WHERE <col_name> = 'col_value';
```

Another useful command is `TRUNCATE`, which deletes *all* the data in a table but keeps the table intact. Essentially, `TRUNCATE` wipes a table clean:

```
TRUNCATE <tablename>;
```

Drop

If you want to delete all the contents of a table and the table itself, you must `DROP` the table. This gets rid of tables entirely:

```
DROP TABLE <tablename>;
```

This deletes an entire database and all of its contents:

```
DROP DATABASE <databasename>;
```

Java Database Connectivity

The Java Database Connectivity (JDBC) is a protocol connecting Java applications with any SQL-compliant database. The JDBC drivers for each database vendor exist as a separate JAR that must be included in build and runtime. The JDBC technology strives for a uniform layer between applications and databases regardless of the vendor.

Connections

Connecting to a database with JDBC is extremely easy and convenient. All you need is a properly formed URI for the database that takes this general form:

```
String uri = "jdbc:<dbtype>:[location]/<dbname>?<parameters>"
```

The `DriverManager.getConnection()` method will throw an exception, and you have two choices for dealing with this. The modern Java way is to put the connection inside the `try` statement, known as a *try with resource*. In this way, the connection will be automatically closed when the block is done executing, so you do not have to explicitly put in a call to `Connection.close()`. Remember that if you decide to put the connection statement in the actual `try` block, you will need to explicitly close the connection, probably in a `finally` block:

```
String uri = "jdbc:mysql://localhost:3306/myDB?user=root";
try(Connection c = DriverManager.getConnection(uri)) {
    // TODO do something here
} catch (SQLException e) {
    System.err.println(e.getMessage());
}
```

Now that you have a connection, you need to ask yourself two questions:

- Are there any variables in the SQL string (will the SQL string be altered in any way)?
- Am I expecting any results to come back from the query other than an indicator that it was successful or not?

Start by assuming that you will create a `Statement`. If the `Statement` will take a variable (e.g., if the SQL will be appended to by an application variable), then use a `PreparedStatement` instead. If you do not expect any results back, you are OK. If you are expecting results to come back, you need to use `ResultSets` to contain and process the results.

Statements

When executing an SQL statement, consider the following example:

```
DROP TABLE IF EXISTS data;
CREATE TABLE IF NOT EXISTS data(
    id INTEGER PRIMARY KEY,
    yr INTEGER,
    city VARCHAR(80));
INSERT INTO data(id, yr, city) VALUES(1, 2015, "San Francisco"),
    (2, 2014, "New York"),(3, 2012, "Los Angeles");
```

All of the SQL statements are hardcoded strings with *no* varying parts. They return no values (other than a Boolean return code) and can be executed, individually, inside the above `try-catch` block with this:

```
String sql = "<sql string goes here>";
Statement stmt = c.createStatement();
stmt.execute(sql);
stmt.close();
```

Prepared statements

You will probably not be hardcoding all your data into an SQL statement. Likewise, you may create a generic update statement for updating a record's `city` column given an `id` by using an SQL `WHERE` clause. Although you may be tempted to build SQL strings by concatenating them, this is not a recommended practice. Anytime external input is substituted into an SQL expression, there is room for an SQL injection attack. The proper method is to use placeholders (as question marks) in the SQL statement and then use the class `PreparedStatement` to properly quote the input variables and execute the query. Prepared statements not only have a security advantage but one of speed as well. The `PreparedStatement` is compiled one time, and for a large number of inserts, this makes the process extremely efficient compared to compiling a new SQL statement for each and every insertion. The preceding `INSERT` statement, with corresponding Java can be written as follows:

```
String insertSQL = "INSERT INTO data(id, yr, city) VALUES(?, ?, ?)";
PreparedStatement ps = c.prepareStatement(insertSQL);
/* set the value for each placeholder ? starting with index = 1 */
ps.setInt(1, 1);
ps.setInt(2, 2015);
ps.setString(3, "San Francisco");
ps.execute();
ps.close();
```

But what if you have a lot of data and need to loop through a list? This is where you execute in *batch mode*. For example, suppose you have a `List` of `Record` objects obtained from an import of CSV:

```
String insertSQL = "INSERT INTO data(id, yr, city) VALUES(?, ?, ?)";
PreparedStatement ps = c.prepareStatement(insertSQL);
List<Record> records = FileUtils.getRecordsFromCSV();
for(Record r: records) {
    ps.setInt(1, r.id);
    ps.setInt(2, r.year);
    ps.setString(3, r.city);
    ps.addBatch();
}
ps.executeBatch();
ps.close();
```

Result sets

`SELECT` statements return results! Anytime you find yourself writing `SELECT` you will need to properly call `Statement.executeQuery()` instead of `execute()` and assign the return value to a `ResultSet`. In database-speak, the `ResultSet` is a cursor that is an iterable data structure. As such, the Java class `ResultSet` implements the Java Iterator class and the familiar `while-next` loop can be used:

```
String selectSQL = "SELECT id, yr, city FROM data";
Statement st = c.createStatement();
ResultSet rs = st.executeQuery(selectSQL);
```

```
while(rs.next()) {
    int id = rs.getInt("id");
    int year = rs.getInt("yr");
    String city = rs.getString("city"));
    // TODO do something with each row of values
}
rs.close();
st.close();
```

As in the case with reading files line by line, you must choose what to do with the data. Perhaps you will store each value in an array of that type, or perhaps you will store each row of data into a class, and build a list with that class. Note that we are retrieving the values from the `ResultSet` instance by calling column values by their column names according to the database schema. We can instead increment through the column indices starting with 1.

Visualizing Data with Plots

Data visualization is an important and exciting component of data science. The combination of broadly available, interesting data and interactive graphical technologies has led to stunning visualizations, capable of telling complex stories. Many times, our visualizations are the eye candy that everyone has been anticipating. Of utmost importance is to realize that the same source of data can be used to tell completely different stories depending on not only the segment of the data you choose to show, but also the graphical styling utilized.

Keeping in mind that data visualization should always take into consideration the audience, there are roughly three kinds of consumers of a visualization. The first is yourself, the all-knowing expert who is most likely iterating quickly on an analysis or algorithm development. Your requirements are to see the data as plainly and quickly as possible. Things such as setting plot titles, axis labels, smoothing, legends, or date formatting might not be important, because you are intimately aware of what you are looking at. In essence, we often plot data to get a quick overview of the data landscape, without concerning ourselves with how others will view it.

The second consumer of data visualizations is the industry expert. After you have solved a data science problem and you think it’s ready to share, it’s essential to fully label the axis, put a meaningful, descriptive title on it, make sure any series of data are described by a legend, and ensure that the graphic you have created can mostly tell a story on its own. Even if it’s not visually stunning, your colleagues and peers will probably not be concerned with eye candy, but rather the message you are trying to convey. In fact, it will be much easier to make a scientific evaluation on the merits of the work if the visualization is clear of graphical widgets and effects. Of course, this format is also essential for archiving your data. One month later, you will not remember what those axes are if you don’t label them now!

The third category of visualization consumer is everybody else. This is the time to get creative and artistic, because a careful choice of colors and styles can make good data seem great. Be cautious, however, of the tremendous amount of time and effort you will spend preparing graphics at this level of consumer. An added advantage of using JavaFX is the interactivity allowed via mouse options. This enables you to build a graphical application similar to many of the web-based dashboards you are accustomed to.

Creating Simple Plots

Java contains native graphics capabilities in the JavaFX package. Since version 1.8, scientific plotting is enabled with charts of many types such as scatter, line, bar, stacked bar, pie, area, stacked area, or bubble via the `javafx.scene.chart` package. A `Chart` object is contained in a `Scene` object, which is contained in a `Stage` object. The general form is to extend an executable Java class with `Application` and place all the plotting directives in the overridden method `Application.start()`. The `Application.launch()` method must be called in the `main` method to create and display the chart.

Scatter plots

An example of a simple plot is a scatter chart, which plots a set of x-y pairs of numbers as points on a grid. These charts utilize the `javafx.scene.chart.XYChart.Data` and `javafx.scene.chart.XYChart.Series` classes. The `Data` class is a container that holds any dimension of mixed types of data, and the `Series` class contains an `ObservableList` of `Data` instances. There are factory methods in the `javafx.collections.FXCollections` class for creating instances of `ObservableList` directly, should you prefer that route. However, for scatter, line, area, bubble, and bar charts, this is unnecessary because they all utilize the `Series` class:

```
public class BasicScatterChart extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage stage) throws Exception {
        int[] xData = {1, 2, 3, 4, 5};
        double[] yData = {1.3, 2.1, 3.3, 4.0, 4.8};

        /* add Data to a Series */
        Series series = new Series();
        for (int i = 0; i < xData.length; i++) {
            series.getData().add(new Data(xData[i], yData[i]));
        }

        /* define the axes */
        NumberAxis xAxis = new NumberAxis();
        xAxis.setLabel("x");
        NumberAxis yAxis = new NumberAxis();
        yAxis.setLabel("y");

        /* create the scatter chart */
        ScatterChart<Number,Number> scatterChart =
            new ScatterChart<>(xAxis, yAxis);
        scatterChart.getData().add(series);


        /* create a scene using the chart */
        Scene scene  = new Scene(scatterChart, 800, 600);

        /* tell the stage what scene to use and render it! */
        stage.setScene(scene);
        stage.show();
    }

}
```

Figure 1-1 depicts the default graphics window that is displayed when rendering a JavaFX chart for a simple set of data.

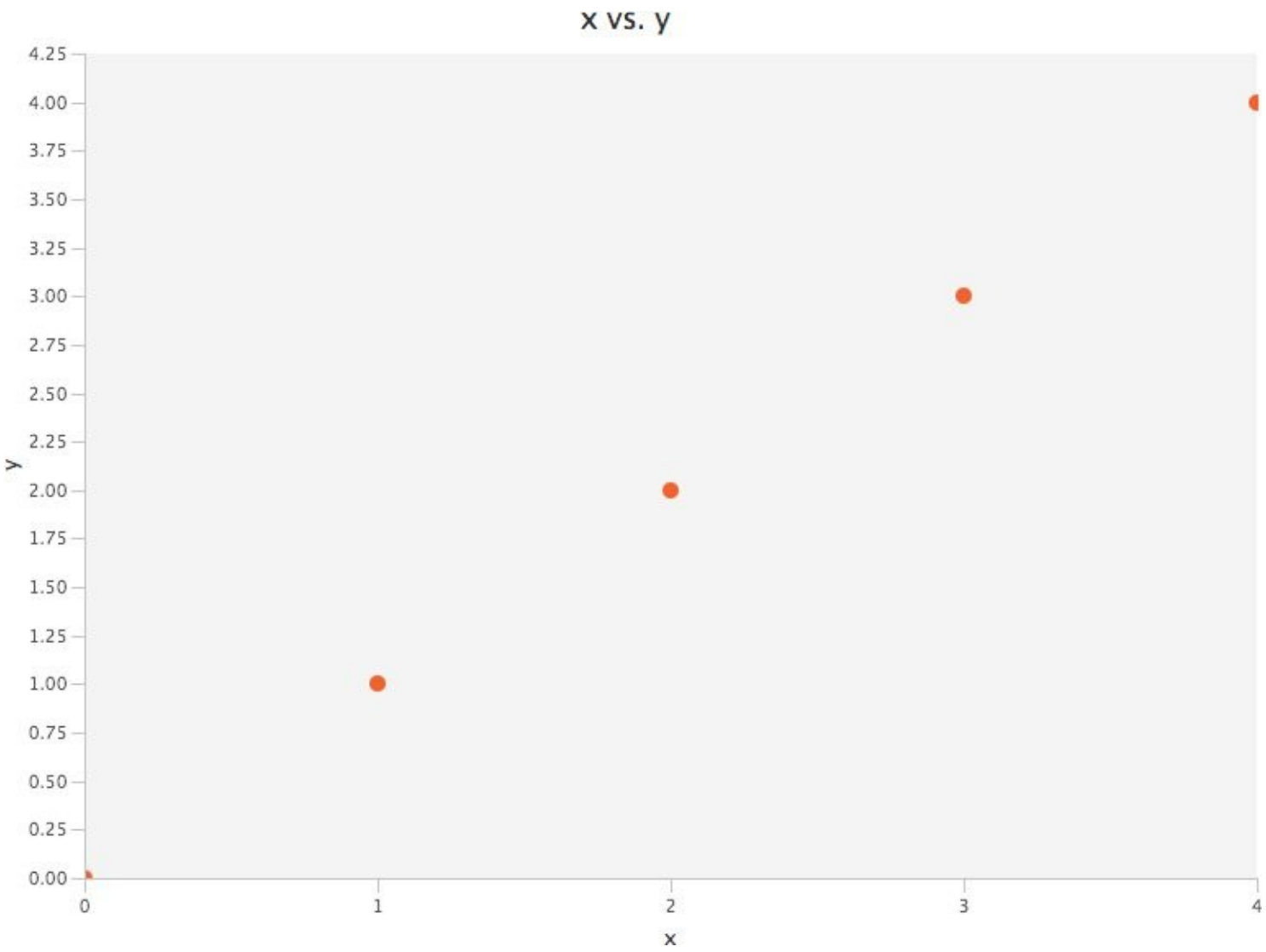

Figure 1-1. Scatter plot example

The `ScatterChart` class can readily be replaced with `LineChart`, `AreaChart`, or `BubbleChart` in the preceding example.

Bar charts

As an x-y chart, the bar chart utilizes the `Data` and `Series` classes. In this case, however, the only difference is that the x-axis must be a string type (as opposed to a numeric type) and utilizes the `CategoryAxis` class instead of the `NumberAxis` class. The y-axis remains as a `NumberAxis`. Typically, the categories in a bar chart are something like days of the week or market segments. Note that the `BarChart` class takes a `String`, `Number` pair of types inside the diamonds. These are useful for making histograms, and we show one in Chapter 3:

```
public class BasicBarChart extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage stage) throws Exception {

        String[] xData = {"Mon", "Tues", "Wed", "Thurs", "Fri"};
        double[] yData = {1.3, 2.1, 3.3, 4.0, 4.8};
```

```
        /* add Data to a Series */
        Series series = new Series();
        for (int i = 0; i < xData.length; i++) {
            series.getData().add(new Data(xData[i], yData[i]));
        }

        /* define the axes */
        CategoryAxis xAxis = new CategoryAxis();
        xAxis.setLabel("x");
        NumberAxis yAxis = new NumberAxis();
        yAxis.setLabel("y");

        /* create the bar chart */
        BarChart<String,Number> barChart = new barChart<>(xAxis, yAxis);
        barChart.getData().add(series);


        /* create a scene using the chart */
        Scene scene  = new Scene(barChart, 800, 600);

        /* tell the stage what scene to use and render it! */
        stage.setScene(scene);
        stage.show();
    }

}
```

Plotting multiple series

Multiple series of any type of plot are easily implemented. In the case of the scatter plot example, you need only to create multiple `Series` instances:

```
Series series1 = new Series();
Series series2 = new Series();
Series series3 = new Series();
```

The series are then added in all at once using the `addAll()` method instead of the `add()` method:

```
scatterChart.getData().addAll(series1, series2, series3);
```

The resultant plot will show the points superimposed in various colors with a legend denoting their label name. The same holds true for line, area, bar, and bubble charts. An interesting feature here is the `StackedAreaChart` and `StackedBarChart` classes, which operate the same way as their respective `AreaChart` and `BarChart` superclasses, except that the data are stacked one above the other so they do not overlap visually.

Of course, sometimes a visualization would benefit from mixing data from multiple plot types, such as a scatter plot of data with a line plot running through the data. Currently, the `Scene` class accepts only charts of one type. However, we will demonstrate some workarounds later in this chapter.

Basic formatting

There are useful options for making your plot look really professional. The first place to cleanup might be the axes. Often the minor ticks are overkill. We can also set the plot range with minimum and maximum values:

```
scatterChart.setBackground(null);
scatterChart.setLegendVisible(false);
scatterChart.setHorizontalGridLinesVisible(false);
```

```
scatterChart.setVerticalGridLinesVisible(false);
scatterChart.setVerticalZeroLineVisible(false);
```

At some point, it might be easier to keep the plotting mechanics simple and include all the style directives in a CSS file. The default CSS for JavaFX8 is called Modena and will be implemented if you don't change the style options. You can create your own CSS and include it in the scene with this:

```
scene.getStylesheets().add("chart.css");
```

The default path is in the *src/main/resources* directory of your Java package.

Plotting Mixed Chart Types

Often we want to display multiple plot types in one graphic — for example, when you want to display the data points as an x-y scatter plot and then overlay a line plot of the best fitted model. Perhaps you will also want to include two more lines to represent the boundary of the model, probably one, two, or three multiples of the standard deviation σ, or the confidence interval 1.96 × σ. Currently, JavaFX does not allow multiple plots of the different types to be displayed simultaneously on the same scene. There is a workaround, however! We can use a `LineChart` class to plot multiple series of `LineChart` instances and then use CSS to style one of the lines to show only points, one to only show a solid line, and two to show only a dashed line. Here is the CSS:

```
.default-color0.chart-series-line {
    -fx-stroke: transparent;
}

.default-color1.chart-series-line {
    -fx-stroke: blue; -fx-stroke-width: 1;
}

.default-color2.chart-series-line {
    -fx-stroke: blue;
    -fx-stroke-width: 1;
    -fx-stroke-dash-array: 1 4 1 4;
}

.default-color3.chart-series-line {
    -fx-stroke: blue;
    -fx-stroke-width: 1;
    -fx-stroke-dash-array: 1 4 1 4;
}

/*.default-color0.chart-line-symbol {
    -fx-background-color: white, green;
}*/

.default-color1.chart-line-symbol {
    -fx-background-color: transparent, transparent;
}

.default-color2.chart-line-symbol {
    -fx-background-color: transparent, transparent;
}

.default-color3.chart-line-symbol {
    -fx-background-color: transparent, transparent;
}
```

The plot looks like Figure 1-2.

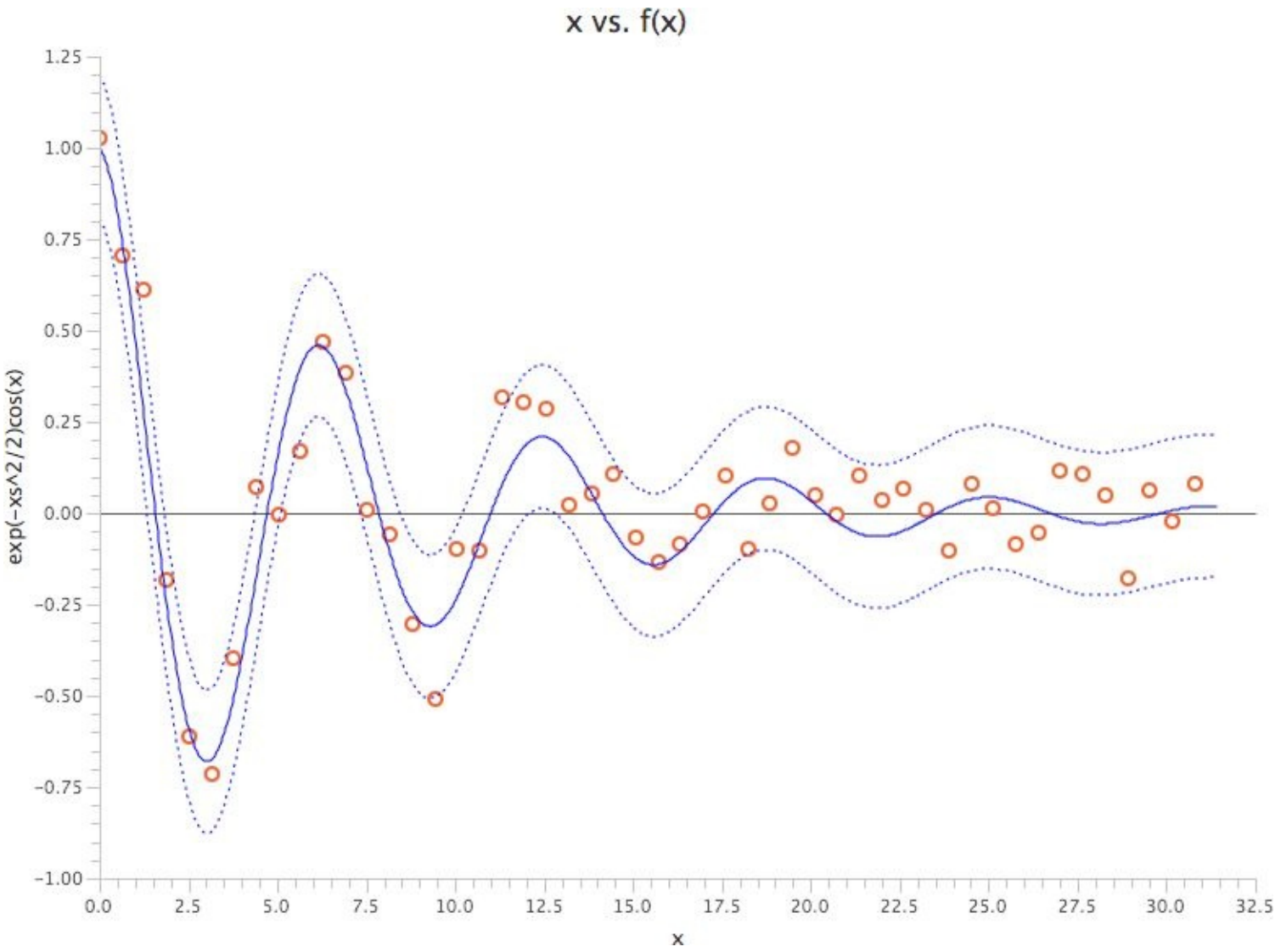

Figure 1-2. Plot of mixed line types with CSS

Saving a Plot to a File

You will undoubtedly have an occasion to save a plot to a file. Perhaps you will be sending the plot off in an email or including it in a presentation. With a mixture of standard Java classes and JavaFX classes, you can easily save plots to any number of formats. With CSS, you can even style your plots to have publication-quality graphics. Indeed, the figures in this chapter (and the rest of the book) were prepared this way.

Each chart type subclasses the abstract class `Chart`, which inherits the method `snapshot()` from the `Node` class. `Chart.snapshot()` returns a `WritableImage`. There is *one* catch that must be addressed: in the time it takes the scene to render the data on the chart, the image will be saved to a file without the actual data on the plot. It is critical to turn off animation via `Chart.setAnimated(false)` someplace after the chart is instantiated and before data is added to the chart with `Chart.getData.add()` or its equivalent:

```
/* do this right after the chart is instantiated */
scatterChart.setAnimated(false);
...
/* render the image */
stage.show();
...
/* save the chart to a file AFTER the stage is rendered */
WritableImage image = scatterChart.snapshot(new SnapshotParameters(), null);
File file = new File("chart.png");
ImageIO.write(SwingFXUtils.fromFXImage(image, null), "png", file);
```

> **NOTE**
>
> All the data plots in this book were created with JavaFX 8.

Chapter 2. Linear Algebra

Now that we have spent a whole chapter acquiring data in some format or another, we will most likely end up viewing the data (in our minds) in the form of spreadsheet. It is natural to envision the names of each column going across from left to right (age, address, ID number, etc.), with each row representing a unique record or data point. Much of data science comes down to this exact formulation. What we are seeking to find is a relationship between any number of columns of interest (which we will call *variables*) and any number of columns that indicate a measurable outcome (which we will call *responses*).

Typically, we use the letter x to denote the variables, and y for the responses. Likewise, the responses can be designated by a matrix **Y** that has a number of columns p and must have the same number of rows m as **X** does. Note that in many cases, there is only one dimension of response variable such that $p = 1$. However, it helps to generalize linear algebra problems to arbitrary dimensions.

In general, the main idea behind linear algebra is to find a relationship between **X** and **Y**. The simplest of these is to ask whether we can multiply **X** by a new matrix of yet-to-be-determined values **W**, such that the result is exactly (or nearly) equal to **Y**. An example of **XW** = **Y** looks like this:

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix} \begin{pmatrix} \omega_{1,1} & \omega_{1,2} & \cdots & \omega_{1,p} \\ \omega_{2,1} & \omega_{2,2} & \cdots & \omega_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{n,1} & \omega_{n,2} & \cdots & \omega_{n,p} \end{pmatrix} = \begin{pmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,p} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m,1} & y_{m,2} & \cdots & y_{m,p} \end{pmatrix}$$

Keep in mind that as the equation is drawn, the sizes of the matrices look similar. This can be misleading, because in most cases the number of data points m is large, perhaps in the millions or billions, while the number of columns n, p for the respective **X** and **Y** matrices is usually much smaller (from tens to hundreds). You will then take notice that regardless of the size of m (e.g., 100,000), the size of the **W** matrix is independent of m; its size is $n \times p$ (e.g., 10 × 10). And this is the heart of linear algebra: that we can explain the contents of extremely large data structures such as **X** and **Y** by using a much more compact data structure **W**. The rules of linear algebra enable us to express any particular value of **Y** in terms of a row of **X** and column of **W**. For example the value of $y_{1,1}$ is written out as follows:

$$y_{1,1} = x_{1,1}\omega_{1,1} + x_{1,2}\omega_{2,1} + \ldots + x_{1,n}\omega_{n,1}$$

In the rest of this chapter, we will work out the rules and operations of linear algebra, and in the final section show the solution to the linear system **XW** = **Y**. More advanced topics in data science such as those presented in Chapters 4 and 5, will rely heavily on the use of linear algebra.

Building Vectors and Matrices

Despite any formal definitions, a *vector* is just a one-dimensional array of a defined length. Many examples may come to mind. You might have an array of integers representing the counts per day of a web metric. Maybe you have a large number of "features" in an array that will be used for input into a machine-learning routine. Or perhaps you are keeping track of geometric coordinates such as *x* and *y*, and you might create an array for each pair `[x,y]`. While we can argue the philosophical meaning of what a vector is (i.e., an element of vector space with magnitude and direction), as long as you are consistent in how you define your vectors throughout the problem you are solving, then all the mathematical formulations will work beautifully, without any concern for the topic of study.

In general, a vector **x** has the following form, comprising *n* components:

$$\mathbf{x} = (x_1 x_2 \ \ldots x_n)$$

Likewise, a matrix **A** is just a two-dimensional array with *m* rows and *n* columns:

$$\mathbf{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

A vector can also be represented in matrix notation as a column vector:

$$\mathbf{X} = \begin{pmatrix} x_{1,1} \\ x_{2,1} \\ \vdots \\ x_{m,1} \end{pmatrix}$$

WARNING

We use bold lowercase letters to represent vectors and use bold uppercase letters to represent matrices. Note that the vector $\mathbf{x}$ can also be represented as a column of the matrix $\mathbf{X}$.

In practice, vectors and matrices are useful to data scientists. A common example is a dataset in which (feature) vectors are stacked on top of each other, and usually the number of rows m is much larger than the number of columns n. In essence, this type of data structure is really a list of vectors, but putting them in matrix form enables efficient calculation of all sorts of linear algebra quantities. Another type of matrix encountered in data science is one in which the components represent a relationship between the variables, such as a covariance or correlation matrix.

Array Storage

The Apache Commons Math library offers several options for creating vectors and matrices of real numbers with the respective `RealVector` and `RealMatrix` classes. Three of the most useful constructor types allocate an empty instance of known dimension, create an instance from an array of values, and create an instance by deep copying an existing instance, respectively. To instantiate an empty, *n*-dimensional vector of type `RealVector`, use the `ArrayRealVector` class with an integer size:

```
int size = 3;
RealVector vector = new ArrayRealVector(size);
```

If you already have an array of values, a vector can be created with that array as a constructor argument:

```
double[] data = {1.0, 2.2, 4.5};
RealVector vector = new ArrayRealVector(data);
```

A new vector can also be created by deep copying an existing vector into a new instance:

```
RealVector vector = new ArrayRealVector(realVector);
```

To set a default value for all elements of a vector, include that value in the constructor along with the size:

```
int size = 3;
double defaultValue = 1.0;
RealVector vector = new ArrayRealVector(size, defaultValue);
```

A similar set of constructors follows for instantiating matrices, an empty matrix of known dimensions is instantiated with the following:

```
int rowDimension = 10;
int colDimension = 20;
RealMatrix matrix = new Array2DRowRealMatrix(rowDimension, colDimension);
```

Or if you already have a two-dimensional array of doubles, you can pass it to the constructor:

```
double[][] data = {{1.0, 2.2, 3.3}, {2.2, 6.2, 6.3}, {3.3, 6.3, 5.1}};
RealMatrix matrix = new Array2DRowRealMatrix(data);
```

Although there is no method for setting the entire matrix to a default value (as there is with a vector), instantiating a new matrix sets all elements to zero, so we can easily add a value to each element afterward:

```
int rowDimension = 10;
int colDimension = 20;
double defaultValue = 1.0;
RealMatrix matrix = new Array2DRowRealMatrix(rowDimension, colDimension);
matrix.scalarAdd(defaultValue);
```

Making a deep copy of a matrix may be performed via the `RealMatrix.copy()` method:

```
/* deep copy contents of matrix */
RealMatrix anotherMatrix = matrix.copy();
```

Block Storage

For large matrices with dimensions greater than 50, it is recommended to use block storage with the `BlockRealMatrix` class. Block storage is an alternative to the two-dimensional array storage discussed in the previous section. In this case, a large matrix is subdivided into smaller blocks of data that are easier to cache and therefore easier to operate on. To allocate space for a matrix, use the following constructor:

```
RealMatrix blockMatrix = new BlockRealMatrix(50, 50);
```

Or if you already have the data in a 2D array, use this constructor:

```
double[][] data = ;
RealMatrix blockMatrix = new BlockRealMatrix(data);
```

Map Storage

When a large vector or matrix is almost entirely zeros, it is termed *sparse*. Because it is not efficient to store all those zeros, only the positions and values of the nonzero elements are stored. Behind the scenes, this is easily achieved by storing the values in a `HashMap`. To create a sparse vector of known dimension, use the following:

```
int dim = 10000;
RealVector sparseVector = new OpenMapRealVector(dim);
```

And to create a sparse matrix, just add another dimension:

```
int rows = 10000;
int cols = 10000;
RealMatrix sparseMatrix = new OpenMapRealMatrix(rows, cols);
```

Accessing Elements

Regardless of the type of storage backing the vector or matrix, the methods for assigning values and later retrieving them are equivalent.

> **CAUTION**
>
> Although the linear algebra theory presented in this book uses an index starting at 1, Java uses a 0-based index system. Keep this in mind as you translate algorithms from theory to code and in particular, when setting and getting values.

Setting and getting values uses the `setEntry(int index, double value)` and `getEntry(int index)` methods:

```
/* set the first value of v */
vector.setEntry(0, 1.2)
/* and get it */
double val = vector.getEntry(0);
```

To set all the values for a vector, use the `set(double value)` method:

```
/* zero the vector */
vector.set(0);
```

However, if **v** is a sparse vector, there is no point to setting all the values. In sparse algebra, missing values are assumed to be zero. Instead, just use `setEntry` to set only the values that are nonzero. To retrieve all the values of an existing vector as an array of doubles, use the `toArray()` method:

```
double[] vals = vector.toArray();
```

Similar setting and getting is provided for matrices, regardless of storage. Use the `setEntry(int row, int column, double value)` and `getEntry(int row, int column)` methods:

```
/* set first row, 3 column to 3.14 */
matrix.setEntry(0, 2, 3.14);
/* and get it */
double val = matrix.getEntry(0, 2);
```

Unlike the vector classes, there is no `set()` method to set all the values of a matrix to one value. As long as the matrix has all entries set to 0, as is the case for a newly constructed matrix, you can set all the entries to one value by adding a constant with code like this:

```
/* for an existing new matrix */
matrix.scalarAdd(defaultValue);
```

Just as with sparse vectors, setting all the values to 0 for each *i*,*j* pair of a sparse matrix is not useful.

To get all the values of a matrix in the form of an array of doubles, use the `getData()` method:

```
double[][] matrixData = matrix.getData();
```

Working with Submatrices

We often need to work with only a specific part of a matrix or want to include a smaller matrix in a larger one. The `RealMatrix` class contains several useful methods for dealing with these common cases. For an existing matrix, there are two ways to create a submatrix from it. The first method selects a rectangular region from the source matrix and uses those entries to create a new matrix. The selected rectangular region is defined by the point of origin, the upper-left corner of the source matrix, and the lower-right corner defining the area that should be included. It is invoked as `RealMatrix.getSubMatrix(int startRow, int endRow, int startColumn, int endColumn)` and returns a `RealMatrix` object with dimensions and values determined by the selection. Note that the `endRow` and `endColumn` values are inclusive.

```
double[][] data = {{1,2,3},{4,5,6},{7,8,9}};
RealMatrix m = new Array2DRowRealMatrix(data);
int startRow = 0;
int endRow = 1;
int startColumn = 1;
int endColumn = 2;
RealMatrix subM = m.getSubMatrix(startRow, endRow, startColumn, endColumn);
// {{2,3},{5,6}}
```

We can also get specific rows and specific columns of a matrix. This is achieved by creating an array of integers designating the row and column indices we wish to keep. The method then takes both of these arrays as `RealMatrix.getSubMatrix(int[] selectedRows, int[] selectedColumns)`. The three use cases are then as follows:

```
/* get selected rows and all columns */
int[] selectedRows = {0, 2};
int[] selectedCols = {0, 1, 2};
RealMatrix subM = m.getSubMatrix(selectedRows, selectedColumns);
// {{1,2,3},{7,8,9}}

/* get all rows and selected columns */
int[] selectedRows = {0, 1, 2};
int[] selectedCols = {0, 2};
RealMatrix subM = m.getSubMatrix(selectedRows, selectedColumns);
// {{1,3},{4,6},{7,9}}

/* get selected rows and selected columns */
int[] selectedRows = {0, 2};
int[] selectedCols = {1};
RealMatrix subM = m.getSubMatrix(selectedRows, selectedColumns);
// {{2},{8}}
```

We can also create a matrix in parts by setting the values of a submatrix. We do this by adding a double array of data to an existing matrix at the coordinates specified by row and column in `RealMatrix.setSubMatrix(double[][] subMatrix, int row, int column)`:

```
double[][] newData = {{-3, -2}, {-1, 0}};
int row = 0;
int column = 0;
m.setSubMatrix(newData, row, column);
// {{-3,-2,3},{-1,0,6},{7,8,9}}
```

Randomization

In learning algorithms, we often want to set all the values of a matrix (or vectors) to random numbers. We can choose the statistical distribution that implements the `AbstractRealDistribution` interface or just go with the easy constructor, which picks random numbers between –1 and 1. We can pass in an existing matrix or vector to fill in the values, or create new instances:

```
public class RandomizedMatrix {

    private AbstractRealDistribution distribution;

    public RandomizedMatrix(AbstractRealDistribution distribution, long seed) {
        this.distribution = distribution;
        distribution.reseedRandomGenerator(seed);
    }

    public RandomizedMatrix() {
        this(new UniformRealDistribution(-1, 1), 0L);
    }

    public void fillMatrix(RealMatrix matrix) {
        for (int i = 0; i < matrix.getRowDimension(); i++) {
            matrix.setRow(i, distribution.sample(matrix.getColumnDimension()));
        }
    }

    public RealMatrix getMatrix(int numRows, int numCols) {
        RealMatrix output = new BlockRealMatrix(numRows, numCols);
        for (int i = 0; i < numRows; i++) {
            output.setRow(i, distribution.sample(numCols));
        }
        return output;
    }

    public void fillVector(RealVector vector) {
        for (int i = 0; i < vector.getDimension(); i++) {
            vector.setEntry(i, distribution.sample());
        }
    }

    public RealVector getVector(int dim) {
        return new ArrayRealVector(distribution.sample(dim));
    }
}
```

We can create a narrow band of normally distributed numbers with this:

```
int numRows = 3;
int numCols = 4;
long seed = 0L;
RandomizedMatrix rndMatrix = new RandomizedMatrix(
    new NormalDistribution(0.0, 0.5), seed);
RealMatrix matrix = rndMatrix.getMatrix(numRows, numCols);

// -0.0217405716,-0.5116704988,-0.3545966969,0.4406692276
// 0.5230193567,-0.7567264361,-0.5376075694,-0.1607391808
// 0.3181005362,0.6719107279,0.2390245133,-0.1227799426
```

Operating on Vectors and Matrices

Sometimes you know the formulation you are looking for in an algorithm or data structure but you may not be sure how to get there. You can do some “mental pattern matching” in your head and then choose to implement (e.g., a dot product instead of manually looping over all the data yourself). Here we explore some common operations used in linear algebra.

Scaling

To scale (multiply) a vector by a constant κ such that

$$\kappa \mathbf{x} = (\kappa x_1, \kappa x_2, \ldots, \kappa x_n)$$

Apache Commons Math implements a mapping method whereby an existing `RealVector` is multiplied by a double, resulting in a new `RealVector` object:

```
double k = 1.2;
RealVector scaledVector = vector.mapMultiply(k);
```

Note that a `RealVector` object may also be scaled in place by altering the existing vector permanently:

```
vector.mapMultiplyToSelf(k);
```

Similar methods exist for dividing the vector by `k` to create a new vector:

```
RealVector scaledVector = vector.mapDivide(k);
```

And for division in place:

```
vector.mapDivideToSelf(k);
```

A matrix **A** can also be scaled by a factor κ:

$$\kappa \mathbf{A} = \begin{pmatrix} \kappa a_{1,1} & \kappa a_{1,2} & \cdots & \kappa a_{1,n} \\ \kappa a_{2,1} & \kappa a_{2,2} & \cdots & \kappa \kappa a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \kappa a_{m,1} & \kappa a_{m,2} & \cdots & \kappa a_{m,n} \end{pmatrix}$$

Here, each value of the matrix is multiplied by a constant of type `double`. A new matrix is returned:

```
double k = 1.2;
RealMatrix scaledMatrix = matrix.scalarMultiply(k);
```

Transposing

Transposing a vector or matrix is analogous to tipping it over on its side. The vector transpose of $\mathbf{x}$ is denoted as $\mathbf{X}^T$. For a matrix, the transpose of $\mathbf{A}$ is denoted as $\mathbf{A}^T$. In most cases, calculating a vector transpose will not be necessary, because the methods of `RealVector` and `RealMatrix` will take into account the need for a vector transpose inside their logic. A vector transpose is undefined unless the vector is represented in matrix format. The transpose of an $m \times 1$ column vector is then a new matrix row vector of dimension $1 \times m$.

$$\mathbf{x}^T = (x_1, x_2 \cdots, x_m)$$

When you absolutely need to transpose a vector, you can simply insert the data into a `RealMatrix` instance. Using a one-dimensional array of `double` values as the argument to the `Array2DRowRealMatrix` class creates a matrix with m rows and one column, where the values are provided by the array of `doubles`. Transposing the column vector will return a matrix with one row and m columns:

```
double[] data = {1.2, 3.4, 5.6};
RealMatrix columnVector = new Array2DRowRealMatrix(data);
System.out.println(columnVector);
/* {{1.2}, {3.4}, {5.6}} */
RealMatrix rowVector = columnVector.transpose();
System.out.println(rowVector);
/* {{1.2, 3.4, 5.6}} */
```

When a matrix of dimension $m \times n$ is transposed, the result is an $n \times m$ matrix. Simply put, the row and column indices i and j are reversed:

$$\mathbf{A}^T = \begin{pmatrix} a_{1,1} & a_{2,1} & \cdots & a_{m,1} \\ a_{1,2} & a_{2,2} & \cdots & a_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,n} & a_{2,n} & \cdots & a_{m,n} \end{pmatrix}$$

Note that the matrix transpose operation returns a new matrix:

```
double[][] data = {{1, 2, 3}, {4, 5, 6}};
RealMatrix matrix = new Array2DRowRealMatrix(data);
RealMatrix transposedMatrix = matrix.transpose();
/* {{1, 4}, {2, 5}, {3, 6}} */
```

Addition and Subtraction

The *addition* of two vectors **a** and **b** of equal length n results in a vector of length n with values equal to the element-wise addition of the vector components:

$$\mathbf{a} + \mathbf{b} = (a_1 + b_1, a_2 + b_2, \ \ldots, a_n + b_n)$$

The result is a new `RealVector` instance:

```
RealVector aPlusB = vectorA.add(vectorB);
```

Similarly, *subtracting* two `RealVector` objects of equal length n is shown here:

$$\mathbf{a} - \mathbf{b} = (a_1 - b_1, a_2 - b_2, \ \ldots, a_n - b_n)$$

This returns a new `RealVector` whose values are the element-wise subtraction of the vector components:

```
RealVector aMinusB = vectorA.subtract(vectorB);
```

Matrices of identical dimensions can also be added and subtracted similarly to vectors:

$$\mathbf{A} + \mathbf{B} = \begin{pmatrix} a_{1,1} + b_{1,1} & a_{1,2} + b_{1,2} & \cdots & a_{1,n} + b_{1,n} \\ a_{2,1} + b_{2,1} & a_{2,2} + b_{2,2} & \cdots & a_{2,n} + b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} + b_{m,1} & a_{m,2} + b_{m,2} & \cdots & a_{m,n} + b_{m,n} \end{pmatrix}$$

The addition or subtraction of `RealMatrix` objects **A** and **B** returns a new `RealMatrix` instance:

```
RealMatrix aPlusB = matrixA.add(matrixB);
RealMatrix aMinusB = matrixA.subtract(matrixB);
```

Length

The *length* of a vector is a convenient way to reduce all of a vector's components to one number and should not be confused with the dimension of the vector. Several definitions of vector length exist; the two most common are the L1 norm and the L2 norm. The L1 norm is useful, for example, in making sure that a vector of probabilities, or fractions of some mixture, all add up to one:

$$|\mathbf{x}| = \sum_{i=1}^{n} |x_i|$$

The L1 norm, which is less common than the L2 norm, is usually referred to by its full name, *L1 norm*, to avoid confusion:

```
double norm = vector.getL1Norm();
```

The L2 norm is usually what is used for normalizing a vector. Many times it is referred to as the *norm* or the *magnitude* of the vector, and it is mathematically formulated as follows:

$$\|\mathbf{x}\| = \sqrt{\sum_{i=1}^{n} |x_i|^2}$$

```
/* calculate the L2 norm of a vector */
double norm = vector.getNorm();
```

> **TIP**
>
> People often ask when to use L1 or L2 vector lengths. In practical terms, it matters what the vector represents. In some cases, you will be collecting counts or probabilities in a vector. In that case, you should normalize the vector by dividing by its sum of parts (L1). On the other hand, if the vector contains some kind of coordinates or features, then you will want to normalize the vector by its Euclidean distance (L2).

The *unit vector* is the direction that a vector points, so called because it has been scaled by its L2 norm to have a length = 1. It is usually denoted with $\hat{x}$ and is calculated as follows:

$$\hat{\mathbf{x}} = \frac{\mathbf{x}}{\|\mathbf{x}\|}$$

The `RealVector.unitVector()` method returns a new `RealVector` object:

```
/* create a new vector that is the unit vector of vector instance*/
RealVector unitVector = vector.unitVector();
```

A vector can also be transformed, in place, to a unit vector. A vector **v** will be permanently changed into its unit vector with the following:

```
/* convert a vector to unit vector in-place */
vector.unitize();
```

We can also calculate the norm of a matrix via the Frobenius norm represented mathematically as the square root of the sum of squares of all elements:

$$\| A \|_F \equiv \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} | a_{ij} |^2}$$

This is rendered in Java with the following:

```
double matrixNorm = matrix.getFrobeniusNorm();
```

Distances

The *distance* between any two vectors **a** and **b** may be calculated in several ways. The L1 distance between **a** and **b** is shown here:

$$d_{L1} = \sum_{i=1}^{n} | a_i - b_i |$$

```
double l1Distance = vectorA.getL1Distance(vectorB);
```

The L2 distance (also known as the Euclidean distance) is formulated as

$$d_{L2} = \sqrt{\sum_{i=1}^{n} | a_i - b_i |^2}$$

This is most often the distance between vectors that is called for. The method `Vector.getDistance(RealVector vector)` returns the Euclidean distance:

```
double l2Distance = vectorA.getDistance(vectorB);
```

The *cosine distance* is a measure between –1 and 1 that is not so much a distance metric as it is a "similarity" measure. If $d = 0$, the two vectors are perpendicular (and have nothing in common). If $d = 1$, the vectors point in the same direction. If $d = –1$, the vectors point in exact opposite directions. The cosine distance may also be thought of as the dot product of two unit vectors:

$$d = \cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{\| \mathbf{a} \| \| \mathbf{b} \|}$$

```
double cosineDistance = vectorA.cosine(vectorB);
```

If both **a** and **b** are unit vectors, the cosine distance is just their inner product:

$$d = \cos(\theta) = \hat{\mathbf{a}} \cdot \hat{\mathbf{b}}$$

and the `Vector.dotProduct(RealVector vector)` method will suffice:

```
/* for unit vectors a and b */
vectorA.unitize();
vectorB.unitize();
double cosineDistance = vectorA.dotProduct(vectorB);
```

Multiplication

The product of an $m \times n$ matrix **A** and an $n \times p$ matrix **B** is a matrix of dimension $m \times p$. The only dimension that must batch is n the number of columns in **A** and the number of rows in **B**:

$$\mathbf{AB} = \begin{pmatrix} (AB)_{1,1} & (AB)_{1,2} & \cdots & (AB)_{1,p} \\ (AB)_{2,1} & (AB)_{2,2} & \cdots & (AB)_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ (AB)_{m,1} & (AB)_{m,2} & \cdots & (AB)_{m,p} \end{pmatrix}$$

The value of each element $(\mathbf{AB})_{ij}$ is the sum of the multiplication of each element of the *i-th* row of **A** and the *j-th* column of **B**, which is represented mathematically as follows:

$$(AB)_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Multiplying a matrix **A** by a matrix **B** is achieved with this:

```
RealMatrix matrixMatrixProduct = matrixA.multiply(matrixB);
```

Note that **AB** ≠ **BA**. To perform **BA**, either do so explicitly or use the `preMultiply` method. Either code has the same result. However, note that in that case, the number of columns of **B** must be equal to the number of rows in **A**:

```
/* BA explicitly */
RealMatrix matrixMatrixProduct matrixB.multiply(matrixA);

/* BA using premultiply */
RealMatrix matrixMatrixProduct = matrixA.preMultiply(matrixB);
```

Matrix multiplication is also commonly called for when multiplying an $m \times n$ matrix **A** with an $n \times 1$ column vector **x**. The result is an $m \times 1$ column vector **b** such that **Ax** = **b**. The operation is performed by summing the multiplication of each element in the *i-th* row of **A** with each element of the vector **x**. In matrix notation:

$$\mathbf{Ax} = \begin{pmatrix} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n \\ \vdots \\ a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n \end{pmatrix}$$

The following code is identical to the preceding matrix-matrix product:

```
/* Ax results in a column vector */
RealMatrix matrixVectorProduct = matrixA.multiply(columnVectorX);
```

We often wish to calculate the vector-matrix product, usually denoted as $\mathbf{x}^T\mathbf{A}$. When **x** is the format of a matrix, we can perform the calculation explicitly as follows:

```
/* x^TA explicitly */
RealMatrix vectorMatrixProduct = columnVectorX.transpose().multiply(matrixA);
```

When **x** is a `RealVector`, we can use the `RealMatrix.preMultiply()` method:

```
/* x^TA with preMultiply */
RealMatrix vectorMatrixProduct = matrixA.preMultiply(columnVectorX);
```

When performing **Ax**, we often want the result as a vector (as opposed to a column vector in a matrix). If **x** is a `RealVector` type, a more convenient way to perform **Ax** is with this:

```
/* Ax */
RealVector matrixVectorProduct = matrixA.operate(vectorX);
```

Inner Product

The *inner product* (also known as the dot product or scalar product) is a method for multiplying two vectors of the same length. The result is a scalar value that is formulated mathematically with a raised dot between the vectors as follows:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i$$

For `RealVector` objects `vectorA` and `vectorB`, the dot product is as follows:

```
double dotProduct = vectorA.dotProduct(vectorB);
```

If the vectors are in matrix form, you can use matrix multiplication, because $\mathbf{a} \cdot \mathbf{b} = \mathbf{a}\mathbf{b}^T$, where the left side is the dot product and the right side is the matrix multiplication:

$$\mathbf{a}^T\mathbf{b} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \end{pmatrix} \begin{pmatrix} b_{1,1} \\ b_{2,1} \\ \vdots \\ b_{n,1} \end{pmatrix}$$

The matrix multiplication of column vectors **a** and **b** returns a 1×1 matrix:

```
/* matrixA and matrixB are both mx1 column vectors */
RealMatrix innerProduct = matrixA.transpose().multiply(matrixB);

/* the result is stored in the only entry for the matrix */
double dotProduct = innerProduct.getEntry(0,0);
```

Although matrix multiplication may not seem practical compared to the dot product, it illustrates an important relationship between vector and matrix operations.

Outer Product

The *outer product* between a vector **a** of dimension m and a vector **b** of dimension n returns a new matrix of dimension $m \times n$:

$$\mathbf{ab}^T = \begin{pmatrix} a_{1,1} \\ a_{2,1} \\ \vdots \\ a_{m,1} \end{pmatrix} \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \end{pmatrix} = \begin{pmatrix} a_{1,1}b_{1,1} & a_{1,1}b_{1,2} & \cdots & a_{1,1}b_{1,n} \\ a_{2,1}b_{1,1} & a_{2,1}b_{1,2} & \cdots & a_{2,1}b_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1}b_{1,1} & a_{m,1}b_{1,2} & \cdots & a_{m,1}b_{1,n} \end{pmatrix}$$

Keep in mind that $\mathbf{ab}^T$ has the dimension $m \times n$ and does not equal $\mathbf{ba}^T$, which has dimension $n \times m$. The `RealMatrix.outerProduct()` method conserves this order and returns a `RealMatrix` instance with the appropriate dimension:

```
/* outer product of vector a with vector b */
RealMatrix outerProduct = vectorA.outerProduct(vectorB);
```

If the vectors are in matrix form, the outer product can be calculated with the `RealMatrix.multiply()` method instead:

```
/* matrixA and matrixB are both nx1 column vectors */
RealMatrix outerProduct = matrixA.multiply(matrixB.transpose());
```

Entrywise Product

Also known as the Hadamard product or the Schur product, the entrywise product multiplies each element of one vector by each element of another vector. Both vectors must have the same dimension, and their resultant vector is therefore of the same dimension:

$$\mathbf{a} \circ \mathbf{b} = (a_1 b_1, a_2 b_2, \ldots, a_n b_n)$$

The method `RealVector.ebeMultiply(RealVector)` performs this operation, in which `ebe` is short for *element by element*.

```
/* compute the entrywise multiplication of vector a and vector b */
RealVector vectorATimesVectorB = vectorA.ebeMultiply(vectorB);
```

A similar operation for entrywise division is performed with `RealVector.ebeDivision(RealVector)`.

> **CAUTION**
>
> Entrywise products should not be confused with matrix products (including inner and outer products). In most algorithms, matrix products are called for. However, the entrywise product will come in handy when, for example, you need to scale an entire vector by a corresponding vector of weights.

The Hadamard product is not currently implemented for matrix-matrix products in Apache Commons Math, but we can easily do so in a naive way with the following:

```
public class MatrixUtils {

    public static RealMatrix ebeMultiply(RealMatrix a, RealMatrix b) {
        int rowDimension = a.getRowDimension();
        int columnDimension = a.getColumnDimension();
        RealMatrix output = new Array2DRowRealMatrix(rowDimension,
            columnDimension);
        for (int i = 0; i < rowDimension; i++) {
            for (int j = 0; j < columnDimension; j++) {
                output.setEntry(i, j, a.getEntry(i, j) * b.getEntry(i, j));
            }
        }
        return output;
    }
}
```

This can be implemented as follows:

```
/* element-by-element product of matrixA and matrixB */
RealMatrix hadamardProduct = MatrixUtils.ebeMultiply(matrixA, matrixB);
```

Compound Operations

You will often run into compound forms involving several vectors and matrices, such as $\mathbf{x}^T\mathbf{A}\mathbf{x}$, which results in a singular, scalar value. Sometimes it is convenient to work the calculation in chunks, perhaps even out of order. In this case, we can first compute the vector $\mathbf{v} = \mathbf{A}\mathbf{x}$ and then find the dot (inner) product $\mathbf{x} \cdot \mathbf{v}$:

```
double[] xData = {1, 2, 3};
double[][] aData = {{1, 3, 1}, {0, 2, 0}, {1, 5, 3}};
RealVector vectorX = new ArrayRealVector(xData);
RealMatrix matrixA = new Array2DRowRealMatrix(aData);
double d = vectorX.dotProduct(matrixA.operate(vectorX));
// d = 78
```

Another method is to first multiply the vector by the matrix by using `RealMatrix.premultiply()` and then compute the inner product (dot product) between the two vectors:

```
double d = matrixA.premultiply(vecotrX).dotProduct(vectorX);
//d = 78
```

If the vectors are in matrix format as column vectors, we can exclusively use matrix methods. However, note that the result will be a matrix as well:

```
RealMatrix matrixX = new Array2DRowRealMatrix(xData);
/* result is 1x1 matrix */
RealMatrix matrixD = matrixX.transpose().multiply(matrixA).multiply(matrixX);
d = matrixD.getEntry(0, 0); // 78
```

Affine Transformation

A common procedure is to transform a vector **x** of length n by applying a linear map matrix **A** of dimensions $n \times p$ and a translation vector **b** of length p, where the relationship

$$f(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$$

is known as an *affine transformation*. For convenience, we can set **z** = *f*(**x**), move the vector **x** to the other side, and define **W** = $\mathbf{A}^T$ with dimensions $p \times n$ such that

$$\mathbf{z} = \mathbf{x}^T\mathbf{W} + \mathbf{b}$$

In particular, we see this form quite a bit in learning and prediction algorithms, where it is important to note that **x** is a multidimensional vector of one observation, not a one-dimensional vector of many observations. Written out, this looks like the following:

$$\begin{pmatrix} z_1 & z_2 & \cdots & z_p \end{pmatrix} = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \end{pmatrix} \begin{pmatrix} \omega_{1,1} & \omega_{1,2} & \cdots & \omega_{1,p} \\ \omega_{2,1} & \omega_{2,2} & \cdots & \omega_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{n,1} & \omega_{n,2} & \cdots & \omega_{n,p} \end{pmatrix} + \begin{pmatrix} \beta_1 & \beta_2 & \cdots & \beta_p \end{pmatrix}$$

We can also express the affine transform of an $m \times n$ matrix **X** with this:

$$\mathbf{Z} = \mathbf{X}\mathbf{W} + \mathbf{B}$$

B has the dimension $m \times p$:

$$\mathbf{Z} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix} \begin{pmatrix} \omega_{1,1} & \omega_{1,2} & \cdots & \omega_{1,p} \\ \omega_{2,1} & \omega_{2,2} & \cdots & \omega_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{n,1} & \omega_{n,2} & \cdots & \omega_{n,p} \end{pmatrix} + \begin{pmatrix} \beta_{1,1} & \beta_{1,2} & \cdots & \beta_{1,p} \\ \beta_{2,1} & \beta_{2,2} & \cdots & \beta_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{m,1} & \beta_{m,2} & \cdots & \beta_{m,p} \end{pmatrix}$$

In most cases, we would like the translation matrix to have equivalent rows of the vector **b** of length p so that the expression is then

$$\mathbf{Z} = \mathbf{X}\mathbf{W} + \mathbf{h}\mathbf{b}^T$$

where **h** is an m-length column vector of ones. Note that the outer product of these two vectors creates an $m \times p$ matrix. Written out, the expression then looks like this:

$$\mathbf{Z} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix} \begin{pmatrix} \omega_{1,1} & \omega_{1,2} & \cdots & \omega_{1,p} \\ \omega_{2,1} & \omega_{2,2} & \cdots & \omega_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{n,1} & \omega_{n,2} & \cdots & \omega_{n,p} \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \begin{pmatrix} \beta_1 & \beta_2 & \cdots & \beta_p \end{pmatrix}$$

This is such an important function that we will include it in our `MatrixOperations` class:

```
public class MatrixOperations {
...
    public static RealMatrix XWplusB(RealMatrix x, RealMatrix w, RealVector b) {
        RealVector h = new ArrayRealVector(x.getRowDimension(), 1.0);
        return x.multiply(w).add(h.outerProduct(b));
    }
...
}
```

Mapping a Function

Often we need to map a function φ over the contents of a vector **z** such that the result is a new vector **y** of the same shape as **z**:

$$\mathbf{y} = \varphi(\mathbf{z})$$

The Commons Math API contains a method `RealVector.map(UnivariateFunction function)`, which does exactly that. Most of the standard and some other useful functions are included in Commons Math that implement the `UnivariateFunction` interface. It is invoked with the following:

```
// map exp over vector input into new vector output
RealVector output = input.map(new Exp());
```

It is straightforward to create your own `UnivariateFunction` classes for forms that are not included in Commons Math. Note that this method does not alter the input vector. If you would like to alter the input vector in place, use this:

```
// map exp over vector input rewriting its values
input.mapToSelf(new Exp());
```

On some occasions, we want to apply a univariate function to each entry of a matrix. The Apache Commons Math API provides an elegant way to do this that works efficiently even for sparse matrices. It is the `RealMatrix.walkInOptimizedOrder(RealMatrixChangingVisitor visitor)` method. Keep in mind, there are other options here. We can visit each entry of the matrix in either row or column order, which may be useful (or required) for some operations. However, if we only want to update each element of a matrix independently, then using the optimized order is the most adaptable algorithm because it will work for matrices with either 2D array, block, or sparse storage. The first step is to build a class (which acts as the mapping function) that extends the `RealMatrixChangingVisitor` interface and implement the required methods:

```
public class PowerMappingFunction implements RealMatrixChangingVisitor {

    private double power;

    public PowerMappingFunction(double power) {
        this.power = power;
    }

    @Override
    public void start(int rows, int columns, int startRow, int endRow,
        int startColumn, int endColumn) {
        // called once before start of operations ... not needed here
    }

    @Override
    public double visit(int row, int column, double value) {
        return Math.pow(value, power);
    }

    @Override
    public double end() {
        // called once after all entries visited ... not needed here
        return 0.0;
    }
```

Then to map the required function over an existing matrix, pass an instance of the class to the `walkInOptimizedOrder()` method like so:

```
/* each element 'x' of matrix is updated in place with x^1.2 */
matrix.walkInOptimizedOrder(new PowerMappingFunction(1.2));
```

We can also utilize Apache Commons Math built-in analytic functions that implement the `UnivariateFunction` interface to easily map any existing function over each entry of a matrix:

```
public class UnivariateFunctionMapper implements RealMatrixChangingVisitor {

    UnivariateFunction univariateFunction;

    public UnivariateFunctionMapper(UnivariateFunction univariateFunction) {
        this.univariateFunction = univariateFunction;
    }

    @Override
    public void start(int rows, int columns, int startRow, int endRow,
            int startColumn, int endColumn) {
        //NA
    }

    @Override
    public double visit(int row, int column, double value) {
        return univariateFunction.value(value);
    }

    @Override
    public double end() {
        return 0.0;
    }
}
```

This interface can be utilized, for example, when extending the affine transformation static method in the preceding section:

```
public class MatrixOperations {
...
    public static RealMatrix XWplusB(RealMatrix X, RealMatrix W, RealVector b,
            UnivariateFunction univariateFunction) {

        RealMatrix z = XWplusB(X, W, b);
        z.walkInOptimizedOrder(new UnivariateFunctionMapper(univariateFunction));
        return z;
    }
...
}
```

So, for example, if we wanted to map the sigmoid (logistic) function over an affine transformation, we would do this:

```
// for input matrix x, weight w and bias b, mapping sigmoid over all entries
MatrixOperations.XWplusB(x, w, b, new Sigmoid());
```

There are a couple of important things to realize here. First, note there is also a *preserving visitor* that visits each element of a matrix but does not change it. The other thing to take note of are the methods. The only method you will really need to implement is the `visit()` method, which should return the new value for each input value. Both the `start()` and `end()` methods are not needed (particularly in this case). The `start()` method is called once before the start of all the operations. So, for example, say we need the

matrix determinant in our further calculations. We could calculate it once in the `start()` method, store it as a class variable, and then use it later in the operations of `visit()`. Similarly, `end()` is called once after all the elements have been visited. We could use this for tallying a running metric, total sites visited, or even an error signal. In any case, the value of `end()` is returned by the method when everything is done. You are not required to include any real logic in the `end()` method, but at the very least you can return a valid double such as 0.0, which is nothing more than a placeholder. Note the method `RealMatrix.walkInOptimizedOrder(RealMatrixChangingVisitor visitor, int startRow, int endRow, int startColumn, int endColumn)`, which operates only on a submatrix whose bounds are indicated by the signature. Use this when you want to update, in-place, only a specific rectangular block of a matrix and leave the rest unchanged.

Decomposing Matrices

Considering what we know about matrix multiplication, it is easy to imagine that any matrix can be decomposed into several other matrices. Decomposing a matrix into parts enables the efficient and numerically stable calculation of important matrix properties. For example, although the matrix inverse and the matrix determinant have explicit, algebraic formulas, they are best calculated by first decomposing the matrix and then taking the inverse. The determinant comes directly from a Cholesky or LU decomposition. All matrix decompositions here are capable of solving linear systems and as a consequence make the matrix inverse available. Table 2-1 lists the properties of various matrix decompositions as implemented by Apache Commons Math.

Table 2-1. Matrix decomposition properties

Decomposition	Matrix Type	Solver	Inverse	Determinant
Cholesky	Symmetric positive definite	Exact	✓	✓
Eigen	Square	Exact	✓	✓
LU	Square	Exact	✓	✓
QR	Any	Least squares	✓	
SVD	Any	Least squares	✓	

Cholesky Decomposition

A *Cholesky decomposition* of a matrix **A** decomposes the matrix such that $\mathbf{A} = \mathbf{L}\mathbf{L}^T$, where **L** is a lower triangular matrix, and the upper triangle (above the diagonal) is zero:

$$\mathbf{A} = \begin{pmatrix} l_{1,1} & 0 & \cdots & 0 \\ l_{2,1} & l_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & l_{n,n} \end{pmatrix} \begin{pmatrix} l_{1,1} & l_{1,2} & \cdots & l_{1,n} \\ 0 & l_{2,2} & \cdots & l_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & l_{n,n} \end{pmatrix}$$

```
CholeskyDecomposition cd = new CholeskyDecomposition(matrix);
RealMatrix l = cd.getL();
```

A Cholesky decomposition is valid only for symmetric matrices. The main use of a Cholesky is in the computation of random variables for the multinormal distribution.

LU Decomposition

The *lower-upper (LU) decomposition* decomposes a matrix **A** into a lower diagonal matrix **L** and an upper diagonal matrix **U** such that **A** = **LU**:

$$\mathbf{A} = \begin{pmatrix} l_{1,1} & 0 & \cdots & 0 \\ l_{2,1} & l_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & l_{n,n} \end{pmatrix} \begin{pmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ 0 & u_{2,2} & \cdots & u_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{n,n} \end{pmatrix}$$

```
LUDecomposition lud = new LUDecomposition(matrix);
RealMatrix u = lud.getU();
RealMatrix l = lud.getL();
```

The LU decomposition is useful in solving systems of linear equations in which the number of unknowns is equal to the number of equations.

QR Decomposition

The *QR decomposition* decomposes the matrix **A** into an orthogonal matrix of column unit vectors **Q** and an upper triangular matrix **R** such that

$$\mathbf{A} = \begin{pmatrix} q_{1,1} & q_{1,2} & \cdots & q_{1,m} \\ q_{2,1} & q_{2,2} & \cdots & q_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ q_{m,1} & q_{m,2} & \cdots & q_{m,m} \end{pmatrix} \begin{pmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,n} \\ 0 & r_{2,2} & \cdots & r_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & r_{m,n} \end{pmatrix}$$

```
QRDecomposition qrd = new QRDecomposition(matrix);
RealMatrix q = lud.getQ();
RealMatrix r = lud.getR();
```

One of the main uses of the QR decomposition (and analogous decompositions) is in the calculation of eigenvalue decompositions because each column of **Q** is orthogonal. The QR decomposition is also useful in solving overdetermined systems of linear equations. This is usually the case for datasets in which the number of data points (rows) is greater than the dimension (number of columns). One advantage of using the QR decomposition solver (as opposed to SVD) is the easy access to the errors on the solution parameters that can be directly calculated from R.

Singular Value Decomposition

The singular value decomposition (SVD) decomposes the $m \times n$ matrix $\mathbf{A}$ such that $\mathbf{A} = \mathbf{U\Sigma V}^{\mathrm{T}}$, where $\mathbf{U}$ is an $m \times m$ unitary matrix, $\mathbf{S}$ is an $m \times n$ diagonal matrix with real, non-negative values, and $\mathbf{V}$ is an $n \times n$ unitary matrix. As unitary matrices, both $\mathbf{U}$ and $\mathbf{V}$ have the property $\mathbf{UU}^{\mathrm{T}} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix.

$$\mathbf{A} = \begin{pmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,m} \\ u_{2,1} & u_{2,2} & \cdots & u_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ u_{m,1} & u_{m,2} & \cdots & u_{m,m} \end{pmatrix} \begin{pmatrix} s_{1,1} & 0 & \cdots & 0 & \cdots & 0 \\ 0 & s_{2,2} & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & & \\ 0 & 0 & \cdots & s_{n,n} & \cdots & 0_{m,n} \end{pmatrix} \begin{pmatrix} v_{1,1} & v_{2,1} & \cdots & v_{n,1} \\ v_{1,2} & v_{2,2} & \cdots & v_{n,2} \\ \vdots & \vdots & \ddots & \vdots \\ v_{1,n} & v_{2,n} & \cdots & v_{n,n} \end{pmatrix}$$

In many cases, $m \geq n$; the number of rows in a matrix will be greater than or equal to the number of columns. In this case, there is no need to calculate the full SVD. Instead, a more efficient calculation called *thin SVD* can be implemented, where $\mathbf{U}$ is $m \times n$, $\mathbf{S}$ is $n \times n$, and $\mathbf{V}$ is $n \times n$. As a practical matter, there may also be cases when $m \leq n$ so we can then just use the smaller of the two dimensions: $p = min(m, n)$. The Apache Commons Math implementation uses that practice:

```
/* matrix is mxn and p = min(m,n) */
SingularValueDecomposition svd = new SingularValueDecomposition(matrix);
RealMatrix u = svd.getU(); // m x p
RealMatrix s = svd.getS(); // p x p
RealMatrix v = svd.getV(); // p x n
/* retrieve values, in decreasing order, from the diagonal of S */
double[] singularValues = svd.getSingularValues();
/* can also get covariance of input matrix */
double minSingularValue = 0;// 0 or neg value means all sv are used
RealMatrix cov = svd.getCovariance(minSingularValue);
```

The singular value decomposition has several useful properties. Like the eigen decomposition, it is used to reduce the matrix $\mathbf{A}$ to a smaller dimension, keeping only the most useful of them. Also, as a linear solver, the SVD works on any shape of matrix and in particular, is stable on underdetermined systems in which the number of dimensions (columns) is much greater than the number of data points (rows).

Eigen Decomposition

The goal of the eigen decomposition is to reorganize the matrix **A** into a set of independent and orthogonal column vectors called *eigenvectors*. Each eigenvector has an associated eigenvalue that can be used to rank the eigenvectors from most important (highest eigenvalue) to least important (lowest eigenvalue). We can then choose to use only the most significant eigenvectors as representatives of matrix **A**. Essentially, we are asking, is there some way to completely (or mostly) describe matrix **A**, but with fewer dimensions?

For a matrix **A**, a solution exists for a vector **x** and a constant λ such that **Ax** = λ**x**. There can be multiple solutions (i.e., **x**, λ pairs). Taken together, all of the possible values of lambda are known as the eigenvalues, and all corresponding vectors are known as the eigenvectors. The eigen decomposition of a symmetric, real matrix **A** is expressed as $\mathbf{A} = \mathbf{V}\mathbf{D}\mathbf{V}^{\mathrm{T}}$. The results are typically formulated as a diagonal $m \times m$ matrix **D**, in which the eigenvalues are on the diagonal and an $m \times m$ matrix **V** whose column vectors are the eigenvectors.

$$\mathbf{A} = \begin{pmatrix} v_{1,1} & v_{1,2} & \cdots & v_{1,m} \\ v_{2,1} & v_{2,2} & \cdots & v_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ v_{m,1} & v_{m,2} & \cdots & v_{m,m} \end{pmatrix} \begin{pmatrix} d_{1,1} & 0 & \cdots & 0 \\ 0 & d_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_{m,m} \end{pmatrix} \begin{pmatrix} v_{1,1} & v_{2,1} & \cdots & v_{m,1} \\ v_{1,2} & v_{2,2} & \cdots & v_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ v_{1,m} & v_{2,m} & \cdots & v_{m,m} \end{pmatrix}$$

Several methods exist for performing an eigenvalue decomposition. In a practical sense, we usually need only the simplest form as implemented by Apache Commons Math in the `org.apache.commons.math3.linear.EigenDecomposition` class. The eigenvalues and eigenvectors are sorted by descending order of the eigenvalues. In other words, the first eigenvector (the zeroth column of matrix Q) is the most significant eigenvector.

```
double[][] data = {{1.0, 2.2, 3.3}, {2.2, 6.2, 6.3}, {3.3, 6.3, 5.1}};
RealMatrix matrix = new Array2DRowRealMatrix(data);

/* compute eigenvalue matrix D and eigenvector matrix V */
EigenDecomposition eig = new EigenDecomposition(matrix);

/* The real (or imag) eigenvalues can be retrieved as an array of doubles */
double[] eigenValues = eig.getRealEigenvalues();

/* Individual eigenvalues can be also be accessed directly from D */
double firstEigenValue = eig.getD().getEntry(0, 0);

/* The first eigenvector can be accessed like this */
RealVector firstEigenVector = eig.getEigenvector(0);

/* Remember that eigenvectors are just the columns of V */
RealVector firstEigenVector = eig.getV.getColumn(0);
```

Determinant

The *determinant* is a scalar value calculated from a matrix **A** and is most often seen as a component as the multinormal distribution. The determinant of matrix **A** is denoted as |**A**|. The Cholesky, eigen, and LU decomposition classes provide access to the determinant:

```
/* calculate determinant from the Cholesky decomp */
double determinant = new CholeskyDecomposition(matrix).getDeterminant();

/* calculate determinant from the eigen decomp */
double determinant = new EigenDecomposition(matrix).getDeterminant();

/* calculate determinant from the LU decomp */
double determinant = new LUDecomposition(matrix).getDeterminant();
```

Inverse

The *inverse* of a matrix is similar to the concept of inverting a real number $\Re$, where $\Re(1/\Re) = 1$. Note that this can also be written as $\Re\Re^{-1} = 1$. Similarly, the inverse of a matrix $\mathbf{A}$ is denoted by $\mathbf{A}^{-1}$ and the relation exists $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix. Although formulas exist for directly computing the inverse of a matrix, they are cumbersome for large matrices and numerically unstable. Each of the decomposition methods available in Apache Commons Math implements a `DecompositionSolver` interface that requires a matrix inverse in its solution of linear systems. The matrix inverse is then retrieved from the accessor method of the `DecompositionSolver` class. Any of the decomposition methods provides a matrix inverse if the matrix type is compatible with the method used:

```
/* the inverse of a square matrix from Cholesky, LU, Eigen, QR,
      or SVD decompositions */
RealMatrix matrixInverse = new LUDecomposition(matrix).getSolver().getInverse();
```

The matrix inverse can also be calculated from the singular value decomposition:

```
/* the inverse of a square or rectangular matrix from QR or SVD decomposition */
RealMatrix matrixInverse =
new SingularValueDecomposition(matrix).getSolver().getInverse();
```

Or the QR decomposition:

```
/* OK on rectangular matrices, but error on non-singular matrices */
RealMatrix matrixInverse = new QRDecomposition(matrix).getSolver().getInverse();
```

A matrix inverse is used whenever matrices are moved from one side of the equation to the other via division. Another common application is in the computation of the Mahalanobis distance and, by extension, for the multinormal distribution.

Solving Linear Systems

At the beginning of this chapter, we described the system **XW** = **Y** as a fundamental concept of linear algebra. Often, we also want to include an intercept or offset term β not dependent on x such that

$$y_{1,1} = \beta + x_{1,1}\omega_{1,1} + x_{1,2}\omega_{2,1} + \cdots + x_{1,n}\omega_{n,1}$$

There are two options for including the intercept term, the first of which is to add a column of 1s to **X** and a row of unknowns to **W**. It does not matter which column-row pair is chosen as long as $j = i$ in this case. Here we choose the last column of **X** and the last row of **W**:

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} & 1 \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} & 1 \end{pmatrix} \begin{pmatrix} \omega_{1,1} & \omega_{1,2} & \cdots & \omega_{1,p} \\ \omega_{2,1} & \omega_{2,2} & \cdots & \omega_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{n+1,1} & \omega_{n+1,2} & \cdots & \omega_{n+1,p} \end{pmatrix} = \begin{pmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,p} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m,1} & y_{m,2} & \cdots & y_{m,p} \end{pmatrix}$$

Note that in this case, the columns of **W** are independent. Therefore, we are simply finding p separate linear models, except for the convenience of performing the operation in one piece of code:

```
/* data */
double[][] xData = {{0, 0.5, 0.2}, {1, 1.2, .9}, {2, 2.5, 1.9}, {3, 3.6, 4.2}};
double[][] yData = {{-1, -0.5}, {0.2, 1}, {0.9, 1.2}, {2.1, 1.5}};

/* create X with offset as last column */
double[] ones = {1.0, 1.0, 1.0, 1.0};
int xRows = 4;
int xCols = 3;
RealMatrix x = new Array2DRowRealMatrix(xRows, xCols + 1);
x.setSubMatrix(xData, 0, 0);
x.setColumn(3, ones); // 4th column is index of 3 !!!

/* create Y */
RealMatrix y = new Array2DRowRealMatrix(yData);

/* find values for W */
SingularValueDecomposition svd = new SingularValueDecomposition(x);
RealMatrix solution = svd.getSolver().solve(y);
System.out.println(solution);
// {{1.7,3.1},{-0.9523809524,-2.0476190476},
//  {0.2380952381,-0.2380952381},{-0.5714285714,0.5714285714}}
```

Given the values for the parameters, the solution for the system of equations is as follows:

$$y_1 = 1.7x_1 - 0.95x_2 + 0.24x_3 - 0.57$$
$$y_2 = 3.1x_1 - 2.05x_2 - 0.24x_3 + 0.57$$

The second option for including the intercept is to realize that the preceding algebraic expression is equivalent to the affine transformation of a matrix described earlier in this chapter:

$$\mathbf{Y} = \mathbf{XW} + \mathbf{hb}^T$$

This form of a linear system has the advantage that we do not need to resize any matrices. In the previous example code resizing the matrices occurs only one time, and this is not too much of a burden. However, in Chapter 5, we will tackle a multilayered linear model (deep network) in which resizing matrices will be cumbersome and inefficient. In that case, it is much more convenient to represent the linear model in algebraic terms, where **W** and **b** are completely separate.

Chapter 3. Statistics

Applying the basic principles of statistics to data science provides vital insight into our data. Statistics is a powerful tool. Used correctly, it enables us to be sure of our decision-making process. However, it is easy to use statistics incorrectly. One example is Anscombe's quartet (Figure 3-1), which demonstrates how four distinct datasets can have nearly identical statistics. In many cases, a simple plot of the data can alert us right away to what is really going on with the data. In the case of Anscombe's quartet, we can instantly pick out these features: in the upper-left panel, x and y appear to be linear, but noisy. In the upper-right panel, we see that x and y form a peaked relationship that is nonlinear. In the lower-left panel, x and y are precisely linear, except for one outlier. The lower-right panel shows that y is statistically distributed for $x = 8$ and that there is possibly an outlier at $x = 19$. Despite how different each plot looks, when we subject each set of data to standard statistical calculations, the results are identical. Clearly, our eyes are the most sophisticated data-processing tool in existence! However, we cannot always visualize data in this manner. Many times the data will be multidimensional in x and perhaps y as well. While we can plot each dimension of x versus y to get some ideas on the characteristics of the dataset, we will be missing all of the dependencies between the variates in x.

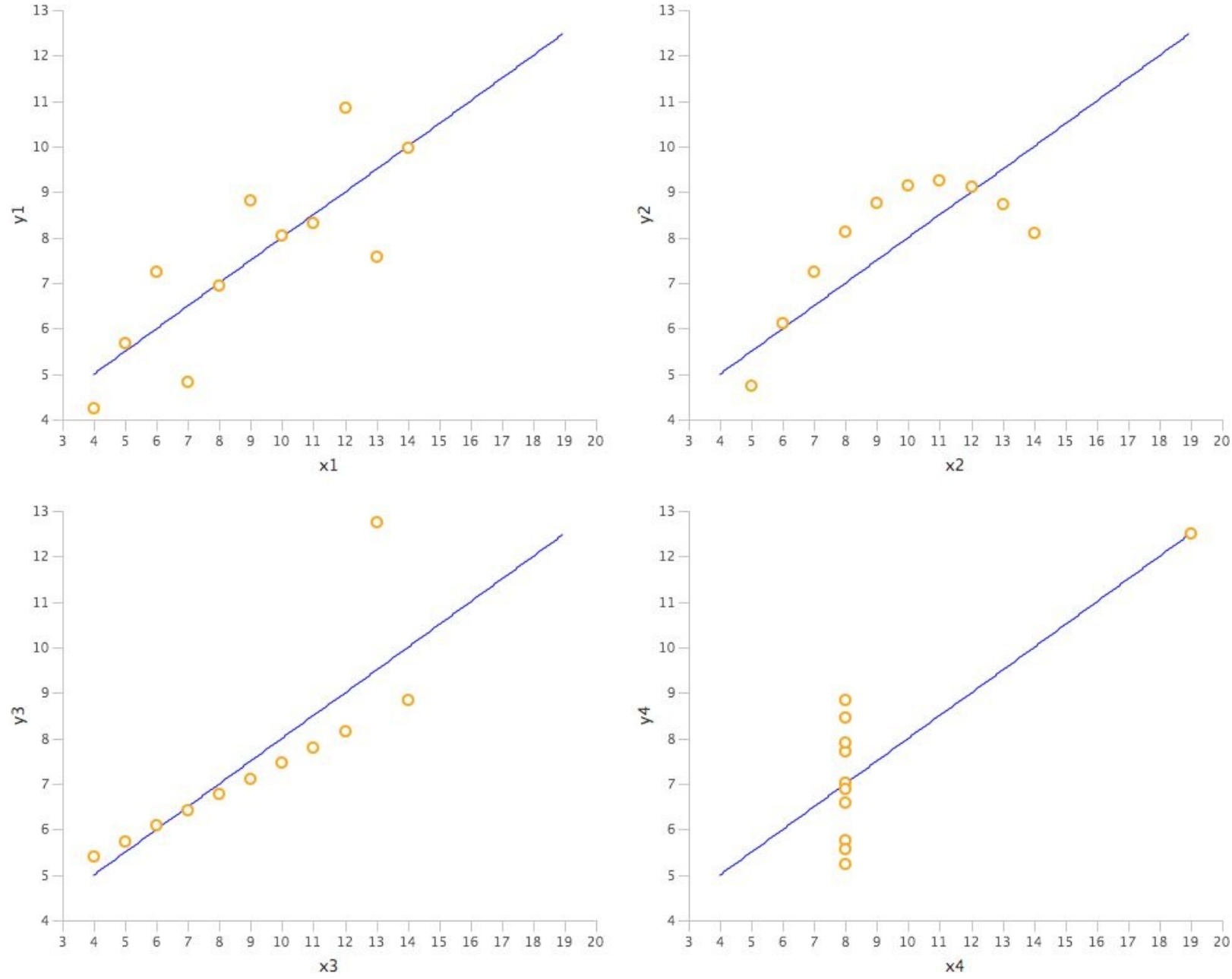

Figure 3-1. Anscombe's quartet

The Probabilistic Origins of Data

At the beginning of the book, we defined a *data point* as a recorded event that occurs at that exact time and place. We can represent a datum (data point) with the dirac delta $\delta(\mathbf{x})$, which is equal to zero everywhere except at $\mathbf{x} = 0$, where the value is ∞. We can generalize a little further with $\delta(x - x_i)$, which means that the dirac delta is equal to zero everywhere except at $x = x_i$, where the value is ∞. We can ask the question, is there something driving the occurrence of the data points?

Probability Density

Sometimes data arrives from a well-known, generating source that can be described by a functional form $f(x)$, where typically the form is modified by some parameters θ and is denoted as $f(x;\ \theta)$. Many forms of $f(x)$ exist, and most of them come from observations of behavior in the natural world. We will explore some of the more common ones in the next few sections for both continuous and discrete random-number distributions.

We can add together all those probabilities for each position as a function of the variate *x*:

$$f(x) = \sum_{i=1}^{n} p_i \delta(x - x_i)$$

Or for a discrete integer variate, *k*:

$$f(x) = f(k)\delta(x - k)$$

Note that *f(x)* can be greater than 1. Probability density is not the probability, but rather, the local density. To determine the probability, we must integrate the probability density over an arbitrary range of *x*. Typically, we use the cumulative distribution function for this task.

Cumulative Probability

We require that probability distribution functions (PDFs) are properly normalized such that integrating over all space returns a 100 percent probability that the event has occurred:

$$F = \int_{-\infty}^{\infty} f(x)\,\mathrm{d}x = 1$$

However, we can also calculate the cumulative probability that an event will occur at point *x*, given that it has not occurred yet:

$$F(x) = \int_{-\infty}^{x} f(x')\,\mathrm{d}x'$$

Note that the cumulative distribution function is monotonic (always increasing as *x* increases) and is (almost) always a sigmoid shape (a slanted *S*). Given that an event has not occurred yet, what is the probability that it will occur at *x*? For large values of *x*, P = 1. We impose this condition so that we can be sure that the event definitely happens in some defined interval.

Statistical Moments

Although integrating over a known probability distribution $f(x)$ gives the cumulative distribution function (or 1 if over all space), adding in powers of *x* are what defines the statistical moments. For a known statistical distribution, the statistical moment is the expectation of order *k* around a central point *c* and can be evaluated via the following:

$$\mu_k = \int_{-\infty}^{\infty} (x - c)^k f(x)\, \mathrm{d}x$$

Special quantity, the expectation or average value of *x*, occurs at c = 0 for the first moment *k* = 1:

$$\mu = \int_{-\infty}^{\infty} x f(x)\, \mathrm{d}x$$

The higher-order moments, *k* > 1, with respect to this mean are known as the *central moments* about the mean and are related to descriptive statistics. They are expressed as follows:

$$\mu_{k>1} = \int_{-\infty}^{\infty} (x - \mu)^k f(x)\, \mathrm{d}x$$

The second, third, and fourth central moments about the mean have useful statistical meanings. We define the variance σ^2 as the second moment:

$$\sigma^2 = \mu_2$$

Its square root is the standard deviation σ, a measure of how far the data is distributed from the mean. The skewness γ is a measure of how asymmetric the distribution is and is related to the third central moment about the mean:

$$\gamma = \frac{\mu_3}{\sigma^3}$$

The *kurtosis* is a measure of how fat the tails of the distribution are and is related to the fourth central moment about the mean:

$$\kappa = \frac{\mu_4}{\sigma^4}$$

In the next section, we will examine the normal distribution, one of the most useful and ubiquitous probability distributions. The normal distribution has a kurtosis kappa = 3. Because we often compare things to the normal distribution, the term *excess kurtosis* is defined by the following:

$$\kappa = \frac{\mu_4}{\sigma^4} - 3$$

We now define *kurtosis* (the fatness of the tails) in reference to the normal distribution. Note that many references to the kurtosis are actually referring to the excess kurtosis. The two terms are used interchangeably.

Higher-order moments are possible and have varied usage and applications on the fringe of data science. In this book, we stop at the fourth moment.

Entropy

In statistics, *entropy* is the measure of the unpredictability of the information contained within a distribution. For a continuous distribution, the entropy is as follows:

$$\mathscr{H}(p) = \int_{-\infty}^{\infty} p(x)\log_b(p(x))\,\mathrm{d}x$$

For a discrete distribution, entropy is shown here:

$$\mathscr{H}(p) = -\sum_i p(x_i)\log_b(p(x_i))$$

In an example plot of entropy (Figure 3-2), we see that entropy is lowest when the probability of a 0 or 1 is high, and the entropy is maximal at $p = 0.5$, where both 0 and 1 are just as likely.

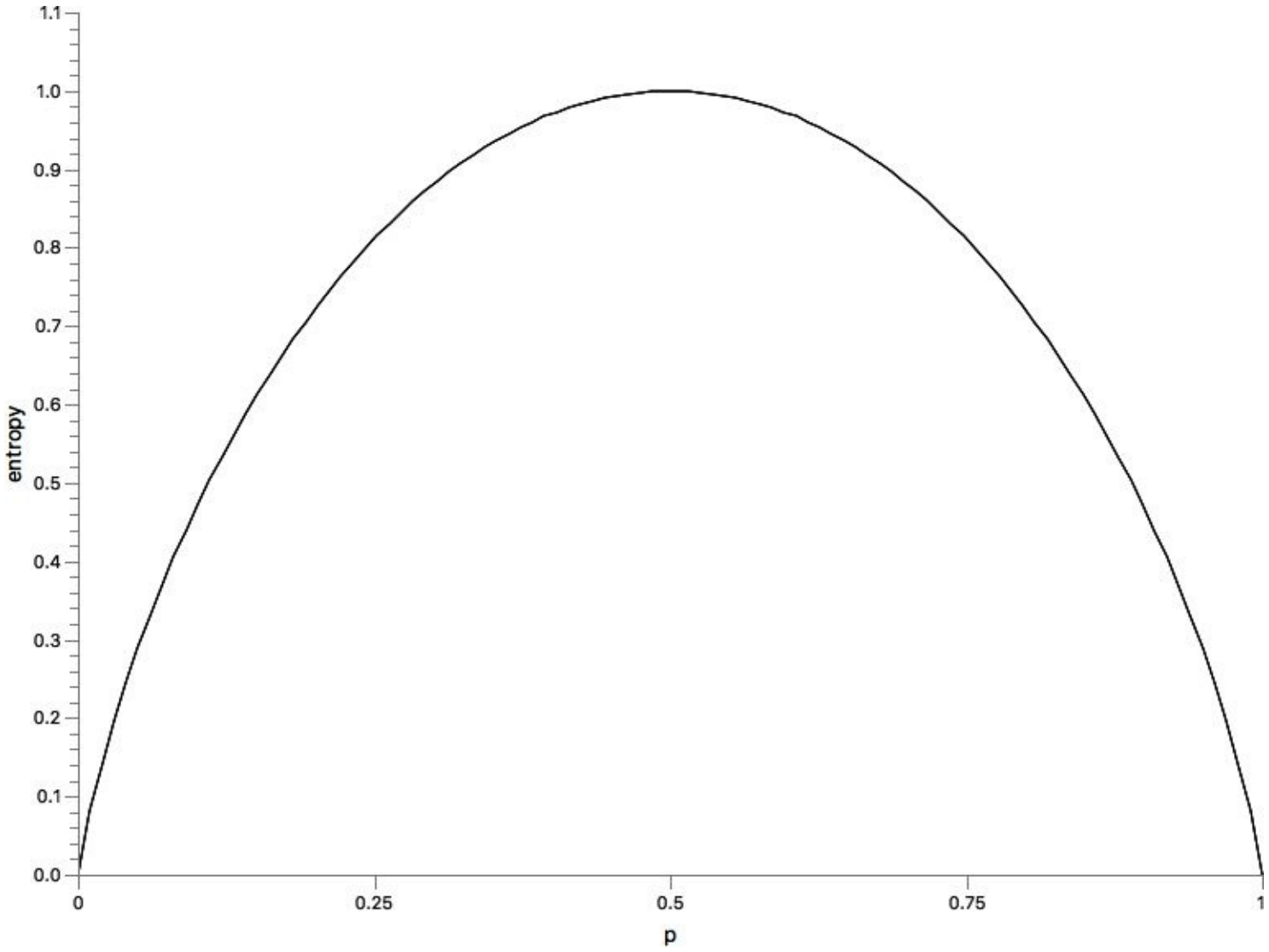

Figure 3-2. Entropy for a Bernoulli distribution

We can also examine the entropy between two distributions by using cross entropy, where *p(x)* is taken as the true distribution, and *q(x)* is the test distribution:

$$\mathcal{H}(p,\ q) = \int_{-\infty}^{\infty} p(x)\log_b(q(x))\,\mathrm{d}x$$

And in the discrete case:

$$\mathcal{H}(p,\ q) = -\sum_{i} p(x_i)\log_b(q(x_i))$$

Continuous Distributions

Some well-known forms of distributions are well characterized and get frequent use. Many distributions have arisen from real-world observations of natural phenomena. Regardless of whether the variates are described by real or integer numbers, indicating respective continuous and discrete distributions, the basic principles for determining the cumulative probability, statistical moments, and statistical measures are the same.

Uniform

The *uniform distribution* has a constant probability density over its supported range $x \in [a, b]$, and is zero everywhere else. In fact, this is just a formal way of describing the random, real-number generator on the interval `[0,1]` that you are familiar with, such as `java.util.Random.nextDouble()`. The default constructor sets a lower bound $a = 0.0$ and an upper bound of $b = 1.0$. The probability density of a uniform distribution is expressed graphically as a *top hat* or *box* shape, as shown in Figure 3-3.

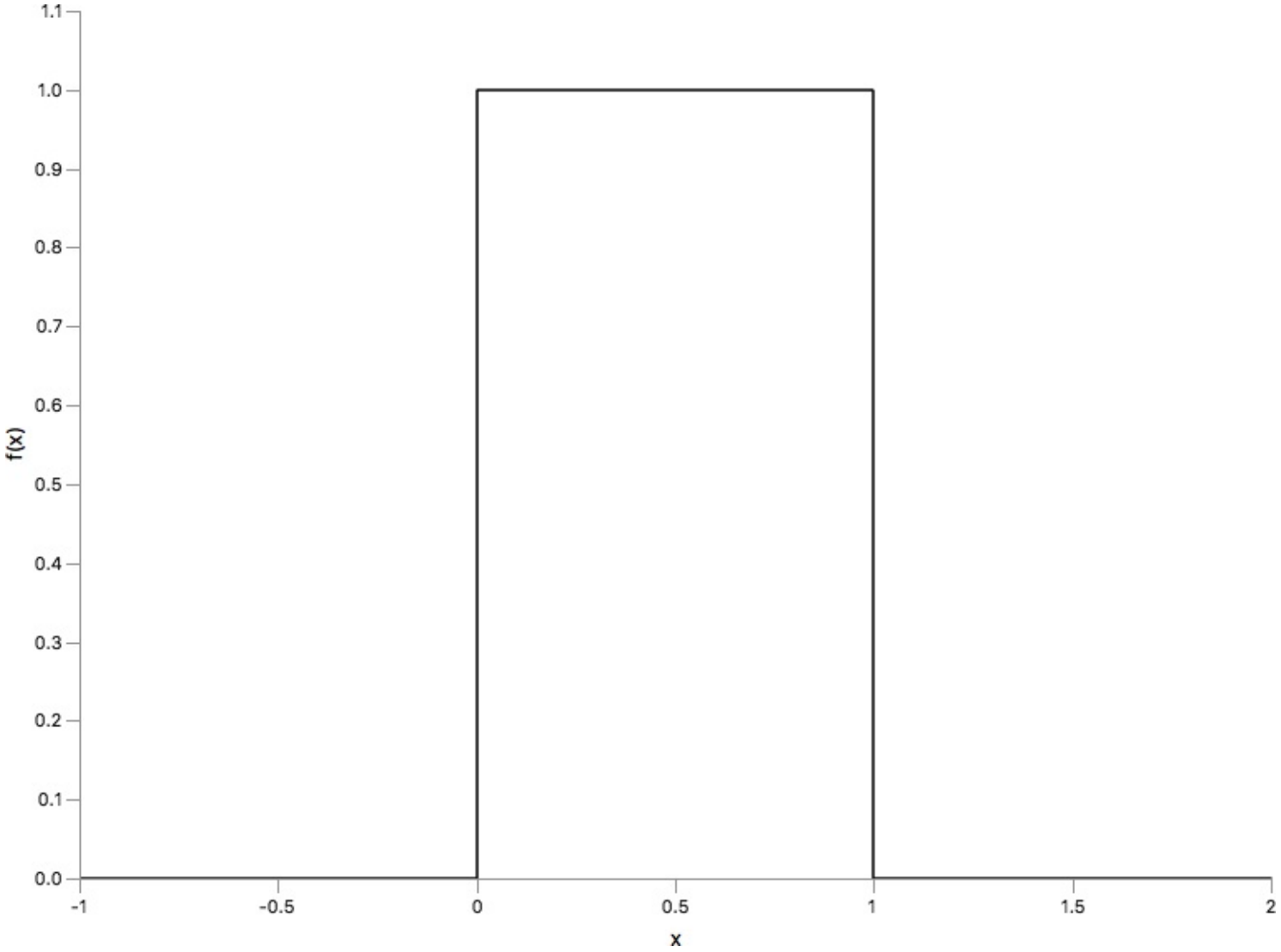

Figure 3-3. Uniform PDF with parameters $a = 0$ *and* $b = 1$

This has the following mathematical form:

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{for } x \in [a, b] \\ 0 & \text{otherwise} \end{cases}$$

The cumulative distribution function (CDF) looks like Figure 3-4.

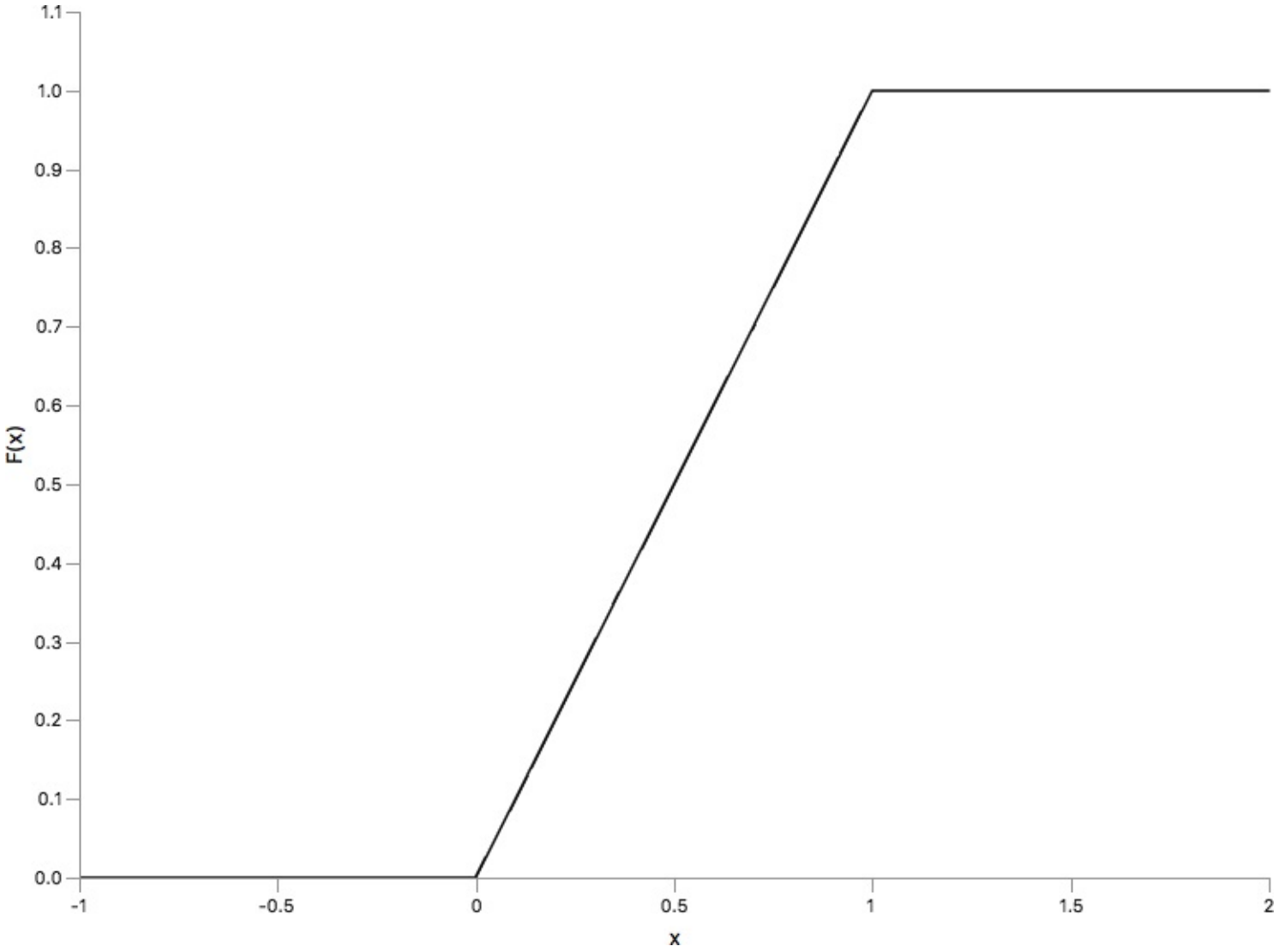

Figure 3-4. Uniform CDF with parameters $a = 0$ *and* $b = 1$

This has the form shown here:

$$F(x) = \begin{cases} 0 & \text{for } x < a \\ \frac{x-a}{b-a} & \text{for } x \in [a,\ b) \\ 1 & \text{for } x \geq b \end{cases}$$

In the uniform distribution, the mean and the variance are not directly specified, but are calculated from the lower and upper bounds with the following:

$$\mu = \frac{1}{2}(a+b)$$

$$\sigma^2 = \frac{1}{12}(b-a)^2$$

To invoke the uniform distribution with Java, use the class `UniformDistribution(a, b)`, where the lower and upper bounds are specified in the constructor. Leaving the constructor arguments blank invokes the standard uniform distribution, where $a = 0.0$ and $b = 1.0$.

```
UniformRealDistribution dist = new UniformRealDistribution();
double lowerBound = dist.getSupportLowerBound(); // 0.0
double upperBound = dist.getSupportUpperBound(); // 1.0
double mean = dist.getNumericalMean();
double variance = dist.getNumericalVariance();
double standardDeviation = Math.sqrt(variance);
double probability = dist.density(0.5));
double cumulativeProbability = dist.cumulativeProbability(0.5);
double sample = dist.sample(); // e.g., 0.023
double[] samples = dist.sample(3); // e.g., {0.145, 0.878, 0.431}
```

Note that we could reparameterize the uniform distribution with $a = \mu - \delta$ and $b = \mu + \delta$, where μ is the center point (the mean) and δ is the distance from the center to either the lower or upper bound. The variance then becomes $\sigma^2 = \frac{\delta^2}{3}$ with standard deviation $\sigma = \frac{\delta}{\sqrt{3}}$. The PDF is then

$$f(x) = \begin{cases} \frac{1}{2\delta} & \text{for } x \in [\mu - \delta,\ \mu + \delta] \\ 0 & \text{otherwise} \end{cases}$$

and the CDF is

$$F(x) = \begin{cases} 0 & \text{for } x < \mu - \delta \\ \frac{1}{2}(1 + \frac{x - \mu}{\delta}) & \text{for } x \in [\mu - \delta, \mu + \delta] \\ 1 & \text{for } x \geq \mu + \delta \end{cases}$$

To express the uniform distribution in this centralized form, calculate $a = \mu - \delta$ and $b = \mu + \delta$ and enter them into the constructor:

```
/* initialize centralized uniform with mean = 10 and half-width = 2 */
double mean = 10.0
double hw = 2.0;
double a = mean - hw;
double b = mean + hw;
UniformRealDistribution dist = new UniformRealDistribution(a, b);
```

At this point, all of the methods will return the correct results without any further alterations. This reparameterization around the mean can be useful when trying to compare distributions. The centered, uniform distribution is naturally extended by the normal distribution (or other symmetric peaked distributions).

Normal

The most useful and widespread distribution, found in so many diverse use cases, is the normal distribution. Also known as the *Gaussian distribution* or the *bell curve*, this distribution is symmetric about a central peak whose width can vary. In many cases when we refer to something as having an average value with plus or minus a certain amount, we are referring to the normal distribution. For example, for exam grades in a classroom, the interpretation is that a few people do really well and a few people do really badly, but most people are average or right in the middle. In the normal distribution, the center of the distribution is the maximum peak and is also the mean of the distribution, μ. The width is parameterized by σ and is the standard deviation of the values. The distribution supports all values of $x \in [-\infty, \infty]$ and is shown in Figure 3-5.

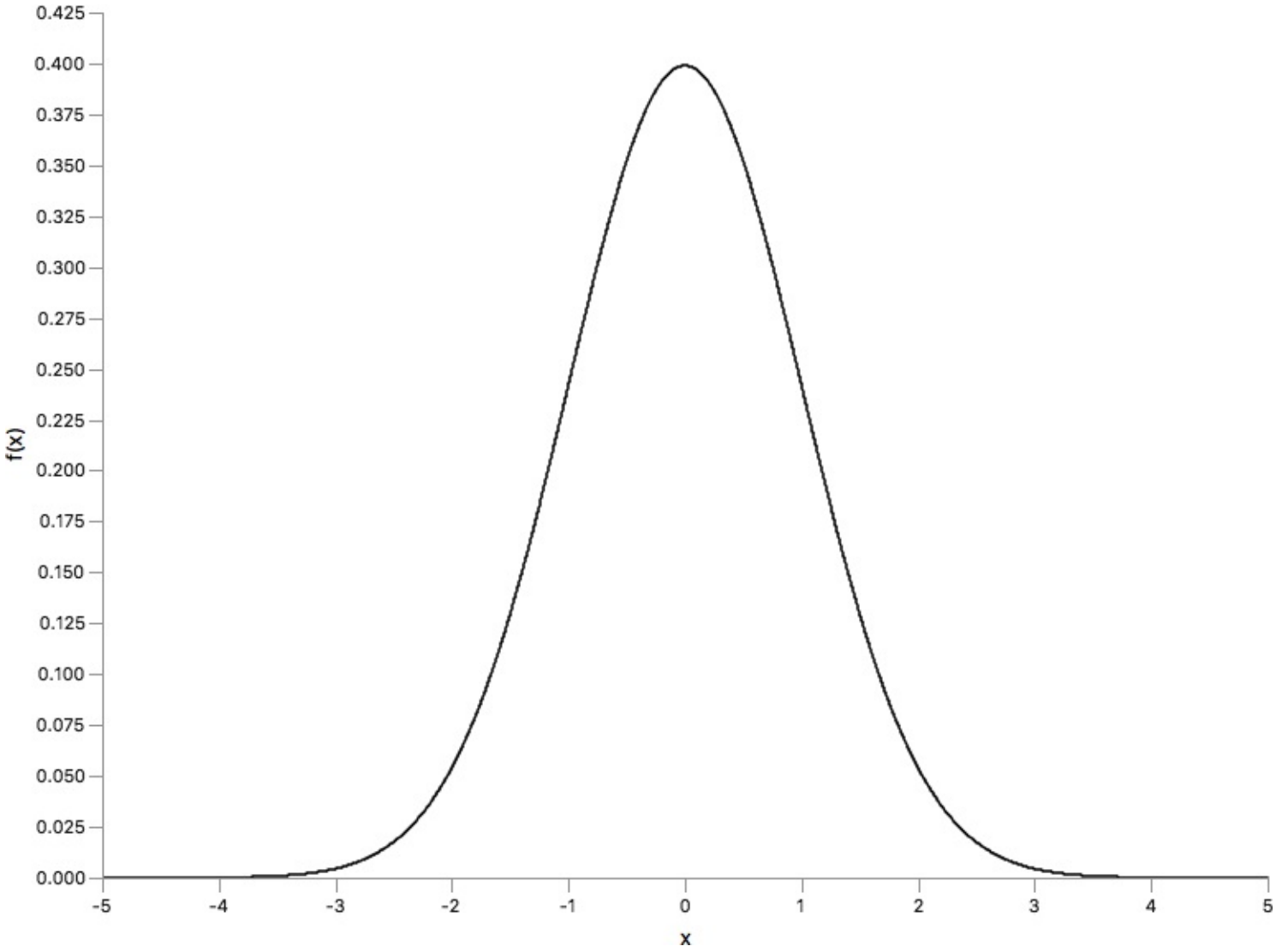

Figure 3-5. Normal PDF with parameters $\mu = 0$ *and* $\sigma = 1$

The probability density is expressed mathematically as follows:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(- \frac{(x-\mu)^2}{2\sigma^2} \right)$$

The cumulative distribution function is shaped like Figure 3-6.

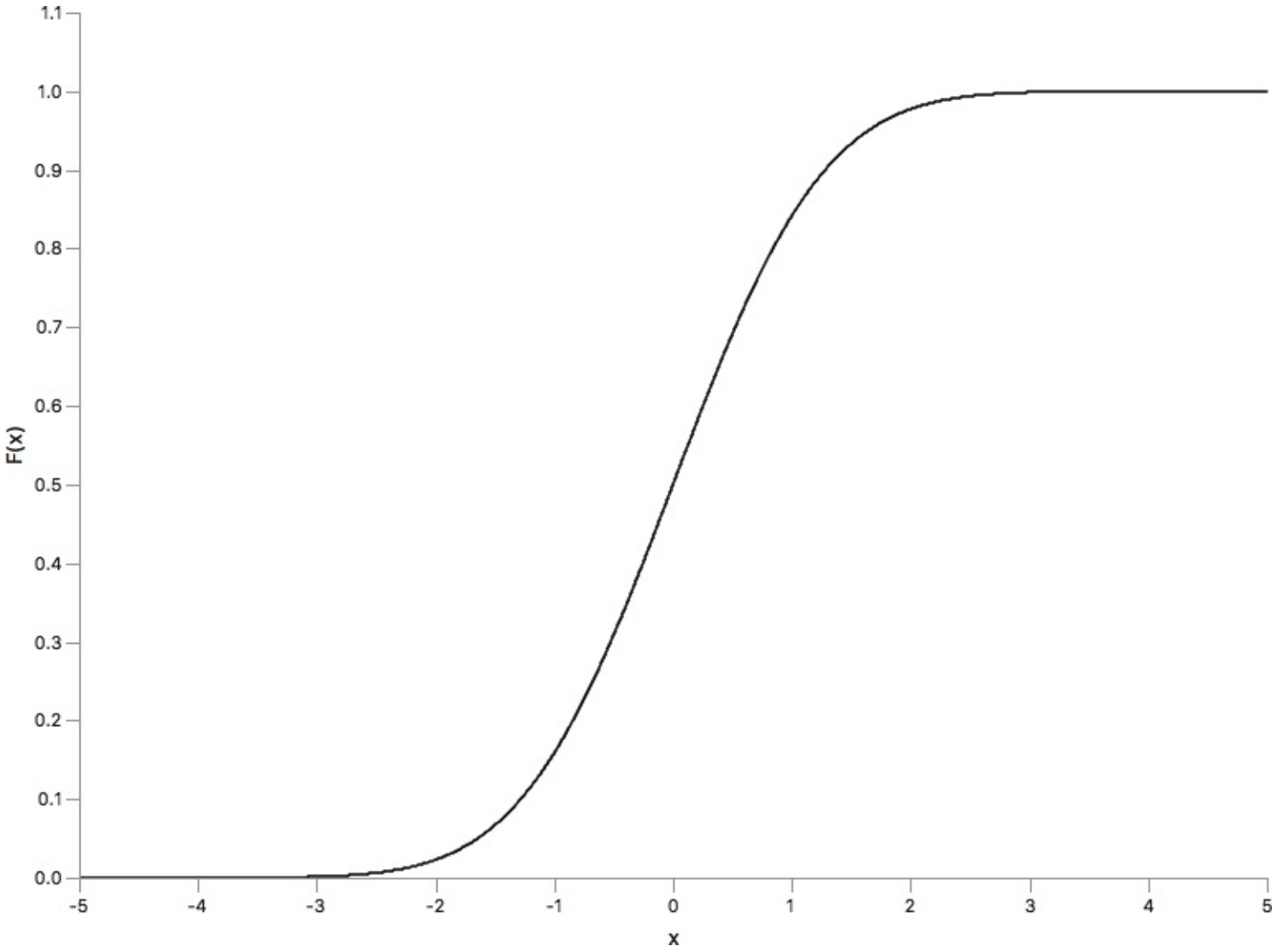

Figure 3-6. Normal CDF with parameters $\mu = 0$ *and* $\sigma = 1$

This is expressed with the error function:

$$F(x) = \frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{x - \mu}{\sigma\sqrt{2}}\right)\right]$$

This is invoked via Java, where the default constructor creates the standard normal distribution with $\mu = 0.0$ and $\sigma = 1.0$. Otherwise, pass the parameters μ and σ to the constructor:

```
/* initialize with default mu=0 and sigma=1 */
NormalDistribution dist = new NormalDistribution();
double mu = dist.getMean(); // 0.0
double sigma = dist.getStandardDeviation(); // 1.0
double mean = dist.getNumericalMean(); // 0.0
double variance = dist.getNumericalVariance(); // 1.0
double lowerBound = dist.getSupportLowerBound(); // -Infinity
double upperBound = dist.getSupportUpperBound(); // Infinity
/* probability at a point x = 0.0 */
double probability = dist.density(0.0);
/* calc cum at x=0.0 */
double cumulativeProbability = dist.cumulativeProbability(0.0);
double sample = dist.sample(); // 1.0120001
```

```
double samples[] = dist.sample(3); // {.0102, -0.009, 0.011}
```

Multivariate normal

The normal distribution can be generalized to higher dimensions as the *multivariate normal* (a.k.a. *multinormal*) distribution. The variate **x** and the mean **μ** are vectors, while the covariance matrix **Σ** contains the variances on the diagonal and the covariances as *i*,*j* pairs. In general, the multivariate normal has a squashed ball shape and is symmetric about the mean. This distribution is perfectly round (or spherical) for unit normals when the covariances are 0 and variances are equivalent. An example of the distribution of random points is shown in Figure 3-7.

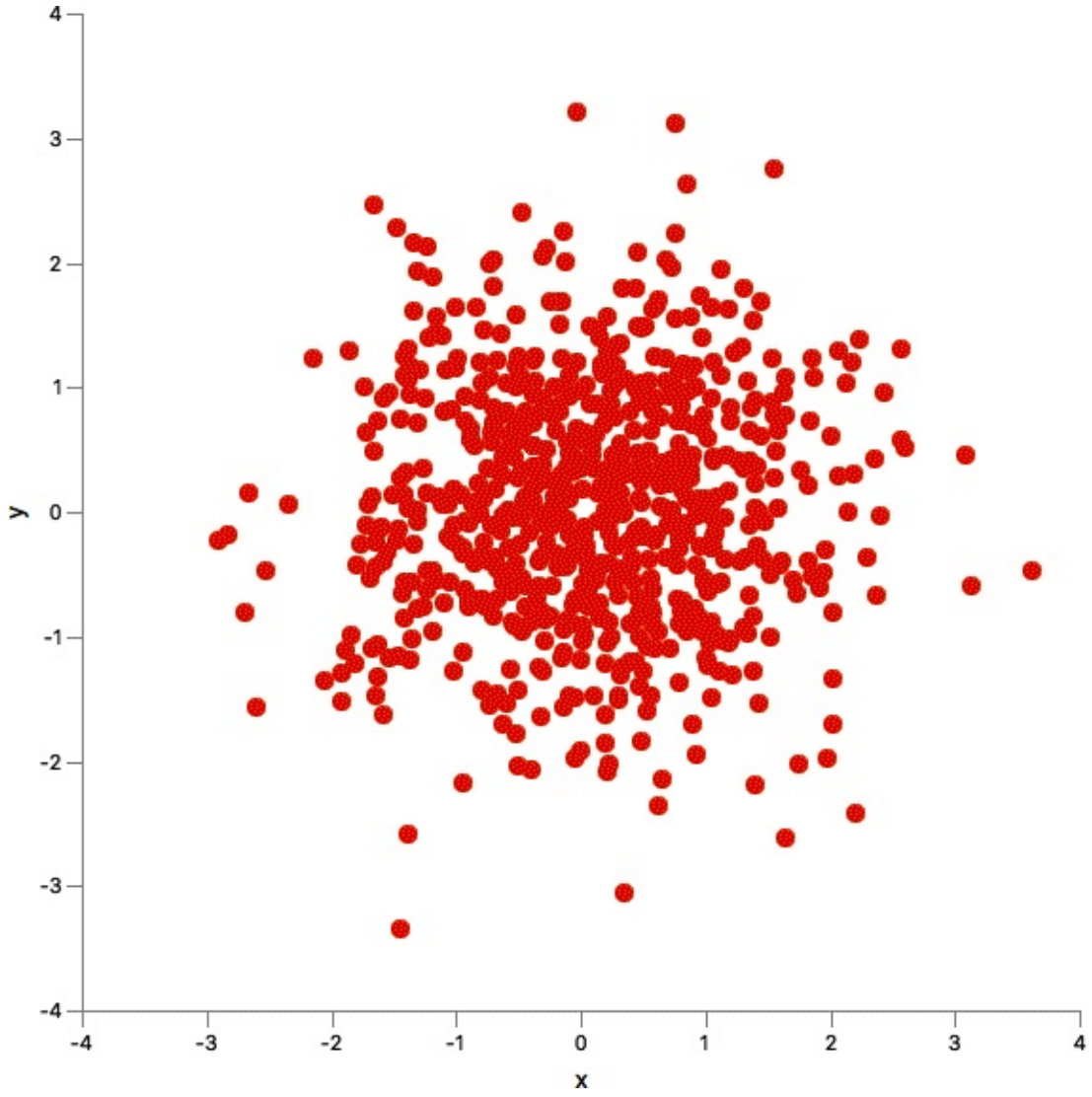

Figure 3-7. Random points generated from 2D multinormal distribution

The probability distribution function of a p dimensional multinormal distribution takes the form

$$f(\mathbf{x}) = \frac{1}{(2\pi)^{p/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right)$$

Note that if the covariance matrix has a determinant equal to zero such that $|\Sigma| = 0$, then $f(\mathbf{x})$ blows up to infinity. Also note that when $|\Sigma| = 0$, it is impossible to calculate the required inverse of the

covariance Σ^{-1}. In this case, the matrix is termed *singular*. Apache Commons Math will throw the following exception if this is the case:

```
org.apache.commons.math3.linear.SingularMatrixException: matrix is singular
```

What causes a covariance matrix to become singular? This is a symptom of co-linearity, in which two (or more) variates of the underlying data are identical or linear combinations of each other. In other words, if we have three dimensions and the covariance matrix is singular, it may mean that the distribution of data could be better described in two or even one dimension.

There is no analytical expression for the CDF. It can be attained via numerical integration. However, Apache Commons Math supports only univariate numerical integration.

The multivariate normal takes means and covariance as arrays of doubles, although you can still use `RealVector`, `RealMatrix`, or `Covariance` instances with their `getData()` methods applied:

```
double[] means = {0.0, 0.0, 0.0};
double[][] covariances = {{1.0, 0.0, 0.0},{0.0, 1.0, 0.0}, {0.0, 0.0, 1.0}};
MultivariateNormalDistribution dist =
  new MultivariateNormalDistribution(means, covariances);

/* probability at point x = {0.0, 0.0, 0.0} */
double probability = dist.density(x); // 0.1
double[] mn = dist.getMeans();
double[] sd = dist.getStandardDeviations();
/* returns a RealMatrix but can be converted to doubles */
double[][] covar = dist.getCovariances().getData();
double[] sample = dist.sample();
double[][] samples = dist.sample(3);
```

Note the special case in which the covariance is a diagonal matrix. This occurs when the variables are completely independent. The determinant of **Σ** is just the product of its diagonal elements $\sigma_{i,i}$. The inverse of a diagonal matrix is yet another diagonal matrix with each term expressed as $1 / \sigma_{i,i}$. The PDF then reduces to the product of univariate normals:

$$f(\mathbf{x}) = \prod_i \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{(x_i - \mu_i)^2}{2\sigma_i^2}\right)$$

As in the case of the unit normal, a unit multivariate normal has a mean vector of 0s and a covariance matrix equal to the identity matrix, a diagonal matrix of 1s.

Log normal

The *log normal distribution* is related to the normal distribution when the variate *x* is distributed logarithmically — that is, *ln(x)* is normally distributed. If we substitute *ln(x)* for *x* in the normal distribution, we get the log normal distribution. There are some subtle differences. Because the logarithm

is defined only for positive x, this distribution has support on the interval $x \in [0, \infty]$, where $x > 0$. The distribution is asymmetric with a peak near the smaller values of x and a long tail stretching, infinitely, to higher values of x, as shown in Figure 3-8.

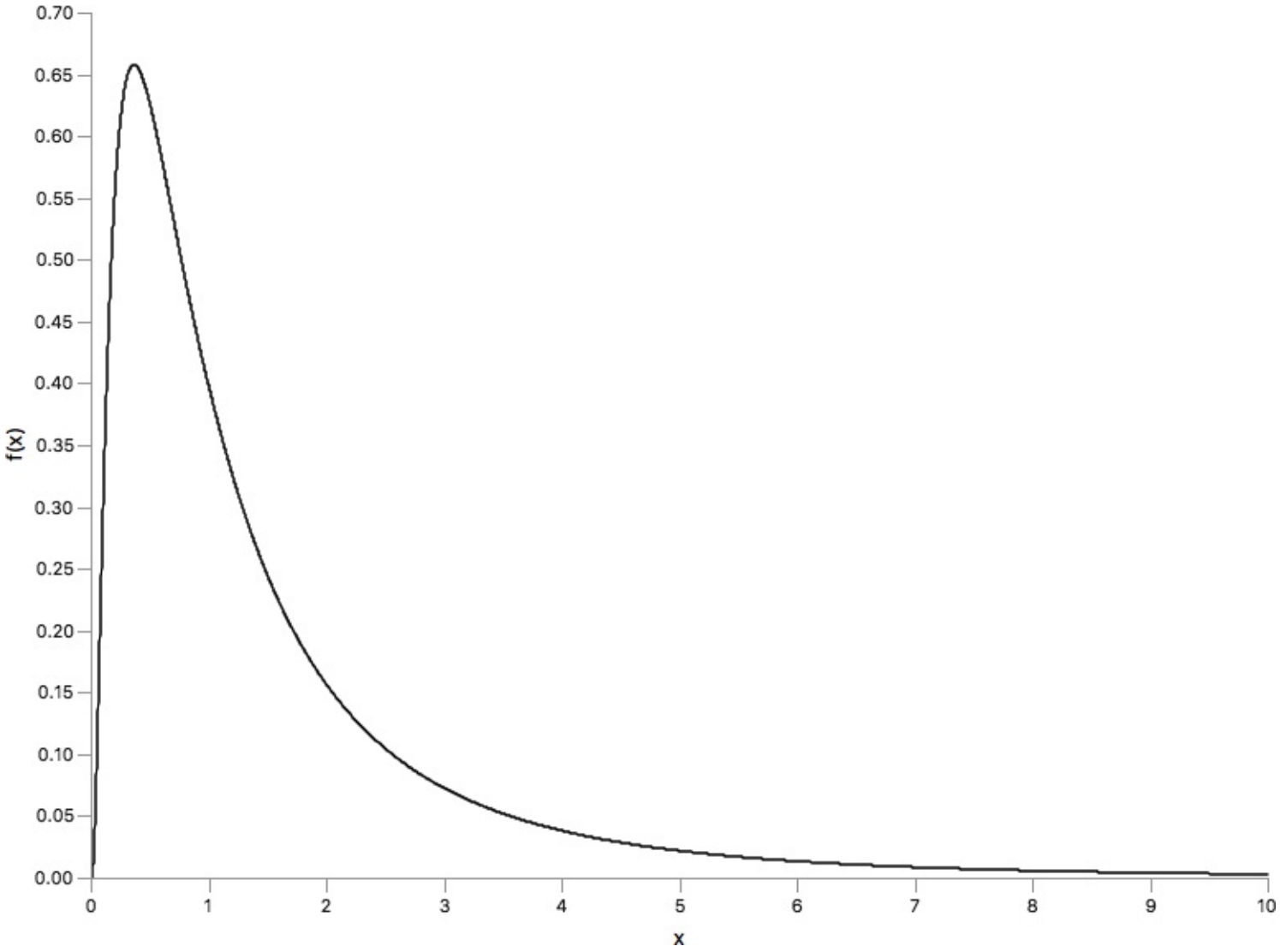

Figure 3-8. Log normal PDF with parameters $m = 0$ *and* $s = 1$

The location (scale) parameter m and the shape parameter s rise to the PDF:

$$f(x) = \frac{1}{\sqrt{2\pi}xs} \exp\left(-\frac{(\ln x - m)^2}{2s^2}\right)$$

Here, m and s are the respective mean and standard deviation of the logarithmically distributed variate X. The CDF looks like Figure 3-9.

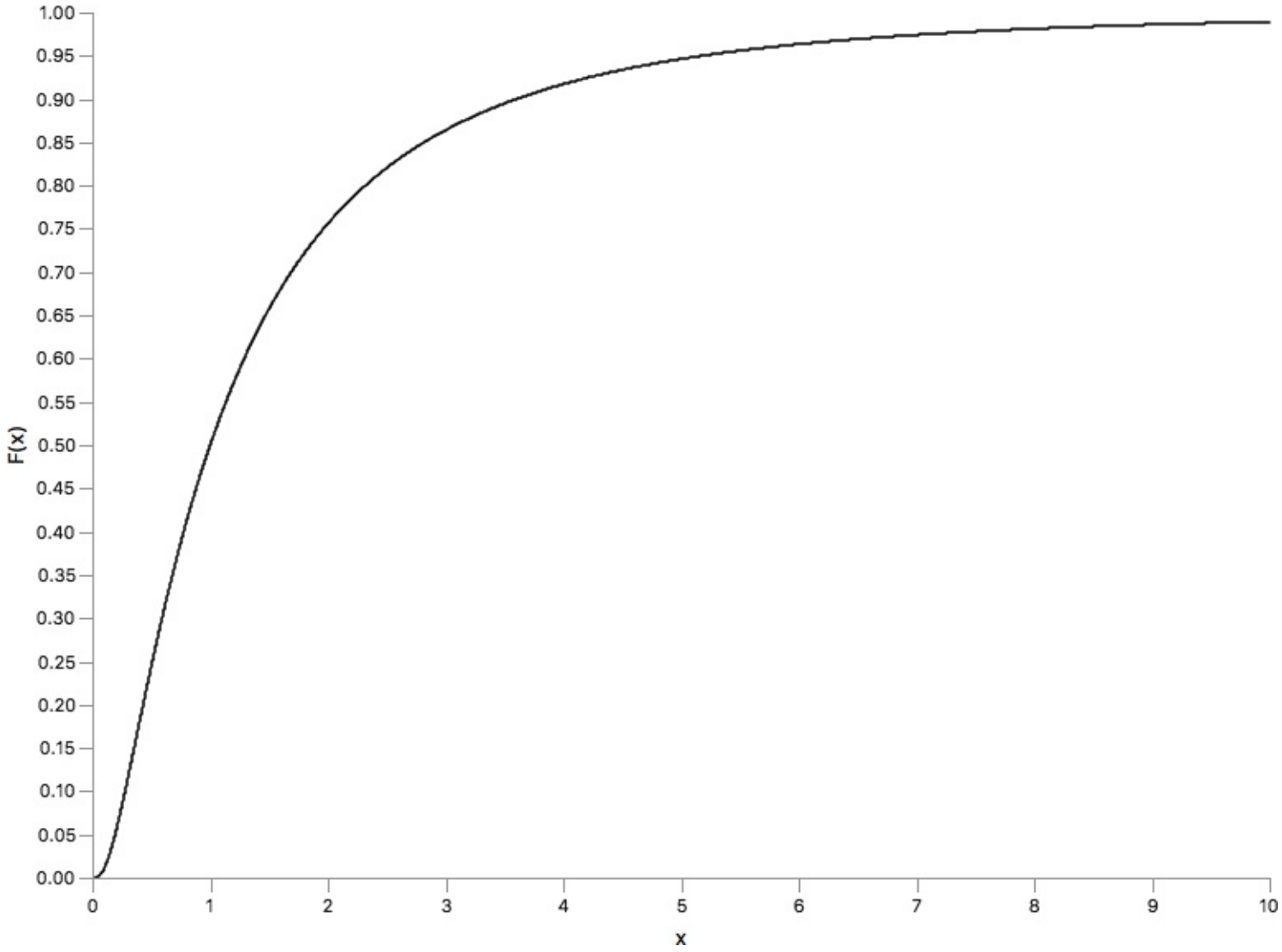

Figure 3-9. Log normal CDF with parameters $m = 0$ *and* $s = 1$

This has the following form:

$$F(x) = \frac{1}{2}\left[1 + \text{erf}\left(\frac{\ln x - m}{s\sqrt{2}}\right)\right]$$

Unlike the normal distribution, the *m* is neither the mean (average value) nor the mode (most likely value or the peak) of the distribution. This is because of the larger number of values stretching off to positive infinity. The mean and variance of *X* are calculated from the following:

$$\mu = \exp\left(m + s^2/2\right)$$

$$\sigma^2 = \left(\exp\left(s^2\right) - 1\right)\exp\left(2m + s^2\right)$$

We can invoke a log normal distribution with this:

```
/* initialize with default m=0 and s=1 */
NormalDistribution dist = new NormalDistribution();
double lowerBound = dist.getSupportLowerBound(); // 0.0
double upperBound = dist.getSupportUpperBound(); // Infinity
double scale = dist.getScale(); // 0.0
double shape = dist.getShape(): // 1.0
double mean = dist.getNumericalMean(); // 1.649
double variance = dist.getNumericalVariance(); // 4.671
double density = dist.density(1.0); // 0.3989
double cumulativeProbability = dist.cumulativeProbability(1.0); // 0.5
double sample = dist.sample(); // 0.428
double[] samples = dist.sample(3); // {0.109, 5.284, 2.032}
```

Where do we see the log normal distribution? The distribution of ages in a human population, and (sometimes) in particle size distribution. Note that the log normal distribution arises from a multiplicative effect of many independent distributions.

Empirical

In some cases you have data, but do not know the distribution that the data came from. You can still approximate a distribution with your data and even calculate probability density, cumulative probability, and random numbers! The first step in working with an empirical distribution is to collect the data into bins of equal size spanning the range of the dataset. The class `EmpiricalDistribution` can input an array of doubles or can load a file locally or from a URL. In those cases, data must be one entry per line:

```
/* get 2500 random numbers from a standard normal distribution */
NormalDistribution nd = new NormalDistribution();
double[] data = nd.sample(2500);

// default constructor assigns bins = 1000
// better to try numPoints / 10
EmpiricalDistribution dist = new EmpiricalDistribution(25);
dist.load(data); // can also load from file or URL !!!
double lowerBound = dist.getSupportLowerBound(); // 0.5
double upperBound = dist.getSupportUpperBound(); // 10.1
double mean = dist.getNumericalMean(); // 5.48
double variance = dist.getNumericalVariance(); // 15.032
double density = dist.density(1.0); // 0.357
double cumulativeProbability = dist.cumulativeProbability(1.0); // 0.153
double sample = dist.sample(); // e.g., 1.396
double[] samples = dist.sample(3); // e.g., [10.098, 0.7934, 9.981]
```

We can plot the data from an empirical distribution as a type of bar chart called a histogram, shown in Figure 3-10.

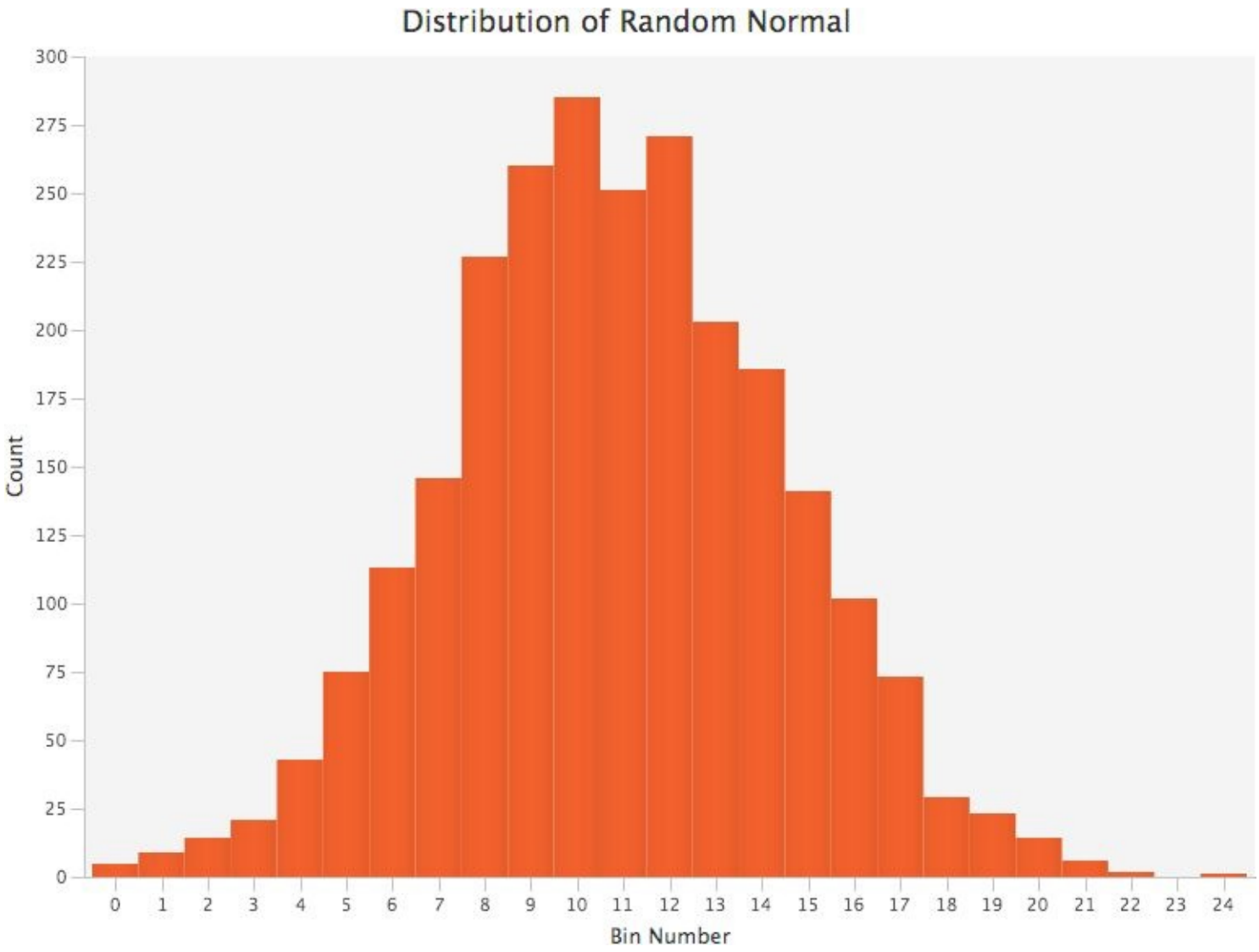

Figure 3-10. Histogram of random normal with parameters $\mu = 0$ *and* $\sigma = 1$

The code for a histogram uses the `BarChart` plot from Chapter 1, except that we add the data directly from the `EmpiricalDistribution` instance that contains a `List` of all the `SummaryStatistics` of each bin:

```
/* for an existing EmpiricalDistribution with loaded data */
List<SummaryStatistics> ss = dist.getBinStats();
int binNum = 0;
for (SummaryStatistics s : ss) {
    /* adding bin counts to the XYChart.Series instance */
    series.getData().add(new Data(Integer.toString(binNum++), s.getN()));
}
// render histogram with JavaFX BarChart
```

Discrete Distributions

There are several discrete random-number distributions. These support only integers as values, which are designated by *k*.

Bernoulli

The *Bernoulli distribution* is the most basic and perhaps most familiar distribution because it is essentially a coin flip. In a "heads we win, tails we lose" situation, the coin has two possible states: tails ($k = 0$) and heads ($k = 1$), where $k = 1$ is designated to have a probability of success equal to p. If the coin is perfect, then $p = 1/2$; it is equally likely to get heads as it is tails. But what if the coin is "unfair," indicating $p \neq 1/2$? The probability mass function (PMF) can then be represented as follows:

$$f(k) = \begin{cases} 1 - p & \text{for } k = 0 \\ p & \text{for } k = 1 \end{cases}$$

The cumulative distribution function is shown here:

$$F(k) = \begin{cases} 0 & \text{for } k < 0 \\ (1 - p) & \text{for } 0 \leq k < 1 \\ 1 & \text{for } k \geq 1 \end{cases}$$

The mean and variance are calculated with the following:

$$\mu = p$$

$$\sigma^2 = p(1 - p)$$

Note that the Bernoulli distribution is related to the binomial distribution, where the number of trials equals $n = 1$. The Bernoulli distribution is implemented with the class `BinomialDistribution(1, p)` setting $n = 1$:

```
BinomialDistribution dist = new BinomialDistribution(1, 0.5);
int lowerBound = dist.getSupportLowerBound(); // 0
int upperBound = dist.getSupportUpperBound(); // 1
int numTrials = dist.getNumberOfTrials(); // 1
double probSuccess = dist.getProbabilityOfSuccess(); // 0.5
double mean = dist.getNumericalMean(); // 0.5
double variance = dist.getNumericalVariance(); // 0.25
// k = 1
double probability = dist.probability(1); // 0.5
```

```
double cumulativeProbability = dist.cumulativeProbability(1); // 1.0
int sample = dist.sample(); // e.g., 1
int[] samples = dist.sample(3); // e.g., [1, 0, 1]
```

Binomial

If we perform multiple Bernoulli trials, we arrive at the binomial distribution. For *n* Bernoulli trials, each with probability of success *p*, the distribution of successes *k* has the form in Figure 3-11.

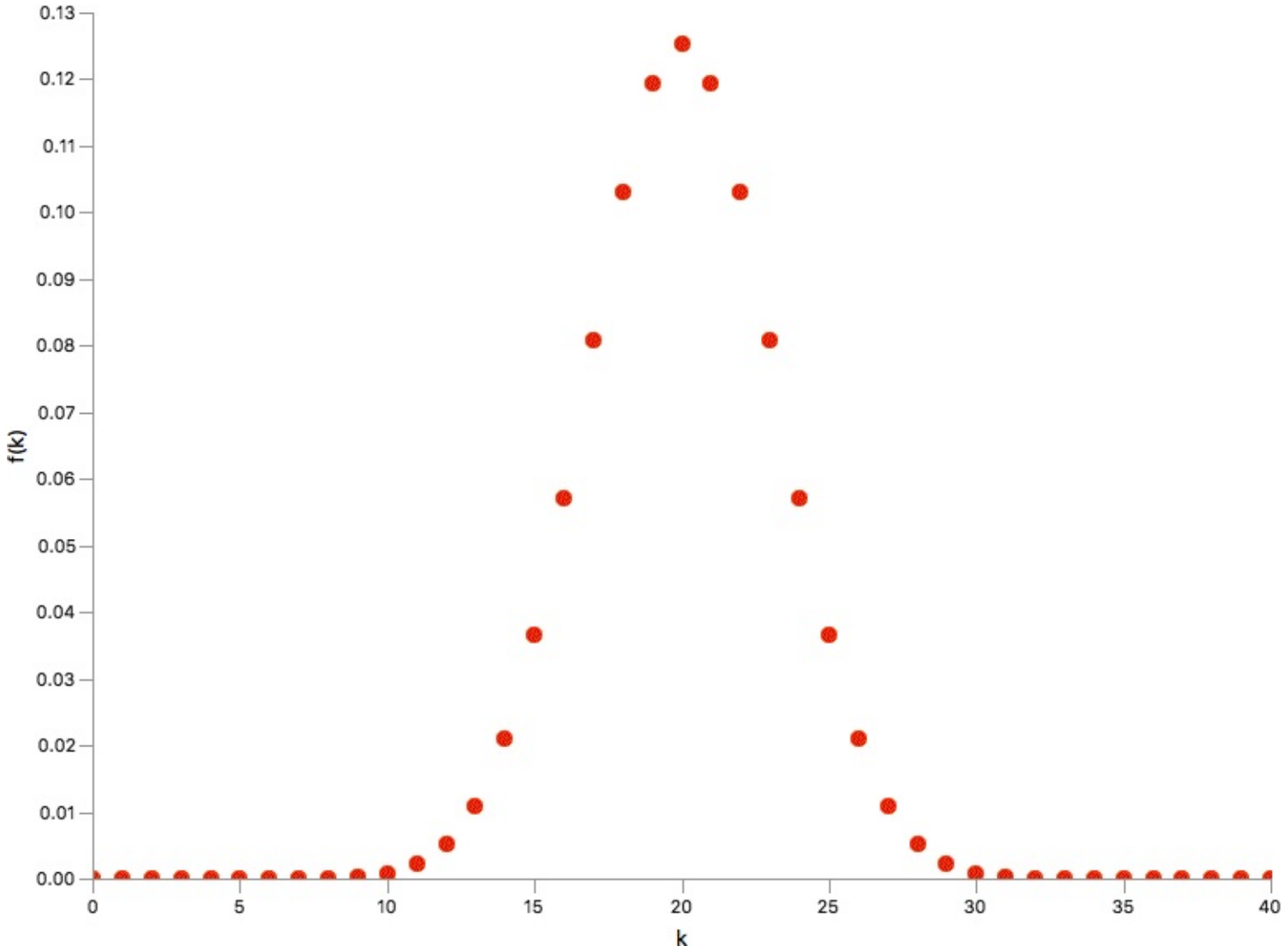

Figure 3-11. Binomial PMF with parameters $n = 40$ *and* $p = 0.5$

The probability mass function is expressed with the following:

$$f(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

The CDF looks like Figure 3-12.

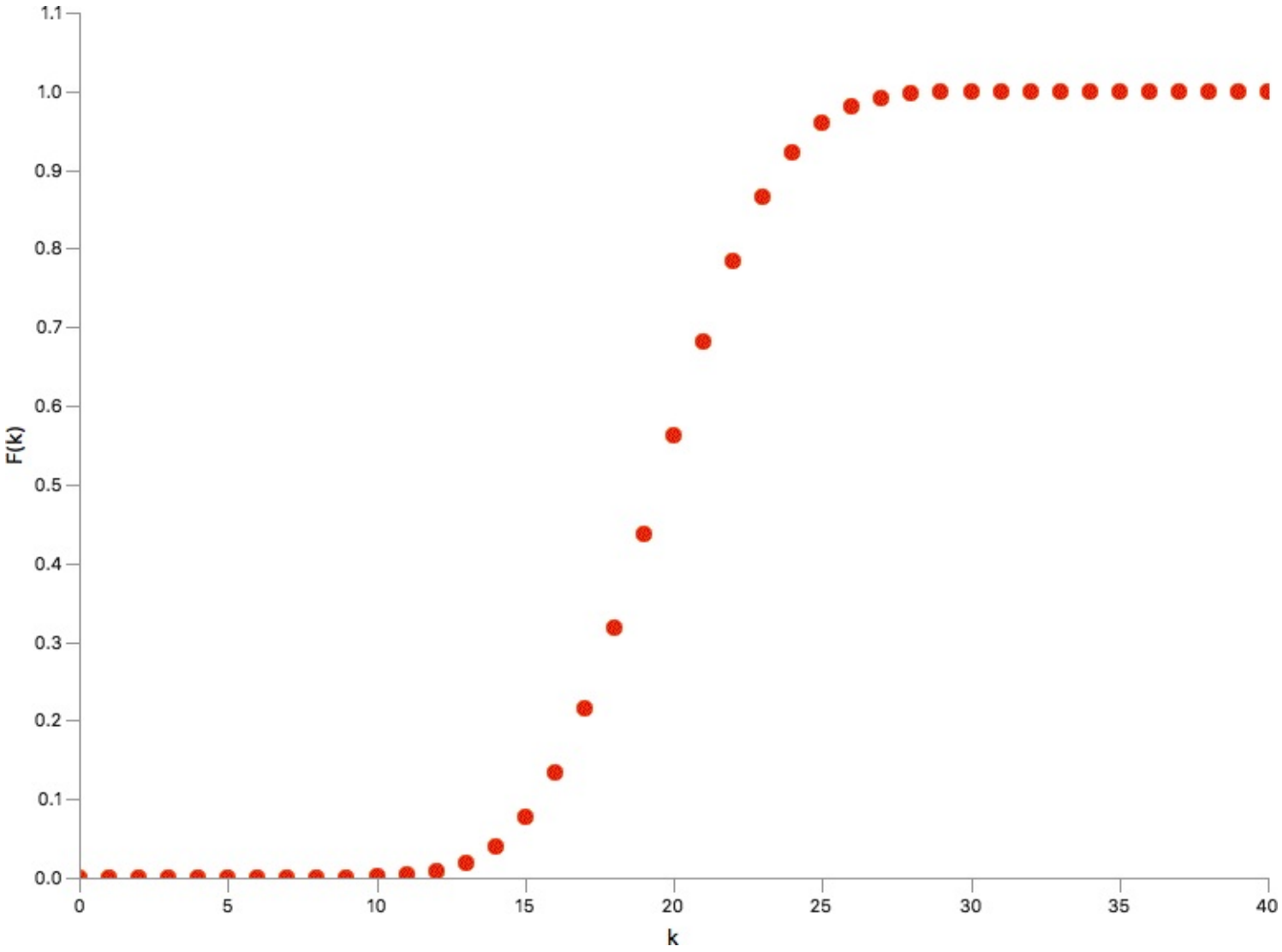

Figure 3-12. Binomial CDF with parameters $n = 40$ *and* $p = 0.5$

The CDF takes the form

$$F(k) = I_{1-p}(n - k, 1 + k)$$

I_{1-p} is the regularized incomplete beta function. The mean and variance are computed via the following:

$$\mu = np$$

$$\sigma^2 = np(1 - p)$$

In Java, `BinomialDistribution` has two required arguments in the constructor: *n*, the number of trials; and *p*, the probability of success for one trial:

```
BinomialDistribution dist = new BinomialDistribution(10, 0.5);
int lowerBound = dist.getSupportLowerBound(); // 0
int upperBound = dist.getSupportUpperBound(); // 10
```

```
int numTrials = dist.getNumberOfTrials(); // 10
double probSuccess = dist.getProbabilityOfSuccess(); // 0.5
double mean = dist.getNumericalMean(); // 5.0
double variance = dist.getNumericalVariance(); // 2.5
// k = 1
double probability = dist.probability(1); // 0.00977
double cumulativeProbability = dist.cumulativeProbability(1); // 0.0107
int sample = dist.sample(); // e.g., 9
int[] samples = dist.sample(3); // e.g., [4, 5, 4]
```

Poisson

The *Poisson distribution* is often used to describe discrete, independent events that occur rarely. The number of events are the integers $k \geq 0$ that occur over some interval with a constant rate $\lambda > 0$ gives rise to the PMF in Figure 3-13.

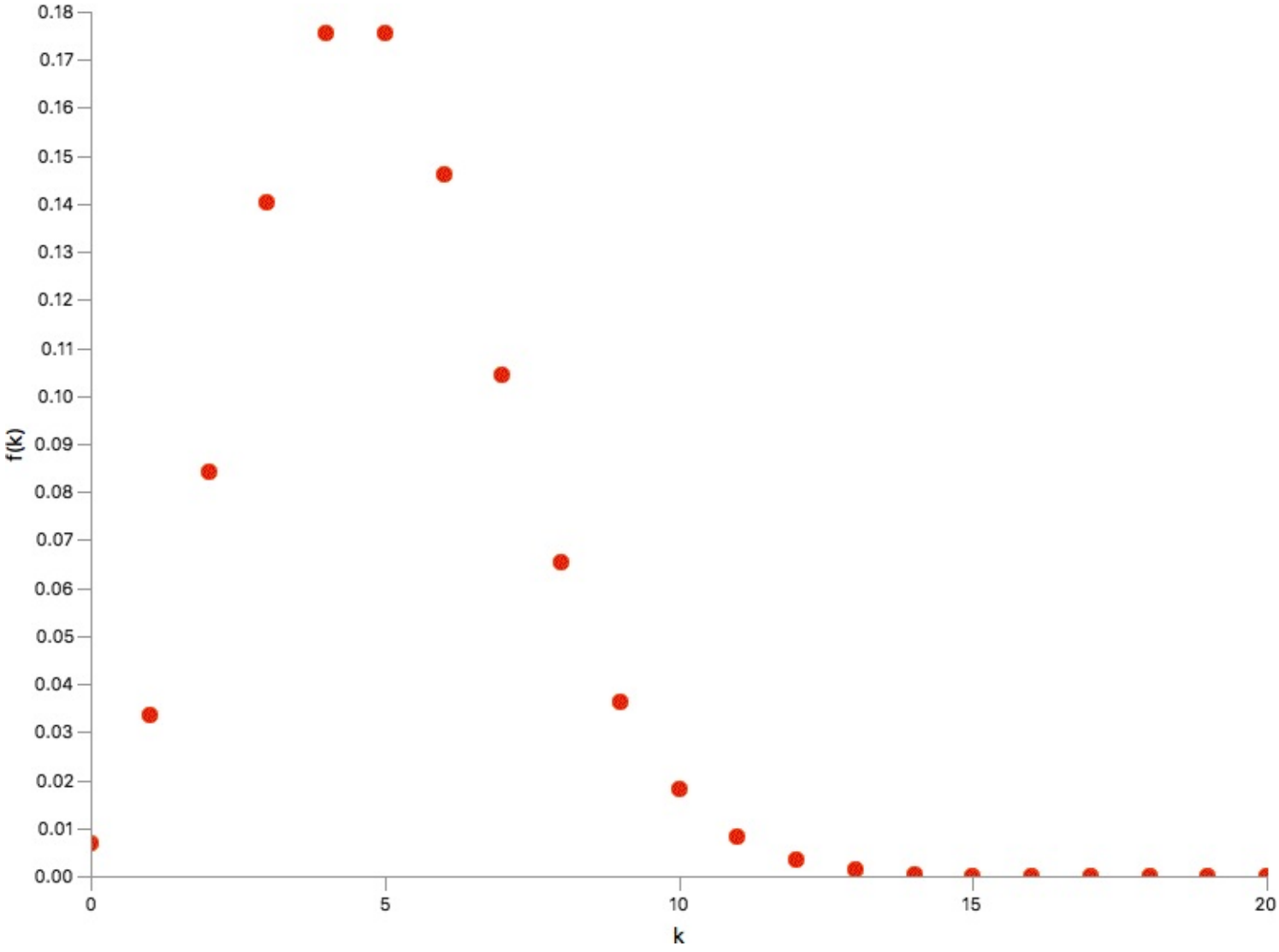

Figure 3-13. Poisson PMF with parameters $\lambda = 5$

The form of the PMF is

$$f(k) = \frac{\lambda^k \exp(-\lambda)}{k!}$$

Figure 3-14 shows the CDF.

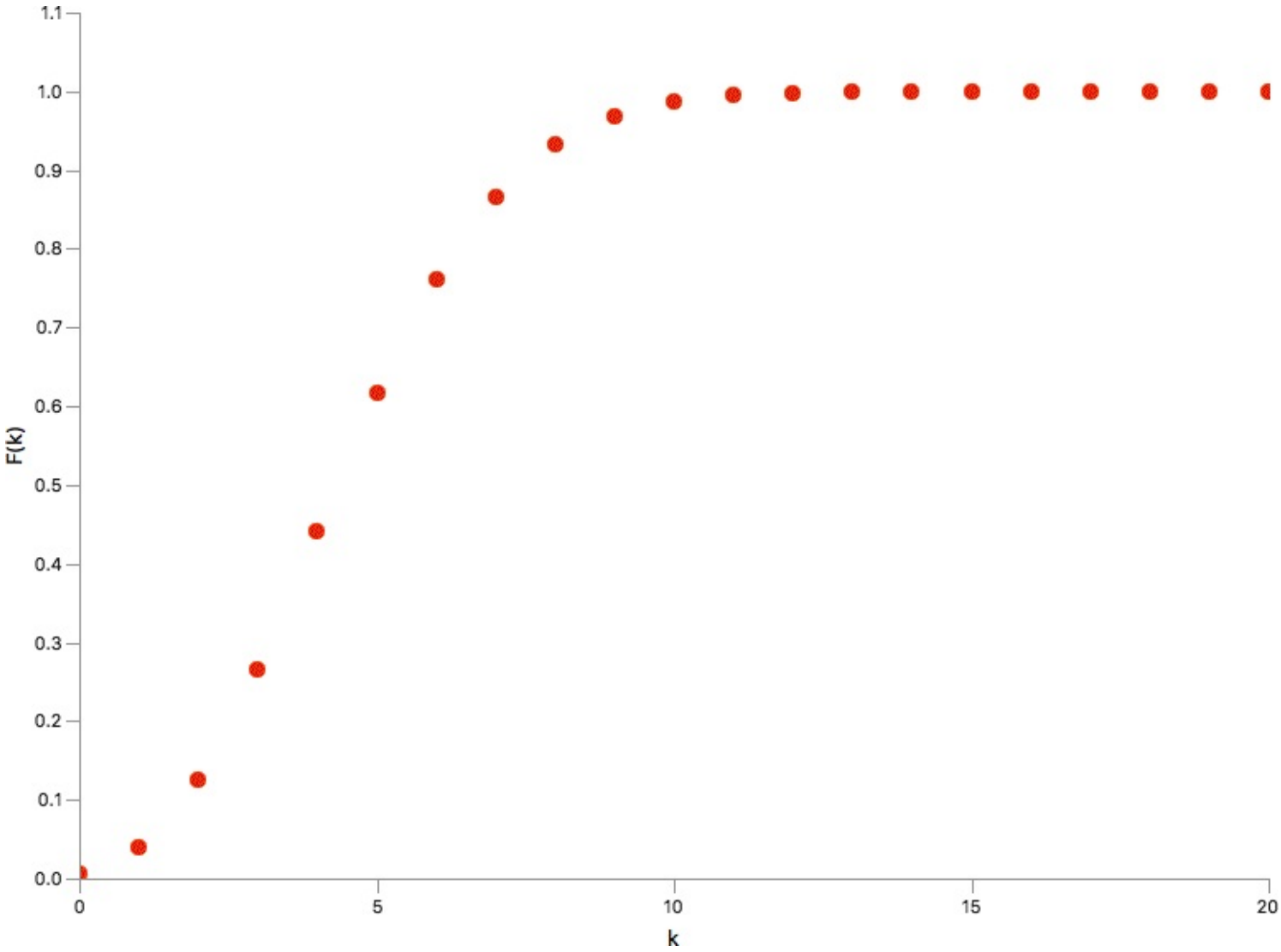

Figure 3-14. Poisson CDF with parameters $\lambda = 5$

The CDF is expressed via

$$F(k) = \frac{\Gamma(\lfloor k+1 \rfloor, \lambda)}{\lfloor k \rfloor !}$$

The mean and variance are both equivalent to the rate parameter λ as

$$\mu = \lambda$$

$$\sigma^2 = \lambda$$

The Poisson is implemented with the parameter λ in the constructor and has an upper bound at `Integer.MAX` $k = 2^{32} - 1 = 2147483647$:

```
PoissonDistribution dist = new PoissonDistribution(3.0);
int lowerBound = dist.getSupportLowerBound(); // 0
int upperBound = dist.getSupportUpperBound(); // 2147483647
double mean = dist.getNumericalMean(); // 3.0
double variance = dist.getNumericalVariance(); // 3.0
// k = 1
double probability = dist.probability(1); // 0.1494
double cumulativeProbability = dist.cumulativeProbability(1); // 0.1991
int sample = dist.sample(); // e.g., 1
int[] samples = dist.sample(3); // e.g., [2, 4, 1]
```

Characterizing Datasets

Once we have a dataset, the first thing we should do is understand the character of the data. We should know the numerical limits, whether any outliers exist, and whether the data has a shape like one of the known distribution functions. Even if we have no idea what the underlying distribution is, we can still check whether two separate datasets come from the same (unknown) distribution. We can also check how related (or unrelated) each pair of variates is via covariance/correlation. If our variate *x* comes with a response *y*, we can check a linear regression to see whether the most basic of relationships exists between *x* and *y*. Most of the classes in this section are best for small, static datasets that can fit entirely into memory, because most of the methods in these classes rely on stored data. In the following section, we deal with data that is so large (or inconvenient) that it cannot fit in memory.

Calculating Moments

In the previous section where we discussed statistical moments, formulas were presented for calculating moments and their various statistical outcomes when we know the probability distribution function *f(x)*. When dealing with real data, we usually do not know the form of *f(x)* and so we must estimate the moments numerically. The moment calculations have another critical feature: robustness. Estimating statistical quantities can result in numerical error as extreme values are encountered. Using the method of updating moments, we can avoid numerical imprecision.

Sample moments

In the case of real data, where the true statistical distribution function may not be known, we can estimate the central moments:

$$m_k = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^k$$

Here, the estimated mean $m_1 = \overline{x}$ is calculated as follows:

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n}x_i$$

Updating moments

Without the factor 1/n for the estimate of the central moment, the form is as follows:

$$M_k = \sum_{i=1}^{n}(x_i - \overline{x})^k$$

In this particular form, the unnormalized moments can be split into parts via straightforward calculations. This gives us a great advantage that the unnormalized moments can be calculated in parts, perhaps in different processes or even on different machines entirely, and we can glue back together the pieces later. Another advantage is that this formulation is less sensitive to extreme values. The combined unnormalized central moment for the two chunks is shown here:

$$M_k = M_{k,1} + M_{k,2} + \sum_{j=1}^{k-2} \binom{j}{k} \left[\left(-\frac{n_2}{n} \right)^j M_{k-j,1} + \left(\frac{n_1}{n} \right)^j M_{k-j,2} \right] \delta_{2,1}^j + \left(\frac{n_1 n_2}{n} \delta_{2,1} \right)^k \left[\frac{1}{n_2^{k-1}} - \left(\frac{-1}{n_1} \right)^{k-1} \right]$$

$\delta_{2,1} = \bar{x}_2 - \bar{x}_1$ is the difference between the means of the two data chunks. Of course, you will also need a method for merging the means, because this formula is applicable only for $k > 1$. Given any two data chunks with known means and counts, the total count is $n = n_1 + n_2$, and the mean is robustly calculated as follows:

$$\bar{x} = \bar{x}_1 + n_2 \frac{\delta_{2,1}}{n}$$

If one of the data chunks has only one point x, then the formulation for combining the unnormalized central moments is simplified:

$$M_k = M_{k,1} + \sum_{j=1}^{k-2} \binom{j}{k} M_{k-j,1} \left(\frac{-\delta}{n} \right)^j + \left(\frac{n-1}{n} \delta \right)^k \left[1 - \left(\frac{-1}{n-1} \right)^{k-1} \right]$$

Here, $\delta = x - \bar{x}_1$ is the difference between the added value x and the mean value of the existing data chunk.

In the next section, we will see how these formulas are used to robustly calculate important properties to statistics. They become essential in distributed computing applications when we wish to break statistical calculations into many parts. Another useful application is storeless calculations in which, instead of performing calculations on whole arrays of data, we keep track of moments as we go, and update them incrementally.

Descriptive Statistics

We instantiate `DescriptiveStatistics` with no argument to add values later, or start it out with an array of doubles (and then can still add values later). Although you can use `StatUtils` static methods, this is not Java-like, and although there is nothing wrong with it, it's probably wiser to use `DescriptiveStatistics` instead. Some of the formulas in this section are not stable, and a more robust method is described in the next section. Indeed, some of those methods are used in the descriptive stats methods as well. In Table 3-1, we display the data from Anscombe's quartet for further analysis in this chapter.

Table 3-1. Anscombe's quartet data

x1	y1	x2	y2	x3	y3	x4	y4
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

We can then create `DescriptiveStatistics` classes with these datasets:

```
/* stats for Anscombe's y1 */
DescriptiveStatistics descriptiveStatistics = new DescriptiveStatistics();
descriptiveStatistics.addValue(8.04);
descriptiveStatistics.addValue(6.95);
//keep adding y1 values
```

However, you may already have all the data that you need, or maybe it's just an initial set that you will add to later. You can always add more values with `ds.addValue(double value)` if needed. At this point, you can display a report of stats with either a call to the method, or by printing the class directly:

```
System.out.println(descriptiveStatistics);
```

This produces the following result:

```
DescriptiveStatistics:
n: 11
min: 4.26
max: 10.84
mean: 7.500909090909091
std dev: 2.031568135925815
median: 7.58
skewness: -0.06503554811157437
kurtosis: -0.5348977343727395
```

All of these quantities (and more) are available via their specific getters, as explained next.

Count

The simplest statistic is the count of the number of points in the dataset:

```
long count = descriptiveStatistics.getN();
```

Sum

We can also access the sum of all values:

```
double sum = descriptiveStatistics.getSum();
```

Min

To retrieve the minimum value dataset, we use this:

```
double min = descriptiveStatistics.getMin();
```

Max

To retrieve the maximum value in the dataset, we use this:

```
double max = descriptiveStatistics.getMax();
```

Mean

The average value of the sample or mean is calculated directly:

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

However, this calculation is sensitive to extreme values, and given $\delta = x - \overline{x}$, the mean can be updated for each added value *x*:

$$\overline{x} = \overline{x}_1 + \frac{x - \overline{x}_1}{n}$$

Commons Math uses the update formula for mean calculation when the `getMean()` method is called:

```
double mean = descriptiveStatistics.getMean();
```

Median

The middle value of a sorted (ascending) dataset is the *median*. The advantage is that it minimizes the

problem of extreme values. While there is no direct calculation of the median in Apache Commons Math, it is easy to calculate by taking the average of the two middle members if the array length is even; and otherwise, just return the middle member of the array:

```
// sort the stored values
double[] sorted = descriptiveStatistics.getSortedValues();
int n = sorted.length;
double median = (n % 2 == 0) ? (sorted[n/2-1]+sorted[n/2])/2.0 : sorted[n/2];
```

Mode

The mode is the most likely value. The concept of *mode* does not make sense if the values are doubles, because there is probably only one of each. Obviously, there will be exceptions (e.g., when many zeros occur) or if the dataset is large and the numerical precision is small (e.g., two decimal places). The mode has two use cases then: if the variate being considered is discrete (integer), then the mode can be useful, as in dataset four of Anscombe's quartet. Otherwise, if you have created bins from empirical distribution, the mode is the max bin. However, you should consider the possibility that your data is noisy and the bin counts may erroneously identify an outlier as a mode.The `StatUtils` class contains several static methods useful for statistics. Here we utilize its mode method:

```
// if there is only one max, it's stored in mode[0]
// if there is more than one value that has equally high counts
// then values are stored in mode in ascending order
double[] mode = StatUtils.mode(x4);
//mode[0] = 8.0
double[] test = {1.0, 2.0, 2.0, 3.0, 3.0, 4.0}
//mode[0] = 2.0
//mode[1] = 3.0
```

Variance

The *variance* is a measure of how much the data is spread out and is always a positive, real number greater than or equal to zero. If all values of x are equal, the variance will be zero. Conversely, a larger spread of numbers will correspond to a larger variance. The variance of a known population of data points is equivalent to the second central moment about the mean and is expressed as follows:

$$s^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

However, most of the time we do not have all of the data — we are only sampling from a larger (possibly unknown) dataset and so a correction for this bias is needed:

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

This form is known as the *sample variance* and is most often the variance we will use. You may note that the sample variance can be expressed in terms of the second-order, unnormalized moment:

$$s^2 = \frac{1}{n-1}M_2$$

As with the mean calculation, the Commons Math variance is calculated using the update formula for the second unnormalized moment for a new data point x, an existing mean $\bar{x}_1$ and the newly updated mean $\bar{x}$:

$$M_2 = M_{2,1} + (x - \bar{x}_1)\delta$$

Here, $\delta = x - \bar{x}_1$.

Most of the time when the variance is required, we are asking for the bias-corrected, sample variance because our data is usually a sample from some larger, possibly unknown, set of data:

```
double variance = descriptiveStatistics.getVariance()
```

However, it is also straightforward to retrieve the population variance if it is required:

```
double populationVariance = descriptiveStatistics.getPopulationVariance();
```

Standard deviation

The variance is difficult to visualize because it is on the order of x^2 and usually a large number compared to the mean. By taking the square root of the variance, we define this as the standard deviation s. This has the advantage of being in the same units of the variates and the mean. It is therefore helpful to use things like $\mu +- \sigma$, which can indicate how much the data deviates from the mean. The standard deviation can be explicitly calculated with this:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

However, in practice, we use the update formulas to calculate the sample variance, returning the standard deviation as the square root of the sample variance when required:

```
double standardDeviation = descriptiveStatistics.getStandardDeviation();
```

Should you require the population standard deviation, you can calculate this directly by taking the square root of the population variance.

Error on the mean

While it is often assumed that the standard deviation is the error on the mean, this is not true. The standard deviation describes how the data is spread around the average. To calculate the accuracy of the mean itself, s_x, we use the standard deviation:

$$s_x = \frac{s}{\sqrt{n}}$$

And we use some simple Java:

```
double meanErr = descriptiveStatistics.getStandardDeviation() /
                        Math.sqrt(descriptiveStatistics.getN());
```

Skewness

The *skewness* measures how asymmetrically distributed the data is and can be either a positive or negative real number. A positive skew signifies that most of the values leans toward the origin ($x = 0$), while a negative skew implies the values are distributed away (to right). A skewness of 0 indicates that the data is perfectly distributed on either side of the data distribution'’s peak value. The skewness can be calculated explicitly as follows:

$$\omega = \frac{1}{(n-1)(n-2)} \sum_{i=1}^{n} \left(\frac{x_i - \bar{x}}{s} \right)^3$$

However, a more robust calculation of the skewness is achieved by updating the third central moment:

$$M_3 = M_{3,1} - 3M_{2,1}\frac{\delta}{n} + (n-1)(n-2)\frac{\delta^3}{n^2}$$

Then you calculate the skewness when it is called for:

$$\omega = \frac{1}{(n-1)(n-2)} \frac{M_3}{s^3}$$

The Commons Math implementation iterates over the stored dataset, incrementally updating M3 and then performing the bias correction, and returns the skewness:

```
double skewness = descriptiveStatistics.getSkewness();
```

Kurtosis

The kurtosis is a measure of how "tailed" a distribution of data is. The sample kurtosis estimate is related to the fourth central moment about the mean and is calculated as shown here:

$$\kappa = \frac{(n+1)}{(n-1)(n-2)(n-3)} \sum_{i=1}^{n} \left(\frac{x_i - \overline{x}}{s} \right)^4$$

This can be simplified as follows:

$$\kappa = \frac{(n+1)}{(n-1)(n-2)(n-3)} \frac{M_4}{s^4}$$

A kurtosis at or near 0 signifies that the data distribution is extremely narrow. As the kurtosis increases, extreme values coincide with long tails. However, we often want to express the kurtosis as related to the normal distribution (which has a kurtosis = 3). We can then subtract this part and call the new quantity the *excess kurtosis*, although in practice most people just refer to the excess kurtosis as the kurtosis. In this definition, $\kappa = 0$ implies that the data has the same peak and tail shape as a normal distribution. A kurtosis higher than 3 is called *leptokurtic* and has wider tails than a normal distribution. When κ is less than 3, the distribution is said to be *platykurtic* and the distribution has few values in the tail (less than the normal distribution). The excess kurtosis calculation is as follows:

$$\kappa = \frac{(n+1)}{(n-1)(n-2)(n-3)} \frac{M_4}{s^4} - \frac{3(n-1)^2}{(n-2)(n-3)}$$

As in the case for the variance and skewness, the kurtosis is calculated by updating the fourth unnormalized central moment. For each point added to the calculation, M_4 can be updated with the

following equation as long as the number of points n >= 4. At any point, the skew can be computed from the current value of M_4:

$$M_4 = M_{4,1} - 4\,M_{3,1}\frac{\delta}{n} + 6\,M_{2,1}\left(\frac{\delta}{n}\right)^2 + (n-1)(n^2-3n+3)\frac{\delta^4}{n^3}$$

The default calculation returned by `getKurtosis` is the excess kurtosis definition. This can be checked by implementing the class `org.apache.commons.math3.stat.descriptive.moment.Kurtosis`, which is called by `getKurtosis()`:

```
double kurtosis = descriptiveStatistics.getKurtosis();
```

Multivariate Statistics

So far, we have addressed the situation where we are concerned with one variate at a time. The class `DescriptiveStatistics` takes only one-dimensional data. However, we usually have several dimensions, and it is not uncommon to have hundreds of dimensions. There are two options: the first is to use the `MultivariateStatisticalSummary` class, which is described in the next section. If you can live without skewness and kurtosis, that is your best option. If you do require the full set of statistical quantities, your best bet is to implement a `Collection` instance of `DescriptiveStatistics` objects. First consider what you want to keep track of. For example, in the case of Anscombe's quartet, we can collect univariate statistics with the following:

```
DescriptiveStatistics descriptiveStatisticsX1 = new DescriptiveStatistics(x1);
DescriptiveStatistics descriptiveStatisticsX2 = new DescriptiveStatistics(x2);
...
List<DescriptiveStatistics> dsList = new ArrayList<>();
dsList.add(descriptiveStatisticsX1);
dsList.add(descriptiveStatisticsX2);
...
```

You can then iterate through the `List`, calling statistical quantities or even the raw data:

```
for(DescriptiveStatistics ds : dsList) {

    double[] data = ds.getValues();
    // do something with data or

    double kurtosis = ds.getKurtosis();
    // do something with kurtosis
}
```

If the dataset is more complex and you know you will need to call specific columns of data in your ensuing analysis, use `Map` instead:

```
DescriptiveStatistics descriptiveStatisticsX1 = new DescriptiveStatistics(x1);
DescriptiveStatistics descriptiveStatisticsY1 = new DescriptiveStatistics(y1);
DescriptiveStatistics descriptiveStatisticsX2 = new DescriptiveStatistics(x2);

Map<String, DescriptiveStatistics> dsMap = new HashMap<>();
dsMap.put("x1", descriptiveStatisticsX1);
dsMap.put("y1", descriptiveStatisticsY1);
dsMap.put("x2", descriptiveStatisticsX2);
```

Of course, now it is trivial to call a specific quantity or dataset by its key:

```
double x1Skewness = dsMap.get("x1").getSkewness();
double[] x1Values = dsMap.get("x1").getValues();
```

This will become cumbersome for a large number of dimensions, but you can simplify this process if the data was already stored in multidimensional arrays (or matrices) where you loop over the column index. Also, you may have already stored your data in the `List` or `Map` of the data container classes, so automating the process of building a multivariate `Collection` of `DescriptiveStatistics` objects will be straightforward. This is particularly efficient if you already have a data dictionary (a list of variable names and their properties) from which to iterate over. If you have high-dimensional numeric data of one type and it already exists in a matrix of double array form, it might be easier to use the

`MultivariateSummaryStatistics` class in the next section.

Covariance and Correlation

The covariance and correlation matrices are symmetric, square $m \times m$ matrices with dimension m equal to the number columns in the original dataset.

Covariance

The covariance is the two-dimensional equivalent of the variance. It measures how two variates, in combination, differ from their means. It is calculated as shown here:

$$\sigma^2_{i,j} = \frac{1}{n-1}\sum_{k=1}^{n}(x_{k,i} - \overline{x}_i)(x_{k,j} - \overline{x}_j)$$

Just as in the case of the one-dimensional, statistical sample moments, we note that the quantity

$$C_2 = \sum_{k=1}^{n}(x_{k,i} - \overline{x}_i)(x_{k,j} - \overline{x}_j)$$

can be expressed as an incremental update to the co-moment of the pair of variables x_i and x_j, given a known means and counts for the existing dimensions:

$$C_2 = C_{2,1} + \frac{n_1 n_2}{n}(x_i - \overline{x}_i)(x_j - \overline{x}_j)$$

Then the covariance can be calculated at any point:

$$\sigma^2_{i,j} = \frac{1}{n-1}C_2$$

The code to calculate the covariance is shown here:

```
Covariance cov = new Covariance();

/* examples using Anscombe's Quartet data */
double cov1 = cov.covariance(x1, y1); // 5.501
double cov2 = cov.covariance(x2, y2); // 5.499
double cov3 = cov.covariance(x3, y3); // 5.497
double cov4 = cov.covariance(x4, y5); // 5.499
```

If you already have the data in a 2D array of doubles or a `RealMatrix` instance, you can pass them directly to the constructor like this:

```
// double[][] myData or RealMatrix myData
Covariance covObj = new Covariance(myData);
// cov contains covariances and can be accessed
// from RealMatrix.get(i,j) retrieve elements
RealMatrix cov = covObj.getCovarianceMatrix();
```

Note that the diagonal of the covariance matrix $\sigma^2_{i,j}$ is just the variance of column *i* and therefore the square root of the diagonal of the covariance matrix is the standard deviation of each dimension of data. Because the population mean is usually not known, we use the biased covariance with the sample mean. If we did know the population mean, the unbiased correction factor 1/*n* would be used to calculate the unbiased covariance:

$$\sigma^2_{i,j} = \frac{1}{n} C_2$$

Pearson's correlation

The Pearson correlation coefficient is related to covariance via the following, and is a measure of how likely two variates are to vary together:

$$\rho_{i,j} = \frac{\sigma_{i,j}}{\sigma_i \sigma_j}$$

The correlation coefficient takes a value between –1 and 1, where 1 indicates that the two variates are nearly identical. –1 indicates that they are opposites. In Java, there are once again two options, but using the default constructor, we get this:

```
PearsonsCorrelation corr = new PearsonsCorrelation();

/* examples using Anscombe's Quartet data */
double corr1 = corr.correlation(x1, y1)); // 0.816
double corr2 = corr.correlation(x2, y2)); // 0.816
double corr3 = corr.correlation(x3, y3)); // 0.816
double corr4 = corr.correlation(x4, y4)); // 0.816
```

However, if we already have data, or a `Covariance` instance, we can use the following:

```
// existing Covariance cov
PearsonsCorrelation corrObj = new PearsonsCorrelation(cov);
// double[][] myData or RealMatrix myData
PearsonsCorrelation corrObj = new PearsonsCorrelation(myData);
// from RealMatrix.get(i,j) retrieve elements
RealMatrix corr = corrObj.getCorrelationMatrix();
```

WARNING

Correlation is not causation! One of the dangers in statistics is the interpretation of correlation. When two variates have a high correlation, we tend to assume that this implies that one variable is responsible for causing the other. This is not the case. In fact, all you can assume is that you can reject the idea that the variates have nothing to do with each other. You should view correlation as a fortunate coincidence, not a foundational basis for the underlying behavior of the system being studied.

Regression

Often we want to find the relationship between our variates **X** and their responses y. We are trying to find a set of values for β such that $y = \mathbf{X}\hat{\beta}$. In the end, we want three things: the parameters, their errors, and a statistic of how good the fit is, R^2.

Simple regression

If **X** has only one dimension, the problem is the familiar equation of a line $y = \hat{\alpha} + \hat{\beta}x$, and the problem can be classified as simple regression. By calculating σ_x^2, the variance of x and $\sigma_{x,y}^2$, and the covariance between x and y, we can estimate the slope:

$$\hat{\beta} = \frac{\sigma_{x,y}^2}{\sigma_x^2}$$

Then, using the slope and the means of *x* and *y*, we can estimate the intercept:

$$\hat{\alpha} = \bar{y} - \hat{\beta}\bar{x}$$

The code in Java uses the `SimpleRegression` class:

```
SimpleRegression rg = new SimpleRegression();

/* x-y pairs of Anscombe's x1 and y1 */
double[][] xyData = {{10.0, 8.04}, {8.0, 6.95}, {13.0, 7.58},
        {9.0, 8.81}, {11.0, 8.33}, {14.0, 9.96}, {6.0, 7.24},
        {4.0, 4.26}, {12.0, 10.84}, {7.0, 4.82}, {5.0, 5.68}};

rg.addData(xyData);

/* get regression results */
double alpha = rg.getIntercept(); // 3.0
double alpha_err = rg.getInterceptStdErr(); // 1.12
double beta = rg.getSlope(); // 0.5
double beta_err = rg.getSlopeStdErr(); // 0.12
double r2 = rg.getRSquare(); // 0.67
```

We can then interpret these results as $y = 3.0 + 0.5x$, or more specifically as $y = (3.0 \pm 1.12) + (0.5 \pm 0.12)x$. How much can we trust this model? With $R^2 = 0.67$, it's a fairly decent fit, but the closer it is to the ideal of $R^2 = 1.0$, the better. Note that if we perform the same regression on the other three datasets from Anscombe's quartet, we get identical parameters, errors, and R^2. This is a profound, albeit perplexing, result. Clearly, the four datasets look different, but their linear fits (the superposed blue lines) are identical. Although linear regression is a powerful yet simple method for understanding our data as in case 1, in case 2 linear regression in *x* is probably the wrong tool

to use here. In case 3, linear regression is probably the right tool, but we could sensor (remove) the data point that appears to be an outlier. In case 4, a regression model is most likely not appropriate at all. This does demonstrate how easy it is to fool ourselves that a model is correct if we look at only a few parameters after blindly throwing data into an analysis method.

Multiple regression

There are many ways to solve this problem, but the most common and probably most useful is the ordinary least squares (OLS) method. The solution is expressed in terms of linear algebra:

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

The `OLSMultipleLinearRegression` class in Apache Commons Math is just a convenient wrapper around a QR decomposition. This implementation also provides additional functions beyond the QR decomposition that you will find useful. In particular, the variance-covariance matrix of β is as follows, where the matrix *R* is from the QR decomposition:

$$\sigma^2_{\hat{\beta}} = (X^T X)^{-1} = (R^T R)^{-1}$$

In this case, *R* must be truncated to the dimension of beta. Given the fit residuals $\hat{\epsilon} = y - X\hat{\beta}$, we can calculate the variance of the errors $s^2_{err} = \hat{\epsilon}^T \hat{\epsilon} / (n - p)$ where n and p are the respective number of rows and columns of X. The square root of the diagonal values of $\sigma^2_{\hat{\beta}}$ times the constant s_{err} gives us the estimate of errors on the fit parameters:

$$\delta\hat{\beta}_i = s_{err} \sqrt{\sigma^2_{\hat{\beta}_{i,i}}}$$

The Apache Commons Math implementation of ordinary least squares regression utilizes the QR decomposition covered in linear algebra. The methods in the example code are convenient wrappers around several standard matrix operations. Note that the default is to include an intercept term, and the corresponding value is the first position of the estimated parameters:

```
double[][] xNData = {{0, 0.5}, {1, 1.2}, {2, 2.5}, {3, 3.6}};
double[] yNData = {-1, 0.2, 0.9, 2.1};
// default is to include an intercept
OLSMultipleLinearRegression mr = new OLSMultipleLinearRegression();
/* NOTE that y and x are reversed compared to other classes / methods */
mr.newSampleData(yNData, xNData);
double[] beta = mr.estimateRegressionParameters();
// [-0.7499, 1.588, -0.5555]
double[] errs = mr.estimateRegressionParametersStandardErrors();
```

```
// [0.2635, 0.6626, 0.6211]
double r2 = mr.calculateRSquared();
// 0.9945
```

Linear regression is a vast topic with many adaptations — too many to be covered here. However, it is worth noting that these methods are relevant only if the relations between X and y are actually linear. Nature is full of nonlinear relationships, and in Chapter 5 we will address more ways of exploring these.

Working with Large Datasets

When our data is so large that it is inefficient to store it in memory (or it just won't fit!), we need an alternative method of calculating statistical measures. Classes such as `DescriptiveStatistics` store all data in memory for the duration of the instantiated class. However, another way to attack this problem is to store only the unnormalized statistical moments and update them one data point at a time, discarding that data point after it has been assimilated into the calculations. Apache Commons Math has two such classes: `SummaryStatistics` and `MultivariateSummaryStatistics`.

The usefulness of this method is enhanced by the fact that we can also sum unnormalized moments in parallel. We can split data into partitions and keep track of moments in each partition as we add one value at a time. In the end, we can merge all those moments and then find the summary statistics. The Apache Commons Math class `AggregateSummaryStatistics` takes care of this. It is easy to imagine terabytes of data distributed over a large cluster in which each node is updating statistical moments. As the jobs complete, the moments can be merged in a simple calculation, and the task is complete.

In general, a dataset $\mathbf{X}$ can be partitioned into k smaller datasets: $\mathbf{X}_1$, $\mathbf{X}_2$, $\cdots \mathbf{X}_k$. Ideally, we can perform all sorts of computations on each partition $\mathbf{X}_i$ and then later merge these results to get the desired quantities for $\mathbf{X}$. For example, if we wanted to count the number of data points in $\mathbf{X}$, we could count the number of points in each subset and then later add those results together to get the total count:

$$n = n_1 + n_2 \cdots + n_k$$

This is true whether the partitions were calculated on the same machine in different threads, or on different machines entirely.

So if we calculated the number of points, and additionally, the sum of values for each subset (and kept track of them), we could later use that information to calculate the mean of $\mathbf{X}$ in a distributed way.

$$\overline{x} = \frac{\sum_{x \in \mathbf{X}_1} x + \sum_{x \in \mathbf{X}_2} x + \cdots + \sum_{x \in \mathbf{X}_k} x}{n_1 + n_2 \cdots + n_k}$$

At the simplest level, we need only computations for pairwise operations, because any number of operations can reduce that way. For example, $\mathbf{X} = (\mathbf{X}_1 + \mathbf{X}_2) + (\mathbf{X}_3 + \mathbf{X}_4)$ is a combination of three pairwise operations. There are then three general situations for pairwise algorithms: first, where we are merging two partitions, each with $n_i > 1$; the second, where one partition has $n_i > 1$ and the other partition is a singleton with $n_i = 1$; and the third where both partitions are singletons.

Accumulating Statistics

We saw in the preceding chapter how stats can be updated. Perhaps it occurred to you that we could calculate and store the (unnormalized) moments on different machines at different times, and update them at our convenience. As long as you keep track of the number of points and all relevant statistical moments, you can recall those at any time and update them with a new set of data points. While the `DescriptiveStatistics` class stored all the data and did these updates in one long chain of calculations, the `SummaryStatistics` class (and `MultivariateSummaryStatistics` class) do not store any of the data you input to them. Rather, these classes store only the relevant n, M_1, and M_2. For massive datasets, this is an efficient way to keep track of stats without incurring the huge costs of storage or processing power whenever we need a statistic such as mean or standard deviation.

```
SummaryStatistics ss = new SummaryStatistics();

/* This class is storeless, so it is optimized to take one value at a time */
ss.addValue(1.0);
ss.addValue(11.0);
ss.addValue(5.0);

/* prints a report */
System.out.println(ss);
```

As with the `DescriptiveStatistics` class, the `SummaryStatistics` class also has a `toString()` method that prints a nicely formatted report:

```
SummaryStatistics:
n: 3
min: 1.0
max: 11.0
sum: 17.0
mean: 5.666666666666667
geometric mean: 3.8029524607613916
variance: 25.333333333333332
population variance: 16.88888888888889
second moment: 50.666666666666664
sum of squares: 147.0
standard deviation: 5.033222956847166
sum of logs: 4.007333185232471
```

For multivariate statistics, the `MultivariateSummaryStatistics` class is directly analogous to its univariate counterpart. To instantiate this class, you must specify the dimension of the variates (the number of columns in the dataset) and indicate whether the input data is a sample. Typically, this option should be set to `true`, but note that if you forget it, the default is `false`, and that will have consequences. The `MultivariateSummaryStatistics` class contains methods that keep track of the covariance between every set of variates. Setting the constructor argument `isCovarianceBiasedCorrected` to `true` uses the biased correction factor for the covariance:

```
MultivariateSummaryStatistics mss = new MultivariateSummaryStatistics(3, true);

/* data could be 2d array, matrix, or class with a double array data field */
double[] x1 = {1.0, 2.0, 1.2};
double[] x2 = {11.0, 21.0, 10.2};
double[] x3 = {5.0, 7.0, 0.2};

/* This class is storeless, so it is optimized to take one value at a time */
mss.addValue(x1);
mss.addValue(x2);
```

```
mss.addValue(x3);

/* prints a report */
System.out.println(mss);
```

As in `SummaryStatistics`, we can print a formatted report with the added bonus of the covariance matrix:

```
MultivariateSummaryStatistics:
n: 3
min: 1.0, 2.0, 0.2
max: 11.0, 21.0, 10.2
mean: 5.666666666666667, 10.0, 3.866666666666667
geometric mean: 3.8029524607613916, 6.649399761150975, 1.3477328201610665
sum of squares: 147.0, 494.0, 105.52
sum of logarithms: 4.007333185232471, 5.683579767338681, 0.8952713646500794
standard deviation: 5.033222956847166, 9.848857801796104, 5.507570547286103
covariance: Array2DRowRealMatrix{{25.3333333333,49.0,24.3333333333},
{49.0,97.0,51.0},{24.3333333333,51.0,30.3333333333}}
```

Of course, each of these quantities is accessible via their getters:

```
int d = mss.getDimension();
long n = mss.getDimension();
double[] min = mss.getMin();
double[] max = mss.getMax();
double[] mean = mss.getMean();
double[] std = mss.getStandardDeviation();
RealMatrix cov = mss.getCovariance();
```

> **NOTE**
>
> At this time, third- and fourth-order moments are not calculated in `SummaryStatistics` and `MultivariateSummaryStatistics` classes, so skewness and kurtosis are not available. They are in the works!

Merging Statistics

The unnormalized statistical moments and co-moments can also be merged. This is useful when data partitions are processed in parallel and the results are merged later when all subprocesses have completed.

For this task, we use the class `AggregateSummaryStatistics`. In general, statistical moments propagate as the order is increased. In other words, in order to calculate the third moment M_3 you will need the moments $M_2$and M_1. It is therefore essential to calculate and update the highest order moment first and then work downward.

For example, after calculating the quantity $\delta_{2,1}$ as described earlier, update M_4 with

$$M_4 = M_{4,1} + M_{4,2} + n_1 n_2 (n_1^2 - n_1 n_2 + n_2^2) \frac{\delta_{2,1}^4}{n^3} + 6(n_1^2 M_{2,2} - n_2^2 M_{2,1}) \frac{\delta_{2,1}^2}{n^2} + 4(n_1 M_{3,2} - n_2 M_{3,1}) \frac{\delta_{2,1}}{n}$$

Then update M_3 with

$$M_3 = M_{3,1} + M_{3,2} + n_1 n_2 (n_1 - n_2) \frac{\delta_{2,1}^3}{n^2} + 3(n_1 M_{2,2} - n_2 M_{2,1}) \frac{\delta_{2,1}}{n}$$

Next, update M_2 with:

$$M_2 = M_{2,1} + M_{2,2} + n_1 n_2 \frac{\delta_{2,1}^2}{n}$$

And finally, update the mean with:

$$\bar{x} = \bar{x}_1 + n_2 \frac{\delta_{2,1}}{n}$$

Note that these update formulas are for merging data partitions where both have $n_i > 1$. If either of the partitions is a singleton ($n_i = 1$), then use the incremental update formulas from the prior section.

Here is an example demonstrating the aggregation of independent statistical summaries. Note that here, any instance of `SummaryStatistics` could be serialized and stored away for future use.

```
// The following three summaries could occur on
// three different machines at different times

SummaryStatistics ss1 = new SummaryStatistics();
ss1.addValue(1.0);
ss1.addValue(11.0);
ss1.addValue(5.0);
```

```
SummaryStatistics ss2 = new SummaryStatistics();
ss2.addValue(2.0);
ss2.addValue(12.0);
ss2.addValue(6.0);

SummaryStatistics ss3 = new SummaryStatistics();
ss3.addValue(0.0);
ss3.addValue(10.0);
ss3.addValue(4.0);

// The following can occur on any machine at
// any time later than above

List<SummaryStatistics> ls = new ArrayList<>();
ls.add(ss1);
ls.add(ss2);
ls.add(ss3);

StatisticalSummaryValues s = AggregateSummaryStatistics.aggregate(ls);

System.out.println(s);
```

This prints the following report as if the computation had occurred on a single dataset:

```
StatisticalSummaryValues:
n: 9
min: 0.0
max: 12.0
mean: 5.666666666666667
std dev: 4.444097208657794
variance: 19.75
sum: 51.0
```

Regression

The `SimpleRegression` class makes this easy, because moments and co-moments add together easily. The Aggregates statistics produce the same result as in the original statistical summary:.

```
SimpleRegression rg = new SimpleRegression();

/* x-y pairs of Anscombe's x1 and y1 */
double[][] xyData = {{10.0, 8.04}, {8.0, 6.95}, {13.0, 7.58},
        {9.0, 8.81}, {11.0, 8.33}, {14.0, 9.96}, {6.0, 7.24},
        {4.0, 4.26}, {12.0, 10.84}, {7.0, 4.82}, {5.0, 5.68}};

rg.addData(xyData);

/**/
double[][] xyData2 = {{10.0, 8.04}, {8.0, 6.95}, {13.0, 7.58},
        {9.0, 8.81}, {11.0, 8.33}, {14.0, 9.96}, {6.0, 7.24},
        {4.0, 4.26}, {12.0, 10.84}, {7.0, 4.82}, {5.0, 5.68}};

SimpleRegression rg2 = new SimpleRegression();
rg2.addData(xyData);

/* merge the regression from rg with rg2 */
rg.append(rg2);

/* get regression results for the combined regressions */
double alpha = rg.getIntercept(); // 3.0
double alpha_err = rg.getInterceptStdErr(); // 1.12
double beta = rg.getSlope(); // 0.5
double beta_err = rg.getSlopeStdErr(); // 0.12
double r2 = rg.getRSquare(); // 0.67
```

In the case of multivariate regression, `MillerUpdatingRegression` enables a storeless regression via `MillerUpdatingRegression.addObservation(double[] x, double y)` or `MillerUpdatingRegression.addObservations(double[][] x, double[] y)`.

```
int numVars = 3;
boolean includeIntercept = true;
MillerUpdatingRegression r =
    new MillerUpdatingRegression(numVars, includeIntercept);
double[][] x = {{0, 0.5}, {1, 1.2}, {2, 2.5}, {3, 3.6}};
double[] y = {-1, 0.2, 0.9, 2.1};
r.addObservations(x, y);
RegressionResults rr = r.regress();
double[] params = rr.getParameterEstimates();
double[] errs = rr.getStdErrorOfEstimates();
double r2 = rr.getRSquared();
```

Using Built-in Database Functions

Most databases have built-in statistical aggregation functions. If your data is already in MySQL you may not have to import the data to a Java application. You can use built-in functions. The use of `GROUP BY` and `ORDER BY` combined with a `WHERE` clause make this a powerful way to reduce your data to statistical summaries. Keep in mind that the computation must be done somewhere, either in your application or by the database server. The trade-off is, is the data small enough that I/O and CPU is not an issue? If you don't want the DB performance to take a hit CPU-wise, exploiting all that I/O bandwidth might be OK. Other times, you would rather have the CPU in the DB app compute all the stats and use just a tiny bit of I/O to shuttle back the results to the waiting app.

> **WARNING**
>
> In MySQL, the built-in function `STDDEV` returns the population's standard deviation. Use the more specific functions `STDDEV_SAMP` and `STDDEV_POP` for respective sample and population standard deviations.

For example, we can query a table with various built-in functions, which in this case are example revenue statistics such as AVG and STDDEV from a sales table:

```
SELECT city, SUM(revenue) AS total_rev, AVG(revenue) AS avg_rev,
    STDDEV(revenue) AS std_rev
    FROM sales_table WHERE <some criteria> GROUP BY city ORDER BY total_rev DESC;
```

Note that we can use the results as is from a JDBC query or dump them directly into the constructor of `StatisticalSummaryValues(double mean, double variance, long count, double min, double max)` for further use down the line. Say we have a query like this:

```
SELECT city, AVG(revenue) AS avg_rev,
             VAR_SAMP(revenue) AS var_rev,
             COUNT(revenue) AS count_rev,
MIN(revenue) AS min_rev, MAX(revenue) AS max_rev
    FROM sales_table WHERE <some criteria> GROUP BY city;
```

We can populate each `StatistialSummaryValues` instance (arbitrarily) in a `List` or `Map` with keys equal to `city` as we iterate through the database cursor:

```
Map<String, StatisticalSummaryValues> revenueStats = new HashMap<>();

Statement st = c.createStatement();
ResultSet rs = st.executeQuery(selectSQL);
while(rs.next()) {
    StatisticalSummaryValues ss = new StatisticalSummaryValues(
        rs.getDouble("avg_rev"),
        rs.getDouble("var_rev"),
        rs.getLong("count_rev"),
        rs.getDouble("min_rev"),
        rs.getDouble("max_rev") );

     revenueStats.put(rs.getString("city"), ss);
}
rs.close();
st.close();
```

Some simple database wizardry can save lots of I/O for larger datasets.

Chapter 4. Data Operations

Now that we know how to input data into a useful data structure, we can operate on that data by using what we know about statistics and linear algebra. There are many operations we perform on data before we subject it to a learning algorithm. Often called *preprocessing*, this step comprises data cleaning, regularizing or scaling the data, reducing the data to a smaller size, encoding text values to numerical values, and splitting the data into parts for model training and testing. Often our data is already in one form or another (e.g., `List` or `double[][]`), and the learning routines we will use may take either or both of those formats. Additionally, a learning algorithm may need to know whether the labels are binary or multiclass or even encoded in some other way such as text. We need to account for this and prepare the data before it goes in the learning algorithm. The steps in this chapter can be part of an automated pipeline that takes raw data from the source and prepares it for either learning or prediction algorithms.

Transforming Text Data

Many learning and prediction algorithms require numerical input. One of the simplest ways to achieve this is by creating a vector space model in which

we define a vector of known length and then assign a collection of text

snippets (or even words) to a corresponding collection of vectors. The

general process of converting text to vectors has many options and

variations. Here we will assume that there exists a large body of text

(corpus) that can be divided into sentences or lines (documents) that can

in turn be divided into words (tokens). Note that the definitions of

corpus, *document*, and

token are user-definable.

Extracting Tokens from a Document

For each document, we want to extract all the tokens. Because there are many ways to approach this problem, we can create an interface with

a method that takes in a document string and returns an array of

`String` tokens:

```
public interface Tokenizer {
    String[] getTokens(String document);
}
```

The tokens may have lots of characters that are undesirable, such as punctuation, numbers, or other characters. Of course, this will entirely depend on your application. In this example, we are concerned only with the actual content of regular English words, so we can clean the tokens to accept only lowercase alphabetical characters. Including a variable for minimum token size enables us to skip words such as *a*, *or*, and *at*.

```
public class SimpleTokenizer implements Tokenizer {


    private final int minTokenSize;

    public SimpleTokenizer(int minTokenSize) {
        this.minTokenSize = minTokenSize;
    }

    public SimpleTokenizer() {
        this(0);
    }


    @Override
    public String[] getTokens(String document) {
        String[] tokens = document.trim().split("\\s+");
        List<String> cleanTokens = new ArrayList<>();
        for (String token : tokens) {
            String cleanToken = token.trim().toLowerCase()
              .replaceAll("[^A-Za-z\']+", "");
            if(cleanToken.length() > minTokenSize) {
                cleanTokens.add(cleanToken);
            }
        }
        return cleanTokens.toArray(new String[0]);
    }
}
```

Utilizing Dictionaries

A *dictionary* is a list of terms that are relevant (i.e., a "vocabulary"). There is more than one strategy to implement a dictionary. The important feature is that each term needs to be associated with an integer value that corresponds to its location in a vector. Of course, this can be an array that is searched by position, but for large dictionaries, this is inefficient and a `Map` is better. For much larger dictionaries, we can skip the term storage and use the hashing trick. In general, we need to know the number of terms in the dictionary for creating vectors as well as a method that returns the index of particular term. Note that int cannot be `null`, so by using the boxed type `Integer`, a returned index can either be an `int` or `null` value.

```
public interface Dictionary {
    Integer getTermIndex(String term);
    int getNumTerms();
}
```

We can build a dictionary of the exact terms collected from the `Tokenizer` instance. Note that the strategy is to add a term and integer for each item. New items will increment the counter, and duplicates will be discarded without incrementing the counter. In this case, the `TermDictionary` class needs methods for adding new terms:

```
public class TermDictionary implements Dictionary {


    private final Map<String, Integer> indexedTerms;
    private int counter;

    public TermDictionary() {
        indexedTerms = new HashMap<>();
        counter = 0;
    }


    public void addTerm(String term) {
        if(!indexedTerms.containsKey(term)) {
            indexedTerms.put(term, counter++);
        }

    }

    public void addTerms(String[] terms) {
        for (String term : terms) {
```

```
            addTerm(term);
        }
    }


    @Override
    public Integer getTermIndex(String term) {
        return indexedTerms.get(term);
    }

    @Override
    public int getNumTerms() {
        return indexedTerms.size();
    }
}
```

For a large number of terms, we can use the hashing trick. Essentially, we use the hash code of the `String` value for each term and then take the modulo of the number of terms that will be in the dictionary. For a large number of terms (about 1 million), collisions are unlikely. Note that unlike with `TermDictionary`, we do not need to add terms or keep track of terms. Each term index is calculated on the fly. The number of terms is a constant that we set. For efficiency in hash table retrieval, it's a good idea to make the number of terms equal to 2^n. For around 2^{20}, it will be approximately 1 million terms.

```
public class HashingDictionary implements Dictionary {


    private int numTerms; // 2^n is optimal

    public HashingDictionary() {
        // 2^20 = 1048576
        this(new Double(Math.pow(2,20)).intValue());
    }

    public HashingDictionary(int numTerms) {
        this.numTerms = numTerms;
    }


    @Override
    public Integer getTermIndex(String term) {
        return Math.floorMod(term.hashCode(), numTerms);
    }

    @Override
    public int getNumTerms() {
        return numTerms;
    }
}
```

Vectorizing a Document

Now that we have a tokenizer and a dictionary, we can turn a list of words into numeric values that can be passed into machine-learning

algorithms. The most straightforward way is to first decide on what the

dictionary is, and then count the number of occurrences that are in the

sentence (or text of interest). This is often called *bag of words*. In some cases, we want to know only whether a word occurred. In such a case, a 1 is

placed in the vector as opposed to a count:

```
public class Vectorizer {


    private final Dictionary dictionary;
    private final Tokenizer tokenzier;
    private final boolean isBinary;

    public Vectorizer(Dictionary dictionary, Tokenizer tokenzier,
        boolean isBinary) {
        this.dictionary = dictionary;
        this.tokenzier = tokenzier;
        this.isBinary = isBinary;
    }



    public Vectorizer() {
        this(new HashingDictionary(), new SimpleTokenizer(), false);
    }

    public RealVector getCountVector(String document) {
        RealVector vector = new OpenMapRealVector(dictionary.getNumTerms());
        String[] tokens = tokenzier.getTokens(document);
        for (String token : tokens) {
            Integer index = dictionary.getTermIndex(token);
            if(index != null) {
                if(isBinary) {
                    vector.setEntry(index, 1);
                } else {
                    vector.addToEntry(index, 1); // increment !
                }
            }
        }
        return vector;
    }

    public RealMatrix getCountMatrix(List<String> documents) {
        int rowDimension = documents.size();
        int columnDimension = dictionary.getNumTerms();
        RealMatrix matrix = new OpenMapRealMatrix(rowDimension, columnDimension);
        int counter = 0;
        for (String document : documents) {
            matrix.setRowVector(counter++, getCountVector(document));
        }
        return matrix;
    }
}
```

In some cases, we will want to reduce the effects of common words.

The term frequency — inverse document frequency (TFIDF) vector does just that. The TFIDF component

is highest when a term occurs many
times within a small number of documents but lowest when the term occurs
in nearly all documents. Note that TFIDF is just term frequency times
the inverse document frequency: $TFIDF = TF \times IDF$. Here TF is the number of times a term has appeared
in a document (its count vector). IDF is the (pseudo)inverse of the
document frequency, DF, the number of documents the term has appeared
in. In general, we can compute TF by counting terms per document, and DF
by computing a binary vector over each document and cumulatively summing
those vectors as we process each document. The most common form of the
TFIDF is shown here, where *N* is the total number of
documents processed:

$$TFIDF_{t,d} = TF_{t,d} \log\left(N / DF_t\right)$$

This is just one strategy for TFIDF. Note that the log function
will cause trouble if either *N* or
DF has zero values. Some strategies avoid this by
adding in small factors or 1. We can handle it in our implementation by
setting `log(0)` to 0. In general, our implementation here is
to first create a matrix of counts and then operate over that matrix,
converting each term into its weighted TFIDF value. Because these
matrices are usually sparse, it's a good idea to use the optimized
order-walking operator:

```
public class TFIDF implements RealMatrixChangingVisitor {


    private final int numDocuments;
    private final RealVector termDocumentFrequency;
    double logNumDocuments;


    public TFIDF(int numDocuments, RealVector termDocumentFrequency) {
        this.numDocuments = numDocuments;
        this.termDocumentFrequency = termDocumentFrequency;
        this.logNumDocuments = numDocuments > 0 ? Math.log(numDocuments) : 0;
    }


    @Override
    public void start(int rows, int columns, int startRow, int endRow,
            int startColumn, int endColumn) {
        //NA
```

```
    }

    @Override
    public double visit(int row, int column, double value) {
        double df = termDocumentFrequency.getEntry(column);
        double logDF = df > 0 ? Math.log(df) : 0.0;
        // TFIDF = TF_i * log(N/DF_i) = TF_i * ( log(N) - log(DF_i) )
        return value * (logNumDocuments - logDF);
    }

    @Override
    public double end() {
        return 0.0;
    }


}
```

Then `TFIDFVectorizer` uses both counts and binary counts:

```
public class TFIDFVectorizer {


    private Vectorizer vectorizer;
    private Vectorizer binaryVectorizer;
    private int numTerms;


    public TFIDFVectorizer(Dictionary dictionary, Tokenizer tokenzier) {
        vectorizer = new Vectorizer(dictionary, tokenzier, false);
        binaryVectorizer = new Vectorizer(dictionary, tokenzier, true);
        numTerms = dictionary.getNumTerms();
    }

    public TFIDFVectorizer() {
       this(new HashingDictionary(), new SimpleTokenizer());
    }


    public RealVector getTermDocumentCount(List<String> documents) {
        RealVector vector = new OpenMapRealVector(numTerms);
        for (String document : documents) {
            vector.add(binaryVectorizer.getCountVector(document));
        }
        return vector;
    }


    public RealMatrix getTFIDF(List<String> documents) {
        int numDocuments = documents.size();
        RealVector df = getTermDocumentCount(documents);
        RealMatrix tfidf = vectorizer.getCountMatrix(documents);
        tfidf.walkInOptimizedOrder(new TFIDF(numDocuments, df));
        return tfidf;
    }
}
```

Here's an example using the sentiment dataset described in Appendix A: `/* sentiment data ... see appendix */ Sentiment sentiment = new Sentiment();`

`/* create a dictionary of all terms */ TermDictionary termDictionary = new TermDictionary();`

```
/* need a basic tokenizer to parse text */ SimpleTokenizer tokenizer = new SimpleTokenizer();

/* add all terms in sentiment dataset to dictionary */ for (String document : sentiment.getDocuments()) { String[]tokens = tokenizer.getTokens(document); termDictionary.addTerms(tokens); }

/* create of matrix of word counts for each sentence */ Vectorizer vectorizer = new Vectorizer(termDictionary, tokenizer, false); RealMatrix counts = vectorizer.getCountMatrix(sentiment.getDocuments());

/* ... or create a binary counter */ Vectorizer binaryVectorizer = new Vectorizer(termDictionary, tokenizer, true); RealMatrix binCounts = binaryVectorizer.getCountMatrix(sentiment.getDocuments());

/* ... or create a matrix TFIDF */ TFIDFVectorizer tfidfVectorizer = new TFIDFVectorizer(termDictionary, tokenizer); RealMatrix tfidf = tfidfVectorizer.getTFIDF(sentiment.getDocuments());
```

Scaling and Regularizing Numeric Data

Should we pull data from our classes or use the arrays as is? Our goal is to apply some transform of each element in the dataset such that

$f(x_{i,j}) \rightarrow x^*_{i,j}$. There are two basic ways to scale data: either by column or row. For column scaling, we just need to collect the statistics for each column of data. In particular, we need the min, max, mean, and standard deviation. So if we add the entire dataset to a `MultivariateSummaryStatistics` instance, we will have all of that. In the other case of row scaling, we need to collect the L1 or L2 normalization of each row. We can store those in a `RealVector` instance, which can be sparse.

> **WARNING**
>
> If you scale the data to train the model, retain any mins, maxes, means, or standard deviations you have used! You must use the same technique, including the stored parameters, when transforming a *new* dataset that you will use for prediction. Note that if you are splitting data into Train/Validate/Test sets, then scale the training data and use those values (e.g., means) to scale the validation and test sets so they will be unbiased.

Scaling Columns

The general form for scaling columns is to use a `RealMatrixChangingVisitor` with precomputed column statistics passed into the constructor. As the
operation visits each matrix entry, the appropriate column statistics
can be utilized.

```
public class MatrixScalingOperator implements RealMatrixChangingVisitor {


    MultivariateSummaryStatistics mss;


    public MatrixScalingOperator(MultivariateSummaryStatistics mss) {
        this.mss = mss;
    }

    @Override
    public void start(int rows, int columns, int startRow, int endRow,
      int startColumn, int endColumn) {
        // nothing
    }

    @Override
    public double visit(int row, int column, double value) {
        // implement specific type here
    }

    @Override
    public double end() {
        return 0.0;
    }
}
```

Min-max scaling

Min-max scaling ensures that the smallest value is 0 and the largest value is 1
for each column independently. We can transform each element
i with min and max for that column
j:

$$x_{i,j}^{*} = \frac{x_{i,j} - x_j^{min}}{x_j^{max} - x_j^{min}}$$

This can be implemented as follows:

```
public class MatrixScalingMinMaxOperator implements RealMatrixChangingVisitor {
...
    @Override
```

```
        public double visit(int row, int column, double value) {
            double min = mss.getMin()[column];
            double max = mss.getMax()[column];
            return (value - min) / (max - min);
        }
    ...
    }
```

At times we want to specify the lower *a* and

upper *b* limits (instead of 0 and 1). In this

case, we calculate the 0:1 scaled data first and then apply a second

round of scaling:

$$x_{i,j}^{a,b} = x_{i,j}^{*}(b-a)+a$$

Centering the data

Centering the data around the mean value ensures that the average of the data column will be zero. However, there can still be extreme mins and

maxes because they are unbounded. Every value in one column is

translated by that column's mean:

$$x_{i,j}^{*} = x_{i,j} - \overline{x}_{j}$$

This can be implemented as follows:

```
    @Override
    public double visit(int row, int column, double value) {
        double mean = mss.getMean()[column];
        return value - mean;
    }
```

Unit normal scaling

Unit normal scaling is also known as a *z-score*. It rescales every data point in a column such that it is a member of unit

normal distribution by centering it about the mean and dividing by the

standard deviation. Each column will then have an average value of

zero, and its distribution of values will mostly be smaller than 1,

although as a distribution, this is not guaranteed because the values

are unbounded.

$$x^*_{i,j} = \frac{x_{i,j} - \overline{x}_j}{\sigma_j}$$

This can be implemented as follows:

```
@Override
public double visit(int row, int column, double value) {
    double mean = mss.getMean()[column];
    double std = mss.getStandardDeviation()[column];
    return (value - mean) / std;
}
```

Scaling Rows

When each row of data is a record across all variables, scaling by row is typically to perform an L1 or L2 regularization:

```
public class MatrixScalingOperator implements RealMatrixChangingVisitor {


    RealVector normals;

    public MatrixScalingOperator(RealVector normals) {
        this.normals = normals;
    }

    @Override
    public void start(int rows, int columns, int startRow, int endRow,
       int startColumn, int endColumn) {
        // nothing
    }

    @Override
    public double visit(int row, int column, double value) {
        //implement
    }

    @Override
    public double end() {
        return 0.0;
    }
}
```

L1 regularization

In this case, we are normalizing each row of data such that the sum of (absolute) values is equal to 1, because we divide each element

j of row *i* by the row L1

normal:

$$x^*_{i,j} = \frac{x_{i,j}}{\| \mathbf{x_i} \|}$$

We implement this as follows:

```
@Override
public double visit(int row, int column, double value) {
    double rowNormal = normals.getEntry(row);
    return ( rowNormal > 0 ) ? value / rowNormal : 0;
}
```

L2 regularization

L2 regularization scales by row, not column. In this case, we are normalizing each row of data as we divide each element

j of row *i* by the row L2

normal. The length of each row will now be equal to 1:

$$x^*_{i,j} = \frac{x_{i,j}}{\| \mathbf{x_i} \|}$$

We implement this with the following:

```
@Override
public double visit(int row, int column, double value) {
    double rowNormal = normals.getEntry(row);
    return ( rowNormal > 0 ) ? value / rowNormal : 0;
}
```

Matrix Scaling Operator

We can collect the scaling algorithms in static methods because we are altering the matrix in place:

```
public class MatrixScaler {


    public static void minmax(RealMatrix matrix) {
        MultivariateSummaryStatistics mss = getStats(matrix);
        matrix.walkInOptimizedOrder(new MatrixScalingMinMaxOperator(mss));
    }


    public static void center(RealMatrix matrix) {
        MultivariateSummaryStatistics mss = getStats(matrix);
        matrix.walkInOptimizedOrder(
          new MatrixScalingOperator(mss, MatrixScaleType.CENTER));
    }


    public static void zscore(RealMatrix matrix) {
        MultivariateSummaryStatistics mss = getStats(matrix);
        matrix.walkInOptimizedOrder(
          new MatrixScalingOperator(mss, MatrixScaleType.ZSCORE));
    }


    public static void l1(RealMatrix matrix) {
        RealVector normals = getL1Normals(matrix);
        matrix.walkInOptimizedOrder(
          new MatrixScalingOperator(normals, MatrixScaleType.L1));
    }


    public static void l2(RealMatrix matrix) {
        RealVector normals = getL2Normals(matrix);
        matrix.walkInOptimizedOrder(
          new MatrixScalingOperator(normals, MatrixScaleType.L2));
    }


    private static RealVector getL1Normals(RealMatrix matrix) {
        RealVector normals = new OpenMapRealVector(matrix.getRowDimension());
        for (int i = 0; i < matrix.getRowDimension(); i++) {
            double l1Norm = matrix.getRowVector(i).getL1Norm();
            if (l1Norm > 0) {
                normals.setEntry(i, l1Norm);
            }
        }
        return normals;
    }


    private static RealVector getL2Normals(RealMatrix matrix) {
        RealVector normals = new OpenMapRealVector(matrix.getRowDimension());
        for (int i = 0; i < matrix.getRowDimension(); i++) {
            double l2Norm = matrix.getRowVector(i).getNorm();
            if (l2Norm > 0) {
                normals.setEntry(i, l2Norm);
            }
        }
        return normals;
    }
```

```
    private static MultivariateSummaryStatistics getStats(RealMatrix matrix) {
        MultivariateSummaryStatistics mss =


        new MultivariateSummaryStatistics(matrix.getColumnDimension(), true);
        for (int i = 0; i < matrix.getRowDimension(); i++) {
            mss.addValue(matrix.getRow(i));
        }
        return mss;
    }
}
```

Now it is really easy to use it: `RealMatrix matrix = new OpenMapRealMatrix(10, 3); matrix.addToEntry(0, 0, 1.0); matrix.addToEntry(0, 2, 2.0); matrix.addToEntry(1, 0, 1.0); matrix.addToEntry(2, 0, 3.0); matrix.addToEntry(3, 1, 5.0); matrix.addToEntry(6, 2, 1.0); matrix.addToEntry(8, 0, 8.0); matrix.addToEntry(9, 1, 3.0);`

`/* scale matrix in-place */ MatrixScaler.minmax(matrix);`

Reducing Data to Principal Components

The goal of a *principal components analysis* (PCA) is to transform a dataset into another dataset with fewer dimensions. We can envision this as applying a function

f to an *m* ×

n matrix X such that the result will be an *m* × *k* matrix X_k, where *k* <

n:

$$\mathbf{X}_k = f(\mathbf{X})$$

This is achieved by finding the eigenvectors and eigenvalues via linear algebra algorithms.

One benefit of this type of transformation is that the new dimensions are

ordered from most significant to least. For multidimensional data, we can

sometimes gain insight into any significant relationships by plotting the

first two dimensions of the principal components. In Figure 4-1, we have plotted the first two principal components of the Iris dataset (see Appendix A). The Iris dataset is a four-dimensional set of features with three possible labels.

In this image, we note that by plotting the original data projected onto

the first two principal components, we can see a separation of the three

classes. This distinction does not occur when plotting any of the two

dimensions from the original dataset.

However, for high-dimensional data, we need a more robust way of

determining the number of principal components to keep. Because the

principal components are ordered from most significant to least, we can

formulate the explained variance of the principal components by computing

the normalized, cumulative sum of the eigenvalues λ:

$$\sigma^2(k) = \frac{1}{\sigma^2_{max}} \sum_{i=1}^{k} \lambda_i$$

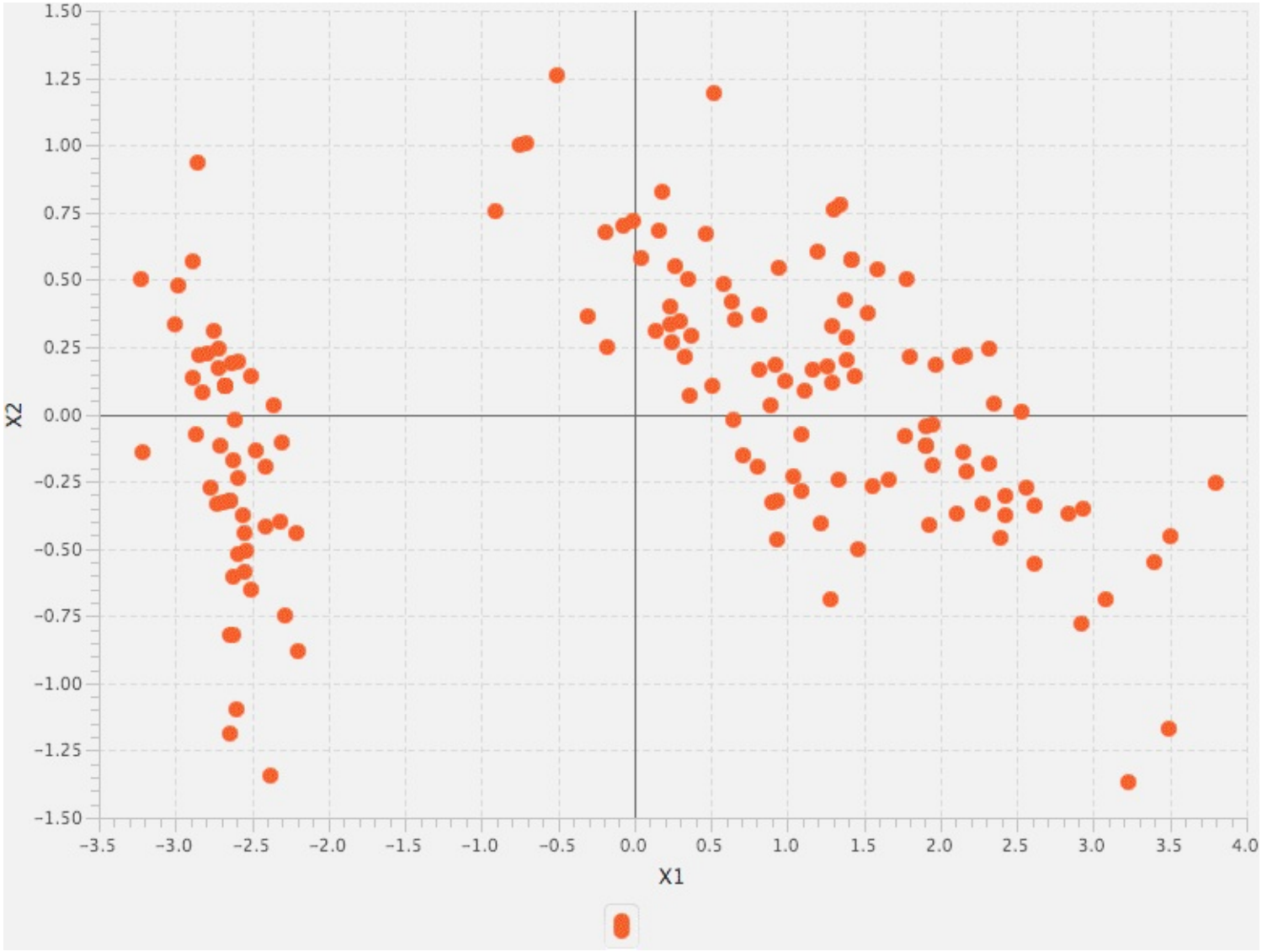

Figure 4-1. IRIS data's first two principal components

Here, each additional component explains an additional percentage of the data. There are then two uses for the explained variance. When we explicitly choose a number of principal components, we can calculate how much of the original dataset is explained by this new transformation. In the other case, we can iterate through the explained variance vector and stop at a particular number of components, *k*, when we have reached a desired coverage.

When implementing a principal components analysis, there are several strategies for computing the eigenvalues and eigenvectors. In the end, we just want to retrieve the transformed data. This is a great case for the strategy pattern in which the implementation details can be contained in separate classes, while the main `PCA` class is mostly just a shell:

```java
public class PCA {



    private final PCAImplementation pCAImplementation;

    public PCA(RealMatrix data, PCAImplementation pCAImplementation) {
        this.pCAImplementation = pCAImplementation;
        this.pCAImplementation.compute(data);
    }

    public RealMatrix getPrincipalComponents(int k) {
        return pCAImplementation.getPrincipalComponents(k);
    }



    public RealMatrix getPrincipalComponents(int k, RealMatrix otherData) {
        return pCAImplementation.getPrincipalComponents(k, otherData);
    }



    public RealVector getExplainedVariances() {
        return pCAImplementation.getExplainedVariances();
    }



    public RealVector getCumulativeVariances() {
        RealVector variances = getExplainedVariances();
        RealVector cumulative = variances.copy();
        double sum = 0;
        for (int i = 0; i < cumulative.getDimension(); i++) {
            sum += cumulative.getEntry(i);
            cumulative.setEntry(i, sum);
        }
        return cumulative;
    }



    public int getNumberOfComponents(double threshold) {
        RealVector cumulative = getCumulativeVariances();
        int numComponents=1;
        for (int i = 0; i < cumulative.getDimension(); i++) {
            numComponents = i + 1;
            if(cumulative.getEntry(i) >= threshold) {

                break;
            }
        }
        return numComponents;
    }



    public RealMatrix getPrincipalComponents(double threshold) {
        int numComponents = getNumberOfComponents(threshold);
        return getPrincipalComponents(numComponents);
    }



    public RealMatrix getPrincipalComponents(double threshold,
        RealMatrix otherData) {
        int numComponents = getNumberOfComponents(threshold);
        return getPrincipalComponents(numComponents, otherData);
    }

}
```

We can then provide an interface `PCAImplementation` for the following methods of decomposing the input data into its principal components:

```
public interface PCAImplementation {


    void compute(RealMatrix data);


    RealVector getExplainedVariances();


    RealMatrix getPrincipalComponents(int numComponents);


    RealMatrix getPrincipalComponents(int numComponents, RealMatrix otherData);
}
```

Covariance Method

One method for calculating the PCA is by finding the eigenvalue decomposition of the covariance matrix of **X**. The principal

components of a centered matrix **X** are

the eigenvectors of the covariance:

$$\mathbf{C} = \frac{1}{n-1}(\mathbf{X} - \overline{\mathbf{X}})^T(\mathbf{X} - \overline{\mathbf{X}})$$

This method of covariance calculation can be computationally

intensive because it requires multiplying together two potentially large

matrices. However, in Chapter 3, we explored

an efficient update formula for computing covariance that does not

require matrix transposition. When using the Apache Commons Math class

`Covariance`, or other classes that implement it (e.g.,

`MultivariateSummaryStatistics`), the efficient update

formula is used. Then the covariance **C** can be decomposed into the following:

$$\mathbf{C} = \mathbf{V}\mathbf{D}\mathbf{V}^T$$

The columns of **V** are the

eigenvectors, and the diagonal components of **D** are the eigenvalues. The Apache Commons Math

implementation orders the eigenvalues (and corresponding eigenvectors)

from largest to smallest. Typically, we want only *k* components, and therefore we need only the first *k* columns of **V**. The mean-centered data

can be projected onto the new components with a matrix

multiplication:

$$\mathbf{X}_k = (\mathbf{X} - \overline{\mathbf{X}})\mathbf{V}_k$$

Here is an implementation of a principal components analysis using

the covariance method:

```
public class PCAEIGImplementation implements PCAImplementation {

    private RealMatrix data;
    private RealMatrix d; // eigenvalue matrix
```

```
    private RealMatrix v; // eigenvector matrix
    private RealVector explainedVariances;
    private EigenDecomposition eig;
    private final MatrixScaler matrixScaler;

    public PCAEIGImplementation() {
        matrixScaler = new MatrixScaler(MatrixScaleType.CENTER);
    }

    @Override
    public void compute(RealMatrix data) {
        this.data = data;
        eig = new EigenDecomposition(new Covariance(data).getCovarianceMatrix());
        d = eig.getD();
        v = eig.getV();
    }


    @Override
    public RealVector getExplainedVariances() {
        int n = eig.getD().getColumnDimension(); //colD = rowD
        explainedVariances = new ArrayRealVector(n);
        double[] eigenValues = eig.getRealEigenvalues();
        double cumulative = 0.0;
        for (int i = 0; i < n; i++) {
            double var = eigenValues[i];
            cumulative += var;
            explainedVariances.setEntry(i, var);
        }
        /* dividing the vector by the last (highest) value maximizes to 1 */
        return explainedVariances.mapDivideToSelf(cumulative);
    }

    @Override
    public RealMatrix getPrincipalComponents(int k) {
        int m = eig.getV().getColumnDimension(); // rowD = colD
        matrixScaler.transform(data);
        return data.multiply(eig.getV().getSubMatrix(0, m-1, 0, k-1));
    }

    @Override
    public RealMatrix getPrincipalComponents(int numComponents,
        RealMatrix otherData) {
        int numRows = v.getRowDimension();
        // NEW data transformed under OLD means
        matrixScaler.transform(otherData);
        return otherData.multiply(
            v.getSubMatrix(0, numRows-1, 0, numComponents-1));
    }

}
```

Then it can be used, for example, to get the first three principal components, or to get all the components that provide 50 percent explained variance: `/* use the eigenvalue decomposition implementation */ PCA pca = new PCA(data, new PCAEIGImplementation());`

`/* get first three components */ RealMatrix pc3 = pca.getPrincipalComponents(3);`

`/* get however many components are needed to satisfy 50% explained variance */ RealMatrix pct = pca.getPrincipalComponents(.5);`

SVD Method

If $\mathbf{X} - \overline{\mathbf{X}}$ is a mean-centered dataset with *m* rows and *n* columns, the principal components are calculated from the following:

$$\mathbf{X} - \overline{\mathbf{X}} = \mathbf{U}\Sigma\mathbf{V}^T$$

Note the familiar form for a singular value decomposition, $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$, in which the column vectors of **V** are the eigenvectors, and the eigenvalues are derived from the diagonal of **Σ** via $\lambda_i = \Sigma_{i,i}^2 / (m-1)$; *m* is the number of rows of data. After performing the singular value decomposition on the mean-centered **X**, the projection is then as follows:

$$\mathbf{X}_k = \mathbf{U}_k\Sigma_k$$

We've kept only the first *k* columns of **U** and the *k* × *k* upper-left submatrix of Σ. It is also correct to compute the projection with the original, mean-centered data and the eigenvectors:

$$\mathbf{X}_k = (\mathbf{X} - \overline{\mathbf{X}})\mathbf{V}_k$$

Here we keep only the first *k* columns of **V**. In particular, this expression is used when we are transforming a new set of data with the existing eigenvectors and means. Note that the means are those that the PCA was trained on, not the means of the input data. This is the same form as in the eigenvalue method in the preceding section.

Apache Commons Math implementation is *compact SVD* because there are at most *p* = *min(m,n)* singular values, so there is no need to calculate the full SVD as discussed in Chapter 2. Following is the SVD implementation of a principal components analysis and is the preferred method:

```
public class PCASVDImplementation implements PCAImplementation {
```

```
    private RealMatrix u;
    private RealMatrix s;
    private RealMatrix v;
    private MatrixScaler matrixScaler;
    private SingularValueDecomposition svd;

    @Override
    public void compute(RealMatrix data) {
        MatrixScaler.center(data);
        svd = new SingularValueDecomposition(data);
        u = svd.getU();
        s = svd.getS();
        v = svd.getV();
    }


    @Override
    public RealVector getExplainedVariances() {
        double[] singularValues = svd.getSingularValues();
        int n = singularValues.length;
        int m = u.getRowDimension(); // number of rows in U is same as in data
        RealVector explainedVariances = new ArrayRealVector(n);
        double sum = 0.0;
        for (int i = 0; i < n; i++) {
            double var = Math.pow(singularValues[i], 2) / (double)(m-1);
            sum += var;
            explainedVariances.setEntry(i, var);
        }
        /* dividing the vector by the last (highest) value maximizes to 1 */
        return explainedVariances.mapDivideToSelf(sum);


    }

    @Override
    public RealMatrix getPrincipalComponents(int numComponents) {
        int numRows = svd.getU().getRowDimension();
        /* submatrix limits are inclusive */
        RealMatrix uk = u.getSubMatrix(0, numRows-1, 0, numComponents-1);
        RealMatrix sk = s.getSubMatrix(0, numComponents-1, 0, numComponents-1);
        return uk.multiply(sk);
    }

    @Override
    public RealMatrix getPrincipalComponents(int numComponents,
        RealMatrix otherData) {
        // center the (new) data on means from original data
        matrixScaler.transform(otherData);
        int numRows = v.getRowDimension();
        // subMatrix indices are inclusive
        return otherData.multiply(v.getSubMatrix(0, numRows-1, 0, numComponents-1));
    }
}
```

Then to implement it, we use the following: `/* use the singular value decomposition implementation */ PCA pca = new PCA(data, new PCASVDImplementation());`

`/* get first three components */ RealMatrix pc3 = pca.getPrincipalComponents(3);`

`/* get however many components are needed to satisfy 50% explained variance */ RealMatrix pct = pca.getPrincipalComponents(.5);`

Creating Training, Validation, and Test Sets

For supervised learning, we build models on one part of the dataset, and then make a prediction using the test set and see whether we were right (using the known labels of the test set). Sometimes we need a third set during the training process for validating model parameters, called the *validation set*.

The training set is used to train the model, whereas the validation set is used for model selection. A test set is used once at the very end to calculate the model error. We have at least two options. First, we can sample random integers and pick lines out of an array or matrix. Second, we can reshuffle the data itself as a `List` and pull off the sublists of length we need for each type of set.

Index-Based Resampling

Create an index for each point in the dataset: `public class Resampler {`

`RealMatrix features; RealMatrix labels; List<Integer> indices; List<Integer> trainingIndices; List<Integer> validationIndices; List<Integer> testingIndices; int[] rowIndices; int[] test; int[] validate;`

`public Resampler(RealMatrix features, RealMatrix labels) { this.features = features; this.labels = labels; indices = new ArrayList<>(); }`

`public void calculateTestTrainSplit(double testFraction, long seed) { Random rnd = new Random(seed); for (int i = 1; i <= features.getRowDimension(); i++) { indices.add(i); } Collections.shuffle(indices, rnd); int testSize = new Long(Math.round(testFraction * features.getRowDimension())).intValue(); /* subList has inclusive fromIndex and exclusive toIndex */ testingIndices = indices.subList(0, testSize); trainingIndices = indices.subList(testSize, features.getRowDimension()); }`

`public RealMatrix getTrainingFeatures() { int numRows = trainingIndices.size(); rowIndices = new int[numRows]; int counter = 0; for (Integer trainingIndex : trainingIndices) { rowIndices[counter] = trainingIndex; } counter++;`

`int numCols = features.getColumnDimension(); int[] columnIndices = new int[numCols]; for (int i = 0; i < numCols; i++) { columnIndices[i] = i; } return features.getSubMatrix(rowIndices, columnIndices); } }`

Here is an example using the Iris dataset: `Iris iris = new Iris();`

`Resampler resampler = new Resampler(iris.getFeatures(), iris.getLabels()); resampler.calculateTestTrainSplit(0.40, 0L);`

`RealMatrix trainFeatures = resampler.getTrainingFeatures(); RealMatrix trainLabels = resampler.getTrainingLabels(); RealMatrix testFeatures = resampler.getTestingFeatures(); RealMatrix testLabels = resampler.getTestingLabels();`

List-Based Resampling

In some cases, we may have defined our data as a collection of objects.

For example, we may a have `List` of type `Record` that holds the data for each record (row) of data. It is straightforward

then to build a `List`-based resampler that takes a generic

type `T`:

```
public class Resampling<T> {


    private final List<T> data;
    private final int trainingSetSize;
    private final int testSetSize;
    private final int validationSetSize;

    public Resampling(List<T> data, double testFraction, long seed) {
        this(data, testFraction, 0.0, seed);
    }



    public Resampling(List<T> data, double testFraction,

    double validationFraction, long seed) {
        this.data = data;
        validationSetSize = new Double(
            validationFraction * data.size()).intValue();
        testSetSize = new Double(testFraction * data.size()).intValue();
        trainingSetSize = data.size() - (testSetSize + validationSetSize);
        Random rnd = new Random(seed);
        Collections.shuffle(data, rnd);
    }

    public int getTestSetSize() {
        return testSetSize;
    }

    public int getTrainingSetSize() {
        return trainingSetSize;
    }

    public int getValidationSetSize() {
        return validationSetSize;
    }



    public List<T> getValidationSet() {
        return data.subList(0, validationSetSize);
    }



    public List<T> getTestSet() {
        return data.subList(validationSetSize, validationSetSize + testSetSize);
    }



    public List<T> getTrainingSet() {
        return data.subList(validationSetSize + testSetSize, data.size());
    }
}
```

Given a predefined class `Record`, we can use the resampler like this:

```
Resampling<Record> resampling = new Resampling<>(data, 0.20, 0L);
//Resampling<Record> resampling = new Resampling<>(data, 0.20, 0.20, 0L);
List<Record> testSet = resampling.getTestSet();
List<Record> trainingSet = resampling.getTrainingSet();
List<Record> validationSet = resampling.getValidationSet();
```

Mini-Batches

In several learning algorithms, it is advantageous to input small batches of data (on the order of 100 data points) randomly sampled from a much larger dataset. We can

reuse the code from our `MatrixResampler` for this task. The

important thing to remember is that when designating batch size, we are

specifically implying the test set, not the training set, as implemented

in the `MatrixResampler`: `public class Batch extends MatrixResampler {`

`public Batch(RealMatrix features, RealMatrix labels) { super(features, labels); }`

`public void calcNextBatch(int batchSize) {`

`super.calculateTestTrainSplit(batchSize); }`

`public RealMatrix getInputBatch() { return super.getTestingFeatures(); }`

`public RealMatrix getTargetBatch() { return super.getTestingLabels(); } }`

Encoding Labels

When labels arrive to us as a text field, such as *red* or *blue*, we convert them to integers for further processing.

> **NOTE**
>
> When dealing with classification algorithms, we refer to each unique instance of the outcome's variables as a *class*. Recall that `class` is a Java keyword, and we will have to use other terms instead, such as `className`, `classLabel`, or `classes` for plural. When using `classes` for the name of a `List`, be aware of your IDE's code completion when building a for...each loop.

A Generic Encoder

Here is an implementation of a label encoder for a generic type `T`. Note that this system creates classes starting at 0 through *n* - 1 classes. In other words, the resulting class is the position in the `ArrayList`:

```
public class LabelEncoder<T> {


    private final List<T> classes;


    public LabelEncoder(T[] labels) {
        classes = Arrays.asList(labels);
    }

    public List<T> getClasses() {
        return classes;
    }

    public int encode(T label) {
        return classes.indexOf(label);
    }

    public T decode(int index) {
        return classes.get(index);
    }
}
```

Here is an example of how you might use label encoding with real data:

```
String[] stringLabels = {"Sunday", "Monday", "Tuesday"};


LabelEncoder<String> stringEncoder = new LabelEncoder<>(stringLabels);

/* note that classes are in the order of the original String array */
System.out.println(stringEncoder.getClasses()); //[Sunday, Monday, Tuesday]

for (Datum datum : data) {
    int classNumber = stringEncoder.encode(datum.getLabel);
    // do something with classes i.e. add to List or Matrix
}
```

Note that in addition to `String` types, this also works for any of the boxed types, but most likely your labels will take on values suitable for `Short`, `Integer`, `Long`, `Boolean`, and `Character`. For example, `Boolean` labels could be true/false bools, `Character` could be Y/N for yes/no or M/F for male/female or even T/F for true/false. It all depends on how someone else originally

coded the labels in the data file you are reading from. Labels are unlikely to be in the form of a floating-point number. If this is the case, you probably have a regression problem instead of a classification problem (that is, you are mistakenly confusing a continuous variable for a discrete one). An example using `Integer` type labels is shown in the next section.

One-Hot Encoding

In some cases, it will be more efficient to convert a multinomial label into a multivariate binomial. This is analogous to converting an integer to binary form, except that we have the requirement that only one position can be *hot* (equal to 1) at a time. For example, we can encode three string labels as integers, or represent each string as a position in a binary string:

```
Sunday  0  100

Monday  1  010

Tuesday 2  001
```

When using a `List` for encoding the labels, we use the following:

```
public class OneHotEncoder {


    private int numberOfClasses;

    public OneHotEncoder(int numberOfClasses) {
        this.numberOfClasses = numberOfClasses;
    }

    public int getNumberOfClasses() {
        return numberOfClasses;
    }

    public int[] encode(int label) {
        int[] oneHot = new int[numberOfClasses];
        oneHot[label] = 1;
        return oneHot;
    }



    public int decode(int[] oneHot) {
        return Arrays.binarySearch(oneHot, 1);
    }
}
```

In the case where the labels are strings, first encode the labels into integers by using a `LabelEncoder` instance, and then convert the integer labels to one hot by using a `OneHotEncoder` instance.

```
String[] stringLabels = {"Sunday", "Monday", "Tuesday"};

LabelEncoder<String> stringEncoder = new LabelEncoder<>(stringLabels);
```

```
int numClasses = stringEncoder.getClasses.size();

OneHotEncoder oneHotEncoder = new oneHotEncoder(numClasses);

for (Datum datum : data) {
    int classNumber = stringEncoder.encode(datum.getLabel);
    int[] oneHot    = oneHotEncoder.encode(classNumber);
    // do something with classes i.e. add to List or Matrix


}
```

Then what about the reverse? Say we have a predictive model that returns the classes we have designated in a learning process. (Usually, a learning process outputs probabilities, but we can assume that we have converted those to classes here.) First we need to convert the one-hot output to its class. Then we need to convert the class back to the original label, as shown here:

```
[1, 0, 0]


[0, 0, 1]

[1, 0, 0]

[0, 1, 0]
```

Then we need to convert output predictions from one hot: `for(Integer[] prediction: predictions) { int classLabel = oneHotEncoder.decode(prediction); String label = labelEncoder.decode(classLabel);`

`}`

`// predicted labels are Sunday, Tuesday, Sunday, Monday`

Chapter 5. Learning and Prediction

In this chapter, we'll learn what our data means and how it drives our decision processes. Learning about our data gives us knowledge, and knowledge enables us to make reasonable guesses about what to expect in the future. This is the reason for the existence of data science: learning enough about the data so we can make predictions on newly arriving data. This can be as simple as categorizing data into groups or clusters. It can span a much broader set of processes that culminate (ultimately) in the path to artificial intelligence. Learning is divided into two major categories: unsupervised and supervised.

In general, we think of data as having variates **X** and responses **Y**, and our goal is to build a model using **X** so that we can predict what happens when we put in a new **X**. If we have the **Y**, we can "supervise" the building of the model. In many cases, we have only the variates **X**. The model will then have to be built in an unsupervised manner. Typical unsupervised methods include clustering, whereas supervised learning may include any of the regression methods (e.g., linear regression) or classification methods such as naive Bayes, logistic, or deep neural net classifiers. Many other methods and permutations of those methods exist, and covering them all would be impossible. Instead, here we dive into a few of the most useful ones.

Learning Algorithms

A few learning algorithms are prevalent in a large variety of techniques. In particular, we often use an iterative learning process to repeatedly optimize or update the model parameters we are searching for. Several methods are available for optimizing the parameters, and we cover the gradient descent method here.

Iterative Learning Procedure

One standard way to learn a model is to loop over a prediction state and update the state. Regression, clustering and expectation-maximization (EM) algorithms all benefit from similar forms of an iterative learning procedure. Our strategy here is to create a class that contains all the boilerplate iterative machinery, and then allow subclasses to define the explicit form of the prediction and parameter update methods.

```
public class IterativeLearningProcess {

    private boolean isConverged;
    private int numIterations;
    private int maxIterations;
    private double loss;
    private double tolerance;
    private int batchSize; // if == 0 then uses ALL data
    private LossFunction lossFunction;

    public IterativeLearningProcess(LossFunction lossFunction) {
        this.lossFunction = lossFunction;
        loss = 0;
        isConverged = false;
        numIterations = 0;
        maxIterations = 200;
        tolerance = 10E-6;
        batchSize = 100;
    }

    public void learn(RealMatrix input, RealMatrix target) {
        double priorLoss = tolerance;
        numIterations = 0;
        loss = 0;
        isConverged = false;
        Batch batch = new Batch(input, target);
        RealMatrix inputBatch;
        RealMatrix targetBatch;
        while(numIterations < maxIterations && !isConverged) {
            if(batchSize > 0 && batchSize < input.getRowDimension()) {
                batch.calcNextBatch(batchSize);
                inputBatch = batch.getInputBatch();
                targetBatch = batch.getTargetBatch();
            } else {
                inputBatch = input;
                targetBatch = target;
            }
            RealMatrix outputBatch = predict(inputBatch);
            loss = lossFunction.getMeanLoss(outputBatch, targetBatch);
            if(Math.abs(priorLoss - loss) < tolerance) {
                isConverged = true;
            } else {
                update(inputBatch, targetBatch, outputBatch);
                priorLoss = loss;
            }
            numIterations++;
        }
    }

    public RealMatrix predict(RealMatrix input) {
        throw new UnsupportedOperationException("Implement the predict method!");
    }

    public void update(RealMatrix input, RealMatrix target, RealMatrix output) {
        throw new UnsupportedOperationException("Implement the update method!");
    }

}
```

Gradient Descent Optimizer

One way to learn parameters is via a gradient descent (an iterative first-order optimization algorithm). This optimizes the parameters by incrementally updating them with corrective learning (the error is used). The term *stochastic* means that we add one point at a time, as opposed to using the whole batch of data at once. In practice, it helps to use a mini-batch of about 100 points at a time, chosen at random in each step of the iterative learning process. The general idea is to minimize a loss function such that parameter updates are given by the following:

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

The parameter update is related to the gradient of an objective function $f(\theta)$ such that

$$\Delta\theta \propto \nabla f(\theta)$$

For deep networks, we will need to back-propagate this error through the network. We cover this in detail in "Deep Networks".

For the purposes of this chapter, we can define an interface that returns a parameter update, given a particular gradient. Method signatures for both matrix and vector forms are included:

```
public interface Optimizer {
    RealMatrix getWeightUpdate(RealMatrix weightGradient);
    RealVector getBiasUpdate(RealVector biasGradient);
}
```

The most common case of gradient descent is to subtract the scaled gradient from the existing parameter such that

$$\Delta\theta_t = -\eta\nabla f(\theta)_t$$

The update rule is as follows:

$$\theta_{t+1} = \theta_t - \eta\nabla f(\theta)_t$$

The most common type of stochastic gradient descent (SGD) is adding the update to the current parameters by using a learning rate:

```
public class GradientDescent implements Optimizer {

    private double learningRate;

    public GradientDescent(double learningRate) {
        this.learningRate = learningRate;
    }

    @Override
    public RealMatrix getWeightUpdate(RealMatrix weightGradient) {
        return weightGradient.scalarMultiply(-1.0 * learningRate);
```

```
    }

    @Override
    public RealVector getBiasUpdate(RealVector biasGradient) {
        return biasGradient.mapMultiply(-1.0 * learningRate);
    }
}
```

One common extension to this optimizer is the inclusion of momentum, which slows the process as the optimum is reached, avoiding an overshoot of the correct parameters:

$$\Delta\theta_t = \rho\Delta\theta_{t-1} - \eta\nabla f(\theta)_t$$

The update rule is as follows:

$$\theta_{t+1} = \theta_t + \rho\Delta\theta_{t-1} - \eta\nabla f(\theta)_t$$

We see that adding momentum is easily accomplished by extending the `GradientDescent` class, making provisions for storing the most recent update to the weights and bias for calculation of the next update. Note that the first time around, no prior updates will be stored yet, so a new set is created (and initialized to zero):

```
public class GradientDescentMomentum extends GradientDescent {

    private final double momentum;
    private RealMatrix priorWeightUpdate;
    private RealVector priorBiasUpdate;

    public GradientDescentMomentum(double learningRate, double momentum) {
        super(learningRate);
        this.momentum = momentum;
        priorWeightUpdate = null;
        priorBiasUpdate = null;
    }

    @Override
    public RealMatrix getWeightUpdate(RealMatrix weightGradient) {
        // creates matrix of zeros same size as gradients if
        // one does not already exist
        if(priorWeightUpdate == null) {
            priorWeightUpdate =
                new BlockRealMatrix(weightGradient.getRowDimension(),
                                    weightGradient.getColumnDimension());
        }
        RealMatrix update = priorWeightUpdate
                            .scalarMultiply(momentum)
                            .subtract(super.getWeightUpdate(weightGradient));
        priorWeightUpdate = update;
        return update;
    }

    @Override
    public RealVector getBiasUpdate(RealVector biasGradient) {
        if(priorBiasUpdate == null) {
            priorBiasUpdate = new ArrayRealVector(biasGradient.getDimension());
        }
        RealVector update = priorBiasUpdate
                            .mapMultiply(momentum)
                            .subtract(super.getBiasUpdate(biasGradient));
        priorBiasUpdate = update;
        return update;
    }
}
```

This is an ongoing and active field. By using this methodology, it is easy to extend capabilities by using ADAM or ADADELTA algorithms, for example.

Evaluating Learning Processes

Iterative processes can operate indefinitely. We always designate a maximum number of iterations we will allow so that any process cannot just run away and compute forever. Typically, this is on the order of 10^3 to 10^6 iterations, but there's no rule. There is a way to stop the iterative process early if a certain criteria has been met. We call this *convergence*, and the idea is that our process has converged on an answer that appears to be a stable point in the computation (e.g., the free parameters are no longer changing in a large enough increment to warrant the continuation of the process). Of course, there is more than one way to do this. Although certain learning techniques lend themselves to specific convergence criteria, there is no universal method.

Minimizing a Loss Function

A *loss function* designates the loss between predicted and target outputs. It is also known as a *cost function* or *error term*. Given a singular input vector **x**, output vector **y**, and prediction vector $\hat{\mathbf{y}}$, the loss of the sample is denoted with $\mathscr{L}(\mathbf{y}, \hat{\mathbf{y}})$. The form of the loss function depends on the underlying statistical distribution of the output data. In most cases, the loss over *p*-dimensional output and prediction is the sum of the scalar losses per dimension:

$$\mathscr{L}(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{p} \mathscr{L}(y, \hat{y})$$

Because we often deal with batches of data, we then calculate the mean loss $\langle \mathscr{L}(\mathbf{y}, \hat{\mathbf{y}}) \rangle$ over the whole batch. When we talk about minimizing a loss function, we are minimizing the mean loss over the batch of data that was input into the learning algorithm. In many cases, we can use the gradient of the loss $\nabla \mathscr{L}(\mathbf{y}, \hat{\mathbf{y}})$ to apply corrective learning. Here the gradient of the loss with respect to the predicted value $\frac{\partial \mathscr{L}}{\partial \hat{y}_i}$ can usually be computed with ease. The idea is then to return a loss gradient that is the same shape as its input.

> **WARNING**
>
> In some texts, the output is denoted as t (for *truth* or *target*), and the prediction is denoted as y. In this text, we denote the output as y and prediction as $\hat{y}$. Note that y has different meanings in these two cases.

Many forms are dependent on the type of variables (continuous or discrete or both) and the underlying statistical distribution. However, a common theme makes using an interface ideal. A reason for leaving implementation up to a specific class is that it takes advantage of optimized algorithms for linear algebra routines.

```
public interface LossFunction {
    public double getSampleLoss(double predicted, double target);
    public double getSampleLoss(RealVector predicted, RealVector target);
    public double getMeanLoss(RealMatrix predicted, RealMatrix target);
    public double getSampleLossGradient(double predicted, double target);
    public RealVector getSampleLossGradient(RealVector predicted,
                                            RealVector target);
    public RealMatrix getLossGradient(RealMatrix predicted, RealMatrix target);
}
```

Linear loss

Also known as the *absolute loss*, the *linear loss* is the absolute difference between the output and the prediction:

$$\mathscr{L}(y, \hat{y}) = |\hat{y} - y|$$

The gradient is misleading because of the absolute-value signs, which cannot be ignored:

$$\frac{\partial \mathscr{L}(y, \hat{y})}{\partial \hat{y}} = \frac{\hat{y} - y}{|\hat{y} - y|}$$

The gradient is not defined at $\hat{y} - y = 0$ because $\mathscr{L}$ has a discontinuity there. However, we can programmatically designate the gradient function to set its value to 0 when the gradient is zero to avoid a 1/0 exception. In this way, the gradient function returns only a –1, 0, or 1. Ideally, we then use the mathematical function *sign(x)*, which returns only –1, 0, or 1, depending on the respective input values of $x < 0$, $x = 0$ and $x > 0$.

```
public class LinearLossFunction implements LossFunction {

    @Override
    public double getSampleLoss(double predicted, double target) {
        return Math.abs(predicted - target);
    }

    @Override
    public double getSampleLoss(RealVector predicted, RealVector target) {
        return predicted.getL1Distance(target);
    }

    @Override
    public double getMeanLoss(RealMatrix predicted, RealMatrix target) {
        SummaryStatistics stats = new SummaryStatistics();
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            double dist = getSampleLoss(predicted.getRowVector(i),
            target.getRowVector(i));
            stats.addValue(dist);
        }
        return stats.getMean();
    }

    @Override
    public double getSampleLossGradient(double predicted, double target) {
        return Math.signum(predicted - target); // -1, 0, 1
    }

    @Override
    public RealVector getSampleLossGradient(RealVector predicted,
        RealVector target) {
        return predicted.subtract(target).map(new Signum());
    }

    //YOUDO SparseToSignum would be nice!!! only process elements of the iterable
    @Override
    public RealMatrix getLossGradient(RealMatrix predicted, RealMatrix target) {
        RealMatrix loss = new Array2DRowRealMatrix(predicted.getRowDimension(),
        predicted.getColumnDimension());
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            loss.setRowVector(i, getSampleLossGradient(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return loss;
    }
```

```
}
```

Quadratic loss

A generalized form for computing the error of a predictive process is by minimizing a distance metric such as L1 or L2 over the entire dataset. For a particular prediction-target pair, the quadratic error is as follows:

$$\mathscr{L}(y, \hat{y}) = \frac{1}{2}(\hat{y} - y)^2$$

An element of the sample loss gradient is then as follows:

$$\frac{\partial \mathscr{L}}{\partial \hat{y}} = (\hat{y} - y)$$

An implementation of a quadratic loss function follows:

```
public class QuadraticLossFunction implements LossFunction {

    @Override
    public double getSampleLoss(double predicted, double target) {
        double diff = predicted - target;
        return 0.5 * diff * diff;
    }

    @Override
    public double getSampleLoss(RealVector predicted, RealVector target) {
        double dist = predicted.getDistance(target);
        return 0.5 * dist * dist;
    }

    @Override
    public double getMeanLoss(RealMatrix predicted, RealMatrix target) {
        SummaryStatistics stats = new SummaryStatistics();
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            double dist = getSampleLoss(predicted.getRowVector(i),
                                        target.getRowVector(i));
            stats.addValue(dist);
        }
        return stats.getMean();
    }

    @Override
    public double getSampleLossGradient(double predicted, double target) {
        return predicted - target;
    }

    @Override
    public RealVector getSampleLossGradient(RealVector predicted,
        RealVector target) {
        return predicted.subtract(target);
    }

    @Override
    public RealMatrix getLossGradient(RealMatrix predicted, RealMatrix target) {
        return predicted.subtract(target);
    }
}
```

Cross-entropy loss

Cross entropy is great for classification (e.g., logistics regression or neural nets). We discussed the origins of cross entropy in Chapter 3. Because cross entropy shows similarity between two samples, it can be used for measuring agreement between known and predicted values. In the case of learning algorithms, we equate p with the known value y, and q with the predicted value $\hat{y}$. We set the loss equal to the cross entropy $\mathcal{H}(p, q)$ such that $L(y, \hat{y}) = \mathcal{H}(p, q)$ where $y_{ik} = p_{ik}$ is the target (label) and $\hat{y}_{ik} = q_{ik}$ is the i-th predicted value for each class k in a K multiclass output. The cross entropy (the loss per sample) is then as follows:

$$\mathcal{L}(y, \hat{y}) = -\sum_{k}^{K} y_k \log(\hat{y}_k)$$

There are several common forms for cross entropy and its associated loss function.

Bernoulli

In the case of Bernoulli output variates, the known outputs y are binary, where the prediction probability is $\hat{y}$, giving a cross-entropy loss:

$$\mathcal{L}(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$$

The sample loss gradient is then as follows:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})}$$

Here is an implementation of the Bernoulli cross-entropy loss:

```
public class CrossEntropyLossFunction implements LossFunction {

    @Override
    public double getSampleLoss(double predicted, double target) {
        return -1.0 * (target * ((predicted>0)?FastMath.log(predicted):0)
            + (1.0 - target)*(predicted<1?FastMath.log(1.0-predicted):0));
    }

    @Override
    public double getSampleLoss(RealVector predicted, RealVector target) {
        double loss = 0.0;
        for (int i = 0; i < predicted.getDimension(); i++) {
            loss += getSampleLoss(predicted.getEntry(i), target.getEntry(i));
        }
```

```
        return loss;
    }

    @Override
    public double getMeanLoss(RealMatrix predicted, RealMatrix target) {
        SummaryStatistics stats = new SummaryStatistics();
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            stats.addValue(getSampleLoss(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return stats.getMean();
    }

    @Override
    public double getSampleLossGradient(double predicted, double target) {
        // NOTE this blows up if predicted = 0 or 1, which it should never be
        return (predicted - target) / (predicted * (1 - predicted));
    }

    @Override
    public RealVector getSampleLossGradient(RealVector predicted,
                                            RealVector target) {
        RealVector loss = new ArrayRealVector(predicted.getDimension());
        for (int i = 0; i < predicted.getDimension(); i++) {
            loss.setEntry(i, getSampleLossGradient(predicted.getEntry(i),
            target.getEntry(i)));
        }
        return loss;
    }

    @Override
    public RealMatrix getLossGradient(RealMatrix predicted, RealMatrix target) {
        RealMatrix loss = new Array2DRowRealMatrix(predicted.getRowDimension(),
        predicted.getColumnDimension());
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            loss.setRowVector(i, getSampleLossGradient(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return loss;
    }
}
```

This expression is most often used with the logistic output function.

Multinomial

When the output is multiclass ($k = 0,1,2 \ldots K-1$) and transformed to a set of binary outputs via one-hot-encoding, the cross entropy loss is the sum over all possible classes:

$$\mathscr{L}(y, \hat{y}) = -\sum_{k} y_k \log(\hat{y}_k)$$

However, in one-hot encoding, only one dimension has $y = 1$ and the rest are $y = 0$ (a sparse matrix). Therefore, the sample loss is also a sparse matrix. Ideally, we could simplify this calculation by taking that into account.

The sample loss gradient is as follows:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = -\frac{y}{\hat{y}}$$

Because most of the loss matrix will be zeros, we need to calculate the gradient only for locations where $y = 1$. This form is used primarily with the softmax output function:

```
public class OneHotCrossEntropyLossFunction implements LossFunction {

    @Override
    public double getSampleLoss(double predicted, double target) {
        return predicted > 0 ? -1.0 * target * FastMath.log(predicted) : 0;
    }

    @Override
    public double getSampleLoss(RealVector predicted, RealVector target) {
        double sampleLoss = 0.0;
        for (int i = 0; i < predicted.getDimension(); i++) {
            sampleLoss += getSampleLoss(predicted.getEntry(i),
                                        target.getEntry(i));
        }
        return sampleLoss;
    }

    @Override
    public double getMeanLoss(RealMatrix predicted, RealMatrix target) {
        SummaryStatistics stats = new SummaryStatistics();
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            stats.addValue(getSampleLoss(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return stats.getMean();
    }

    @Override
    public double getSampleLossGradient(double predicted, double target) {
        return -1.0 * target / predicted;
    }

    @Override
    public RealVector getSampleLossGradient(RealVector predicted,
                                            RealVector target) {
        return target.ebeDivide(predicted).mapMultiplyToSelf(-1.0);
    }

    @Override
    public RealMatrix getLossGradient(RealMatrix predicted, RealMatrix target) {
        RealMatrix loss = new Array2DRowRealMatrix(predicted.getRowDimension(),
        predicted.getColumnDimension());
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            loss.setRowVector(i, getSampleLossGradient(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return loss;
    }
}
```

Two-Point

When the output is binary but takes on the values of –1 and 1 instead of 0 and 1, we can rescale for use with the Bernoulli expression with the substitutions $y^* = (y + 1)/2$ and $\hat{y}^* = (\hat{y} + 1)/2$:

$$\mathscr{L}(y^*, \hat{y}^*) = -\left(\left(\frac{y+1}{2}\right)\log\left(\frac{\hat{y}+1}{2}\right) + \left(\frac{1-y}{2}\right)\log\left(\frac{1-\hat{y}}{2}\right)\right)$$

The sample loss gradient is as follows:

$$\frac{\partial \mathscr{L}}{\partial \hat{y}} = \frac{\hat{y} - y}{1 - \hat{y}^2}$$

The Java code is shown here:

```
public class TwoPointCrossEntropyLossFunction implements LossFunction {

    @Override
    public double getSampleLoss(double predicted, double target) {
        // convert -1:1 to 0:1 scale
        double y = 0.5 * (predicted + 1);
        double t = 0.5 * (target + 1);
        return -1.0 * (t * ((y>0)?FastMath.log(y):0) +
                (1.0 - t)*(y<1?FastMath.log(1.0-y):0));
    }

    @Override
    public double getSampleLoss(RealVector predicted, RealVector target) {
        double loss = 0.0;
        for (int i = 0; i < predicted.getDimension(); i++) {
            loss += getSampleLoss(predicted.getEntry(i), target.getEntry(i));
        }
        return loss;
    }

    @Override
    public double getMeanLoss(RealMatrix predicted, RealMatrix target) {
        SummaryStatistics stats = new SummaryStatistics();
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            stats.addValue(getSampleLoss(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return stats.getMean();
    }

    @Override
    public double getSampleLossGradient(double predicted, double target) {
        return (predicted - target) / (1 - predicted * predicted);
    }

    @Override
    public RealVector getSampleLossGradient(RealVector predicted,
                                            RealVector target) {
        RealVector loss = new ArrayRealVector(predicted.getDimension());
        for (int i = 0; i < predicted.getDimension(); i++) {
            loss.setEntry(i, getSampleLossGradient(predicted.getEntry(i),
            target.getEntry(i)));
        }
        return loss;
    }

    @Override
    public RealMatrix getLossGradient(RealMatrix predicted, RealMatrix target) {
        RealMatrix loss = new Array2DRowRealMatrix(predicted.getRowDimension(),
        predicted.getColumnDimension());
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            loss.setRowVector(i, getSampleLossGradient(predicted.getRowVector(i),
            target.getRowVector(i)));
```

```
            }
            return loss;
        }
    }
```

This form of loss is compatible with a tanh activation function.

Minimizing the Sum of Variances

When data is split into more than one group, we can monitor the spread of the group from its mean position via the variance. Because variances add, we can define a metric *s* over *n* groups, where σ_i^2 is the variance of each group:

$$s = \sum_{i=1}^{n} \sigma_i^2$$

As *s* decreases, it signifies that the overall error of the procedure is also decreasing. This works great for clustering techniques, such as *k*-means, which are based on finding the mean value or center point of each cluster.

Silhouette Coefficient

In unsupervised learning techniques such as clustering, we seek to discover how closely packed each group of points is. The *silhouette coefficient* is a metric that relates the difference between the minimum distance inside any given cluster and its nearest cluster. The silhouette coefficient, *s*, is the average over all distances s_i for each sample; *a* = the mean distance between that sample and all other points in the class, and *b* = the mean distance between that sample and all the points in the next nearest cluster:

$$s_i = \frac{b_i - a_i}{max(a_i, b_i)}$$

Then the silhouette score is the mean of all the sample silhouette coefficients:

$$s = \frac{1}{n}\sum_{i}^{n} s_i$$

The silhouette score is between –1 and 1, where –1 is incorrect clustering, 1 is highly dense clustering, and 0 indicates overlapping clusters. *s* increases as clusters are dense and well separated. The goal is in monitor processes for a maximal value of *s*. Note that the silhouette coefficient is defined only for $2 <= n_{labels} <= n_{samples} - 1$. Here is the Java code:

```
public class SilhouetteCoefficient {

    List<Cluster<DoublePoint>> clusters;
    double coefficient;
    int numClusters;
    int numSamples;

    public SilhouetteCoefficient(List<Cluster<DoublePoint>> clusters) {
        this.clusters = clusters;
        calculateMeanCoefficient();
    }

    private void calculateMeanCoefficient() {
        SummaryStatistics stats = new SummaryStatistics();
        int clusterNumber = 0;
        for (Cluster<DoublePoint> cluster : clusters) {
            for (DoublePoint point : cluster.getPoints()) {
                double s = calculateCoefficientForOnePoint(point, clusterNumber);
                stats.addValue(s);
            }
            clusterNumber++;
        }
        coefficient = stats.getMean();
    }

    private double calculateCoefficientForOnePoint(DoublePoint onePoint,
    int clusterLabel) {

        /* all other points will compared to this one */
        RealVector vector = new ArrayRealVector(onePoint.getPoint());
```

```
        double a = 0;
        double b = Double.MAX_VALUE;
        int clusterNumber = 0;
        for (Cluster<DoublePoint> cluster : clusters) {
            SummaryStatistics clusterStats = new SummaryStatistics();
            for (DoublePoint otherPoint : cluster.getPoints()) {
                RealVector otherVector =
                    new ArrayRealVector(otherPoint.getPoint());
                double dist = vector.getDistance(otherVector);
                clusterStats.addValue(dist);
            }
            double avgDistance = clusterStats.getMean();
            if(clusterNumber==clusterLabel) {
                /* we have included a 0 distance of point with itself */
                /* and need to subtract it out of the mean */
                double n = new Long(clusterStats.getN()).doubleValue();
                double correction = n / (n - 1.0);
                a = correction * avgDistance;
            } else {
                b = Math.min(avgDistance, b);
            }
            clusterNumber++;
        }
        return (b-a) / Math.max(a, b);
    }
}
```

Log-Likelihood

In unsupervised learning problems for which each outcome prediction has a probability associated with it, we can utilize the log-likelihood. One particular example is the Gaussian clustering example in this chapter. For this expectation-maximization algorithm, a mixture of multivariate normal distributions are optimized to fit the data. Each data point has a probability density p_i associated with it, given the overlying model, and the log-likelihood can be computed as the mean of the log of the probabilities for each point:

$$\mathscr{L}(\mathbf{p}) = \sum_i \log\left(p_i\right)$$

We can then accumulate the average log-likelihood over all the data points $\langle L(\mathbf{p}) \rangle$. In the case of the Gaussian clustering example, we can obtain this parameter directly via the `MultivariateNormalMixtureExpectationMaximization.getLogLikelihood()` method.

Classifier Accuracy

How do we know how accurate a classifier really is? A binary classification scheme has four possible outcomes:

1. true positive (TP) — both data and prediction have the value of 1
2. true negative (TN) — both data and prediction have a value of 0
3. false positive (FP) — the data is 0 and prediction is 1
4. false negative (FN) — the data is 1 and the prediction is 0

Given a tally of each of the four possible outcomes, we can calculate, among other things, the accuracy of the classifier.

Accuracy is calculated as follows:

$$accuracy = \frac{tp + tn}{tp + tn + fp + fn}$$

Or, considering that the denominator is the total number of rows in the dataset *N*, the expression is equivalent to the following:

$$accuracy = \frac{tp + tn}{N}$$

We can then calculate the accuracy for each dimension. The average of the accuracy vector is the average accuracy of the classifier. This is also the Jaccard score.

In the special case that we are using one-hot-encoding we require only true positives and the accuracy per dimension is then as folllows:

$$accuracy = \frac{tp}{N_t}$$

N_t is the total class count (of 1s) for that dimension. The accuracy score for the classifier is then as follows:

$$accuracy = \frac{\sum tp}{N}$$

In this implementation, we have two use cases. In one case, there is one-hot encoding. In the other case, the binary, multilabel outputs are independent. In that case, we can choose a threshold (between 0 and 1) at which point to decide whether the class is 1 or 0. In the most basic sense, we can choose the threshold to be 0.5, where all probabilities below 0.5 are classified as 0, and probabilities greater than or equal to 0.5 are classified as 1. Examples of this class's use are in "Supervised Learning".

```
public class ClassifierAccuracy {

    RealMatrix predictions;
    RealMatrix targets;
    ProbabilityEncoder probabilityEncoder;
    RealVector classCount;

    public ClassifierAccuracy(RealMatrix predictions, RealMatrix targets) {
        this.predictions = predictions;
        this.targets = targets;
        probabilityEncoder = new ProbabilityEncoder();
        //tally the binary class occurrences per dimension
        classCount = new ArrayRealVector(targets.getColumnDimension());
        for (int i = 0; i < targets.getRowDimension(); i++) {
            classCount = classCount.add(targets.getRowVector(i));
        }
    }

    public RealVector getAccuracyPerDimension() {
        RealVector accuracy =
            new ArrayRealVector(predictions.getColumnDimension());
        for (int i = 0; i < predictions.getRowDimension(); i++) {
            RealVector binarized = probabilityEncoder.getOneHot(
                predictions.getRowVector(i));
            // 0*0, 0*1, 1*0 = 0 and ONLY 1*1 = 1 gives true positives
            RealVector decision = binarized.ebeMultiply(targets.getRowVector(i));
            // append TP counts to accuracy
            accuracy = accuracy.add(decision);
        }
        return accuracy.ebeDivide(classCount);
    }

    public double getAccuracy() {
        // convert accuracy_per_dim back to counts
        // then sum and divide by total rows
        return getAccuracyPerDimension().ebeMultiply(classCount).getL1Norm() /
        targets.getRowDimension();
    }

    // implements Jaccard similarity scores
    public RealVector getAccuracyPerDimension(double threshold) {
    // assumes un-correlated multi-output
        RealVector accuracy = new ArrayRealVector(targets.getColumnDimension());
        for (int i = 0; i < predictions.getRowDimension(); i++) {
            //binarize the row vector according to the threshold
            RealVector binarized = probabilityEncoder.getBinary(
            predictions.getRowVector(i), threshold);
            // 0-0 (TN) and 1-1 (TP) = 0 while 1-0 = 1 and 0-1 = -1
            RealVector decision = binarized.subtract(
            targets.getRowVector(i)).map(new Abs()).mapMultiply(-1).mapAdd(1);
            // append either TP and TN counts to accuracy
            accuracy = accuracy.add(decision);
        }
        return accuracy.mapDivide((double) predictions.getRowDimension());
        // accuracy for each dimension, given the threshold
    }
```

```
    public double getAccuracy(double threshold) {
        // mean of the accuracy vector
        return getAccuracyPerDimension(threshold).getL1Norm() /
            targets.getColumnDimension();
    }
}
```

Unsupervised Learning

When we have only independent variables, we must discern patterns in the data without the aid of dependent variables (responses) or labels. The most common of the unsupervised techniques is clustering. The goal of all clustering is to classify each data point **X** into a series of **K** sets, $S = S_1, \ldots S_K$, where the number of sets is less than the number of points. Typically, each point X_i will belong to only one subset S_k. However, we can also designate each point X_i to belong to all sets with a probability $p(X_i) = p_1, p_2, \ldots p_K$ such that the sum = 1. Here we explore two varieties of *hard assignment*, *k*-means and DBSCAN clustering; and a *soft assignment* type, mixture of Gaussians. They all vary widely in their assumptions, algorithms, and scope. However, the result is generally the same: to classify a point **X** into one or more subsets, or clusters.

k-Means Clustering

k-means is the simplest form of clustering and uses hard assignment to find the cluster centers for a predetermined number of clusters. Initially, an integer number of *K* clusters is chosen to start with, and the centroid location μ_k of each is chosen by an algorithm (or at random). A point *x* will belong to a cluster of set S_k if its Euclidean distance (can be others, but usually L2) is closest to μ_k. Then the objective function to minimize is as follows:

$$\mathcal{L} = \sum_{k=1}^{K} \sum_{\mathbf{x} \in S_k} \| \mathbf{x} - \boldsymbol{\mu}_\mathbf{k} \|^2$$

Then we update the new centroid (the mean position of all **x** in a cluster) via this equation:

$$\boldsymbol{\mu}_k = \frac{1}{N} \sum_{\mathbf{x} \in S_k} \mathbf{x}$$

We can stop when L does not change anymore, and therefore the centroids are not changing. How do we know what number of clusters is optimal? We can keep track of the sum of all cluster variances and vary the number of clusters. When plotting the sum-of-variances versus the number of clusters, ideally the shape will look like a hockey stick, with a sharp bend in the plot indicating the ideal number of clusters at the point.

$$\sigma_K^2 = \sum_{k=1}^{K} \frac{1}{N_k - 1} \sum_{\mathbf{x} \in S_k} \| \mathbf{x} - \boldsymbol{\mu}_\mathbf{k} \|^2$$

The algorithm used by Apache Commons Math is the *k*-means++, which does a better job of picking out random starting points. The class `KMeansPlusPlusClusterer<T>` takes several arguments in its constructor, but only one is required: the number of clusters to search for. The data to be clustered must be a `List` of `Clusterable` points. The class `DoublePoint` is a convenient wrapper around an array of doubles that implements `Clusterable`. It takes an array of doubles in its constructor.

```
double[][] rawData = ...
```

```
List<DoublePoint> data = new ArrayList<>();

for (double[] row : rawData) {
    data.add(new DoublePoint(row));
}

/* num clusters to search for */
int numClusters = 1;

/* the basic constructor */
KMeansPlusPlusClusterer<DoublePoint> kmpp =
    new KMeansPlusPlusClusterer<>(numClusters);

/* this performs the clustering and returns a list with length numClusters */
List<CentroidCluster<DoublePoint>> results = kmpp.cluster(data);

/* iterate the list of Clusterables */
for (CentroidCluster<DoublePoint> result : results) {

    DoublePoint centroid = (DoublePoint) result.getCenter();

    System.out.println(centroid); // DoublePoint has toString() method

    /* we also have access to all the points in only this cluster */
    List<DoublePoint> clusterPoints = result.getPoints();

}
```

In the *k*-means scheme, we want to iterate over several choices of `numClusters`, keeping track of the sum of variances for each cluster. Because variances add, this gives us a measure of total error. Ideally, we want to minimize this number. Here we keep track of the cluster variances as we iterate through various cluster searches:

```
/* search for 1 through 5 clusters */
for (int i = 1; i < 5; i++) {

    KMeansPlusPlusClusterer<DoublePoint> kmpp = new KMeansPlusPlusClusterer<>(i);
    List<CentroidCluster<DoublePoint>> results = kmpp.cluster(data);

    /* this is the sum of variances for this number of clusters */
    SumOfClusterVariances<DoublePoint> clusterVar =
    new SumOfClusterVariances<>(new EuclideanDistance());

    for (CentroidCluster<DoublePoint> result : results) {
        DoublePoint centroid = (DoublePoint) result.getCenter());
    }
}
```

One way we can improve the *k*-means is to try several starting points and take the best result — that is, lowest error. Because the starting points are random, at times the clustering algorithm takes a wrong turn, which even our strategies for handling empty clusters can't handle. It's a good idea to repeat each clustering attempt and choose the one with the best results. The class `MultiKMeansPlusPlusClusterer<T>` performs the same clustering operation `numTrials` times and uses only the best result. We can combine these with the previous code:

```
/* repeat each clustering trial 10 times and take the best */
int numTrials = 10;

/* search for 1 through 5 clusters */
for (int i = 1; i < 5; i++) {

    /* we still need to create a cluster instance ... */
    KMeansPlusPlusClusterer<DoublePoint> kmpp = new KMeansPlusPlusClusterer<>(i);

    /* ... and pass it to the constructor of the multi */
```

```
    MultiKMeansPlusPlusClusterer<DoublePoint> multiKMPP =
        new MultiKMeansPlusPlusClusterer<>(kmpp, numTrials);

    /* NOTE this clusters on multiKMPP NOT kmpp */
    List<CentroidCluster<DoublePoint>> results = multikKMPP.cluster(data);

    /* this is the sum of variances for this number of clusters */
    SumOfClusterVariances<DoublePoint> clusterVar =
        new SumOfClusterVariances<>(new EuclideanDistance());

    /* the sumOfVariance score for 'i' clusters */
    double score = clusterVar.score(results)

    /* the 'best' centroids */
    for (CentroidCluster<DoublePoint> result : results) {
        DoublePoint centroid = (DoublePoint) result.getCenter());
    }
}
```

DBSCAN

What if clusters have irregular shapes? What if clusters are intertwined? The DBSCAN (density-based spatial clustering of applications with noise) algorithm is ideal for finding hard-to-classify clusters. It does not assume the number of clusters, but rather optimizes itself to the number of clusters present. The only input parameters are the maximum radius of capture and the minimum number of points per cluster. It is implemented as follows:

```
/* constructor takes eps and minpoints */
double eps = 2.0;
int minPts = 3;
DBSCANClusterer clusterer = new DBSCANClusterer(eps, minPts);
List<Cluster<DoublePoint>> results = clusterer.cluster(data);
```

Note that unlike the previous *k*-means++, DBSCAN does not return a `CentroidCluster` type because the centroids of the irregularly shaped clusters may not be meaningful. Instead, you can access the clustered points directly and use them for further processing. But also note that if the algorithm cannot find any clusters, the `List<Cluster<T>>` instance will comprise an empty `List` with a size of 0:

```
if(results.isEmpty()) {
    System.out.println("No clusters were found");
} else {

    for (Cluster<DoublePoint> result : results) {
        /* each clusters points are in here */
        List<DoublePoint> points = result.getPoints();
        System.out.println(points.size());
        // TODO do something with the points in each cluster
    }
}
```

In this example, we have created four random multivariate (two-dimensional) normal clusters. Of note is that two of the clusters are close enough to be touching and could even be considered one angular-shaped cluster. This demonstrates a trade-off in the DBSCAN algorithm.

In this case, we need to set the radius of capture small enough ($\epsilon = 0.225$) to allow detection of the separate clusters, but there are outliers. A larger radius ($\epsilon = 0.8$) here would combine the two leftmost clusters into one, but there would be almost no outliers. As we decrease ϵ, we are enabling a finer resolution for cluster detection, but we also are increasing the likelihood of outliers. This may become less of an issue in higher-dimensional space in which clusters in close proximity to each other are less likely. An example of four Gaussian clusters that fit the DBSCAN algorithm is shown in Figure 5-1.

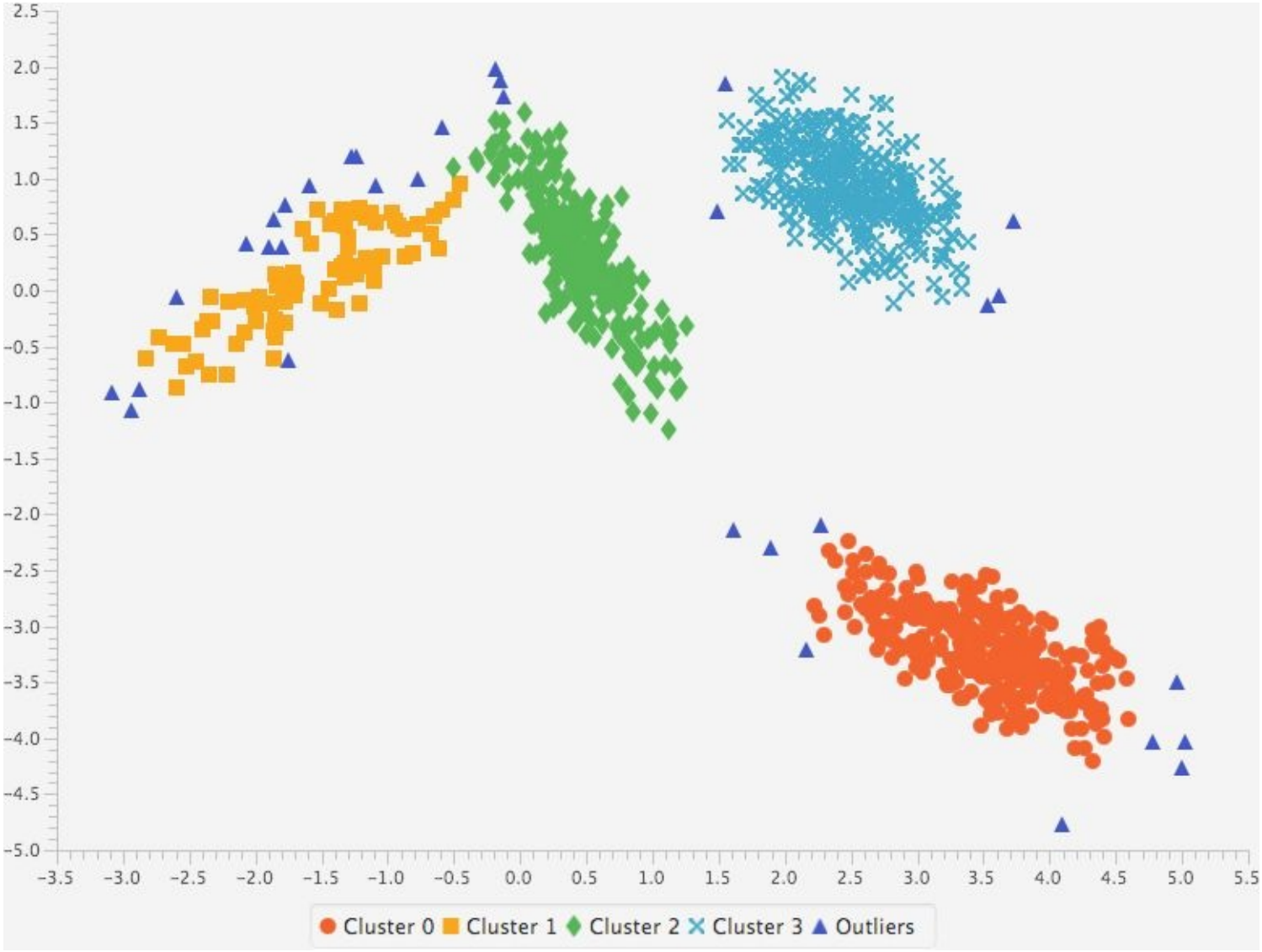

Figure 5-1. DBSCAN on simulation of four Gaussian clusters

Dealing with outliers

The DBSCAN algorithm is well suited for dealing with outliers. How do we access them? Unfortunately, the current Math implementation does not allow access to the points labeled as noise in the DBSCAN algorithm. But we can try to keep track of that like this:

```
/* we are going to keep track of outliers */
// NOTE need a completely new list, not to reference same object
// e.g., outliers = data is not a good idea

// List<DoublePoint> outliers = data; // will remove points from data as well

List<DoublePoint> outliers = new ArrayList<>();

for (DoublePoint dp : data) {
    outliers.add(new DoublePoint(dp.getPoint()));
}
```

Then when we are iterating through the results clusters, we can remove each cluster from the complete dataset data, which will become the outliers after we remove everything else:

```
for (Cluster<DoublePoint> result : results) {
```

```
    /* each clusters points are in here */
    List<DoublePoint> points = result.getPoints();


    /* remove these cluster points from the data copy "outliers"
       which will contain ONLY the outliers after all of the
       cluster points are removed
     */

    outliers.removeAll(points);
}

// now the LIST outliers only contains points NOT in any cluster
```

Optimizing radius of capture and minPoints

The radius of capture is easy to see in 2D, but how do you know what is optimal? Clearly, this is entirely subjective and will depend on your use case. In general, the number of minimum points should follow this relation:

$$n_{min} \geq p + 1$$

So in a 2D case, we at least want three minimum points per cluster. The radius of capture, ϵ, can be estimated at the bend in hockey stick of the *k*-distance graph. Both the number of minimum points and the radius of capture can be grid-searched against the silhouette score as a metric. First, find the silhouette coefficient, *s*, for each sample; *a* = the mean distance between that sample and all other points in the class, and *b* = the mean distance between that sample and all the points in the next nearest cluster:

$$s = \frac{b - a}{max(a, b)}$$

Then the silhouette score is the mean of all the sample silhouette coefficients. The silhouette score is between –1 and 1: –1 is incorrect clustering, 1 is highly dense clustering, and 0 indicates overlapping clusters. *s* increases as clusters are dense and well separated. As in the case for *k*-means previously, we can vary the ϵ value and output the silhouette score:

```
double[] epsVals = {0.15, 0.16, 0.17, 0.18, 0.19, 0.20,
                    0.21, 0.22, 0.23, 0.24, 0.25};

for (double epsVal : epsVals) {

    DBSCANClusterer clusterer = new DBSCANClusterer(epsVal, minPts);
    List<Cluster<DoublePoint>> results = clusterer.cluster(dbExam.clusterPoints);

    if(results.isEmpty()) {

        System.out.println("No clusters where found");

     } else {

        SilhouetteCoefficient s = new SilhouetteCoefficient(results);
        System.out.println("eps = " + epsVal +
                           " numClusters = " + results.size() +
                           " s = " + s.getCoefficient());
     }
}
```

This gives the following output:

```
eps = 0.15 numClusters = 7 s = 0.54765
eps = 0.16 numClusters = 7 s = 0.53424
eps = 0.17 numClusters = 7 s = 0.53311
eps = 0.18 numClusters = 6 s = 0.68734
eps = 0.19 numClusters = 6 s = 0.68342
eps = 0.20 numClusters = 6 s = 0.67743
eps = 0.21 numClusters = 5 s = 0.68348
eps = 0.22 numClusters = 4 s = 0.70073 // best one!
eps = 0.23 numClusters = 3 s = 0.68861
eps = 0.24 numClusters = 3 s = 0.68766
eps = 0.25 numClusters = 3 s = 0.68571
```

We see a bump in the silhouette score at $\epsilon = 0.22$, where $s = 0.7$, indicating that the ideal ϵ is approximately 0.22. At this particular ϵ, the DBSCAN routine also converged on four clusters, which is the number we simulated. In practical situations, of course, we won't know the number of clusters beforehand. But this example does indicate that *s* should approach a maximal value of 1 if we have the right number of clusters and therefore the right ϵ.

Inference from DBSCAN

DBSCAN is not for predicting membership of new points as in the *k*-means algorithm. It is for segmenting the data for further use. If you want a predictive model based on DBSCAN, you can assign class values to the clustered data points and try a classification scheme such as Gaussian, naive Bayes, or others.

Gaussian Mixtures

A similar concept to DBSCAN is to cluster based on the density of points, but use the multivariate normal distribution N(μ, Σ) because it comprises a mean and covariance. Data points located near the mean have the highest probability of belonging to that cluster, whereas the probability drops off to almost nothing as the data point is located very far from the mean.

Gaussian mixture model

A Gaussian mixture model is expressed mathematically as a weighted mixture of *k* multivariate Gaussian distributions (as discussed in Chapter 3).

$$f(\mathbf{x}) = \sum_{i=1}^{k} \alpha_i \mathcal{M}(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$$

Here the weights satisfy the relation $\sum_i^k \alpha_i = 1$. We must create a `List` of `Pair` objects, where the first member of `Pair` is the weight, and the second member is the distribution itself:

```
List<Pair<Double, MultivariateNormalDistribution>> mixture = new ArrayList<>();

/* mixture component 1 */
double alphaOne = 0.70;
double[] meansOne = {0.0, 0.0};
double[][] covOne = {{1.0, 0.0},{0.0, 1.0}};
MultivariateNormalDistribution distOne =
    new MultivariateNormalDistribution(meansOne, covOne);
Pair pairOne = new Pair(alphaOne, distOne);
mixture.add(pairOne);

/* mixture component 2 */
double alphaTwo = 0.30;
double[] meansTwo = {5.0, 5.0};
double[][] covTwo = {{1.0, 0.0},{0.0, 1.0}};
MultivariateNormalDistribution distTwo =
    new MultivariateNormalDistribution(meansTwo, covTwo);
Pair pairTwo = new Pair(alphaTwo, distTwo);
mixture.add(pairTwo);

/* add the list of pairs to the mixture model and sample the points */
MixtureMultivariateNormalDistribution dist =
    new MixtureMultivariateNormalDistribution(mixture);

/* we don't need a seed, but it helps if we want to recall the same data */
dist.reseedRandomGenerator(0L);

/* generate 1000 random data points from the mixture */
double[][] data = dist.sample(1000);
```

Note that the data sampled from the distribution mixture model does not keep track of what component the sampled data point comes from. In other words, you will not be able to tell what `MultivariateNormal` each sampled data point belongs to. If you require this feature, you can always sample from the individual distributions and then add them together later.

For purposes of testing, creating mixture models can be tedious and is fraught with problems. If you are

not building a dataset from existing, real data, it is best to try simulating data with some known problems averted. A method for generating random mixture-model, is presented in Appendix A. In Figure 5-2, a plot of a multivariate Gaussian mixture model is demonstrated. There are two clusters in two dimensions.

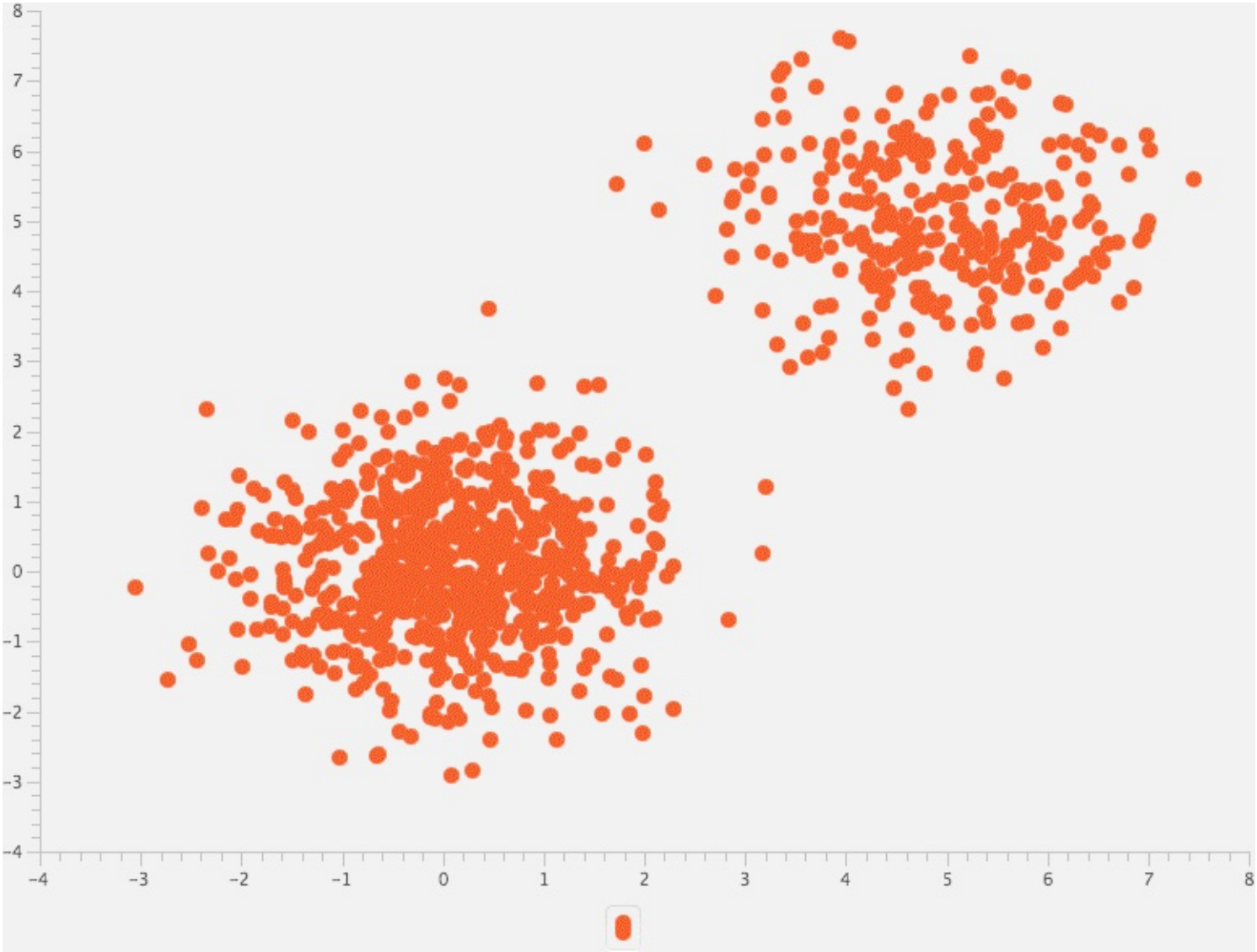

Figure 5-2. Gaussian clusters in 2D

The data can be generated with the example code:

```
int dimension = 5;
int numClusters = 7;
double boxSize = 10;
long seed = 0L;
int numPoints = 10000;

/* see Appendix for this dataset */
MultiNormalMixtureDataset mnd = new MultiNormalMixtureDataset(dimension);
mnd.createRandomMixtureModel(numClusters, boxSize, 0L);
double[][] data = mnd.getSimulatedData(numPoints);
```

Fitting with the EM algorithm

The expectation maximization algorithm is useful in many other places. Essentially, what is the maximum likelihood that the parameters we've chosen are correct? We iterate until they don't change anymore, given a certain tolerance. We need to provide a starting guess of what the mixture is. Using the method in

the preceding section, we can create a mixture with known components. However, the static method `MultivariateNormalMixtureExpectationMaximization.estimate(data, numClusters)` is used to estimate the starting point given the dataset and number of clusters as input:

```
MultivariateNormalMixtureExpectationMaximization mixEM =
    new MultivariateNormalMixtureExpectationMaximization(data);

/* need a guess as where to start */
MixtureMultivariateNormalDistribution initialMixture =
    MultivariateNormalMixtureExpectationMaximization.estimate(data, numClusters);

/* perform the fit */
mixEM.fit(initialMixture);

/* this is the fitted model */
MixtureMultivariateNormalDistribution fittedModel = mixEM.getFittedModel();

for (Pair<Double, MultivariateNormalDistribution> pair :
    fittedModel.getComponents()) {
    System.out.println("************ cluster *****************");
    System.out.println("alpha: " + pair.getFirst());
    System.out.println("means: " + new ArrayRealVector(
        pair.getSecond().getMeans()));
    System.out.println("covar: " + pair.getSecond().getCovariances());
}
```

Optimizing the number of clusters

Just as in *k*-means clustering, we would like to know the optimal number of clusters needed to describe our data. In this case, though, each data point belongs to all clusters with finite probability (soft assignment). How do we know when the number of clusters is good enough? We start with a low number (e.g., 2) and work our way up, calculating the log-likelihood for each trial. To make things easier, we can plot the loss (the negative of the log-likelihood), and watch as it hopefully drops toward zero. Realistically, it never will, but the idea is to stop when the loss becomes somewhat constant. Usually, the best number of clusters will be at the elbow of the hockey stick.

Here is the code:

```
MultivariateNormalMixtureExpectationMaximization mixEM =
    new MultivariateNormalMixtureExpectationMaximization(data);

int minNumClusters = 2;
int maxNumClusters = 10;

for(int i = minNumCluster; i <= maxNumClusters; i++) {

    /* need a guess as where to start */
    MixtureMultivariateNormalDistribution initialMixture =
        MultivariateNormalMixtureExpectationMaximization.estimate(data, i);

    /* perform the fit */
    mixEM.fit(initialMixture);

    /* this is the fitted model */
    MixtureMultivariateNormalDistribution fittedModel = mixEM.getFittedModel();

    /* print out the log-likelihood */
    System.out.println(i + " ll: " + mixEM.getLogLikelihood());
}
```

This outputs the following:

```
2 ll: -6.370643787350135
3 ll: -5.907864928786343
```

```
4 ll: -5.5789246749261014
5 ll: -5.366040927493185
6 ll: -5.093391683217386
7 ll: -5.1934910558216165
8 ll: -4.984837507547836
9 ll: -4.9817765145490664
10 ll: -4.981307556011888
```

When plotted, this shows a characteristic hockey-stick shape with the inflection point at `numClusters` = 7, the number of clusters we simulated. Note that we could have stored the loglikelihoods in an array, and fit the results in a `List` for later retrieval programatically. In Figure 5-3, the log-likelihood loss is plotted versus the number of clusters. Note the sharp decline in loss and the bend around seven clusters, the original number of clusters in the simulated dataset.

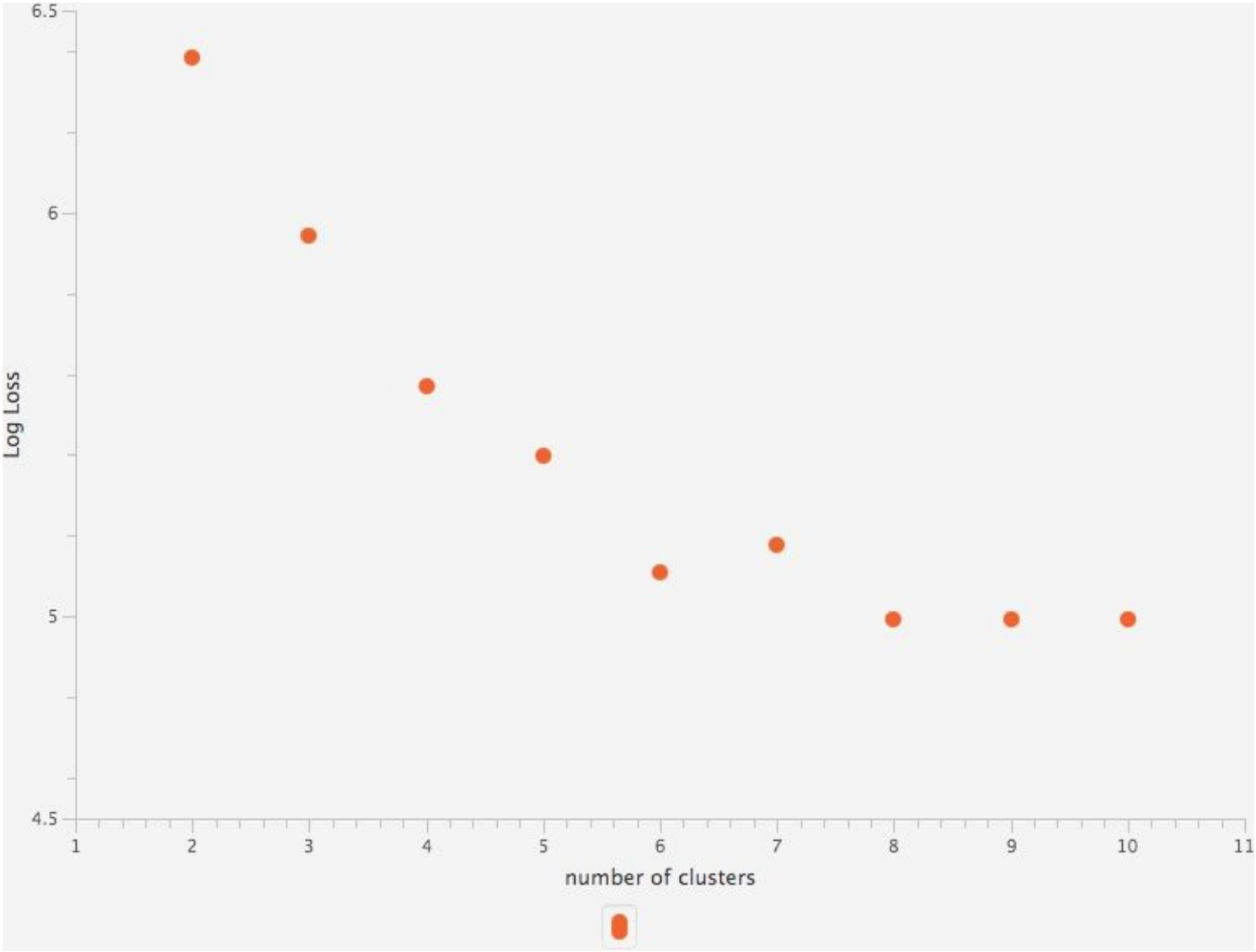

Figure 5-3. Log loss of 7, 5 dimensional clusters

Supervised Learning

Given numeric variates **X** and potentially non-numeric responses **Y**, how can we formulate mathematical models to learn and predict? Recall that linear regression models rely on both **X** and **Y** to be continuous variables (e.g., real numbers). Even when **Y** contains 0s or 1s, (and any other integers), a linear regression would most likely fail.

Here we examine methods specifically designed for the common use cases that collect numeric data as variates and their associated labels. Most classification schemes lend themselves easily to a multidimensional variate **X** and a single dimension of classes **Y**. However, several techniques, including neural networks, can handle multiple output classes **Y** in a way analogous to the multiresponse models of linear regression.

Naive Bayes

Naive Bayes is perhaps the most elementary of classification schemes and is a logical next step after clustering. Recall that in clustering, our goal is to separate or classify data into distinct groups. We can then look at each group individually and try to learn something about that group, such as its center position, its variance, or any other statistical measure.

In a naive Bayes classification scheme, we split the data into groups (classes) for each label type. We then learn something about the variates in each group. This will depend on the type of variable. For example, if the variables are real numbers, we can assume that each dimension (variate) of the data is a sample from a normal distribution.

For integer data (counts), we can assume a multinomial distribution. If the data is binary (0 or 1), we can assume a Bernoulli distributed dataset. In this way, we can estimate statistical quantities such as mean and variance for each of the datasets belonging to only the class it was labeled for. Note that unlike more sophisticated classification schemes, we never use the labels themselves in any computation or error propagation. They serve the purpose only of splitting our data into groups.

According to Bayes' theorem (posterior = prior × likelihood / evidence), the joint probability is the prior × likelihood. In our case, the evidence is the sum of joint probabilities over all classes. For a set of K classes, where $k = \{1, 2 \ldots K\}$, the probability of a particular class k given an input vector $\mathbf{x}$ is determined as follows:

$$p(k \mid \mathbf{x}) = \frac{p(k)p(\mathbf{x} \mid k)}{p(\mathbf{x})}$$

Here, the naive independence assumption allows us to express the likelihood as the product of probabilities for each dimension of the n-dimensional variate $\mathbf{x}$:

$$p(\mathbf{x} \mid k) = p(x_1 \mid k)p(x_2 \mid k) \cdots p(x_n \mid k)$$

This is expressed more compactly as follows:

$$p(\mathbf{x} \mid k) = \prod_{i=1}^{n} p(x_i \mid k)$$

The normalization is the sum over all terms in the numerator expressed:

$$p(\mathbf{x}) = \sum_{k=1}^{K} p(k)p(\mathbf{x} \mid k)$$

The probability of any class is the number of times it occurs divided by the total: $p(k) = n_k / N$. Here we take the product for each class *k* over each feature x_i. The form of $p(x_i \mid C_K)$, is probability density function we choose based on our assumptions of the data. In the following sections, we explore normal, multinomial, and Bernoulli distributions.

> **WARNING**
>
> Note that if any one calculation $p(x_i \mid k) = 0$, the entire expression will be $p(k \mid \mathbf{x}) = 0$. For some conditional probability models such as Gaussian or Bernoulli distributions, this will never be the case. But for a multinomial distribution, this can occur, so we include a small factor α to avoid this.

After we calculate posterior probabilities for each class, a Bayes classifier is then a decision rule on the posterior probabilities, where we take the maximum position as the most likely class:

$$\hat{k} = \underset{k \in \{1,2,\cdots K\}}{\arg\max}\, p(k \mid \mathbf{x})$$

We can use the same class for all types, because training the model relies on the types of quantities that are easily accumulated with `MultivariateSummaryStatistics` per class. We can then use a strategy pattern to implement whichever type of conditional probability we require and pass it directly into the constructor:

```
public class NaiveBayes {

    Map <Integer, MultivariateSummaryStatistics> statistics;
    ConditionalProbabilityEstimator conditionalProbabilityEstimator;
    int numberOfPoints; // total number of points the model was trained on

    public NaiveBayes(
        ConditionalProbabilityEstimator conditionalProbabilityEstimator) {
        statistics = new HashMap<>();
        this.conditionalProbabilityEstimator = conditionalProbabilityEstimator;
        numberOfPoints = 0;
    }

    public void learn(RealMatrix input, RealMatrix target) {
        // if numTargetCols == 1 then multiclass e.g. 0, 1, 2, 3
        // else one-hot e.g. 1000, 0100, 0010, 0001
        numberOfPoints += input.getRowDimension();
        for (int i = 0; i < input.getRowDimension(); i++) {
            double[] rowData = input.getRow(i);
```

```
            int label;
            if (target.getColumnDimension()==1) {
                label = new Double(target.getEntry(i, 0)).intValue();
            } else {
                label = target.getRowVector(i).getMaxIndex();
            }

            if(!statistics.containsKey(label)) {
                statistics.put(label, new MultivariateSummaryStatistics(
                rowData.length, true));
            }
            statistics.get(label).addValue(rowData);
        }
    }

    public RealMatrix predict(RealMatrix input) {

        int numRows = input.getRowDimension();
        int numCols = statistics.size();
        RealMatrix predictions = new Array2DRowRealMatrix(numRows, numCols);

        for (int i = 0; i < numRows; i++) {
            double[] rowData = input.getRow(i);
            double[] probs = new double[numCols];
            double sumProbs = 0;
            for (Map.Entry<Integer, MultivariateSummaryStatistics> entrySet :
            statistics.entrySet()) {

                Integer classNumber = entrySet.getKey();
                MultivariateSummaryStatistics mss = entrySet.getValue();

                /* prior prob n_k / N ie num points in class / total points */
                double prob = new Long(mss.getN()).doubleValue()/numberOfPoints;

                /* depends on type ... Gaussian, Multinomial, or Bernoulli */
                prob *= conditionalProbabilityEstimator.getProbability(mss,
                                                                       rowData);

                probs[classNumber] = prob;
                sumProbs += prob;
            }

            /* L1 norm the probs */
            for (int j = 0; j < numCols; j++) {
                probs[j] /= sumProbs;
            }
            predictions.setRow(i, probs);
        }
        return predictions;
    }
}
```

All that is needed then is an interface that designates the form of the conditional probability:

```
public interface ConditionalProbabilityEstimator {
    public double getProbability(MultivariateSummaryStatistics mss,
                                 double[] features);
}
```

In the following three subsections, we explore three kinds of naive Bayes classifiers, each of which implements the `ConditionalProbabilityEstimator` interface for use in the `NaiveBayes` class.

Gaussian

If the features are continuous variables, we can use the Gaussian naive Bayes classifier:

$$p(\mathbf{x} \mid k) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma_{ki}}} \exp\left(-\frac{(x_i - \mu_{ki})^2}{2\sigma_{ki}^2}\right)$$

We can then implement a class like this:

```
import org.apache.commons.math3.distribution.NormalDistribution;
import org.apache.commons.math3.stat.descriptive.MultivariateSummaryStatistics;

public class GaussianConditionalProbabilityEstimator
implements ConditionalProbabilityEstimator{

    @Override
    public double getProbability(MultivariateSummaryStatistics mss,
                                 double[] features) {
        double[] means = mss.getMean();
        double[] stds = mss.getStandardDeviation();
        double prob = 1.0;
        for (int i = 0; i < features.length; i++) {
            prob *= new NormalDistribution(means[i], stds[i])
                .density(features[i]);
        }
        return prob;
    }

}
```

And test it like this:

```
double[][] features = {{6, 180, 12},{5.92, 190, 11}, {5.58, 170, 12},
                       {5.92, 165, 10}, {5, 100, 6}, {5.5, 150, 8},
                       {5.42, 130, 7}, {5.75, 150, 9}};
String[] labels = {"male", "male", "male", "male",
                   "female", "female", "female", "female"};
NaiveBayes nb = new NaiveBayes(new GaussianConditionalProbabilityEstimator());
nb.train(features, labels);

double[] test = {6, 130, 8};
String inference = nb.inference(test); // "female"
```

This will yield the correct result, `female`.

Multinomial

Features are integer values — for example, counts. However, continuous features such as TFIDF also can work. The likelihood of observing any feature vector **x** for class k is as follows:

$$p(\mathbf{x} \mid k) = \frac{(\sum_{i=1}^{n} x_i)!}{\prod_{i=1}^{n} x_i!} \prod_{i=1}^{n} p_{ki}^{x_i}$$

We note that the front part of the term depends only on the input vector **x**, and therefore is equivalent for each calculation of $p(\mathbf{x}|k)$. Fortunately, this computationally intense term will drop out in the final, normalized expression for $p(k|\mathbf{x})$, allowing us to use the much simpler formulation:

$$p(\mathbf{x} \mid k) = \prod_{i=1}^{n} p_{ki}^{x_i}$$

We can easily calculate the required probability $p_{ki} = N_{ik} / N_k$ where $N_i k$ is the sum of values for each feature, given class k, and N_k is the total count for all features, given class k. When estimating the conditional probabilities, any zeros will cancel out the entire calculation. It is therefore useful to estimate the probabilities with a small additive factor α — known as *Lidstone smoothing* for generalized α, and *Laplace smoothing* when $\alpha = 1$. As a result of this L1 normalization of the numerator, the factor n is just the dimension of the feature vector.

$$p_{ki} = \frac{N_{ik} + \alpha}{N_k + \alpha n}$$

The final expression is as follows:

$$p(\mathbf{x} \mid k) = \prod_{i=1}^{n} \left(\frac{N_{ik} + \alpha}{N_k + \alpha n} \right)^{x_i}$$

For large x_i (large counts of words, for example), the problem may become numerically intractable. We can solve the problem in log space and convert it back by using the relation $z = \exp(\ln(z))$. The preceding expression can be written as follows:

$$p(\mathbf{x} \mid k) = \exp\left(\sum_{i=1}^{n} x_i \ln \left(\frac{N_{ik} + \alpha}{N_k + \alpha n} \right) \right)$$

In this strategy implementation, the smoothing coefficient is designated in the constructor. Note the use of the logarithmic implementation to avoid numerical instability. It would be wise to add an assertion (in the constructor) that the smoothing constant alpha hold the relation $0 > \alpha \geq 1$:

```
public class MultinomialConditionalProbabilityEstimator
        implements ConditionalProbabilityEstimator {

    private double alpha;

    public MultinomialConditionalProbabilityEstimator(double alpha) {
        this.alpha = alpha; // Lidstone smoothing 0 > alpha > 1
    }

    public MultinomialConditionalProbabilityEstimator() {
        this(1); // Laplace smoothing
    }

    @Override
    public double getProbability(MultivariateSummaryStatistics mss,
                                 double[] features) {
        int n = features.length;
        double prob = 0;
        double[] sum = mss.getSum(); // array of x_i sums for this class
        double total = 0.0; // total count of all features
        for (int i = 0; i < n; i++) {
            total += sum[i];
        }
        for (int i = 0; i < n; i++) {
            prob += features[i] * Math.log((sum[i] + alpha) /(total+alpha*n));
        }
        return Math.exp(prob);
    }

}
```

Bernoulli

Features are binary values — for example, occupancy status. The probability per feature is the mean value for that column. For an input feature, we can then calculate the probability:

$$p(\mathbf{x} \mid k) = \prod_{i=1}^{n} (p_{ki}x_i + (1 - p_{ki})(1 - x_i))$$

In other words, if the input feature is a 1, the probability for that feature is the mean value for the column. If the input feature is a 0, the probability for that feature is 1 – mean of that column. We implement the Bernoulli conditional probability as shown here:

```
public class BernoulliConditionalProbabilityEstimator
    implements ConditionalProbabilityEstimator {

    @Override
    public double getProbability(MultivariateSummaryStatistics mss,
                                 double[] features) {
        int n = features.length;
        double[] means = mss.getMean();
        // this is actually the prob per features e.g. count / total
        double prob = 1.0;
        for (int i = 0; i < n; i++) {
            // if x_i = 1, then p, if x_i = 0 then 1-p, but here x_i is a double
            prob *= (features[i] > 0.0) ? means[i] : 1-means[i];
        }
        return prob;
    }
}
```

Iris example

Try the Iris dataset by using a Gaussian conditional probability estimator:

```
Iris iris = new Iris();
MatrixResampler mr = new MatrixResampler(iris.getFeatures(), iris.getLabels());
mr.calculateTestTrainSplit(0.4, 0L);

NaiveBayes nb = new NaiveBayes(new GaussianConditionalProbabilityEstimator());
nb.learn(mr.getTrainingFeatures(), mr.getTrainingLabels());

RealMatrix predictions = nb.predict(mr.getTestingFeatures());

ClassifierAccuracy acc = new ClassifierAccuracy(predictions,
                                                mr.getTestingLabels());
System.out.println(acc.getAccuracyPerDimension()); // {1; 1; 0.9642857143}
System.out.println(acc.getAccuracy()); // 0.9833333333333333
```

Linear Models

If we rotate, translate, and scale a dataset **X**, can we relate it to the output **Y** by mapping a function? In general, these all seek to solve the problem in which an input matrix **X** is the data, and **W** and **b** are the free parameters we want to optimize for. Using the notation developed in Chapter 2, for a weighted input matrix and intercept $\mathbf{Z} = \mathbf{XW} + \mathbf{hb}^T$, we apply a function $\varphi(\mathbf{Z})$ to each element of **Z** to compute a prediction matrix $\hat{\mathbf{Y}}$ such that

$$\hat{\mathbf{Y}} = \varphi(\mathbf{XW} + \mathbf{hb}^T)$$

We can view a linear model as a box with input **X** and predicted output $\hat{\mathbf{Y}}$. When optimizing the free parameters **W** and **b**, the error on the output can be sent back through the box, providing incremental updates dependent on the algorithm chosen. Of note is that we can even pass the error back to the input, calculating the error on the input. For linear models, this is not necessary, but as we will see in "Deep Networks", this is essential for the back-propagation algorithm. A generalized linear model is shown in Figure 5-4.

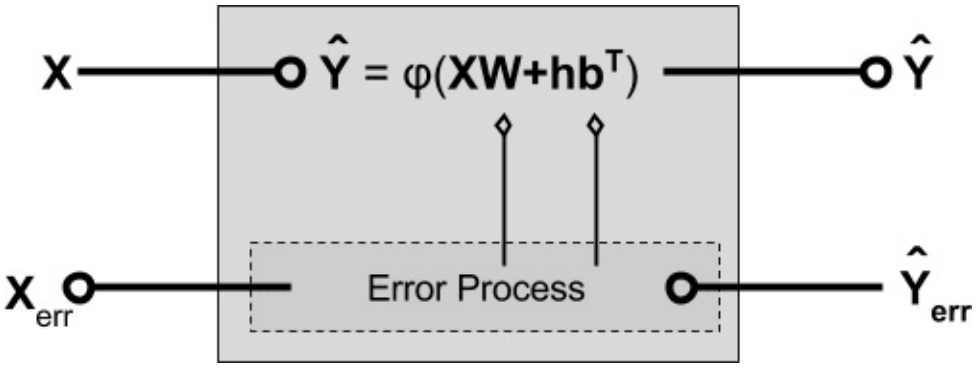

Figure 5-4. Linear model

We can then implement a `LinearModel` class that is responsible only for holding the type of output function, the state of the free parameters, and simple methods for updating the parameters:

```
public class LinearModel {

    private RealMatrix weight;
    private RealVector bias;
    private final OutputFunction outputFunction;

    public LinearModel(int inputDimension, int outputDimension,
    OutputFunction outputFunction) {
        weight = MatrixOperations.getUniformRandomMatrix(inputDimension,
        outputDimension, 0L);
        bias = MatrixOperations.getUniformRandomVector(outputDimension, 0L);
        this.outputFunction = outputFunction;
    }

    public RealMatrix getOutput(RealMatrix input) {
        return outputFunction.getOutput(input, weight, bias);
    }

    public void addUpdateToWeight(RealMatrix weightUpdate) {
        weight = weight.add(weightUpdate);
    }
```

```
    public void addUpdateToBias(RealVector biasUpdate) {
        bias = bias.add(biasUpdate);
    }
}
```

The interface for an output function is shown here:

```
public interface OutputFunction {
    RealMatrix getOutput(RealMatrix input, RealMatrix weight, RealVector bias);
    RealMatrix getDelta(RealMatrix error, RealMatrix output);
}
```

In most cases, we can never precisely determine **W** and **b** such that the relation between **X** and **Y** is exact. The best we can do is to estimate **Y**, calling it $\hat{\mathbf{Y}}$, and then proceed to minimize a loss function of our choice $\mathscr{L}(\mathbf{y}, \hat{\mathbf{y}})$. The goal is then to incrementally update the values of **W** and **b** over a set of iterations (annotated by *t*) according to the following:

$$\mathbf{W}_{t+1} = \mathbf{W}_t + \Delta \mathbf{W}_t$$

$$\mathbf{b}_{t+1} = \mathbf{b}_t + \Delta \mathbf{b}_t$$

In this section, we will focus on the use of the gradient descent algorithm for determining the values of **W** and **b**. Recalling that the loss function is ultimately a function of both **W** and **b**, we can use the gradient descent optimizer to make the incremental updates with the gradient of the loss:

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \eta \nabla \mathscr{L}(\mathbf{W})_t$$

$$\mathbf{b}_{t+1} = \mathbf{b}_t - \eta \nabla \mathscr{L}(\mathbf{b})_t$$

The objective function to be optimized is the mean loss $\langle \mathscr{L}(\mathbf{y}, \hat{\mathbf{y}}) \rangle$, where the gradient of any particular term with respect to a parameter w_ij and b_j can be expressed as follows:

$$\frac{\partial \mathscr{L}}{\partial w} = \frac{\partial \mathscr{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w}$$

The first term is the derivative of the loss function, which we covered in the prior section. The second part of the term is the derivative of the output function:

$$\frac{\partial \hat{y}}{\partial z} = \varphi'(z)$$

The third term is simply the derivative of z with respect to either w or b:

$$\frac{\partial z}{\partial w} = x$$

$$\frac{\partial z}{\partial b} = 1$$

As we will see, the choice of the appropriate pair of loss function and output function will lead to a mathematical simplification that leads to the *delta rule*. In this case, the updates to the weights and bias are always as follows:

$$\Delta \mathbf{W} = -\eta \mathbf{X}^T(\mathbf{Y} - \mathbf{T})$$

$$\Delta \mathbf{b} = -\eta \mathbf{h}^T(\mathbf{Y} - \mathbf{T})$$

When the mean loss $\langle \mathscr{L}(\mathbf{y}, \hat{\mathbf{y}}) \rangle$ stops changing within a certain numerical tolerance (e.g., 10E – 6), the process can stop, and we assume **W** and **b** are at their optimal values. However, iterative algorithms are prone to iterate forever because of numerical oddities. Therefore, all iterative solvers will set a maximum number of iterations (e.g., 1,000), after which the process will terminate. It is good practice to always check whether the maximum number of iterations was reached, because the change in loss may still be high, indicating that the optimal values of the free parameters have not yet been attained. The form of both the transformation function $\varphi(\mathbf{Z})$ and the loss function $L(\hat{\mathbf{Y}}, \mathbf{Y})$ will depend on the problem at hand. Several common scenarios are detailed next.

Linear

In the case of linear regression, $\varphi(\mathbf{Z})$ is set to the identity function, and the output is equivalent to the input:

$$\varphi(\mathbf{Z}) = \mathbf{Z}$$

This provides the familiar form of a linear regression model:

$$\hat{\mathbf{Y}} = \mathbf{XW} + \mathbf{hb}^T$$

We solved this problem in both Chapters 2 and 3 by using different methods. In the case of Chapter 2, we solved for the free parameters by posing the problem in matrix notation and then using a back-solver, whereas in Chapter 3 we took the least-squares approach. There are, however, even more ways to solve this problem! Ridge regression, lasso regression, and elastic nets are just a few examples. The idea is to eliminate variables that are not useful by penalizing their parameters during the optimization process:

```
public class LinearOutputFunction implements OutputFunction {

    @Override
    public RealMatrix getOutput(RealMatrix input, RealMatrix weight,
    RealVector bias) {
        return MatrixOperations.XWplusB(input, weight, bias);
    }

    @Override
    public RealMatrix getDelta(RealMatrix errorGradient, RealMatrix output) {
        // output gradient is all 1's ... so just return errorGradient
        return errorGradient;
    }

}
```

Logistic

Solve the problem where *y* is a 0 or 1 that can also be multidimensional, such as **y** = 0,1,1,0,1. The nonlinear function $\varphi(z) = \frac{1}{1 + \exp(-z)}$

For gradient descent, we need the derivative of the function:

$$\varphi'(z) = \frac{\exp(-z)}{(1 + \exp(-z))^2}$$

It is convenient to note that the derivative may also be expressed in terms of the original function. This is useful, because it allows us to reuse the calculated values of φ rather than having to recompute all the computationally costly matrix algebra:

$$\varphi'(z) = \varphi(z)(1 - \varphi(z))$$

In the case of gradient descent, we can then implement it as follows:

```
public class LogisticOutputFunction implements OutputFunction {

    @Override
    public RealMatrix getOutput(RealMatrix input, RealMatrix weight,
    RealVector bias) {
        return MatrixOperations.XWplusB(input, weight, bias, new Sigmoid());
    }
```

```
    @Override
    public RealMatrix getDelta(RealMatrix errorGradient, RealMatrix output) {

        // this changes output permanently
        output.walkInOptimizedOrder(new UnivariateFunctionMapper(
        new LogisticGradient()));

        // output is now the output gradient
        return MatrixOperations.ebeMultiply(errorGradient, output);
    }

    private class LogisticGradient implements UnivariateFunction {

        @Override
        public double value(double x) {
            return x * (1 - x);
        }
    }
}
```

When using the cross-entropy loss function to calculate the loss term, note that $\hat{y} = \varphi(z)$ such that:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})} \hat{y}(1 - \hat{y})$$

So it then reduces to this:

$$\frac{\partial \mathcal{L}}{\partial z} = \hat{y} - y$$

And considering

$$\frac{\partial z}{\partial w} = x$$

the gradient of the loss with respect to the weight is as follows:

$$\frac{\partial \mathcal{L}}{\partial w} = (\hat{y} - y)x$$

We can include a learning rate η to slow the update process. The final formulas, adapted for use with matrices of data, are given as follows:

$$\Delta \mathbf{W} = -\eta \mathbf{X}^T(\mathbf{Y} - \mathbf{T})$$

$$\Delta \mathbf{b} = -\eta \mathbf{h}^T(\mathbf{Y} - \mathbf{T})$$

Here, **h** is an *m*-dimensional vector of 1s. Notice the inclusion of the learning rate η, which usually takes on values between 0.0001 and 1 and limits how fast the parameters converge. For small values of η, we are more likely to find the correct values of the weights, but at the cost of performing many more time-consuming iterations. For larger values of η, we will complete the algorithmic learning task much quicker. However, we may inadvertently skip over the best solution, giving nonsensical values for the weights.

Softmax

Softmax is similar to logistic regression, but the target variable can be multinomial (an integer between 0 and `numClasses` – 1). We then transform the output with one-hot encoding such that **Y** = {0,0,1,0}. Note that unlike multi-output logistic regression, only one position in each row can be set to 1, and all others must be 0. Each element of the transformed matrix is then exponentiated and then L1 normalized row-wise:

$$\varphi(z_{ij}) = \frac{\exp(z_{ij})}{\sum_j \exp(z_{ij})}$$

Because the derivative involves more than one variable, the Jacobian takes the place of the gradient with terms:

$$\varphi'(z) = \begin{cases} \varphi_i(z)(1 - \varphi_i(z)) & \text{for } i = j \\ -\varphi_i(z)\varphi_j(z) & \text{for } i \neq j \end{cases}$$

Then for a single *p*-dimensional output and prediction, we can calculate the quantity:

$$\frac{\partial \mathscr{L}}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial z} = \begin{pmatrix} \frac{-y_1}{\hat{y}_1} & \frac{-y_2}{\hat{y}_2} & \cdots & \frac{-y_p}{\hat{y}_p} \end{pmatrix} \begin{pmatrix} \hat{y}_1(1-\hat{y}_1) & -\hat{y}_1\hat{y}_2 & \cdots & -\hat{y}_1\hat{y}_p \\ -\hat{y}_2\hat{y}_1 & \hat{y}_2(1-\hat{y}_2) & \cdots & -\hat{y}_2\hat{y}_p \\ \vdots & \vdots & \ddots & \vdots \\ -\hat{y}_p\hat{y}_1 & -\hat{y}_p\hat{y}_2 & \cdots & \hat{y}_p(1-\hat{y}_p) \end{pmatrix}$$

This simplifies to the following:

$$\frac{\partial \mathcal{L}}{\partial z} = ((\hat{y}_1 - y_1) \quad (\hat{y}_2 - y_2) \cdots (\hat{y}_p - y_p))$$

Each term has the exact same update rule as the other linear models under gradient descent:

$$\frac{\partial \mathcal{L}}{\partial w} = (\hat{y} - y)x$$

As a practical matter, it will take two passes over the input to compute the softmax output. First, we raise each argument by the exponential function, keeping track of the running sum. Then we iterate over that list again, dividing each term by the running sum. If (and only if) we use the softmax cross entropy as an error, the update formula for the coefficients is identical to that of logistic regression. We show this calculation explicitly in "Deep Networks".

```
public class SoftmaxOutputFunction implements OutputFunction {

    @Override
    public RealMatrix getOutput(RealMatrix input, RealMatrix weight,
                                RealVector bias) {
        RealMatrix output = MatrixOperations.XWplusB(input, weight, bias,
            new Exp());
        MatrixScaler.l1(output);
        return output;
    }

    @Override
    public RealMatrix getDelta(RealMatrix error, RealMatrix output) {

        RealMatrix delta = new BlockRealMatrix(error.getRowDimension(),
        error.getColumnDimension());

        for (int i = 0; i < output.getRowDimension(); i++) {
            delta.setRowVector(i, getJacobian(output.getRowVector(i)).
            preMultiply(error.getRowVector(i)));
        }

        return delta;
    }

    private RealMatrix getJacobian(RealVector output) {

        int numRows = output.getDimension();
        int numCols = output.getDimension();
        RealMatrix jacobian = new BlockRealMatrix(numRows, numCols);
        for (int i = 0; i < numRows; i++) {
            double output_i = output.getEntry(i);
            for (int j = i; j < numCols; j++) {
                double output_j = output.getEntry(j);
                if(i==j) {
                    jacobian.setEntry(i, i, output_i*(1-output_i));
                } else {
                    jacobian.setEntry(i, j, -1.0 * output_i * output_j);
                    jacobian.setEntry(j, i, -1.0 * output_j * output_i);
                }
            }
        }
        return jacobian;
    }
}
```

Tanh

Another common activation function utilizes the hyperbolic tangent $\tanh(z)$ with the form shown here:

$$\varphi(z) = \tanh(z)$$

$$\varphi'(z) = 1 - \tanh^2(z) = 1 - \varphi(z)^2$$

Once again, the derivative $\varphi'(z)$ reuses the value calculated from $\varphi(z)$:

```
public class TanhOutputFunction implements OutputFunction {

    @Override
    public RealMatrix getOutput(RealMatrix input, RealMatrix weight,
                                RealVector bias) {
        return MatrixOperations.XWplusB(input, weight, bias, new Tanh());
    }

    @Override
    public RealMatrix getDelta(RealMatrix errorGradient, RealMatrix output) {
        // this changes output permanently
        output.walkInOptimizedOrder(
            new UnivariateFunctionMapper(new TanhGradient()));

        // output is now the output gradient
        return MatrixOperations.ebeMultiply(errorGradient, output);
    }

    private class TanhGradient implements UnivariateFunction {
        @Override
        public double value(double x) {
            return (1 - x * x);
        }
    }
}
```

Linear model estimator

Using the gradient descent algorithm and the appropriate loss functions, we can build a simple linear estimator that updates the parameters iteratively using the delta rule. This applies only if the correct pairing of output function and loss function are used, as shown in Table 5-1.

Table 5-1. Delta rule pairings

Output function	Loss function
Linear	Quadratic
Logistic	Bernoulli cross-entropy
Softmax	Multinomial cross-entropy
Tanh	Two-point cross-entropy

We can then extend the `IterativeLearningProcess` class and add code for the output function prediction and updates:

```
public class LinearModelEstimator extends IterativeLearningProcess {

    private final LinearModel linearModel;
```

```
    private final Optimizer optimizer;

    public LinearModelEstimator(
            LinearModel linearModel,
            LossFunction lossFunction,
            Optimizer optimizer) {
        super(lossFunction);
        this.linearModel = linearModel;
        this.optimizer = optimizer;
    }

    @Override
    public RealMatrix predict(RealMatrix input) {
        return linearModel.getOutput(input);
    }

    @Override
    protected void update(RealMatrix input, RealMatrix target,
                          RealMatrix output) {
        RealMatrix weightGradient =
            input.transpose().multiply(output.subtract(target));
        RealMatrix weightUpdate = optimizer.getWeightUpdate(weightGradient);
        linearModel.addUpdateToWeight(weightUpdate);

        RealVector h = new ArrayRealVector(input.getRowDimension(), 1.0);
        RealVector biasGradient = output.subtract(target).preMultiply(h);
        RealVector biasUpdate = optimizer.getBiasUpdate(biasGradient);
        linearModel.addUpdateToBias(biasUpdate);

    }

    public LinearModel getLinearModel() {
        return linearModel;
    }

    public Optimizer getOptimizer() {
        return optimizer;
    }
}
```

Iris example

The Iris dataset is a great example to explore a linear classifier:

```
/* get data and split into train / test sets */
Iris iris = new Iris();
MatrixResampler resampler = new MatrixResampler(iris.getFeatures(),
iris.getLabels());
resampler.calculateTestTrainSplit(0.40, 0L);

/* set up the linear estimator */
LinearModelEstimator estimator = new LinearModelEstimator(
    new LinearModel(4, 3, new SoftmaxOutputFunction()),
    new SoftMaxCrossEntropyLossFunction(),
    new DeltaRule(0.001));

estimator.setBatchSize(0);
estimator.setMaxIterations(6000);
estimator.setTolerance(10E-6);

/* learn the model parameters */
estimator.learn(resampler.getTrainingFeatures(), resampler.getTrainingLabels());

/* predict on test data */
RealMatrix prediction = estimator.predict(resampler.getTestingFeatures());

/* results */
ClassifierAccuracy accuracy = new ClassifierAccuracy(prediction,
 resampler.getTestingLabels());

estimator.isConverged();            // true
estimator.getNumIterations();       // 3094
estimator.getLoss();                // 0.0769
accuracy.getAccuracy();             // 0.983
accuracy.getAccuracyPerDimension(); // {1.0, 0.92, 1.0}
```

Deep Networks

Feeding the output of a linear model into another linear model creates a nonlinear system capable of modeling complicated behavior. A system with multiple layers is known as a deep network. While a linear model has an input and output, a deep network adds multiple “hidden” layers between the input and output. Most explanations of deep networks address the input, hidden, and output layers as separate quantities. In this book, however, we take the alternative viewpoint that a deep network is nothing more than a composition of linear models. We can then view a deep network purely as a linear algebra problem. Figure 5-5 demonstrates how a multilayer neural network can be viewed as a chain of linear models.

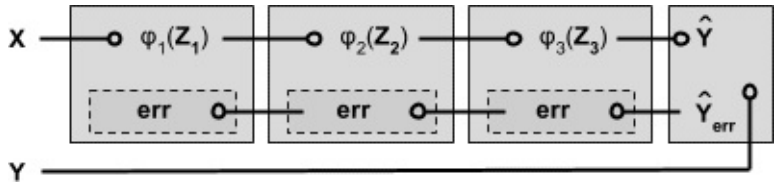

Figure 5-5. Deep network

A network layer

We can extend the concept of a linear model to the form of a network layer where we must persist the input, output, and errors. The code for the network layer is then an extension of the `LinearModel` class:

```
public class NetworkLayer extends LinearModel {

    RealMatrix input;
    RealMatrix inputError;
    RealMatrix output;
    RealMatrix outputError;
    Optimizer optimizer;

    public NetworkLayer(int inputDimension, int outputDimension,
            OutputFunction outputFunction, Optimizer optimizer) {
        super(inputDimension, outputDimension, outputFunction);
        this.optimizer = optimizer;
    }

    public void update() {

        //back propagate error
        /* D = eps o f'(XW) where o is Hadamard product
        or J f'(XW) where J is Jacobian */
        RealMatrix deltas = getOutputFunction().getDelta(outputError, output);

        /* E_out = D W^T */
        inputError = deltas.multiply(getWeight().transpose());

        /* W = W - alpha * delta * input */
        RealMatrix weightGradient = input.transpose().multiply(deltas);

        /* w_{t+1} = w_{t} + \delta w_{t} */
        addUpdateToWeight(optimizer.getWeightUpdate(weightGradient));

        // this essentially sums the columns of delta and that vector is grad_b
        RealVector h = new ArrayRealVector(input.getRowDimension(), 1.0);
        RealVector biasGradient = deltas.preMultiply(h);
        addUpdateToBias(optimizer.getBiasUpdate(biasGradient));
    }
}
```

Feed forward

To calculate the network output, we must feed the input through each layer of the network in the forward direction. The network input $\mathbf{X}_1$ is used to compute the output of the first layer:

$$\hat{\mathbf{Y}}_1 = \varphi(\mathbf{X}_1\mathbf{W}_1 + \mathbf{h}\mathbf{b}_1^T)$$

We set the first-layer output to the second-layer input:

$$\mathbf{X}_2 = \hat{\mathbf{Y}}_1$$

The output of the second layer is shown here:

$$\hat{\mathbf{Y}}_2 = \varphi(\mathbf{X}_2\mathbf{W}_2 + \mathbf{h}\mathbf{b}_2^T) = \varphi(\hat{\mathbf{Y}}_1\mathbf{W}_2 + \mathbf{h}\mathbf{b}_2^T)$$

In general, the output of each layer l after the first layer is expressed in terms of the prior-layer output:

$$\hat{\mathbf{Y}}_l = \varphi(\hat{\mathbf{Y}}_{l-1}\mathbf{W}_l) + \mathbf{h}\mathbf{b}_l^T)$$

The feed-forward process for L layers then appears as a series of nested linear models:

$$\hat{\mathbf{Y}}_L = \varphi_L(\cdots \varphi_2(\varphi_1(\mathbf{X}_1\mathbf{W}_1 + \mathbf{h}\mathbf{b}_1^T)\mathbf{W}_2 + \mathbf{h}\mathbf{b}_2^T)\cdots \mathbf{W}_L + \mathbf{h}\mathbf{b}_L^T)$$

Another simpler way to write this expression is as a composition of functions:

$$\hat{\mathbf{Y}}_L = \varphi_L \circ \cdots \circ \varphi_2 \circ \varphi_1(\mathbf{Z})$$

In addition to unique weights, each layer can take a different form for the activation function. In this way, it is clear that a feed-forward, deep neural network (a.k.a. multilayer perceptron) is nothing more than a composition of arbitrary linear models. The result, however, is a rather complex nonlinear model.

Back propagation

At this point, we need to back-propagate the network output error. For the last layer (the output layer), we back-propagate the loss gradient $\nabla\mathscr{L}(\mathbf{Y}, \hat{\mathbf{Y}})$. For compatible loss function–output function pairs, this is identical to a linear model estimator. As formulated in "Gradient Descent Optimizer", it is then convenient to define a new quantity, the layer delta, **D**, which is the matrix multiplication of $\mathbf{Y}_{err}$ with the tensor of output function Jacobians:

$$\mathbf{D} = \hat{\mathbf{Y}}_{err}\mathbf{J}_{\varphi}^{(m)}$$

In most cases, the gradient of the output function will suffice and the prior expression can be simplified as as follows:

$$\mathbf{D} = \hat{\mathbf{Y}}_{err} \circ \varphi'(\mathbf{Z})$$

We store the quantity **D** since it is used in two places that must be calculated in order. The back-propagated error is updated first:

$$\mathbf{X}_{err} = \mathbf{D}\mathbf{W}^T$$

The weight and bias gradients are then calculated as follows, where **h** is an m-length vector of 1s.

$$\nabla\mathbf{W} = \mathbf{X}^T\mathbf{D}$$

$$\nabla\mathbf{b} = \mathbf{h}^T\mathbf{D}$$

Note that the expression $\mathbf{h}^T\mathbf{D}$ is the equivalent of summing each column of **D**. The layer weights can then be updated using the optimization rule of choice (usually gradient descent of some variety). That completes all the calculations needed for the network layer! Then set the $\mathbf{Y}_{err}$ of the next layer down to the freshly calculated $\mathbf{X}_{err}$ and repeat the process until the first layer's parameters are updated.

> **WARNING**
>
> Make sure to calculate the back-propagated error before updating the weight!

Deep network estimator

Learning the parameters of a deep network is accomplished with the same iterative process as a linear model. In this case, the entire feed-forward process acts as one prediction step and the back propagation process acts as one update step. We implement a deep network estimator by extending the `IterativeLearningProcess` and building layers of linear models subclassed as `NetworkLayers`:

```
public class DeepNetwork extends IterativeLearningProcess {

    private final List<NetworkLayer> layers;

    public DeepNetwork() {
```

```
        this.layers = new ArrayList<>();
    }

    public void addLayer(NetworkLayer networkLayer) {
        layers.add(networkLayer);
    }

    @Override
    public RealMatrix predict(RealMatrix input) {

        /* the initial input MUST BE DEEP COPIED or is overwritten */
        RealMatrix layerInput = input.copy();

        for (NetworkLayer layer : layers) {
            layer.setInput(layerInput);

            /* calc the output and set to next layer input*/
            RealMatrix output = layer.getOutput(layerInput);
            layer.setOutput(output);

            /*
                does not need a deep copy, but be aware that
                every layer input shares memory of prior layer output
            */
            layerInput = output;
        }
        /* layerInput is holding the final output ... get a deep copy */
        return layerInput.copy();
    }

    @Override
    protected void update(RealMatrix input, RealMatrix target,
                          RealMatrix output) {

        /* gradient of the network error and starts the back prop process */
        RealMatrix layerError = getLossFunction()
                                    .getLossGradient(output, target).copy();

        /* create list iterator and set cursor to last! */
        ListIterator li = layers.listIterator(layers.size());

        while (li.hasPrevious()) {
            NetworkLayer layer = (NetworkLayer) li.previous();
            /* get error input from higher layer */
            layer.setOutputError(layerError);
            /* this back propagates the error and updates weights */
            layer.update();
            /* pass along error to next layer down */
            layerError = layer.getInputError();
        }
    }
}
```

MNIST example

MNIST, the classic handwritten digits dataset, is often used for testing learning algorithms. Here we get 94 percent accuracy by using a simple network with two hidden layers:

```
MNIST mnist = new MNIST();

DeepNetwork network = new DeepNetwork();

/* input, hidden and output layers */
network.addLayer(new NetworkLayer(784, 500, new TanhOutputFunction(),
    new GradientDescentMomentum(0.0001, 0.95)));

network.addLayer(new NetworkLayer(500, 300, new TanhOutputFunction(),
    new GradientDescentMomentum(0.0001, 0.95)));

network.addLayer(new NetworkLayer(300, 10, new SoftmaxOutputFunction(),
    new GradientDescentMomentum(0.0001, 0.95)));

/* runtime parameters */
network.setLossFunction(new SoftMaxCrossEntropyLossFunction());
```

```
network.setMaxIterations(6000);
network.setTolerance(10E-9);
network.setBatchSize(100);

/* learn */
network.learn(mnist.trainingData, mnist.trainingLabels);

/* predict */
RealMatrix prediction = network.predict(mnist.testingData);

/* compute accuracy */
ClassifierAccuracy accuracy =
    new ClassifierAccuracy(prediction, mnist.testingLabels);

/* results */
network.isConverged(); // false
network.getNumIterations(); // 10000
network.getLoss(); // 0.00633
accuracy.getAccuracy(); // 0.94
```

Chapter 6. Hadoop MapReduce

You write MapReduce jobs in Java when you need low-level control and want to optimize or streamline your big data pipeline. Using MapReduce is not required, but it is rewarding, because it is a beautifully designed system and API. Learning the basics can get you very far, very quickly, but before you embark on writing a customized MapReduce job, don’t overlook the fact that tools such as Apache Drill enable you to write standard SQL queries on Hadoop.

This chapter assumes you have a running Hadoop Distributed File System (HDFS) on your local machine or have access to a Hadoop cluster. To simulate how a real MapReduce job would run, we can run Hadoop in pseudodistributed mode on one node, either your localhost or a remote machine. Considering how much CPU, RAM, and storage resources we can fit on one box (laptop) these days, you can, in essence, create a mini supercomputer capable of running fairly massive distributed jobs. You can get pretty far on your localhost (on a subset of data) and then scale up to a full cluster when your application is ready.

If the Hadoop client is properly installed, you can get a complete listing of all available Hadoop operations by simply typing the following:

```
bash$ hadoop
```

Hadoop Distributed File System

Apache Hadoop comes with a command-line tool useful for accessing the Hadoop filesystem and launching MapReduce jobs. The filesystem access command `fs` is invoked as follows:

```
bash$ hadoop fs <command> <args>
```

The command is any number of standard Unix filesystem commands such as `ls`, `cd`, or `mkdir` preceded by a hyphen. For example, to list all the items in the HDFS root directory, type this:

```
bash$ hadoop fs -ls /
```

Note the inclusion of the `/` for root. If it were not included at all, the command would return nothing and might fool you into thinking that your HDFS is empty! Typing `hadoop fs` will print out all the filesystem operations available. Some of the more useful operations involve copying data to and from HDFS, deleting directories, and merging data in a directory.

To copy local files into a Hadoop filesystem:

```
bash$ hadoop fs -copyFromLocal <localSrc> <dest>
```

To copy a file from HDFS to your local drive:

```
bash$ hadoop fs -copyToLocal <hdfsSrc> <localDest>
```

After a MapReduce job, there will most likely be many files contained in the output directory of the job. Instead of retrieving these one by one, Hadoop has a convenient operation for merging the files into one and then storing the results locally:

```
bash$ hadoop fs -getmerge <hdfs_output_dir> <my_local_dir>
```

One essential operation for running MapReduce jobs is to first remove the output directory if it already exists, because MapReduce will fail, almost immediately, if it detects the output directory:

```
bash$ hadoop fs -rm rf <hdfs_dir>
```

MapReduce Architecture

MapReduce invokes the *embarrassingly parallel* paradigm of distributed computing. Initially, the data is broken into chunks, and portions are sent to identical mapper classes that extract key-value pairs from the data, line by line. The key-value pairs are then partitioned into key-list pairs where the lists are sorted. Typically, the number of partitions is the number of reduce jobs, but this is not required. In fact, multiple key-list groups can be in the same partition and reducer, but each key-list group is guaranteed not to be split across partitions or reducers. The general flow of data through a MapReduce framework is displayed in Figure 6-1.

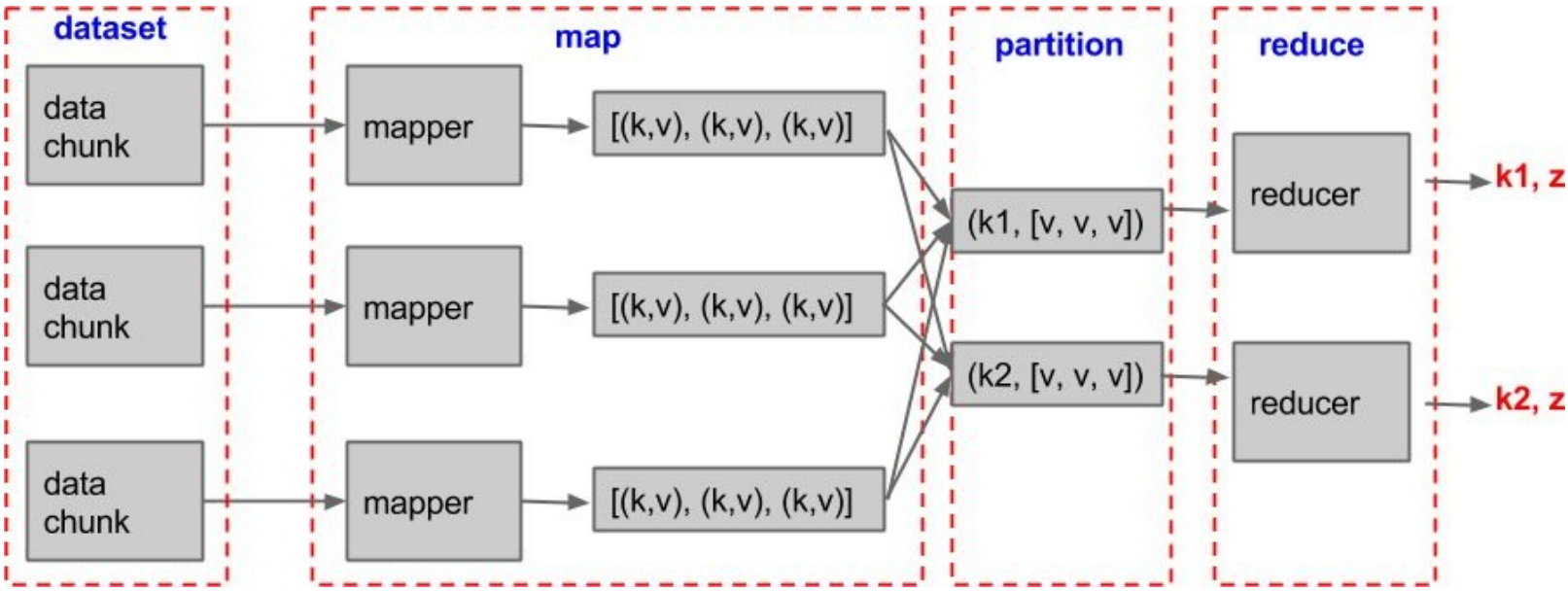

Figure 6-1. MapReduce schema

Say, for example, we have data like this:

```
San Francisco, 2012
New York, 2012
San Francisco, 2017
New York, 2015
New York, 2016
```

The mapper could output key-value pairs such as (San Francisco, 2012) for each line in the dataset. Then the partitioner would collect the data by key and sort the list of values:

```
(San Francisco, [2012, 2017])
(New York, [2012, 2015, 2016])
```

We could designate the reducer's function to output the maximum year such that the final output (written to the output directory) would look like this:

```
San Francisco, 2017
New York, 2016
```

It is important to consider that the Hadoop MapReduce API allows compound keys and customizable comparators for partitioning keys and sorting values.

Writing MapReduce Applications

Although there are more than a few ways to store and shuttle around data in the Hadoop ecosystem, we will focus on plain old text files. Whether the underlying data is stored as a string, CSV, TSV, or JSON data string, we easily read, share, and manipulate the data. Hadoop also provides resources for reading and writing its own Sequence and Map file formats, and you may want to explore various third-party serialization formats such as Apache Avro, Apache Thrift, Google Protobuf, Apache Parquet, and others. All of these provide operational and efficiency advantages. However, they do add a layer of complexity that you must consider.

Anatomy of a MapReduce Job

A basic MapReduce job has just a few essential features. The main guts of the overriden `run()` method contain a singleton instance of the `Job` class:

```
public class BasicMapReduceExample extends Configured implements Tool {

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new BasicMapReduceExample(), args);
        System.exit(exitCode);
    }

    @Override
    public int run(String[] args) throws Exception {

        Job job = Job.getInstance(getConf());
        job.setJarByClass(BasicMapReduceExample.class);
        job.setJobName("BasicMapReduceExample");

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        return job.waitForCompletion(true) ? 0 : 1;
    }

}
```

Note that because we have not defined any `Mapper` or `Reducer` classes, this job will use the default classes that copy the input files to the output directory, unchanged. Before we delve into customizing `Mapper` and `Reducer` classes, we must first understand the exclusive data types that are required by Hadoop MapReduce.

Hadoop Data Types

Data must be shuttled around through the MapReduce universe in a format that is both reliable and efficient. Unfortunately (according to the authors of Hadoop), the native Java primitive types (e.g., `boolean`, `int`, `double`) and the more complex types (e.g., `String`, `Map`) do not travel well! For this reason, the Hadoop ecosystem has its own version of serializable types that are required in all MapReduce applications. Note that all the regular Java types are perfectly fine inside our MapReduce code. It is only for the connections between MapReduce components (between mapper and reducers) where we need to convert native Java types to Hadoop types.

Writable and WritableComparable types

The Java primitive types are all represented, but the most useful ones are `BooleanWritable`, `IntWritable`, `LongWritable`, and `DoubleWritable`. A Java `String` type is represented by `Text`. A null is `NullWritable`, which comes in handy when we have no data to pass through a particular key or value in a MapReduce job. There is even an `MD5Hash` type, which could be used, among other things, when we are using a key that is a hash corresponding to `userid` or some other unique identifier. There is also a `MapWritable` for creating a `Writable` comparable `HashMap`. All of these types are comparable (e.g., they have `hash()` and `equals()` methods that enable comparison and sorting events in the MapReduce job). Of course, there are more types, but these are a few of the more useful ones. In general, a Hadoop type takes the Java primitive as an argument in the constructor:

```
Int count = 42;
IntWritable countWritable = new IntWritable(count);

String data = "The is a test string";
Text text = new Text(data);
```

Note that Java types are used inside your code for `Mapper` and `Reducer` classes. *Only* the key and value inputs and outputs for those instances must use the Hadoop writable (and writable comparable if a key) types, because this is how data is shuttled between the MapReduce components.

Custom Writable and WritableComparable types

At times we need a specialized type not covered by Hadoop. In general, a Hadoop type must implement `Writable`, which handles the object's serialization with a `write()` method and deserialization with a `read()` method. However, if the object will be used as a key, it must implement `WritableComparable`, because the `compareTo()` and `hashCode()` methods will be required during partitioning and sorting.

Writable

Because the `Writable` interface has only two methods, `write()` and `readFields()`, a basic custom writable needs to override only these methods. However, we can add a constructor that takes arguments so that we can instantiate the object in the same way we created `IntWritable` and `Text` instances in the previous example. In addition, if we add a static `read()` method, we will require a no-argument constructor:

```
public class CustomWritable implements Writable {
```

```
    private int id;
    private long timestamp;

    public CustomWritable() {
    }

    public CustomWritable(int id, long timestamp) {
        this.id = id;
        this.timestamp = timestamp;
    }

    public void write(DataOutput out) throws IOException {
        out.writeInt(id);
        out.writeLong(timestamp);
    }

    public void readFields(DataInput in) throws IOException {
        id = in.readInt();
        timestamp = in.readLong();
    }

    public static CustomWritable read(DataInput in) throws IOException {
        CustomWritable w = new CustomWritable();
        w.readFields(in);
        return w;
    }
}
```

WritableComparable

If our custom writable will be used a key, we will need `hashCode()` and `compareTo()` methods in addition to the `write()` and `readField()` methods:

```
public class CustomWritableComparable implements WritableComparable {

    private int id;
    private long timestamp;

    public CustomWritable() {
    }

    public CustomWritable(int id, long timestamp) {
        this.id = id;
        this.timestamp = timestamp;
    }

    public void write(DataOutput out) throws IOException {
        out.writeInt(id);
        out.writeLong(timestamp);
    }

    public void readFields(DataInput in) throws IOException {
        id = in.readInt();
        timestamp = in.readLong();
    }

    public int compareTo(CustomWritableComparable o) {
        int thisValue = this.value;
        int thatValue = o.value;
        return (thisValue < thatValue ? -1 : (thisValue==thatValue ? 0 : 1));
    }

    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + id;
        result = prime * result + (int) (timestamp ^ (timestamp >>> 32));
        return result
    }
}
```

Mappers

The `Mapper` class is what maps the raw input data into a new and typically smaller sized data structure. In general, you do not need every piece of data from each line of the input files, but rather a select few items. In some cases, the line may be discarded entirely. This is your chance to decide what data will go into the next round of processing. Think of this step as a way of transforming and filtering the raw data into only the parts we actually need. If you do not include a `Mapper` instance in a MapReduce job, the `IdentityMapper` will be assumed, which just passes all the data directly through to the reducer. And if there is no reducer, input will be essentially copied to output.

Generic mappers

Several common mappers that are already included with Hadoop can be designated in the MapReduce job. The default is the `IdentityMapper`, which outputs the exact data it inputs. The `InverseMapper` switches the key and value. There is also `TokenCounterMapper`, which outputs each token and its count as a `Text`, `IntWritable` key-value pair. The `RegexMapper` outputs a regex match as the key and constant value of 1. If none of these work for your application, consider writing your own customized mapper instance.

Customizing a mapper

Parsing text files within a `Mapper` class is much the same as parsing lines from a regular text file, as in Chapter 1. The only required method is the `map()` method. The fundamental purpose of this `map()` method is to parse one line of input and output a key-value pair via the `context.write()` method:

```
public class ProductMapper extends
    Mapper<LongWritable, Text, IntWritable, Text> {

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        try {

            /* each line of file is <userID>, <productID>, <timestamp> */

            String[] items = value.toString().split(",");
            int userID = Integer.parseInt(items[0]);
            String productID = items[1];
            context.write(new IntWritable(userID), new Text(productID));

        } catch (NumberFormatException | IOException | InterruptedException e) {

            context.getCounter("mapperErrors", e.getMessage()).increment(1L);
        }
    }

}
```

There are also `startup()` and `cleanup()` methods. The `startup()` method is run once when the `Mapper` class is instantiated. You probably won't need it, but it comes in handy, for example, when you need a data structure that needs to be used by each call to the `map()` method. Likewise, you probably won't need the `cleanup()` method, but it is called once after the last call to `map()` and is used to do any cleanup actions. There is also a `run()` method, which does the actual business of mapping the data. There is no real reason to override this method, and it's best to leave it alone unless you have a good reason to

implement your own `run()` method. In "MapReduce Examples", we show how to utilize the `setup()` method for some unique computations.

To use a custom mapper, you must designate it in the MapReduce application and set the map output key and value types:

```
job.setMapperClass(ProductMapper.class);
job.setMapOutputKeyClass(IntWritable.class);
job.setMapOutputValueClass(Text.class);
```

Reducers

The role of the `Reducer` is to iterate over the list of values associated with a key and calculate a singular output value. Of course, we can customize the output type of the `Reducer` to return anything we would like as long as it implements `Writable`. It is important to note that each reducer will process at least one key and all its values, so you do not need to worry that some values belonging to a key have been sent somewhere else. The number of reducers is also the number of output files.

Generic reducers

If a `Reducer` instance is not specified, the MapReduce job sends mapped data directly to the output. There are some useful reducers in the Hadoop library that come in handy. The `IntSumReducer` and `LongSumReducer` take respective `IntWritable` and `LongWritable` integers as values in the `reduce()` method. The outputs are then the sum of all the values. Counting is such a common use case for MapReduce that these classes are purely convenient.

Customizing a reducer

The code for a reducer has a similar structure to the mapper. We usually need to override only `reduce()` with our own code, and on occasion we'll use the `setup()` method when we are building a specific data structure or file-based resource that must be utilized by all reducers. Note that the reducer signature takes an `Iterable` of the value type because after the mapper phase, all the data for a particular key is grouped and sorted into a list (`Iterable`) and input here:

```
public class CustomReducer extends
    Reducer<IntWritable, Text, IntWritable, IntWritable>{

    @Override
    protected void reduce(IntWritable key, Iterable<Text> values,
        Context context) throws IOException, InterruptedException {

        int someValue = 0;

        /* iterate over the values and do something */
        for (Text value : values) {
            // use value to augment someValue
        }

        context.write(key, new IntWritable(someValue));

    }
```

The `Reducer` class and its key and value output types need to be specified in the MapReduce job:

```
job.setReducerClass(CustomReducer.class);
job.setOutputKeyClass(IntWritable.class);
job.setOutputValueClass(IntWritable.class);
```

The Simplicity of a JSON String as Text

JSON data (where each row of a file is a separate JSON string) is everywhere, and for good reason. Many tools are capable of ingesting JSON data, and its human readability and built-in schema are really helpful. In the MapReduce world, using JSON data as input data eliminates the need for custom writables because the JSON string can be serialized in the Hadoop `Text` type. This process can be as simple as using `JSONObject` right in the `map()` method. Or you can create a class to consume the `value.toString()` for more complicated mapping schemas.

```
public class JSONMapper extends Mapper<LongWritable, Text, Text, Text> {

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        JSONParser parser = new JSONParser();
        try {
            JSONObject obj = (JSONObject) parser.parse(value.toString());

            // get what you need from this object
            String userID = obj.get("user_id").toString();
            String productID = obj.get("product_id").toString();
            int numUnits = Integer.parseInt(obj.get("num_units").toString());

            JSONObject output = new JSONObject();
            output.put("productID", productID);
            output.put("numUnits", numUnits);

            /* many more key value pairs, including arrays, can be added here */

            context.write(new Text(userID), new Text(output.toString()));


        } catch (ParseException ex) {
            //error parsing json
        }

    }
}
```

This also works great for outputting the data from the final reducer as a `Text` object. The final data file will be in JSON data format to enable efficient use down the rest of your pipeline. Now the reducer can input a `Text`, `Text` key-value pair and process the JSON with `JSONObject`. The advantage is that we did not have to create a complicated custom `WritableComparable` for this data structure.

Deployment Wizardry

There are many options and command-line switches for running a MapReduce job. Remember that before you run a job, the output directory needs to be deleted first:

```
bash$ hadoop fs -rm -r <path>/output
```

Running a standalone program

You will certainly see (and probably write yourself) one file that contains the entire MapReduce job. The only real difference is that you must define any custom `Mapper`, `Reducer`, `Writable`, and so forth, as static. Otherwise, the mechanics are all the same. The obvious advantage is that you have a completely self-contained job without any worry of dependencies, and as such, you don't have to worry about JARs, and so forth. Just build the Java file (at the command line with `javac`) and run the class like this:

```
bash$ hadoop BasicMapReduceExample input output
```

Deploying a JAR application

If the MapReduce job is part of a larger project that has become a JAR possibly containing many such jobs, you will need to deploy from the JAR and designate the full URI of the job:

```
hadoop jar MyApp.jar com.datascience.BasicMapReduceExample input output
```

Including dependencies

Include a comma-separated list of files that must be used in the MapReduce job as follows:

```
-files file.dat, otherFile.txt, myDat.json
```

Any JARs required can be added with a comma-separated list:

```
-libjars myJar.jar, yourJar.jar, math.jar
```

Note that command-line switches such as `-files` and `-libjars` must be placed before any command arguments such as input and output.

Simplifying with a BASH script

At some point, typing all this text in the command line is error prone and laborious. So is scrolling through your bash history to find that command you launched last week. You can create custom scripts for specific tasks that take command-line arguments, like the input and output directories, or even which class to run. Consider putting it all in an executable bash script like this:

```
#!/bin/bash

# process command-line input and output dirs
INPUT=$1
OUTPUT=$2

# these are hardcoded for this script
LIBJARS=/opt/math3.jar, morejars.jar
FILES=/usr/local/share/meta-data.csv, morefiles.txt
```

```
APP_JAR=/usr/local/share/myApp.jar
APP_CLASS=com.myPackage.MyMapReduceJob

# clean the output dir
hadoop fs -rm -r $OUTPUT

# launch the job
hadoop jar $APP_JAR $APP_CLASS -files $FILES -libjars $LIBJARS $INPUT $OUTPUT
```

Then you have to remember to make the script executable (just once):

```
bash$ chmod +x runMapReduceJob.sh
```

And then run it like this:

```
bash$ myJobs/runMapReduceJob.sh inputDirGoesHere outputDirGoesHere
```

Or if you are running it from the same directory that the script is located in, use this:

```
bash$ ./runMapReduceJob.sh inputDirGoesHere outputDirGoesHere
```

MapReduce Examples

To really master MapReduce, you need to practice. There is no better way to understand how it all works than to jump in and start solving problems. Although the system may seem complex and cumbersome at first, its beauty will reveal itself as you have some successes. Here are some typical examples and some insightful computations.

Word Count

Here we use the built-in mapper class for counting tokens, `TokenCounterMapper`, and the built-in reducer class for summing integers, `IntSumReducer`:

```
public class WordCountMapReduceExample extends Configured implements Tool {

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new WordCountMapReduceExample(), args);
        System.exit(exitCode);
    }

    @Override
    public int run(String[] args) throws Exception {
        Job job = Job.getInstance(getConf());
        job.setJarByClass(WordCountMapReduceExample.class);
        job.setJobName("WordCountMapReduceExample");

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(TokenCounterMapper.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setNumReduceTasks(1);

        return job.waitForCompletion(true) ? 0 : 1;
    }
}
```

The job can be run on an input directory that has any type of text file:

```
hadoop jar MyApp.jar \\
com.datascience.WordCountMapReduceExample input output
```

The output can be viewed with the following:

```
hadoop fs -cat output/part-r-00000
```

Custom Word Count

We may notice the built-in `TokenCounterMapper` class is not producing the results we like. We can always use our `SimpleTokenizer` class from Chapter 4:

```
public class SimpleTokenMapper extends
    Mapper<LongWritable, Text, Text, LongWritable> {

    SimpleTokenizer tokenizer;

    @Override
    protected void setup(Context context) throws IOException {
        // only keep words greater than three chars
        tokenizer = new SimpleTokenizer(3);
    }

    @Override
    protected void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {

        String[] tokens = tokenizer.getTokens(value.toString());
        for (String token : tokens) {
            context.write(new Text(token), new LongWritable(1L));
        }

    }
}
```

Just be sure to set the appropriate changes in the job:

```
/* mapper settings */
job.setMapperClass(SimpleTokenMapper.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(LongWritable.class);

/* reducer settings */
job.setReducerClass(LongSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(LongWritable.class);
```

Sparse Linear Algebra

Imagine that we have a large matrix (either sparse or dense) in which the `i,j` coordinates and corresponding value are stored in each line of the file in the format `<i,j,value>`. This matrix is so large that it is not practical to load it into RAM for further linear algebra routines. Our goal is to perform the matrix vector multiplication with an input vector we provide. The vector has been serialized so that the file can be included in the MapReduce job.

Imagine we have stored text files of comma- (or tab-) separated values across many nodes in our distributed filesystem. If the data is stored as a literal `i,j,value` text string (e.g., 34, 290, 1.2362) in each line of the file, then we can write a simple mapper to parse each line. In this case, we will do matrix multiplication, and as you may recall, that process requires multiplying each row of the matrix by the column vector of the same length. Each position `i` of the output vector will then take the same index as the corresponding matrix row. So we will use the matrix row i as the key. We will create a custom writable `SparseMatrixWritable` that contains the row index, column index, and value for each entry in the matrix:

```
public class SparseMatrixWritable implements Writable {
    int rowIndex; // i
    int columnIndex; // j
    double entry; // the value at i,j

    public SparseMatrixWritable() {
    }

    public SparseMatrixWritable(int rowIndex, int columnIndex, double entry) {
        this.rowIndex = rowIndex;
        this.columnIndex = columnIndex;
        this.entry = entry;
    }

    @Override
    public void write(DataOutput d) throws IOException {
        d.writeInt(rowIndex);
        d.writeInt(rowIndex);
        d.writeDouble(entry);
    }

    @Override
    public void readFields(DataInput di) throws IOException {
        rowIndex = di.readInt();
        columnIndex = di.readInt();
        entry = di.readDouble();
    }

}
```

A custom mapper will read in each line of text and parse the three values, using the row index as the key and the `SparseMatrixWritable` as the value:

```
public class SparseMatrixMultiplicationMapper
 extends Mapper<LongWritable, Text, IntWritable, SparseMatrixWritable> {

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        try {
            String[] items = value.toString().split(",");
            int rowIndex = Integer.parseInt(items[0]);
            int columnIndex = Integer.parseInt(items[1]);
            double entry = Double.parseDouble(items[2]);
            SparseMatrixWritable smw = new SparseMatrixWritable(
```

```
            rowIndex, columnIndex, entry);
            context.write(new IntWritable(rowIndex), smw);
            // NOTE can add another context.write() for
            // e.g., a symmetric matrix entry if matrix is sparse upper triag
        } catch (NumberFormatException | IOException | InterruptedException e) {
            context.getCounter("mapperErrors", e.getMessage()).increment(1L);
        }
    }
}
```

The reducer must load in the input vector in the `setup()` method, and then in the `reduce()` method we extract column indices from the list of `SparseMatrixWritable`, adding them to a sparse vector. The dot product of the input vector and sparse vector give the value for the output for that key (e.g., the value of the resultant vector at that index).

```
public class SparseMatrixMultiplicationReducer extends Reducer<IntWritable,
                    SparseMatrixWritable, IntWritable, DoubleWritable>{

    private RealVector vector;

    @Override
    protected void setup(Context context)
        throws IOException, InterruptedException {

        /* unserialize the RealVector object */

        // NOTE this is just the filename
        // please include the resource itself in the dist cache
        // via -files at runtime
        // set the filename in Job conf with
        // set("vectorFileName", "actual file name here")

        String vectorFileName = context.getConfiguration().get("vectorFileName");
        try (ObjectInputStream in = new ObjectInputStream(
        new FileInputStream(vectorFileName))) {
            vector = (RealVector) in.readObject();
        } catch(ClassNotFoundException e) {
            // err
        }
    }

    @Override
    protected void reduce(IntWritable key, Iterable<SparseMatrixWritable> values,
    Context context)
        throws IOException, InterruptedException {

        /* rely on the fact that rowVector dim == input vector dim */
        RealVector rowVector = new OpenMapRealVector(vector.getDimension());

        for (SparseMatrixWritable value : values) {
            rowVector.setEntry(value.columnIndex, value.entry);
        }

        double dotProduct = rowVector.dotProduct(vector);

        /* only write the nonzero outputs,
        since the Matrix-Vector product is probably sparse */
        if(dotProduct != 0.0) {
            /* this outputs the vector index and its value */
            context.write(key, new DoubleWritable(dotProduct));
        }
    }
}
```

The job can be set up to run like this:

```
public class SparseAlgebraMapReduceExample extends Configured implements Tool {

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new SparseAlgebraMapReduceExample(), args);
```

```
        System.exit(exitCode);
    }

    @Override
    public int run(String[] args) throws Exception {
        Job job = Job.getInstance(getConf());
        job.setJarByClass(SparseAlgebraMapReduceExample.class);
        job.setJobName("SparseAlgebraMapReduceExample");

        // third command-line arg is the filepath to the serialized vector file
        job.getConfiguration().set("vectorFileName", args[2]);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(SparseMatrixMultiplicationMapper.class);
        job.setMapOutputKeyClass(IntWritable.class);
        job.setMapOutputValueClass(SparseMatrixWritable.class);
        job.setReducerClass(SparseMatrixMultiplicationReducer.class);
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(DoubleWritable.class);
        job.setNumReduceTasks(1);

        return job.waitForCompletion(true) ? 0 : 1;
    }
}
```

This can be run with the following command:

```
hadoop jar MyApp.jar \\
com.datascience.SparseAlgebraMapReduceExample \\
-files /<path>/RandomVector.ser input output RandomVector.ser
```

You can view the output with this:

```
hadoop fs -cat output/part-r-00000
```

Appendix A. Datasets

All datasets are stored under *src/main/resources/datasets*. While Java class codes are stored under *src/main/java*, user resources are stored under *src/main/resources*. In general, we use the JAR loader functionality to retrieve contents of a file directly from the JAR, not from the filesystem.

Anscombe's Quartet

Anscombe's quartet is a set of four *x-y* pairs of data with remarkable properties. Although the x-y plots of each pair look completely different, the data has the properties that make statistical measures almost identical. The values for each of the four x-y data series are in Table A-1.

Table A-1. Anscombe's quartet data

x1	y1	x2	y2	x3	y3	x4	y4
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

We can easily hardcode the data as static members of the class:

```
public class Anscombe {
    public static final double[] x1 = {10.0, 8.0, 13.0, 9.0, 11.0,
                                       14.0, 6.0, 4.0, 12.0, 7.0, 5.0};
    public static final double[] y1 = {8.04, 6.95, 7.58, 8.81, 8.33,
                                       9.96, 7.24, 4.26, 10.84, 4.82, 5.68};
    public static final double[] x2 = {10.0, 8.0, 13.0, 9.0, 11.0,
                                       14.0, 6.0, 4.0, 12.0, 7.0, 5.0};
    public static final double[] y2 = {9.14, 8.14, 8.74, 8.77, 9.26,
                                       8.10, 6.13, 3.10, 9.13, 7.26, 4.74};
    public static final double[] x3 = {10.0, 8.0, 13.0, 9.0, 11.0,
                                       14.0, 6.0, 4.0, 12.0, 7.0, 5.0};
    public static final double[] y3 = {7.46, 6.77, 12.74, 7.11, 7.81,
                                       8.84, 6.08, 5.39, 8.15, 6.42, 5.73};
    public static final double[] x4 = {8.0, 8.0, 8.0, 8.0, 8.0, 8.0,
                                       8.0, 19.0, 8.0, 8.0, 8.0};
    public static final double[] y4 = {6.58, 5.76, 7.71, 8.84, 8.47,
                                       7.04, 5.25, 12.50, 5.56, 7.91, 6.89};
}
```

Then we can call any array:

```
double[] x1 = Anscombe.x1;
```

Sentiment

This is the sentiment-labeled dataset from *https://archive.ics.uci.edu/ml/datasets/Sentiment+Labelled+Sentences*. Download three files and place them in *src/main/resources/datasets/sentiment*. They contain data from IMDb, Yelp, and Amazon. There is a single sentence and then a tab-delimited 0 or 1 corresponding to respective negative or positive sentiment. Not all sentences have a corresponding label.
IMDb has 1,000 sentences, with 500 positive (1) and 500 negative (0). Yelp has 3,729 sentences, with 500 positive (1) and 500 negative (0). Amazon has 15,004 sentences, with 500 positive (1) and 500 negative (0):

```
public class Sentiment {


    private final List<String> documents = new ArrayList<>();
    private final List<Integer> sentiments = new ArrayList<>();
    private static final String IMDB_RESOURCE =


    "/datasets/sentiment/imdb_labelled.txt";
    private static final String YELP_RESOURCE =


    "/datasets/sentiment/yelp_labelled.txt";
    private static final String AMZN_RESOURCE =


    "/datasets/sentiment/amazon_cells_labelled.txt";
    public enum DataSource {IMDB, YELP, AMZN};


    public Sentiment() throws IOException {
        parseResource(IMDB_RESOURCE); // 1000 sentences
        parseResource(YELP_RESOURCE); // 1000 sentences
        parseResource(AMZN_RESOURCE); // 1000 sentences
    }


    public List<Integer> getSentiments(DataSource dataSource) {
        int fromIndex = 0; // inclusive
        int toIndex = 3000; // exclusive
        switch(dataSource) {
            case IMDB:
                fromIndex = 0;
                toIndex = 1000;
                break;
            case YELP:
                fromIndex = 1000;
                toIndex = 2000;
                break;
            case AMZN:
                fromIndex = 2000;
                toIndex = 3000;
                break;
        }
        return sentiments.subList(fromIndex, toIndex);
    }
```

```
    public List<String> getDocuments(DataSource dataSource) {
        int fromIndex = 0; // inclusive
        int toIndex = 3000; // exclusive
        switch(dataSource) {
            case IMDB:
                fromIndex = 0;
                toIndex = 1000;
                break;
            case YELP:
                fromIndex = 1000;
                toIndex = 2000;
                break;
            case AMZN:
                fromIndex = 2000;
                toIndex = 3000;
                break;
        }
        return documents.subList(fromIndex, toIndex);
    }


    public List<Integer> getSentiments() {
        return sentiments;
    }

    public List<String> getDocuments() {
        return documents;
    }


    private void parseResource(String resource) throws IOException {
        try(InputStream inputStream = getClass().getResourceAsStream(resource)) {
            BufferedReader br =

                new BufferedReader(new InputStreamReader(inputStream));
            String line;
            while ((line = br.readLine()) != null) {
                String[] splitLine = line.split("\t");
                // both yelp and amzn have many sentences with no label
                if (splitLine.length > 1) {
                    documents.add(splitLine[0]);
                    sentiments.add(Integer.parseInt(splitLine[1]));
                }
            }
        }
    }
}
```

Gaussian Mixtures

Generate a mixture of multivariate normal distributions data:

```
public class MultiNormalMixtureDataset {
    int dimension;
    List<Pair<Double, MultivariateNormalDistribution>> mixture;
    MixtureMultivariateNormalDistribution mixtureDistribution;


    public MultiNormalMixtureDataset(int dimension) {
        this.dimension = dimension;
        mixture = new ArrayList<>();
    }

    public MixtureMultivariateNormalDistribution getMixtureDistribution() {
        return mixtureDistribution;
    }

    public void createRandomMixtureModel(
    int numComponents, double boxSize, long seed) {
        Random rnd = new Random(seed);
        double limit = boxSize / dimension;
        UniformRealDistribution dist =

            new UniformRealDistribution(-limit, limit);
        UniformRealDistribution disC = new UniformRealDistribution(-1, 1);
        dist.reseedRandomGenerator(seed);
        disC.reseedRandomGenerator(seed);

        for (int i = 0; i < numComponents; i++) {
            double alpha = rnd.nextDouble();
            double[] means = dist.sample(dimension);
            double[][] cov = getRandomCovariance(disC);
            MultivariateNormalDistribution multiNorm =

            new MultivariateNormalDistribution(means, cov);
            addMultinormalDistributionToModel(alpha, multiNorm);
        }

        mixtureDistribution = new MixtureMultivariateNormalDistribution(mixture);
        mixtureDistribution.reseedRandomGenerator(seed);

        // calls to sample() will return same results
    }



    /**
     * NOTE this is for adding both internal and external, known distros but
     * need to figure out clean way to add the mixture to mixtureDistribution!!!
     * @param alpha
     * @param dist
     */
    public void addMultinormalDistributionToModel(
    double alpha, MultivariateNormalDistribution dist) {
        // note all alpha will be L1 normed
        mixture.add(new Pair(alpha, dist));
    }



    public double[][] getSimulatedData(int size) {
        return mixtureDistribution.sample(size);
    }



    private double[] getRandomMean(int dimension, double boxSize, long seed) {
```

```
        double limit = boxSize / dimension;

        UniformRealDistribution dist =

            new UniformRealDistribution(-limit, limit);
        dist.reseedRandomGenerator(seed);
        return dist.sample(dimension);
    }


    private double[][] getRandomCovariance(AbstractRealDistribution dist) {
        double[][] data = new double[2*dimension][dimension];
        double determinant = 0.0;
        Covariance cov = new Covariance();
        while(Math.abs(determinant) == 0) {
            for (int i = 0; i < data.length; i++) {
                data[i] = dist.sample(dimension);
            }
            // check if cov matrix is singular ... if so, keep going
            cov = new Covariance(data);
            determinant = new CholeskyDecomposition(
            cov.getCovarianceMatrix()).getDeterminant();

        }
        return cov.getCovarianceMatrix().getData();
    }


}
```

Iris

Iris is the famous dataset containing measurements of irises and three types:

```
package com.datascience.javabook.datasets;


import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;




/**
 * Sentiment-labeled sentences
 * https://archive.ics.uci.edu/ml/datasets/Sentiment+Labelled+Sentences
 * @author mbrzusto
 */
public class IMDB {

    private final List<String> documents = new ArrayList<>();
    private final List<Integer> sentiments = new ArrayList<>();
    private static final String FILEPATH = "datasets/imdb/imdb_labelled.txt";

    public IMDB() throws IOException {
        ClassLoader classLoader = getClass().getClassLoader();
        String filename = classLoader.getResource(FILEPATH).getFile();
        try(BufferedReader br = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = br.readLine()) != null) {
                String[] splitLine = line.split("\t");
                documents.add(splitLine[0]);
                sentiments.add(Integer.parseInt(splitLine[1]));
            }
        }
    }

    public List<Integer> getSentiments() {
        return sentiments;
    }

    public List<String> getDocuments() {
        return documents;
    }
}
```

MNIST

The Modified National Institute of Standards (MNIST) database is the famous handwritten digits dataset of 70,000 images of digits 0 through 9. Of these, 60,000 are in a training set, and 10,000 are in a test set. The first 5,000 are good, and the last 5,000 are hard to read. All data is labeled.

All the integers in the files are stored in the MSB (Most Significant Bit) first (high endian) format used by most non-Intel processors. Users of Intel processors and other low-endian machines must flip the bytes of the header.

There are four files:

- *train-images-idx3-ubyte*: training set images
- *train-labels-idx1-ubyte*: training set labels
- *t10k-images-idx3-ubyte*: test set images
- *t10k-labels-idx1-ubyte*: test set labels

The training set contains 60,000 examples, and the test set 10,000 examples. The first 5,000 examples of the test set are taken from the original MNIST training set. The last 5,000 are taken from the original MNIST test set. The first 5,000 are cleaner and easier to read than the last 5,000.

```
public class MNIST {


    public RealMatrix trainingData;
    public RealMatrix trainingLabels;
    public RealMatrix testingData;
    public RealMatrix testingLabels;

    public MNIST() throws IOException {
        trainingData = new BlockRealMatrix(60000, 784); // image to vector
        trainingLabels = new BlockRealMatrix(60000, 10); // the one hot label
        testingData = new BlockRealMatrix(10000, 784); // image to vector
        testingLabels = new BlockRealMatrix(10000, 10); // the one hot label
        loadData();
```

```
}


private void loadData() throws IOException {
    ClassLoader classLoader = getClass().getClassLoader();
    loadTrainingData(classLoader.getResource(
    "datasets/mnist/train-images-idx3-ubyte").getFile());
    loadTrainingLabels(classLoader.getResource(
    "datasets/mnist/train-labels-idx1-ubyte").getFile());
    loadTestingData(classLoader.getResource(
    "datasets/mnist/t10k-images-idx3-ubyte").getFile());
    loadTestingLabels(classLoader.getResource(
    "datasets/mnist/t10k-labels-idx1-ubyte").getFile());
}


private void loadTrainingData(String filename)

throws FileNotFoundException, IOException {
    try (DataInputStream di = new DataInputStream(

    new BufferedInputStream(new FileInputStream(filename)))) {
        int magicNumber = di.readInt(); //2051
        int numImages = di.readInt(); // 60000
        int numRows = di.readInt(); // 28
        int numCols = di.readInt(); // 28
        for (int i = 0; i < numImages; i++) {
            for (int j = 0; j < 784; j++) {
                // values are 0 to 255, so normalize
                trainingData.setEntry(i, j, di.readUnsignedByte() / 255.0);
            }
        }
    }
}


private void loadTestingData(String filename)

throws FileNotFoundException, IOException {
    try (DataInputStream di = new DataInputStream(

    new BufferedInputStream(new FileInputStream(filename)))) {
        int magicNumber = di.readInt(); //2051
        int numImages = di.readInt(); // 10000
        int numRows = di.readInt(); // 28
        int numCols = di.readInt(); // 28
        for (int i = 0; i < numImages; i++) {
            for (int j = 0; j < 784; j++) {
                // values are 0 to 255, so normalize
                testingData.setEntry(i, j, di.readUnsignedByte() / 255.0);
            }
        }
    }

}


private void loadTrainingLabels(String filename)

throws FileNotFoundException, IOException {
    try (DataInputStream di = new DataInputStream(

    new BufferedInputStream(new FileInputStream(filename)))) {
        int magicNumber = di.readInt(); //2049
        int numImages = di.readInt(); // 60000
        for (int i = 0; i < numImages; i++) {
            // one-hot-encoding, column of 0-9 is given one all else 0
```

```
                trainingLabels.setEntry(i, di.readUnsignedByte(), 1.0);
            }
        }

    }


    private void loadTestingLabels(String filename)

    throws FileNotFoundException, IOException {
        try (DataInputStream di = new DataInputStream(

        new BufferedInputStream(new FileInputStream(filename)))) {
            int magicNumber = di.readInt(); //2049
            int numImages = di.readInt(); // 10000
            for (int i = 0; i < numImages; i++) {
                // one-hot-encoding, column of 0-9 is given one all else 0
                testingLabels.setEntry(i, di.readUnsignedByte(), 1.0);
            }
        }

    }


}
```

Index

About the Author

Michael Brzustowicz, a physicist turned data scientist, specializes in building distributed data systems and extracting knowledge from massive data. He spends most of his time writing customized, multithreaded code for statistical modeling and machine-learning approaches to everyday big data problems. Michael teaches data science at the University of San Francisco.

Colophon

The animal on the cover of *Data Science with Java* is a jack snipe (*Lymnocryptes minimus*), a small wading bird found in coastal areas, marshes, wet meadows, and bogs of Great Britain, Africa, India, and countries near the Mediterranean Sea. They are migratory and breed in northern Europe and Russia.

Jack snipes are the smallest snipe species, at 7–10 inches long and 1.2–2.6 ounces in weight. They have mottled brown feathers, white bellies, and yellow stripes down their backs that are visible during flight. Snipes spend much of their time near bodies of water, walking in shallow water and across mudflats to find food: insects, worms, larvae, plants, and seeds. Their long narrow bills help them extract their meal from the ground.

During courtship, the male jack snipe carries out an aerial display and uses a mating call that sounds somewhat like a galloping horse. The female nests on the ground, laying 3–4 eggs. Due to the camouflage effect of their plumage and well-hidden nesting locations, it can be difficult to observe jack snipes in the wild.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from *Wood's Illustrated Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.