Scala is a multi-paradigm programming and scripting language used to build applications for the JVM. It is particularly good at analyzing large sets of data, and is therefore increasingly popular among developers and data scientists.

This book will introduce you to the libraries for ingesting, storing, manipulating, processing, and visualizing data in Scala. You will learn to read data from flat files and web APIs and store it in a SQL or NoSQL database. This book will show you how to design scalable architectures to process and model your data, starting from simple concurrency constructs, such as parallel collections and futures, through to actor systems and Apache Spark. As well as Scala’s emphasis on functional structures and immutability, you will learn how to use the right parallel construct for the job at hand, minimizing development time without compromising scalability. Finally, you will learn how to build beautiful interactive visualizations using web frameworks.

Who this book is written for

If you are a Scala developer or data scientist, or if you want to enter the field of data science, then this book will give you all the tools you need to implement data science solutions.

What you will learn from this book

- Transform and filter tabular data to extract features for machine learning
- Implement your own algorithms or take advantage of MLLib’s extensive suite of models to build distributed machine learning pipelines
- Read, transform, and write data to both SQL and NoSQL databases in a functional manner
- Read and transform data from web APIs such as GitHub and deploy your own APIs
- Build fault-tolerant concurrent systems using Akka
- Create Scala web applications that couple with Javascript libraries, such as D3, to create compelling interactive visualizations
- Deploy scalable, parallel applications using Apache Spark, loading data from a variety of different inputs

Pascal Bugnion

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'Parallel Collections and Futures'
- A synopsis of the book’s content
- More information on Scala for Data Science
About the Author

Pascal Bugnion is a data engineer at the ASI, a consultancy offering bespoke data science services. Previously, he was the head of data engineering at SCL Elections. He holds a PhD in computational physics from Cambridge University.

Besides Scala, Pascal is a keen Python developer. He has contributed to NumPy, matplotlib and IPython. He also maintains scikit-monom, an open source library for Monte Carlo integration. He currently lives in London, UK.
Preface

Data science is fashionable. Data science startups are sprouting across the globe and established companies are scrambling to assemble data science teams. The ability to analyze large datasets is also becoming increasingly important in the academic and research world.

Why this explosion in demand for data scientists? Our view is that the emergence of data science can be viewed as the serendipitous collusion of several interlinked factors. The first is data availability. Over the last fifteen years, the amount of data collected by companies has exploded. In the world of research, cheap gene sequencing techniques have drastically increased the amount of genomic data available. Social and professional networking sites have built huge graphs interlinking a significant fraction of the people living on the planet. At the same time, the development of the World Wide Web makes accessing this wealth of data possible from almost anywhere in the world.

The increased availability of data has resulted in an increase in data awareness. It is no longer acceptable for decision makers to trust their experience and "gut feeling" alone. Increasingly, one expects business decisions to be driven by data.

Finally, the tools for efficiently making sense of and extracting insights from huge data sets are starting to mature: one doesn't need to be an expert in distributed computing to analyze a large data set any more. Apache Spark, for instance, greatly eases writing distributed data analysis applications. The explosion of cloud infrastructure facilitates scaling computing needs to cope with variable data amounts.

Scala is a popular language for data science. By emphasizing immutability and functional constructs, Scala lends itself well to the construction of robust libraries for concurrency and big data analysis. A rich ecosystem of tools for data science has therefore developed around Scala, including libraries for accessing SQL and NoSQL databases, frameworks for building distributed applications like Apache Spark and libraries for linear algebra and numerical algorithms. We will explore this rich and growing ecosystem in the fourteen chapters of this book.
What this book covers

We aim to give you a flavor for what is possible with Scala, and to get you started using libraries that are useful for building data science applications. We do not aim to provide an entirely comprehensive overview of any of these topics. This is best left to online documentation or to reference books. What we will teach you is how to combine these tools to build efficient, scalable programs, and have fun along the way.

Chapter 1, Scala and Data Science, is a brief description of data science, and of Scala’s place in the data scientist’s tool-belt. We describe why Scala is becoming increasingly popular in data science, and how it compares to alternative languages such as Python.

Chapter 2, Manipulating Data with Breeze, introduces Breeze, a library providing support for numerical algorithms in Scala. We learn how to perform linear algebra and optimization, and solve a simple machine learning problem using logistic regression.

Chapter 3, Plotting with breeze-viz, introduces the breeze-viz library for plotting two-dimensional graphs and histograms.

Chapter 4, Parallel Collections and Futures, describes basic concurrency constructs. We will learn to parallelize simple problems by distributing them over several threads using parallel collections, and apply what we have learned to build a parallel cross-validation pipeline. We then describe how to wrap computation in a future to execute it asynchronously. We apply this pattern to query a web API, sending several requests in parallel.

Chapter 5, Scala and SQL through JDBC, looks at interacting with SQL databases in a functional manner. We learn how to use common Scala patterns to wrap the Java interface exposed by JDBC. Besides learning about JDBC, this chapter introduces type classes, the loan pattern, implicit conversions, and other patterns that are frequently leveraged in libraries and existing Scala code.

Chapter 6, Slick - A Functional Interface for SQL, describes the Slick library for mapping data in SQL tables to Scala objects.

Chapter 7, Web APIs, describes how to query web APIs in a concurrent, fault-tolerant manner using futures. We learn to parse JSON responses and formulate complex HTTP requests with authentication. We walk through querying the GitHub API to obtain information about GitHub users programmatically.

Chapter 8, Scala and MongoDB, walks the reader through interacting with MongoDB, a leading NoSQL database. We build a pipeline that fetches user data from the GitHub API and stores it in a MongoDB database.
Chapter 9, *Concurrency with Akka*, introduces the Akka framework for building concurrent applications with actors. We use Akka to build a scalable crawler that explores the GitHub follower graph.

Chapter 10, *Distributed Batch Processing with Spark*, explores the Apache Spark framework for building distributed applications. We learn how to construct and manipulate distributed datasets in memory. We touch briefly on the internals of Spark, learning how the architecture allows for distributed, fault-tolerant computation.

Chapter 11, *Spark SQL and DataFrames*, describes DataFrames, one of the more powerful features of Spark for the manipulation of structured data. We learn how to load JSON and Parquet files into DataFrames.

Chapter 12, *Distributed Machine Learning with MLlib*, explores how to build distributed machine learning pipelines with MLlib, a library built on top of Apache Spark. We use the library to train a spam filter.

Chapter 13, *Web APIs with Play*, describes how to use the Play framework to build web APIs. We describe the architecture of modern web applications, and how these fit into the data science pipeline. We build a simple web API that returns JSON.

Chapter 14, *Visualization with D3 and the Play Framework*, builds on the previous chapter to program a fully fledged web application with Play and D3. We describe how to integrate JavaScript into a Play framework application.

Appendix, *Pattern Matching and Extractors*, describes how pattern matching provides the programmer with a powerful construct for control flow.
Parallel Collections and Futures

Data science often involves processing medium or large amounts of data. Since the previously exponential growth in the speed of individual CPUs has slowed down and the amount of data continues to increase, leveraging computers effectively must entail parallel computation.

In this chapter, we will look at ways of parallelizing computation and data processing over a single computer. Virtually all new computers have more than one processing unit, and distributing a calculation over these cores can be an effective way of hastening medium-sized calculations.

Parallelizing calculations over a single chip is suitable for calculations involving gigabytes or a few terabytes of data. For larger data flows, we must resort to distributing the computation over several computers in parallel. We will discuss Apache Spark, a framework for parallel data processing in Chapter 10, Distributed Batch Processing with Spark.

In this book, we will look at three common ways of leveraging parallel architectures in a single machine: parallel collections, futures, and actors. We will consider the first two in this chapter, and leave the study of actors to Chapter 9, Concurrency with Akka.

Parallel collections

Parallel collections offer an extremely easy way to parallelize independent tasks. The reader, being familiar with Scala, will know that many tasks can be phrased as operations on collections, such as `map`, `reduce`, `filter`, or `groupBy`. Parallel collections are an implementation of Scala collections that parallelize these operations to run over several threads.
Parallel Collections and Futures

Let's start with an example. We want to calculate the frequency of occurrence of each letter in a sentence:

```scala
scala> val sentence = "The quick brown fox jumped over the lazy dog"
sentence: String = The quick brown fox jumped ...
```

Let's start by converting our sentence from a string to a vector of characters:

```scala
scala> val characters = sentence.toVector
Vector[Char] = Vector(T, h, e, , q, u, i, c, k, ...)
```

We can now convert `characters` to a parallel vector, a `ParVector`. To do this, we use the `par` method:

```scala
scala> val charactersPar = characters.par
ParVector[Char] = ParVector(T, h, e, , q, u, i, c, k, , ...)
```

ParVector collections support the same operations as regular vectors, but their methods are executed in parallel over several threads.

Let's start by filtering out the spaces in `charactersPar`:

```scala
scala> val lettersPar = charactersPar.filter { _ != ' ' }
ParVector[Char] = ParVector(T, h, e, q, u, i, c, k, ...
```

Notice how Scala hides the execution details. The `filter` operation was performed using multiple threads, and you barely even noticed! The interface and behavior of a parallel vector is identical to its serial counterpart, save for a few details that we will explore in the next section.

Let's now use the `toLower` function to make the letters lowercase:

```scala
scala> val lowerLettersPar = lettersPar.map { _.toLower }
ParVector[Char] = ParVector(t, h, e, q, u, i, c, k, ...
```

As before, the `map` method was applied in parallel. To find the frequency of occurrence of each letter, we use the `groupBy` method to group characters into vectors containing all the occurrences of that character:

```scala
scala> val intermediateMap = lowerLettersPar.groupBy(identity)
ParMap[Char,ParVector[Char]] = ParMap(e -> ParVector(e, e, e, e), ...
```
Note how the `groupBy` method has created a `ParMap` instance, the parallel equivalent of an immutable map. To get the number of occurrences of each letter, we do a `mapValues` call on `intermediateMap`, replacing each vector by its length:

```scala
scala> val occurrenceNumber = intermediateMap.mapValues { _.length }
ParMap[Char,Int] = ParMap(e -> 4, x -> 1, n -> 1, j -> 1, ...)
```

Congratulations! We've written a multi-threaded algorithm for finding the frequency of occurrence of each letter in a few lines of code. You should find it straightforward to adapt this to find the frequency of occurrence of each word in a document, a common preprocessing problem for analyzing text data.

Parallel collections make it very easy to parallelize some operation pipelines: all we had to do was call `.par` on the `characters` vector. All subsequent operations were parallelized. This makes switching from a serial to a parallel implementation very easy.

### Limitations of parallel collections

Part of the power and the appeal of parallel collections is that they present the same interface as their serial counterparts: they have a `map` method, a `foreach` method, a `filter` method, and so on. By and large, these methods work in the same way on parallel collections as they do in serial. There are, however, some notable caveats. The most important one has to do with side effects. If an operation on a parallel collection has a side effect, this may result in a race condition: a situation in which the final result depends on the order in which the threads perform their operations.

Side effects in collections arise most commonly when we update a variable defined outside of the collection. To give a trivial example of unexpected behavior, let's define a `count` variable and increment it a thousand times using a parallel range:

```scala
scala> var count = 0
count: Int = 0

scala> (0 until 1000).par.foreach { i => count += 1 }

scala> count
count: Int = 874 // not 1000!
```
What happened here? The function passed to `foreach` has a side effect: it increments `count`, a variable outside of the scope of the function. This is a problem because the `+=` operator is a sequence of two operations:

- Retrieve the value of `count` and add one to it
- Assign the result back to `count`

To understand why this causes unexpected behavior, let's imagine that the `foreach` loop has been parallelized over two threads. **Thread A** might read the `count` variable when it is 832 and add one to it to give 833. Before it has time to reassign 833 to `count`, **Thread B** reads `count`, still at 832, and adds one to give 833. **Thread A** then assigns 833 to `count`. **Thread B** then assigns 833 to `count`. We've run through two updates but only incremented the count by one. The problem arises because `+=` can be separated into two instructions: it is not atomic. This leaves room for threads to interleave their operations:

![The anatomy of a race condition: both thread A and thread B are trying to update count concurrently, resulting in one of the updates being overwritten. The final value of count is 833 instead of 834.](image)

To give a somewhat more realistic example of problems caused by non-atomicity, let's look at a different method for counting the frequency of occurrence of each letter in our sentence. We define a mutable `Char -> Int` hash map outside of the loop. Each time we encounter a letter, we increment the corresponding integer in the map:

```scala
scala> import scala.collection.mutable
import scala.collection.mutable

scala> val occurrenceNumber = mutable.Map.empty[Char, Int]
```
occurrenceNumber: mutable.Map[Char, Int] = Map()

scala> lowerLettersPar.foreach { c =>
    occurrenceNumber(c) = occurrenceNumber.getOrElse(c, 0) + 1
}

scala> occurrenceNumber('e') // Should be 4
Int = 2

The discrepancy occurs because of the non-atomicity of the operations in the foreach loop.

In general, it is good practice to avoid side effects in higher-order functions on collections. They make the code harder to understand and preclude switching from serial to parallel collections. It is also good practice to avoid exposing mutable state: immutable objects can be shared freely between threads and cannot be affected by side effects.

Another limitation of parallel collections occurs in reduction (or folding) operations. The function used to combine items together must be associative. For instance:

scala> (0 until 1000).par.reduce { _ - _ } // should be -499500
Int = 63620

The minus operator, -, is not associative. The order in which consecutive operations are applied matters: \((a - b) - c\) is not the same as \(a - (b - c)\). The function used to reduce a parallel collection must be associative because the order in which the reduction occurs is not tied to the order of the collection.

**Error handling**

In single-threaded programs, exception handling is relatively straightforward: if an exception occurs, the function can either handle it or escalate it. This is not nearly as obvious when parallelism is introduced: a single thread might fail, but the others might return successfully.

Parallel collection methods will throw an exception if they fail on any element, just like their serial counterparts:

scala> Vector(2, 0, 5).par.map { 10 / _ }
java.lang.ArithmeticException: / by zero
...
There are cases when this isn’t the behavior that we want. For instance, we might be using a parallel collection to retrieve a large number of web pages in parallel. We might not mind if a few of the pages cannot be fetched.

Scala’s `Try` type was designed for sandboxing code that might throw exceptions. It is similar to `Option` in that it is a one-element container:

```scala
import scala.util._
import scala.util._

scala> Try { 2 + 2 }
Try[Int] = Success(4)
```

Unlike the `Option` type, which indicates whether an expression has a useful value, the `Try` type indicates whether an expression can be executed without throwing an exception. It takes on the following two values:

- `Try { 2 + 2 } == Success(4)` if the expression in the `Try` statement is evaluated successfully
- `Try { 2 / 0 } == Failure(java.lang.ArithmeticException: / by zero)` if the expression in the `Try` block results in an exception

This will make more sense with an example. To see the `Try` type in action, we will try to fetch web pages in a fault tolerant manner. We will use the built-in `Source.fromURL` method which fetches a web page and opens an iterator of the page's content. If it fails to fetch the web page, it throws an error:

```scala
import scala.io.Source
import scala.io.Source

scala> val html = Source.fromURL("http://www.google.com")
scala.io.BufferedSource = non-empty iterator

scala> val html = Source.fromURL("garbage")
java.net.MalformedURLException: no protocol: garbage
...
```

Instead of letting the expression propagate out and crash the rest of our code, we can wrap the call to `Source.fromURL` in `Try`:

```scala
scala> Try { Source.fromURL("http://www.google.com") }
```
Try[BufferedSource] = Success(non-empty iterator)

scala> Try { Source.fromURL("garbage") }
Try[BufferedSource] = Failure(java.net.MalformedURLException: no protocol: garbage)

To see the power of our Try statement, let's now retrieve a list of URLs in parallel in a fault tolerant manner:

scala> val URLs = Vector("http://www.google.com",
   "http://www.bbc.co.uk",
   "not-a-url"
)

scala> val pages = URLs.par.map { url =>
   url -> Try { Source.fromURL(url) }
}
pages: ParVector[(String, Try[BufferedSource])] = ParVector((http://www.google.com,Success(non-empty iterator)), (http://www.bbc.co.uk,Success(non-empty iterator)), (not-a-url,Failure(java.net.MalformedURLException: no protocol: not-a-url)))

We can then use a collect statement to act on the pages we could fetch successfully. For instance, to get the number of characters on each page:

scala> pages.collect { case(url, Success(it)) => url -> it.size }
ParVector[(String, Int)] = ParVector((http://www.google.com,18976),
(http://www.bbc.co.uk,132893))

By making good use of Scala's built-in Try classes and parallel collections, we have built a fault tolerant, multithreaded URL retriever in a few lines of code. (Compare this to the myriad of Java/C++ books that prefix code examples with 'error handling is left out for clarity'.)
The Try type versus try/catch statements
Programmers with imperative or object-oriented backgrounds will be more familiar with try/catch blocks for handling exceptions. We could have accomplished similar functionality here by wrapping the code for fetching URLs in a try block, returning null if the call raises an exception. However, besides being more verbose, returning null is less satisfactory: we lose all information about the exception and null is less expressive than Failure(exception). Furthermore, returning a Try[T] type forces the caller to consider the possibility that the function might fail, by encoding this possibility in the type of the return value. In contrast, just returning T and coding failure with a null value allows the caller to ignore failure, raising the possibility of a confusing NullPointerException being thrown at a completely different point in the program.

In short, Try[T] is just another higher-order type, like Option[T] or List[T]. Treating the possibility of failure in the same way as the rest of the code adds coherence to the program and encourages programmers to tackle the possibility of exceptions explicitly.

Setting the parallelism level
So far, we have considered parallel collections as black boxes: add par to a normal collection and all the operations are performed in parallel. Often, we will want more control over how the tasks are executed.

Internally, parallel collections work by distributing an operation over multiple threads. Since the threads share memory, parallel collections do not need to copy any data. Changing the number of threads available to the parallel collection will change the number of CPUs that are used to perform the tasks.

Parallel collections have a tasksupport attribute that controls task execution:

```
scala> val parRange = (0 to 100).par
parRange: ParRange = ParRange(0, 1, 2, 3, 4, 5,...

scala> parRange.tasksupport
TaskSupport = scala.collection.parallel.ExecutionContextTaskSupport@311a0b3e

scala> parRange.tasksupport.parallelismLevel
Int = 8 // Number of threads to be used
```
The task support object of a collection is an execution context, an abstraction capable of executing Scala expressions in a separate thread. By default, the execution context in Scala 2.11 is a work-stealing thread pool. When a parallel collection submits tasks, the context allocates these tasks to its threads. If a thread finds that it has finished its queued tasks, it will try and steal outstanding tasks from the other threads. The default execution context maintains a thread pool with number of threads equal to the number of CPUs.

The number of threads over which the parallel collection distributes the work can be changed by changing the task support. For instance, to parallelize the operations performed by a range over four threads:

```
scala> import scala.collection.parallel._
import scala.collection.parallel._

scala> parRange.tasksupport = new ForkJoinTaskSupport(
    new scala.concurrent.forkjoin.ForkJoinPool(4)
)
parRange.tasksupport: scala.collection.parallel.TaskSupport = scala.collection.parallel.ForkJoinTaskSupport@6e1134e1

scala> parRange.tasksupport.parallelismLevel
Int: 4
```

### An example – cross-validation with parallel collections

Let's apply what you have learned so far to solve data science problems. There are many parts of a machine learning pipeline that can be parallelized trivially. One such part is cross-validation.

We will give a brief description of cross-validation here, but you can refer to *The Elements of Statistical Learning*, by Hastie, Tibshirani, and Friedman for a more in-depth discussion.

Typically, a supervised machine learning problem involves training an algorithm over a training set. For instance, when we built a model to calculate the probability of a person being male based on their height and weight, the training set was the (height, weight) data for each participant, together with the male/female label for each row. Once the algorithm is trained on the training set, we can use it to classify new data. This process only really makes sense if the training set is representative of the new data that we are likely to encounter.
Parallel Collections and Futures

The training set has a finite number of entries. It will thus, inevitably, have idiosyncrasies that are not representative of the population at large, merely due to its finite nature. These idiosyncrasies will result in prediction errors when predicting whether a new person is male or female, over and above the prediction error of the algorithm on the training set itself. Cross-validation is a tool for estimating the error caused by the idiosyncrasies of the training set that do not reflect the population at large.

Cross-validation works by dividing the training set in two parts: a smaller, new training set and a cross-validation set. The algorithm is trained on the reduced training set. We then see how well the algorithm models the cross-validation set. Since we know the right answer for the cross-validation set, we can measure how well our algorithm is performing when shown new information. We repeat this procedure many times with different cross-validation sets.

There are several different types of cross-validation, which differ in how we choose the cross-validation set. In this chapter, we will look at repeated random subsampling: we select $k$ rows at random from the training data to form the cross-validation set. We do this many times, calculating the cross-validation error for each subsample. Since each iteration is independent of the previous ones, we can parallelize this process trivially. It is therefore a good candidate for parallel collections. We will look at an alternative form of cross-validation, $k$-fold cross-validation, in Chapter 12, Distributed Machine Learning with MLlib.

We will build a class that performs cross-validation in parallel. I encourage you to write the code as you go, but you will find the source code corresponding to these examples on GitHub (https://github.com/pbugnion/s4ds). We will use parallel collections to handle the parallelism and Breeze data types in the inner loop. The build.sbt file is identical to the one we used in Chapter 2, Manipulating Data with Breeze:

```scala
scalaVersion := "2.11.7"

libraryDependencies ++= Seq(
  "org.scalanlp" %% "breeze" % "0.11.2",
  "org.scalanlp" %% "breeze-natives" % "0.11.2"
)
```

We will build a RandomSubsample class. The class exposes a type alias, CVFunction, for a function that takes two lists of indices—the first corresponding to the reduced training set and the second to the validation set—and returns a Double corresponding to the cross-validation error:

```scala
type CVFunction = (Seq[Int], Seq[Int]) => Double
```
The RandomSubsample class will expose a single method, \texttt{mapSamples}, which takes a \texttt{CVFunction}, repeatedly passes it different partitions of indices, and returns a vector of the errors. This is what the class looks like:

```scala
// RandomSubsample.scala
import breeze.linalg._
import breeze.numerics._

/**
 * Random subsample cross-validation
 *
 * @param nElems Total number of elements in the training set.
 * @param nCrossValidation Number of elements to leave out of training set.
 */
class RandomSubsample(val nElems:Int, val nCrossValidation:Int) {
    type CVFunction = (Seq[Int], Seq[Int]) => Double

    require(nElems > nCrossValidation,
        "nCrossValidation, the number of elements " +
        "withheld, must be < nElems")

    private val indexList = DenseVector.range(0, nElems)

    /**
     * Perform multiple random sub-sample CV runs on \texttt{f}
     *
     * @param nShuffles Number of random sub-sample runs.
     * @param f user-defined function mapping from a list of indices
     * in the training set and a list of indices in the test-set to
     * a double indicating the out-of-sample score for this split.
     * @returns DenseVector of the CV error for each random split.
     */
    def mapSamples(nShuffles:Int)(f:CVFunction):DenseVector[Double] = {
        val cvResults = (0 to nShuffles).par.map { i =>

            // Randomly split indices between test and training
            val shuffledIndices = breeze.linalg.shuffle(indexList)
            val Seq(testIndices, trainingIndices) =
                split(shuffledIndices, Seq(nCrossValidation))

            // Apply f for this split

        }.
    }
}
```
Let's look at what happens in more detail, starting with the arguments passed to the constructor:

```scala
class RandomSubsample(val nElems:Int, val nCrossValidation:Int)
```

We pass the total number of elements in the training set and the number of elements to leave out for cross-validation in the class constructor. Thus, passing 100 to `nElems` and 20 to `nCrossValidation` implies that our training set will have 80 random elements of the total data and that the test set will have 20 elements.

We then construct a list of all integers between 0 and `nElems`:

```scala
private val indexList = DenseVector.range(0, nElems)
```

For each iteration of the cross-validation, we will shuffle this list and take the first `nCrossValidation` elements to be the indices of rows in our test set and the remaining to be the indices of rows in our training set.

Our class exposes a single method, `mapSamples`, that takes two curried arguments: `nShuffles`, the number of times to perform random subsampling, and `f`, a `CVFunction`:

```scala
def mapSamples(nShuffles:Int)(f:CVFunction):DenseVector[Double]
```

With all this set up, the code for doing cross-validation is deceptively simple. We generate a parallel range from 0 to `nShuffles` and, for each item in the range, generate a new train-test split and calculate the cross-validation error:

```scala
val cvResults = (0 to nShuffles).par.map { i =>
  val shuffledIndices = breeze.linalg.shuffle(indexList)
  val Seq(testIndices, trainingIndices) = split(shuffledIndices, Seq(nCrossValidation))
  f(trainingIndices.toScalaVector, testIndices.toScalaVector)
}
```

The only tricky part of this function is splitting the shuffled index list into a list of indices for the training set and a list of indices for the test set. We use Breeze's `split` method. This takes a vector as its first argument and a list of split-points as its second, and returns a list of fragments of the original vector. We then use pattern matching to extract the individual parts.
Finally, mapSamples converts cvResults to a Breeze vector:

DenseVector(cvResults.toArray)

Let's see this in action. We can test our class by running cross-validation on the logistic regression example developed in Chapter 2, Manipulating Data with Breeze. In that chapter, we developed a LogisticRegression class that takes a training set (in the form of a DenseMatrix) and target (in the form of a DenseVector) at construction time. The class then calculates the parameters that best represent the training set. We will first add two methods to the LogisticRegression class to use the trained model to classify previously unseen examples:

- The predictProbabilitiesMany method uses the trained model to calculate the probability of having the target variable set to one. In the context of our example, this is the probability of being male, given a height and weight.
- The classifyMany method assigns classification labels (one or zero) to members of a test set. We will assign a one if predictProbabilitiesMany returns a value greater than 0.5.

With these two functions, our LogisticRegression class becomes:

```scala
// Logistic Regression.scala
class LogisticRegression(
  val training:DenseMatrix[Double],
  val target:DenseVector[Double]
) {
  ...  
  /** Probability of classification for each row in test set. */
  def predictProbabilitiesMany(test:DenseMatrix[Double]) :DenseVector[Double] = {
    val xBeta = test * optimalCoefficients
    sigmoid(xBeta)
  }

  /** Predict the value of the target variable for each row in test set. */
  def classifyMany(test:DenseMatrix[Double]) :DenseVector[Double] = {
    val probabilities = predictProbabilitiesMany(test)
    I((probabilities :> 0.5).toDenseVector)
  }
  ...  
}
```
We can now put together an example program for our RandomSubsample class. We will use the same height-weight data as in Chapter 2, Manipulating Data with Breeze. The data preprocessing will be similar. The code examples for this chapter provide a helper module, HWData, to load the height-weight data into Breeze vectors. The data itself is in the data/ directory of the code examples for this chapter (available on GitHub at https://github.com/pbugnion/s4ds/tree/master/chap04).

For each new subsample, we create a new LogisticRegression instance, train it on the subset of the training set to get the best coefficients for this train-test split, and use classifyMany to generate predictions on the cross-validation set in this split. We then calculate the classification error and report the average classification error over every train-test split:

```
// RandomSubsampleDemo.scala

import breeze.linalg._
import breeze.linalg.functions.manhattanDistance
import breeze.numerics._
import breeze.stats._

object RandomSubsampleDemo extends App {

  /* Load and pre-process data */
  val data = HWData.load

  val rescaledHeights:DenseVector[Double] =
    (data.heights - mean(data.heights)) / stddev(data.heights)

  val rescaledWeights:DenseVector[Double] =
    (data.weights - mean(data.weights)) / stddev(data.weights)

  val featureMatrix:DenseMatrix[Double] =
    DenseMatrix.horzcat(
      DenseMatrix.ones[Double](data.npoints, 1),
      rescaledHeights.toDenseMatrix.t,
      rescaledWeights.toDenseMatrix.t
    )

  val target:DenseVector[Double] = data.genders.values.map { gender =>
    if(gender == 'M') 1.0 else 0.0
  }

  /* Cross-validation */
  val testSize = 20
```
val cvCalculator = new RandomSubsample(data.npoints, testSize)

// Start parallel CV loop
val cvErrors = cvCalculator.mapSamples(1000) {
  (trainingIndices, testIndices) =>

  val regressor = new LogisticRegression(
    data.featureMatrix(trainingIndices, ::).toDenseMatrix,
    data.target(trainingIndices).toDenseVector
  )

  // Predictions on test-set
  val genderPredictions = regressor.classifyMany(
    data.featureMatrix(testIndices, ::).toDenseMatrix
  )

  // Calculate number of mis-classified examples
  val dist = manhattanDistance(
    genderPredictions, data.target(testIndices)
  )

  // Calculate mis-classification rate
  dist / testSize.toDouble
}

println(s"Mean classification error: ${mean(cvErrors)}")

Running this program on the height-weight data gives a classification error of 10%.

We now have a fully working, parallelized cross-validation class. Scala's parallel range made it simple to repeatedly compute the same function in different threads.

**Futures**

Parallel collections offer a simple, yet powerful, framework for parallel operations. However, they are limited in one respect: the total amount of work must be known in advance, and each thread must perform the same function (possibly on different inputs).

Imagine that we want to write a program that fetches a web page (or queries a web API) every few seconds and extracts data for further processing from this web page. A typical example might involve querying a web API to maintain an up-to-date value of a particular stock price. Fetching data from an external web page takes a few hundred milliseconds, typically. If we perform this operation on the main thread, it will needlessly waste CPU cycles waiting for the web server to reply.
The solution is to wrap the code for fetching the web page in a future. A future is a one-element container containing the future result of a computation. When you create a future, the computation in it gets off-loaded to a different thread in order to avoid blocking the main thread. When the computation finishes, the result is written to the future and thus made accessible to the main thread.

As an example, we will write a program that queries the "Markit on demand" API to fetch the price of a given stock. For instance, the URL for the current price of a Google share is http://dev.markitondemand.com/MODApis/Api/v2/Quote?symbol=GOOG. Go ahead and paste this in the address box of your web browser. You will see an XML string appear with, among other things, the current stock price. Let's fetch this programmatically without resorting to a future first:

```scala
scala> import scala.io._
import scala.io_

scala> val url = "http://dev.markitondemand.com/MODApis/Api/v2/Quote?symbol=GOOG"
url: String = http://dev.markitondemand.com/MODApis/Api/v2/Quote?symbol=GOOG

scala> val response = Source.fromURL(url).mkString
response: String = <StockQuote><Status>SUCCESS</Status>...
```

Notice how it takes a little bit of time to query the API. Let's now do the same, but using a future (don't worry about the imports for now, we will discuss what they mean in more detail further on):

```scala
scala> import scala.concurrent._
import scala.concurrent._

scala> import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.ExecutionContext.Implicits.global

scala> val response = Future { Source.fromURL(url).mkString }
response: Future[String] = Promise$DefaultPromise@3301801b
```
If you run this, you will notice that control returns to the shell instantly before the API has had a chance to respond. To make this evident, let's simulate a slow connection by adding a call to `Thread.sleep`:

```scala
scala> val response = Future {
    Thread.sleep(10000) // sleep for 10s
    Source.fromURL(url).mkString
}
```

When you run this, you do not have to wait for ten seconds for the next prompt to appear: you regain control of the shell straightaway. The bit of code in the future is executed asynchronously: its execution is independent of the main program flow.

How do we retrieve the result of the computation? We note that `response` has type `Future[String]`. We can check whether the computation wrapped in the future has finished by querying the future's `isCompleted` attribute:

```scala
scala> response.isCompleted
Boolean = true
```

The future exposes a `value` attribute that contains the computation result:

```scala
scala> response.value
Option[Try[String]] = Some(Success(<StockQuote><Status>SUCCESS</Status>...
```

The `value` attribute of a future has type `Option[Try[T]]`. We have already seen how to use the `Try` type to handle exceptions gracefully in the context of parallel collections. It is used in the same way here. A future's `value` attribute is `None` until the future is complete, then it is set to `Some(Success(value))` if the future ran successfully, or `Some(Failure(error))` if an exception was thrown.

Repeatedly calling `f.value` until the future completes works well in the shell, but it does not generalize to more complex programs. Instead, we want to tell the computer to do something once the future is complete: we want to bind a `callback` function to the future. We can do this by setting the future's `onComplete` attribute. Let's tell the future to print the API response when it completes:

```scala
scala> response.onComplete {
    case Success(s) => println(s)
```
Parallel Collections and Futures

```scala
case Failure(e) => println(s"Error fetching page: $e")
```

```
scala>
// Wait for response to complete, then prints:
<StockQuote><Status>SUCCESS</Status><Name>Alphabet Inc</Name><Symbol>GOOGL</Symbol><LastPrice>695.22</LastPrice><Chan...
```

The function passed to `onComplete` runs when the future is finished. It takes a single argument of type `Try[T]` containing the result of the future.

**Failure is normal: how to build resilient applications**

By wrapping the output of the code that it runs in a `Try` type, futures force the client code to consider the possibility that the code might fail. The client can isolate the effect of failure to avoid crashing the whole application. They might, for instance, log the exception. In the case of a web API query, they might add the offending URL to be queried again at a later date. In the case of a database failure, they might roll back the transaction.

By treating failure as a first-class citizen rather than through exceptional control flow bolted on at the end, we can build applications that are much more resilient.

**Future composition – using a future's result**

In the previous section, you learned about the `onComplete` method to bind a callback to a future. This is useful to cause a side effect to happen when the future is complete. It does not, however, let us transform the future's return value easily.

To carry on with our stocks example, let's imagine that we want to convert the query response from a string to an XML object. Let's start by including the `scala-xml` library as a dependency in `build.sbt`:

```scala
libraryDependencies += "org.scala-lang" % "scala-xml" % "2.11.0-M4"
```

Let's restart the console and reimport the dependencies on `scala.concurrent._`, `scala.concurrent.ExecutionContext.Implicits.global`, and `scala.io._`. We also want to import the `XML` library:

```scala
scala> import scala.xml.XML
import scala.xml.XML
```
We will use the same URL as in the previous section:

http://dev.markitondemand.com/MODApis/Api/v2/Quote?symbol=GOOG

It is sometimes useful to think of a future as a collection that either contains one element if a calculation has been successful, or zero elements if it has failed. For instance, if the web API has been queried successfully, our future contains a string representation of the response. Like other container types in Scala, futures support a `map` method that applies a function to the element contained in the future, returning a new future, and does nothing if the calculation in the future failed. But what does this mean in the context of a computation that might not be finished yet? The `map` method gets applied as soon as the future is complete, like the `onComplete` method.

We can use the future's `map` method to apply a transformation to the result of the future asynchronously. Let's poll the "Markit on demand" API again. This time, instead of printing the result, we will parse it as XML.

```scala
scala> val strResponse = Future {
    |   Thread.sleep(20000) // Sleep for 20s
    |   val res = Source.fromURL(url).mkString
    |   println("finished fetching url")
    |   res
    | }
strResponse: Future[String] = Promise$DefaultPromise@1dda9bc8

scala> val xmlResponse = strResponse.map { s =>
    |   println("applying string to xml transformation")
    |   XML.loadString(s)
    | }
xmlResponse: Future[xml.Elem] = Promise$DefaultPromise@25d1262a

// wait while the remainder of the 20s elapses
finished fetching url
applying string to xml transformation

scala> xmlResponse.value
Option[Try[xml.Elem]] = Some(Success(<StockQuote><Status>SUCCESS</Status>...)
```

By registering subsequent maps on futures, we are providing a road map to the executor running the future for what to do.
If any of the steps fail, the failed `Try` instance containing the exception gets propagated instead:

```
scala> val strResponse = Future {
    | Source.fromURL("empty").mkString
  }
```

```
scala> val xmlResponse = strResponse.map {
    | s => XML.loadString(s)
  }
```

```
scala> xmlResponse.value
Option[Try[xml.Elem]] = Some(Failure(MalformedURLException: no protocol: empty))
```

This behavior makes sense if you think of a failed future as an empty container. When applying a map to an empty list, it returns the same empty list. Similarly, when applying a map to an empty (failed) future, the empty future is returned.

**Blocking until completion**

The code for fetching stock prices works fine in the shell. However, if you paste it in a standalone program, you will notice that nothing gets printed and the program finishes straightaway. Let’s look at a trivial example of this:

```
// BlockDemo.scala
import scala.concurrent._
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._

object BlockDemo extends App {
  val f = Future { Thread.sleep(10000) }
  f.onComplete { _ => println("future completed") }
  // "future completed" is not printed

  // BlockDemo.scala
  import scala.concurrent._
  import scala.concurrent.ExecutionContext.Implicits.global
  import scala.concurrent.duration._

  object BlockDemo extends App {
    val f = Future { Thread.sleep(10000) }
    f.onComplete { _ => println("future completed") }
    // "future completed" is not printed
  }
```

The program stops running as soon as the main thread has completed its tasks, which, in this example, just involves creating the futures. In particular, the line "future completed" is never printed. If we want the main thread to wait for a future to execute, we must explicitly tell it to block execution until the future has finished running. This is done using the `Await.ready` or `Await.result` methods. Both these methods block the execution of the main thread until the future completes. We could make the above program work as intended by adding this line:

```scala
Await.ready(f, 1 minute)
```

The `Await` methods take the future as their first argument and a `Duration` object as the second. If the future takes longer to complete than the specified duration, a `TimeoutException` is thrown. Pass `Duration.Inf` to set an infinite timeout.

The difference between `Await.ready` and `Await.result` is that the latter returns the value inside the future. In particular, if the future resulted in an exception, that exception will get thrown. In contrast, `Await.ready` returns the future itself.

In general, one should try to avoid blocking as much as possible: the whole point of futures is to run code in background threads in order to keep the main thread of execution responsive. However, a common, legitimate use case for blocking is at the end of a program. If we are running a large-scale integration process, we might dispatch several futures to query web APIs, read from text files, or insert data into a database. Embedding the code in futures is more scalable than performing these operations sequentially. However, as the majority of the intensive work is running in background threads, we are left with many outstanding futures when the main thread completes. It makes sense, at this stage, to block until all the futures have completed.

**Controlling parallel execution with execution contexts**

Now that we know how to define futures, let's look at controlling how they run. In particular, you might want to control the number of threads to use when running a large number of futures.

When a future is defined, it is passed an `execution context`, either directly or implicitly. An execution context is an object that exposes an `execute` method that takes a block of code and runs it, possibly asynchronously. By changing the execution context, we can change the "backend" that runs the futures. We have already seen how to use execution contexts to control the execution of parallel collections.

So far, we have just been using the default execution context by importing `scala.concurrent.ExecutionContext.Implicits.global`. This is a fork / join thread pool with as many threads as there are underlying CPUs.
Let’s now define a new execution context that uses sixteen threads:

```scala
scala> import java.util.concurrent.Executors
import java.util.concurrent.Executors

scala> val ec = ExecutionContext.fromExecutorService(
      Executors.newFixedThreadPool(16)
)
```

Having defined the execution context, we can pass it explicitly to futures as they are defined:

```scala
scala> val f = Future { Thread.sleep(1000) } (ec)
f: Future[Unit] = Promise$DefaultPromise@458b456
```

Alternatively, we can define the execution context implicitly:

```scala
scala> implicit val context = ec
context: ExecutionContextExecutorService = ExecutionContextImpl$$anon$1@1351ce60
```

Futures example – stock price fetcher

Let’s bring some of the concepts that we covered in this section together to build a command-line application that prompts the user for the name of a stock and fetches the value of that stock. The catch is that, to keep the UI responsive, we will fetch the stock using a future:

```scala
// StockPriceDemo.scala

import scala.concurrent._
```
import scala.concurrent.ExecutionContext.Implicits.global
import scala.io._
import scala.xml.XML
import scala.util._

object StockPriceDemo extends App {

  /* Construct URL for a stock symbol */
  def urlFor(stockSymbol:String) =
    ("http://dev.markitondemand.com/MODApis/Api/v2/Quote?" +
    s"symbol=${stockSymbol}")

  /* Build a future that fetches the stock price */
    val url = urlFor(stockSymbol)
    val strResponse = Future { Source.fromURL(url).mkString }
    val xmlResponse = strResponse.map { s => XML.loadString(s) }
    val price = xmlResponse.map {
      r => BigDecimal((r \\ "LastPrice").text)
    }
    price
  }

  /* Command line interface */
  println("Enter symbol at prompt.")
  while (true) {
    val symbol = readLine("> ") // Wait for user input
    // When user puts in symbol, fetch data in background
    // thread and print to screen when complete
    fetchStockPrice(symbol).onComplete { res =>
      println()
      res match {
        case Success(price) => println(s"$symbol: USD $price")
        case Failure(e) => println(s"Error fetching $symbol: $e")
      }
    print("> ") // Simulate the appearance of a new prompt
  }
}
Try running the program and entering the code for some stocks:

```
[info] Running StockPriceDemo
Enter symbol at prompt:
> GOOG
> MSFT
>
GOOG: USD 695.22
>
MSFT: USD 47.48
>
AAPL: USD 111.01
```

Let's summarize how the code works. When you enter a stock, the main thread constructs a future that fetches the stock information from the API, converts it to XML, and extracts the price. We use `(r \ "LastPrice") .text` to extract the text inside the `LastPrice` tag from the XML node `r`. We then convert the value to a big decimal. When the transformations are complete, the result is printed to screen by binding a callback through `onComplete`. Exception handling is handled naturally through our use of `.map` methods to handle transformations.

By wrapping the code for fetching a stock price in a future, we free up the main thread to just respond to the user. This means that the user interface does not get blocked if we have, for instance, a slow internet connection.

This example is somewhat artificial, but you could easily wrap much more complicated logic: stock prices could be written to a database and we could add additional commands to plot the stock price over time, for instance.

We have only scratched the surface of what futures can offer in this section. We will revisit futures in more detail when we look at polling web APIs in Chapter 7, `Web APIs` and Chapter 9, `Concurrency with Akka`.

Futures are a key part of the data scientist's toolkit for building scalable systems. Moving expensive computation (either in terms of CPU time or wall time) to background threads improves scalability greatly. For this reason, futures are an important part of many Scala libraries such as `Akka` and the `Play` framework.
Summary

By providing high-level concurrency abstractions, Scala makes writing parallel code intuitive and straightforward. Parallel collections and futures form an invaluable part of a data scientist's toolbox, allowing them to parallelize their code with minimal effort. However, while these high-level abstractions obviate the need to deal directly with threads, an understanding of the internals of Scala's concurrency model is necessary to avoid race conditions.

In the next chapter, we will put concurrency on hold and study how to interact with SQL databases. However, this is only temporary: futures will play an important role in many of the remaining chapters in this book.

References

Aleksandar Prokopec, Learning Concurrent Programming in Scala. This is a detailed introduction to the basics of concurrent programming in Scala. In particular, it explores parallel collections and futures in much greater detail than this chapter.

Daniel Westheide's blog gives an excellent introduction to many Scala concepts, in particular:

- **Futures**: http://danielwestheide.com/blog/2013/01/09/the-neophytes-guide-to-scala-part-8-welcome-to-the-future.html
- **The Try type**: http://danielwestheide.com/blog/2012/12/26/the-neophytes-guide-to-scala-part-6-error-handling-with-try.html

For a discussion of cross-validation, see *The Elements of Statistical Learning* by Hastie, Tibshirani, and Friedman.
Where to buy this book
You can buy Scala for Data Science from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.