Statistical Computing with Scala:
A functional approach to data science

Darren J Wilkinson
`@darrenjw`

Notes for the short course:

Scala for statistical computing and data science
`https://github.com/darrenjw/scala-course`

`#scscala`

Please cite these notes as:

Wilkinson, D. J. (2017) *Statistical Computing with Scala: A functional approach to data science*, `https://github.com/darrenjw/scala-course`

Last update:

May 31, 2017

These notes are currently under active development. Use the date above to check when this file was generated.

Contents

List of Figures

Part I

Scala

Chapter 1

Introduction

1.1 Course outline

This course is aimed at statisticians and data scientists already familiar with a dynamic programming language (such as R, Python or Octave) who would like to learn how to use Scala. Scala is a free modern, powerful, strongly-typed, functional programming language, well-suited to statistical computing and data science applications. In particular, it is fast and efficient, runs on the Java virtual machine (JVM), and is designed to easily exploit modern multi-core and distributed computing architectures.

`www.scala-lang.org`

The course will begin with an introduction to the Scala language and basic concepts of functional programming (FP), as well as essential Scala tools such as SBT for managing builds and library dependencies. The course will continue with an overview of the Scala collections library, including parallel collections, and we will see how parallel collections enable trivial parallelisation of many statistical computing algorithms on multi-core hardware. We will next survey the wider Scala library ecosystem, paying particular attention to Breeze, the Scala library for scientific computing and numerical linear algebra. We will see how to exploit non-uniform random number generation and matrix computations in Breeze for statistical applications. Both maximum-likelihood and simulation-based Bayesian statistical inference algorithms will be considered. Much of the final day will be dedicated to understanding Apache Spark, the distributed Big Data analytics platform for Scala. We will understand how Spark relates to the parallel collections we have already examined, and see how it can be used not only for the processing of very large data sets, but also for the parallel and distributed analysis of large or otherwise computationally-intensive models. As time permits, we will discuss more advanced FP concepts, such as type-classes, higher-kinded types, monoids, functors, monads, applicatives, streams and streaming data, and see how these enable the development of flexible, scalable, generic code in strongly-typed functional languages.

1.1.1 Prerequisite

The course assumes a basic familiarity with essential concepts in statistical computing, as well as some basic programming experience. It is assumed that participants will be familiar with writing their own functions in a language such as **R**, including essential control structures such as “for-loops” and “if-statements”. The course is not suitable for people completely new to programming. However, no prior knowledge of Scala or functional programming is assumed. All participants will be expected to bring their own (multi-core) laptop and to have a recent version of Java pre-installed. Other set-up instructions will be provided in advance to registered participants.

The course website and code repository can be found at `github.com/darrenjw/scala-course`

1.1.2 Course structure

The course will be delivered through a combination of lectures, live demos and hands-on practical sessions. For the practical sessions, participants will be expected to actively engage with the material, run demos, follow examples, and write code to solve simple problems.

1.1.3 Presenters

The course will be delivered by Prof Darren Wilkinson (Newcastle University, U.K.). Prof Wilkinson is co-Director of Newcastle’s EPSRC Centre for Doctoral Training in Cloud Computing for Big Data. He is a well-known expert in computational Bayesian statistics and a leading proponent of the use of strongly-typed FP languages (such as Scala) for scalable statistical computing.

1.2 Introduction to Scala and functional programming

1.2.1 Statistical computing

- The current state of serious statistical computing is far from ideal...
- **R** has become the de facto standard programming language for statistical computing — the S language was designed by statisticians for statisticians in the mid 1970’s, and it shows!
  - Many dubious language design choices, meaning it will always be inelegant, slow, dangerous and inefficient (without many significant breaking changes to the language)
  - R’s inherent inefficiencies mean that much of the R codebase isn’t in R at all, but instead in other languages, such as *FORTRAN*, *C* and *C++*
  - Although faster and more efficient than R, these languages are actually all even *worse* languages for statistical computing than R!

- The fundamental problem is that all of the programming languages commonly used for scientific and statistical computing were designed 30-50 years ago, in the dawn of the computing age, and haven't significantly changed
  - Think how much computing *hardware* has changed in the last 40 years!
  - But the languages that most people are using were designed for that hardware using the knowledge of programming languages that existed at that time
- We have learned just as much about programming and programming languages in the last 40 years as we have about everything else (including statistical methodology)
- Our understanding has developed in parallel with developments in hardware
- People have been thinking a lot about how languages can and should exploit modern computing hardware such as multi-core processors and parallel computing clusters
- Modern functional programming languages are emerging as better suited to modern hardware

1.2.2 Functional Programming, concurrency, parallel programming and shared mutable state

- FP languages emphasise the use of *immutable* data, *pure*, *referentially transparent functions*, and *higher-order functions*
- Unlike commonly used *imperative* programming languages, they are closer to the Church end of the *Church-Turing thesis* — eg. closer to *Lambda–calculus* than a *Turing–machine*
- The original Lambda–calculus was *untyped*, corresponding to a *dynamically–typed* programming language, such as *Lisp*

R is a dynamically typed language which has some functional characteristics. The original version of R was built from a *Scheme* interpreter. Scheme is a dialect of Lisp.

- *Statically–typed* FP languages (such as *Haskell*) are arguably more scalable, corresponding to the *simply–typed Lambda–calculus*, closely related to *Cartesian closed categories*...
- In pure FP, all state is *immutable* — you can assign names to things, but you can't change what the name points to — no "variables" in the usual sense
- Functions are *pure* and *referentially transparent* — they can't have side-effects — they are just like functions in mathematics...
- Functions can be recursive, and *recursion* can be used to iterate over recursive data structures — useful since no conventional "for" or "while" loops in pure FP languages

- Functions are first class objects, and *higher-order functions* (HOFs) are used extensively — functions which return a function or accept a function as argument
- Modern computer architectures have processors with several cores, and possibly several processors — parallel programming is required to properly exploit this hardware
- The main difficulties with parallel and concurrent programming using imperative languages all relate to issues associated with *shared mutable state*
- In pure FP, state is not mutable, so there is no mutable state, and hence no shared mutable state — most of the difficulties associated with parallel and concurrent programming just don't exist in FP — this has been one of the main reasons for the recent resurgence of FP languages
- We should approach the problem of statistical modelling and efficient computation in a modular, composable, functional way
- To do this we need programming languages which are:
  - *Strongly statically typed* (but with type inference)
  - *Compiled* (but possibly to a VM)
  - *Functional* (with support for immutable values, immutable collections, ADTs and higher-order functions)
  - and have support for *type-classes* and *higher-kinded types*, allowing the adoption of design patterns from *category theory*

Very few languages have proper support for higher kinded types, with Scala and Haskell (and various Haskell derivatives) being the best known examples

The relative merits of lazy and strict evaluation is controversial. Haskell is lazy by default. This enables some very elegant solutions to programming problems, but leads to some unpredictable behaviour and inefficiencies. Scala is strict by default, but as we will see, supports lazy evaluation when required.

- For efficient statistical computing, it can be argued that evaluation should be *strict* rather than *lazy* by default
- *Scala* is a popular language which meets the above constraints

1.2.3 What is Scala?

Scala

The name *Scala* derives from "*Sca*lable *La*nguage"

- is a *general purpose* language with a sizeable user community and an array of general purpose libraries, including good GUI libraries, networking and web frameworks
- is free, *open-source* and *platform independent*, *fast* and efficient with a strong type system, and is *statically typed* with good compile-time type checking and *type safety*
- has a good, well-designed *library for scientific computing*, including non-uniform random number generation and linear algebra
- has reasonable *type inference* and a *REPL* for interactive use

- has good *tool support* (including build tools, doc tools, testing tools, and intelligent IDEs)
- has excellent support for *functional programming*, including support for *immutability* and immutable data structures and monadic design exploiting *higher kinded types*
- allows imperative programming for those (rare) occasions where it makes sense
- is designed with *concurrency* and *parallelism* in mind, having excellent language and library support for building really *scalable* concurrent and parallel applications

Scala is *not* a *pure* FP language. It is possible to use mutable variables and data structures, and to write code in an imperative style. This can sometimes be useful for reasons of performance and efficiency in the context of certain iterative computations.

- It is a hybrid object-oriented/functional language
- It supports both functional and imperative styles of programming, but functional style is idiomatic
- It is statically typed and compiled — compiling to Java byte-code to run on the JVM
- It was originally developed by Martin Odersky, one of the authors of the Java compiler, `javac` as well as the creator of Java generics, introduced in Java 5.
- Development driven by perceived shortcomings of the Java programming language
- Scala is widely used in many large high-profile companies and organisations — it is now a mainstream general purpose language
- Many large high-traffic websites are built with Scala (eg. Twitter, Foursquare, LinkedIn, Coursera, The Guardian, etc.)
- Scala is widely used as a Data Science and Big Data platform due to its speed, robustness, concurrency, parallelism and general scalability (in addition to seamless Java interoperability)
- Scala programmers are being actively recruited by many high profile data science teams

Static versus dynamic typing, compiled versus interpreted

- It is fun to quickly throw together a few functions in a scripting language without worrying about declaring the types of anything, but for any code you want to keep or share with others you end up adding lots of boilerplate argument checking code that would be much cleaner, simpler and faster in a statically typed language

The `lm` function in **R** is a good example of this — take a look at the source for it — only the last few lines correspond to fitting a regression model — the rest is boilerplate

- Scala has a strong type system offering a high degree of compile-time checking, making it a safe and efficient language

- By maximising the work done by the compiler at build time you minimise the overheads at runtime
- Coupled with type inference, statically typed code is actually *more concise* than dynamic code

Many people find this counter-intuitive, but it is true that Scala code to solve a problem is typically shorter than the corresponding solution in a dynamic language like Python.

Functional versus imperative programming

- Functional programming offers many advantages over other programming styles
- In particular, it provides the best route to building scalable software, in terms of both program complexity and data size/complexity
- Scala has good support for functional programming, including immutable named values, immutable data structures and for-comprehensions
- Many languages are attempting to add functional features retrospectively (eg. lambdas in C++, lambdas, Streams and the Optional monad in Java 8, etc.)
- Many new and increasingly popular languages are functional, and several are inspired by Scala (eg. Apple's Swift is essentially a cut down Scala, as is Red Hat's Ceylon)

Using Scala

- Scala is completely free and open source — the entire Scala software distribution, including compiler and libraries, is released under a BSD license
- Scala is platform independent, running on any system for which a JVM exists
- It is easy to install `scala` and `scalac`, the Scala compiler, but not really necessary
- The "simple build tool" for Scala, *`sbt`*, is all that is needed to build and run most Scala projects, and this can be bootstrapped from a 1M Java file, `sbt-launch.jar`

`sbt` is a very powerful build-tool for Scala that we will be using in this course: `www.scala-sbt.org/`

1.2.4 Scala tools and libraries

Versions, packages, platform independence, cloud

- All dependencies, including Scala library versions, and associated "packages", can be specified in a `sbt` build file, and pulled and cached at build time — there is no need to "install" anything, ever — this means that most basic library/package versioning problems simply disappear

- This is particularly convenient when scaling out to virtual machines and lightweight containers (such as Docker) in the cloud — all that is required is a container with a JVM, and you can either build from source or push a redistributable binary package

Reproducible research

- Reproducible research is very important — others should be able to run your code and reproduce your results
  - Many within the statistics community have come to associate reproducible research with dynamic documents and literate programming, but in reality this is a minor part of the reproducibility problem — you can more-or-less guarantee that R code and documentation written with the latest trendy literate programming framework will not build and run in 2 years time due to incompatible library and package version changes in the meantime
- `sbt` build files specify the particular versions of Scala and any associated dependencies required, and so projects should build and run *reproducibly* without issues for many years
- There is a standard code documentation format, *Scaladoc*, an improved Scala version of Javadoc, and standard testing frameworks such as *ScalaTest* and *ScalaCheck*.

Example sbt build file

```
name := "app-template"

version := "0.1"

scalacOptions ++= Seq(
  "-unchecked", "-deprecation", "-feature"
)

libraryDependencies  ++= Seq(
  "org.scalacheck" %% "scalacheck" % "1.13.4" % "test",
  "org.scalatest" %% "scalatest" % "3.0.1" % "test",
  "org.scalanlp" %% "breeze" % "0.13",
  "org.scalanlp" %% "breeze-natives" % "0.13"
)

resolvers ++= Seq(
  "Sonatype Snapshots" at
    "https://oss.sonatype.org/content/repositories/snapshots/",
  "Sonatype Releases" at
    "https://oss.sonatype.org/content/repositories/releases/"
)

scalaVersion := "2.12.1"
```

IDEs

An *integrated development environment* (IDE) is more than just a text editor with some syntax highlighting — it can also analyse your code, find bugs, compile, build and help with re-factoring.

- There are several very good IDEs for Scala
- In general, IDEs are much better and much more powerful for statically typed languages — IDEs can do lots of things to help you when programming in a statically typed language which simply aren't possible when using a dynamically typed language

`scala-ide.org`

- The "Scala IDE" is based on Eclipse (a popular IDE for Java programmers) — this is a good "first Scala IDE" for many people

`www.jetbrains.com/idea/features/scala.html`

- IntelliJ is another Java IDE which some prefer to Eclipse. It has a Scala plugin which is good and is getting progressively better.
- For users already familiar with Emacs, the ENhanced Scala Interaction Mode for Emacs (ENSIME) arguably makes the best Scala IDE
- If you use an IDE, your code will (almost) always compile first time

Data structures and parallelism

- Scala has an extensive "Collections framework", providing a large range of data structures for almost any task (Array, List, Vector, Map, Range, Stream, Queue, Stack, ...)
- Most are collections available as either a *mutable* or *immutable* version — idiomatic Scala code favours immutable collections
- Most collections have an associated *parallel* version, providing concurrency and parallelism "for free"

Functional approaches to concurrency and parallelisation

- Functional languages emphasise immutable state and referentially transparent functions
- *Immutable state*, and *referentially transparent* (*side-effect* free) declarative workflow patterns are widely used for systems which really need to scale (leads to naturally parallel code)
- *Shared mutable state* is the enemy of concurrency and parallelism (synchronisation, locks, waits, deadlocks, race conditions, ...) — by avoiding *mutable state*, code becomes easy to parallelise
- The recent resurgence of functional programming and functional programming languages is partly driven by the realisation that functional programming provides a natural way to develop algorithms which can exploit multi-core parallel and distributed architectures, and efficiently scale

Category theory

When you start functional programming you start to see terms like "functor", "monad", "monoid" and "applicative" being used. This can be a bit intimidating for new users. These terms are associated with a branch of mathematics known as "category theory". Due to the close connection between category theory and the typed lambda calculus that strongly typed functional programming languages are based on, it is natural to borrow concepts from category theory, and to name them accordingly. In fact, these concepts are not difficult, as the brief guide below illustrates.
Dummies guide:

- A "collection" (or parametrised "container" type) together with a "map" function (defined in a sensible way) represents a *functor*
- If the collection additionally supports a (sensible) "apply" (or "zip") operation, it is an *applicative*
- If the collection additionally supports a (sensible) "flattening" operation, it is a *monad* (required for composition)
- For a "reduce" operation on a collection to parallelise cleanly, the type of the collection together with the reduction operation must define a *monoid* (must be an *associative* operation, so that reductions can proceed in multiple threads in parallel, using tree-reduction)

We will unpack these ideas properly later.

Scala ecosystem

There are many useful libraries in the Scala ecosystem. Some commonly encountered libraries are given below:

- *Breeze* — library for scientific and statistical computing
- *Spire* — numeric type-classes
- *Akka* — actor-based concurrency framework (inspired by Erlang)
- *Apache Spark* — big data framework built on Akka
- *Cats* — category theory types (functors, monads, etc.)
- *Shapeless* — type-safe dynamic type support
- *Play* — web framework
- *Scala.js* — use Scala client-side by compiling to JS

Breeze

https://github.com/scalanlp/breeze

- Breeze is a Scala library for scientific and numerical computing
- It includes all of the usual special functions, probability distributions, (non-uniform) random number generators, matrices, vectors, numerical linear algebra, optimisation, etc.
- For numerical linear algebra it provides a Scala wrapper over *netlib-java*, which calls out to a native optimised BLAS/LAPACK if one can be found on the system (so, will run as fast as native code), or will fall back to a Java implementation if no native libraries can be found (so that code will always run)

Apache Spark

- Spark is a scalable analytics library, including some ML (originally from Berkeley's AMP Lab)
- It is rapidly replacing MapReduce as the standard library for big data processing and analytics
- It is written in Scala, but also has APIs for Java and Python (and an experimental API for R)
- Only the Scala API leads to both concise elegant code and good performance
- Central to the approach is the notion of a *resilient distributed dataset* (RDD) — hooked up to Spark via a *connector* — using RDDs is very similar to using other Scala collections

Calling Scala from R (and vice versa)

- R CRAN package *rscala* allows bi-directional calling between R and Scala
- Useful for calling out to computationally intensive routine written in Scala from an R session
- Also useful for calling to R from Scala for a model-fitting procedure "missing" from the Scala libraries
- Can inline R code in Scala files and Scala code in R files

1.2.5 Hello, world!

No introduction to Scala would be complete without a small, complete program.

```
/*
  multi-line
  comment
*/
```

```scala
object MyApp {

  def main(args: Array[String]): Unit =
    println("Hello, world!")

} // single line comment
```

1.3 Basic tools for Scala development

1.3.1 Java and the JVM ecosystem

Scala is a JVM language. This means that Scala source code is compiled to Java byte-code designed to be run on a Java Virtual Machine (JVM). Typically a single `.scala` source code file will be compiled to a **.class** byte-code file, and a collection of (linked) **.class** files are often packaged together in a `.jar` Java ARchive file. Once a Scala application is packaged up into a stand-alone (assembly) JAR file, it should be possible to run it on any JVM on any machine (even machines with different architectures running different OSs). This platform–independence of JVM languages is very convenient for many reasons. In particular, it is one of the reasons for the popularity of JVM languages in Cloud computing and Big Data scenarios — you can easily develop on a laptop and then deploy your pre-built application to a cluster in the Cloud.

Scala isn't only a JVM language, but the version we will be using for this course is. There is also a version of Scala which compiles to JavaScript for use in web development, and Scala–native, which compiles to native code. The JVM version is the most stable and mature.

Scala 2.12.x projects require Java 1.8 or higher, and this is usually referred to simply as Java 8

In principle, only a basic Java Runtime Environment (JRE) is required for compiling and deploying Scala applications. However, recent versions of `sbt` require a full Java Development Kit (JDK), so now you need a JDK for `sbt`–based development (compiling and assembling), but still require only a basic JRE for deployment (running the application).

Before attempting to use Scala you should make sure that you have a Java 8 JDK installed. It doesn't matter whether this JDK is the OpenJDK or Oracle's. On Debian–based Linux systems, this can be done with

```
sudo apt-get install openjdk-8-jdk
```

Other Linux-based systems should be similar. For other OSs, try the Oracle Java JDK download page. To check whether you have Java installed correctly, type `java -version` into a terminal window. If you get a version number of the form 1.8.x you should be fine.

`http://www.oracle.com/technetwork/java/javase/downloads`
Note that the Standard Edition (SE) of Java should always be used — not the Enterprise Edition (EE).

1.3.2 Scala, scalac, and the Scala REPL

It is possible to do a system-wide installation of `scala` the Scala compiler, `scalac`, though it is not necessary. On Debian-based Linux systems, this can be done with

```
sudo apt-get install scala
```

If you have Scala installed, you can check which version with

```
scala -version
```

For other OSs, just download and install from the Scala language homepage. With Scala installed, you can start a Scala REPL just by typing **`scala`** at your command prompt.

Read–Eval–Print–Loop (REPL)

The disadvantage of this approach is that it inevitably makes this one particular version of Scala "special" on your system. It also doesn't make it easy to run code with dependencies on libraries that are not part of the Scala Standard Library. Using **`sbt`** is usually preferable for everything other than small stand-alone scripts, so we won't be assuming a system-wide Scala installation for this course.

1.3.3 SBT

SBT is the Scala Build Tool — it is the most widely used build–tool for Scala. Other than a recent Java installation, SBT is all you need for building Scala projects. Detailed instructions for installing SBT and testing that it works are provided in the course GitHub repo. Once SBT is installed, it can be started by typing **`sbt`** at a command prompt. Note that SBT creates files and directories, so it should only be started from a directory containing a Scala project. On running SBT you will get an SBT command-prompt, **`>`**. SBT is designed to be used interactively. Type **`exit`** to exit back to your OS command prompt or **`help`** to get an SBT command summary. Type **`compile`** to compile the project, and **`run`** to run the application. Note that the **`run`** task has a dependency on the **`compile`** task, and so it is safe to just type **`run`** and a compilation will first be triggered if required. The **`test`** task will run any tests associated with the project. The **`console`** task will start up a Scala REPL. Type Scala expressions and then **`:quit`** to exit the REPL back to the SBT prompt. Typing **`:help`** at the Scala REPL prompt will give a REPL command summary. Note that the REPL provided by the **`console`** task will include all library dependencies specified in the SBT build file and also all of the functions, classes and objects defined in this particular Scala project. **`clean`** is another commonly used SBT task (for deleting compiled artifacts), as is **`reload`** (after editing **`build.sbt`**). Note that prefixing a task with a tilde ~ causes it to wait after completion and watch for changes to source code files and automatically re-run whenever any source code file changes. For example, running **`~test`** will cause all tests to re-run whenever any project source code file is saved. Similarly for **`~compile`** and **`~run`**. SBT–based Scala projects usually assume a standard directory layout along the lines of

```
.
 build.sbt
 project
  build.properties
 src
     main
      scala
          scala-files.scala
     test
```

```
        scala
            scala-test-files.scala
```

Note that the version of SBT to be used is specified in the file `project/build.properties` with a line like `sbt.version=0.13.8`. All other library and version dependencies are specified in the top-level SBT build file `build.sbt`. Some simple examples of build files are included in the course GitHub repo.

The SBT version can't be specified in `build.sbt`, as this file is interpreted by the required version of SBT...

Note that although SBT is mainly intended to be used interactively, it is also possible to use directly from an OS command prompt, so that typing `sbt run` at your OS command prompt is like running the `run` task from the SBT prompt. Read the official SBT Getting Started Guide for more information.

1.3.4 The Scala IDE

The Scala IDE, based on Eclipse, is the IDE most commonly used by programmers new to Scala. Some basic installation instructions are provided in the course GitHub repo. Note that to use the Scala IDE with SBT projects, you must also install the `sbteclipse` SBT plugin which provides a new SBT task, `eclipse`.

The main thing to understand is that the ScalaIDE needs to know about the structure of your sbt project. This information is encoded in Eclipse project files in the top-level directory of your sbt project (where the file `build.sbt` will often be present). An initial set of project files for an sbt project can be generated using the `eclipse` sbt task provided by the `sbteclipse` plugin.

So, before using the ScalaIDE with a particular SBT project for the first time, first run

```
sbt eclipse
```

to analyse the project and create eclipse project files for it. Then start the ScalaIDE. If it asks about a work-space, make sure you select something *different to* the SBT project directory. Then import the project using the *Import Wizard* (under the File menu) to import *Existing Projects into Workspace*. You may need to repeat this process if you make significant changes to the `build.sbt` file.

Once you are up-and-running, Eclipse provides fairly sophisticated IDE functionality. Some commonly used commands include:

- Shift-Ctrl-F - Reformat source file
- Shift-Ctrl-P - Go to matching bracket
- Ctrl-Space - Content assist
- Shift-Ctrl-W - Close all windows (from package explorer)

Scala worksheet

- Shift-Ctrl-B - Re-run all code

See the ScalaIDE Documentation for further information.

1.3.5 Emacs and Ensime

People familiar with Emacs will probably find that Emacs together with Ensime provides the best Scala development experience. Some basic installation instructions are provided in the course GitHub repo. Note that to use Ensime with SBT projects, you must also install the `sbt-ensime` SBT plugin which provides a new SBT task, `ensimeConfig`.

The main thing to understand is that Ensime needs to know about the structure of your sbt project. This information is encoded in a file `.ensime` in the top-level directory of your sbt project (where the file `build.sbt` will often be present). An initial `.ensime` file for an SBT project can be generated using the `ensimeConfig` SBT task provided by the `sbt-ensime` plugin.

So, before using Emacs/Ensime with a particular SBT project for the first time, first run

```
sbt ensimeConfig
```

to analyse the project and create a `.ensime` file for it. You should probably re-run this after editing `build.sbt` or other build configuration files. Then start emacs with a command like

```
emacs src/main/scala/blah/*.scala &
```

This will start up emacs and some basic syntax highlighting will be provided by `scala-mode`. However, you still need to start up Ensime with `M-x ensime`. Once you are up-and-running, Ensime provides fairly sophisticated IDE functionality. Some commonly used commands include:

- M-x ensime - Start up Ensime
- C-c C-v f - Reformat source code in this buffer
- C-c C-b c - sbt compile
- C-c C-b r - sbt run

See the Emacs Ensime User Guide for further details.

1.4 Summary

Summary

- Strong arguments can be made that a language to be used as a platform for serious statistical computing should be general purpose, platform independent, functional, statically typed and compiled
- For basic exploratory data analysis, visualisation and model fitting, R works perfectly well (currently better than Scala)
- Scala is worth considering if you are interested in any of the following:
  - Writing your own statistical routines or algorithms

- Working with computationally intensive (parallel) algorithms
- Working with large and complex models for which an out-of-the-box solution doesn't exist in R
- Working with large data sets (big data)
- Integrating statistical analysis workflow with other system components (including web infrastructure, relational databases, no-sql databases, etc.)

Chapter 2

Scala and FP basics

2.1 Basic concepts and syntax

2.1.1 A Scala REPL session

We'll start with a few simple Scala concepts in order to introduce some syntax. There's a lot of introductory material on Scala on-line, and we will examine some of this during the practical sessions. Here we will just look at some of the most relevant concepts for this course.

```
Welcome to Scala 2.12.1 (OpenJDK 64-Bit Server VM, Java 1.8.0_121).
Type in expressions for evaluation. Or try :help.

scala> val a = 5
a: Int = 5

scala> a
res0: Int = 5
```

So far, so good. Using the Scala REPL is much like using the Python or **R** command line, so will be very familiar to anyone used to these or similar languages. The first thing to note is that labels need to be declared on first use. We have declared `a` to be a **val**. These are *immutable values*, which can not be just re-assigned, as the following code illustrates.

```
scala> a = 6
:8: error: reassignment to val
       a = 6
         ^
scala> a
res1: Int = 5
```

Immutability can be a difficult concept for people unfamiliar with functional programming. But fear not, as Scala allows declaration of *mutable variables* as well:

```
scala> var b = 7
b: Int = 7

scala> b
res2: Int = 7
```

```
scala> b = 8
b: Int = 8

scala> b
res3: Int = 8
```

The Zen of functional programming is to realise that immutability is generally a good thing, and we shall explore this more during the course. Scala has excellent support for both mutable and immutable collections as part of the standard library, and we shall examine these further in the next Chapter. For example, it has immutable lists.

```
scala> val c = List(3,4,5,6)
c: List[Int] = List(3, 4, 5, 6)

scala> c(1)
res4: Int = 4

scala> c.sum
res5: Int = 18

scala> c.length
res6: Int = 4

scala> c.product
res7: Int = 360
```

Again, this should be pretty familiar stuff for anyone familiar with dynamic languages such as **R** or Python. Note that the `sum` and `product` methods are special cases of *reduce* operations, which are well supported in Scala. For example, we could compute the sum reduction using

```
scala> c.foldLeft(0)((x,y) => x+y)
res8: Int = 18
```

or the slightly more condensed form given below, and similarly for the product reduction.

```
scala> c.foldLeft(0)(_+_)
res9: Int = 18

scala> c.foldLeft(1)(_*_)
res10: Int = 360
```

In fact, if the reduction function being applied is *associative*, and the reduction is initialised with an *identity* of the operation (so that the values and operator together form a *monoid*), then we can use the even simpler `reduce` method

```
scala> c.reduce(_*_)
res11: Int = 360
```

Scala also has a nice immutable `Vector` class, which offers a range of (essentially) constant time operations (but note that this has nothing to do with the mutable `Vector` class that is part of the Breeze library).

```
scala> val d = Vector(2,3,4,5,6,7,8,9)
d: Vector[Int] = Vector(2, 3, 4, 5, 6, 7, 8, 9)

scala> d
res11: Vector[Int] = Vector(2, 3, 4, 5, 6, 7, 8, 9)

scala> d.slice(3,6)
res12: Vector[Int] = Vector(5, 6, 7)

scala> val e = d.updated(3,0)
e: Vector[Int] = Vector(2, 3, 4, 0, 6, 7, 8, 9)

scala> d
res13: Vector[Int] = Vector(2, 3, 4, 5, 6, 7, 8, 9)

scala> e
res14: Vector[Int] = Vector(2, 3, 4, 0, 6, 7, 8, 9)
```

Note that when `e` is created as an updated version of `d` the whole of `d` is *not* copied — only the parts that have been updated. And we don't have to worry that aspects of `d` and `e` point to the same information in memory, as they are both immutable. As should be clear by now, Scala has excellent support for functional programming techniques. In addition to the *reduce* operations mentioned already, *maps* and *filters* are also well covered.

```
scala> val f=(1 to 10).toList
f: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> f
res15: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> f.map(x => x*x)
res16: List[Int] = List(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)

scala> f map {x => x*x}
res17: List[Int] = List(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)

scala> f filter {_ > 4}
res18: List[Int] = List(5, 6, 7, 8, 9, 10)
```

Note how Scala allows methods with a single argument to be written as an infix operator, making for more readable code.

For most single expressions, it doesn't matter whether they are enclosed in regular parentheses or curly braces. However, braces allow the use of multiple expressions on new lines without semicolons, so tend to be used more often by default.

2.1.2 Scala as a mathematical calculator

Many of the most often used mathematical functions can be found in the package `scala.math`.

At this point we will switch to a different way of showing Scala commands and the output that is returned by the REPL. This will make copying-and-pasting of code fragments more straightforward. This is similar to how the "Scala Worksheet" works within the Scala IDE.

```
math.log(2.0)
// res0: Double = 0.6931471805599453
math.sin(1.0)
// res1: Double = 0.8414709848078965
log(2.0)
// <console>:8: error: not found: value log
//               log(2.0)
//               ^
```

```
import math.log
// import math.log
log(2.0)
// res3: Double = 0.6931471805599453
import math._
// import math._
sin(Pi/2)
// res4: Double = 1.0
exp(log(sin(Pi/2)))
// res5: Double = 1.0
sin(asin(0.1))
// res6: Double = 0.1
atan(1)*4
// res7: Double = 3.141592653589793
log(sqrt(exp(1)))
// res8: Double = 0.5
abs(min(-1,2))
// res9: Int = 1
pow(2,8)
// res10: Double = 256.0
random
// res11: Double = 0.0954535018607291
random
// res12: Double = 0.5669552981874513
random
// res13: Double = 0.9598287994663521
floor(random*3)
// res14: Double = 2.0
floor(random*3)
// res15: Double = 1.0
floor(random*3)
// res16: Double = 1.0
floor(random*3)
// res17: Double = 1.0
```

Doing wild-card imports like **import** `math._` is often frowned upon in production code, as it clutters the name-space, but it can be especially useful for interactive experiments in the REPL.

Note that the REPL has TAB–completion, so typing `import math.` and then hitting the TAB key will show a list of items in the `math` package.

2.1.3 Type inference

Given that Scala is a strongly statically typed language, it is possibly surprising that none of the code in the previous sessions has involved the manual writing of any kind of type annotation. As previously mentioned, Scala has *type inference* which means that type annotations are optional when the compiler can figure out what the types must be. In all of the previous examples, it is possible for the compiler to figure out the type of the result of each expression. But it is fine to include type annotations if desired.

```
val a1 = 1
// a1: Int = 1
val a2: Int = 1
// a2: Int = 1
val l1 = List(1, 2, 3)
// l1: List[Int] = List(1, 2, 3)
```

```
val l2: List[Int] = List(2, 3, 4)
// l2: List[Int] = List(2, 3, 4)
```

Sometimes it can be useful to include type annotations on numeric types, for example to ensure that a numeric constant is interpreted as a `Double`.

```
val a3: Double = 1
// a3: Double = 1.0
val a4: Int = 1.0
// <console>:7: error: type mismatch;
//  found   : Double(1.0)
//  required: Int
//        val a4: Int = 1.0
//                      ^
```

But the second example shows that care must be taken with numeric types. There are no automatic *down–castings* or *type conversions*.

Generally speaking, it is rare to see type annotations inside the body of methods or functions. The main place that type annotations are required are in the argument list of functions and methods. For type safety, it is usually considered good practice to also annotate the return type of functions and methods, though this is often not required.

2.1.4 Defining methods

Let us now think about defining our own methods. To illustrate, we will define our own factorial function

There is a subtle distinction between methods and functions in Scala that we will return to shortly. For now it is fine to think of methods as functions.

$$n! = \prod_{i=1}^{n} i.$$

The canonical functional definition is just

```
def fact1(n: Int): Int = (1 to n).product
// fact1: (n: Int)Int
fact1(5)
// res0: Int = 120
```

Note that we explicitly annotate the method argument `n` to be an `Int`. This annotation is required. We also annotate the return type of the method to be `Int`. This is not required, but it is good practice to do so. There is nothing really wrong with this definition, but for very large values of the argument `n`, this would involve creating a large collection in heap memory to then reduce using the `product` method. This isn't really a problem here, as the factorial function will overflow `Int` long before heap space is a problem. But conceptually we shouldn't need to consume significant heap space in order to evaluate this function. We could implement this function in an imperative style using mutable variables as follows.

```
def fact2(n: Int): Int = {
  var acc = 1
  var i = 2
  while (i <= n) {
    acc *= i
    i += 1
    }
  acc
  }
// fact2: (n: Int)Int
fact2(5)
// res1: Int = 120
```

This works fine, but isn't very functional. The standard functional approach to this problem is to exploit recursion, as follows.

```
def fact3(n: Int): Int = {
  if (n == 1) 1 else
    n * fact3(n-1)
  }
// fact3: (n: Int)Int
fact3(5)
// res2: Int = 120
```

There are numerous important things to notice here. The first is that in Scala `if` is an *expression* and not a *statement* — it simply returns a value depending on the outcome of evaluating the predicate. The next thing to note is that although `n` is an immutable value, the value it takes depends on the value it is passed when it is called, and this can be different at different times. Further, there is no problem with the function calling itself recursively, although annotating the function with its return type is required in this case.

Although the above function is *recursive*, it is not *tail–recursive*, in the sense that the value of the recursive value has to be modified (by multiplying by `n`) in order to obtain the return value of the function. This means that each recursive call consumes a stack frame, and so very deep recursions of this kind can cause a *stack overflow*. Contrast this with the following.

```
@annotation.tailrec
def fact4(n: Int, acc: Int = 1): Int = {
  if (n == 1) acc else
    fact4(n-1, acc*n)
  }
// fact4: (n: Int, acc: Int)Int
fact4(5)
// res3: Int = 120
```

Here we avoid manipulating the return of the recursive call by passing through an *accumulator* which can be returned directly on termination. This makes the function tail recursive. Knowing that a function is tail recursive is important in Scala, as the Scala compiler can (usually) perform *tail call elimination*. That is, the compiler will re-write the tail recursive function in a form not dissimilar to the imperative version, `fact2`. This does not consume stack frames, and hence will not blow the stack. The annotation, `@annotation.tailrec` is not required in order for tail call elimination to be performed. Its purpose is simply to cause the compiler to report an error if tail call elimination can not be performed. If you have a function that you think is tail recursive you usually want to know if the compiler can't perform tail call elimination, and so it is a good idea to always use the annotation on any method or function that you consider should be tail recursive. Also note that this function introduces *default arguments* — if this function is called without a second argument, `acc` defaults to a value of `1`.

The factorial function quickly overflows `Int` for large values of the argument `n`. It is therefore simpler to illustrate the stack problem using the log-factorial function, $\log(n!)$. We can compute this without first computing $n!$ using the fact that

$$\log(n!) = \sum_{i=1}^{n} \log(i).$$

We can write a simple function to recursively compute this as follows.

```
math.log(fact4(5))
// res4: Double = 4.787491742782046

def lfact(n: Int): Double = {
  if (n == 1) 0.0 else
    math.log(n) + lfact(n-1)
  }
// lfact: (n: Int)Double
lfact(5)
// res5: Double = 4.787491742782046
// lfact(10000) // will cause a stack overflow
```

Evaluating this function for an argument of $10,000$ should certainly not overflow `Double`, but does blow the stack — try it! Contrast this with the tail recursive version:

```
@annotation.tailrec
def lfacttr(n: Int, acc: Double = 0.0): Double = {
  if (n == 1) acc else
    lfacttr(n-1, acc + math.log(n))
  }
// lfacttr: (n: Int, acc: Double)Double
lfacttr(5)
// res6: Double = 4.787491742782046
lfacttr(10000)
// res7: Double = 82108.92783681446
```

This one works fine for even large arguments since it is not gobbling up stack frames.

This technique of making a non–tail–recursive function tail–recursive by introducing and explicitly passing a *continuation* for the computation is a very simple example of a powerful functional programming technique known as *continuation passing style*.

If you do want to work with large factorials directly, this is also possible: just switch from `Int` to `BigInt`, as follows.

```
@annotation.tailrec
def factbi(n: BigInt, acc: BigInt = 1): BigInt = {
  if (n == 1) acc else
    factbi(n-1, acc*n)
  }
// factbi: (n: BigInt, acc: BigInt)BigInt
factbi(5)
// res8: BigInt = 120
factbi(10000)
// res9: BigInt = 28462596809170545189064132121...
```

2.1.5 Complete programs

So far we have been using the Scala REPL interactively. But we typically want to build complete programs that can be compiled and then run (usually using SBT). The Scala compiler expects that all methods will be contained in either `Classes` or `Objects`, and that the entry point to the program will be a method called `main` with signature `main(args: Array[String]): Unit`, where `args` will contain any command–line arguments for the program. `Unit` is the Scala type containing a single value, `()`. Here it is used to signal that the `main` method does not return a useful value. A simple example program for computing a log–factorial and printing the result to screen is given below.

```scala
/*
log-fact.scala
Program to compute the log-factorial function
*/

object LogFact {

  import annotation.tailrec
  import math.log

  @tailrec
  def logfact(n: Int, acc: Double = 0.0): Double =
    if (n == 1) acc else
      logfact(n-1, acc + log(n))

  def main(args: Array[String]): Unit = {
    val n = if (args.length == 1) args(0).toInt else 5
    val lfn = logfact(n)
    println(s"logfact($n) = $lfn")
  }

}

// eof
```

If you have a system–wide scala installation, this program could be run from the OS command line with

```
scala log-fact.scala
```

It can be run with a particular argument with

```
scala log-fact.scala 1000
```

In this course we are using SBT to build and run Scala applications. If the file `log-fact.scala` is placed into an empty directory, SBT can be started from this directory (no build file is necessary), and then the code can be compiled and run with the `run` task. It can be run with a particular argument by typing, say, `run 100` at the SBT prompt. If you want to run the program directly from an OS command prompt, you can do so with

```
sbt run
```

If you want to use a non-default argument, you must quote the argument together with the `run` task as

```
sbt "run␣100"
```

2.2 Functional programming in Scala

We have already started to think about how recursion can be used in place of iteration in order to avoid mutating state. It's now time to look at some other techniques that are widely used in functional programming.

2.2.1 HOFs, closures, partial application and currying

Introduction

Functional programming (FP) emphasises the use of referentially transparent pure functions and immutable data structures. Higher order functions (HOFs) tend to be used extensively to enable a clean functional programming style. A HOF is just a function that either takes a function as an argument or returns a function. For example, the default `List` collection in Scala is immutable. So, if one defines a list via

```
val l1 = List(1,2,3)
```

we add a new value to the front of the list by creating a new list from the old list and leaving the old list unchanged:

```
val l2 = 4 :: l1
// List(4, 1, 2, 3)
```

We can create a new list the same length as an existing list by applying the same function to each element of the list in turn using `map`:

```
val l3 = l2 map { x => x*x }
// List(16, 1, 4, 9)
```

We could write this slightly differently as

```
val l4 = l2.map(x => x*x)
```

which makes it clearer that `map` is a higher order function on lists. HOFs are ubiquitous in FP, and very powerful. But there are a few techniques for working with functions in Scala (and other FP languages) which make creating and using HOFs more convenient.

In fact, the presence of a `map` method on `List[_]` makes it a *functor*, but that is a topic we will consider later.

Plotting a function of one scalar variable

There are many, many reasons for using functions and HOFs in scientific and statistical computing (optimising, integrating, differentiating, or sampling, to name just a few). But the basic idea can be illustrated simply by considering the problem of plotting a function of one scalar variable. For this we will use the plotting library *breeze-viz* which is not part of the Scala standard library, so the following

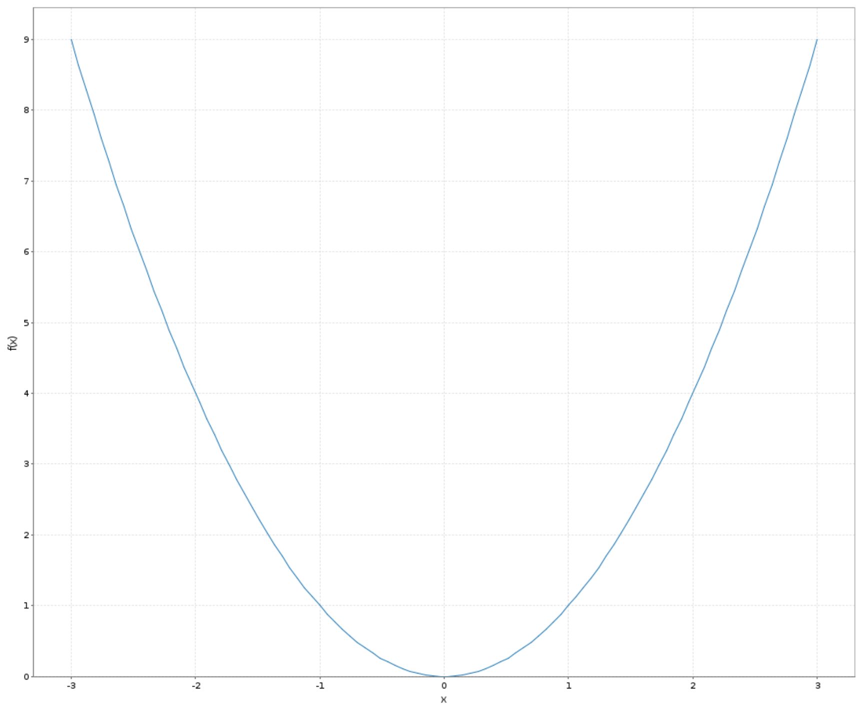

Figure 2.1: Plot of a quadratic function

code should be run from an SBT console from a project directory containing a build file with a dependence on `breeze-viz`. We will start by defining a function, `plotFun`, to plot on the screen the graph of a real valued function of a single real parameter. We will look at `breeze` and `breeze-viz` in more detail tomorrow so it doesn't matter if you don't understand exactly how this function works.

Running `sbt console` from the `sbt-test` directory of the course GitHub repo will be fine for this example.

```
import breeze.plot._
def plotFun(fun: Double => Double, xmin: Double =
               -3.0, xmax: Double = 3.0): Figure = {
  val f = Figure()
  val p = f.subplot(0)
  import breeze.linalg._
  val x = linspace(xmin, xmax)
  p += plot(x, x map fun)
  p.xlabel = "x"
  p.ylabel = "f(x)"
  f
}
```

Once we have this defined, we can use it to start plotting functions. For example,

```
plotFun(x => x*x)
```

produces the plot shown in Figure 2.1.

We can use this method to plot other functions, for example:

```
def myQuad1(x: Double): Double = x*x - 2*x + 1
plotFun(myQuad1)
def myQuad2(x: Double): Double = x*x - 3*x - 1
plotFun(myQuad2)
```

Now technically, `myQuad1` and `myQuad2` are *methods* rather than *functions*. The distinction is a bit subtle, and they can often be used interchangeably, but the difference does sometimes matter, so it is good to understand it. To actually define a function and plot it, we could instead use code like:

```
val myQuad3: (Double => Double) = x => -x*x + 2
plotFun(myQuad3)
```

Note that here `myQuad3` is a *value* whose *type* is a *function* mapping a `Double` to a `Double`. This is a proper function. This style of function declaration should make sense to people coming from other functional languages such as Haskell, but is potentially confusing to those coming from O-O languages such as Java. Note that it is easy to convert a method to a function using an underscore, so that, for example, `myQuad2 _` will give the function corresponding to `myQuad2`. Note that there is no corresponding way to get a method from a function, so that is one reason for preferring method declaration syntax (and there are others, such as the ability to parametrise method declarations with generic types). Now, rather than defining lots of specific instances of quadratic functions from scratch, it would make more sense to define a generic quadratic function and then just plot particular instances of this generic quadratic. It is simple enough to define a generic quadratic with:

```
def quadratic(a: Double, b: Double, c: Double,
                             x: Double): Double =
  a*x*x + b*x + c
```

But we clearly can't pass that in to the plotting function directly, as it has the wrong type signature (not `Double =>Double`), and the specific values of `a`, `b` and `c` need to be given. This issue crops up a lot in scientific and statistical computing — there is a function which has some additional parameters which need to be fixed before the function can actually be used. This is sometimes referred to as the "function environment problem". Fortunately, in functional languages it's easy enough to use this function to create a new "partially specified" or "partially applied" function and pass that in. For example, we could just do

```
plotFun(x => quadratic(3,2,1,x))
```

We can define another function, `quadFun`, which allows us to construct these partially applied function closures, and use it as follows:

```
def quadFun(a: Double, b: Double, c: Double):
    Double => Double = x => quadratic(a,b,c,x)
val myQuad4 = quadFun(2,1,3)
plotFun(myQuad4)
plotFun(quadFun(1,2,3))
```

Here, `quadFun` is a HOF in the sense that it returns a function closure corresponding to the partially applied `quadratic` function. The returned function has the type `Double =>Double`, so we can use it wherever a function with this signature is expected. Note that the function carries around with it its lexical "environment", specifically, the values of `a`, `b` and `c` specified at the time `quadFun` was called. This style of constructing closures works in most lexically scoped languages which have functions as first class objects. Again, the intention is perhaps slightly more explicit if we re-write the above using function syntax as:

```
val quadFunF: (Double,Double,Double) => Double =>
     Double = (a,b,c) => x => quadratic(a,b,c,x)
val myQuad5 = quadFunF(-1,1,2)
plotFun(myQuad5)
plotFun(quadFunF(1,-2,3))
```

Now, this concept of partial application is so prevalent in FP that some languages have special syntactic support for it. In Scala, we can partially apply using an underscore to represent unapplied parameters as:

```
val myQuad6 = quadratic(1,2,3,_: Double)
plotFun(myQuad6)
```

In Scala we can also directly write our functions in *curried* form, with parameters (or parameter lists) ordered as they are to be applied. So, for this example, we could define (partially) curried `quad` and use it with:

```
def quad(a: Double, b: Double, c: Double)(x: Double):
      Double = a*x*x + b*x + c
plotFun(quad(1,2,-3))
val myQuad7 = quad(1,0,1) _
plotFun(myQuad7)
```

Note the use of an underscore to convert a partially applied method to a function. Also note that every function has a method `curried` which turns an uncurried function into a (fully) curried function. So in the case of our quadratic function, the fully curried version will be a chain of four functions.

```
def quadCurried = (quadratic _).curried
plotFun(quadCurried(1)(2)(3))
```

Again, note the strategic use of an underscore. The underscore isn't necessary if we have a true function to start with, as the following illustrates:

```
val quadraticF: (Double,Double,Double,Double) => Double =
                     (a,b,c,x) => a*x*x + b*x + c
def quadCurried2 = quadraticF.curried
plotFun(quadCurried2(-1)(2)(3))
```

Working with functions, closures, HOFs and partial application is fundamental to effective functional programming style. Currying functions is one approach to handling the function environment problem, and is the standard approach in languages such as Haskell. However, in Scala there are other possible approaches, such as using underscores for partial application. The preferred approach will depend on the context. Currying is often used for HOFs accepting a function as argument (as it can help with type inference), and also in conjunction with *implicits* (which we will consider briefly in the final chapter). In other contexts partial application using underscores seems to be more commonly used.

2.2.2 Function composition

Many programs fundamentally consist of the application of a function to input data to get output data. In other words, the program itself can be considered to be a function. However, this function may be very complicated, and when writing software we tend to break down big complex functions into smaller, simpler functions, and repeat, until we are dealing with small functions of manageable size and complexity. But then we compose together the smaller pieces to get larger programs. It is worth considering carefully the act of building a function from smaller components. Here we consider a trivial example, but the same principles apply for functions of arbitrary size and complexity.

We start out thinking about a function to measure the length of a string.

```
val aLongString = (1 to 10000).map(_.toString).
                                   reduce(_+_)
// aLongString: String = 1234567891011121314151617...

val stringLength: String => Int = s => s.length
// stringLength: String => Int = <function1>

stringLength(aLongString)
// res0: Int = 38894
```

Clearly this is trivial, using the method `length` on `String`. Now suppose that for large strings, we would rather measure the length in kilo–characters, and that we have a function to convert an `Int` to a `Double` (in K). A very imperative way of composing these functions would be the following.

```
def convertToK: Int => Double = i => i.toDouble/1024
// convertToK: Int => Double

def stringLengthInK1(s: String): Double = {
  val l = stringLength(s)
  val lk = convertToK(l)
  lk
}
// stringLengthInK1: (s: String)Double

stringLengthInK1(aLongString)
// res1: Double = 37.982421875
```

There isn't anything fundamentally wrong with this, but it's useful to consider more functional approaches. A first attempt at improving this might be the following.

```
val stringLengthInK2: String => Double =
                      s => convertToK(stringLength(s))
// stringLengthInK2: String => Double = <function1>
```

```
stringLengthInK2(aLongString)
// res2: Double = 37.982421875
```

This captures the fundamental notion of function compostition. But since it is such a fundamental concept, it is helpful to abstract the notion as a HOF, which in Scala is called `compose`. We can modify this slightly using `compose` as follows.

```
val stringLengthInK3: String => Double =
            s => (convertToK compose stringLength)(s)
// stringLengthInK3: String => Double = <function1>

stringLengthInK3(aLongString)
// res3: Double = 37.982421875
```

But this is missing the point of introducing the abstraction. When we look at this carefully, it becomes clear that we don't need the string value `s` at all, and we could just directly define and use the composition.

```
val stringLengthInK4: String => Double =
                      convertToK compose stringLength
// stringLengthInK4: String => Double = <function1>

stringLengthInK4(aLongString)
// res4: Double = 37.982421875
```

This style of programming without values is known as *point–free style*, and is used extensively in functional programming.

Chapter 3

The Scala collections library

3.1 Scala collections overview and basics

3.1.1 Introduction

The Scala Standard Library provides a rich set of data structures for dealing with “collections” of objects. We have already encountered one of these: the `List` collection. Note that `List` itself is not a concrete type. It is a *parametrised* type (called a *generic* type in Java), and more commonly referred to as a *type constructor* in Scala. It is a type constructor in the sense that if you provide a type parameter it will return a concrete type. So, for example, `List` and `List[_]` are type constructors, but `List[Int]` and `List[Double]` are concrete types. The `List` collection that we have previously encountered is *immutable*. That means that it is not possible to change any of the elements in a `List`. But the Scala collections library also contains a *mutable* list, `MutableList`. Let’s see how this works.

```
val l1 = List(6,7,8,9,10)
// l1: List[Int] = List(6, 7, 8, 9, 10)
l1.head
// res0: Int = 6
l1.tail
// res1: List[Int] = List(7, 8, 9, 10)
l1(0)
// res2: Int = 6
l1(2)
// res3: Int = 8
l1(2) = 22
// <console>:9: error: value update is not a member of List[Int]
//                l1(2) = 22
//                ^
l1
// res5: List[Int] = List(6, 7, 8, 9, 10)
val l2: List[Double] = List(1,2,3)
// l2: List[Double] = List(1.0, 2.0, 3.0)
```

```
import scala.collection.mutable
// import scala.collection.mutable
val l3 = mutable.MutableList(5,6,7,8,9)
// l3: MutableList[Int] = MutableList(5, 6, 7, 8, 9)
l3.head
// res6: Int = 5
l3.tail
// res7: MutableList[Int] = MutableList(6, 7, 8, 9)
l3(2)
// res8: Int = 7
l3(2) = 22
l3
// res10: MutableList[Int] = MutableList(5, 6, 22, 8, 9)
```

Note that `l3` is an immutable **val** even though it holds a reference to a `MutableList` containing elements which can be changed. FP advocates the use of immutable references to immutable collections, but when mutability is required/desired, you will usually want *either* an immutable reference to a mutable collection (as above), *or* a mutable (**var**) reference to an immutable collection (so that the collection itself is immutable, but the collection referenced by the variable can be changed). Which of these two possibilities is most desirable will be problem–dependent. However, it is very rare to really require a mutable reference to a mutable collection — this is usually indicative of confusion.

3.1.2 The collections hierarchy

See the Scala Collections Overview for additional information: `docs.scala-lang.org/overviews/collections/overview`

`List` is by no means the only collection provided by the Scala Standard Library. There are many collections, abstract and concrete, arranged in a hierarchy in an Object-Oriented style. So, if we start with concrete collections like `List` and `Stream`, these are both a `LinearSeq`, and a `LinearSeq` is a `Seq`, and a `Seq` is an `Iterable` and an `Iterable` is a `Traversable`. On the other hand, `Vector` and `Range` are both examples of `IndexedSeq`, which is itself an example of `Seq`. So the common superclass of `Vector` and `List` is `Seq`. Unfortunately we don't have time to go through the full collections hierarchy here in detail, but the official documentation available on-line is good. See the Collections Overview in the Scala Documentation for further details.

Here we will just focus on a few of the most useful Scala collections from the perspective of statistical computing: `List`, `Array`, `Vector`, `Map`, `Stream` and `Range`. As we will see, most functionality is common to all Scala collections.

List

We have already seen `List` — it is an immutable linked list. It has very fast prepend (`::`) and `head` and `tail` operations. But it has relatively slow indexed access and append operations, so it should be avoided if these are the main focus — `Array` or `Vector` will be better in this case. Here are some more examples of using `List`.

```
val list1 = List(3,4,5,6)
// list1: List[Int] = List(3, 4, 5, 6)
val list2 = 2 :: list1
// list2: List[Int] = List(2, 3, 4, 5, 6)
val list3 = list1 ++ list2
// list3: List[Int] = List(3, 4, 5, 6, 2, 3, 4, 5, 6)
list3.take(3)
// res0: List[Int] = List(3, 4, 5)
list3.drop(2)
// res1: List[Int] = List(5, 6, 2, 3, 4, 5, 6)
list3.filter(_<5)
// res2: List[Int] = List(3, 4, 2, 3, 4)
val list4 = list1 map (_*2)
// list4: List[Int] = List(6, 8, 10, 12)
list4.length
// res3: Int = 4
list4.sum
// res4: Int = 36
list4.reverse
// res5: List[Int] = List(12, 10, 8, 6)
list1.foldLeft(0)(_+_)
// res6: Int = 18
list1.scanLeft(0)(_+_)
// res7: List[Int] = List(0, 3, 7, 12, 18)
list1.reduce(_+_)
// res8: Int = 18
list1.sortWith(_>_)
// res9: List[Int] = List(6, 5, 4, 3)
list1.foreach{e => println(e)}
// 3
// 4
// 5
// 6
```

Array

`Array` is a mutable data structure providing a thin layer over Java arrays. Consequently, `Array` has very fast indexed access and in-place update. When used with primitive numeric types such as `Int` and `Double`, they should be more-or-less as fast as numeric arrays in natively compiled C code. `Array`s tend to be used whenever fast in-place mutation is required, or inter-op with Java arrays.

```
val arr1 = Array(2,3,4,5)
// arr1: Array[Int] = Array(2, 3, 4, 5)
arr1(1)=6
arr1
// res12: Array[Int] = Array(2, 6, 4, 5)
val arr2 = arr1 map(_+1)
// arr2: Array[Int] = Array(3, 7, 5, 6)
arr1
// res13: Array[Int] = Array(2, 6, 4, 5)
arr1.reduce(_+_)
// res14: Int = 17
arr1 ++ arr2
// res15: Array[Int] = Array(2, 6, 4, 5, 3, 7, 5, 6)
```

Vector

`Vector` is an immutable alternative to `Array` which tends to be used a lot in Scala code. It has fast indexed access, prepend and append, and a fast immutable update, which returns a new `Vector`, leaving the old `Vector` unchanged, but without copying (most of) the original `Vector`. `Vector` has many applications in statistical computing, and as we will see shortly, also has a parallel implementation.

```
val vec1 = Vector(6,5,4,3)
// vec1: Vector[Int] = Vector(6, 5, 4, 3)
vec1(2)
// res16: Int = 4
vec1.toList
// res17: List[Int] = List(6, 5, 4, 3)
vec1 map (_*0.5)
// res18: Vector[Double] = Vector(3.0, 2.5, 2.0, 1.5)
val vec2 = vec1.updated(2,10)
// vec2: Vector[Int] = Vector(6, 5, 10, 3)
vec2.length
// res19: Int = 4
vec1
// res20: Vector[Int] = Vector(6, 5, 4, 3)
vec2
// res21: Vector[Int] = Vector(6, 5, 10, 3)
vec1 ++ vec2
// res22: Vector[Int] = Vector(6, 5, 4, 3, 6, 5, 10, 3)
vec1 :+ 22
// res23: Vector[Int] = Vector(6, 5, 4, 3, 22)
33 +: vec1
// res24: Vector[Int] = Vector(33, 6, 5, 4, 3)
vec1.reduce(_+_)
// res25: Int = 18
Vector.fill(5)(0)
// res26: Vector[Int] = Vector(0, 0, 0, 0, 0)
Vector.fill(5)(math.random)
// res27: Vector[Double] = Vector(0.35160713387930087,
//   0.0892297132191533, 0.9061352705408103,
//   0.7020295276855067, 0.09434089580898397)
```

Note that the final example relies on the fact that the second argument to `.fill` is lazily evaluated.

Map

`Map` is an immutable hash–map implementation, used for storing “keys” and “values”. These have many potential applications requiring fast look-up operations.

```
val map1 = Map("a"->10,"b"->20,"c"->30)
// map1: Map[String,Int] = Map(a -> 10, b -> 20, c -> 30)
map1("b")
// res28: Int = 20
val map2 = map1.updated("d",40)
// map2: Map[String,Int] = Map(a -> 10, b -> 20,
//   c -> 30, d -> 40)
map1
// res29: Map[String,Int] = Map(a -> 10, b -> 20, c -> 30)
map2
```

```
// res30: Map[String,Int] = Map(a -> 10, b -> 20,
//   c -> 30, d -> 40)
map2.updated("d",50)
// res31: Map[String,Int] = Map(a -> 10, b -> 20,
//   c -> 30, d -> 50)
map2.keys
// res32: Iterable[String] = Set(a, b, c, d)
map2.values
// res33: Iterable[Int] = MapLike(10, 20, 30, 40)
map2 mapValues (_*2)
// res34: Map[String,Int] = Map(a -> 20, b -> 40,
//   c -> 60, d -> 80)
val mapK = List(3,4,5,6)
// mapK: List[Int] = List(3, 4, 5, 6)
val mapV = List(0.3,0.4,0.5,0.6)
// mapV: List[Double] = List(0.3, 0.4, 0.5, 0.6)
val pairs = mapK zip mapV
// pairs: List[(Int, Double)] = List((3,0.3),
//   (4,0.4), (5,0.5), (6,0.6))
val map3 = pairs.toMap
// map3: Map[Int,Double] = Map(3 -> 0.3, 4 -> 0.4,
//   5 -> 0.5, 6 -> 0.6)
```

Stream

A `Stream` is a *lazy* immutable `List`. It is lazy in the sense that just like a regular list, it has a `head` and a `tail`, but for a `Stream` only the `head` is eagerly evaluated. Evaluation of the tail is evaluated only when required. `Stream`s are therefore useful for dealing with data of undetermined or infinite size. In particular, `Stream`s can be used to define and operate on `List`s of infinite length, as the examples below demonstrate. This is a very powerful concept, and there are many potential applications of `Stream`s in statistical computing — they provide an elegant framework for any kind of iterative algorithm. However, great care must be exercised when working with infinite `Stream`s, since it is very easy to apply an operation requiring an infinite amount of computation.

```
val stream1 = Stream(1,2,3)
// stream1: Stream[Int] = Stream(1, ?)
val stream2 = 0 #:: stream1
// stream2: Stream[Int] = Stream(0, ?)
stream2.toList
// res35: List[Int] = List(0, 1, 2, 3)
stream2.foldLeft(0)(_+_)
// res36: Int = 6
def fibFrom(a: Int,b: Int): Stream[Int] =
  a #:: fibFrom(b,a+b)
// fibFrom: (a: Int, b: Int)Stream[Int]
val fib = fibFrom(1,1)
// fib: Stream[Int] = Stream(1, ?)
fib.take(8).toList
// res37: List[Int] = List(1, 1, 2, 3, 5, 8, 13, 21)
val naturals = Stream.iterate(1)(_+1)
// naturals: Stream[Int] = Stream(1, ?)
```

```
naturals.take(5).toList
// res38: List[Int] = List(1, 2, 3, 4, 5)
val evens = naturals map (_*2)
// evens: Stream[Int] = Stream(2, ?)
evens.take(6).toList
// res39: List[Int] = List(2, 4, 6, 8, 10, 12)
val triangular = naturals.scanLeft(0)(_+_).drop(1)
// triangular: Stream[Int] = Stream(1, ?)
triangular.take(8).toList
// res40: List[Int] = List(1, 3, 6, 10, 15, 21, 28, 36)
```

Range

`Range` is a very simple collection used to represent a simple linear numerical sequence. It has constant size since it can be represented by three numbers (start, end and step size). It is intended that a `Range` is constructed using one of the convenience methods `to` or `until` (`until` excludes the final value).

```
1 to 5
// res41: Range.Inclusive = Range(1, 2, 3, 4, 5)
0 to 5
// res42: Range.Inclusive = Range(0, 1, 2, 3, 4, 5)
0 until 5
// res43: Range = Range(0, 1, 2, 3, 4)
0 to 10 by 2
// res44: Range = Range(0, 2, 4, 6, 8, 10)
val range = 0 to 10 by 2
// range: Range = Range(0, 2, 4, 6, 8, 10)
range.toList
// res45: List[Int] = List(0, 2, 4, 6, 8, 10)
(0 to 5) map (_*2)
// res46: IndexedSeq[Int] = Vector(0, 2, 4, 6, 8, 10)
(0 to 5).sum
// res47: Int = 15
```

3.1.3 Writing collection–generic code

Suppose we want to write our own function to compute the mean of a collection of `Double`s. We could write a function for `List[Double]` as follows.

```
def meanList(ld: List[Double]): Double = ld.sum/ld.length
// meanList: (ld: List[Double])Double
meanList(List(1,2,3,4))
// res0: Double = 2.5
```

This works fine for a `List`, but essentially the same code could be used for a `Vector`, or an `Array`. Rather than writing a different version of the function for every collection type we are interested in, we can instead write it once for the common supertype.

```
def mean(sq: Seq[Double]): Double = sq.sum/sq.length
// mean: (sq: Seq[Double])Double
mean(List(2,3,4))
// res1: Double = 3.0
```

```
mean(Vector(1,2,3))
// res2: Double = 2.0
```

This will work for any collection derived from `Seq` in the collections hierarchy.

3.1.4 Writing type–generic code

A similar issue arises when we want to write code that abstracts over the type parametrising the collection. Suppose we want to write a function which replicates a `List`. We can write such a function for `List[Double]` as follows.

```
def repD(ld: List[Double], n: Int): List[Double] =
  if (n <= 1) ld else ld ++ repD(ld,n-1)
// repD: (ld: List[Double], n: Int)List[Double]
repD(List(1,2,3),3)
// res3: List[Double] = List(1.0,2.0,3.0,1.0,2.0,3.0,1.0,2.0,3.0)
```

This works fine, but clearly we did not rely on the fact that the type parameter was `Double`. The same code should work for `List[Int]` or even `List[String]`. We can write the function for a `List[T]` where `T` is any type as follows.

```
def repl[T](l: List[T], n: Int): List[T] =
  if (n <=1) l else l ++ repl(l,n-1)
// repl: [T](l: List[T], n: Int)List[T]
repl(List(1,2,3),3)
// res5: List[Int] = List(1, 2, 3, 1, 2, 3, 1, 2, 3)
repl(List(1.0,2.0),3)
// res6: List[Double] = List(1.0,2.0,1.0,2.0,1.0,2.0)
repl(List("a","b","c"),2)
// res7: List[String] = List(a, b, c, a, b, c)
```

3.1.5 Writing (type– and) collection–generic code returning a collection

The method above replicates a `List`, but we haven't used anything `List`–specific, so it's natural to want to generalise this code to work for any collection. We kind of know how to do this already, by replacing `List` with a desired super–type, say `Seq` (or `Traversable`, or `Iterable`, ...).

```
def reps[T](l: Seq[T], n: Int): Seq[T] =
  if (n <=1) l else l ++ reps(l,n-1)
// reps: [T](l: Seq[T], n: Int)Seq[T]
reps(List(1,2,3),3)
// res8: Seq[Int] = List(1, 2, 3, 1, 2, 3, 1, 2, 3)
reps(Vector(1.0,2.0),3)
// res9: Seq[Double] = Vector(1.0,2.0,1.0,2.0,1.0,2.0)
reps(Array("a","b","c"),2)
// res10: Seq[String] = ArrayBuffer(a, b, c, a, b, c)
```

This works fine, but may not be exactly what you want. Look carefully at the return types. When you replicate a (say) `List[Int]`, you do actually get back a `List[Int]`, but its type is (obviously)

`Seq[Int]`. This might be OK, but this means that at compile time all you can assume about the return collection is that it is a `Seq`, which would mean that you couldn't call any `List`–specific methods on the result. This is likely to be problematic in many situations. What is required is a function that works for any `Seq`, but returns a collection of the same type as was passed in. It is perfectly possible to do this, but it requires a bit of "type magic" that is beyond the scope of this course to explain properly.

This works, but is a bit ugly. Many think that the ugliness of this kind of code is a symptom of the fundamental inadequacy of traditional inheritance–based object–oriented programming paradigms for providing polymorphism. It is possible to develop a more elegant approach to polymorphism and generic code using *type classes*, which we will consider briefly in the final Chapter.

```
import scala.collection.SeqLike
// import scala.collection.SeqLike
def rep[T, C <: Seq[T]](l: C with SeqLike[T,C], n: Int): C =
  if (n <=1) l else (l ++ rep(l,n-1)).asInstanceOf[C]
// rep: [T, C <: Seq[T]](l: C with SeqLike[T,C], n: Int)C
rep(List(1,2,3),3)
// res11: List[Int] = List(1, 2, 3, 1, 2, 3, 1, 2, 3)
rep(Vector(1.0,2.0),3)
// res12: Vector[Double] = Vector(1.0,2.0,1.0,2.0,1.0,2.0)
rep(Array("a","b","c").seq,2)
// res13: IndexedSeq[String] = ArrayBuffer(a, b, c, a, b, c)
```

Note now that if you pass in a `List[Int]` the return type is also `List[Int]`, and so you could still use `List`–specific methods on the return value.

3.1.6 Writing type–generic numerical code

The above approach to abstracting over the type parameter works fine so long as you don't need to know anything about the abstract type. But in numerical code, you often want to use numerical operations within generic functions. For example, suppose we want a function to compute the sum–of–squares of a collection of numerical values. We can write such a function for `Seq[Double]` easily.

```
def ssd(sq: Seq[Double]): Double = (sq map (x => x*x)).sum
// ssd: (sq: Seq[Double])Double
ssd(List(2,3,4))
// res7: Double = 29.0
```

Now this will work for other numeric types, but will always return a `Double`. This isn't necessarily what we want. For example, if we pass in a list of `Integer`s, we might want to get the sum–of–squares back as an `Integer`. But if we naively abstract over the numeric type we have a problem.

```
def ssg[T](sq: Seq[T]): T = (sq map (x => x*x)).sum
// <console>:7: error: value * is not a member of type paramete
//           def ssg[T](sq: Seq[T]): T = (sq map (x => x*x)).sum
```

The error message helpfully explains that since we know nothing about `T`, there is no reason to suppose that it will have a multiplication operation defined. We need to tell the compiler that we expect the type `T` to be a numeric type, for which basic numerical operations are defined. Unfortunately in Scala this is more difficult than it should be, due to the need to keep compatibility with the underlying Java numeric types (for performance reasons). We can do it, but using some advanced features of the language that we are not yet ready to discuss in detail.

```
def ss[T](sq: Seq[T])(implicit num: Numeric[T]): T = {
  import num._
  (sq map (x => x*x)).sum
  }
// ss: [T](sq: Seq[T])(implicit num: Numeric[T])T
ss(List(2,3,4))
// res8: Int = 29
ss(Vector(1.0,2.0,3.0))
// res9: Double = 14.0
```

This works, but is somewhat mysterious and a bit ugly. We can write essentially the same code slightly differently as follows.

Implicits are a mechanism for using values not explicitly passed in to functions. They are commonly used in Scala to implement *ad hoc polymorphism* via the *typeclass pattern*. We will touch briefly on these issues in the final Chapter, but they are largely beyond the scope of this course.

```
def ssg[T: Numeric](sq: Seq[T]): T = {
  val num = implicitly[Numeric[T]]
  import num._
  (sq map (x => x*x)).sum
  }
// ssg: [T](sq: Seq[T])(implicit evidence$1: Numeric[T])T
ssg(List(2,3,4))
// res10: Int = 29
ssg(Vector(1.0,2.0,3.0))
// res11: Double = 14.0
```

It is still a little ugly. *Spire* is a nice Scala library for numerical programming in Scala. We don't have time to explore it in detail in this course, but using it, we can solve this problem in a slightly neater way.

`https://github.com/non/spire`

```
import spire.math._
import spire.implicits._
def ss[T: Numeric](sq: Seq[T]): T =
  (sq map (x=>x*x)).reduce(_+_)
ss(List(1,2,3))
// res0: Int = 14
ss(Vector(2.0,3.0,4.0))
// res1: Double = 29.0
```

By bringing in Spire implicits associated with the `Numeric` typeclass and declaring that `T` belongs to the numeric typeclass, we ensure that the multiplication operation `*` is available for objects of type `T`.

Note that *Breeze* (which we will be discussing more later) has a dependency on Spire, so if you have a REPL with a dependency on Breeze, you do not need to add an additional dependency on Spire in order to use it.

3.2 Parallel collections

3.2.1 Introduction

In addition to the regular (serial) collections built in to the Scala standard library, there are also parallel versions of many of the collection classes. For example, the parallel version of `Vector` is `ParVector` and the parallel version of `Array` is `ParArray`. A serial collection can be converted to a parallel version by calling the method `.par`.

```
(1 to 12).par
// res0: ParRange = ParRange(1,2,3,4,5,6,7,8,9,10,11,12)
Vector(2,4,6,8).par
```

```
// res1: ParVector[Int] = ParVector(2, 4, 6, 8)
Array("a","b","c").par
// res2: ParArray[String] = ParArray(a, b, c)
List(1,2,3,4).par
// res3: ParSeq[Int] = ParVector(1, 2, 3, 4)
```

Note in the final example above, calling `.par` on a `List` returns a `ParVector`. There is no `ParList`, since a linked list is a fundamentally serial data structure. It should be clear that mapping a pure function over a collection can be parallelised easily, since the mapping of each element can happen independently of the mapping of any other element. Thus, a mapping operation can be split among available threads or processors to give a speed–up. The precise speed–up obtained in practice will depend on many factors, such as the size of the collection and the computational complexity of the function to be mapped. But in the ideal case, it should be possible to get close to an N times speed–up by using N cores or processors, and in the theoretical limit of infinite parallel hardware, a `map` on a collection of size n should have its time–complexity reduced from $\mathcal{O}(n)$ to $\mathcal{O}(1)$. `filter` operations parallelise similarly.

Reduction operations can also be parallelised, provided that the reduction function is *associative*, using a technique called *tree–reduction*. The details are not important here, but in the theoretical best case limit on infinite parallel hardware it should reduce the time–complexity of a `fold` or `reduce` operation from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$.

```
def isPrime(n: Int): Boolean = (2 until n) forall (
                                  n % _ != 0)
// isPrime: (n: Int)Boolean
(2 to 20) filter isPrime
// res4: IndexedSeq[Int] = Vector(2,3,5,7,11,13,17,19)
((2 to 100000) filter isPrime).length
// res5: Int = 9592
((2 to 100000).par filter isPrime).length
// res6: Int = 9592
(100000000 to 100000100) filter isPrime
// res7: IndexedSeq[Int] = Vector(100000007, 100000037, 10000000
(100000000 to 100000100).par filter isPrime
// res8: ParSeq[Int] = ParVector(100000007, 100000037, 100000039
```

By entering these commands into a Scala REPL, it should be clear that filtering a large collection executes significantly faster when using a parallel collection. The final example should also illustrate that the collection doesn't have to be large to benefit from parallelisation, so long as the function used to map or filter is expensive. Again, if you want to work with numbers bigger than `Int.MaxValue = 2,147,483,647` (roughly 2 billion) you can switch to `BigInt`s.

Scala auto-detects the number of cores available on a system and sets the level of parallelism used accordingly. You can easily find out how many threads it will use.

Typically you will want the default level of parallelism, but if you don't, you will typically want to change the default, and use that new default for all parallel collections in a given program, rather than to explicitly set the parallelism level on each collection. In earlier versions of Scala this used to be possible, but that functionality has been removed from the standard library. A solution to this problem can be found on Stack–Overflow: http://stackoverflow.com/questions/17865823/how-do-i-set-the-default-number-of-threads-for-scala-2-10-parallel-collections

```
List().par.tasksupport.parallelismLevel
// res16: Int = 8
```

It is possible to change the level of parallelism used on a per-collection basis (see the on-line docs for details).

3.2.2 Writing parallelism–agnostic code

We previously considered the problem of writing collection–generic code. That is, code which will work when passed in any sequential collection, irrespective of whether it is an `Array`, `List`, `Vector`, etc. The trick was to parametrise the method signature using the `Seq` collection supertype. But this doesn't work for parallel collections as the following generic sum–of–squares method illustrates.

```
def sss(sq: Seq[Int]): Int = (sq map (x => x*x)).
  reduce(_+_)
// sss: (sq: Seq[Int])Int
sss(List(2,3,4))
// res9: Int = 29
sss(Vector(2,3,4))
// res10: Int = 29
sss(List(2,3,4).par)
// <console>:9: error: type mismatch;
//  found   : scala.collection.parallel.immutable.ParSeq[Int]
//  required: Seq[Int]
//               sss(List(2,3,4).par)
//                                ^
```

The problem is that `Seq` is not a supertype of the parallel collections. `Seq` is a supertype of the *serial* sequential collections. `ParSeq` is the supertype of the *parallel* sequential collections. So if we want to write code that is agnostic to the kind of parallel collection used, we can used `ParSeq`. However, for code that is agnostic to whether a serial or parallel collection is used, we need to know that there is a class `GenSeq` that is a parent of `Seq` and `ParSeq`. So if we want to write code that is parallelism–agnostic, `GenSeq` is often what we want. This requires an **import**.

```
import scala.collection.GenSeq
// import scala.collection.GenSeq
def ssp(sq: GenSeq[Int]): Int = (sq map (x => x*x)).
  reduce(_+_)
// ssp: (sq: scala.collection.GenSeq[Int])Int
ssp(List(2,3,4))
// res12: Int = 29
ssp(List(2,3,4).par)
// res13: Int = 29
ssp(Vector(2,3,4).par)
// res14: Int = 29
ssp(Array(2,3,4).par)
// res15: Int = 29
```

This approach works fine so long as we don't need to return a collection. If we want to return a collection of the same type as passed in, we need to use the same sub-typing tricks that we used with `Seq` earlier, or typeclasses, which we will consider in the final chapter.

3.3 Collections as monads

3.3.1 First steps with monads in Scala

Before leaving the topic of collections, it is worth pointing out that Scala collections turn out be a special case of the more general functional programming notion of a *monad*. This is why Scala collections are often referred to as *monadic* collections, and why collection classes are often referred to as monads, such as the *List monad*. Consequently, Scala collections provide quite an intuitive introduction to the very powerful notion of a monad. Monads are one of those concepts that turns out to be very simple and intuitive once you "get it", but completely impenetrable until you do! The term "monad" is borrowed from that of the corresponding concept in category theory. The connection between functional programming and category theory is strong and deep. We will explore this a little more later, but for now the connection is not important and no knowledge of category theory is assumed.

Functors and Monads

Maps and Functors The `map` method is one of the first concepts one meets when beginning functional programming. It is a higher order method on many (immutable) *collection* and other *container* types. Let's start by reminding ourselves of how `map` operates on `Lists`.

```
val x = (0 to 4).toList
// x: List[Int] = List(0, 1, 2, 3, 4)
val x2 = x map { x => x * 3 }
// x2: List[Int] = List(0, 3, 6, 9, 12)
val x3 = x map { _ * 3 }
// x3: List[Int] = List(0, 3, 6, 9, 12)
val x4 = x map { _ * 0.1 }
// x4: List[Double] = List(0.0, 0.1, 0.2, 0.30000000000000004,
```

The last example shows that a `List[T]` can be converted to a `List[S]` if `map` is passed a function of type `T =>S`. Of course there's nothing particularly special about `List` here. It works with other collection types in the same way, as the following example with (immutable) `Vector` illustrates:

```
val xv = x.toVector
// xv: Vector[Int] = Vector(0, 1, 2, 3, 4)
val xv2 = xv map { _ * 0.2 }
// xv2: Vector[Double] = Vector(0.0, 0.2, 0.4, 0.60000000000000
val xv3 = for (xi <- xv) yield (xi * 0.2)
// xv3: Vector[Double] = Vector(0.0, 0.2, 0.4, 0.60000000000000
```

Note here that the **for** comprehension generating `xv3` is exactly equivalent to the `map` call generating `xv2` — the for–comprehension is just *syntactic sugar* for the `map` call. The benefit of this syntax will become apparent in the more complex examples we consider later. Many collection and other container types have a `map` method that behaves this way. Any parametrised type that does have a `map` method like this is known as a Functor. Again, the name is due to category theory. From a Scala-programmer perspective, a functor can be thought of as a trait, in pseudo-code as

```
trait F[T] {
  def map(f: T => S): F[S]
}
```

with `F` representing the functor. In fact it turns out to be better to think of a functor as a type class, but as previously mentioned, that is a concept we will return to later. Also note that to be a functor in the strict sense (from a category theory perspective), the map method must behave sensibly — that is, it must satisfy the functor laws. But again, I'm keeping things informal and intuitive for now.

FlatMap and Monads Once we can map functions over elements of containers, we soon start mapping functions which themselves return values of the container type. eg. we can map a function returning a `List` over the elements of a `List`, as illustrated below.

```
val x5 = x map { x => List(x - 0.1, x + 0.1) }
// x5: List[List[Double]] = List(List(-0.1, 0.1),
//   List(0.9, 1.1), List(1.9, 2.1), List(2.9, 3.1),
//   List(3.9, 4.1))
```

Clearly this returns a list-of-lists. Sometimes this is what we want, but very often we actually want to *flatten* down to a single list so that, for example, we can subsequently map over all of the elements of the base type with a single map. We could take the list-of-lists and then flatten it, but this pattern is so common that the act of mapping and then flattening is often considered to be a basic operation, usually known in Scala as `flatMap`. So for our toy example, we could carry out the `flatMap` as follows.

```
val x6 = x flatMap { x => List(x - 0.1, x + 0.1) }
// x6: List[Double] = List(-0.1, 0.1, 0.9, 1.1, 1.9,
//   2.1, 2.9, 3.1, 3.9, 4.1)
```

The ubiquity of this pattern becomes more apparent when we start thinking about iterating over multiple collections. For example, suppose now that we have two lists, `x` and `y`, and that we want to iterate over all pairs of elements consisting of one element from each list.

```
val y = (0 to 12 by 2).toList
// y: List[Int] = List(0, 2, 4, 6, 8, 10, 12)
val xy = x flatMap { xi => y map { yi => xi * yi } }
// xy: List[Int] = List(0, 0, 0, 0, 0, 0, 0, 0, 2, 4, 6,
//   8, 10, 12, 0, 4, 8, 12, 16, 20, 24, 0, 6, 12, 18,
//   24, 30, 36, 0, 8, 16, 24, 32, 40, 48)
```

This pattern of having one or more nested `flatMaps` followed by a final `map` in order to iterate over multiple collections is very common. It is exactly this pattern that the for–comprehension is syntactic sugar for. So we can re–write the above using a for–comprehension

```
val xy2 = for {
  xi <- x
  yi <- y
} yield (xi * yi)
```

```
// xy2: List[Int] = List(0, 0, 0, 0, 0, 0, 0, 0, 2, 4,
//   6, 8, 10, 12, 0, 4, 8, 12, 16, 20, 24, 0, 6, 12,
//   18, 24, 30, 36, 0, 8, 16, 24, 32, 40, 48)
```

This for–comprehension (usually called a for–expression in Scala) has an intuitive syntax reminiscent of the kind of thing one might write in an imperative language. But it is important to remember that <− is not actually an imperative assignment. The for–comprehension really does expand to the pure-functional nested `flatMap` and `map` call given above. Recalling that a functor is a parametrised type with a `map` method, we can now say that a *monad* is just a functor which also has a `flatMap` method. We can write this in pseudo-code as

```
trait M[T] {
  def map(f: T => S): M[S]
  def flatMap(f: T => M[S]): M[S]
}
```

Not all functors can have a flattening operation, so not all functors are monads, but all monads are functors. Monads are therefore more powerful than functors. Of course, more power is not always good. The *principle of least power* is one of the main principles of functional programming, but monads are useful for sequencing dependent computations, as illustrated by for–comprehensions. In fact, since for–comprehensions de–sugar to calls to `map` and `flatMap`, monads are precisely what are required in order to be usable in for–comprehensions. Collections supporting `map` and `flatMap` are referred to as *monadic*. Most Scala collections are monadic, and operating on them using `map` and `flatMap` operations, or using for–comprehensions is referred to as *monadic–style*. People will often refer to the monadic nature of a collection (or other container) using the word monad, eg. the "List monad". So far the functors and monads we have been working with have been collections, but not all monads are collections, and in fact collections are in some ways atypical examples of monads. Many monads are *containers* or *wrappers*, so it will be useful to see examples of monads which are not collections.

3.3.2 Option monad

One of the first monads that many people encounter is the `Option` monad (referred to as the Maybe monad in Haskell, and Optional in Java 8). You can think of it as being a strange kind of "collection" that can contain at most one element. So it will either contain an element or it won't, and so can be used to wrap the result of a computation which might fail. If the computation succeeds, the value computed can be wrapped in the Option (using the type `Some`), and if it fails, it will not contain a value of the required type, but simply be the value `None`. It provides a referentially transparent and type-safe alternative to raising exceptions or returning `NULL` references. We can transform Options using `map`.

```
val three = Option(3)
// three: Option[Int] = Some(3)
```

```
val twelve = three map (_ * 4)
// twelve: Option[Int] = Some(12)
```

But when we start combining the results of multiple computations that could fail, we run into exactly the same issues as before.

```
val four = Option(4)
// four: Option[Int] = Some(4)
val twelveB = three map (i => four map (i * _))
// twelveB: Option[Option[Int]] = Some(Some(12))
```

Here we have ended up with an Option wrapped in another Option, which is not what we want. But we now know the solution, which is to replace the first `map` with `flatMap`, or better still, use a for–comprehension.

```
val twelveC = three flatMap (i => four map (i * _))
// twelveC: Option[Int] = Some(12)
val twelveD = for {
  i <- three
  j <- four
} yield (i * j)
// twelveD: Option[Int] = Some(12)
```

Again, the for-comprehension is a little bit easier to understand than the chaining of calls to `flatMap` and `map`. Note that in the for–comprehension we don't worry about whether or not the `Option`s actually contain values — we just concentrate on the "happy path", where they both do, safe in the knowledge that the `Option` monad will take care of the failure cases for us. Two of the possible failure cases are illustrated below.

```
val oops: Option[Int] = None
// oops: Option[Int] = None
val oopsB = for {
  i <- three
  j <- oops
} yield (i * j)
// oopsB: Option[Int] = None
val oopsC = for {
  i <- oops
  j <- four
} yield (i * j)
// oopsC: Option[Int] = None
```

This is a typical benefit of code written in a monadic style. We chain together multiple computations thinking only about the canonical case and trusting the monad to take care of any additional computational context for us.

IEEE floating point and NaN Those with a background in scientific computing are probably already familiar with the `NaN` value in IEEE floating point. In many regards, this value and the rules around its behaviour mean that `Float` and `Double` types in IEEE compliant languages behave as an Option monad with `NaN` as the `None` value. This is simply illustrated below.

```
val nan = Double.NaN
3.0 * 4.0
// res0: Double = 12.0
3.0 * nan
// res1: Double = NaN
nan * 4.0
// res2: Double = NaN
```

The `NaN` value arises naturally when computations fail. eg.

```
val nanB = 0.0 / 0.0
// nanB: Double = NaN
```

This Option-like behaviour of `Float` and `Double` means that it is quite rare to see examples of `Option[Float]` or `Option[Double]` in Scala code. But note that this only works for `Floats` and `Doubles`, and not for other types, including, say, `Int`.

```
val nanC=0/0
// This raises a runtime exception!
```

But there are some disadvantages of the IEEE approach, as discussed elsewhere: `https://blogs.janestreet.com/making-something-out-of-nothing-or-why-none-is-better-than-nan-and-null/`

Option for matrix computations Good practical examples of scientific computations which can fail crop up frequently in numerical linear algebra, so it's useful to see how `Option` can simplify code in that context. Note that the code in this section requires the Breeze library, which we will look at in more detail in the next Chapter, so should be run from an `sbt console` using an `sbt` build file including a Breeze dependency. In statistical applications, one often needs to compute the Cholesky factorisation of a square symmetric matrix. This operation is built into Breeze as the function `cholesky`. However the factorisation will fail if the matrix provided is not positive semi-definite, and in this case the `cholesky` function will throw a runtime exception. We will use the built in `cholesky` function to create our own function, `safeChol` (using a monad called `Try` which is closely related to the `Option` monad) returning an `Option` of a matrix rather than a matrix. This function will not throw an exception, but instead return `None` in the case of failure, as illustrated below.

```
import breeze.linalg._
def safeChol(m: DenseMatrix[Double]):
  Option[DenseMatrix[Double]] =
    scala.util.Try(cholesky(m)).toOption
val m = DenseMatrix((2.0, 1.0), (1.0, 3.0))
val c = safeChol(m)
// c: Option[breeze.linalg.DenseMatrix[Double]] =
// Some(1.4142135623730951  0.0
// 0.7071067811865475  1.5811388300841898  )
val m2 = DenseMatrix((1.0, 2.0), (2.0, 3.0))
val c2 = safeChol(m2)
// c2: Option[breeze.linalg.DenseMatrix[Double]] = None
```

A Cholesky factorisation is often followed by a forward or backward solve. This operation may also fail, independently of whether the Cholesky factorisation fails. There doesn't seem to be a forward solve function directly exposed in the Breeze API, but we can easily define one, which I call `dangerousForwardSolve`, as it will throw an exception if it fails, just like the `cholesky` function. But just as for the `cholesky` function, we can wrap up the dangerous function into a safe one that returns an Option.

```
import com.github.fommil.netlib.BLAS.{getInstance => blas}
def dangerousForwardSolve(A: DenseMatrix[Double],
  y: DenseVector[Double]): DenseVector[Double] = {
  val yc = y.copy
  blas.dtrsv("L", "N", "N", A.cols, A.toArray, A.rows, yc.data, 1)
  yc
}
def safeForwardSolve(A: DenseMatrix[Double],
  y: DenseVector[Double]): Option[DenseVector[Double]] =
  scala.util.Try(dangerousForwardSolve(A, y)).toOption
```

Now we can write a very simple function which chains these two operations together, as follows.

```
def safeStd(A: DenseMatrix[Double], y: DenseVector[Double]):
  Option[DenseVector[Double]] = for {
  L <- safeChol(A)
  z <- safeForwardSolve(L, y)
} yield z

safeStd(m,DenseVector(1.0,2.0))
// res15: Option[breeze.linalg.DenseVector[Double]] =
//   Some(DenseVector(0.7071067811865475, 0.9486832980505138))
```

Note how clean and simple this function is, concentrating purely on the “happy path” where both operations succeed and letting the Option monad worry about the three different cases where at least one of the operations fails.

3.3.3 The Future monad

Let’s finish this Chapter with a monad for parallel and asynchronous computation, the `Future` monad. The Future monad is used for wrapping up slow computations and dispatching them to another thread for completion. The call to `Future` returns immediately, allowing the calling thread to continue while the additional thread processes the slow work. So at that stage, the `Future` will not have completed, and will not contain a value, but at some (unpredictable) time in the future it (hopefully) will (hence the name). In the following code snippet I construct two `Futures` that will each take at least 10 seconds to complete. On the main thread I then use a for–comprehension to chain the two computations together. Again, this will return immediately returning another `Future` that at some point in the future will contain the result of the derived computation. Then, purely for illustration, I force the main thread to stop and wait for that third future (`f3`) to complete, printing the result to the console.

```
import scala.concurrent.duration._
import scala.concurrent.{Future,ExecutionContext,Await}
import ExecutionContext.Implicits.global
val f1=Future{
  Thread.sleep(10000)
  1 }
val f2=Future{
  Thread.sleep(10000)
  2 }
val f3=for {
  v1 <- f1
  v2 <- f2
  } yield (v1+v2)
println(Await.result(f3,30.second))
```

When you paste this into your console you should observe that you get the result in 10 seconds, as `f1` and `f2` execute in parallel on separate threads. So the `Future` monad is an alternative to parallel collections to get started with parallel and async programming in Scala.

Here I've tried to give a quick informal introduction to the monad concept, and tried to use examples that will make sense to those interested in scientific and statistical computing. There's loads more to say about monads, and there are many more commonly encountered useful monads that haven't been covered here. I've skipped over lots of details, especially those relating to the formal definitions of functors and monads, including the laws that `map` and `flatMap` must satisfy and why. We'll return to these issues later.

Part II

Scientific and statistical computing

Chapter 4

Scala Breeze

4.1 A quick introduction

4.1.1 Introduction

In this Chapter I want to give a quick taste of the Breeze numerical library to indicate its capabilities. To follow along you should run a `console` from an `sbt` session with a Breeze dependency, or you could use a Scala Worksheet from inside a ScalaIDE project with a Breeze dependency.

4.1.2 Non-uniform random number generation

We begin by taking a quick look at everyone's favourite topic of non-uniform random number generation. Let's start by generating a couple of draws from a Poisson distribution with mean 3.

```
import breeze.stats.distributions._
// import breeze.stats.distributions._
val poi = Poisson(3.0)
// poi: Poisson = Poisson(3.0)
poi.draw
// res0: Int = 7
poi.draw
// res1: Int = 1
```

If more than a single draw is required, an iid `sample` can be obtained.

```
val x = poi.sample(10)
// x: IndexedSeq[Int] = Vector(7,2,2,2,4,3,4,2,0,4)
x
// res2: IndexedSeq[Int] = Vector(7,2,2,2,4,3,4,2,0,4)
x.sum
// res3: Int = 30
x.length
// res4: Int = 10
x.sum.toDouble/x.length
// res5: Double = 3.0
```

The probability mass function (PMF) of the Poisson distribution is also available.

```
poi.probabilityOf(2)
// res6: Double = 0.22404180765538775
x map {x => poi.probabilityOf(x)}
// res7: IndexedSeq[Double] = Vector(0.0216040314524838 2, ...
x map {poi.probabilityOf(_)}
// res8: IndexedSeq[Double] = Vector(0.0216040314524838 2, ...
```

Obviously, Gaussian variables (and Gamma, and several others) are supported in a similar way.

```
val gau=Gaussian(0.0,1.0)
// gau: Gaussian = Gaussian(0.0, 1.0)
gau.draw
// res9: Double = -1.051465465460726
gau.draw
// res10: Double = 2.4371714130683357
val y=gau.sample(20)
// y: IndexedSeq[Double] = Vector(0.8392352891144425, ...
y
// res11: IndexedSeq[Double] = Vector(0.8392352891144425, ...
y.sum/y.length
// res12: Double = -0.0678965896649274
y map {gau.logPdf(_)}
// res13: IndexedSeq[Double] = Vector(-1.2710964684521735, ...

Gamma(2.0,3.0).sample(5)
// res14: IndexedSeq[Double] = Vector(2.9045346978407203, ...
```

This is all good stuff for those of us who like to do Markov chain Monte Carlo. There are not masses of statistical data analysis routines built into Breeze, but a few basic tools are provided, including some basic summary statistics.

```
import breeze.stats._
// import breeze.stats._
mean(y)
// res15: Double = -0.06789658966492741
variance(y)
// res16: Double = 1.0934019878098706
meanAndVariance(y)
// res17: breeze.stats.MeanAndVariance = MeanAndVariance(
//   -0.06789658966492741,1.0934019878098706,20)
```

4.1.3 Vectors and matrices

Support for linear algebra is an important part of any scientific library. Here the Breeze developers have made the wise decision to provide a nice Scala interface to netlib-java. This in turn calls out to any native optimised BLAS or LAPACK libraries installed on the system, but will fall back to Java code if no optimised libraries are available. This means that linear algebra code using Scala and Breeze should run as fast as code written in any other language, including C, C++ and Fortran, provided that optimised libraries are installed on the system. For further details see the *Breeze linear algebra cheat sheet*. Let's start by creating and messing with a dense vector.

```
import breeze.linalg._
// import breeze.linalg._
val v=DenseVector(y.toArray)
// v: DenseVector[Double] = DenseVector(0.8392352891144425, ...
v(1) = 0
v
// res19: DenseVector[Double] = DenseVector(0.8392352891144425, ...
v(1 to 3) := 1.0
// res20: DenseVector[Double] = DenseVector(1.0, 1.0, 1.0)
v
// res21: DenseVector[Double] = DenseVector(0.8392352891144425,
//  1.0, 1.0, 1.0, −1.9181255616813044, 1.2403995799964067, ...
v(1 to 3) := DenseVector(1.0,1.5,2.0)
// res22: DenseVector[Double] = DenseVector(1.0, 1.5, 2.0)
v
// res23: DenseVector[Double] = DenseVector(0.8392352891144425,
// 1.0, 1.5, 2.0, −1.9181255616813044, 1.2403995799964067, ...
v >:> 0.0
// res24: BitVector = BitVector(0, 1, 2, 3, 5, 9, 10, 12, 17, 19)
(v >:> 0.0).toArray
// res25: Array[Boolean] = Array(true, true, true, true, false, ...
```

Next let's create and mess around with some dense matrices.

```
val m = new DenseMatrix(5,4,linspace(1.0,20.0,20).
  toArray)
// m: breeze.linalg.DenseMatrix[Double] =
// 1.0  6.0   11.0  16.0
// 2.0  7.0   12.0  17.0
// 3.0  8.0   13.0  18.0
// 4.0  9.0   14.0  19.0
// 5.0  10.0  15.0  20.0
m
// res26: breeze.linalg.DenseMatrix[Double] =
// 1.0  6.0   11.0  16.0
// 2.0  7.0   12.0  17.0
// 3.0  8.0   13.0  18.0
// 4.0  9.0   14.0  19.0
// 5.0  10.0  15.0  20.0
m.rows
// res27: Int = 5
m.cols
// res28: Int = 4
m(::,1)
// res29: DenseVector[Double] = DenseVector(
//     6.0,7.0,8.0,9.0,10.0)
m(1,::)
// res30: Transpose[DenseVector[Double]] =
//   Transpose(DenseVector(2.0, 7.0, 12.0, 17.0))
m(1,::) := linspace(1.0,2.0,4).t
// res31: Transpose[DenseVector[Double]] =
//   Transpose(DenseVector(1.0, 1.3333333333333333,
//     1.6666666666666665, 2.0))
m
// res32: breeze.linalg.DenseMatrix[Double] =
// 1.0  6.0                 11.0                16.0
// 1.0  1.3333333333333333  1.6666666666666665  2.0
// 3.0  8.0                 13.0                18.0
//  4.0   9.0                        14.0                      19.0
//  5.0   10.0                       15.0                      20.0
```

Note in the above how the `::` notation is used to mean *all*, so that `m(1,::)` means all entries for row 1 (which is the second row) and `m(::,1)` means all entries for column 1.

Users of R will be familiar with the `apply` command for applying functions to rows and columns of matrices. This is referred to as *broadcasting* in Breeze. In Breeze the `::` notation is accompanied by the `*` notation, where `*` denotes “foreach” — the index to be looped over and kept. So `sum(m(::,*))` means “foreach `k` ranging over the columns of `m`, compute `sum(m(::,k))` and keep those column sums in a row vector.

```
// broadcasting (like "apply" in R)
sum(m(::,*)) // column sum
// res33: Transpose[DenseVector[Double]] =
//  Transpose(DenseVector(14.0, 34.33333333333333, ...
sum(m(*,::)) // row sum
// res34: DenseVector[Double] =
//  DenseVector(34.0, 6.0, 42.0, 46.0, 50.0)

val n = m.t
// n: breeze.linalg.DenseMatrix[Double] =
// 1.0   1.0                 3.0   4.0   5.0
// 6.0   1.3333333333333333  8.0   9.0   10.0
// 11.0  1.6666666666666665  13.0  14.0  15.0
// 16.0  2.0                 18.0  19.0  20.0
val o = m*n
// o: breeze.linalg.DenseMatrix[Double] =
// 414.0              59.33333333333333  482.0
516.0              550.0
// 59.33333333333333  9.555555555555555  71.33333333333333
77.33333333333333  83.33333333333333
// 482.0              71.33333333333333  566.0
608.0              650.0
// 516.0              77.33333333333333  608.0
654.0              700.0
// 550.0              83.33333333333333  650.0
700.0              750.0
val p = n*m
// p: breeze.linalg.DenseMatrix[Double] =
// 52.0               117.33333333333333  182.66666666666666
248.0
// 117.33333333333333  282.77777777777777  448.2222222222223
613.6666666666667
// 182.66666666666666  448.2222222222223  713.7777777777778
979.3333333333334
// 248.0              613.6666666666667  979.3333333333334
1345.0

DenseMatrix.eye[Double](3)
// res35: breeze.linalg.DenseMatrix[Double] =
// 1.0  0.0  0.0
// 0.0  1.0  0.0
// 0.0  0.0  1.0
```

So, messing around with vectors and matrices is more-or-less as convenient as in well-known dynamic and math languages. To conclude this section, let us see how to simulate some data from a regression model and then solve the least squares problem to obtain the estimated regression coefficients. We will simulate 1,000 observations from a model with 2 covariates (plus an intercept).

```
val N = 1000
// N: Int = 1000
val P = 2
// P: Int = 2
val XX = new DenseMatrix(N,P,gau.sample(P*N).toArray)
// XX: breeze.linalg.DenseMatrix[Double] =
//  -0.7489949781984457     -0.5742924772515515
//  -0.5240133331383998     -0.4361331555220949
//  -0.5764525022050057     -1.2691562428327328
//  0.7326519916718431      0.19642905294418214
//  -1.1493500841218598     -1.458947619159962
//  0.08783097116056983     0.3500859440...
val X = DenseMatrix.horzcat(
  DenseVector.ones[Double](N).toDenseMatrix.t,
  XX)
// X: breeze.linalg.DenseMatrix[Double] =
//  1.0   -0.7489949781984457     -0.5742924772515515
//  1.0   -0.5240133331383998     -0.4361331555220949
//  1.0   -0.5764525022050057     -1.2691562428327328
//  1.0   0.7326519916718431      0.19642905294418214
//  1.0   -1.1493500841218598     -1.458947619159962
//  1.0   2.044959173084444       1.7879360132192752...
val b0 = linspace(1.0,2.0,P+1)
// b0: DenseVector[Double] = DenseVector(1.0,1.5,2.0)
val y0 = X * b0
// y0: DenseVector[Double] = DenseVector(
//  -1.2720774218007718, -0.6582863107517896, ...
val y = y0 + DenseVector(gau.sample(1000).toArray)
// y: DenseVector[Double] = DenseVector(
//  -1.2085936819146115, 0.27627139418016755, ...
// now fit model
val b = X \ y   // linear solve
// b: DenseVector[Double] = DenseVector(
//  1.0242406534162087, 1.4241005745776305,
//  1.9714098738779253)
```

Note the use of the notation `X \ y` for the solution of the linear equation $Xb = y$ (for b). In the case of a (square) invertible X, this will give $X^{-1}y$, but for an over-determined system (like this) it will give the b which minimises $\|Xb - y\|^2$ — the *least squares solution*, and this is computed using a QR–decomposition of X.

So all of the most important building blocks for statistical computing are included in the Breeze library.

4.1.4 Reading and writing matrices from and to disk

The easiest way to get data in and out of matrices is via Breeze's simple CSV reader and writer, as illustrated below.

```
import java.io.File
// import java.io.File
csvwrite(new File("x-matrix.csv"),X)
val X3 = csvread(new File("x-matrix.csv"))
// X3: breeze.linalg.DenseMatrix[Double] =
// 1.0  -0.7489949781984457   -0.5742924772515515
// 1.0  -0.5240133331383998   -0.4361331555220949
// 1.0  -0.5764525022050057   -1.2691562428327328
// 1.0  0.7326519916718431    0.19642905294418214
// 1.0  -1.1493500841218598   -1.458947619159962
// 1.0  2.044959173084444     1.787936013219275...
```

4.2 Numerical linear algebra

The discussion of the `\` operator near the end of the previous section hints at some of the numerical linear algebra routines that are included in Breeze. We will now look at these in a bit more detail.

4.2.1 Symmetric eigen–decomposition

```
val e=eigSym(p)
// e: breeze.linalg.eigSym.DenseEigSym =
// EigSym(DenseVector(-1.589173864914946E-13, ...
e.eigenvalues
// res37: DenseVector[Double] = DenseVector(
// -1.589173864914946E-13, 1.2783114180198827E-13,
// 7.857939135975561, 2385.6976164195803)
e.eigenvectors
// res38: breeze.linalg.DenseMatrix[Double] =
// 0.2906711712049423   -0.46423083722468117   -0.824890701078881
-0.13984037783705755
// -0.733578798035299   0.4023209503283427     -0.426695744781777
-0.3434104561382918
// 0.5951440824557162   0.588050611017405      -0.028500788484721
-0.546980534439526
// -0.1522364556253744  -0.5261407241210538    0.369694167812350
-0.7505506127407601
```

4.2.2 Singular value decomposition (SVD)

We start with the full SVD:

```
val s=svd(p) // full SVD
// s: breeze.linalg.svd.DenseSVD =
// SVD(-0.13984037783705727  -0.8248907010788752  ...
s.U
// res39: breeze.linalg.DenseMatrix[Double] =
// -0.13984037783705727  -0.8248907010788752    -0.007271090327
-0.5476742930295687
// -0.34341045613829163  -0.4266957447817886    0.417907103351
0.7248128399583851
// -0.5469805344395258   -0.028500788484715434  -0.814000935719
0.19339719917197024
// -0.7505506127407597   0.3696941678123502     0.403364922695
-0.3705357461007773
```

```
s.S
// res40: DenseVector[Double] = DenseVector(
//   2385.69761641958, 7.857939135975376,
//   3.5396639195325335E-14, 1.3843384666519122E-15)
s.Vt
// res41: breeze.linalg.DenseMatrix[Double] =
// -0.13984037783705763   -0.3434104561382918    -0.5469805344395259
-0.7505506127407602
// -0.8248907010788822    -0.42669574478178196   -0.028500788484708495
0.3696941678123435
// -0.3564872784300705    0.7852597534511133     -0.5010576716119877
0.0722851965909525
// 0.41583268307998733    -0.28873364821216885   -0.6700307528157005
0.5429317179478612
```

In statistical applications we rarely want the full SVD, as we typically deal with tall, thin $N \times p$ matrices with $N >> p$. In this case we can get the “thin” SVD, called `reduced` in Breeze.

```
val ts=svd.reduced(m) // thin SVD
// ts: breeze.linalg.svd.DenseSVD =
// SVD(-0.41409513816680055   0.7907147500096334  ...
ts.U
// res42: breeze.linalg.DenseMatrix[Double] =
// -0.41409513816680055   0.7907147500096334     0.4178037940705934
-0.1538134689154079
// -0.061634609473825375   -0.250403446766538    0.1326486825541926
-0.34803887195107813
// -0.4870128530154548    0.14117668550853296    -0.6056510931508465
0.5325848276234026
// -0.523471710439782      -0.18359234674201727  -0.37148053494455324
-0.6819416942192263
// -0.5599305678641092     -0.5083613789925672   0.5504845885211932
0.3263729269746367
ts.S
// res43: DenseVector[Double] = DenseVector(
//  48.843603638752754, 2.8032015867531554,
//  1.1493769006712675E-15, 5.567163475013936E-16)
ts.Vt
// res44: breeze.linalg.DenseMatrix[Double] =
// -0.13984037783705755   -0.3434104561382917    -0.5469805344395258
-0.7505506127407601
// -0.8248907010788697    -0.42669574478179484   -0.028500788484719514
0.3696941678123552
// 0.4927694123431541     -0.4787964643981302    -0.5207153082332023
0.5067423602881782
// 0.23911985751706738    -0.6861151111000621    0.6548706496489223
-0.20787539606592725
ts.U * diag(ts.S) * ts.Vt
// res45: breeze.linalg.DenseMatrix[Double] =
// 1.0000000000000009   6.000000000000004    11.000000000000007
16.000000000000014
// 1.0000000000000002   1.3333333333333335   1.6666666666666659
2.0
// 3.000000000000001    8.000000000000004    13.000000000000005
18.00000000000001
// 4.000000000000001    9.000000000000004    14.000000000000005
19.00000000000001
```

```
// 5.000000000000001   10.000000000000004   15.000000000000005
20.00000000000001
```

Crucially, unlike the full SVD, the reduced SVD does not involve any $N \times N$ matrices.

4.2.3 Principal components analysis (PCA)

Breeze has a function for PCA built-in: `breeze.linalg.PCA`. However, this works via a spectral decomposition of a covariance matrix (analagous to the **R** function `princomp`). Generally it is better to construct principal components directly from the SVD of the centred data matrix (analagous to the **R** function `prcomp`). We can write some simple code to do this as follows:

```
import breeze.linalg._
import breeze.stats._

case class Pca(mat: DenseMatrix[Double]) {
    val xBar = mean(mat(::,*)).t
    val x = mat(*,::) - xBar
    val SVD = svd.reduced(x)
    val loadings = SVD.Vt.t
    val sdev = SVD.S / math.sqrt(x.rows - 1)
    lazy val scores = x * loadings
  }
```

Case classes are a simple way to define data structures, and are used extensively in Scala code.

Calling this with `Pca(myMatrix)` will return an object of type `Pca`. See how the Breeze broadcast notation is used first to compute the column sums of a matrix, and then to *sweep out* the column sums from the matrix. Then the usual PCA attributes can be constructed simply from the SVD. Note how `scores` has been declared a `lazy val`, to be computed only if required. We can use this as follows.

```
val X = DenseMatrix((1.0,1.5),(1.5,2.0),(2.0,1.5))
val pca = Pca(X)
pca.sdev
pca.loadings
pca.scores
```

Note that the `loadings` defined here are consistent with how they are usually defined for PCA in languages such as R, but they are the *transpose* of the `loadings` returned by the Breeze PCA class.

4.2.4 QR and Cholesky decompositions

As above, we will typically want the reduced QR decomposition in statistical applications.

```
val q=qr.reduced(m) // thin QR
// q: breeze.linalg.qr.DenseQR =
// QR(-0.13867504905630734  0.881744764716757  ...
q.q
// res46: breeze.linalg.DenseMatrix[Double] =
// -0.13867504905630734  0.881744764716757     0.14026891908258
0.42793087216408154
// -0.1386750490563073   -0.21741651732741957  0.96220458385429
0.08741261434249865
```

```
//  -0.41602514716892186  0.2898886897698929    0.07426633834284485
-0.7332302324447282
//  -0.5547001962252291   -0.006039347703539422  -0.06813863044651419
-0.18698794753853681
//  -0.6933752452815364   -0.30196738517697175   -0.2105435992358733
0.48645980019635027
q.r
// res47: breeze.linalg.DenseMatrix[Double] =
//  -7.211102550927979   -16.271205755940056   -25.33130896095213
-34.39141216596421
//  0.0                  4.245661435588219     8.49132287117644
12.736984306764665
//  0.0                  0.0                   -9.930136612989092E-16
-3.574849180676073E-15
//  0.0                  0.0                   0.0
1.7455587204148017E-15


cholesky(DenseMatrix((3.0,1.0),(1.0,2.0)))
// res48: breeze.linalg.DenseMatrix[Double] =
// 1.7320508075688772  0.0
// 0.5773502691896258  1.2909944487358056
```

4.3 Scientific computing

4.3.1 Constants and special functions

There are some built-in constants.

```
import breeze.numerics.constants._
// import breeze.numerics.constants._
Pi
// res49: Double = 3.141592653589793
E
// res50: Double = 2.718281828459045
eulerMascheroni
// res51: Double = 0.5772156649015329
```

There are also various physics constants, but these typically aren't required in statistical applications. There are also a number of special functions.

```
import breeze.numerics._
// import breeze.numerics._
erf(2.0) // error function
// res52: Double = 0.9953222650189527
erfinv(erf(2.0))
// res53: Double = 1.999999999999999
sigmoid(1.0) // expit/logistic
// res54: Double = 0.7310585786300049
lgamma(4.0) // log gamma function
// res55: Double = 1.791759469228055
exp(lgamma(4.0))
// res56: Double = 6.0
gammp(2.0,1.0) // incomplete gamma
// res57: Double = 0.2642411176571152
```

```
digamma(2.0)
// res58: Double = 0.42278433220079321
lbeta(1.0,2.0) // log beta function
// res59: Double = -0.6931471805599453
Bessel.i0(1.0) // Bessel functions
// res60: Double = 1.2660658777520086
Bessel.i1(1.0)
// res61: Double = 0.5651591039924851
// UFuncs
erf(DenseVector(1.0,2.0,3.0))
// res62: DenseVector[Double] = DenseVector(
//  0.8427007929497151, 0.9953222650189527,
//  0.9999779095030014)
```

The last example shows that special functions can typically be used as Breeze `UFuncs` in vectorised operations.

4.3.2 Integration and ODE–solving

There a couple of simple functions for 1–D integration.

```
// https://github.com/scalanlp/breeze/wiki/Integration
import breeze.integrate._
// import breeze.integrate._
trapezoid(x => sin(x), 0, Pi, 10)
// res63: Double = 1.9796508112164832
simpson(x => sin(x), 0, Pi, 100)
// res64: Double = 2.0000000007042207
```

There are also functions for numerically integrating (forward–solving) ODE models.

```
val ode=new HighamHall54Integrator(0.001, 0.1)
// min and max time steps
// ode: HighamHall54Integrator =
//   HighamHall54Integrator@116e4be2
ode.integrate((x,t) => x, DenseVector(1.0),
  linspace(0,1,5).toArray)
// res65: Array[DenseVector[Double]] = Array(
//   DenseVector(1.0),
//   DenseVector(1.2840254150249555),
//   DenseVector(1.6487212664298634),
//   DenseVector(2.1170000083877234),
//   DenseVector(2.718281814377245))
```

4.3.3 Optimisation

There are also various routines for solving optimisation problems, with and without gradient information. LBFGS is a good default. It is a quasi-Newton method, but if gradients are not available a numerical gradient calculation can be used.

```
import breeze.optimize._
// import breeze.optimize._
def f(x: DenseVector[Double]) = (x(0)-5.0)*(x(0)-5.0)
// f: (x: DenseVector[Double])Double
val ag = new ApproximateGradientFunction(f)
```

```
// ag: ApproximateGradientFunction[Int,
// DenseVector[Double]] = <function1>
val opt = new LBFGS[DenseVector[Double]]()
// opt: LBFGS[DenseVector[Double]] = LBFGS@69398816
opt.minimize(ag,DenseVector(0.0))
// res66: DenseVector[Double] = DenseVector(4.999995)
```

But if gradients are available it's better to use them.

```
def eg(x: DenseVector[Double]) =
  DenseVector(2.0*x(0) - 10.0)
// eg: (x: DenseVector[Double])DenseVector[Double]
val df = new DiffFunction[DenseVector[Double]] {
  def calculate(x: DenseVector[Double]) = (f(x),eg(x))
}
// df: DiffFunction[DenseVector[Double]] = <function1>
opt.minimize(df,DenseVector(0.0))
// res67: DenseVector[Double] = DenseVector(5.0)
```

4.4 Breeze–viz

Breeze–viz is the plotting library for Breeze. Note that this is separate from the main Breeze library, and therefore requires an additional dependency in your SBT build file:

```
  "org.scalanlp" %% "breeze-viz" % "0.13"
```

Then from a REPL:

```
import breeze.plot._
// import breeze.plot._
val fig = Figure("My Figure")
// fig: Figure = Figure@2e23d4da
fig.width=1000
// fig.width: Int = 1000
fig.height=800
// fig.height: Int = 800
val p1 = fig.subplot(0)
// p1: Plot = breeze.plot.Plot@593c763a
p1 += hist(y,50)
// res68: Plot = Plot@593c763a
p1.xlim = (-10,15)
// p1.xlim: (Double, Double) = (-10.0,15.0)
p1.xlabel = "y"
// p1.xlabel: String = y
p1.title = "Distribution of observed response"
// p1.title: String = Distribution of observed response
fig.refresh
fig.saveas("hist.pdf") // or "eps", or "png" for bitmap
```

So `Figure` corresponds to a window on your graphics device, and within a window you can have multiple sub–plots, but above we just have one, so it has index 0. You can add stuff to a `subplot` with the `+=` operator. Here we just add a histogram to an empty subplot. Note that there are lots of options which can be tweaked. It can sometimes be useful to call `refresh` to get the plot to update. Plots can be saved in vector format as PDF or EPS, or in bitmap format as PNG. Also note that plots are interactive, so right–clicking on a subplot will give a menu for manual tweaking of a plot. We can add a new subplot with

```
val p2=fig.subplot(1,2,1) // rows, cols, subplot index
// p2: breeze.plot.Plot = breeze.plot.Plot@6f6f7d5f
```

Here the arguments to `subplot` are the number of rows and columns in the array of subplots to be displayed in the figure, and the final number is the index of the subplot in question — here `1`, the second subplot in the array (row-major ordering). We can then add stuff to the subplot as before. Here we will first add some points and then a line.

```
p2 += plot(y0,y,'+',colorcode="red")
// res71: Plot = Plot@6f6f7d5f
val xvals=linspace(min(y0),max(y0),2)
// xvals: DenseVector[Double] = DenseVector(
//   -6.869237935312322, 9.103663726097341)
p2 += plot(xvals,xvals,colorcode="[40,40,200]")
// res72: Plot = Plot@6f6f7d5f
p2.xlabel = "y0"
// p2.xlabel: String = y0
p2.ylabel = "y"
// p2.ylabel: String = y
p2.title = "Observed␣against␣true␣response"
// p2.title: String = Observed against true response
```

The final result of the above commands is shown in Figure 4.1. Some more plotting features are illustrated below: producing an image of a matrix and plotting multiple functions with a legend. The result of this is shown in Figure 4.2.

```
val fig2 = Figure("More␣plots")
// fig2: Figure = Figure@246f133f
fig2.width=800
// fig2.width: Int = 800
fig2.height=600
// fig2.height: Int = 600
val p3 = fig2.subplot(1,2,0)
// p3: Plot = Plot@7f2d3ec2
p3 += image(X)
// res73: Plot = Plot@7f2d3ec2
p3.xlabel = "p"
// p3.xlabel: String = p
p3.ylabel = "N"
// p3.ylabel: String = N
p3.title = "Covariate␣matrix"
// p3.title: String = Covariate matrix

val p4 = fig2.subplot(1,2,1)
// p4: breeze.plot.Plot = breeze.plot.Plot@7d3d1f9e
val xs = linspace(0,4,200)
// xs: DenseVector[Double] = DenseVector(0.0,
//  0.020100502512562814, 0.04020100502512563, ...
p4 += plot(xs,xs map (Bessel.i0(_:Double)),name="i0")
// res74: Plot = Plot@7d3d1f9e
p4 += plot(xs,xs map (Bessel.i1(_:Double)),name="i1")
```

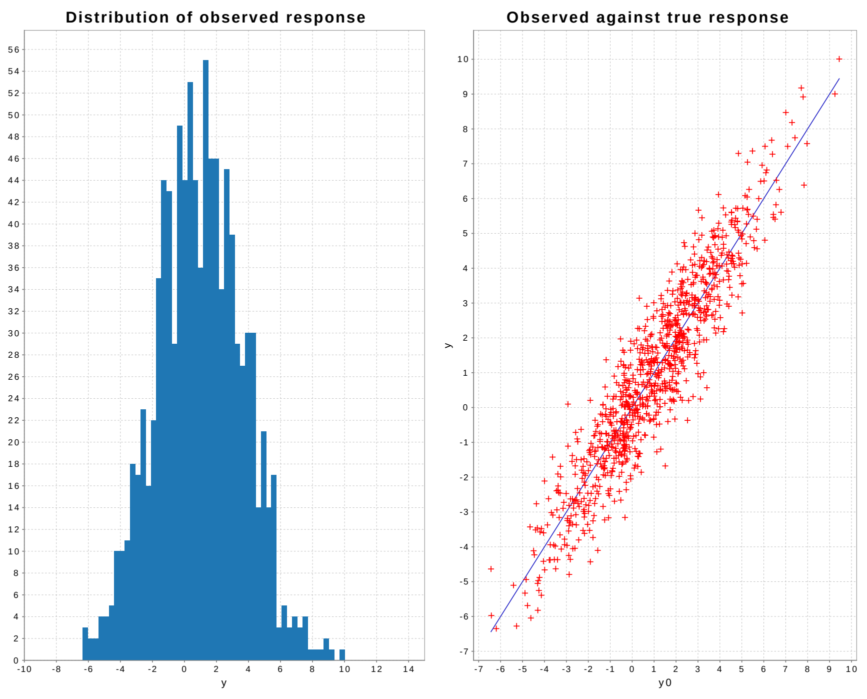

Figure 4.1: A Breeze–viz Figure with two subplots: a histogram and a scatter–plot

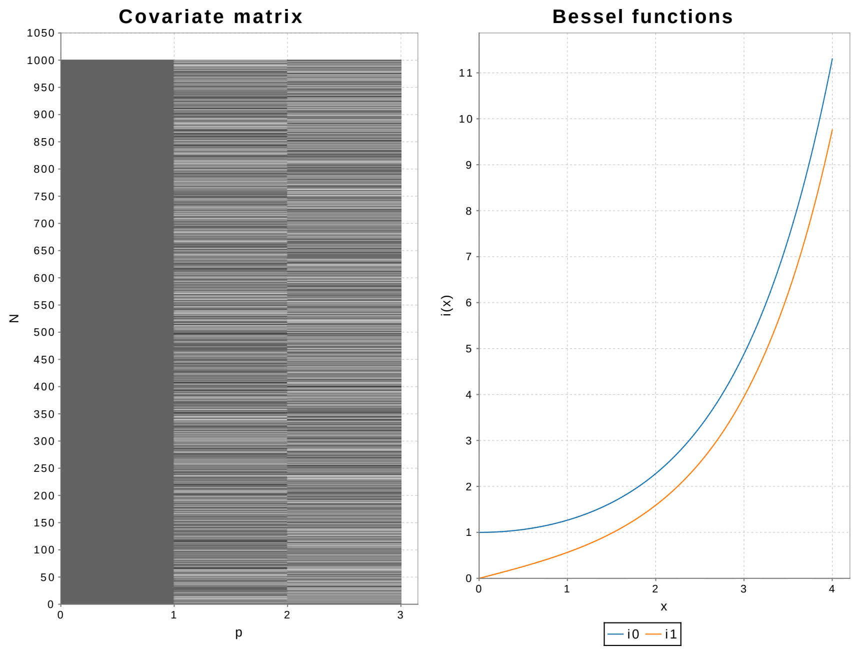

Figure 4.2: A Breeze–viz Figure with two subplots: a matrix image and a line graph

```
// res75: Plot = Plot@7d3d1f9e
p4.legend = true
// p4.legend: Boolean = true
p4.xlabel = "x"
// p4.xlabel: String = x
p4.ylabel = "i(x)"
// p4.ylabel: String = i(x)
p4.title = "Bessel␣functions"
// p4.title: String = Bessel functions
```

Chapter 5

Monte Carlo simulation

Monte Carlo is a standard technique in statistical computing for avoiding integration problems associated with marginalisation or expectation calculations. Some (pure) Monte Carlo algorithms parallelise well, and others, such as Markov chain Monte Carlo (MCMC), don't. Amenability to parallelisation is closely related to amenability to vectorisation, so MCMC algorithms don't vectorise well, either. This means that MCMC algorithms tend to run slowly in dynamic and interpreted languages. The desire to want to write fast efficient Monte Carlo algorithms is one reason why statisticians might want to consider using a fast, efficient, compiled language like Scala.

5.1 Parallel Monte Carlo

5.1.1 Monte Carlo integration

We begin with a very simple Monte Carlo algorithm. Suppose that we want to construct a Monte Carlo estimate of the integral

$$I = \int_0^1 \exp\{-u^2\}du.$$

We first note that we can consider this integral as an expectation

$$I = \mathrm{E}(\exp\{-U^2\}),$$

with the expectation taken wrt to a $U(0,1)$ random variable, U. Then given iid realisations $u_1, u_2, \ldots, u_n$ of U, we can form the Monte Carlo estimate

$$\hat{I}_n = \frac{1}{n}\sum_{i=1}^{n} \exp\{-u_i^2\}.$$

Some very simple serial serial Scala code to implement this algorithm follows.

```
import java.util.concurrent.ThreadLocalRandom
import scala.math.exp
import scala.annotation.tailrec

val N = 1000000L
def rng = ThreadLocalRandom.current()
def mc(its: Long): Double = {
  @tailrec def sum(its: Long, acc: Double): Double = {
     if (its == 0) acc else {
       val u = rng.nextDouble()
       sum(its-1, acc + exp(-u*u))
     }
  }
  sum(its,0.0)/its
}

mc(N)
// res0: Double = 0.7469182341226777
```

`ThreadLocalRandom` is a parallel random number generator introduced into recent versions of the Java programming language, which can be easily utilised from Scala code. It is the default generator used by Breeze.

Note that this code has no dependency on Breeze. The above code is serial, but fast — comparable with serial code written in C/C++. Now that we have a good working implementation we should think how to parallelise it.

5.1.2 Parallel implementation

As we have already seen, the simplest way to introduce parallelisation into Scala code is to parallelise a `map` over a collection. We therefore need a collection and a `map` to apply to it. Here we will just divide our `N` iterations into (say) 4 separate computations, and use a `map` to compute the required Monte Carlo sums.

```
def mcp(its: Long,np: Int = 4): Double =
  (1 to np).par.map(i => mc(its/np)).sum/np

mcp(N)
// res1: Double = 0.7468289488326496
```

Running this new code confirms that it works and gives similar estimates for the Monte Carlo integral as the previous version. It's also faster, though we will look at timings shortly. The function `par` converts the collection (here a `Range`) to a parallel collection, and then subsequent `map`s, `filter`s, etc., can be computed in parallel on appropriate multicore architectures.

5.1.3 Timings, and varying the size of the parallel collection

We can define a function to do a simple timing with

```
def time[A](f: => A) = {
    val s = System.nanoTime
    val ret = f
    println("time: "+(System.nanoTime-s)/1e6+"ms")
    ret
  }
```

which we can test as follows.

```
val bigN = 100000000L
// bigN: Long = 100000000
```

```
time(mc(bigN))
// time: 6225.859951ms
// res2: Double = 0.7468159872240743
time(mcp(bigN))
// time: 2197.872294ms
// res3: Double = 0.7468246533834739
```

So this works, and confirms that the parallel version appears to be around 3 times faster than the serial version on my laptop. Now benchmarking code on the JVM is non-trivial, for various reasons. We will look at this topic later if time permits. For now, we can now run this code with varying sizes of `np`, and do each test 3 times, in order to see how the runtime of the code changes as the size of the parallel collection increases. Timings are all done on my laptop.

```
(1 to 12).foreach{i =>
  println("np = "+i)
  (1 to 3).foreach(j => time(mcp(bigN,i)))
}
// np = 1
// time: 6201.480532ms
// time: 6186.176627ms
// time: 6198.14735ms
// np = 2
// time: 3127.512337ms
// time: 3122.648652ms
// time: 3148.509354ms
// np = 3
// time: 2488.273962ms
// time: 2402.957878ms
// time: 2555.286948ms
// np = 4
// time: 2133.996ms
// time: 2238.847511ms
// time: 2177.260599ms
// np = 5
// time: 2867.889727ms
// time: 2890.128312ms
// time: 2784.020295ms
// np = 6
// time: 3358.373499ms
// time: 2600.759805ms
// time: 2559.704485ms
// np = 7
// time: 3248.162029ms
// time: 3359.006061ms
// time: 2882.463352ms
// np = 8
// time: 1847.027762ms
// time: 2545.40533ms
// time: 2556.063328ms
// np = 9
// time: 2344.998373ms
// time: 2253.718886ms
// time: 2260.407902ms
// np = 10
// time: 2158.32923ms
// time: 2125.176623ms
// time: 2049.69822ms
// np = 11
// time: 1945.826366ms
// time: 1945.175903ms
// time: 1952.519595ms
// np = 12
// time: 1822.598809ms
// time: 1827.48165ms
// time: 2722.349404ms
```

So we see that the timings decrease steadily until the size of the parallel collection hits 4 (my laptop is a Quad-core), and then increases very slightly, but not much as the size of the collection increases, with another sweet-spot at 8 (the number of processors my hyper-threaded Quad-core presents to Linux). The Scala compiler and JVM runtime manage an appropriate number of threads for the collection irrespective of the actual size of the collection.

5.1.4 Rejection sampling

Rejection sampling is another technique that can be used for Monte Carlo integration. The implementation issues are very similar to the Monte Carlo integration algorithms we have just considered. There is a complete example of a rejection sampling algorithm in the course repo examples directory. This example uses Breeze, for illustrative purposes.

5.2 MCMC

Pure Monte Carlo is concerned with generating iid samples from a distribution of interest. Often, and especially for high dimensional distributions, this can be difficult. It turns out that it is often easier to generate a Markov chain with an equilibrium distribution corresponding to a distribution of interest. In applications of Bayesian statistics, this distribution is the posterior distribution, but MCMC has other applications, including statistical physics, from where the techniques originate.

5.2.1 Metropolis-Hastings algorithms

We will begin our investigation of MCMC methods by considering a very simple one-dimensional problem. This will allow us to focus on the structure of the problem and the code, and not get lost in the technical details of building complex samplers.

A simple MH sampler

I will just consider a trivial toy Metropolis algorithm using a Uniform random walk proposal to target a standard normal distribution. So, if the current state of a Markov chain is x, a new candidate $x' = x + u$

is formed, where $u \sim U(-\varepsilon, \varepsilon)$, for some fixed tuning parameter ε. This candidate should be accepted as the new state of the chain with probability $\min\{1, A\}$, where $A = \phi(x')/\phi(x)$, and $\phi(.)$ is the standard normal density, otherwise the new state of the chain should be the old value, x.

A fairly direct translation of this algorithm into Scala (using Breeze) is:

```scala
def metrop1(n: Int = 1000, eps: Double = 0.5):
  DenseVector[Double] = {
    val vec = DenseVector.fill(n)(0.0)
    var x = 0.0
    var oldll = Gaussian(0.0, 1.0).logPdf(x)
    vec(0) = x
    (1 until n).foreach { i =>
      val can = x + Uniform(-eps, eps).draw
      val loglik = Gaussian(0.0, 1.0).logPdf(can)
      val loga = loglik - oldll
      if (math.log(Uniform(0.0, 1.0).draw) < loga) {
        x = can
        oldll = loglik
      }
      vec(i) = x
    }
    vec
}
```

This code works, and is reasonably fast and efficient, but there are several issues with it from a functional programmers perspective. One issue is that we have committed to storing all MCMC output in RAM in a `DenseVector`. This probably isn't an issue here, but for some big problems we might prefer to not store the full set of states, but to just print the states to (say) the console, for possible re-direction to a file. It is easy enough to modify the code to do this:

```scala
def metrop2(n: Int = 1000, eps: Double = 0.5): Unit =
{
    var x = 0.0
    var oldll = Gaussian(0.0, 1.0).logPdf(x)
    (1 to n).foreach { i =>
      val can = x + Uniform(-eps, eps).draw
      val loglik = Gaussian(0.0, 1.0).logPdf(can)
      val loga = loglik - oldll
      if (math.log(Uniform(0.0, 1.0).draw) < loga) {
        x = can
        oldll = loglik
      }
      println(x)
    }
}
```

But now we have two versions of the algorithm. One for storing results locally, and one for streaming results to the console. This is clearly unsatisfactory, but we shall return to this issue shortly. Another issue that will jump out at functional programmers is the reliance on mutable variables for storing the state and old likelihood. Let's fix that now by re-writing the algorithm as a tail-recursion.

```
@tailrec
def metrop3(n: Int = 1000, eps: Double = 0.5,
  x: Double = 0.0, oldll: Double = Double.MinValue):
  Unit = {
    if (n > 0) {
      println(x)
      val can = x + Uniform(-eps, eps).draw
      val loglik = Gaussian(0.0, 1.0).logPdf(can)
      val loga = loglik - oldll
      if (math.log(Uniform(0.0, 1.0).draw) < loga)
        metrop3(n - 1, eps, can, loglik)
      else
        metrop3(n - 1, eps, x, oldll)
    }
  }
```

This has eliminated the vars, and is just as fast and efficient as the previous version of the code. However, this is for the print-to-console version of the code. What if we actually want to keep the iterations in RAM for subsequent analysis? We can keep the values in an accumulator, as follows.

```
@tailrec
def metrop4(n: Int = 1000, eps: Double = 0.5,
  x: Double = 0.0, oldll: Double = Double.MinValue,
  acc: List[Double] = Nil): DenseVector[Double] = {
    if (n == 0)
      DenseVector(acc.reverse.toArray)
    else {
      val can = x + Uniform(-eps, eps).draw
      val loglik = Gaussian(0.0, 1.0).logPdf(can)
      val loga = loglik - oldll
      if (math.log(Uniform(0.0, 1.0).draw) < loga)
        metrop4(n - 1, eps, can, loglik, can :: acc)
      else
        metrop4(n - 1, eps, x, oldll, x :: acc)
    }
}
```

Factoring out the updating logic

This is all fine, but we haven't yet addressed the issue of having different versions of the code depending on what we want to do with the output. The problem is that we have tied up the logic of advancing the Markov chain with what to do with the output. What we need to do is separate out the code for advancing the state. We can do this by defining a new function.

```
def newState(x: Double, oldll: Double, eps: Double):
  (Double, Double) = {
    val can = x + Uniform(-eps, eps).draw
    val loglik = Gaussian(0.0, 1.0).logPdf(can)
    val loga = loglik - oldll
    if (math.log(Uniform(0.0, 1.0).draw) < loga)
      (can, loglik) else (x, oldll)
}
```

This function takes as input a current state and associated log likelihood and returns a new state and log likelihood following the execution of one step of a MH algorithm. This separates the concern of state updating from the rest of the code. So now if we want to write code that prints the state, we can write it as

```
@tailrec
def metrop5(n: Int = 1000, eps: Double = 0.5,
x: Double = 0.0,
oldll: Double = Double.MinValue): Unit = {
  if (n > 0) {
    println(x)
    val ns = newState(x, oldll, eps)
    metrop5(n - 1, eps, ns._1, ns._2)
  }
}
```

Note the use of `._1` and `._2` to access the first and second elements of a tuple. Referring to elements of a tuple by position can sometimes lead to code that is difficult to read. Often it is better to unpack the tuple, as follows.

It may be confusing that Scala generally uses 0–indexing for collections and the like, but the first element of a tuple is `._1`

```
@tailrec
def metrop5b(n: Int = 1000, eps: Double = 0.5,
x: Double = 0.0,
oldll: Double = Double.MinValue): Unit = {
  if (n > 0) {
    println(x)
    val (nx, ll) = newState(x, oldll, eps)
    metrop5b(n - 1, eps, nx, ll)
  }
}
```

Now, if instead of printing to console, we want to accumulate the set of states visited, we can write that as

```
@tailrec
def metrop6(n: Int = 1000, eps: Double = 0.5,
  x: Double = 0.0, oldll: Double = Double.MinValue,
  acc: List[Double] = Nil): DenseVector[Double] = {
  if (n == 0) DenseVector(acc.reverse.toArray) else {
    val (nx, ll) = newState(x, oldll, eps)
    metrop6(n - 1, eps, nx, ll, nx :: acc)
  }
}
```

Both of these functions call `newState` to do the real work, and concentrate on what to do with the sequence of states. However, both of these functions repeat the logic of how to iterate over the sequence of states.

MCMC as a stream

Ideally we would like to abstract out the details of how to do state iteration from the code as well. We can do this using Scala's `Stream`, which we saw earlier can embody the logic of how to perform state iteration, allowing us to abstract those details away from our code.

To do this, we will restructure our code slightly so that it more clearly maps old state to new state.

```
def nextState(eps: Double)(state: (Double, Double)):
  (Double, Double) = {
    val (x, oldll) = state
    val can = x + Uniform(-eps, eps).draw
    val loglik = Gaussian(0.0, 1.0).logPdf(can)
    val loga = loglik - oldll
    if (math.log(Uniform(0.0, 1.0).draw) < loga)
      (can, loglik) else (x, oldll)
  }
```

The "real" state of the chain is just `x`, but if we want to avoid recalculation of the old likelihood, then we need to make this part of the chain's state. We can use this `nextState` function in order to construct a `Stream`.

```
def metrop7(eps: Double = 0.5, x: Double = 0.0,
 oldll: Double = Double.MinValue): Stream[Double] =
  Stream.iterate((x, oldll))(nextState(eps)) map (_._1)
```

The result of calling this is an infinite stream of states. Obviously it isn't computed — that would require infinite computation, but it captures the logic of iteration and computation in a `Stream`, that can be thought of as a lazy `List`. We can get values out by converting the `Stream` to a regular collection, being careful to truncate the `Stream` to one of finite length beforehand! eg. `metrop7().drop(1000) .take(10000).toArray` will do a burn-in of 1,000 iterations followed by a main monitoring run of length 10,000, capturing the results in an `Array`. Note that `metrop7().drop(1000).take(10000)` is a `Stream`, and so nothing is actually computed until the `toArray` is encountered. Conversely, if printing to console is required, just replace the `.toArray` with `.foreach(println)`.

The above stream-based approach to MCMC iteration is clean and elegant, and deals nicely with issues like burn-in and thinning (which can be handled similarly). This is how I typically write MCMC codes these days. However, functional programming purists would still have issues with this approach, as it isn't quite pure functional. The problem is that the code isn't pure — it has a side-effect, which is to mutate the state of the under-pinning pseudo-random number generator. If the code was pure, calling `nextState` with the same inputs would always give the same result. Clearly this isn't the case here, as we have specifically designed the function to be stochastic, returning a randomly sampled value from the desired probability distribution. So `nextState` represents a function for randomly sampling from a conditional probability distribution.

A pure functional approach

Now, ultimately all code has side-effects, or there would be no point in running it! But in functional programming the desire is to make as much of the code as possible pure, and to push side-effects to the very edges of the code. So it's fine to have side-effects in your `main` method, but not buried deep in your code. Here the side-effect is at the very heart of the code, which is why it is potentially an issue.

To keep things as simple as possible, at this point we will stop worrying about carrying forward the old likelihood, and hard-code a value of `eps`. Generalisation is straightforward. We can make our code pure by instead defining a function which represents the conditional probability distribution itself. For this we use a *probability monad*, which in Breeze is called `Rand`. We can couple together such functions using monadic binds (`flatMap` in Scala), expressed most neatly using **for**–comprehensions. So we can write our transition kernel in the form of a simple *probabilistic program* as

```
def kernel(x: Double): Rand[Double] = for {
    innov <- Uniform(-0.5, 0.5)
    can = x + innov
    oldll = Gaussian(0.0, 1.0).logPdf(x)
    loglik = Gaussian(0.0, 1.0).logPdf(can)
    loga = loglik - oldll
    u <- Uniform(0.0, 1.0)
} yield if (math.log(u) < loga) can else x
```

This is now pure — the same input `x` will always return the same probability distribution — the conditional distribution of the next state given the current state, and no RNG state will be mutated. We can draw random samples from this distribution if we must, but it's probably better to work as long as possible with pure functions. So next we need to encapsulate the iteration logic. Breeze has a `MarkovChain` object which can take kernels of this form and return a stochastic `Process` object representing the iteration logic, as follows.

```
MarkovChain(0.0)(kernel).
  steps.
  drop(1000).
  take(10000).
  foreach(println)
```

The `steps` method contains the logic of how to advance the state of the chain. But again note that no computation actually takes place until it is triggered, here by `.drop` — this is when the sampling starts and side-effects happen.

Metropolis-Hastings is a common use-case for Markov chains, so Breeze actually has a helper method built-in that will construct a MH sampler directly from an initial state, a proposal kernel, and a (log) target.

```
MarkovChain.
  metropolisHastings(0.0, (x: Double) =>
  Uniform(x - 0.5, x + 0.5))(x =>
  Gaussian(0.0, 1.0).logPdf(x)).
  steps.
  drop(1000).
  take(10000).
  toArray
```

Note that if you are using the MH functionality in Breeze, it is important to make sure that you are using version 0.13 (or later), as I fixed a few issues with the MH code shortly prior to the 0.13 release.

Summary

Viewing MCMC algorithms as infinite streams of state is useful for writing elegant, generic, flexible code. Streams occur everywhere in programming, and so there are lots of libraries for working with them. Here I used the simple `Stream` from the Scala standard library, but there are much more powerful and flexible stream libraries for Scala, including **fs2** and **akka-streams**. But whatever libraries you are using, the fundamental concepts are the same. The most straightforward approach to implementation is to define impure stochastic streams to consume. However, a pure functional approach is also possible, and the Breeze library defines some useful functions to facilitate this.

5.2.2 A Gibbs sampler

Gibbs sampling is another commonly used MCMC procedure. Gibbs sampling samples from a multivariate distribution by sequentially sampling from *full–conditional* distributions. Like MH, Gibbs sampling is typically used in statistics in order to sample from a Bayesian posterior distribution, but it doesn't have to be. Here we'll illustrate the ideas with a very simple bivariate distribution. The target distribution has density

$$f(x,y) = kx^2\exp\{-xy^2-y^2+2y-4x\},\quad x>0, y\in\mathbb{R},$$

with unknown normalising constant $k>0$, ensuring that the density integrates to one. The two full-conditionals for this distribution can be calculated using elementary algebra, and are given by

$$\begin{aligned} x|y &\sim \Gamma(3, y^2+4),\\ y|x &\sim N\left(\frac{1}{1+x}, \frac{1}{2(1+x)}\right), \end{aligned}$$

using the parametrisation of normal and gamma distributions most commonly adopted by statisticians. One iteration of a Gibbs sampler consists of iterating through the sampling of each full–conditional in turn, using the latest available value of each component.

The second parameter of the normal represents variance, and the expectation of a $\Gamma(\alpha,\beta)$ is α/β. These are both different to Breeze, and this is a common source of bugs.

The state of our Gibbs sampler will be the tuple (x, y). We could implement our sampler using a Scala tuple `(x,y)`, but this is another good opportunity to use Scala *case classes*. We will use a case class to represent the state of our Gibbs sampler.

```
case class State(x: Double, y: Double)
// defined class State
```

Case classes are a convenient way to build data structures, and are used extensively in Scala code.

So our case class has two attributes, `x` and `y`, which are both `Doubles`. We can create, access, copy, and update objects of our case class as follows:

```
val s = State(1.0,2.0)
// s: State = State(1.0,2.0)
s.x
// res0: Double = 1.0
```

```
s.y
// res1: Double = 2.0
s.copy()
// res2: State = State(1.0,2.0)
s.copy(y=3)
// res3: State = State(1.0,3.0)
```

Note that the final example shows how to create a copy of an existing state with just one attribute updated. This can be useful for constructing Gibbs samplers, though we won't actually use it in this very simple example. Since the Gibbs sampler is an MCMC algorithm, the iteration pattern is exactly the same as for the MH algorithm. We will begin by implementing an impure stochastic stream approach. First we need a function to execute one iteration of the Gibbs sampler.

```
import breeze.stats.distributions._
// import breeze.stats.distributions._

def nextState(state: State): State = {
  val sy = state.y
  val x = Gamma(3.0,1.0/(sy*sy+4)).draw
  val y = Gaussian(1.0/(x+1),1.0/math.sqrt(2*x+2)).draw
  State(x,y)
}
```

Note how we have translated the parameters of the full-conditionals to the form that Breeze expects. Once we have this state updating function, creating a `Stream` for our Markov chain is trivial.

```
val gs = Stream.iterate(State(1.0,1.0))(nextState)
// gs: scala.collection.immutable.Stream[State] =
//   Stream(State(1.0,1.0), ?)
val output = gs.drop(1000).take(100000).toArray
// output: Array[State] = Array(
//   State(0.20703194113971382,0.874650780098001),
//   State(0.5813103371812548,0.4780234809903935), ...
```

If we want to look at the output, we can do it with code such as the following.

```
import breeze.linalg._
val xv = DenseVector(output map (_.x))
val yv = DenseVector(output map (_.y))

import breeze.plot._
val fig = Figure("Bivariate Gibbs sampler")
fig.subplot(2,2,0)+=hist(xv,50)
fig.subplot(2,2,1)+=hist(yv,50)
fig.subplot(2,2,2)+=plot(xv,yv,'.')
```

Thinning MCMC streams

For Markov chains with strong dependence, as well as truncating a burn-in period (using `drop`), we may also want to *thin* the output, keeping just one in every n iterations, for some suitably chosen $n > 1$. Typical values might be $n = 10$ or $n = 100$. This is done primarily to reduce storage requirements associated with very long MCMC runs. Unfortunately there is no `thin` method pre-defined on the Scala `Stream`, but we can define our own function for thinning `Stream`s as

```
def thin[T](s: Stream[T], th: Int): Stream[T] = {
    val ss = s.drop(th - 1)
    if (ss.isEmpty) Stream.empty else
      ss.head #:: thin(ss.tail, th)
  }
```

which we could use as

```
thin(gs.drop(1000),10).take(10000).toArray
```

This will give us a final sample of size 10,000 following a burn-in of 1,000 iterations and a thinning factor of 10. Ideally we might like to define our own method `thin` on the existing `Stream` class. In fact this is perfectly possible to do in Scala, and would allow us to write code like

```
//  gs.drop(1000).thin(10).take(10000)
```

which would be more consistent. However, this requires advanced features of the Scala language (*type classes* and *implicits*) that we are not yet ready to discuss, but we will see how to implement exactly this in the final chapter.

A monadic approach

A pure functional monadic approach to defining a Gibbs sampler is not much more effort, as the code below illustrates.

```
def kernel(state: State): Rand[State] = for {
  x <- Gamma(3.0,1.0/(state.y*state.y+4))
  y <- Gaussian(1.0/(x+1),1.0/math.sqrt(2*x+2))
  ns = State(x,y)
} yield ns

val out3 = MarkovChain(State(1.0,1.0))(kernel).
  steps.
  drop(1000).
  take(10000).
  toArray
```

It is really a matter of taste as to which of these approaches is to be preferred. Also note that the two approaches are not mutually exclusive. For example, starting from the pure monadic approach, one can take a Breeze `Process` object defined, say, by a `MarkovChain` construction, and act on it with `.steps.toStream` to turn the pure `Process` into an impure stochastic `Stream`.

Chapter 6

Statistical modelling

6.1 Linear regression

6.1.1 Introduction

Statistical modelling is a huge topic and we certainly don't have time to discuss it in detail in the context of this course. But *linear modelling*, using *multiple linear regression* is almost certainly the most widely used statistical modelling technique in practice. So it makes sense to start our investigation of statistical modelling with linear regression. We assume n observations of a univariate response $y_i,\ i = 1, 2, \ldots, n$ with associated covariates $\mathbf{x}_i$, where each $\mathbf{x}_i$ is a p-vector. We model y_i as a linear combination of the $\mathbf{x}_i$. That is, we want to find a p-vector $\boldsymbol{\beta}$ so that $\mathbf{x}_i \cdot \boldsymbol{\beta}$ is close to $y_i,\ \forall i$. To avoid explicit consideration of an intercept, we assume that the first element of each covariate vector is 1. Stacking the observations and covariates, we have an n-vector $\mathbf{y}$ and an associated $n \times p$ matrix X of covariates (where the ith row of X corresponds to $\mathbf{x}_i^{\mathsf{T}}$) with first column consisting of 1s, and our model is

$$\mathbf{y} \sim \mathsf{X}\boldsymbol{\beta}.$$

In other words, we want to find the $\widehat{\boldsymbol{\beta}}$ which minimises $\|\mathbf{y} - \mathsf{X}\widehat{\boldsymbol{\beta}}\|_2^2$. The solution of this *least squares problem* is given by the solution of the *normal equations*

$$\mathsf{X}^{\mathsf{T}}\mathsf{X}\widehat{\boldsymbol{\beta}} = \mathsf{X}^{\mathsf{T}}\mathbf{y}.$$

Mathematically, we can write this solution as $\widehat{\boldsymbol{\beta}} = (\mathsf{X}^{\mathsf{T}}\mathsf{X})^{-1}\mathsf{X}^{\mathsf{T}}\mathbf{y}$, but in numerical linear algebra it is usually better to solve a linear system directly rather than compute a matrix inverse explicitly.

There are many ways to solve linear systems. The normal equations are typically solved using a thin QR-factorisation of the covariate matrix. So if $\mathsf{X} = \mathsf{Q}\mathsf{R}$, where Q is $n \times p$ st $\mathsf{Q}^{\mathsf{T}}\mathsf{Q} = \mathsf{I}$ and R is $p \times p$ upper triangular, then the normal equations simplify to

$$\mathsf{R}\widehat{\boldsymbol{\beta}} = \mathsf{Q}^{\mathsf{T}}\mathbf{y},$$

and this can be solved for $\widehat{\boldsymbol{\beta}}$ with a single back-solve.

6.1.2 Linear regression in Scala

Let's start by creating a small toy dataset.

```
import breeze.linalg._
// import breeze.linalg._

val y = DenseVector(1.0,2.0,3.0,2.0,1.0)
// y: DenseVector[Double] = DenseVector(1.0, 2.0,
//                         3.0, 2.0, 1.0)
val Xwoi = DenseMatrix((2.1,1.4),(1.5,2.2),
           (3.0,3.1),(2.5,2.2),(2.0,1.0))
// Xwoi: breeze.linalg.DenseMatrix[Double] =
// 2.1  1.4
// 1.5  2.2
// 3.0  3.1
// 2.5  2.2
// 2.0  1.0
val X = DenseMatrix.horzcat(DenseVector.ones[Double](
              Xwoi.rows).toDenseMatrix.t, Xwoi)
// X: breeze.linalg.DenseMatrix[Double] =
// 1.0  2.1  1.4
// 1.0  1.5  2.2
// 1.0  3.0  3.1
// 1.0  2.5  2.2
// 1.0  2.0  1.0
```

Note how the final line demonstrates how to prepend a column of ones onto a covariate matrix.

There is a simple least squares function included in Breeze which can be used to solve this problem.

```
import breeze.stats.regression._
val mod = leastSquares(X,y)
// mod: LeastSquaresRegressionResult = <function1>
mod.coefficients
// res3: DenseVector[Double] = DenseVector(
//   -0.2804082840767416, 0.05363661640849975,
//    0.9905732301261981)
mod.rSquared
// res4: Double = 0.08519073288426877
```

I'm not sure what `rSquared` is supposed to be. It certainly doesn't correspond to the usual notion of R^2 in linear regression modelling

This solves the problem, but doesn't really offer much over a more direct solution:

```
X \ y
// res5: DenseVector[Double] = DenseVector(
//   -0.2804082840767416, 0.05363661640849975,
//    0.9905732301261981)
```

because as previously discussed, the Breeze linear solver uses a QR–based solver by default for over-determined systems.

It is useful to understand exactly how a QR–based solution works, as we can then build on it. First we need to define a `backSolve` function, since one is not exported by the Breeze API.

```
import com.github.fommil.netlib.BLAS.{ getInstance =>
    blas }
def backSolve(A: DenseMatrix[Double],
  y: DenseVector[Double]): DenseVector[Double] = {
    val yc = y.copy
    blas.dtrsv("U", "N", "N", A.cols, A.toArray,
        A.rows, yc.data, 1)
    yc
}
```

We can now use this with a QR–factorisation of the model matrix to solve the least squares problem.

```
val QR = qr.reduced(X)
// QR: breeze.linalg.qr.DenseQR =
// QR(-0.44721359549995787   -0.10656672499880869   0.36158387049672375
//  -0.44721359549995787   -0.6394003499928489    -0.620639682977601
//  -0.44721359549995787   0.6926837124922532     -0.3519495634209888
//  -0.44721359549995787   0.24865569166388585    0.010942669765032 4
//  -0.44721359549995787   -0.1953723291644815    0.6000627061368334
,
//  -2.23606797749979   -4.964070910049532   -4.427414595449584
//  0.0                 1.12605506082074     0.9431155162394527
//  0.0                 0.0                  -1.3260969508404699
)
val q = QR.q
// q: breeze.linalg.DenseMatrix[Double] =
//  -0.44721359549995787   -0.10656672499880869   0.36158387049672375
//  -0.44721359549995787   -0.6394003499928489    -0.620639682977601
//  -0.44721359549995787   0.6926837124922532     -0.3519495634209888
//  -0.44721359549995787   0.24865569166388585    0.0109426697650324
//  -0.44721359549995787   -0.1953723291644815    0.6000627061368334
val r = QR.r
// r: breeze.linalg.DenseMatrix[Double] =
//  -2.23606797749979   -4.964070910049532   -4.427414595449584
//  0.0                 1.12605506082074     0.9431155162394527
//  0.0                 0.0                  -1.3260969508404699
backSolve(r, q.t * y)
// res6: DenseVector[Double] = DenseVector(
//    -0.2804082840767417, 0.053636616408499954,
//     0.9905732301261979)
```

If we are only interested in regression coefficients, then we are done. But in practice regression modelling involves diagnostics, such as indicators of the relative importance of different covariates, etc. For people used to regression modelling in **R**, such diagnostics are essential. It's easy enough to write a function to provide such diagnostics, as required. We will try and make the code reasonably efficient by using a QR–based approach.

```
case class Lm(y: DenseVector[Double],
  X: DenseMatrix[Double],names: List[String]) {
  require(y.size == X.rows)
  require(names.length == X.cols)
  require(X.rows >= X.cols)
  val QR = qr.reduced(X)
  val q = QR.q
  val r = QR.r
  val qty = q.t * y
  val coefficients = backSolve(r,qty)
  import breeze.stats._
  import org.apache.commons.math3.special.Beta
  def tCDF(t: Double, df: Double): Double = {
    val xt = df / (t * t + df)
    1.0 - 0.5 * Beta.regularizedBeta(xt, 0.5 * df, 0.5)
  }
  def fCDF(x: Double, d1: Double, d2: Double) = {
    val xt = x * d1 / (x * d1 + d2)
    Beta.regularizedBeta(xt, 0.5 * d1, 0.5 * d2)
  }
  lazy val fitted = q * qty
  lazy val residuals = y - fitted
  lazy val n = X.rows
  lazy val pp = X.cols
  lazy val df = n - pp
  lazy val rss = sum(residuals ^:^ 2.0)
  lazy val rse = math.sqrt(rss / df)
  lazy val ri = inv(r)
  lazy val xtxi = ri * (ri.t)
  lazy val se = breeze.numerics.sqrt(diag(xtxi))*rse
  lazy val t = coefficients / se
  lazy val p = t.map {1.0 - tCDF(_, df)}.map {_ * 2}
  lazy val ybar = mean(y)
  lazy val ymyb = y - ybar
  lazy val ssy = sum(ymyb ^:^ 2.0)
  lazy val rSquared = (ssy - rss) / ssy
  lazy val adjRs = 1.0 - ((n-1.0)/(n-pp))*(1-rSquared)
  lazy val k = pp-1
  lazy val f = (ssy - rss) / k / (rss/df)
  lazy val pf = 1.0 - fCDF(f,k,df)
  def summary: Unit = {
    println(
    "Estimate\t␣S.E.\t␣t-stat\tp-value\t\tVariable")
    println(
    "----------------------------------------------------------------")
    (0 until pp).foreach(i => printf(
     "%8.4f\t%6.3f\t%6.3f\t%6.4f␣%s\t%s\n",
      coefficients(i),se(i),t(i),p(i),
      if (p(i) < 0.05) "*" else "␣",
      names(i)))
    printf(
     "\nResidual␣standard␣error:␣%8.4f␣on␣%d␣degrees␣of␣freedom\
    rse,df)
    printf(
     "Multiple␣R-squared:␣%6.4f,␣Adjusted␣R-squared:␣%6.4f\n",
     rSquared,adjRs)
    printf(
     "F-statistic:␣%6.4f␣on␣%d␣and␣%d␣DF,␣p-value:␣%6.5f\n\n",
     f,k,df,pf)
  }
}
```

There are few things worth noting about this code. First, it is a *case class*, so when we "call" it we will actually get back an *object*. Next, note the use of `require` to make explicit our expectations (*pre–conditions*). Also note the use of **val** versus `lazy` **val**. Everything up to and including the regression coefficients will be computed eagerly when the object is first constructed, but all of the "optional extras" will be computed lazily if and when they are required. Finally note the `summary` method which will print to the console a textual summary of the regression fit that will look very familiar to **R** users. Note that all of the numerical output matches up exactly with the corresponding output from **R**.

We can create a model fit using the default case class constructor:

```
val mod2 = Lm(y, X, List("Intercept","Var1","Var2"))
// mod2: Lm =
// Lm(DenseVector(1.0, 2.0, 3.0, 2.0, 1.0),
// 1.0  2.1  1.4
// 1.0  1.5  2.2
// 1.0  3.0  3.1
// 1.0  2.5  2.2
// 1.0  2.0  1.0  ,List(Intercept, Var1, Var2))
```

All of the attributes of the model fit can be accessed in the obvious way.

```
mod2.coefficients
// res7: DenseVector[Double] = DenseVector(
//  -0.2804082840767417, 0.05363661640849954,
//   0.9905732301261979)
mod2.p
// res8: DenseVector[Double] = DenseVector(
//  0.5711265415017199, 0.8337153027851611,
//  0.02380731603365338)
mod2.rse
// res9: Double = 0.20638644926965116
mod2.rSquared
// res10: Double = 0.9695747382556184
mod2.adjRs
// res11: Double = 0.9391494765112367
mod2.f
// res12: Double = 31.86742472099393
mod2.pf
// res13: Double = 0.03042526174438165
```

An **R**–style model fit summary can also be printed to the console.

```
mod2.summary
// Estimate      S.E.    t-stat  p-value         Variable
// ---------------------------------------------------------
//  -0.2804      0.418   -0.671   0.5711         Intercept
//   0.0536      0.225    0.238   0.8337         Var1
//   0.9906      0.156    6.365   0.0238 *       Var2
// Residual standard error:   0.2064 on 2 degrees of freedom
// Multiple R-squared: 0.9696, Adjusted R-squared: 0.9391
// F-statistic: 31.8674 on 2 and 2 DF, p-value: 0.03043
```

Now we understand how it works, we can try it out on a bigger, synthetic data example. First, let's simulate some data from a linear regression model.

```
// Synthetic data...
val N = 1000
// N: Int = 1000
val P = 2
```

```
// P: Int = 2
val gau=breeze.stats.distributions.Gaussian(0.0,1.0)
// gau: Gaussian = Gaussian(0.0, 1.0)
val XX = new DenseMatrix(N,P,gau.sample(P*N).toArray)
// XX: breeze.linalg.DenseMatrix[Double] =
// 0.11002285761491036    -0.20149313279570508
// 0.6185281490520133     -0.037185723563396854
// 1.7071005345717225     1.2413540328321024
// -0.2220569665834186    0.4686688219889745
// ...
val X = DenseMatrix.horzcat(
  DenseVector.ones[Double](N).toDenseMatrix.t,XX)
// X: breeze.linalg.DenseMatrix[Double] =
// 1.0  0.11002285761491036    -0.20149313279570508
// 1.0  0.6185281490520133     -0.037185723563396854
// 1.0  1.7071005345717225     1.2413540328321024
// 1.0  -0.2220569665834186    0.4686688219889745
// 1.0  ...
val b0 = linspace(1.0,2.0,P+1)
// b0: DenseVector[Double] = DenseVector(1.0,1.5,2.0)
val y0 = X * b0
// y0: DenseVector[Double] = DenseVector(...
val y = y0 + DenseVector(gau.sample(1000).toArray)
// y: DenseVector[Double] = DenseVector(...
```

Now we have our synthetic data, we can do a “quick–and–dirty” regression fit with

```
val b = X \ y
// b: DenseVector[Double] = DenseVector(
//   0.985909926499153, 1.4939286812873727,
//   1.9694751417388345)
```

Alternatively, we can use our regression class to get more information about the fit.

```
val mod3 = Lm(y,X, List("Intercept","Var1","Var2"))
// mod3: Lm =
// Lm(DenseVector(0.2863458414136328, ...
mod3.summary
// Estimate     S.E.    t-stat  p-value        Variable
// ---------------------------------------------------------
//   0.9859     0.031   31.845  0.0000 *       Intercept
//   1.4939     0.032   46.488  0.0000 *       Var1
//   1.9695     0.033   60.401  0.0000 *       Var2
// Residual standard error:  0.9778 on 997 degrees of freedom
// Multiple R-squared: 0.8535, Adjusted R-squared: 0.8532
// F-statistic: 2903.0978 on 2 and 997 DF, p-value: 0.00000
```

Finally, we can plot some diagnostic information.

```
import breeze.plot._
// import breeze.plot._
val fig = Figure("Regression diagnostics")
// fig: Figure = Figure@37569f24
fig.width = 1000
// fig.width: Int = 1000
fig.height = 800
// fig.height: Int = 800
val p = fig.subplot(1,1,0)
```

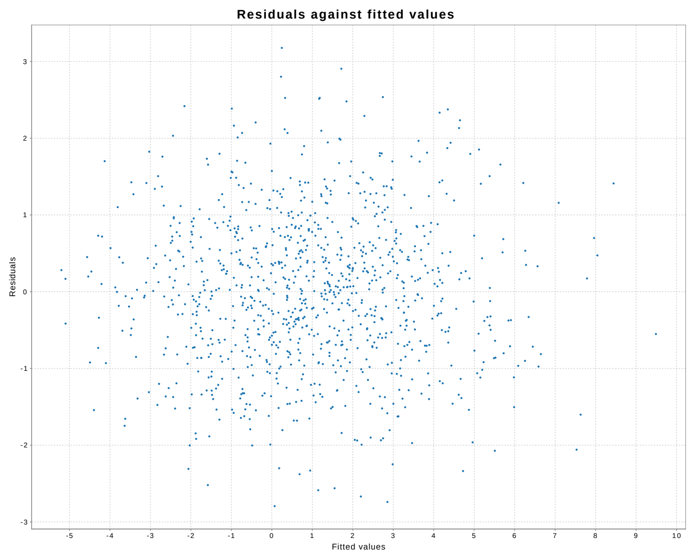

Figure 6.1: Linear regression diagnostics

```
// p: Plot = Plot@298c0508
p += plot(mod3.fitted,mod3.residuals,'.')
// res16: Plot = Plot@298c0508
p.xlabel = "Fitted␣values"
// p.xlabel: String = Fitted values
p.ylabel = "Residuals"
// p.ylabel: String = Residuals
p.title = "Residuals␣against␣fitted␣values"
// p.title: String = Residuals against fitted values
```

This gives the plot shown in Figure 6.1.

6.1.3 Case study: linear regression for a real dataset

We will conclude this section on linear regression with the analysis of a real dataset. We will use the "yacht hydrodynamics dataset" from the ML repository:

archive.ics.uci.edu/ml/

http://archive.ics.uci.edu/ml/datasets/Yacht+Hydrodynamics

We will begin by downloading the dataset to local disk (if it hasn't been already), converting the white-space separated file to CSV in the process.

```
val url = "http://archive.ics.uci.edu/ml/machine-learning-databases/00243/yacht_hydrodyn
val fileName = "yacht.csv"
val file = new java.io.File(fileName)
if (!file.exists) {
      val s = new java.io.PrintWriter(file)
      val data = scala.io.Source.fromURL(url).getLines
      data.foreach(l => s.write(l.trim.split('␣').
        filter(_ != "").mkString("",",","\n")))
      s.close
}
```

Since the Breeze CSV parser is rather rudimentary, it's best to make the parsing job as simple as possible. Now that the file is on disk, we can parse it and extract the key information as follows.

```
val mat = csvread(new java.io.File(fileName))
// mat: breeze.linalg.DenseMatrix[Double] =
// -2.3  0.568  4.78  3.99  3.17  0.125  0.11
// -2.3  0.568  4.78  3.99  3.17  0.15   0.27
// -2.3  0.568  4.78  3.99  3.17  0.175  0.47
// -2.3  0.568  4.78  3.99  3.17  0.2    0.78
// -2.3  0.568  4.78  3.99  3.17  0.225  1.18
// ....
println("Dim: " + mat.rows + " " + mat.cols)
// Dim: 308 7
val y = mat(::, 6) // response is the final column
// y: breeze.linalg.DenseVector[Double] = DenseVector(0.11, 0.27
val x = mat(::, 0 to 5)
// x: breeze.linalg.DenseMatrix[Double] =
// -2.3  0.568  4.78  3.99  3.17  0.125
// -2.3  0.568  4.78  3.99  3.17  0.15
// -2.3  0.568  4.78  3.99  3.17  0.175
// -2.3  0.568  4.78  3.99  3.17  0.2
// -2.3  0.568  4.78  3.99  3.17  0.225
// -2.3  0.568  4.78  3.99  3.17  0.25
// -2.3  0.568  4.78  3.99  3.17  0.275
// ...
```

Now we have our response and covariate matrix, we can fit the linear model (without an intercept).

```
Lm(y,x,List("LongPos","PrisCoef","LDR","BDR","LBR",
  "Froude")).summary
// Estimate      S.E.     t-stat  p-value         Variable
// ---------------------------------------------------------
//    0.1943      0.338    0.575   0.5655          LongPos
// -35.6159     16.005   -2.225   0.0268 *        PrisCoef
//   -4.1631      7.779   -0.535   0.5929          LDR
//    1.3747      3.297    0.417   0.6770          BDR
//    3.3232      8.911    0.373   0.7095          LBR
// 121.4745      5.054   24.034   0.0000 *        Froude
// Residual standard error:  8.9522 on 302 degrees of freedom
// Multiple R-squared: 0.6570, Adjusted R-squared: 0.6513
// F-statistic: 115.6888 on 5 and 302 DF, p-value: 0.00000
```

This suggests that two covariates are significant. However, fitting without an intercept doesn't really make sense here, so we can refit including an intercept.

```
val X = DenseMatrix.horzcat(
   DenseVector.ones[Double](x.rows).toDenseMatrix.t,x)
// X: breeze.linalg.DenseMatrix[Double] =
// 1.0  -2.3  0.568  4.78  3.99  3.17  0.125
// 1.0  -2.3  0.568  4.78  3.99  3.17  0.15
// 1.0  -2.3  0.568  4.78  3.99  3.17  0.175
// 1.0  -2.3  0.568  4.78  3.99  3.17  0.2
// 1.0  -2.3  0.568  4.78  3.99  3.17  0.225
// ...
val mod = Lm(y,X,List("(Intercept)","LongPos",
  "PrisCoef","LDR","BDR","LBR","Froude"))
// mod: Lm =
// Lm(DenseVector(0.11, 0.27, 0.47, ...
mod.summary
// Estimate      S.E.     t-stat  p-value         Variable
// ---------------------------------------------------------------
//  -19.2367       27.113   -0.709   0.4786           (Intercept)
//    0.1938        0.338    0.573   0.5668           LongPos
//   -6.4194       44.159   -0.145   0.8845           PrisCoef
//    4.2330       14.165    0.299   0.7653           LDR
//   -1.7657        5.521   -0.320   0.7493           BDR
//   -4.5164       14.200   -0.318   0.7507           LBR
//  121.6676        5.066   24.018   0.0000 *         Froude
// Residual standard error:   8.9596 on 301 degrees of freedom
// Multiple R-squared: 0.6576, Adjusted R-squared: 0.6507
// F-statistic: 96.3327 on 6 and 301 DF, p-value: 0.00000
```

This time only the `Froude` variable is significant.

6.2 Generalised linear models

6.2.1 Introduction

At its most basic level (computing regression coefficients), linear regression can be viewed purely as a least squares problem, independently of any modelling assumptions. But in practice, diagnostic statistics rely on certain assumptions, including iid Gaussian errors. This means that linear regression doesn't work very well when errors aren't approximately Gaussian, and this can be a particular problem for a binary response or count data. *Generalised linear models* (GLMs) extend linear modelling theory to cover response variables in the *exponential family*. The theory of GLMs is out of scope for this course, but we just need some basics. Typically GLM theory is presented for a two–parameter (or, in fact, a scaled one–parameter) exponential family. But for the most commonly used cases: Bernoulli (logistic), Binomial, and Poisson regression, a simple one–parameter exponential family is sufficient. So we will assume that our response variable has a probability mass (or density) function of the form

$$f(y|\theta) = \exp\{\theta y - b(\theta) + c(y)\},$$

where θ is the *canonical* parameter of the distribution. Then setting up a regression model with a linear predictor for the canonical parameter (a *canonical link*), we can construct a Newton–Raphson scheme for finding the maximum likelihood estimator, which takes the form of an *iteratively reweighted least squares* (IRLS) algorithm, with updates of the form

$$\boldsymbol{\beta}_{n+1} = \boldsymbol{\beta}_n + [\mathsf{X}^\mathsf{T}\mathsf{W}_\mathsf{n}\mathsf{X}]^{-1}\mathsf{X}^\mathsf{T}\mathbf{z}_n,$$

where

$$\mathsf{W}_n = \operatorname{diag}\{b''(\mathsf{X}\boldsymbol{\beta}_n)\} \text{ and } \mathbf{z}_n = \mathbf{y} - b'(\mathsf{X}\boldsymbol{\beta}_n).$$

There are efficient QR–based solutions to this problem, but here we will present a simple illustrative implementation.

6.2.2 Logistic regression

Arguably the simplest, and most widely used non-Gaussian GLM is *logistic regression*, with a binary response variable. Here we assume that

$$y_i \sim \text{Bern}(p_i)$$

for some predictor p_i, and so the response distribution is

$$\begin{aligned} f(y|p) &= p^y(1-p)^{1-y} \\ &= \left(\frac{p}{1-p}\right)^y (1-p) \\ &= \exp\left\{y\log\left(\frac{p}{1-p}\right) + \log(1-p)\right\} \end{aligned}$$

which is one–parameter exponential with canonical parameter

$$\theta = \log\{p/(1-p)\} \equiv \text{logit}(p)$$

and

$$b(\theta) = \log(1+e^\theta).$$

Let's now see how this works in practice. We begin by simulating some synthetic data from a logistic regression model.

```
import breeze.linalg._
// import breeze.linalg._
import breeze.stats.distributions.{Gaussian,Binomial}
// import distributions.{Gaussian, Binomial}
val N = 2000
// N: Int = 2000
val beta = DenseVector(0.1,0.3)
// beta: DenseVector[Double] = DenseVector(0.1, 0.3)
val ones = DenseVector.ones[Double](N)
// ones: DenseVector[Double] = DenseVector(1.0, ...
val x = DenseVector(Gaussian(1.0,3.0).sample(N).toArray)
// x: DenseVector[Double] = DenseVector(6.272108997968 15,
//  3.0135444386214765, 4.373649007468049, ...
val X = DenseMatrix.vertcat(ones.toDenseMatrix,
  x.toDenseMatrix).t
// X: breeze.linalg.DenseMatrix[Double] =
// 1.0   6.27210899796815
// 1.0   3.0135444386214765
// 1.0   4.373649007468049
// 1.0   -0.34689004119207434
// ...
val theta = X * beta
// theta: DenseVector[Double] = DenseVector(
// 1.981632699390445,
// 1.0040633315864 43, 1.4120947022404147, ...
def expit(x: Double): Double = 1.0/(1.0+math.exp(-x))
// expit: (x: Double)Double
val p = theta map expit
// p: breeze.linalg.DenseVector[Double] = DenseVector(
//  0.8788551012256975, 0.7318567276541773, ...
val y = p map (pi => new Binomial(1,pi).draw) map (
   _.toDouble)
// y: DenseVector[Double] = DenseVector(1.0, 1.0, 1.0, ...
```

Note that the linear predictor is constructed for the canonical parameter `theta`, and that this is pushed through the `expit` function (often called the *sigmoid*, or *logistic* function, the inverse of the *logit*) to get a probability. Now we have our synthetic data, we need an implementation of the IRLS algorithm for finding the optimal regression coefficients. A very simple implementation is given below.

```
@annotation.tailrec
def IRLS(
  b: Double => Double,
  bp: Double => Double,
  bpp: Double => Double,
  y: DenseVector[Double],
  X: DenseMatrix[Double],
  bhat0: DenseVector[Double],
  its: Int,
  tol: Double = 0.0000001
): DenseVector[Double] = if (its == 0) {
  println("WARNING: IRLS did not converge")
  bhat0
} else {
  val eta = X * bhat0
  val W = diag(eta map bpp)
  val z = y - (eta map bp)
  val bhat = bhat0 + (X.t * W * X) \ (X.t * z)
  if (norm(bhat-bhat0) < tol) bhat else
    IRLS(b,bp,bpp,y,X,bhat,its-1,tol)
}
```

Note that this function is illustrative. It is not efficient and should not be used in practice. Improving its efficiency is part of the end–of–chapter exercises.

Though very simple, this function works for any one–parameter exponential family GLM, accepting as arguments the function `b` $= b(\cdot)$ (which isn't actually required), its derivative `bp` and second derivative `bpp`. We can write a simple wrapper for this for the case of logistic regression as follows.

```
def logReg(
  y: DenseVector[Double],
  X: DenseMatrix[Double],
  its: Int = 30
): DenseVector[Double] = {
  def b(x: Double): Double = math.log(1.0+math.exp(x))
  def bp(x: Double): Double = expit(x)
  def bpp(x: Double): Double = {
    val e = math.exp(-x)
     e/((1.0+e)*(1.0+e))
  }
  val bhat0 = DenseVector.zeros[Double](X.cols)
  IRLS(b,bp,bpp,y,X,bhat0,its)
}
```

We can now call this to fit a logistic regression model as follows.

```
val betahat = logReg(y,X)
// betahat: DenseVector[Double] = DenseVector(
//   0.0563791369276622, 0.3160688304687223б)
```

For diagnostic purposes, we could plot our fitted probabilities along with the observed data.

Note that the estimated regression coefficients are not fantastic, despite having 2,000 training data points. This is not due to convergence problems — it is simply that binary outcomes have limited information.

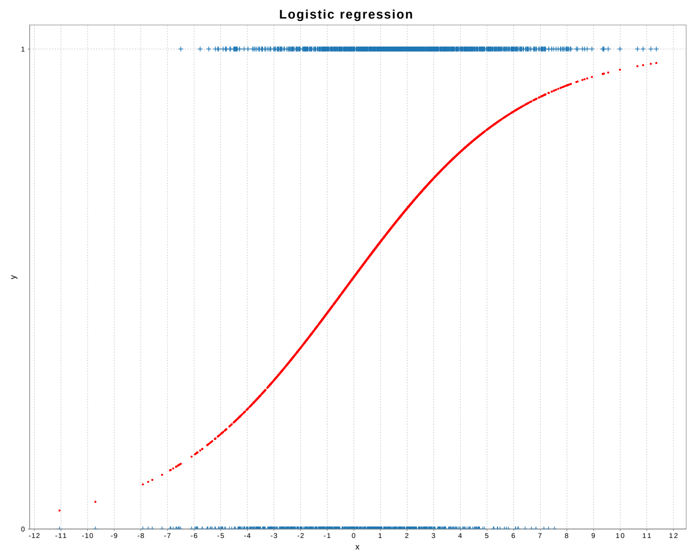

Figure 6.2: Logistic regression

```
import breeze.plot._
// import breeze.plot._
val fig = Figure("Logistic regression")
// fig: Figure = breeze.plot.Figure@11befd2e
fig.width = 1000
// fig.width: Int = 1000
fig.height = 800
// fig.height: Int = 800
val p = fig.subplot(1,1,0)
// p: breeze.plot.Plot = breeze.plot.Plot@62b33d65
p += plot(x,y,'+')
// res0: breeze.plot.Plot = breeze.plot.Plot@62b33d65
p += plot(x,x map (xi =>
  expit(betahat(0)+betahat(1)*xi)),
  '.',colorcode="red")
// res1: breeze.plot.Plot = breeze.plot.Plot@62b33d65
p.xlabel = "x"
// p.xlabel: String = x
p.ylabel = "y"
// p.ylabel: String = y
p.title = "Logistic regression"
// p.title: String = Logistic regression
```

This gives the plot shown in Figure 6.2.

6.2.3 Poisson regression

Poisson regression is often used when the response variable is a non-negative integer “count” of some sort. This is also exponential family, as we model the response variable as

$$y_i \sim \text{Pois}(\mu_i)$$

for some predictor μ_i, and so the response distribution is

$$f(y|\mu) = \frac{\mu^x e^{-\mu}}{x!} = \exp\{x \log \mu - \mu - \log x!\}$$

which is exponential family with canonical parameter $\theta = \log \mu$ and $b(\theta) = e^{\theta}$. So here we have a log link, which means that the linear predictor is exponentiated to get the mean of the Poisson distribution.

This is good, because the mean of a Poisson distribution can't be negative.

Again we will begin by simulating some synthetic data consistent with the assumptions of a Poisson regression model.

```
import breeze.linalg._
// import breeze.linalg._
import breeze.numerics._
// import breeze.numerics._
import breeze.stats.distributions.{Gaussian,Poisson}
// import distributions.{Gaussian, Poisson}
val N = 2000
// N: Int = 2000
val beta = DenseVector(-3.0,0.1)
// beta: DenseVector[Double] = DenseVector(-3.0, 0.1)
val ones = DenseVector.ones[Double](N)
// ones: DenseVector[Double] = DenseVector(1.0, 1.0, ...
val x = DenseVector(Gaussian(50.0,10.0).sample(N).
   toArray)
// x: DenseVector[Double] = DenseVector(
//   54.811589088666324,
//   35.54528051285478, 59.11931256262003, ...
val X = DenseMatrix.vertcat(ones.toDenseMatrix,
          x.toDenseMatrix).t
// X: breeze.linalg.DenseMatrix[Double] =
// 1.0   54.811589088666324
// 1.0   35.54528051285478
// 1.0   59.11931256262003
// 1.0   54.17607633281474
// ...
val theta = X * beta
// theta: DenseVector[Double] = DenseVector(
//   2.4811589088666324,
//   0.5545280512854784, 2.9119312562620037, ...
val mu = exp(theta)
// mu: DenseVector[Double] = DenseVector(
//   11.955111277135977,
//    1.7411190719311496, 18.392284504390723, ...
val y = mu map (mui => new Poisson(mui).draw) map
  (_.toDouble)
// y: DenseVector[Double] = DenseVector(6.0, 2.0, ...
```

Now we have a synthetic dataset we can define a function for fitting a Poisson regression model, reusing our `IRLS` function from previously.

```
def poiReg(
  y: DenseVector[Double],
  X: DenseMatrix[Double],
  its: Int = 30
): DenseVector[Double] = {
  val bhat0 = DenseVector.zeros[Double](X.cols)
  IRLS(math.exp,math.exp,math.exp,y,X,bhat0,its)
}
```

We can now use this function to fit a model.

```
poiReg(y,X)
// WARNING: IRLS did not converge
// res0: DenseVector[Double] = DenseVector(
//   -77.71670795773245, 1.1994502808957552)
```

This runs, but prints a warning to the console indicating that the maximum number of iterations was reached. The estimated parameters are terrible. However, we can easily re-fit the model, bumping up the maximum iteration count.

```
val betahat = poiReg(y,X,its=100)
// betahat: DenseVector[Double] = DenseVector(
//   -3.0118026291813274, 0.10020295518543317)
```

This time there is no warning and the estimated parameters look good.

Finally, we can plot the data and the fitted means.

```
import breeze.plot._
// import breeze.plot._
val fig = Figure("Poisson regression")
// fig: breeze.plot.Figure = breeze.plot.Figure@18a68150
fig.width = 1000
// fig.width: Int = 1000
fig.height = 800
// fig.height: Int = 800
val p = fig.subplot(1,1,0)
// p: breeze.plot.Plot = breeze.plot.Plot@7bdf24a7
p += plot(x,y,'+')
// res1: breeze.plot.Plot = breeze.plot.Plot@7bdf24a7
p += plot(x,x map (xi => math.exp(
  betahat(0)+betahat(1)*xi)),
  '.',colorcode="red")
// res2: breeze.plot.Plot = breeze.plot.Plot@7bdf24a7
p.xlabel = "x"
// p.xlabel: String = x
p.ylabel = "y"
// p.ylabel: String = y
p.title = "Poisson regression"
// p.title: String = Poisson regression
```

This gives the plot shown in Figure 6.3.

6.3 The `scala-glm` library

github.com/darrenjw/scala-glm

While writing these notes it became apparent that it would be quite useful to have the code from this Chapter for fitting linear and generalised linear models tidied, improved, documented and packaged as a Scala library, for convenience. I've now done this, and the library is available on-line, called **`scala-glm`**. We will explore this (small) library as part of the end–of–chapter exercises.

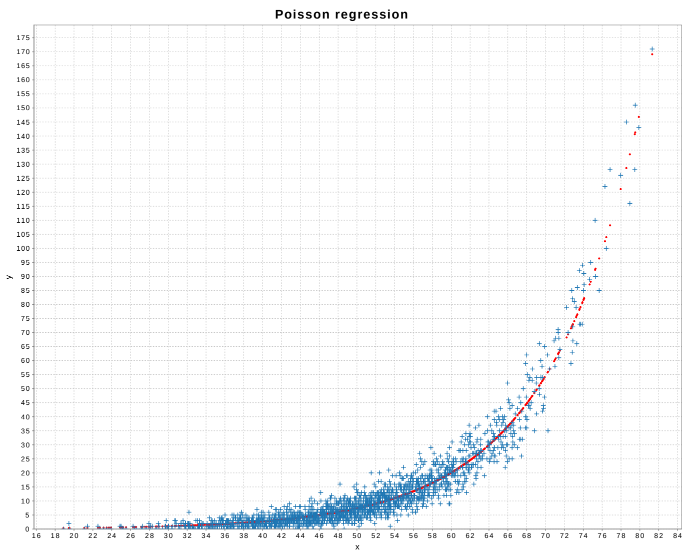

Figure 6.3: Poisson regression

6.4 Data frames and tables in Scala

Introduction

To statisticians and data scientists used to working in **R**, the concept of a data frame is one of the most natural and basic starting points for statistical computing and data analysis. It always surprises me that data frames aren't a core concept in most programming languages' standard libraries, since they are essentially a representation of a relational database table, and relational databases are pretty ubiquitous in data processing and related computing. For statistical modelling and data science, having functions designed for data frames is more convenient than using functions designed to work directly on vectors and matrices, for example. Trivial things like being able to refer to columns by a readable name rather than a numeric index makes a huge difference, before we even get into issues like columns of heterogeneous types, coherent handling of missing data, etc. This is why modelling in **R** is typically nicer than in certain other languages, where libraries for scientific and numerical computing existed for a long time before libraries for data frames were added to the language ecosystem. To build good libraries for statistical computing in Scala, it would be helpful to build those libraries using a good data frame implementation. There are a few different examples in existence, but unfortunately none have gained significant traction as yet.

For this course, we are not going to spend significant time working with the existing data frame libraries, since they are all rather experimental. Nevertheless, it seems sensible to briefly describe some of the available libraries, and to compare and contrast their features.

A simple data manipulation task

In order to illustrate the issues, I'm going to consider a very simple data manipulation task: first reading in a CSV file from disk into a data frame object, then filtering out some rows, then adding a derived column, then finally writing the data frame back to disk as a CSV file. We will start by looking at how this would be done in R. First we need an example CSV file. Since many **R** packages contain example datasets, we will use one of those. We could export `Cars93` from the **R** MASS package with the following **R** commands:

```
library(MASS)
write.csv(Cars93,"cars93.csv",row.names=FALSE)
```

If MASS isn't installed, it can be installed with a simple `install .packages("MASS")`. The above code snippet generates a CSV file to be used for the example. Typing `?Cars93` will give some information about the dataset, including the original source. Our analysis task is going to be to load the file from disk, filter out cars with `EngineSize` larger than 4 (litres), add a new column to the data frame, `WeightKG`, containing the weight of the car in KG, derived from the column `Weight` (in pounds), and then write back to disk in CSV format. This is the kind of thing that R is good at:

```
df=read.csv("cars93.csv")
print(dim(df))
df = df[df$EngineSize<=4.0,]
print(dim(df))
df$WeightKG = df$Weight*0.453592
print(dim(df))
write.csv(df,"cars93m.csv",row.names=FALSE)
```

That's how the task could be accomplished using **R**. Now let's see how a similar task could be accomplished using some different Scala data frames implementations.

6.4.1 Data frames and tables in Scala

Saddle

Saddle is probably the best known data frame library for Scala. It is strongly influenced by the Pandas library for Python. A simple Saddle session for accomplishing this task might proceed as follows:

```
val file = CsvFile("cars93.csv")
val df = CsvParser.parse(file).withColIndex(0)
println(df)
val df2 = df.rfilter(_("EngineSize").
  mapValues(CsvParser.parseDouble).
  at(0)<=4.0)
println(df2)
val wkg=df2.col("Weight").
  mapValues(CsvParser.parseDouble).
  mapValues(_*0.453592).setColIndex(Index("WeightKG"))
val df3=df2.
  joinPreserveColIx(wkg.mapValues(_.toString))
println(df3)
```

```
df3.writeCsvFile("saddle-out.csv")
```

Although this looks OK, it's not completely satisfactory, as the data frame is actually representing a matrix of `Strings`. Although you can have a data frame containing columns of any type, since Saddle data frames are backed by a matrix object (with type corresponding to the common super-type), the handling of columns of heterogeneous types always seems rather cumbersome. I suspect that it is this clumsy handling of heterogeneously typed columns that has motivated the development of alternative data frame libraries for Scala.

Scala-datatable

Scala-datatable is a lightweight minimal immutable data table for Scala, with good support for columns of differing types. However, it is currently really very minimal, and doesn't have CSV import or export, for example. That said, there are several CSV libraries for Scala, so it's quite easy to write functions to import from CSV into a datatable and write CSV back out from one. I've a couple of example functions, `readCsv()` and `writeCsv()` in the full code examples associated with this chapter. Now since datatable supports heterogeneous column types and I don't want to write a type guesser, my `readCsv()` function expects information regarding the column types. This could be relaxed with a bit of effort. An example session follows:

```
    val colTypes=Map("DriveTrain" -> StringCol,
                     "Min.Price" -> Double,
                     "Cylinders" -> Int,
                     "Horsepower" -> Int,
                     "Length" -> Int,
                     "Make" -> StringCol,
                     "Passengers" -> Int,
                     "Width" -> Int,
                     "Fuel.tank.capacity" -> Double,
                     "Origin" -> StringCol,
                     "Wheelbase" -> Int,
                     "Price" -> Double,
                     "Luggage.room" -> Double,
                     "Weight" -> Int,
                     "Model" -> StringCol,
                     "Max.Price" -> Double,
                     "Manufacturer" -> StringCol,
                     "EngineSize" -> Double,
                     "AirBags" -> StringCol,
                     "Man.trans.avail" -> StringCol,
                     "Rear.seat.room" -> Double,
                     "RPM" -> Int,
                     "Turn.circle" -> Double,
                     "MPG.highway" -> Int,
                     "MPG.city" -> Int,
                     "Rev.per.mile" -> Int,
                     "Type" -> StringCol)
    val df=readCsv("Cars93",new FileReader(
      "cars93.csv"),colTypes)
    println(df.length,df.columns.length)
    val df2=df.
      filter(row=>row.as[Double]("EngineSize")<=4.0).
      toDataTable
    println(df2.length,df2.columns.length)

    val oldCol=df2.columns("Weight").as[Int]
    val newCol=new DataColumn[Double]("WeightKG",
      oldCol.data.map{_.toDouble*0.453592})
    val df3=df2.columns.add(newCol).get
    println(df3.length,df3.columns.length)

    writeCsv(df3,new File("out.csv"))
```

Apart from the declaration of column types, the code is actually a little bit cleaner than the corresponding Saddle code, and the column types are all properly preserved and appropriately handled. However, a significant limitation of this data frame is that it doesn't seem to have special handling of missing values, requiring some kind of manually coded "special value" approach from users of this data frame. This is likely to limit the appeal of this library for general statistical and data science applications.

Framian

Framian is a more fully-featured data frame library for Scala, open-sourced by Pellucid analytics. It is strongly influenced by **R** data frame libraries, and aims to provide most of the features that **R** users would expect. It has good support for clean handling of heterogeneously typed columns (using Shapeless), handles missing data, and includes good CSV import:

```
val df=Csv.parseFile(new File("cars93.csv")).
  labeled.toFrame
println(""+df.rows+"␣"+df.cols)
val df2=df.
  filter(Cols("EngineSize").as[Double])( _ <= 4.0 )
println(""+df2.rows+"␣"+df2.cols)
val df3=df2.
  map(Cols("Weight").as[Int],"WeightKG")(r =>
    r.toDouble*0.453592)
println(""+df3.rows+"␣"+df3.cols)
println(df3.colIndex)
val csv = Csv.
  fromFrame(new CsvFormat(",", header = true))(df3)
new PrintWriter("out.csv") {
  write(csv.toString); close
}
```

This is arguably the cleanest solution so far. Unfortunately the output isn't quite right(!), as there currently seems to be a bug in `Csv.fromFrame` which causes the ordering of columns to get out of sync with the ordering of the column headers. This bug should be straightforward to fix, but no-one seems to be actively developing the library any more. Nevertheless, it is easy to write a CSV writer for these frames, as I did above for scala-datatable.

Spark DataFrames

The three data frames considered so far are all standard single-machine, non-distributed, in-memory objects. However, a `DataFrame` object has recently been added to Apache Spark. Spark is a Scala framework for the distributed processing and analysis of huge datasets on a cluster. We will examine it in more detail tomorrow. If you have a legitimate need for this kind of set-up, then Spark is a pretty impressive piece of technology. However, for datasets that can be analysed on a single machine, Spark is a rather slow and clunky sledgehammer to crack a nut. So, for datasets in the terabyte range and above, Spark DataFrames are great, but for datasets smaller than a few gigs, it's probably not the best solution. With those caveats in mind, here's an example of how to solve our problem using Spark DataFrames in the Spark Shell:

```
val df = spark.read.
         option("header", "true").
         option("inferSchema","true").
         csv("../r/cars93.csv")
val df2=df.filter("EngineSize␣<=␣4.0")
val col=df2.col("Weight")*0.453592
val df3=df2.withColumn("WeightKG",col)
df3.write.format("com.databricks.spark.csv").
                         option("header","true").
                         save("out-csv")
```

6.4.2 Summary

If you really need a distributed data frame library, then you will probably want to use Spark. However, for the vast majority of statistical modelling and data science tasks, Spark is likely to be unnecessarily complex and heavyweight. The other three libraries considered all have pros and cons. They are all largely one-person hobby projects, quite immature, and not currently under very active development. In particular, at the time of writing Saddle is the only library that has been updated for Scala 2.12 (and it still hasn't published artifacts). Saddle is fine for when you just want to add column headings to a matrix. Scala-datatable is lightweight and immutable, if you don't care about missing values. On balance, I think Framian is probably the most full-featured "batteries included" R-like data frame, and so is likely to be most attractive to statisticians and data scientists. However, it's very immature, appears to be abandoned, and the dependence on Shapeless may be of concern to those who prefer libraries to be lean and lightweight.

Actual runnable code for these examples is included in the Examples section of the course repo.

Part III

Tools and libraries

Chapter 7

Scala tools

7.1 Using SBT to experiment with new libraries in the Scala REPL

7.1.1 Setting dependencies interactively

We've seen how easy it is to use SBT to add dependencies on third party libraries by adding them into a `build.sbt` file. But when you just want to quickly experiment with a library/API, creating a new project with an appropriate build file can seem like a bit of a chore. Fortunately, it is very easy to interactively add dependencies into a running SBT session, and this can be very convenient. To illustrate this, start SBT from an empty directory, or some kind of `/tmp/` directory which doesn't contain any files ending `.sbt` or `.scala`. This will give you a "vanilla" SBT session. We know that if we type `console` we will get a Scala REPL, but which version of Scala will it be? Before typing `console`, we can set the Scala version from the *SBT prompt* with

```
set scalaVersion := "2.12.1"
```

Now when we type `console` we will get a 2.12.1 REPL. Obviously you can set this to any version of Scala you like, so this is also a good way to experiment with different versions of Scala. Having set an appropriate version of Scala, we can next add any dependencies (again, this should be done from the SBT prompt, not the Scala REPL). For example, we can add a dependency on Breeze with:

```
set libraryDependencies += "org.scalanlp" %% "breeze" % "0.13"
set libraryDependencies += "org.scalanlp" %% "breeze-natives" % "0.13"
```

Now when we type `console` we will get a REPL with a Breeze dependency. This approach to adding in dependencies interactively can be very useful for "quick–and–dirty" experiments with new libraries in the REPL.

7.1.2 sbt new

Recent versions of SBT support the command line option **new**, allowing quick–and–easy creation of SBT project templates pre-configured for particular applications, utilising the *giter8* templating system. For example, a minimal seed project can be created by typing:

```
sbt new scala/scala-seed.g8
```

at your OS command prompt, from a directory where you want to create a new SBT project as a sub-directory. Templates typically query a project name, and can also query other setup information. There are templates for many applications. For example, if you are interested in the ScalaFX GUI library for building graphical applications, you can create a minimal project template with:

A list of available templates can be found at: https://github.com/foundweekends/giter8/wiki/giter8-templates

```
sbt new scalafx/scalafx.g8
```

After creating the template, entering the project directory and typing `sbt run` will compile and run a minimal GUI application.

To search for a template using, eg., Google, just search for the name of the library and add `g8` onto the end. eg. To search for an `akka-streams` template, search for `akka-streams g8`. But beware that this can often return very old templates.

I have created a giter8 template for Breeze projects, which can be used with

```
sbt new darrenjw/breeze.g8
```

This is arguably easier and better than copying the `app-template` directory in the GitHub repo associated with this course, though it does require an internet connection.

7.2 Building standalone applications (assembly JARs)

So far we have been building and running our code using SBT. But sometimes we would like to be able to run our code without using SBT. Indeed, we would sometimes like to be able to run our code on systems where SBT isn't installed. For a project with (multiple) dependencies on third party libraries, the simplest way to do this is to build a "fat" (assembly) Java archive (JAR) file containing all of the compiled code associated with the project in addition to the Scala standard library and all third party libraries required to run the application. This assembly JAR can then be run on any machine with a (compatible) JVM. This is simplest to illustrate with an example. Suppose we have an SBT project containing a single Scala source file as follows.

This is example `C7-MetropAssembly` in the GitHub repo.

```
object Metropolis {

  import breeze.stats.distributions._

  val chain = MarkovChain.
    metropolisHastings(0.0,
      (x: Double) => Uniform(x-0.5, x+0.5))(x =>
        Gaussian(0.0, 1.0).logPdf(x)).steps

  def main(args: Array[String]): Unit = {
    val n = if (args.size == 0) 10 else args(0).toInt
    chain.take(n).toArray.foreach(println)
  }

}
```

We can build and run this by typing `sbt run` from the OS command prompt in the top level directory of the SBT project. If we want to run the chain for 20 iterations, we can do so with `sbt "run 20"`. To build an assembly JAR we need to use the `sbt-assembly` SBT plugin. To add this to a project, create a file called `project/plugins.sbt` and add the single line

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.14.4")
```

This adds a new SBT task, `assembly`. So we can now build an assembly JAR by running `sbt assembly` from the OS command prompt. This should build the JAR and put it in `target/scala-2.12/`. We can run this without SBT with a command like

```
java -jar target/scala-2.12/metropolis-assembly-assembly-0.1.jar
```

or

```
java -jar target/scala-2.12/metropolis-assembly-assembly-0.1.jar 20
```

So we can now deploy and run this on any system with a (compatible) JVM. This is exceptionally convenient in the context of cluster/-Cloud computing environments. Also, we shall see shortly that it is also convenient if you want to call Scala code from other systems and languages, such as **R**.

7.3 Interfacing Scala with R

Unfortunately, for the time being **R** remains the "default" language for statistical computing and data science, and has thousands of packages (on CRAN) implementing a huge variety of statistical methods and algorithms. Also, the statistical plotting libraries available in **R** are still somewhat superior to those easily available in Scala. Consequently, **R** is likely to remain part of the statistical computing ecosystem for the foreseeable future (for better or worse), and so having mechanisms for using **R** and Scala together in statistical applications is clearly valuable.

There are obviously two different ways that one could use **R** and Scala together. One would be to call **R** functionality from a Scala application (embedding **R** in Scala), perhaps to fit a model using a sophisticated model–fitting package like `lme4`, not routinely available as a Scala library, or alternatively to call a plotting function that will produce a publication–quality plot, possibly using **R**'s `ggplot` library. The other way to use **R** and Scala together would be to call a Scala function from an **R** script (embedding Scala in **R**). Here a typical use–case would be that you have a pre–existing **R** application that is too slow, and so you want to re-write the slow parts in Scala and then call the Scala version from your **R** script. Both of these ways of combining **R** and Scala are addressed by the `rscala` package.

https://github.com/dbdahl/rscala

7.3.1 Calling R from Scala

For this to work, you must have a system–wide installation of a recent version of R on your computer.

We will begin by seeing how easy it is to use `rscala` to embed **R** in a Scala application. Start by running SBT from an empty/temp directory, and add a dependency on `rscala` by entering:

```
set scalaVersion := "2.12.1"
set libraryDependencies += "org.ddahl" %% "rscala" % "2.0.1"
console
```

to get an appropriately configured Scala REPL.

An **R** session can be started from within Scala with

```
val R = org.ddahl.rscala.RClient()
// R: org.ddahl.rscala.RClient = RClient@9fc5dc1
```

The return value (here `R`) is a handle on the **R** session. Note that this starts **R** using the default **R** command, which can be obtained with

```
org.ddahl.rscala.RClient.defaultRCmd
// res0: String = R
```

If you start up **R** with a non-standard command (for example, because you have multiple versions of **R** installed on your system), this can be passed into the `RClient` method. Now we have a handle on an **R** session, we can evaluate **R** expressions and have the results returned back to Scala. For example, we can evaluate a function returning a scalar `Double` with

```
val d0 = R.evalD0("rnorm(1)")
// d0: Double = 0.945922465932532
```

Here the `D` stands for `Double` and the `0` for 0–dimensional, ie. a scalar. So we can evaluate a function returning a vector of `Double`s with

```
val d1 = R.evalD1("rnorm(5)")
// d1: Array[Double] = Array(-0.8272179841496433, ...
```

Note how the Scala return type is `Array[Double]`. Similarly, an **R** expression returning a matrix of `Double`s can be evaluated with

```
val d2 = R.evalD2("matrix(rnorm(6),nrow=2)")
// d2: Array[Array[Double]] = Array(Array(
//      -0.7545734628207127, ...
```

Note how the return type is `Array[Array[Double]]`. We can send data from Scala to **R** by creating **R** variables containing the data. For example, we can create an **R** variable called `vec` with

```
R.vec = (1 to 10).toArray // send data to R
// R.vec: (Any, String) = ([I@1e009fac,Array[Int])
R.evalI1("vec")
// res1: Array[Int] = Array(1,2,3,4,5,6,7,8,9,10)
```

Here, `I` is for `Int`. We can evaluate multi–line expressions (and discard any return value) as follows.

```
R eval """
vec2␣=␣rep(vec,3)
vec3␣=␣vec2␣+␣1
```

```
mat1 = matrix(vec3,ncol=5)
"""
```

If we want to just "get" an **R** variable, we can do so:

```
R.getI2("mat1") // get data back from R
// res3: Array[Array[Int]] = Array(Array(2, 8, 4, ...
```

We will finish this section with a more realistic (Breeze–based) example, where we have data in Scala that we want to fit using a model–fitting routine available in **R**. We start by simulating some data from Poisson regression model (in Scala).

```
import breeze.stats.distributions._
import breeze.linalg._
import org.ddahl.rscala.RClient
val x = Uniform(50,60).sample(1000)
// x: IndexedSeq[Double] = Vector(50.54008541753607, ...
val eta = x map (xi => (xi * 0.1) - 3)
// eta: IndexedSeq[Double] = Vector(2.054008541753607, ...
val mu = eta map math.exp
// mu: IndexedSeq[Double] = Vector(7.799101554600703, ...
val y = mu map (Poisson(_).draw)
// y: IndexedSeq[Int] = Vector(8, 15, 12, ...
```

Now we have our data in Scala, we can create a connection to **R** and use it to fit our model.

```
val R = RClient() // initialise an R interpreter
// R: RClient = RClient@45768f22
R.x=x.toArray // send x to R
// R.x: (Any, String) = ([D@6c7e65fb,Array[Double])
R.y=y.toArray // send y to R
// R.y: (Any, String) = ([I@65a9a726,Array[Int])
R.eval("mod = glm(y~x,family=poisson())") // fit
// pull the fitted coefficents back into scala
DenseVector[Double](R.evalD1("mod$coefficients"))
// res6: breeze.linalg.DenseVector[Double] =
//   DenseVector(-2.98680078148213,
//     0.09959046899061315)
```

7.3.2 Calling Scala from R

Introduction

It can also sometimes be useful to embed Scala in **R** so that Scala code can be called from **R**. To do this we need to install the CRAN package `rscala` in **R**. Assuming that both **R** and a JVM is installed on your system, installing this should be as simple as typing

```
install.packages("rscala")
```

at the R command prompt. Calling

```
library(rscala)
```

will check that it has worked. The package will do a sensible search for a Scala installation and use it if it can find one. If it can't find one it will fail. In this case you can download and install a Scala installation specifically for `rscala` using the command

```
scalaInstall()
```

This option is likely to be attractive to `sbt` (or IDE) users who don't like to rely on a system-wide scala installation.

Getting started

A connection to a Scala interpreter can be constructed using

```
> sc = scala()
```

It's possible to find out information about the interpreter with

```
> scalaInfo(verbose=TRUE)

Searching for a suitable Scala installation.
  * FAILURE: 'scala.home' argument is NULL
  * FAILURE: SCALA_HOME () environment variable
  * ATTEMPT: Found a candidate (/usr/share/scala-2.11/
     bin/scala)
  * SUCCESS: 'scala' in the shell's search path

$cmd
[1] "/usr/share/scala-2.11/bin/scala"

$home
[1] "/usr/share/scala-2.11"

$version
[1] "2.11.6"

$major.version
[1] "2.11"
```

As usual in **R**, it is possible to get help on commands in the usual way.

```
> help(package="rscala")
> ?scala
```

We can send a command to the Scala interpreter using the `%~%` operator

```
> sc %~% 'println("hello world")'
hello world
```

If we want, we can send instructions one line at a time.

```
> sc %~% 'val x = Array(1,2,3)'
[1] 1 2 3
> sc %~% 'val y = x map (_*2)'
[1] 2 4 6
```

And we can get data back into R using `$` notation.

```
> ry = sc$y # get data back to R
> print(ry)
[1] 2 4 6
```

However, it is often more convenient to send a collection of expressions and use the fact that the result of the final expression will be returned to **R**.

```
> s = sc %~% '
+ val x = List(1,2,3)
+ (x map (_+1)).sum
+ '
> print(s)
[1] 9
```

It is similarly straightforward to send data from **R** to Scala.

```
> rx = rnorm(5)
> print(rx)
[1] -0.1331895  1.3776800 -0.7991593 -0.3040138
   0.6587695
> sc$sx = rx # send data to Scala
> sc %~% 'val sy = sx map (xi => xi*xi)'
[1] 0.01773945 1.89800218 0.63865559 0.09242442
   0.43397726
> ry = sc$sy # get data back to R
> print(ry)
[1] 0.01773945 1.89800218 0.63865559 0.09242442
   0.43397726
```

Note that for **R** to understand the result, you should typically use expressions evaluating to primitive numeric types or `Array`s of primitive types. There is also a convenient string interpolation notation `@{ }`, which is useful for passing arguments from **R** into Scala code.

```
> n=10
> sc %~% '(1 to @{n}).toArray' # pass argument from R
 [1]  1  2  3  4  5  6  7  8  9 10
```

Finally, when one is finished with the Scala interpreter, it should be closed to free up resources.

```
> close(sc)
```

Calling into compiled Scala code (with dependencies)

Typically one will want to run Scala code from compiled SBT projects, usually having third–party library dependencies (such as Breeze). In this case, the simplest way to call such code from **R** via `rscala` is via an assembly JAR, constructed as described in section 7.2. Using the example from section 7.2, we can call this from an **R** session (with working directory the top level of the SBT project) roughly as follows.

```
library(rscala)
sc = scala(
 "target/scala-2.12/metropolis-assembly-assembly-0.1.jar",
 scala.home="~/.rscala/scala-2.12.2"
)
met = sc %~% 'Metropolis.chain.take(10000).toArray'
```

The first argument to the `scala` function is the path to the assembly JAR. This should be the only argument required if the system-wide installation of Scala is compatible with the version used to build the assembly JAR. If not, the `scala.home` option must be set to a Scala installation which is. Here I point at an installation provided by `scalaInstall()`. We can then run the compiled code by evaluating an appropriate expression.

To check that this has worked as expected, we can use the `mcmcSummary` function from my `smfsb` CRAN package.

You can install this with `install.packages("smfsb")` if you don't have it.

```
library(smfsb)
mcmcSummary(matrix(met,ncol=1))
```

This should provide some plots and summary statistics indicating that the chain is targeting a standard normal distribution.

Summary

The CRAN package `rscala` makes it very easy to embed a Scala interpreter within an **R** session and vice versa. However, for most non-trivial statistical computing problems, the Scala code will have dependence on external scientific libraries such as Breeze. The standard way to easily manage external dependencies in the Scala ecosystem is SBT. Given an SBT–based Scala project, it is easy to generate an assembly JAR in order to initialise the `rscala` Scala interpreter with the classpath needed to call arbitrary Scala functions. This provides very convenient inter-operability between **R** and Scala for many statistical computing applications.

7.4 CSV parsing libraries

As we have seen, the CSV parsing function built in to Breeze is rather rudimentary. When it proves inadequate, a more customisable parsing library is needed. There are many parsing libraries for Scala, and we unfortunately do not have time to explore them in the context of this course, but there are a couple of popular CSV parsing libraries worth mentioning here for future reference. The most popular library seems to be `scala-csv`. This is very simple, but flexible. Other popular Scala libraries for CSV parsing include `kantan.csv` and `PureCSV`, but there are many others. However, some prefer to use some of the more sophisticated Java parsing libraries, such as `univocity`.

`github.com/tototoshi/scala-csv`

`github.com/nrinaudo/kantan.csv`

`github.com/melrief/PureCSV`

`github.com/uniVocity/univocity-parsers`

7.5 Testing

Testing code to check that it behaves as expected is very important, but is not often taken sufficiently seriously in the scientific and statistical computing communities. Fortunately the Scala and SBT ecosystem is full of tools to make testing code less of a chore. We will begin with some very simple features built into the Scala standard library.

7.5.1 Assert and require

We have already seen the use of the `require` function in section 6.1.2. This is typically used to document consistency requirements of function arguments. An expression returning a `Boolean` is provided to the function, with **true** corresponding to satisfying the requirements. If the expression evaluates to **false**, the function will throw an exception (an `IllegalArgumentException`). Note that `require` provides testing at runtime, and therefore there will always be some runtime penalty associated with using `require`. It should therefore be used sparingly for functions that are likely to be called frequently in an inner–loop of some sort.

A useful alternative to `require` is `assert`, which can be used in many different contexts within a code–base. Again, it has a `Boolean` expression with **true** corresponding to the assertion and **false** corresponding to a violation of the assertion, typically resulting in the throwing of an exception, this time an `AssertionError`. If the assertion is used near the start of a function, it can be used to specify *pre–conditions*, similarly to `require`. If it is used near the end of a function, in can be used to specify *post–conditions*. In fact, `assert` can be used anywhere in code to document the programmer's expectations about the way that things should be at that point. Like `require`, `assert` is called at runtime, and so there is potentially a runtime penalty associated with using `assert`. However, there is an important difference with `assert`, in that assertions can be disabled at compile time, so that they are not checked, and therefore incur no runtime penalty. The idea is that they can be used for development, testing and debugging, but then turned off in production code for performance reasons. This means that one can be much more relaxed about the liberal use of assertions in code, safe in the knowledge that they can be disabled when runtime performance is critical. Some simple examples are given below.

Pre– and post–conditions are an important part of a *design–by–contract* approach to software engineering, and are built–in to some languages (such as *Eiffel*).

```
require(1 == 1) // satisfied
// require(1 == 2) // throws exception
assert (1 == 1) // satisfied
// assert (1 == 2) // throws exception
```

A more interesting example is given below, which uses `require` for checking a pre–condition and `assert` for checking a post–condition.

```
def sqrt(x: Double): Double = {
  require(x >= 0.0) // pre-condition
  val ans = math.sqrt(x)
  assert(math.abs(x-ans*ans) < 0.00001) // post-condition
  ans
}

sqrt(2.0) // works as expected
// sqrt(-2.0) // throws exception
```

Note that assertions can be disabled simply by adding the line

```
scalacOptions += "-Xdisable-assertions"
```

to the project's SBT build file.

There is also a function called `assume`, which behaves identically to `assert`, but is semantically different. The intention is that `assume` corresponds to an axiom and `assert` corresponds to a statement that is required to be proved. This difference is important mainly in the context of static code analysis.

7.5.2 scalatest

www.scalatest.org

`scalatest` is a widely used unit–testing framework for Scala. The idea behind unit–testing is that every function you write should be tested with a few test cases to ensure that it behaves as expected. These test cases can be kept in a test–suite, and re-run frequently to ensure that the code hasn't been changed in such a way as to cause the test cases to fail. Unit–tests can be run frequently, potential every time a source–file is saved. But the philosophy is quite consistent with that of statically–typed compiled languages in general — do as much work and checking as possible at compile–time to save processor cycles at runtime.

Within an SBT project, unit test code is put in `src/test/scala` (rather than `src/main/scala`), and can be run with the `test` task in SBT. To re-run all unit tests whenever a source code file changes, run the `~test` task.

`scalatest` supports the development of tests in a number of different styles. The `FlatSpec` style seems to be the most popular. A complete `FlatSpec` test file is given below.

```
import org.scalatest.FlatSpec

class SetSpec extends FlatSpec {

  "An empty Set" should "have size 0" in {
    assert(Set.empty.size == 0)
  }

  it should "throw an exception with head" in {
    assertThrows[NoSuchElementException] {
      Set.empty.head
    }
  }

}
```

An example of a more statistical test might be:

```
  "A Gamma(3.0,4.0)" should "have mean 12.0" in {
    import breeze.stats.distributions.Gamma
    val g = Gamma(3.0,4.0)
    val m = g.mean
    assert(math.abs(m - 12.0) < 0.000001)
  }
```

As well as testing your own code, tests can be useful for testing your understanding of the API of a third–party library.

7.5.3 scalacheck

`scalacheck` is an interesting alternative to conventional unit–test based frameworks, which instead advocates *property–based testing*. The idea is that rather than focusing on a few carefully chosen test–cases, one instead specifies the expected behaviour of a function (corresponding loosely to a post–condition), and the test–suite randomly generates a large number of test–cases to check that the requirements are satisfied. Time constraints prevent us from exploring this in detail within the context of this course, but I recommend exploring the use of this library as time permits.

www.scalacheck.org

7.6 Benchmarking and profiling

In Chapter 5 we looked briefly at how to time functions to get some sort of idea about the relative performance of different implementations of an algorithm. But benchmarking isn't completely straightforward, and especially so on the JVM, which has garbage collection, dynamic class loading, JIT compilation, etc. So to benchmark at all reliably on the JVM, one needs to warm up the JVM and do several repeats of the benchmark, typically averaging the results. It's not perfect, but it's better than not doing so. `scalameter` is a library to help automate this process. Again, time constraints prevent us from examining this library in detail now, but I recommend using it for benchmarking code performance.

scalameter.github.io

Similarly, the profiling of running code is a often of interest, especially in performance–critical situations. Here, *VisualVM* seems to be the most popular choice. It will profile any application running on the JVM.

visualvm.github.io

7.7 Documentation using ScalaDoc

Specially formatted comments can be used to document code. The example below shows how to document a function. Note how the comment starts `/**` rather than the usual `/*` for a multi-line comment.

http://docs.scala-lang.org/style/scaladoc.html

```scala
/**
  *  Take every th value from the stream s of type T
  *
  * @param s A Stream to be thinned
  * @param th Thinning interval
  *
  * @return The thinned stream, with values of
  *  the same type as the input stream
  */
def thinStream[T](s: Stream[T],th: Int): Stream[T] = {
  val ss = s.drop(th-1)
  if (ss.isEmpty) Stream.empty else
    ss.head #:: thinStream(ss.tail, th)
}
```

HTML documentation can be generated for an application by running the `doc` SBT task, and this will be written to `target/scala-2.12/api/`. Pointing a web browser at this directory will then allow browsing of the interactive API documentation. IDEs will often assist with the formatting of ScalaDoc comments.

Chapter 8

Apache Spark

8.1 Getting started with the Spark Shell

8.1.1 Introduction

Apache Spark is a Scala library for analysing “big data”. It can be used for analysing huge (internet-scale) datasets distributed across large clusters of machines. The analysis can be anything from the computation of simple descriptive statistics associated with the datasets, through to rather sophisticated machine learning pipelines involving data pre-processing, transformation, nonlinear model fitting and regularisation parameter tuning (via methods such as cross-validation). A relatively impartial overview can be found in the Apache Spark Wikipedia page.

`spark.apache.org`

Although Spark is really aimed at data that can’t easily be analysed on a laptop, it is actually very easy to install and use (in standalone mode) on a laptop, and a good laptop with a fast multicore processor and plenty of RAM is fine for datasets up to a few gigabytes in size. This chapter will walk through getting started with Spark, installing it locally (not requiring admin/root access) doing some simple descriptive analysis, and moving on to fit simple regression models to some simulated data. After this it should be relatively easy to take things further by reading the Spark documentation, which is generally pretty good.

Anyone who is interested in learning more about setting up and using Spark clusters may want to have a quick look over on my personal blog (mainly concerned with the Raspberry Pi), where I have previously considered installing Spark on a Raspberry Pi 2, setting up a small Spark cluster, and setting up a larger Spark cluster. Although these posts are based around the Raspberry Pi, most of the material there is quite generic, since the Raspberry Pi is just a small (Debian-based) Linux server.

`darrenjw2.wordpress.com`

8.1.2 Getting started — installing Spark

The only pre-requisite for installing Spark is a recent Java installation. Once you have Java installed, you can download and install Spark in any appropriate place in your file-system. If you are running Linux, or a Unix-alike, unpack and test your download with the following commands:

```
tar xvfz spark-2.1.0-bin-hadoop2.7.tgz
cd spark-2.1.0-bin-hadoop2.7
bin/run-example SparkPi 10
```

If all goes well, the last command should run an example. Don't worry if there are lots of INFO and WARN messages — we will sort that out shortly. On other systems it should simply be a matter of downloading and unpacking Spark somewhere appropriate, then running the example from the top-level Spark directory. Get Spark from the downloads page. You should get version 2.1.0 built for Hadoop 2.7. It doesn't matter if you don't have Hadoop installed — it is not required for single-machine use.

The INFO messages are useful for debugging cluster installations, but are too verbose for general use. On a Linux system you can turn down the verbosity with:

```
sed 's/rootCategory=INFO/rootCategory=WARN/g' < \
  conf/log4j.properties.template > \
  conf/log4j.properties
```

On other systems, copy the file `log4j.properties.template` in the `conf` sub-directory to `log4j.properties` and edit the file, replacing `INFO` with `WARN` on the relevant line. Check it has worked by re-running the `SparkPi` example — it should be much less verbose this time. You can also try some other examples:

```
bin/run-example SparkLR
ls examples/src/main/scala/org/apache/spark/examples/
```

There are several different ways to use Spark. For this walk-through we are just going to use it interactively from the "Spark shell". We can pop up a shell with:

```
bin/spark-shell --master local[4]
```

The "4" refers to the number of worker threads to use. Four is probably fine for most decent laptops. `Ctrl-D` or `:quit` will exit the Spark shell and take you back to your OS shell. It is more convenient to have the Spark `bin` directory in your path. If you are using `bash` or a similar OS shell, you can temporarily add the Spark `bin` to your path with the OS shell command:

```
export PATH=$PATH:`pwd`/bin
```

You can make this permanent by adding a line like this (but with the full path hard-coded) to your `.profile` or similar start-up dot-file. I prefer not to do this, as I typically have several different Spark versions on my laptop and want to be able to select exactly the version I need. If you are not running `bash`, Google how to add a directory to your path. Check the path update has worked by starting up a shell with:

```
spark-shell --master local[4]
```

Note that if you want to run a script containing Spark commands to be run in "batch mode", you could do it with a command like:

```
spark-shell --driver-memory 25g --master local[4] < \
  spark-script.scala | tee script-out.txt
```

There are much better ways to develop and submit batch jobs to Spark clusters, but we will discuss these later. Note that while Spark is running, diagnostic information about the "cluster" can be obtained by pointing a web browser port 4040 on the master, which here is just `http://localhost:4040/` — this is extremely useful for debugging purposes.

The Spark shell is essentially just a Scala REPL with a dependency on Spark and a couple of pre-defined values.

8.1.3 First Spark shell commands

Counting lines in a file

We are now ready to start using Spark. From a Spark shell in the top-level directory, enter:

```
// Spark context available as 'sc' (master = local[4]).
// Spark session available as 'spark'.
// Welcome to
//       ____              __
//      / __/__  ___ _____/ /__
//     _\ \/ _ \/ _ ‘/ __/  ’_/
//    /___/ .__/\_,_/_/ /_/\_\   version 2.1.0
//       /_/
//
// Using Scala version 2.11.8
// Type in expressions to have them evaluated.
// Type :help for more information.

sc.textFile("README.md").count
// res0: Long = 104
```

If all goes well, you should get a count of the number of lines in the file "README.md". The value `sc` is the "Spark context", containing information about the Spark cluster (here it is just a laptop, but in general it could be a large cluster of machines, each with many processors and each processor with many cores). The `textFile` method loads up the file into an RDD (Resilient Distributed Dataset). **The RDD is the fundamental abstraction provided by Spark.** *It is a lazy distributed parallel immutable monadic collection.* After loading a text file like this, each element of the collection represents one line of the file. The point is that although RDDs are potentially huge and distributed over a large cluster, using them is very similar to using any other monadic collection in Scala. We can unpack the previous command slightly as follows:

```
val rdd1 = sc.textFile("README.md")
// rdd1: RDD[String] = README.md MapPartitionsRDD[3]
rdd1
// res1: RDD[String] = README.md MapPartitionsRDD[3]
rdd1.count
// res2: Long = 104
```

Note that RDDs are “lazy” (unlike most other Scala collections), and this is important for optimising complex pipelines. So here, after assigning the value `rdd1`, no data is actually loaded into memory. All of the actual computation is deferred until an “action” is called — `count` is an example of such an action, and therefore triggers the loading of data into memory and the counting of elements.

Counting words in a file

We can now look at a very slightly more complex pipeline — counting the number of words in a text file rather than the number of lines. This can be done as follows:

```
sc.textFile("README.md").
  map(_.trim).
  flatMap(_.split(' ')).
  count
// res3: Long = 535
```

Note that `map` and `flatMap` are both lazy (“transformations” in Spark terminology), and so no computation is triggered until the final action, `count` is called. The call to `map` will just trim any redundant white-space from the line ends. So after the call to `map` the RDD will still have one element for each line of the file. However, the call to `flatMap` splits each line on white-space, so after this call each element of the RDD will correspond to a word, and not a line. So, the final `count` will again count the number of elements in the RDD, but here this corresponds to the number of words in the file.

Counting character frequencies in a file

A final example before moving on to look at quantitative data analysis: counting the frequency with which each character occurs in a file. This can be done as follows:

```
sc.textFile("README.md").
  map(_.toLowerCase).
  flatMap(_.toCharArray).
  map{(_,1)}.
  reduceByKey(_+_).
  collect
// res4: Array[(Char, Int)] = Array((d,97), (z,3),
//  (  ,1), (",12), (`,6), (p,144), (x,13), (t,233),
// (.,61), (0,13), (b,44), (h,116), ( ,464), ...
```

The first call to `map` converts upper case characters to lower case, as we don’t want separate counts for upper and lower case characters. The call to `flatMap` then makes each element of the RDD correspond to a single character in the file. The second call to `map` transforms each element of the RDD to a key-value pair, where the key is the character and the value is the integer 1. RDDs have special methods for key-value pairs in this form — the method `reduceByKey` is one such — it applies the reduction operation (here just `+`) to all values corresponding to a particular value of the key. Since each character has the value 1, the sum of the values will be a character count. Note that the reduction will be done in parallel, and for this to work it is vital that the reduction operation is associative. Simple addition of integers is clearly associative, so here we are fine. Note that `reduceByKey` is a (lazy) transformation, and so the computation needs to be triggered by a call to the action `collect`.

On most Unix-like systems there is a file called `words` that is used for spell-checking. The example below applies the character count to this file. Note the calls to `filter`, which filter out any elements of the RDD not matching the predicate. Here it is used to filter out special characters. We also sort the results by value (rather than by key), using **false** to indicate descending order.

```
sc.textFile("/usr/share/dict/words").
  map(_.trim).
  map(_.toLowerCase).
  flatMap(_.toCharArray).
  filter(_ > '/').
  filter(_ < '}').
  map{(_,1)}.
  reduceByKey(_+_).
  sortBy(_._2,false).
  collect
// res5: Array[(Char, Int)] = Array((s,90113),
//   (e,88833), (i,66986), (a,64439),
//   (r,57347), (n,57144), ...
```

8.1.4 Caching RDDs with `persist`

Before continuing further, it is worth dwelling briefly on issues of resilience, re-computation and persistence. The RDD is resilient in the sense that it knows how to re-compute itself if (say) one node of the Spark cluster fails. Spark computations are largely sequences of lazy transformations, and these computations can be "re-played" if necessary. But care must be taken not to unnecessarily re-do computations that have already been done when it is not necessary. For example, suppose that `rdd` is an `RDD[Double]`, resulting from a long chain of transformations, starting with loading a huge file from disk. Creating the `rdd` is near-instantaneous, since it is a sequence of transformations, and so no actual computation will take place. But now suppose that there are a couple of different actions we want to call on this RDD. First, we would like to know how many elements are in the RDD, so we call `rdd.count`. At this point the computation is triggered, and it could take a long time to get our result. But now suppose that we also want to know the sum of the elements in the RDD, and call `rdd.sum`. We have again triggered a computation by calling an action, but potentially, we will re-execute the entire computational pipeline which results in `rdd`, which we have already done to get our count. This is clearly very unsatisfactory.

The lazy transformation `persist` is used to cache a snapshot of the current state of an RDD, precisely to avoid this kind of unnecessary re-computation. So if in forming our `rdd`, the final transformation used was `persist`, then when the actual content of the RDD is computed when `count` triggers it, the state of `rdd` will be cached, so that when the new action `sum` is called, computation of `rdd` does not have to begin again from scratch. These lazy evaluation semantics take a bit of getting used to, since they are somewhat different to regular Scala collections. Knowing when or whether to persist an RDD is a common source of confusion for Spark beginners. But note that the lazy semantics are very powerful, as they allow analysis and optimisation of the entire computational pipeline, and can help to speed up computations by minimising shuffling between nodes of a Spark cluster.

8.2 Commonly used RDD methods

Remember that **transformations** are *lazy* and **actions** are *strict*. An action must be called to trigger computation. Methods returning an RDD are transformations and methods returning other types are actions.

8.2.1 Transformations of an `RDD[T]`

```
def ++(other: RDD[T]): RDD[T]
def cartesian[U](other: RDD[U]): RDD[(T, U)]
def distinct: RDD[T]
def filter(f: (T) => Boolean): RDD[T]
def flatMap[U](f: (T) => TraversableOnce[U]): RDD[U]
def map[U](f: (T) => U): RDD[U]
def persist: RDD[T]
def sample(withReplacement: Boolean,
  fraction: Double): RDD[T]
def sortBy[K](f: (T) => K,
  ascending: Boolean = true): RDD[T]
def zip[U](other: RDD[U]): RDD[(T, U)]
```

8.2.2 Actions on an `RDD[T]`

```
def aggregate[U](zeroValue: U)(seqOp: (U, T) => U,
  combOp: (U, U) => U): U
def collect: Array[T]
def count: Long
def fold(zeroValue: T)(op: (T, T) => T): T
def foreach(f: (T) => Unit): Unit
def reduce(f: (T, T) => T): T
def take(num: Int): Array[T]
```

8.2.3 Transformations of a Pair RDD, `RDD[(K,V)]`

```
def aggregateByKey[U](zeroValue: U)(seqOp: (U, V) => U,
  combOp: (U, U) => U): RDD[(K, U)]
def groupByKey(): RDD[(K, Iterable[V])]
def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]
def keys: RDD[K]
def mapValues[U](f: (V) => U): RDD[(K, U)]
def reduceByKey(func: (V, V) => V): RDD[(K, V)]
def values: RDD[V]
```

8.2.4 Actions on a `RDD[(K,V)]`

```
def countByKey: Map[K, Long]
```

8.2.5 `DataFrames`

```
def toDF(colNames: String*): DataFrame
def col(colName: String): Column
def drop(col: Column): DataFrame
def select(cols: Column*): DataFrame
def show(numRows: Int): Unit
def withColumn(colName: String, col: Column): DataFrame
def withColumnRenamed(existingName: String,
  newName: String): DataFrame
```

8.3 Analysis of quantitative data

8.3.1 Descriptive statistics

We first need some quantitative data, so let's simulate some using Breeze. *Spark has a dependence on Breeze*, and therefore can be used from inside the Spark shell — this is very useful. So, we start by using Breeze to simulate a vector of normal random quantities:

```
import breeze.stats.distributions._
// import breeze.stats.distributions._
val x = Gaussian(1.0,2.0).sample(10000)
// x: IndexedSeq[Double] = Vector(0.17848673281799982,
// 5.270572111051524, -1.5529505975275635,
// 0.11506296796076387, -0.29372435649818396, ...
```

Note, though, that `x` is just a regular Breeze Vector, a simple serial collection all stored in RAM on the master thread. To use it as a Spark RDD, we must convert it to one, using the `parallelize` function:

```
val xRdd = sc.parallelize(x)
// xRdd: RDD[Double] = ParallelCollectionRDD[23]
```

Now `xRdd` is an RDD, and so we can do Spark transformations and actions on it. There are some special methods for RDDs containing numeric values:

```
xRdd.mean
// res6: Double = 0.979624756565341
xRdd.sampleVariance
// res7: Double = 4.068248733184799
```

Each summary statistic is computed with a single pass through the data, but if several summary statistics are required, it is inefficient to make a separate pass through the data for each summary, so the `stats` method makes a single pass through the data returning a `StatsCounter` object that can be used to compute various summary statistics.

```
val xStats = xRdd.stats
// xStats: StatCounter = (count: 10000,
//  mean: 0.979625, stdev: 2.016889,
//  max: 10.478370, min: -8.809859)
xStats.mean
// res8: Double = 0.979624756565341
xStats.sampleVariance
// res9: Double = 4.068248733184799
xStats.sum
// res10: Double = 9796.24756565341
```

The `StatsCounter` methods are: `count`, `mean`, `sum`, `max`, `min`, `variance`, `sampleVariance`, `stdev`, `sampleStdev`.

8.3.2 Linear regression

Moving beyond very simple descriptive statistics, we will next look at a simple linear regression model, which will also allow us to introduce Spark `DataFrame`s - a high level abstraction layered on top of RDDs which makes working with tabular data much more convenient, especially in the context of statistical modelling.

We start with some standard (non-Spark) Scala Breeze code to simulate some data from a simple linear regression model. We use the `x` already simulated as our first covariate. Then we simulate a second covariate, `x2`. Then, using some residual noise, `eps`, we simulate a regression model scenario, where we know that the “true” intercept is 1.5 and the “true” covariate regression coefficients are 2.0 and 1.0.

```
val x2 = Gaussian(0.0,1.0).sample(10000)
// x2: IndexedSeq[Double] = Vector(0.5814691557760885,
// 1.1171111465724972, 0.29188249592720783, ...
val xx = x zip x2
// xx: IndexedSeq[(Double, Double)] = Vector(
// (0.17848673281799982,0.5814691557760885),
// (5.270572111051524,1.1171111465724972), ...
val lp = xx map {p => 2.0*p._1 + 1.0*p._2 + 1.5}
// lp: IndexedSeq[Double] = Vector(2.4384426214120847,
//  13.158255368675546, -1.314018699127919, ...
val eps = Gaussian(0.0,1.0).sample(10000)
// eps: IndexedSeq[Double] = Vector(1.427893572186197,
//  -0.6504024834850954, -0.30314296354173065, ...
val y = (lp zip eps) map (p => p._1 + p._2)
// y: IndexedSeq[Double] = Vector(3.866336193598282,
//  12.50785288519045, -1.6171616626696497, ...
val yx = (y zip xx) map (p => (p._1,p._2._1,p._2._2))
// yx: IndexedSeq[(Double, Double, Double)] = Vector(
// (3.866336193598282,0.17848673281799982,0.5814691557760885),
// (12.50785288519045,5.270572111051524,1.1171111465724972),
// ...

val rddLR = sc.parallelize(yx)
// rddLR: RDD[(Double, Double, Double)] =
//    ParallelCollectionRDD[27]
```

Note that the last line converts the regular Scala Breeze collection into a Spark RDD using **parallelize**. We could, in principle, do regression modelling using raw RDDs, and early versions of Spark required this. However, statisticians used to statistical languages such as R know that data frames are useful for working with tabular data. We can convert an RDD of tuples to a Spark **DataFrame** as follows:

```
val dfLR = rddLR.toDF("y","x1","x2")
// dfLR: org.apache.spark.sql.DataFrame = [y: double, x1: double ... 1 more field]
dfLR.show
// +--------------------+--------------------+--------------------+
// |                   y|                  x1|
x2|
// +--------------------+--------------------+--------------------+
// |   3.866336193598282|  0.1784867328179982|   0.581469155776088 5|
// |   12.50785288519045|   5.270572111051524|   1.117111465724972|
// |  -1.617161662669649 7|  -1.552950597527563 5|  0.291882495927207 83|
// |  0.868345187137751 1|  0.115062967960763 87|   -0.29705322884738 9|
// |  1.332068075665698 8| -0.293724356498183 96| -0.03601773761744...|
// |   8.408493708252392|    1.88109002482217|   1.195030171395341 5|
// |  1.567251700231130 8|  0.712859802056508 2| -0.486655143573601 96|
// |   3.168738194841887|  2.034073979064973 3|   -0.66011311490554 77|
// |   8.039748451446279|  3.865367416271596 7|   -1.231616957732694 3|
// |  10.629872301438459|    4.019661153356504|   0.768792224085477 6|
// |   2.962051294451586|  1.175205784166855 3|   -0.681867725559664 2|
// |  1.283325187221878 5|  -1.506565545411926 9|   0.748193509034269 1|
// |  2.409137039945633 8| 0.321612136239801 45|   -0.600933640294529 6|
// | -3.007647488126058 6|  -1.799053079035363 8| -0.321403075091690 86|
// |    8.80509036070586|   4.577527777964653| -0.466447324553018 74|
// |   0.705669160914749| -0.243740191691102 17|    0.35953682875035 5|
// |   6.357141836034998|  2.306495747470165 3|   0.383281588216774 2|
// | -0.711107549605103 1|  -1.560690946502622 2|   0.262222232753217 1|
// |  4.958346748104724 5|  1.842123547352094 3|  0.129907874539189 62|
// |  5.333195695423528 4|  2.144204188345498 5| -6.86911622420159...|
// +--------------------+--------------------+--------------------+
// only showing top 20 rows
dfLR.show(5)
// +--------------------+--------------------+--------------------+
// |                   y|                  x1|
x2|
// +--------------------+--------------------+--------------------+
// |   3.866336193598282|  0.1784867328179982|   0.581469155776088 5|
// |   12.50785288519045|   5.270572111051524|   1.117111465724972|
// |  -1.617161662669649 7|  -1.552950597527563 5|  0.291882495927207 83|
// |  0.868345187137751 1|  0.115062967960763 87|   -0.29705322884738 9|
// |  1.332068075665698 8| -0.293724356498183 96| -0.03601773761744...|
// +--------------------+--------------------+--------------------+
// only showing top 5 rows
```

Note that **show** shows the first few rows of a **DataFrame**, and giving it a numeric argument specifies the number to show. This is very useful for quick sanity-checking of **DataFrame** contents.

Note that there are other ways of getting data into a Spark **DataFrame**. One of the simplest ways to get data into Spark from other systems is via a CSV file. A properly formatted CSV file with a header row can be read into Spark with a command like:

```
// Don't run unless you have an appropriate CSV file...
val df = spark.read.
  option("header","true").
  option("inferSchema","true").
  csv("myCsvFile.csv")
```

This requires two passes over the data — one to infer the schema and one to actually read the data. The result is a Spark `DataFrame`. For very large datasets it is better to declare the schema and not use automatic schema inference. However, for very large datasets, CSV probably isn't a great choice of format anyway. Spark supports many more efficient data storage formats. Note that Spark also has functions for querying SQL (and other) databases, and reading query results directly into `DataFrame` objects. For people familiar with databases, this is often the most convenient way of ingesting data into Spark. See the Spark DataFrames guide and the API docs for DataFrameReader for further information.

Spark has an extensive library of tools for the development of sophisticated machine learning pipelines. Included in this are functions for fitting linear regression models, regularised regression models (Lasso, ridge, elastic net), generalised linear models, including logistic regression models, etc., and tools for optimising regularisation parameters, for example, using cross-validation. Let's start by seeing how to fit a simple OLS linear regression model: see the ML pipeline documentation for further information, especially the docs on classification and regression.

We start by creating an object for fitting linear regression models:

```
import org.apache.spark.ml.regression.LinearRegression
// import org.apache.spark.ml.regression.LinearRegression
import org.apache.spark.ml.linalg._
// import org.apache.spark.ml.linalg._

val lm = new LinearRegression
// lm: LinearRegression = linReg_830e2ec56b44
lm.getStandardization
// res14: Boolean = true
lm.setStandardization(false)
// res15: lm.type = linReg_830e2ec56b44
lm.getStandardization
// res16: Boolean = false
lm.explainParams
// res17: String =
// aggregationDepth: suggested depth for treeAggregate (>= 2) (
// elasticNetParam: the ElasticNet mixing parameter, in range [0
//   For alpha = 0, the penalty is an L2 penalty.
//   For alpha = 1, it is an L1 penalty (default: 0.0)
// featuresCol: features column name (default: features)
// fitIntercept: whether to fit an intercept term (default: true
// labelCol: label column name (default: label)
// maxIter: maximum number of iterations (>= 0) (default: 100)
// predictionCol: prediction column name (default: prediction)
// regParam: regularization parameter (>= 0) (default: 0.0)
// solver: the solver algorithm for optimization.
//   If this is not set or empty, default value is 'auto' (default: auto)
// standardization: whether to standardize the training features ...
```

Note that there are many parameters associated with the fitting algorithm, including regularisation parameters. These are set to defaults corresponding to no regularisation (simple OLS). Note, however, that the algorithm defaults to standardising covariates to be mean zero variance one. We can turn that off before fitting the model if desired.

Note also that `fitIntercept` defaults to `true`, and so you do not need to prepend your design matrix with a column of ones.

Also note that the model fitting algorithm assumes that the **`DataFrame`** to be fit has (at least) two columns, one called **`label`** containing the response variable, and one called **`features`**, where each element is actually a **`Vectors`** of covariates. So we first need to transform our **`DataFrame`** into the required format.

In Spark, `Vectors` is just a type alias for a Breeze `Vector`.

```
val dflr = (dfLR map {row => (row.getDouble(0),
  Vectors.dense(row.getDouble(1),
  row.getDouble(2)))}).toDF("label","features")
// dflr: DataFrame = [label: double, features: vector]
dflr.show(5)
// +--------------------+--------------------+
// |               label|            features|
// +--------------------+--------------------+
// |   3.866336193598282|[0.17848673281799...|
// |   12.50785288519045|[5.27057211105152...|
// |-1.6171616626696497|[-1.5529505975275...|
// |  0.8683451871377511|[0.11506296796076...|
// |  1.3320680756656988|[-0.2937243564981...|
// +--------------------+--------------------+
// only showing top 5 rows
```

Now we have the data in the correct format, it is simple to fit the model and look at the estimated parameters.

```
val fit = lm.fit(dflr)
// fit: LinearRegressionModel = linReg_830e2ec56b44
fit.intercept
// res19: Double = 1.4799669653918834
fit.coefficients
// res20: Vector = [2.004976921569354,1.0004609409395846]
```

You should see that the estimated parameters are close to the "true" parameters that were used to simulate from the model. More detailed diagnostics can be obtained from the fitted summary object.

```
val summ = fit.summary
// summ: LinearRegressionTrainingSummary =
//   LinearRegressionTrainingSummary@35e2c4b
summ.r2
// res21: Double = 0.9451196184400352
summ.rootMeanSquaredError
// res22: Double = 1.0061563554694304
summ.coefficientStandardErrors
// res23: Array[Double] = Array(0.004989649710643566,
//  0.010061968865893115, 0.011187314859319352)
summ.pValues
// res24: Array[Double] = Array(0.0, 0.0, 0.0)
summ.tValues
```

```
// res25: Array[Double] = Array(401.8271898511191,
//   99.42993804431556, 132.28973922719524)
summ.predictions
// res26: DataFrame = [label: double, features: vector
summ.residuals
// res27: DataFrame = [residuals: double]
```

So, that's how to fit a simple OLS linear regression model.

8.4 Logistic regression

To illustrate the fitting of logistic regression models, we will again use synthetic data, and start from exactly the same linear predictor as we used in the linear regression analysis. But this time we will push it through a sigmoid and simulate some binary outcomes.

```
val p = lp map (x => 1.0/(1.0+math.exp(-x)))
// p: IndexedSeq[Double] = Vector(0.9994499079755785,
// 0.49379319998757 72, 0.999720196336271, ...
val yl = p map (pi => new Binomial(1,pi).draw) map
  (_.toDouble)
// yl: IndexedSeq[Double] = Vector(1.0, 1.0, 1.0, ...
val yxl = (yl zip xx) map (p => (p._1,p._2._1,p._2._2))
// yxl: IndexedSeq[(Double, Double, Double)] = Vector(
//   (1.0,3.1381503063526694,-0.27142587948210634),
//   (1.0,0.2902050889768183,-2.1052386533907854), ...

val rddLogR = sc.parallelize(yxl)
// rddLogR: RDD[(Double, Double, Double)] =
//    ParallelCollectionRDD[59]
val dfLogR = rddLogR.toDF("y","x1","x2").persist
// dfLogR: Dataset[org.apache.spark.sql.Row] =
//    [y: double, x1: double ... 1 more field]
dfLogR.show(5)
// +---+------------------+--------------------+
// |  y|                x1|                  x2|
// +---+------------------+--------------------+
// |1.0|3.1381503063526694|-0.27142587948210634|
// |1.0|0.2902050889768183| -2.1052386533907854|
// |1.0|3.2133859767485955|  0.2543706054055738|
// |1.0|0.8247934159943499| -1.0013955800392003|
// |1.0| 3.274443477406557| -1.6514890824757613|
// +---+------------------+--------------------+
// only showing top 5 rows
```

Now that we have our synthetic logistic regression data, we can fit it in a fairly similar way to simple linear regression.

```
import org.apache.spark.ml.classification._
// import org.apache.spark.ml.classification._
val lr = new LogisticRegression
// lr: LogisticRegression = logreg_09792f5b38dd
lr.setStandardization(false)
// res29: lr.type = logreg_09792f5b38dd
lr.explainParams
// res30: String =
// aggregationDepth: suggested depth for treeAggregate (>= 2) (
// elasticNetParam: the ElasticNet mixing parameter, in range [0, 1].
//  For alpha = 0, the penalty is an L2 penalty.
//  For alpha = 1, it is an L1 penalty (default: 0.0)
// family: The name of family which is a description of the
//  label distribution to be used in the model.
//  Supported options: auto, binomial, multinomial. (default: auto)
// featuresCol: features column name (default: features)
// fitIntercept: whether to fit an intercept term (default: true)
// labelCol: label column name (default: label)
// maxIter: maximum number of iterations (>= 0) (default: 100)
// predictionCol: prediction column name (default: prediction)
// probabilityCol: Column name for predicted class conditional probabilities.

val dflogr = (dfLogR map {row => (row.getDouble(0),
  Vectors.dense(row.getDouble(1),
  row.getDouble(2)))}).toDF("label","features")
// dflogr: DataFrame = [label: double, features: vector]
dflogr.show(5)
// +-----+--------------------+
// |label|            features|
// +-----+--------------------+
// |  1.0|[3.13815030635266...|
// |  1.0|[0.29020508897681...|
// |  1.0|[3.21338597674859...|
// |  1.0|[0.82479341599434...|
// |  1.0|[3.27444347740655...|
// +-----+--------------------+
// only showing top 5 rows

val logrfit = lr.fit(dflogr)
// logrfit: LogisticRegressionModel = logreg_09792f5b38dd
logrfit.intercept
// res32: Double = 1.482650505103195
logrfit.coefficients
// res33: org.apache.spark.ml.linalg.Vector =
//   [2.003478489528172,0.9368494184176968]
```

So we see that we are able to recover the regression coefficients that we used to simulate the data fairly well. Additional diagnostics can be obtained, similar to the linear regression case.

8.5 Tuning regularisation parameters using cross-validation

We have seen that there are many possible tuning constants associated with linear and logistic regression models in Spark. In particular, the models allow regularisation using elastic net penalties (which includes ridge and lasso regression as special cases). But then there is a question as to how to set the tuning parameters. One way is to search over a grid of possible tuning parameters and see which give the best predictive performance under k–fold cross–validation. The example below illustrates this for a ridge penalty in our logistic regression example, searching over a grid of 60 different ridge parameters and evaluating with 8–fold cross–validation.

```
import breeze.linalg.linspace
// import breeze.linalg.linspace
val lambdas = linspace(-12,4,60).
  toArray.
  map{math.exp(_)}
// lambdas: Array[Double] = Array(6.14421235332821E-6,
//  8.058254732499574E-6, 1.0568558767126194E-5, ...
import org.apache.spark.ml.tuning._
// import org.apache.spark.ml.tuning._
import org.apache.spark.ml.evaluation._
// import org.apache.spark.ml.evaluation._
val paramGrid = new ParamGridBuilder().
  addGrid(lr.regParam,lambdas).
  build()
// paramGrid: Array[org.apache.spark.ml.param.ParamMap] =
// Array({
//      logreg_09792f5b38dd-regParam: 6.14421235332821E-6
// }, {
//      logreg_09792f5b38dd-regParam: 8.058254732499574E-6
//  ...
val cv = new CrossValidator().
  setEstimator(lr).
  setEvaluator(new BinaryClassificationEvaluator).
  setEstimatorParamMaps(paramGrid).
  setNumFolds(8)
// cv: CrossValidator = cv_6d06cc600072
val cvMod = cv.fit(dflogr)
// cvMod: CrossValidatorModel = cv_6d06cc600072
cvMod.explainParams
// res34: String =
// estimator: estimator for selection (current: logreg_09792f5b3
// estimatorParamMaps: param maps for the estimator (current: [
// evaluator: evaluator used to select hyper-parameters that ma
// numFolds: number of folds for cross validation (>= 2) (defau
// seed: random seed (default: -1191137437)
cvMod.bestModel.explainParams
// res35: String =
// aggregationDepth: suggested depth for treeAggregate (>= 2) (
// elasticNetParam: the ElasticNet mixing parameter, in range [0
// family: The name of family which is a description of the labe
// featuresCol: features column name (default: features)
// fitIntercept: whether to fit an intercept term (default: true
// labelCol: label column name (default: label)
// maxIter: maximum number of iterations (>= 0) (default: 100)
// predictionCol: prediction column name (default: prediction)
// probabilityCol: Column name for predicted class conditional
cvMod.bestModel.extractParamMap
// res36: org.apache.spark.ml.param.ParamMap =
// {
//      logreg_09792f5b38dd-aggregationDepth: 2,
//      logreg_09792f5b38dd-elasticNetParam: 0.0,
//      logreg_09792f5b38dd-family: auto,
//      logreg_09792f5b38dd-featuresCol: features,
//      logreg_09792f5b38dd-fitIntercept: true,
//      logreg_09792f5b38dd-labelCol: label,
//      logreg_09792f5b38dd-maxIter: 100,
//      logreg_09792f5b38dd-predictionCol: prediction,
//       logreg_09792f5b38dd-probabilityCol: probability,
//       logreg_09792f5b38dd-rawPredictionCol: rawPrediction,
//       logreg_09792f5b38dd-regParam: 2.737241171E-4,
//       logreg_09792f5b38dd-standardization: false,
//       logreg_09792f5b38dd-threshold: 0.5,
//       logreg_09792f5b38dd-tol: 1.0E-6
// }
val lambda = cvMod.bestModel.
  extractParamMap.
  getOrElse(cvMod.bestModel.
    getParam("regParam"),0.0).
  asInstanceOf[Double]
// lambda: Double = 2.737241171E-4
```

The most important output is the optimal regularisation parameter, which for this dataset was around 2.74×10^{-4}. Generally speaking it is better to standardise the predictors for tuning and fitting elastic net models (since you want the shrinkage to be on the same scale for each variable). It also is important to fit the intercept as we have done here, since you don't want to shrink the intercept along with the other regression coefficients.

These big grid–search jobs can take a long time to run.

8.6 Compiling Spark jobs

Working interactively in the Spark Shell can be very convenient for learning about how Spark works, similarly to how it can be convenient to explore Scala by using the Scala REPL interactively. But for significant work, there are many advantages to writing code in an intelligent IDE and compiling using SBT. Exactly the same is true for Spark, so once you have the idea of how it works, you will want to start developing Spark code as you do other Scala applications, using the same tool chain, and regarding Spark to some extent as just another third–party Scala library.

A complete example of how to set up an SBT project for building Spark applications is provided in the course repo. But essentially, the SBT build file should look something like:

```
name := "spark-template"

version := "0.1"

libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "2.1.0" % Provided,
  "org.apache.spark" %% "spark-sql" % "2.1.0" % Provided
)

scalaVersion := "2.11.8"
```

Note that Spark 2.1.0 is not compatible with Scala 2.12. Also note that if you are using MLLib, then you will need an additional dependence on "spark-mllib". The following is a complete example application:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SparkApp {

  def main(args: Array[String]): Unit = {

    val spark = new SparkConf().
      setAppName("Spark␣Application")
    val sc = new SparkContext(spark)

    sc.textFile("/usr/share/dict/words").
      map(_.trim).
      map(_.toLowerCase).
      flatMap(_.toCharArray).
      filter(_ > '/').
      filter(_ < '}').
      map{(_,1)}.
      reduceByKey(_+_).
      sortBy(_._2,false).
      collect.
      foreach(println)

  }

}
```

So, you just need a few imports, then you must start by creating a `SparkConf` and giving your application an appropriate name, and then create the `SparkContext`, `sc`. This Spark context can then be used in the same way as the one that is created for you when you start up a Spark shell.

You can use SBT to build your application with `sbt package`. This will place the compiled JAR in `target/scala-2.11/`. This JAR can be submitted to a Spark cluster using `spark-submit`. For example, this application could be submitted to the local standalone Spark cluster with a command like:

```
spark-submit --class "SparkApp" \
--master local[4] \
target/scala-2.11/spark-template_2.11-0.1.jar
```

Note that you can include additional third party libraries in your build in the usual way, but in this case you will need to build and submit an assembly JAR to the Spark cluster.

8.6.1 Further reading

As previously mentioned, once you are up and running with a Spark shell, the official Spark documentation is reasonably good. First go through the quick start guide, then the programming guide, then the ML guide, and finally, consult the API docs.

Chapter 9

Advanced topics

9.1 Typeclasses and implicits

Scala supports an advanced programming feature called “implicits”, which is essentially a mechanism for automatically (implicitly) providing arguments for functions without requiring the function caller to pass the value explicitly. The value used must be be of the correct type, and must have already been declared and “in scope” (exactly how implicits are resolved is non-trivial, and beyond the scope of this course). There are many possible applications of implicits, but in Scala they are most often used as the mechanism by which the “typeclass” pattern can be implemented. Typeclasses are a powerful device for implementing “ad hoc polymorphism” popularised by Haskell. They are a (arguably superior) alternative to traditional inheritance based O–O approaches to polymorphism. Unlike Haskell, Scala does not have first class language support for typeclasses, but the available implicit mechanisms are powerful enough to support the pattern indirectly. This probably all sounds like terrifying CS theory. In fact it is all very practical and useful, but is most easily understood by looking at a few examples.

9.1.1 Using implicits to add a method to a pre-existing class

Scala’s `.sum` method magically works on any Scala collection parametrised by a numeric type.

```
Vector(1,2,3).sum
// res0: Int = 6
List(1.0,5.0).sum
// res1: Double = 6.0
```

This is great, but there isn’t a corresponding `.mean` method.

```
Vector(1,2,3).mean
// <console>:8: error: value mean is not a member of Vector[Int]
//               Vector(1,2,3).mean
//                             ^
```

Is it possible to add one? The answer is “yes”, but before diving in to the details, think about how difficult this would be to do cleanly in a typical O–O language like Java. You can't go in to the Scala source code for the standard library and retrospectively include a `.mean` method. You would probably be forced to define a new abstract class (or interface) which does implement a `.mean` method, and then define new classes inheriting from the old collections that also implement the new interface, and then you'd be forced to work with the new subclass wherever you needed the `.mean` methods. So there'd be lots of converting and casting back and forth between the original and new classes, and it would all be very unsatisfactory and not very extensible or inter-operable...

We have already seen (in Chapter 3) how to define a mean function which accepts a numeric collection as input and returns a mean as output. Let's define this again now, but wrap it in an object to keep it out of scope.

```
object Meanable {
  def mean[T: Numeric](it: Iterable[T]): Double =
    it.map(implicitly[Numeric[T]].toDouble(_)).
      sum / it.size
}
```

We will now try to better understand how this works and how to go one step further and define a `.mean` method (on the `Iterable` collection). The *context bound* on the type parameter `T` (the `:` notation) is just syntactic sugar for the addition of an implicit parameter. So the above could have been written equivalently as:

```
object Meanable {
  def mean[T](it: Iterable[T])(
    implicit num: Numeric[T]): Double =
    it.map(num.toDouble(_)).sum / it.size
}
```

Recall that we could implement this a bit more cleanly using *Spire*, but we will stick to vanilla Scala here.

So when the `mean` function is called, there will be a search (at compile time) for an implicit value of type `Numeric[T]`. If such a value can be found, it will be passed in implicitly, and made available in the body of the function as `num`. In the first version of the code, we don't bind a name to the implicit value, but we can refer to it using `implicitly`. For every numeric type `T`, a function `.toDouble()` is defined, and so the code will compile. We can use this as follows:

```
import Meanable._
// import Meanable._
mean(Vector(1,2,3))
// res3: Double = 2.0
mean(List(1.0,5.0))
// res4: Double = 3.0
```

So far so good, but we have already seen how to do this. What about adding a method? For this, we need to use an *implicit class*:

```
implicit class MeanableInstance[T: Numeric](
    it: Iterable[T]) {
  def mean[T] = Meanable.mean(it)
}
```

This says that any instance of an iterable class over a numeric type should be regarded as an instance of the class `MeanableInstance`.

But `MeanableClass` has a `.mean` method defined in terms of the `Meanable.mean` function we have already defined. We can then use this as follows:

```
Vector(1,2,3).mean
// res5: Double = 2.0
List(1.0,3.0,5.0,7.0).mean
// res6: Double = 4.0
```

So this shows that implicits can do powerful things, but in this example we've somewhat skirted around the typeclass issue, so it's worth walking through another example, this time explicitly considering the typeclass.

9.1.2 Defining and using a simple typeclass

In statistical computing we quite often want to output data in CSV format. Suppose, for example, that we have an MCMC algorithm with multivariate state, and that we want to serialise the MCMC iterations as rows in a CSV file. It would be useful to define a `.toCsv` method on the state which serialises the state as a `String` representing one row of a CSV file. If we always define our state type ourselves, we could ensure that our state always included a `.toCsv` method. But what if we sometimes like to use a `Vector[Double]` as our state (or a Breeze `DenseVector[Double]`)? We are then back to the same problem as we just considered. A good way to deal with this kind of problem is using typeclasses. First, define the typeclass itself.

```
trait CsvRow[T] {
 def toCsv(row: T): String
}
```

This is our typeclass. Superficially, this is a parametrised type with a `toCsv` function, `T =>String`. But as a typeclass, we interpret this as meaning that the type `T` belongs to the `CsvRow` typeclass if it has a `toCsv` method. As before, we have the issue that in the typeclass we have a method which accepts a value of type `T` and returns a `String`, but we actually just want a method on `T` which returns a `String`. We solve this using an implicit class, as before.

```
implicit class CsvRowSyntax[T](row: T) {
  def toCsv(implicit inst: CsvRow[T]) = inst.toCsv(row)
}
```

Now that we have our typeclass and its associated syntax, we can define functions that are generic over any type `T` belonging to our typeclass.

```
def printRows[T: CsvRow](it: Iterable[T]): Unit =
  it.foreach(row => println(row.toCsv))
```

The context bound notation often leads to more readable code. Here we just read this as the type `T` belonging to the `CsvRow` typeclass, and can ignore the fact that this is actually shorthand for the addition of an implicit parameter. Let's see how we can define instances of this typeclass. First let's suppose that we have our own state type, `MyState`.

```
case class MyState(x: Int, y: Double)
```

This doesn't currently have a `.toCsv` method, but we can declare one at the same time as declaring it to be an instance of the `CsvRow` typeclass.

```
implicit val myStateCsvRow = new CsvRow[MyState] {
  def toCsv(row: MyState) = row.x.toString+","+row.y
}
```

Now we have declared it to be a member of the typeclass, we can use it as follows.

```
MyState(1,2.0).toCsv
// res7: String = 1,2.0
printRows(List(MyState(1,2.0),MyState(2,3.0)))
// 1,2.0
// 2,3.0
```

Although we can do this, we could have just defined a `.toCsv` method directly on our `MyState` class, which obviously would have been a bit easier. But we can't easily add methods to classes in the Scala standard library. So now suppose that we would like to use a `Vector[Double]` to represent state. We can just as easily declare a typeclass instance for this, and use it as follows.

```
implicit val vectorDoubleCsvRow =
    new CsvRow[Vector[Double]] {
  def toCsv(row: Vector[Double]) = row.mkString(",")
}
// vectorDoubleCsvRow: CsvRow[Vector[Double]] =
//   $anon$1@4604e051

Vector(1.0,2.0,3.0).toCsv
// res9: String = 1.0,2.0,3.0
printRows(List(Vector(1.0,2.0),Vector(4.0,5.0),
  Vector(3.0,3.0)))
// 1.0,2.0
// 4.0,5.0
// 3.0,3.0
```

So typeclasses provide a nice basis for the development of flexible, generic code.

9.1.3 A parametrised typeclass, using higher–kinded types

We have just seen an example of how to declare a `Vector[Double]` as belonging to a typeclass. But what if we have an example where we want to declare `Vector[T]` as belonging to a typeclass irrespective of the type parameter `T`. In other words, we want to declare a typeclass over a parametrised type. Since typeclasses are by definition parametrised types, we will need to declare a generic type with a type parameter corresponding to a generic type. It would be impossible to do this in a language like Java, but in Scala we can do it, using another advanced language feature: higher–kinded types. Higher–kinded types are generic types with type parameters that are also generic types. Very few languages support them: Scala and Haskell are the best known languages which do. Again, this probably sounds like advanced CS theory, but again, examples reveal that they are actually very useful in practice.

To illustrate this problem, we will reconsider the problem of “thinning” a `Stream[T]`. We saw previously how to write a function that would thin the `Stream`, irrespective of the type parameter `T`, but this was a function taking a `Stream[T]` as input and returning another `Stream[T]` as an output. We did not define it as a new method on the existing `Stream` class. We will define a typeclass, `Thinnable`, representing parametrised classes which can be thinned. We do this as follows.

```scala
import scala.language.higherKinds
trait Thinnable[F[_]] {
  def thin[T](f: F[T], th: Int): F[T]
}
```

Note that we need to explicitly import `higherKinds` in order to prevent compiler warnings. In declaring the trait we use an underscore to indicate that `F` is generic, but use a type parameter `T` in the declaration of `thin`, to indicate that the type of the input and output are exactly the same. As usual, we can declare method syntax using an implicit class.

```scala
implicit class ThinnableSyntax[T,F[T]](value: F[T]) {
  def thin(th: Int)(implicit inst: Thinnable[F]): F[T] =
    inst.thin(value,th)
}
```

Finally, we can declare an instance of our `Thinnable` typeclass for `Stream`.

```scala
implicit val streamThinnable: Thinnable[Stream] =
    new Thinnable[Stream] {
  def thin[T](s: Stream[T],th: Int): Stream[T] = {
  val ss = s.drop(th-1)
  if (ss.isEmpty) Stream.empty else
    ss.head #:: thin(ss.tail, th)
  }
}
```

Now that evidence has been provided that `Stream` can be regarded as belonging to the `Thinnable` typeclass, we can use our new `.thin` method syntax in an intuitive way.

```scala
Stream.iterate(0)(_ + 1).
  drop(10).
  thin(2).
  take(5).
  toArray
// res11: Array[Int] = Array(11, 13, 15, 17, 19)
```

Typeclasses are a powerful framework for developing flexible generic code. But using them in Scala via implicits involves a little bit of “boiler–plate” code. In particular, the definition of the implicit class providing method syntax for the typeclass seems as though it https://github.com/mpilquist/simulacrum

ought to be possible to automatically generate. In fact it is possible, and this functionality is provided by the library *Simulacrum*. This library makes typeclasses more of a first class concept in Scala, and eliminates quite a bit of code. Once you understand typeclasses and implicits, I recommend learning about how to use Simulacrum to make working with typeclasses simpler and more consistent. We don't have time to cover it in detail within this course, but the end–of–chapter exercises will get you started.

9.1.4 Case study: A scalable particle filter in Scala

Introduction

Many modern algorithms in computational Bayesian statistics have at their heart a particle filter or some other sequential Monte Carlo (SMC) procedure. In particular, particle MCMC algorithms use a particle filter in the inner-loop in order to compute a (noisy, unbiased) estimate of the marginal likelihood of the data. These algorithms are often highly computationally intensive, either because the forward model used to propagate the particles is expensive, or because the likelihood associated with each particle/observation is expensive (or both). In this case it is desirable to parallelise the particle filter to run on all available cores of a machine, or in some cases, it would even be desirable to distribute the the particle filter computation across a cluster of machines. Parallelisation is difficult when using the conventional imperative programming languages typically used in scientific and statistical computing, but is much easier using modern functional languages such as Scala. In Scala it is possible to describe algorithms at a higher level of abstraction, so that exactly the same algorithm can run in serial, run in parallel across all available cores on a single machine, or run in parallel across a cluster of machines, all without changing any code. Doing so renders parallelisation a non-issue. Here I'll walk through how to do this for a simple bootstrap particle filter, but the same principle applies for a large range of statistical computing algorithms.

Typeclasses and monadic collections

Many computational tasks in statistics can be accomplished using a sequence of operations on monadic collections. We would like to write code that is independent of any particular implementation of a monadic collection, so that we can switch to a different implementation without changing the code of our algorithm (for example, switching from a serial to a parallel collection). But in strongly typed languages we need to know at compile time that the collection we use has the methods that we require. Typeclasses provide a nice solution to this problem. We can describe a simple typeclass for our monadic collection as follows:

```scala
trait GenericColl[C[_]] {
  def map[A, B](ca: C[A])(f: A => B): C[B]
  def reduce[A](ca: C[A])(f: (A, A) => A): A
  def flatMap[A, B, D[B] <: GenTraversable[B]](
    ca: C[A])(f: A => D[B]): C[B]
  def zip[A, B](ca: C[A])(cb: C[B]): C[(A, B)]
  def length[A](ca: C[A]): Int
}
```

In the typeclass we just list the methods that we expect our generic collection to provide, but do not say anything about how they are implemented. For example, we know that operations such as `map` and `reduce` can be executed in parallel, but this is a separate concern. We can now write code that can be used for any collection conforming to the requirements of this typeclass. The full code for this example is provided in the associated github repo, and includes the obvious syntax for this typeclass, and typeclass instances for the Scala collections `Vector` and `ParVector`, that we will exploit later in the example.

SIR step for a bootstrap filter

We can now write some code for a single observation update of a bootstrap particle filter.

```
def update[S: State, O: Observation, C[_]: GenericColl](
    dataLik: (S, O) => LogLik, stepFun: S => S
  )(x: C[S], o: O): (LogLik, C[S]) = {
    import breeze.stats.distributions.Poisson
    val xp = x map (stepFun(_))
    val lw = xp map (dataLik(_, o))
    val max = lw reduce (math.max(_, _))
    val rw = lw map (lwi => math.exp(lwi - max))
    val srw = rw reduce (_ + _)
    val l = rw.length
    val z = rw zip xp
    val rx = z flatMap { case (rwi, xpi) =>
      Vector.fill(Poisson(rwi * l / srw).draw)(xpi) }
    (max + math.log(srw / l), rx)
}
```

This is a very simple bootstrap filter, using Poisson resampling for simplicity and data locality, but does include use of the log-sum-exp trick to prevent over/underflow of raw weight calculations, and tracks the marginal (log-)likelihood of the observation. With this function we can now pass in a "prior" particle distribution in any collection conforming to our typeclass, together with a propagator function, an observation (log-)likelihood, and an observation, and it will return back a new collection of particles of exactly the same type that was provided for input. Note that all of the operations we require can be accomplished with the standard monadic collection operations declared in our typeclass.

Filtering as a functional fold

Once we have a function for executing one step of a particle filter, we can produce a function for particle filtering as a functional fold over a sequence of observations:

```
def pFilter[S: State, O: Observation,
  C[_]: GenericColl, D[O] <: GenTraversable[O]](
  x0: C[S], data: D[O],
  dataLik: (S, O) => LogLik, stepFun: S => S
  ): (LogLik, C[S]) = {
    val updater = update[S, O, C](dataLik, stepFun) _
    data.foldLeft((0.0, x0))((prev, o) => {
      val (oll, ox) = prev
      val (ll, x) = updater(ox, o)
      (oll + ll, x)
    })
}
```

Folding data structures is a fundamental concept in functional programming, and is exactly what is required for any kind of filtering problem.

Marginal likelihoods and parameter estimation

So far we haven't said anything about parameters or parameter estimation, but this is appropriate, since parametrisation is a separate concern from filtering. However, once we have a function for particle filtering, we can produce a function concerned with evaluating marginal likelihoods trivially:

```
def pfMll[S: State, P: Parameter, O: Observation,
    C[_]: GenericColl, D[O] <: GenTraversable[O]](
  simX0: P => C[S], stepFun: P => S => S,
  dataLik: P => (S, O) => LogLik, data: D[O]
): (P => LogLik) = (th: P) =>
    pFilter(simX0(th), data, dataLik(th), stepFun(th))._1
```

Note that this higher-order function does not return a value, but instead a function which will accept a parameter as input and return a (log-)likelihood as output. This can then be used for parameter estimation purposes, perhaps being used in a PMMH pMCMC algorithm, or something else. Again, this is a separate concern.

Example

Here I'll just give a completely trivial toy example, purely to show how the functions work. Here we will just look at inferring the auto-regression parameter of a linear Gaussian AR(1)-plus-noise model using the functions we have developed. First we can simulate some synthetic data from this model, using a value of 0.8 for the auto-regression parameter:

```
val inNoise = Gaussian(0.0, 1.0).sample(99)
val state = DenseVector(inNoise.scanLeft(0.0)(
  (s, i) => 0.8 * s + i).toArray)
val noise = DenseVector(
  Gaussian(0.0, 2.0).sample(100).toArray)
val data = (state + noise).toArray.toList
```

Now assuming that we don't know the auto-regression parameter, we can construct a function to evaluate the likelihood of different parameter values as follows:

```
val mll = pfMll(
  (th: Double) => Gaussian(0.0, 10.0).
    sample(10000).toVector.par,
  (th: Double) => (s: Double) =>
    Gaussian(th * s, 1.0).draw,
  (th: Double) => (s: Double, o: Double) =>
    Gaussian(s, 2.0).logPdf(o),
  data
)
```

Note that the 4 characters `.par` at the end of the third line are the only difference between running this code serially or in parallel! Now we can run this code by calling the returned function with different values. So, hopefully `mll(0.8)` will return a larger log-likelihood than (say) `mll(0.6)` or `mll(0.9)`. The example code in the GitHub repo plots the results of calling `mll()` for a range of values. In this particular example, both the forward model and the likelihood are very cheap operations, so there is little to be gained from parallelisation. Nevertheless, I still get a speedup of more than a factor of two using the parallel version on my laptop.

Note that if that was the genuine use-case, then it would be much better to parallelise the parameter range than the particle filter, due to providing better parallelisation granularity, but many other examples require parallelisation of the particle filter itself.

Conclusion

This short case study has illustrated how typeclasses can be used in Scala to write code that is parallelisation–agnostic. Code written in this way can be run on one or many cores as desired. We've illustrated the concept with a scalable particle filter, but nothing about the approach is specific to that application. It would be easy to build up a library of statistical routines this way, all of which can effectively exploit available parallel hardware. Further, although we haven't demonstrated it here, it is straightforward to extend this idea to allow code to be distributed over a cluster of parallel machines if necessary. For example, if an Apache Spark cluster is available, it is possible to define a Spark RDD instance for our generic collection typeclass, that will then allow us to run our (unmodified) particle filter code over a Spark cluster. This also highlights the fact that Spark can be useful for distributing computation as well as just processing "big data".

9.2 Typeclasses from category theory

As we have seen frequently throughout this course, there are some typeclasses that crop up over–and–over in functional programming: types which can be *combined*, types which can be *mapped* over, and types which can be *flat-mapped* over. These types also crop up in the mathematical discipline of category theory, and so they usually go by the names of *monoid*, *functor* and *monad*. Although it is somewhat helpful to know a little category theory for advanced FP, really these concepts boil down to very simple typeclasses.

9.2.1 Typeclasses for Monoid, Functor and Monad

- In Scala, we can define typeclasses for `Monoid`, `Functor` and `Monad` (using parametric and higher-kinded types):

```
trait Monoid[A] {
  def combine(a1: A, a2: A): A
  def id: A
}

trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B): F[B]
}

trait Monad[M[_]] extends Functor[M] {
  def unit[A](a: A): M[A]
  def flatMap[A,B](ma: M[A])(f: A => M[B]): M[B]
}
```

Note that since `Monad` extends `Functor`, all monads are also functors, but the converse is not always true. We could easily define and work with these typeclasses ourselves, but they are so ubiquitous in FP that it makes sense to have them provided by a library. This is the purpose of Cats.

9.2.2 FP with Cats

http://typelevel.org/cats/

Cats is a Scala library providing typeclasses and data types from category theory. In addition to providing the typeclasses themselves, Cats also provides instances and convenient syntax for a large number of classes and types in the Scala standard library. To use Cats with SBT projects, just add

```
libraryDependencies += "org.typelevel" %% "cats" % "0.9.0"
```

to the build file.

Monoids

We don't have time to explore Cats in detail now, but an example using monoids should illustrate some of the potential. Note that monoids represent types which can be combined, and Cats provides the syntax `|+|` as an infix operator for monoid values. From a Scala REPL with a Cats dependency, we first import the monoid typeclass, the syntax and some instances.

```
import cats.Monoid
// import cats.Monoid
import cats.syntax.semigroup._
// import cats.syntax.semigroup._
import cats.instances.all._
// import cats.instances.all._
```

We can now use the infix combination operator on any monoid type that Cats knows about.

```
1 |+| 3
// res0: Int = 4
1.0 |+| 2.0
// res1: Double = 3.0
"Hi" |+| "There"
// res2: String = HiThere
List(1,2,3) |+| List(4,5)
// res3: List[Int] = List(1, 2, 3, 4, 5)
```

This is nice, but perhaps not life–changing. But Cats also understands that `Map` will form a monoid provided that its values do, as the following illustrates.

```
val m1 = Map("a" -> 2, "b" -> 3)
// m1: Map[String,Int] = Map(a -> 2, b -> 3)
val m2 = Map("b" -> 4, "c" -> 5)
// m2: Map[String,Int] = Map(b -> 4, c -> 5)
m1 |+| m2
// res3: Map[String,Int] = Map(b -> 7, c -> 5, a -> 2)
```

Here the values are `Int`, which is a monoid, and hence so is a `Map[String,Int]`. Note that when the maps are combined, the values corresponding to the same key are combined using the monoid combination operation for the value type. This allows us to do operations similar to the `reduceByKey` operations we saw how to do in Spark. In particular, we can use Cats monoids to help us write a simple character frequency count as follows.

This is an example of reading text files from local storage. Using `fromURL` instead of `fromFile` allows reading directly from a URL.

```
scala.io.Source.
  fromFile("/usr/share/dict/words").
  getLines.
  map(_.trim).
  map(_.toLowerCase).
  flatMap(_.toCharArray).
  filter(_ > '/').
  filter(_ < '}').
  map(ch => Map(ch -> 1)).
  reduce(_ |+| _)
// res4: Map[Char,Int] = Map(e -> 88833, s -> 90113,
//  x -> 2124, n -> 57144, j -> 1948, y -> 12652,
// t -> 53006, u -> 26118, f -> 10675, a -> 64439, ...
```

Note how the monoid structure of `Map` is exploited in the final reduction.

There is much more to say about Cats, but time prevents us from exploring this important library further at this time...

9.3 Category theory primer

Category theory (CT) is a large, rather technical area of pure mathematics. Fortunately the amount of category theory you need to know for advanced FP is fairly limited. We clearly don't have time to delve into category theory in any detail in the context of this course, but here's a quick primer on the essential concepts that crop up most frequently in the context of FP. The main connection between FP and CT is that pure FP languages correspond closely with the typed lambda calculus, and the typed lambda calculus is a Cartesian closed category (CCC).

Category

- A category $\mathcal{C}$ consists of a collection of *objects*, $\mathrm{ob}(\mathcal{C})$, and *morphisms*, $\hom(\mathcal{C})$. Each morphism is an ordered pair of objects (an arrow between objects). For $x, y \in \mathrm{ob}(\mathcal{C})$, the set of morphisms from x to y is denoted $\hom_{\mathcal{C}}(x, y)$. $f \in \hom_{\mathcal{C}}(x, y)$ is often written $f : x \longrightarrow y$.
- Morphisms are closed under *composition*, so that if $f : x \longrightarrow y$ and $g : y \longrightarrow z$, then there must also exist a morphism $h : x \longrightarrow z$ written $h = g \circ f$.
- Composition is associative, so that $f \circ (g \circ h) = (f \circ g) \circ h$ for all composable $f, g, h \in \hom(\mathcal{C})$.
- For every $x \in \mathrm{ob}(\mathcal{C})$ there exists an *identity* morphism $\mathrm{id}_x : x \longrightarrow x$, with the property that for any $f : x \longrightarrow y$ we have $f = f \circ \mathrm{id}_x = \mathrm{id}_y \circ f$.

Examples of categories

- The category **Set** has an object for every *set*, and its morphisms represent set *functions*
  - Note that this is a category, since functions are composable and we have identity functions, and function composition is associative
  - Note that objects are "atomic" in category theory — it is not possible to "look inside" the objects to see the set elements — category theory is "point-free"
- For a pure FP language, we can form a category where objects represent *types*, and morphisms represent *functions* from one type to another
  - In Haskell this category is often referred to as **Hask**
  - This category is very similar to **Set**, in practice (both CCCs)
  - By modelling FP types and functions as a category, we can bring ideas and techniques from CT into FP

Set and Hask

- $0 \in \mathrm{ob}($**Set**$)$ is the empty set, $\emptyset$
  - There is a unique morphism from 0 to every other object — it is an example of the concept of an *initial object*
  - 0 in **Set** corresponds to the type `Void` in **Hask**, the type with no values

- $1 \in \mathrm{ob}(\textbf{Set})$ is a set containing exactly one element (and all such objects are *isomorphic*)
  - There is a unique morphism from every other object to 1 — it is an example of the concept of a *terminal object*
  - 1 in **Set** corresponds to the type `Unit` in **Hask**, the type with exactly one value, `()`
  - Morphisms from 1 to other objects must represent *constant* functions, and hence must correspond to *elements* of a set or *values* of a type — so we can use morphisms from 1 to "look inside" our objects if we must...

Monoid as a category with one object

- Given our definition of a category, we can now reconsider the notion of a *monoid* now as a *category with one object*
- The object represents the "type" of the monoid, and the *morphisms represent the "values"*
- From our definition of a category, we know that there is an *identity* morphism, that the morphisms are closed under *composition*, and that they are *associative*...
- For a monoid type object, M in **Hask**, the *(endo)morphisms* represent *functions*, $f_a : M \longrightarrow M$ defined by $f_a(m) = m \star a$
- Again, we see that it is the morphisms that really matter, and that these can be used to "probe" the "internal structure" of an object...

Functors

- A *functor* is a mapping from one category to another which preserves some structure
- A functor F from $\mathcal{C}$ to $\mathcal{D}$, written $F : \mathcal{C} \longrightarrow \mathcal{D}$ is a pair of functions (both denoted F):
  - $F : \mathrm{ob}(\mathcal{C}) \longrightarrow \mathrm{ob}(\mathcal{D})$
  - $F : \hom(\mathcal{C}) \longrightarrow \hom(\mathcal{D})$, where $\forall f \in \hom(\mathcal{C})$, we have $F(f : x \longrightarrow y) : F(x) \longrightarrow F(y)$
  - In other words, if $f \in \hom_{\mathcal{C}}(x, y)$, then $F(f) \in \hom_{\mathcal{D}}(F(x), F(y))$
- The functor must satisfy the *functor laws*:
  - $F(\mathrm{id}_x) = \mathrm{id}_{F(x)}, \forall x \in \mathrm{ob}(\mathcal{C})$
  - $F(f \circ g) = F(f) \circ F(g)$ for all composable $f, g \in \hom(\mathcal{C})$
- The laws ensure that we can form a category, called **Cat**, which has categories as objects and functors as morphisms
- A functor $F : \mathcal{C} \longrightarrow \mathcal{C}$ is called an *endofunctor* — in the context of functional programming, the word functor usually refers to an endofunctor F : **Hask** $\longrightarrow$ **Hask**

In the context of FP, the 2nd functor laws says that `x.map(f compose g) == x.map(g).map(f)`. Replacing the RHS with the LHS is known as *map fusion*, and is an optimisation automatically applied by Spark on RDDs. So category theory laws sometimes correspond to optimisations which can be automatically applied to code.

Natural transformations

- Often there are multiple functors between pairs of categories, and sometimes it is useful to be able to transform one to another
- Suppose we have two functors $F, G : \mathcal{C} \longrightarrow \mathcal{D}$
- A *natural transformation* $\alpha : F \Rightarrow G$ is a family of morphisms in $\mathcal{D}$, where $\forall x \in \mathcal{C}$, the *component* $\alpha_x : F(x) \longrightarrow G(x)$ is a morphism in $\mathcal{D}$
- To be considered *natural*, this family of morphisms must satisfy the *naturality law*:
  - $\alpha_y \circ F(f) = G(f) \circ \alpha_x, \quad \forall f : x \longrightarrow y \in \hom(\mathcal{C})$
- *Naturality* is one of the most fundamental concepts in category theory
- In the context of FP, a natural transformation could (say) transform an `Option` to a `List` (with at most one element)

Monads

- A *monad* on a category $\mathcal{C}$ is an endofunctor $T : \mathcal{C} \longrightarrow \mathcal{C}$ together with two natural transformations $\eta : \mathrm{Id}_{\mathcal{C}} \longrightarrow T$ (*unit*) and $\mu : T^2 \longrightarrow T$ (*multiplication*) fulfilling the *monad laws*:
  - *Associativity*: $\mu \circ T\mu = \mu \circ \mu_T$, as transformations $T^3 \longrightarrow T$
  - *Identity*: $\mu \circ T\eta = \mu \circ \eta_T = 1_T$, as transformations $T \longrightarrow T$
- The associativity law says that the two ways of *flattening* $T(T(T(x)))$ to $T(x)$ are the same
- The identity law says that the two ways of *lifting* $T(x)$ to $T(T(x))$ and then flattening back to $T(x)$ both get back to the original $T(x)$
- In FP, we often use `M` (for monad) rather than T (for triple), and say that there are three (or four) monad laws — the additional law(s) correspond either to the left and right identity laws being considered different, or to the naturality of μ (or both)

Kleisli category

- Kleisli categories formalise monadic composition
- For any monad T over a category $\mathcal{C}$, the *Kleisli category* of $\mathcal{C}$, written $\mathcal{C}_T$ is a category with the same objects as $\mathcal{C}$, but with morphisms given by:
  - $\hom_{\mathcal{C}_T}(x, y) = \hom_{\mathcal{C}}(x, T(y)), \ \forall x, y \in \mathrm{ob}(\mathcal{C})$

- The identity morphisms in $\mathcal{C}_T$ are given by $\mathrm{id}_x = \eta(x), \forall x$, and morphisms $f : x \longrightarrow T(y)$ and $g : y \longrightarrow T(z)$ in $\mathcal{C}$ can compose to form $g \circ_T f : x \longrightarrow T(z)$ via
  - $g \circ_T f = \mu_z \circ T(g) \circ f$

  leading to composition of morphisms in $\mathcal{C}_T$.
- In FP, the morphisms in C_T are often referred to as *Kleisli arrows*, or *Kleislis*, or sometimes just *arrows* (although *Arrow* usually refers to a generalisation of Kleisli arrows, sometimes known as *Hughes arrows*)

9.4 Other Scala libraries

9.4.1 Typeclasses and advanced FP

Spire

Library for numeric types, including support for the development of highly optimised numerical code.

github.com/non/spire

Simulacrum

As already mentioned in this chapter, first class support for typeclasses.

github.com/mpilquist/simulacrum

Shapeless

Generic programming support for Scala, including `HList`s (heterogeneous lists), which have many potential applications in data analysis.

github.com/milessabin/shapeless

9.4.2 Probabilistic programming

Figaro

General purpose library for probabilistic programming in Scala. See the end–of–chapter exercises.

github.com/p2t2/figaro

Factorie

Another probabilistic modelling library in Scala.

http://factorie.cs.umass.edu/

9.4.3 Big/streaming data

BIDData

BIDData is the umbrella project for various Scala libraries relating to machine learning, including BIDMat, a CPU/GPU accelerated matrix library, and BIDMach, an accelerated machine learning library.

github.com/BIDData

Akka–streams

Akka-streams is a Scala library for working with “reactive streams”. It provides a high-performance platform on which to build streaming data applications.

http://doc.akka.io/docs/akka/2.5/scala/stream/

FS2

FS2 is a more functional approach to building streaming data systems.

https://github.com/functional-streams-for-scala/fs2

9.5 Further reading

9.5.1 Introduction

I will focus here mainly on books, but it’s worth briefly noting that there are a number of other resources available, on-line and otherwise, that are also worth considering. I particularly like the Coursera course Functional Programming Principles in Scala — I still think this is probably the best way to get started with Scala and functional programming for most people. In fact, there is an entire Functional Programming in Scala Specialization which is very good.

9.5.2 Reading list

Getting started with Scala

Before one can dive into statistical computing and data science using Scala, it’s a good idea to understand a bit about the language and about functional programming. There are by now many books on Scala, and I haven’t carefully reviewed all of them, but I’ve looked at enough to have an idea about good ways of getting started.

- *Programming in Scala: Third edition*, Odersky et al, Artima.
  - This is *the* Scala book, often referred to on-line as **PinS**. It is a weighty tome, and works through the Scala language in detail, starting from the basics. Every serious Scala programmer should own this book. However, it isn’t the easiest introduction to the language.
- *Scala for the Impatient*, Horstmann, Addison-Wesley.
  - As the name suggests, this is a much quicker and easier introduction to Scala than PinS, but assumes reasonable familiarity with programming in general, and sort-of assumes that the reader has a basic knowledge of Java and the JVM ecosystem. That said, it does not assume that the reader is a Java expert. My feeling is that for someone who has a reasonable programming background and a passing familiarity with Java, then this book is probably the best introduction to the language. Note that there is a second edition in the works.

- *Functional Programming in Scala* Chiusano and Bjarnason, Manning.
  - It is possible to write Scala code in the style of "Java-without-the-semi-colons", but really the whole point of Scala is to move beyond that kind of Object-Oriented programming style. How much you venture down the path towards pure Functional Programming is very much a matter of taste, but many of the best Scala programmers are pretty hard-core FP, and there's probably a reason for that. But many people coming to Scala don't have a strong FP background, and getting up to speed with strongly-typed FP isn't easy for people who only know an imperative (Object-Oriented) style of programming. *This* is the book that will help you to make the jump to FP. Sometimes referred to online as **FPiS**, or more often even just as the **red book**, this is also a book that every serious Scala programmer should own (and read!). Note that it isn't really a book *about* Scala — it is a book about strongly typed FP that just "happens" to *use* Scala for illustrating the ideas. Consequently, you will probably want to augment this book with a book that really is about Scala, such as one of the books above. Since this is the first book on the list published by Manning, I should also mention how much I like computing books from this publisher. They are typically well-produced, and their paper books (pBooks) come with complimentary access to well-produced DRM-free eBook versions, however you purchase them.
- *Functional and Reactive Domain Modeling*, Ghosh, Manning.
  - This is another book that isn't really *about* Scala, but about software engineering using a strongly typed FP language. But again, it uses Scala to illustrate the ideas, and is an excellent read. You can think of it as a more practical "hands-on" follow-up to the red book, which shows how the ideas from the red book translate into effective solutions to real-world problems.
- *Structure and Interpretation of Computer Programs, second edition* Abelson et al, MIT Press.
  - This is not a Scala book! This is the only book in this list which doesn't use Scala at all. I've included it on the list because it is one of the best books on programming that I've read, and is the book that I wish someone had told me about 20 years ago! In fact the book uses Scheme (a Lisp derivative) as the language to illustrate the ideas. There are obviously important differences between Scala and Scheme - eg. Scala is strongly statically typed and compiled, whereas Scheme is dynamically typed and interpreted. However, there are also similarities — eg. both

languages support and encourage a functional style of programming but are not pure FP languages. Referred to on-line as **SICP**, or sometimes even as just the "Wizard book", this book is a classic. Note that there is no need to buy a paper copy if you like eBooks, since electronic versions are available free on-line.

Scala for statistical computing and data science

- *Scala for Data Science*, Bugnion, Packt.
  - Not to be confused with the (terrible) book, *Scala for machine learning* by the same publisher. *Scala for Data Science* is my top recommendation for getting started with statistical computing and data science applications using Scala. I have a blog post which reviews this book in detail, but to summarise I would highly recommend this book as a good complement to these course notes.
- *Scala Data Analysis Cookbook*, Manivannan, Packt.
  - I'm not a huge fan of the cookbook format, but this book is really mis–named, as it isn't really a cookbook and isn't really about data analysis in Scala! It is really a book about Apache Spark, and proceeds fairly sequentially in the form of a tutorial introduction to Spark. Spark is an impressive piece of technology, and it is obviously one of the factors driving interest in Scala, but it's important to understand that Spark isn't Scala, and that many typical data science applications will be better tackled using Scala without Spark. I've not read this book cover-to-cover as it offers little over Scala for Data Science, but its coverage of Spark is a bit more up-to-date than the Spark books I mention below, so it could be of interest to those who are mainly interested in Scala for Spark.
- *Scala High Performance Programming*, Theron and Diamant, Packt.
  - This is an interesting book, fundamentally about developing high performance streaming data processing algorithm pipelines in Scala. It makes no reference to Spark. The running application is an on-line financial trading system. It takes a deep dive into understanding performance in Scala and on the JVM, and looks at how to benchmark and profile performance, diagnose bottlenecks and optimise code. This is likely to be of more interest to those interested in developing efficient algorithms for scientific and statistical computing rather than applied data scientists, but it covers some interesting material not covered by any of the other books in this list.
- *Learning Spark*, Karau et al, O'Reilly.

  - This book provides an introduction to Apache Spark, written by some of the people who developed it. Spark is a big data analytics framework built on top of Scala. It is arguably the best available framework for big data analytics on computing clusters in the cloud, and hence there is a lot of interest in it. The book is a perfectly good introduction to Spark, and shows most examples implemented using the Java and Python APIs in addition to the canonical Scala (Spark Shell) implementation. This is useful for people working with multiple languages, but can be mildly irritating to anyone who is only interested in Scala. However, the big problem with this (and every other) book on Spark is that Spark is evolving very quickly, and so by the time any book on Spark is written and published it is inevitably very out of date. It's not clear that it is worth buying a book specifically about Spark at this stage, or whether it would be better to go for a book like *Scala for Data Science*, which has a couple of chapters of introduction to Spark, which can then provide a starting point for engaging with Spark's on-line documentation (which is reasonably good).

- *Advanced Analytics with Spark*, Ryza et al, O'Reilly.

  - This book has a bit of a "cookbook" feel to it, which some people like and some don't. It's really more like an "edited volume" with different chapters authored by different people. Unlike Learning Spark it focuses exclusively on the Scala API. The book basically covers the development of a bunch of different machine learning pipelines for a variety of applications. My main problem with this book is that it has aged particularly badly, as all of the pipelines are developed with raw RDDs, which isn't how ML pipelines in Spark are constructed any more. So again, it's difficult for me to recommend. The message here is that if you are thinking of buying a book about Spark, check very carefully when it was published and what version of Spark it covers and whether that is sufficiently recent to be of relevance to you.

9.5.3 Summary

There are lots of books to get started with Scala for statistical computing and data science applications. My "bare minimum" recommendation would be some generic Scala book (doesn't really matter which one), the *red book*, and *Scala for data science*. After reading those, you will be very well placed to top-up your knowledge as required with on-line resources.