# Chapter 15: High Performance Computing with R

Mark Andrews

## Contents

## Introduction

R is a high level programming language. Like most programming languages of this kind, R is designed for expressiveness — expressing complex statements and instructions in a simple, clear, and minimal syntax — rather than speed of execution. Often, R's relative lack of speed has no practical consequences for us. For example, if some task in R takes just a few hundred milliseconds to complete, even if we could speed it up by a few orders of magnitude, we might barely notice this in practice. Moreover, it might well be a waste of our time re-programming the task in order to achieve this speed up. Nonetheless, there are times when speed matters, and matters greatly. For example, many algorithms for statistical inference, both classical and Bayesian, use computationally intensive numerical methods such as numerical optimization, numerical integration or differentiation, and Monte Carlo methods. For these algorithms to be used for anything other than trivial problems, they must be implemented as computationally efficiently as possible. In these cases, implementing them using the set of R programming methods we explored in Chapter 7 might be highly inefficient. Fortunately, there are many ways in which we can continue using R, or at least using R as our computing environment, and achieve considerable improvement in computational speed and efficiencies. In this chapter, we will explore two of these approaches: interfacing with C++ code using `Rcpp` and coarse-grained parallel processing. We chose these two approaches over others because both are, in their own ways, very powerful, widely used in the R community, and relatively very straightforward to implement.

In addition to using C++ and parallel programming, in this chapter, we will also provide a very brief introduction to using Spark with R. Spark is a framework for big data analysis. It is usually used for analysing data sets that are too large for the RAM of one machine. A comprehensive introduction to Spark would require a book in itself. Here, we just hope to a brief introduction to what Spark is and how it can be used in R with `sparklyr`.

## Using C++ code with `Rcpp`

C++ is a general purpose programming language. As the name implies, it is an extension of the C programming language, particularly by the inclusion of object oriented programming. Since the 1990s, C++ has consistently been one of the most widely used programming of any kind, and its key strengths have been its power, flexibility, speed, and efficiency. C++ is a low level (relative to R, Python, etc), statically typed, compiled language. By contrast, R is high level, dynamically typed, interpreted language. What this means is that C++ code is more explicit, detailed, and less expressive than R. In addition, we must declare the data type of each variable in C++. For example, if `x` is an integer, we must explicitly declare that it is at the start of our code, and it can not be changed to another type such as a string in the rest of the code. This contrasts with R, where we never need to declare variables types in advance and can change them on the fly. Finally, C++ must be compiled into binary machine code by a compiler before it can be executed. By contrast, R code is executed one statement at a time by the R interpreter, which is another program (mostly written in C) that reads R code and converts it into machine instructions. In practice, this makes C++ more difficult to master as a programming language, and leads to more verbose, lengthy, and syntactically precise and unforgiving code. Also, because of its write-compile-execute operation, we can not use it interactively as we do with R. In return for this pain and inconvenience, however, C++ is much faster than R. As we will see, we can easily obtain orders of magnitude speed-ups if we rewrite computationally demanding R code in C++.

The `Rcpp` package (Eddelbuettel and Balamuta 2017) has made the use of C++ based functions etc from R particularly easy to accomplish. It allows us to focus on just writing the C++ code and not worry about the details of how C++ and R communicate. In most real-world cases, we put the C++ code in R packages, and use an R package builder commands to compile it. We then load that R package just like any other, e.g. using `library()`, and then execute the compiled C++ code just like any other R function. We will demonstrate this procedure in this section. To begin, however, we show how `Rcpp` can be used interactively in the console to create C++ functions that are accessible in the R session. This is very helpful for learning more about how `Rcpp` works and for prototyping code.

We should be clear that our coverage of `Rcpp` here is just a brief introduction, aimed at illustrating the general principles and providing some illustrative examples. It is emphatically not intended to be a comprehensive introduction. For a lengthier introduction, consider *Chapter 19: High performance function with Rcpp* in Wickham (2019), and for a comprehensive introduction, consider Eddelbuettel (2013), written by the author of `Rcpp`.

### Using `Rcpp` interactively

The C++ function we will use as our first example, named `average`, calculates the arithmetic mean of vector of values. Its code is as follows.

```
double average(NumericVector x){

  // Declare loop variable
  int i;
  // Declare vector length variable
  int n = x.size();
  // Declare variable that accumulates values
  double total = 0;

  for (i = 0; i < n; i++){
    total += x[i];
  }
  return total/n;
}
```

Obviously, the arithmetic mean is already implemented in R as `mean`, and so we can compare `average` and `mean`. In addition, it is useful to compare `average` to the following pure R implementation.

```
averageR <- function(x){

  total <- 0

  for (i in seq_along(x)){
    total <- total + x[i]
  }

  total/length(x)
}
```

As we can see, there are a number of differences between the C++ and the pure R implementations of this function. One of the most obvious differences is that we must declare the data types of the variables in the C++ code. More specifically, in `average`, we declare the following:

- The return value of `average` is stated to be a `double`, which is a double precision floating point number. A double is basically a single decimal number. Note that we make this declaration when we define the function, i.e. `double average(...`.
- The input vector `x` is declared as a `NumericVector`, which is an `Rcpp` specific, rather than C++ general, data type for using R numeric vectors in C++ functions. Note that we declare this data type in the function definition.
- The for loop variable `i` is declared as a integer using the key word `int`.
- The length of the input vector `x` is also declared as an integer with `int`. Note that we obtain the length of the `x` vector with `x.size()`.
- The variable `total` which calculates a running sum of the elements of `x` is declared as a `double`.

Other differences between `average` and `averageR` include the syntax of the for loop in `average` follows the standard C/C++ syntax where we specify the starting value of the loop variable (`i = 0`), an expression that determines whether the loop continues on each iteration (`i < n`; therefore, it terminates when `i == n`), and the increment to `i` on each iteration (`i++`). Very importantly, notice that C++ starts the vector index at 0 rather than 1. Thus, for example, in C++, `x[1]` is the *second* element of the vector `x`, while `x[0]` is the first element. In R, `x[1]` is the first element of `x`, and `x[0]` returns an empty vector. Also in the for loop, we use the expression `i++`. This is shorthand for `i += 1`, which is itself a shorthand for `i = i + 1`. The expression `total += 1` is therefore the same as `total = total + 1`.

Other differences between the C++ and pure R functions include that in the C++, we must terminate statements with `;`, we must also have an explicit `return` statement, we use `=` and not `<-` for assignment, and we do not use the `function` keyword in function definitions.

Now, let us compile `average` using the `cppFunction` from `Rcpp`. To do so, we simply paste the above function into `cppFunction` as a string.

```
library(Rcpp)
cppFunction('double average(NumericVector x){

// Declare loop variable
int i;
// Declare vector length variable
int n = x.size();
// Declare variable that accumulates values
double total = 0;
for (i = 0; i < n; i++){
  total += x[i];
}

return total/n;
}')
```

With this, `average` is now available to use in our R session just like any other function in R.

```
x <- runif(1e4)
average(x)
#> [1] 0.5003048
```

We can verify that this is correct with `mean` or `averageR`.

```
mean(x)
#> [1] 0.5003048
averageR(x)
#> [1] 0.5003048
```

Now, let us compare the performance of these functions using the `microbenchmark` tool.

```
library(tidyverse)
theme_set(theme_classic())
library(magrittr)
library(microbenchmark)
results <- microbenchmark(average(x),
                          mean(x),
                          averageR(x))
results
#> Unit: microseconds
#>         expr     min      lq      mean   median       uq     max neval
#>   average(x)   8.466   8.586  17.51699   8.6560   8.7715 890.911   100
#>      mean(x)  25.568  25.643  25.85332  25.7435  25.8585  30.898   100
#>  averageR(x) 358.994 361.323 364.64616 362.8650 365.6010 384.992   100
```

We see the `average` function is about 21 times faster than the base R `averageR` function in terms of its mean performance, and about 42 faster in terms of its median performance, on 100 iterations. We also see that `average` is roughly comparable in speed to the built in `mean` function. The `mean` is faster in terms of its mean value over the 100 iterations, but `average` has a faster median. This is notable because the `mean` is much more carefully designed and optimized function than `average`. This results show the substantial gains that can be achieved by rewriting our code in C++.

## Using `Rcpp` code in R packages

Using `cppFunction` is very useful but is intended for interactive use. For C++ code is intended to be used across multiple R sessions and projects, it preferable to put the code in its own files, compile it once, and then load the compiled functions into R sessions, R scripts, RMarkdown scripts, etc, and use them like any R function. The ideal way of doing this is to include the C++ code as a part of an R package, and use standard package build tools to compile and document the resulting functions.

We will demonstrate this by making a package that calculates a simple moving or rolling average. We will name this `smra` (*s*imple *m*oving/*r*olling *a*verage). We will use the `create_package` function in the `usethis` package to create a bare package directory with some of the necessary files and directories. To do this, we first load the `usethis` and the `fs` packages.

```
library(usethis)
library(fs)
```

The `usethis` package is a package for automating the setup and building of R packages, while `fs` provides tools for creating, listing, and manipulating files. Now, we create a directory called `smra` inside a temporary directory created by the `tempdir` command, and we save the path to this directory.

```
path_to_package <- path(tempdir(), 'smra')
```

Now, we create a list with all the information that we need to pass to the `create_package` command. This information is put in the `DESCRIPTION` file inside the `smra` package directory.

```
fields <- list(Title = "Simple Moving/Rolling Average",
               Version = "0.0.1",
               `Authors@R` = person(given = "Mark",
                                    family = "Andrews",
                                    role = c("aut", "cre"),
                                    email = "mark.andrews@ntu.ac.uk"),
               Description = "Calculate a simple moving/rolling average.",
               License = 'MIT Licence',
               LinkingTo = 'Rcpp',
               Imports = 'Rcpp'
)
```

Most of the fields here are self-explanatory. Note that the `Authors@R` field uses the function `person` to define the author the package. This author is assigned the roles `aut` (author) and `cre` (creator). Note also that the `LinkingTo` and `Imports` fields, both with values `Rcpp`, are necessary to allow us to build and export the `Rcpp` based functions. Now, we may execute the `create_package` command.

```
create_package(path_to_package,
               fields = fields,
               rstudio = F,
               open = F)
#> Package: smra
#> Title: Simple Moving/Rolling Average
#> Version: 0.0.1
#> Authors@R (parsed):
#>     * Mark Andrews <mark.andrews@ntu.ac.uk> [aut, cre]
#> Description: Calculate a simple moving/rolling average.
#> License: MIT Licence
#> Imports:
#>     Rcpp
#> LinkingTo:
#>     Rcpp
#> Encoding: UTF-8
#> LazyData: true
#> Roxygen: list(markdown = TRUE)
#> RoxygenNote: 7.1.1
```

In `create_package`, by setting `rstudio = T`, we set the directory to be an RStudio project, although this is not strictly necessary. By setting `open = F`, we do not open the resulting RStudio project/package in a new RStudio session.

At this point, the `smra` directory has two files, `DESCRIPTION` and `NAMESPACE`, and one empty directory named `R`, which we can see with the `dir_tree` command.

```
dir_tree(path_to_package)
/tmp/Rtmp8rS1ij/smra
+-- DESCRIPTION
+-- NAMESPACE
\-- R
```

We now create an empty file `code.cpp` and put this in a new directory named `src` in the package, which we will do using the `file_create` function from the `fs` package.

```r
src_dir <- dir_create(path(path_to_package, 'src'))
file_create(path(src_dir, 'code.cpp'))
```

We will now put the following C++ code in this `code.cpp` file.

```cpp
// code.cpp
NumericVector rolling_mean(NumericVector x, int k = 1) {

  // Declare outer loop counter
  int i;
  // Declare inner loop counter
  int j;
  // Declare vector length
  int n = x.size();
  // Declare inner loop summation variable
  double total;
  // Declare output vector
  NumericVector y(n);

  for (i = 0; i < n; i++) {
    if (i < k - 1){
      y[i] = NumericVector::get_na();
    } else {
      total = 0;
      for (j = 0; j < k; j++) {
        total += x[i - j];
      }
      y[i] = total/k;
    }
  }

  return y;
}
```

This program calculates a length $k$ moving or rolling mean of a vector. Specifically, given a vector whose values are $x_1, x_2, \ldots x_i \ldots x_n$, the length $k$ moving average is vector $y_1, y_2 \ldots y_i \ldots y_n$ whose values are

$$y_i = \frac{1}{k}\sum_{j=1}^{k} x_{i-(j-1)} = \frac{1}{k}\sum_{j=0}^{k-1} x_{i-j},$$

if $i \geq k$ and undefined otherwise. For example, if $k = 5$, for $i \geq k$,

$$y_i = \frac{x_i + x_{i-1} + x_{i-2} + x_{i-3} + x_{i-4}}{5}.$$

To do this, we loop over all values of $i$ from 0 to $n-1$ (remembering that C++ starts its index at 0), and for values of $i \leq k-1$, we assign the `NA` value to `y[i]`. For all other values of $i$, we calculate the mean of `x[i]` and the $k-1$ previous values. At the beginning, we declare the data types of all the variables in this program. Note that the output of the function is of type `NumericVector`. We declare that `y` is a `NumericVector` of length `n`, where `n` is the length of `x`, by the following statement.

```cpp
// code.cpp
  NumericVector y(n);
```

Before we compile this function, we add the following comments immediately before the function begins.

```
// code.cpp
//' Simple moving average of a numeric vector
//'
//'
//' @param x A numeric vector
//' @param k The window length of the moving average
//' @return A vector of the same length of x
//' @export
// [[Rcpp::export]]
NumericVector rolling_mean(NumericVector x, int k = 1) {
```

These are, in fact, more than just normal code comments. Those preceded by `//'` will be parsed by `roxygen2` to make the documentation for the resulting `rolling_mean` function. In addition, the `//' @export` statement ensures that `rolling_mean` will be an exported function of the package we make. The `roxygen2` documentation functions will write `export(rolling_mean)` to the `NAMESPACE` file in the R package. In addition, the `// [[Rcpp::export]]` comment will ensure that the C++ function is exported to R.

In order to ensure that more necessary information is included in the `NAMESPACE` file, we create the following file and place it in the `R` directory in the R package.

```
// smra-package.R
#' @useDynLib smra, .registration = TRUE
#' @importFrom Rcpp sourceCpp
NULL
```

The file structure of the package is now as follows.

```
dir_tree(path_to_package)
/tmp/Rtmp8rS1ij/smra
+-- DESCRIPTION
+-- NAMESPACE
+-- R
|   \-- smra-package.R
\-- src
    \-- code.cpp
```

Now, we are ready to build the package. First, we document the package, which also ensures that the necessary lines are added to `NAMESPACE`. We can do this with the `devtools` package's `document` function.

```
library(devtools)
document(path_to_package)
```

We can load the package with `load_all`, which is roughly equivalent to installing and loading with the normal `library` function.

```
load_all(path = path_to_package)
```

Now, we can use the `rolling_mean` function.

```
x <- rnorm(20)
rolling_mean(x, k = 5)
#>  [1]          NA          NA          NA          NA  0.99988573  0.92298972
#>  [7]  0.49955783 -0.08995363 -0.03230148  0.17451006  0.38450327  0.69945225
#> [13]  0.99568605  1.06842425  0.62634885  0.38136451  0.55827531  0.16512014
#> [19] -0.20141630 -0.26458559
```

We can verify that this rolling mean calculating is correct by comparing it to the `rollmean` function from the `zoo` package.

```
zoo::rollmean(x, k = 5, na.pad = T, align = 'right')
#>  [1]          NA          NA          NA          NA  0.99988573  0.92298972
#>  [7]  0.49955783 -0.08995363 -0.03230148  0.17451006  0.38450327  0.69945225
#> [13]  0.99568605  1.06842425  0.62634885  0.38136451  0.55827531  0.16512014
#> [19] -0.20141630 -0.26458559
```

## Parallel processing in R

Parallel processing is arguably *the* defining feature of high performance computing. At the very least, it is one of its defining features and one of its most important topics. Simply put, parallel processing is whenever we simultaneously execute multiple programs in order perform some task. To do this, we need more than one *computing units* (*processing units*) on our computer. In general, these units can be either cores on the central processing unit (CPU) or on the graphics processing unit (GPU). Modern supercomputing and high performance computing generally almost always uses a mixture of both CPUs and GPUs. However, although most modern desktops and laptop machines usually have multicore CPUs, most do not have general purpose GPUs, and so we will not consider GPU computing here.

The topic of parallel computing generally is a highly technical one, often focusing on relatively low level programming concepts, the details of the hardware, and the algorithmic details of the task be carried out. Here, we will avoid all of this complexity. We will focus exclusively on what are called *embarrassingly parallel* problems. These are computing problems can be easily broken down into multiple independent parts. We will also assume that we are working on a single computer (i.e., node), such as a laptop or desktop machine, rather than on a cluster of multiple nodes. And we will focus on parallel computing using R itself, as opposed to the parallelism that we can obtain using libraries such as OpenMP (Open Multi-Processing) or MPI (Message Passing Interface) when programming with C/C++ and other fast level languages.

### The `parallel` package

R provides many packages related to parallel computing. See the webpage https://cran.r-project.org/web/views/HighPerformanceComputing.html for a curated list of relevant packages. Here, we will focus exclusively on the R `parallel` package. This package builds upon and incorporates two other packages: `multicore` and `snow` (*s*imple *n*etwork *o*f *w*orkstations). The principal way that parallel processing is done using `parallel` is by using multiple new processes that are started by a command in R. These processes are known as *workers* and the R session that starts them is known as the *master*. The communication between the master and the workers can be done using *sockets*, and the code for doing this was developed by `snow`, or else by *forks*, and the code for this was developed by `multicore`. Forks create copies of the master process, including all its objects, and workers and the master processes share memory allocations Sockets are independent processes to which information from and to the master must be explicitly copied. In both forks and sockets, tasks are farmed out to the workers from the master using parallel version of "map" functionals, e.g. `lapply`, `purrr::map`, which we explored in Chapter 6. While there are many advantages to using forks, their key disadvantage is that they are not available on Windows. For that reason, we will only consider sockets here.

The `parallel` package is pre-installed in R and is loaded with the usual `library` command.

```
library(parallel)
```

We can use the `detectCores` function from `parallel` to list them number of available cores on our computer. The option `logical = FALSE` will list only physical, rather than virtual, cores. It is advisable to always set `logical = FALSE` and so report only physical cores as this in general list the maximum number of separate processes that can be executed simultaneously. The machine on which I am currently working has two Xeon Gold 6154 CPUs, each with 18 physical cores, and so there are 36 physical cores

```
detectCores(logical = F)
#> [1] 16
```

## Using clusterCall, clusterApply etc

The parallel packages provides many functions for socket based execution of parallel tasks or farming out tasks to workers. We will consider clusterCall, and clusterApply and related functions like parLapply, parSapply, etc. These are similar to one another, and are representative of how other socket based parallel functions in parallel work.

The clusterCall function calls the same function on each worker. As a very simple example, let us create a function that returns the square of a given number.

```
square <- function(x) x^2
```

To apply this in parallel, first, we start a set of workers. We can specify as many workers as we wish. Specifying more workers than there are cores will obviously not allow for all workers to occupy one whole core. Usually, if specifying the maximum number of workers, we set it to the total number of cores minus one. Assuming that the workers are involved in computationally intensive tasks, this allows one core free for other tasks, like R or RStudio, running on your computer. For now, we will use just 4 cores. We do this using the makeCluster command.

```
the_cluster <- makeCluster(4)
```

This command returns an object, which we name the_cluster here, and which we need to use in subsequent commands. Now we call square with input argument 10 on each worker.

```
clusterCall(cl = the_cluster, square, 10)
#> [[1]]
#> [1] 100
#>
#> [[2]]
#> [1] 100
#>
#> [[3]]
#> [1] 100
#>
#> [[4]]
#> [1] 100
```

The clusterCall returns a list with 4 elements. Each element is the value returned by the function square with argument 10 that was executed on each worker. We could also use clusterCall with anonymous functions, as in the following example, where we calculate the cube of 10.

```
clusterCall(cl = the_cluster, function(x) x^3, 10)
#> [[1]]
#> [1] 1000
#>
#> [[2]]
#> [1] 1000
#>
#> [[3]]
#> [1] 1000
#>
#> [[4]]
#> [1] 1000
```

As another example, here we call rnorm with input argument 5, which determines the number of random values.

```
clusterCall(cl = the_cluster, rnorm, 5)
#> [[1]]
#> [1]  0.09633051  0.54456749 -0.22439985 -1.02967480 -0.39163237
#>
#> [[2]]
#> [1] -0.3597894 -0.7848778 -0.5617933  0.2901544  1.0833893
#>
#> [[3]]
#> [1] -0.4014709  0.2447361 -1.0787930 -0.5953200 -1.1920969
#>
#> [[4]]
#> [1]  0.5727435 -0.1223220 -0.8355434 -0.4987002  0.1683486
```

Usually, we need the workers to perform different tasks, and so calling the same function with the same arguments is not usually what we what to do. To call the same function with different arguments, we can use `clusterApply` and related function. In the following, we apply `square` to each element of vector of five elements.

```
clusterApply(cl = the_cluster, x = c(2, 3, 5, 10), square)
#> [[1]]
#> [1] 4
#>
#> [[2]]
#> [1] 9
#>
#> [[3]]
#> [1] 25
#>
#> [[4]]
#> [1] 100
```

With `clusterApply`, we can still use anonymous functions and we can also supply optional input arguments.

```
clusterApply(cl = the_cluster, x = c(2, 3, 5, 10), function(x, k) x^k, 3)
#> [[1]]
#> [1] 8
#>
#> [[2]]
#> [1] 27
#>
#> [[3]]
#> [1] 125
#>
#> [[4]]
#> [1] 1000
```

Very similar to `clusterApply` is `parLapply`, which is the parallel counterpart to base R's `lapply`. In `lapply`, we supply a list, each element of which is applied to a function, and the result is returned as a new list.

```
lapply(list(x = 1, y = 2, z = 3), function(x) x^2)
#> $x
#> [1] 1
#>
#> $y
#> [1] 4
#>
#> $z
#> [1] 9
```

The `parLapply` uses an identical syntax but the the cluster being the first argument.

```
parLapply(cl = the_cluster, list(x = 1, y = 2, z = 3), function(x) x^2)
#> $x
#> [1] 1
#>
#> $y
#> [1] 4
#>
#> $z
#> [1] 9
```

Just as `sapply` can be used to simplify the output from `lapply`, we can use `parSapply` to simplify the output of `parLapply`. For example, the following, the returned list is simplified as a vector.

```
parSapply(cl = the_cluster, list(x = 1, y = 2, z = 3), function(x) x^2)
#> x y z
#> 1 4 9
```

When we are finished with the parallel processing, we must shut down the cluster.

```
stopCluster(the_cluster)
```

## Example 1: Bootstrapping

Bootstrapping, which have not covered in this book, is a means to obtaining a sampling distribution for an estimator. We sample with replacement from the observed data, and draw a sample that is the same size as the original. With each sample, we calculate the estimator. We repeat this process a large number of times to obtain a disribution of the estimators. As example, let us consider the `housing_df` data set that we considered in Chapter 8.

```
housing_df <- read_csv('data/housing.csv')
housing_df
#> # A tibble: 546 x 1
#>    price
#>    <dbl>
#>  1 42000
#>  2 38500
#>  3 49500
#>  4 60500
#>  5 61000
#>  6 66000
#>  7 66000
#>  8 69000
#>  9 83800
#> 10 88500
#> # ... with 536 more rows
```

This contains the prices of 546 houses in the city of Windsor, Ontario in 1987. We can sample the rows of `housing_df` with replacement 546 using the `sample` function as follows,

```
n <- nrow(housing_df)
housing_df[sample(seq(n), n, replace = T),]
#> # A tibble: 546 x 1
#>     price
#>     <dbl>
#>   1  64500
#>   2  77500
#>   3  56000
#>   4  66000
#>   5  73000
#>   6 123500
#>   7  65000
#>   8  64500
#>   9  73500
#> 10  65000
#> # ... with 536 more rows
```

A function that can be used with any `lm` model is as follows,

```
bootstrap_lm <- function(formula, data){
  n <- nrow(data)
  resampled_data <- data[sample(seq(n), n, replace = T), ]
  lm(formula, data = resampled_data) %>%
    coef()
}
```

We can apply this to the `housing_df` data as follows.

```
bootstrap_lm(price ~ 1, data = housing_df)
#> (Intercept)
#>    69189.93
```

Note that this will produce just one bootstrap estimate. We need to reapply this a large number of times to obtain our distribution of bootstrapped estimates. This can be done in parallel. For that, we first start a new cluster of 4 workers.

```
the_cluster <- makeCluster(4)
```

Then, for code re-use, we create a function to perform a parallel distributed `bootstrap_lm(price ~ 1, data = housing_df)` call a specified $n$ times.

```
parallel_bootstrap <- function(n){
  parLapply(the_cluster,
            seq(n),
            function(x) {bootstrap_lm(price ~ 1, data = housing_df)}
  )
}
```

Were we to immediately try `parallel_bootstrap`, it would fail because the workers to do not have the function `bootstrap_lm` or `housing_df`, which are both defined in the master process's R environment. To send `bootstrap_lm` and `housing_df` to the workers we must do the following.

```
clusterExport(cl = the_cluster, varlist = c('bootstrap_lm', 'housing_df'))
```

Note that `varlist` is a vector of names, rather than the objects themselves. This is not sufficient, however, because `bootstrap_lm` uses the pipe `%>%`, which must be loaded by a `library` call such as `library("tidyverse")`. We can execute this package load function on all workers with `clusterEvalQ` as follows.

```
clusterEvalQ(cl = the_cluster, library("tidyverse"))
```

Now, we may execute the parallel bootstrapping.

```
estimates <- parallel_bootstrap(10000)
```

The resulting `estimates` will be a list, which we can easily convert to a vector and then plot their histogram. We could have also used `parSapply` instead `parLapply` to directly return a vector. The histogram of these bootstrapped estimates are shown in Figure 1.

```r
tibble(estimates = unlist(estimates)) %>%
  ggplot(aes(x = estimates)) + geom_histogram(bins = 100, col = 'white')
```

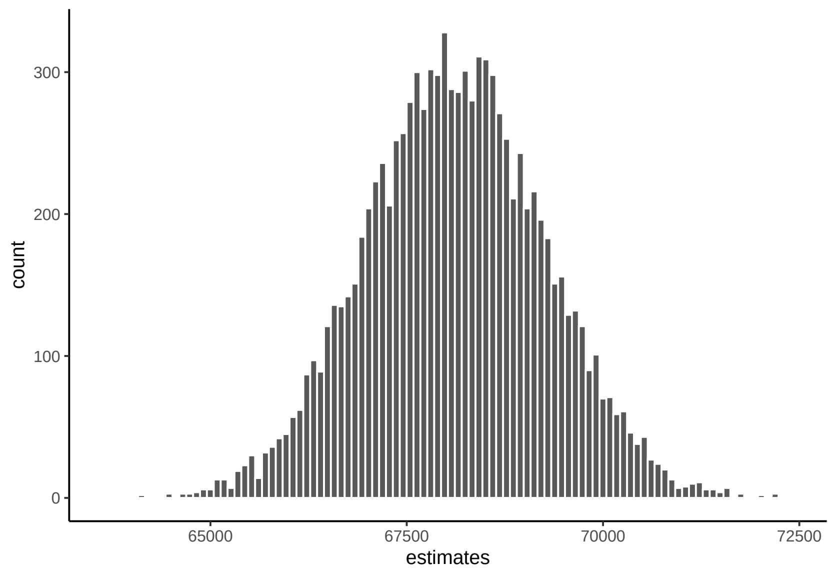

Figure 1: Bootstrap estimates of the mean of the prices in the `housing_df` data.'

Let us now compare the time taken by `parLapply` to sample 10000 estimates and compare that to the time take by `lapply` to do the same thing. For this timing, we will use the relatively crude timing procedure of using `Sys.time()`.

```r
sequential_bootstrap <- function(n){
  lapply(seq(n),
           function(x) {bootstrap_lm(price ~ 1, data = housing_df)}
  )
}

# parallel version with 4 workers
start_time <- Sys.time()
estimates <- parallel_bootstrap(10000)
parallel_version <- Sys.time() - start_time

# sequential version
start_time <- Sys.time()
estimates <- sequential_bootstrap(10000)
sequential_version <- Sys.time() - start_time
```

The times are as follows.

```r
parallel_version
#> Time difference of 1.739148 secs
sequential_version
#> Time difference of 5.815549 secs
```

and so the parallel version is over 3.34 times as fast. However, it is interesting to note that this roughly linear speed up, which is the ideal speed up in parallel processing, is not always going to happen. Consider, for example, the case of obtaining just 10 bootstrap estimates.

```r
# parallel version with 4 workers
start_time <- Sys.time()
```

```
estimates <- parallel_bootstrap(10)
parallel_version <- Sys.time() - start_time

# sequential version
start_time <- Sys.time()
estimates <- sequential_bootstrap(10)
sequential_version <- Sys.time() - start_time
```

The times are now as follows.

```
parallel_version
#> Time difference of 0.007232904 secs
sequential_version
#> Time difference of 0.00904274 secs
```

and so the sequential version is actually faster. This occurs because there is overhead in farming out the tasks and communicating between the master and workers. For this reason, in general, it is always possible for a parallel processing task to be slower than its sequential counterpart.

```
stopCluster(the_cluster)
```

## Example 2: Parallel execution of mcmc models

Throughout this book, we have used MCMC based Bayesian models, mostly using `brms`, and in Chapter 17, we cover Stan directly. In general, MCMC is very computational intensive. Although `brms` and Stan allow us to easily execute the chains within each model in parallel, often we need to perform multiple separate `brms` or Stan models. When working on a high end workstation or cluster, we would like to take advantage of all the cores available to us to do this.

As an example, let us reconsider the `affairs_df` data set that we covered in Chapter 10.

```
affairs_df <- read_csv('data/affairs.csv')
affairs_df
#> # A tibble: 601 x 9
#>    affairs gender   age yearsmarried children religiousness education occupation
#>      <dbl> <chr>  <dbl>        <dbl> <chr>            <dbl>     <dbl>      <dbl>
#>  1       0 male      37        10    no                   3        18          7
#>  2       0 female    27         4    no                   4        14          6
#>  3       0 female    32        15    yes                  1        12          1
#>  4       0 male      57        15    yes                  5        18          6
#>  5       0 male      22         0.75 no                   2        17          6
#>  6       0 female    32         1.5  no                   2        17          5
#>  7       0 female    22         0.75 no                   2        12          1
#>  8       0 male      57        15    yes                  2        14          4
#>  9       0 female    32        15    yes                  4        16          1
#> 10       0 male      22         1.5  no                   4        14          4
#> # ... with 591 more rows, and 1 more variable: rating <dbl>
```

This gives us the number of extramarital affairs (`affairs`) in the last 12 months by each of `nrows(affairs_df)` people. We could model this variable using, amongst other things, a Poisson, or negative binomial, or by the zero-inflated counterparts of these models. Likewise, we could use any combination of the eight predictor variables that we have available to us. Each one of these models could around at least a minute to complete. For many `brms` or Stan models, however, the running time could many hours or even days. Even though each model can use up to four cores, with one core per chain, if we were working on a workstation or cluster, we could execute many of these models simultaneously. For example, the workstation I am using now has 36 cores and so I could comfortably execute up to 8 models simultaneously, leaving a few hours free for other tasks. This could be accomplished by opening 8 different RStudio sessions and running a different model in each one, or running 8 different R scripts using `Rscript` in 8 different DOS or Unix terminals. It is, however, much more convenient and manageable to have a single R script that creates 8 workers and farms out one of the 8 models to each one.

Let us consider the following Poisson regression as a prototypical model.

```
M <- brm(affairs ~ gender + age + yearsmarried,
         data = affairs_df,
         cores  4,
         iter = 25000,
         family = poisson(link = 'log'))
```

Although not at all necessary for this model, we will use 25000 iterations per chain in order to resemble for larger and hence slower model. The running time, including the C++ compilation, for this model is approximately 35 seconds. Variants of this model might use either a different formula, or a different `family` or both. We can make a function that accepts different formulas and families. We do this, for reasons that will be soon clear, by creating a function that accepts one input argument that is a list with elements named `formula` and `family`.

```
affairs_model <- function(input){
  brm(input[['formula']],
      data = affairs_df,
      cores = 4,
      iter = 25000,
      family = input[['family']])
}
```

Now, we create a list of models specifications. Each element of this list is itself a list with two elements: `formula` and `family`. The `formula` is a `brmsformula` specifying the outcome variable and predictors. The `family` is one of the probability distribution families that `brms` accepts. Here, we specify six models, which are each combination of two sets of predictors and three different families.

```
model_specs <- list(

  # Poisson model with 3 predictors
  v1 = list(formula = brmsformula(affairs ~ gender + age + yearsmarried),
            family = poisson(link = 'log')),

  # Poisson model with 5 predictors
  v2 = list(formula = brmsformula(affairs ~ gender + age + yearsmarried + religiousness + rating),
            family = poisson(link = 'log')),

  # Negative binomial with 3 predictors
  v3 = list(formula = brmsformula(affairs ~ gender + age + yearsmarried),
            family = negbinomial(link = "log", link_shape = "log")),

  # Negative binomial with 5 predictors
  v4 = list(formula = brmsformula(affairs ~ gender + age + yearsmarried + religiousness + rating),
            family = negbinomial(link = "log", link_shape = "log")),

  # Zero-inflated Poisson with 3 predictors
  v5 = list(formula = brmsformula(affairs ~ gender + age + yearsmarried),
            family = zero_inflated_poisson(link = "log", link_zi = "logit")),

  # Zero-inflated Poisson with 3 predictors
  v6 = list(formula = brmsformula(affairs ~ gender + age + yearsmarried + religiousness + rating),
            family = zero_inflated_poisson(link = "log", link_zi = "logit"))
)
```

Using `parLapply` or a related function, we can farm each one of these model specification out to a worker. That worker will then compile the model and sample from it using four chains, with each chain on its own core. Thus, when sampling, $6 \times 4$ cores will be in use. The 6 resulting models are then passed back to a list named `results`. First, we start the cluster of 6 workers, and to each, we export the `affairs_df` data frame and load the `brms` package.

```
the_cluster <- makeCluster(6)

clusterExport(cl = the_cluster, varlist = 'affairs_df')
clusterEvalQ(cl = the_cluster, library("brms"))
```

We then run the `parLapply` using `model_specs` as the list and `affairs_model` as the function to which each element of the list will be applied. For comparison with the sequential model, we will time it.

```
start_time <- Sys.time()

results <- parLapply(cl = the_cluster,
                     model_specs,
                     affairs_model)

parallel_version <- Sys.time() - start_time
```

The running time is 79 seconds. For comparison, executing these models in a sequential functional like `lapply` could be done as follows.

```
start_time <- Sys.time()

sequential_results <- lapply(model_specs, affairs_model)

sequential_version <- Sys.time() - start_time
```

The running time in this case is 217 seconds, and so it is about 2.7 times slower than the parallel version.

Now, all the results of these models are in the list `results` with names `v1`, `v2`, etc. We can use these models completely as normal. For example, to extract the WAIC value of each model, we could create a helper function `get_waic` and apply it to each element by `results`.

```
get_waic <- function(model){
  waic(model)$estimates['waic', 'Estimate']
}
```

We will apply `get_waic` to `results` using a parallel functional like `parLapply` or `parSapply`, etc, but for these models, it could also be done using, for example, `lapply` or `purrr::map`.

```
waic_results <- parSapply(cl = the_cluster,
                          results,
                          get_waic)

waic_results
#>       v1       v2       v3       v4       v5       v6
#> 3262.297 2910.549 1495.519 1471.374 1646.557 1617.750

stopCluster(the_cluster)
```

## Spark

Apache Spark is a very popular framework for big data analysis. Put very simply, it is used for doing data procesing and data analysis where both the data and the processing are distributed across multiple nodes on a cluster. Spark is not an R based tool. It is written in the programming language Scala, which is derived from Java. However, Spark can be used from R via packages like `sparklyr`. In this chapter, we will provide a brief introduction to using `sparklyr`.

## Installing sparklyr and Spark

The usual way of working with Spark is on a remote Spark cluster. When learning about Spark, especially when using `sparklyr`, it is easier to use a local Spark installation on the computer on which you are working. Using Spark locally often defeats the whole purpose of using Spark, which is to perform parallel and distributed computer on large data sets that do on fit in the the RAM of a single node. Nonetheless, the basics of Spark and `sparkylr` can be learned more easily this way.

We install `sparklyr` just as we would any R package using `install.packages`, and load it with `library`.

```
library(sparklyr)
```

Once installed and loaded, we can install Spark locally as follows.

```
spark_install()
```

Note that this requires Java to be installed on your machine. Now that we've installed Spark, we can create a connection to it as follows.

```
connection <- spark_connect(master = 'local')
```

Were we to connect to a remote Spark cluster, we would still use `spark_connect` but with different arguments. We can now verify the version of Spark we are using.

```
spark_version(connection)
#> [1] '2.4.3'
```

## Copying data to Spark

Now, we need to get data to our local Spark cluster. Again, in practice, the data would be very large and would reside on the cluster, and so would not copy it from our local device or R session to the cluster. For the present example, our Spark installation begins empty and so we have no choice but to copy data there in order to use Spark. There are no restrictions on the kind of data frame we can use. We will arbitrarily choose the `HI` data frame from the `Ecdat` package. This gives the health insurance and hours worked by wives.

```
data(HI, package = 'Ecdat')
HI %>% as_tibble()
#> # A tibble: 22,272 x 13
#>    whrswk hhi   whi   hhi2  education race  hispanic experience kidslt6 kids618
#>     <int> <fct> <fct> <fct> <ord>     <fct> <fct>         <dbl>   <int>   <int>
#>  1      0 no    no    no    13-15yea~ white no             13         2       1
#>  2     50 no    yes   no    13-15yea~ white no             24         0       1
#>  3     40 yes   no    yes   12years   white no             43         0       0
#>  4     40 no    yes   yes   13-15yea~ white no             17         0       1
#>  5      0 yes   no    yes   9-11years white no             44.5       0       0
#>  6     40 yes   yes   yes   12years   white no             32         0       0
#>  7     40 yes   no    yes   16years   white no             14         0       0
#>  8     25 no    no    no    12years   white no              1         1       0
#>  9     45 no    yes   no    16years   white no              4         0       0
#> 10     30 no    no    yes   13-15yea~ white no              7         1       0
#> # ... with 22,262 more rows, and 3 more variables: husby <dbl>, region <fct>,
#> #   wght <int>
```

We can copy `HI` to our Spark cluster with `dplyr::copy_to` as follows.

```
hi_df <- copy_to(connection, HI)
```

We can verify that it was copied to the cluster by listing the tables there.

```
src_tbls(connection)
#> [1] "hi"
```

The major class of `hi_df` is now `tbl_spark`, or a Spark based tibble or data-frame.

```
class(hi_df)
#> [1] "tbl_spark" "tbl_sql"   "tbl_lazy"  "tbl"
```

Although this appears just like a tibble in our local R session, this is just essentially a link to a data frame stored on our cluster. By typing the name `hi_df`, we see the first few rows.

```
hi_df
#> # Source: spark<HI> [?? x 13]
#>    whrswk hhi   whi   hhi2  education race  hispanic experience kidslt6 kids618
#>     <int> <chr> <chr> <chr> <chr>     <chr> <chr>         <dbl>   <int>   <int>
#>  1      0 no    no    no    13-15yea~ white no             13         2       1
#>  2     50 no    yes   no    13-15yea~ white no             24         0       1
#>  3     40 yes   no    yes   12years   white no             43         0       0
#>  4     40 no    yes   yes   13-15yea~ white no             17         0       1
#>  5      0 yes   no    yes   9-11years white no             44.5       0       0
#>  6     40 yes   yes   yes   12years   white no             32         0       0
#>  7     40 yes   no    yes   16years   white no             14         0       0
#>  8     25 no    no    no    12years   white no              1         1       0
#>  9     45 no    yes   no    16years   white no              4         0       0
#> 10     30 no    no    yes   13-15yea~ white no              7         1       0
#> # ... with more rows, and 3 more variables: husby <dbl>, region <chr>,
#> #   wght <int>
```

Note that it does not list the number of rows, because it has not read them all, and just read the top. We can, however, see more rows with the `print` function. For example, to see the first 100 rows, we would do the following (we will omit the results).

```
print(hi_df, n = 100)
```

## Data wrangling with Spark

We can do exploratory analysis of Spark based data frames using the `dplyr` verbs just like we would normally. For example, we can select with `select`.

```
hi_df %>% select(whrswk, hhi, starts_with('kids'))
#> # Source: spark<?> [?? x 4]
#>    whrswk hhi   kidslt6 kids618
#>     <int> <chr>   <int>   <int>
#>  1      0 no          2       1
#>  2     50 no          0       1
#>  3     40 yes         0       0
#>  4     40 no          0       1
#>  5      0 yes         0       0
#>  6     40 yes         0       0
#>  7     40 yes         0       0
#>  8     25 no          1       0
#>  9     45 no          0       0
#> 10     30 no          1       0
#> # ... with more rows
```

We can filter with `filter`.

```
hi_df %>% filter(education == '13-15years', hispanic == 'no')
#> # Source: spark<?> [?? x 13]
#>    whrswk hhi   whi   hhi2  education race  hispanic experience kidslt6 kids618
#>     <int> <chr> <chr> <chr> <chr>     <chr> <chr>         <dbl>   <int>   <int>
#>  1      0 no    no    no    13-15yea~ white no               13       2       1
#>  2     50 no    yes   no    13-15yea~ white no               24       0       1
#>  3     40 no    yes   yes   13-15yea~ white no               17       0       1
#>  4     30 no    no    yes   13-15yea~ white no                7       1       0
#>  5     45 no    yes   yes   13-15yea~ white no               16       0       2
#>  6     20 yes   no    yes   13-15yea~ white no               13       1       1
#>  7     10 yes   no    yes   13-15yea~ white no               24       0       3
#>  8      0 no    no    no    13-15yea~ white no               14       2       1
#>  9     40 no    yes   no    13-15yea~ white no               10       0       5
#> 10     22 yes   no    yes   13-15yea~ white no               15       2       0
#> # ... with more rows, and 3 more variables: husby <dbl>, region <chr>,
#> #   wght <int>
```

We can modify the data frame with `mutate`. For example, here we select sum `kidslt6` and `kids618` as `kids`

```
hi_df %>% mutate(kids = kidslt6 + kids618) %>% select(starts_with('kids'))
#> # Source: spark<?> [?? x 3]
#>    kidslt6 kids618  kids
#>      <int>   <int> <int>
#>  1       2       1     3
#>  2       0       1     1
#>  3       0       0     0
#>  4       0       1     1
#>  5       0       0     0
#>  6       0       0     0
#>  7       0       0     0
#>  8       1       0     1
#>  9       0       0     0
#> 10       1       0     1
#> # ... with more rows
```

Other `dplyr` verbs can also be used, but not all the verbs work. For example, trying `slice` will raise the error `Slice is not supported in this version of sparklyr`.

When dealing with large data frames, we often want to reduce them. For this, `summarize` and `group_by` are vital.

```
hi_df %>%
  group_by(education) %>%
  summarise(whrswk = mean(whrswk, na.rm = T))
#> # Source: spark<?> [?? x 2]
#>   education  whrswk
#>   <chr>       <dbl>
#> 1 <9years      12.5
#> 2 16years      30.3
#> 3 13-15years   27.4
#> 4 12years      24.4
#> 5 9-11years    16.9
#> 6 >16years     34.9
```

If we wish to save the results of an analysis, such as the `summarise` based results above, we can pipe them to the function `compute` as follows.

```
hi_df %>%
  group_by(education) %>%
  summarise(whrswk = mean(whrswk, na.rm = T)) %>%
  compute('whrswk_summary')
#> # Source: spark<whrswk_summary> [?? x 2]
#>   education  whrswk
#>   <chr>       <dbl>
#> 1 <9years      12.5
#> 2 16years      30.3
#> 3 13-15years   27.4
#> 4 12years      24.4
#> 5 9-11years    16.9
#> 6 >16years     34.9
```

Now `whrswk_summary` is another table on the Spark cluster.

```
src_tbls(connection)
#> [1] "hi"             "whrswk_summary"
```

On the other hand, if we wanted to return this intermediate data frame to R, we'd use `collect` as in the following example.

```
whrswk_summary_df <-  hi_df %>%
  group_by(education) %>%
  summarise(whrswk = mean(whrswk, na.rm = T)) %>%
  collect()
```

We see that `whrswk` is just a regular tibble.

```
class(whrswk_summary_df)
#> [1] "tbl_df"     "tbl"        "data.frame"
```

In `sparklyr`, there are a number of functions that are similar to `mutate` in that they can be used to create new variables. These are primarily intended to produce data for use with machine learning methods. In the context of machine learning, predictor variables for use in predictive models are often referred to a "features". Hence, these variable creation or transformation function often named `ft_<verb>`, with `ft` for feature. For example, if we want to discretize a continuous variable according to the range of values it is in, similar to R's `cut` function, we could use `ft_bucketizer` as in the following example.

```
hi_df %>%
  ft_bucketizer("whrswk", "whrswk_cat", splits = c(0, 10, 30, 50, 70, Inf)) %>%
  select(starts_with('whrs'))
#> # Source: spark<?> [?? x 2]
#>    whrswk whrswk_cat
#>     <int>      <dbl>
#>  1      0          0
#>  2     50          3
#>  3     40          2
#>  4     40          2
#>  5      0          0
#>  6     40          2
#>  7     40          2
#>  8     25          1
#>  9     45          2
#> 10     30          2
#> # ... with more rows
```

Note the `ft_bucketizer` function, like all `ft_` function, requires the name of the variable to which the transformation is appled, and the name of the variable created, to be character strings.

As another example, if we want to convert a categorical variable into a type of dummy code known as a one-hot code, where each value of the variable is represented by a 1 in an vector of zeros, we can use `ft_one_hot_encoder`. This only applies to numerical variables, so to apply it to a character vector, we'd first transform the character vector to

```
hi_df %>%
  ft_string_indexer('region', 'iregion') %>%
  ft_one_hot_encoder('iregion', 'one_hot_region') %>%
  select(region, one_hot_region)
#> # Source: spark<?> [?? x 2]
#>    region       one_hot_region
#>    <chr>        <list>
#>  1 northcentral <dbl [3]>
#>  2 northcentral <dbl [3]>
#>  3 northcentral <dbl [3]>
#>  4 northcentral <dbl [3]>
#>  5 northcentral <dbl [3]>
#>  6 northcentral <dbl [3]>
#>  7 northcentral <dbl [3]>
#>  8 northcentral <dbl [3]>
#>  9 northcentral <dbl [3]>
#> 10 northcentral <dbl [3]>
#> # ... with more rows
```

Clearly, the `one_hot_region` is a list column. To see what it has done, we can collect the results using `collect`, and using `unnest_wider` (not available on Spark) to unnest the list column into new variables.

```
regions_df <- hi_df %>%
  ft_string_indexer('region', 'iregion') %>%
  ft_one_hot_encoder('iregion', 'one_hot_region') %>%
  select(region, one_hot_region) %>%
  collect()
```

```
regions_df %>% unnest_wider(one_hot_region) %>% distinct()
#> # A tibble: 4 x 4
#>   region        ...1  ...2  ...3
#>   <chr>        <dbl> <dbl> <dbl>
#> 1 northcentral     0     1     0
#> 2 other            0     0     1
#> 3 south            1     0     0
#> 4 west             0     0     0
```

## Machine learning using Spark

Spark provides many machine learning tools. As mentioned in Chapter 1, machine learning is related to statistical modelling, which is the major topic of this book, though it is not something that we have directly addressed. However, some tools and methods used in machine learning are identical to widely used traditional statistical modelling methods, even if there sometimes is some change in terminology. One example of this is logistic regression, which we covered in Chapter 10. Logistic regression is also a widely used in machine learning, particularly for predictive modelling.

In R, a logistic regression predicting `whi`, whether a wife has health insurance through her job, from `husby`, husband's income, in the `HI` data set is accomplished as follows.

```
M <- glm(whi ~ husby, data = HI, family = binomial(link = 'logit'))
```

Using `sparklyr`, we can do the following.

```
M_spark <- ml_logistic_regression(hi_df, whi ~ husby)
```

What is returned by the `ml_logistic_regression` output are the coefficients.

```
M_spark
#> Formula: whi ~ husby
#>
#> Coefficients:
#>  (Intercept)        husby
#> -0.437475437 -0.003021714
```

We can verify that these match those returned by `glm`.

```
coefficients(M)
#>  (Intercept)        husby
#> -0.437475397 -0.003021715
```

Spark will not provide measures of uncertainty in the estimators, e.g. standard errors, from which we can obtain p-values and confidence intervals on coefficients. Likewise, other quantities like the log of the likelihood function or deviance are not provided, so that we can not easily do model comparison based on log likelihood ratio tests. However, it is standard in machine learning to asses model fit using out of sample predictive performance by dividing the data into so-called "training" and "test" tests, fitting the model with the training set, and testing its performance on the test set. In the following, we split the data into a training set, which is 90%, and a test set, which is 10%.

```
partitions <- hi_df %>%
  sdf_random_split(training = 0.9, test = 0.1)

M_spark <- partitions$training %>%
  ml_logistic_regression(whi ~ husby)
```

We can now test how well the model predicts the test set using `ml_binary_classification_evaluator`.

```
predictions <- ml_predict(M_spark, partitions$test)

ml_binary_classification_evaluator(predictions)
#> [1] 0.5248326
```

The value that is returned here is the area under the ROC curve. This has multiple interpretations, but ultimately is based on the model's true positive rate (correct classification of positive instances, i.e. where outcome variable is equal to 1) versus false positive rate (misclassification of a negative instance). A value of 0.5 indicates the logistic regression is at chance level.

## References

Eddelbuettel, D. 2013. *Seamless R and C++ Integration with Rcpp.* Use R! Springer New York.

Eddelbuettel, Dirk, and James Joseph Balamuta. 2017. "Extending *R* with *C++*: A Brief Introduction to *Rcpp*." *PeerJ Preprints* 5 (August).

Wickham, Hadley. 2019. *Advanced R.* 2nd ed. Chapman; Hall/CRC.