Machine Learning Training: Hands-on Sessions

Ewen Gallic

December 2021

Contents

List of Tables

List of Figures

Hands-on

This notebook provides hands-on sessions for the European Central Bank Machine Learning Training: "Machine learning for econometricians."

The hands-on session use the programming language R. To run the codes you will need to download and install R and RStudio. More details on how to install R are available in the next chapter.

The schedule is the following:

Session	Subject	Schedule
Session 1	Gradient descent and overfitting	1 December 9.30-11.30
Session 2	Random forest and SVM	17 December 15.30-17.30
Session 3	Deep learning	21 December 14.30-16.30

Required packages

After you have successfully installed R, you will need to install a few packages.

- **Session 1:**
  - **Optimisation:**

    Required packages: tidyverse, numDeriv, plot3D

    If these are not installed, please run the following instructions in R:

    ```
    install.packages("tidyverse", "numDeriv", "plot3D")
    ```

  - **Overfitting:**

    Required packages: tidyverse, readxl, kableExtra, randomForest, ISLR, glmnet

    If these are not installed, please run the following instructions in R:

```
install.packages(
   c("tidyverse", "readxl", "kableExtra",
     "randomForest", "ISLR", "glmnet")
)
```

- **Session 2**: Random forest and SVM: 17 December 15.30-17.30
  - **Trees and Ensemble Methods**:

    Required packages: tidyverse, lubridate, arsenal, cowplot, corrplot, rpart, rpart.plot, ggtext, foreach

    If these are not installed, please run the following instructions in R:

```
install.packages(c("tidyverse", "lubridate", "arsenal", "cowplot",
   "corrplot", "rpart", "rpart.plot", "ggtext", "foreach",
   "ipred", "lattice", "caret", "randomForest"))
```

  - **Support Vector Machines**:

    Required packages: tidyverse, e1071

    If these are not installed, please run the following instructions in R:

```
install.packages(c("tidyverse", "e1071"))
```

- **Session 3**: Deep learning: 21 December 14.30-16.30
  - **Single notebook**:

    Required packages: tidyverse, fastDummies, reshape2, keras

```
install.packages("tidyverse")
install.packages("fastDummies")
install.packages("reshape2")
```

    To install Keras: please run the following instructions in R:

```
install.packages("tensorflow")
install.packages("keras")
library(keras)
tensorflow::install_tensorflow()
```

```
tensorflow::tf_config()
install_keras()
```

On one of my machines, I had to install Miniconda:

```
unlink(reticulate::miniconda_path(), recursive = TRUE)
reticulate::install_miniconda(path = reticulate::miniconda_path(),
                      update = TRUE, force = FALSE)
keras::install_keras()
```

Chapter 1

Install R

The hands-on sessions will use R and the integrated development environment RStudio.

There are at least two software to install:

- r
- RStudio.

The installation of R depends on your operating system (OS). First pick a mirror site at the following address: https://cran.biotools.fr/.

The installation process is described on each mirror site. Here are some key points you should focus on, depending on your OS.

1.1 Windows Users

1.1.1 Install R

You need to install the **latest version of R** by going to the following address:

`https://cran.your_mirror/bin/windows/base/`

by replacing `your_mirror` by the mirror you picked. For example, if your mirror is "Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)": https://ftp.fau.de/cran/bin/windows/base/.

1.1.2 Install RStudio

To install RStudio, go to the following address: https://www.rstudio.com/products/rstudio/download/#download.

Then, pick RStudio Desktop.

1.1.3 Install Rtools

If at any point when using R you encounter an error mentioning Rtools, you can install it:

`https://cran.your_mirror/bin/windows/Rtools/`

Once again, you need to replace `your_mirror` by the mirror you selected, e.g., https://ftp.fau.de/cran/bin/windows/Rtools/.

1.1.4 Once installed

Once the software is installed, you can launch RStudio.

1.1.5 Troubleshooting

Sometimes you can't install packages on windows; you will get this kind of error: `‘lib = "C:/Program Files/R/R-4.0.2/library"‘ is not writable "Error in install.packages : unable to install packages"`

In such a case, close RStudio, and open it again in administrator mode (right-click on the icon of the file allowing to launch RStudio, choose "Run as administrator"). To avoid doing this every time, you can set the software to run in administrator mode by default (see this tutorial on cnet.com).

1.2 Mac OS Users

1.2.1 Install R

You need to install the **latest version of R** by going to the following address:

`https://cran.your_mirror/bin/macosx`

by replacing `your_mirror` by the mirror you picked. For example, if your mirror is "Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)": https://ftp.fau.de/cran/bin/macosx.

1.2.2 Install RStudio

To install RStudio, go to the following address: https://www.rstudio.com/products/rstudio/download/#download.

Then, pick RStudio Desktop.

1.2.3 Once installed

Once the software is installed, you can launch RStudio.

1.2.4 Troubleshooting

It is possible that some packages will not install and require the prior installation of libraries outside of R. You must go case by case and search for solutions on a web search engine. Think also of installing the xcode software (and running it at least once) if you have enough space on your computer. If you continue to use R, you will have to install this software sooner or later.

1.3 Linux Users

If you have a Linux distribution, the installation depends on the distribution of Linux that you have:

- Debian: https://cran.r-project.org/bin/linux/debian
- Fedora/Redhat: https://cran.r-project.org/bin/linux/fedora
- Ubuntu : https://cran.r-project.org/bin/linux/ubuntu

1.3.1 Install RStudio

To install RStudio, go to the following address: https://www.rstudio.com/products/rstudio/download/#download

Then, pick RStudio Desktop.

1.3.2 Once installed

Once the software is installed, you can launch RStudio.

Chapter 2

Gradient Descent

This chapter presents the gradient descent algorithm used to optimise functions. It begins with the vanilla version of the gradient descent algorithm, then moves to variants (stochastic gradient descent, batch gradient descent, mini-batch gradient descents). It end with two other techniques: Newton's method and coordinate descent algorithm.

2.1 Vanilla Gradient Descent

2.1.1 Concept

Let us consider a very general model:

$$y = m(\boldsymbol{X}) + \varepsilon,$$

where y is a variable to predict (or **target variable**, or **response variable**), $m(\cdot)$ is an unknown model, $\boldsymbol{X}$ is a set of p predictors (or **features**, or **inputs**, or **explanatory variables**) and ε is an error term.

Let us assume that the response variable is linearly dependent on the set of explanatory variables:

$$y = \boldsymbol{X}\beta + \varepsilon,$$

We do not know the true generating data process and only observe some realizations of y and $\boldsymbol{X}$ for n examples (or observations, or individuals). We need to make an assumption on the distribution of the error term to estimate the vector of coefficientsβ.

With **linear least squares**, we assume that the error term is normally distributed with zero mean and standard error σ. The vector of coefficients β can be estimated with **Ordinary Least Squares**

(OLS). The OLS estimates are such that they minimise the the sum of squared residuals, *i.e.*, the squared difference between the observed values y_i and the values predicted by the model $f(\boldsymbol{X_i})$:

$$RSS = \sum_{i=1}^{n} \left(y_i - f(\boldsymbol{X_i})\right)^2,$$

where $i = 1, \ldots, n$ denotes the examples (or individuals, or observations).

The problem boils down to estimating the coefficients of vector β which minimise an objective function:

$$\arg\min_{\beta} \sum_{i=1}^{n} \mathcal{L}\left(y_i, f(\boldsymbol{X_i})\right),$$

where here:

$$\mathcal{L}\left(y_i, f(\boldsymbol{X_i})\right) = \left(y_i - f(\boldsymbol{X_i})\right)^2$$

Here, with OLS, an analytical solution exists:

$$\hat{\beta} = \left(\boldsymbol{X}^t \boldsymbol{X}\right)^{-1} \boldsymbol{X}^t y.$$

In a more general case, if we do not assume that the response variable is linearly dependent on the set of explanatory variables, the aim is to find the solution $\hat{m}$ to the following optimization problem:

$$\text{minimise}_m \sum_{i=1}^{n} \mathcal{L}\left(y_i, m(\boldsymbol{X_i})\right).$$

The Gradient Descent algorithm is a popular technique that performs this kind of optimisation task, when the function to optimize is convex and differentiable.

2.1.2 A First Example in Dimension 1

If we know the functional form of the objective function, it is easy to find its minimum. As an illustration, consider the following function $\mathcal{L}(x) = 3x^2 - 2x + 5$.

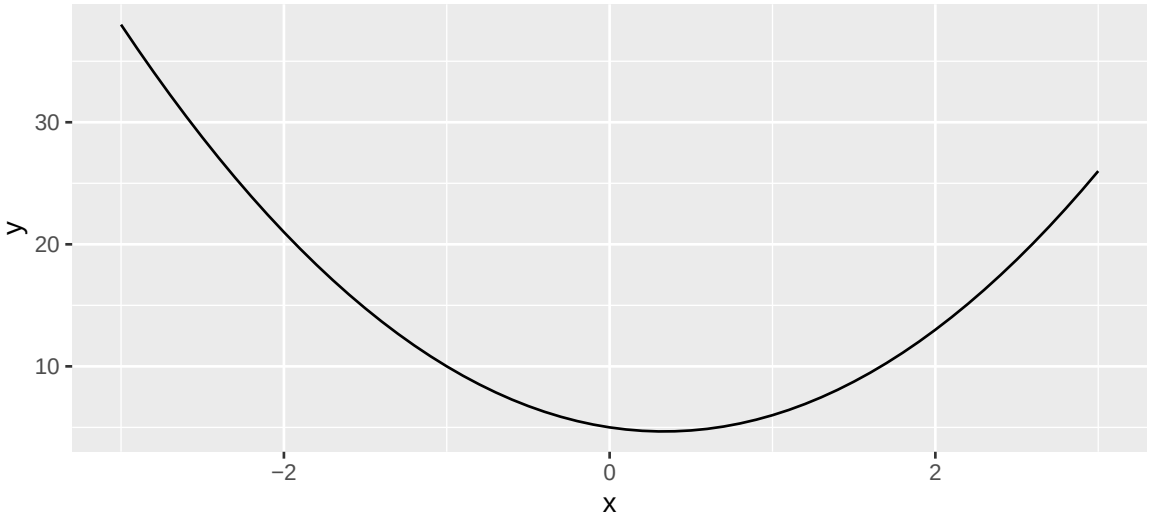

Figure 2.1: Minimising a simple loss function with a single input.

The value of x that minimises this function is obtained by canceling the first derivative of $\mathcal{L}(\cdot)$ with respect to x, *i.e.*:

$$\frac{\partial \mathcal{L}}{\partial x}(x) = 6x - 2 = 0,$$

which is $x = 1/3$:

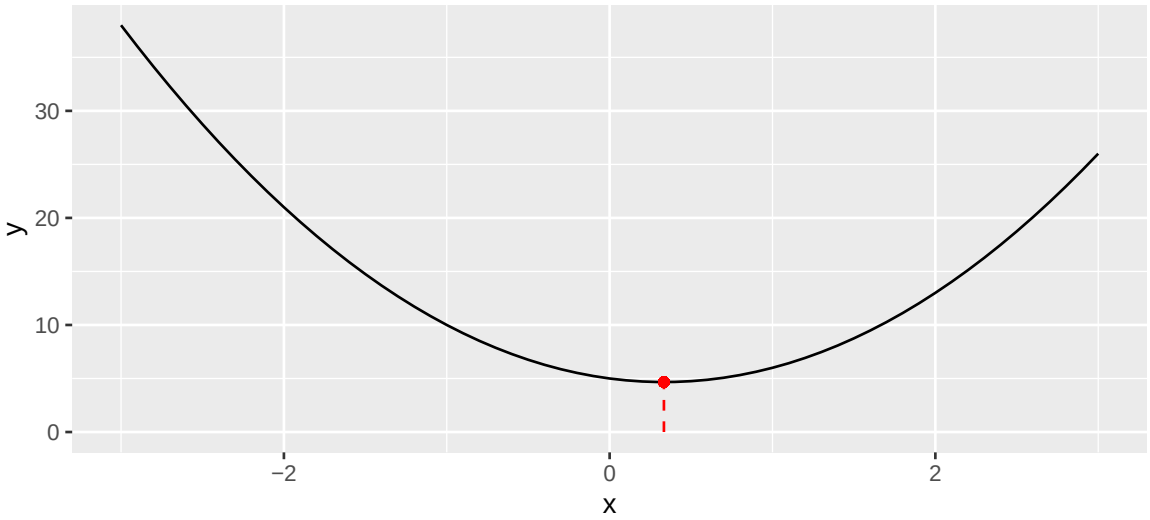

Figure 2.2: Function with a single input: minimum.

But with more complex functions, finding the minimum is not always feasible. Let us illustrate this with a simple example.

Let us consider the following function: $f(x) = (x+3) \times (x-2)^2 \times (x+1)$. The global minimum of that function is reached in $x = -1 - \sqrt{\frac{3}{2}}$. Let us generate some values from this process, for $x \in [-3, 3]$.

```
x <- seq(-3, 3, by = .1)
f <- function(x) (x+3)*(x-2)^2*(x+1)
y <- f(x)
df <- tibble(x = x, y = y)
df
```

```
## # A tibble: 61 x 2
##         x       y
##     <dbl>   <dbl>
##  1   -3      0
##  2   -2.9   -4.56
##  3   -2.8   -8.29
##  4   -2.7  -11.3
##  5   -2.6  -13.5
##  6   -2.5  -15.2
##  7   -2.4  -16.3
##  8   -2.3  -16.8
##  9   -2.2  -16.9
## 10   -2.1  -16.6
## # ... with 51 more rows
```

Here is a graph of this function.

```
ggplot(data = df, aes(x=x, y=y)) +
  geom_line() +
  geom_segment(
    data = tibble(x = -1-sqrt(3/2), xend = -1-sqrt(3/2),
                  y = -Inf, yend = f(-1-sqrt(3/2))),
    mapping = aes(x = x, y=y, xend=xend, yend = yend),
    colour = "red", linetype = "dashed") +
  geom_point(x=-1-sqrt(3/2), y = f(-1-sqrt(3/2)), colour = "red")
```

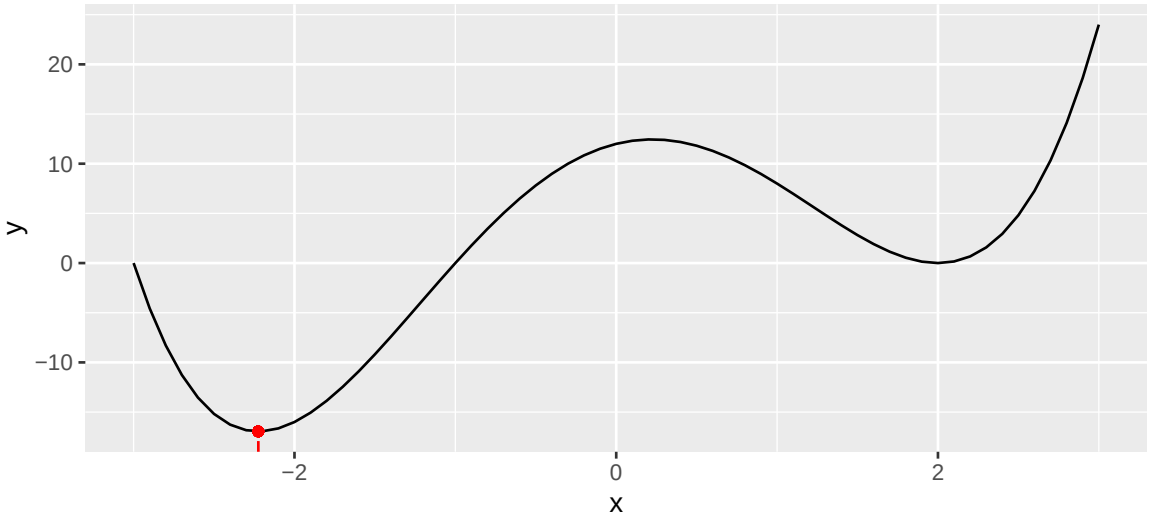

Figure 2.3: Function with a single input: a more complex function.

If we want to minimise this function using gradient descent, we can proceed as follows. In a first step, we start at a random point:

```
starting_value <- -.5
f(starting_value)
```

```
## [1] 7.8125
```

```
ggplot(data = df, aes(x=x, y=y)) +
  geom_line() +
  geom_point(x=starting_value, y = f(starting_value), colour = "red")
```

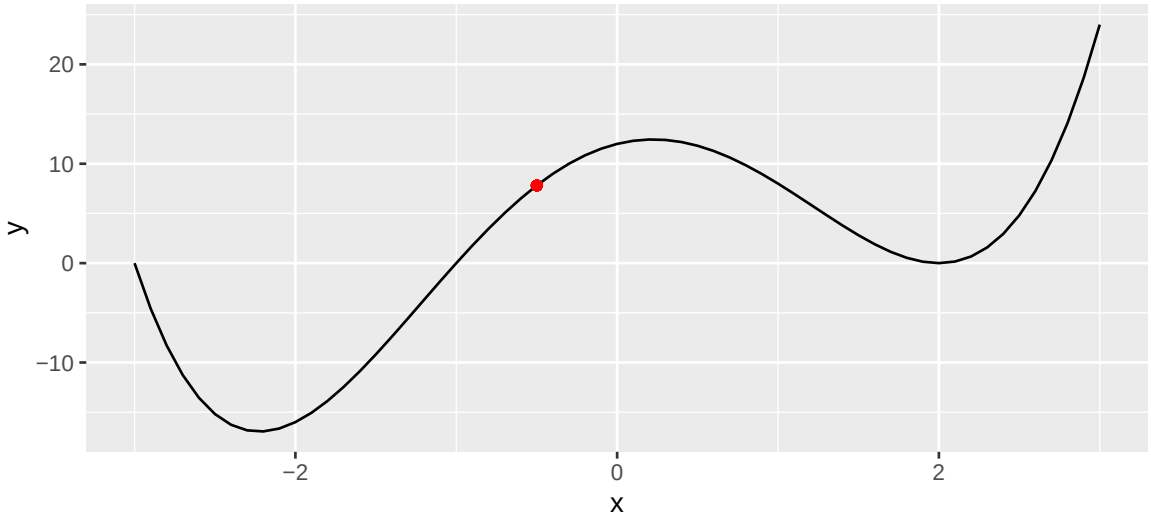

Figure 2.4: Function with a single input: start at a random point.

Then, from that point, we need to decide on two things so as to reduce the objective function:

1. in which direction to go next (left or right)
2. and how far we want to go.

To decide the direction, wan can compute the derivative of the function at this specific point of interest. The slope of the derivative will guide us:

- if it is positive: we need to shift to the left
- if it is negative: we need to shift to the right.

The first derivative can be obtained by numerical approximation, using the `grad()` function from {`numDeriv`}.

```
library(numDeriv)
grad <- grad(func = f, x = c(starting_value))
grad
```

```
## [1] 12.5
```

The intercept of the derivative can be computed as follows. We need it for the graph only, we could avoid computing it during the minimisation process.

```
(intercept <- -grad*starting_value + f(starting_value))
```

```
## [1] 14.0625
```

```
ggplot(data = df, aes(x=x, y=y)) +
  geom_line() +
  geom_point(x=starting_value, y = f(starting_value), colour = "red") +
  geom_abline(slope = grad, intercept = intercept, colour = "blue")
```

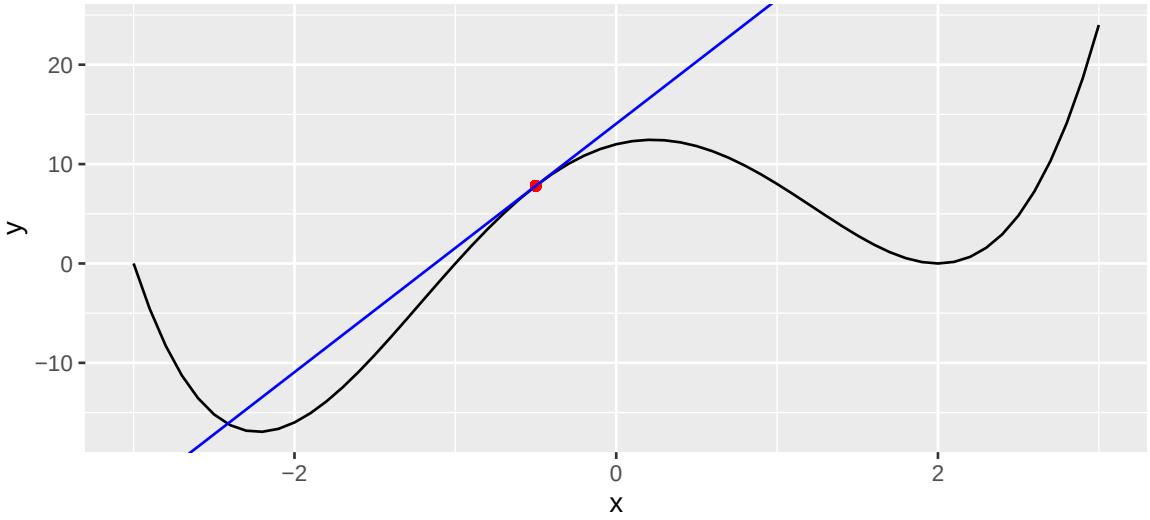

Figure 2.5: Compute the derivative of the function at that point.

Here, the slope is positive We thus need to go left We still need to decide how far we want to go, *i.e.*, we must decide the size of the step we will take. This step is called the **learning rate**. On the one hand, if this learning rate is too small, we increase the risk of ending up in a local minimum; on the other hand, if we pick a too large value for the learning rate, we face a risk of overshooting the minimum and keeping bouncing around a (local) minimum forever.

Let us first pick a small value for the learning rate:

```
learning_rate <- 10^-2
```

Once we have both the direction and the magnitude of the step, we can update our parameter:

```
(x_1 <- starting_value - learning_rate * grad)
```

```
## [1] -0.625
```

```
ggplot(data = df, aes(x=x, y=y)) +
  geom_line() +
  geom_point(x=starting_value, y = f(starting_value), colour = "red") +
  geom_point(x=x_1, y = f(x_1), colour = "green")
```

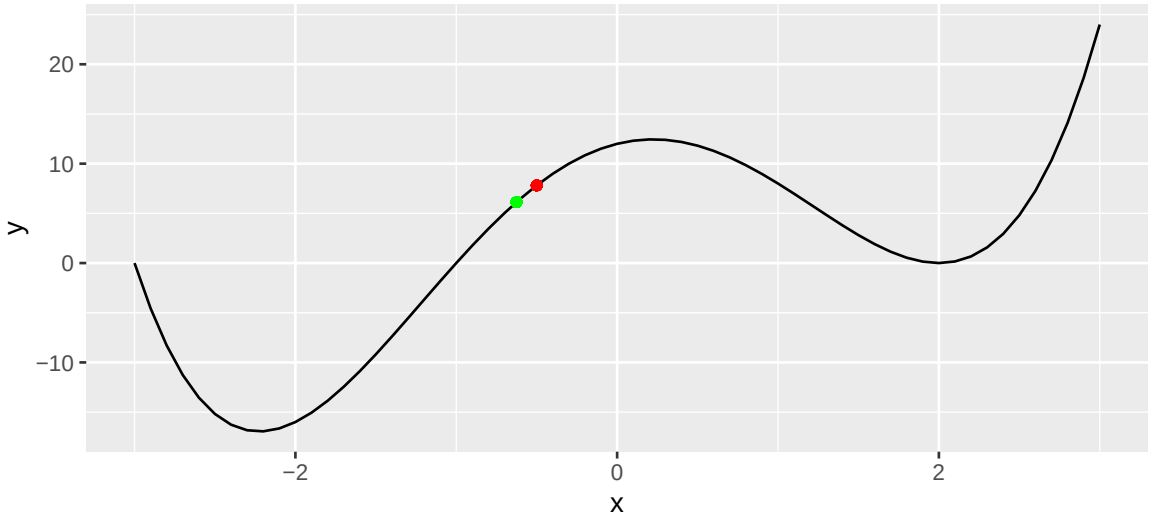

Figure 2.6: Second iteration.

Then, we can repeat the procedure mulltiple times. Let us do it through a loop. We will update our parameter from one iteration to the other and will stop either when a maximum number of iterations is reached or when the improvement (reduction in the objective function from one step to the next) is too small (below a threshold we will call `tolerance`).

```
nb_max_iter <- 100
tolerance <- 10^-5

x_1 <- -.5
# To keep track of the values through the iterations
```

```
x_1_values <- x_1
y_1_values <- f(x_1)
gradient_values <- NULL
intercept_values <- NULL

for(i in 1:nb_max_iter){
  # Steepest ascent:
  grad <- grad(func = f, x = c(x_1))

  intercept_value <- -grad*x_1 + f(x_1)
  # Keeping track
  gradient_values <- c(gradient_values, grad)
  intercept_values <- c(intercept_values, intercept_value)

  # Updating the value
  x_1 <- x_1 - learning_rate * grad
  y_1 <- f(x_1)

  # Keeping track
  x_1_values <- c(x_1_values, x_1)
  y_1_values <- c(y_1_values, y_1)

  # Stopping if no improvement (decrease of the cost function too small)
  if(abs(y_1_values[i] - y_1 < tolerance)) break

}
```

If we exit the loop before the maximum number of iterations has been reached, we can suppose we ended up in a (at least local) minimum. Otherwise, the algorithm did not converge.

```
i
```

```
## [1] 22
```

```
ifelse(i < nb_max_iter,
       "The algorithm converged.",
       "The algorithm did not converge.")
```

```
## [1] "The algorithm converged."
```

Let us put the computed derivative and intercept at each step in a tibble, to have a look at a graphical representation of the iterations:

```
df_plot <-
  tibble(x_1 = x_1_values[-length(x_1_values)],
         y = f(x_1),
         gradient = gradient_values,
         intercept = intercept_values
  )
df_plot
```

```
## # A tibble: 22 x 4
##        x_1       y gradient intercept
##      <dbl>   <dbl>    <dbl>     <dbl>
##   1 -0.5      7.81     12.5       14.1
##   2 -0.625    6.14     14.3       15.1
##   3 -0.768    3.97     16.0       16.3
##   4 -0.928    1.28     17.5       17.5
##   5 -1.10    -1.88     18.5       18.5
##   6 -1.29    -5.33     18.6       18.7
##   7 -1.47    -8.73     17.7       17.4
##   8 -1.65   -11.7      15.7       14.2
##   9 -1.81   -14.0      12.9        9.35
## 10 -1.94   -15.4       9.79       3.52
## # ... with 12 more rows
```

```
ggplot() +
  geom_line(data = df, aes(x = x, y = y)) +
  geom_point(data = df_plot, aes(x = x_1, y= f(x_1)),
             colour = "red", size = 2) +
  coord_cartesian(ylim = c(-20, 20))
```

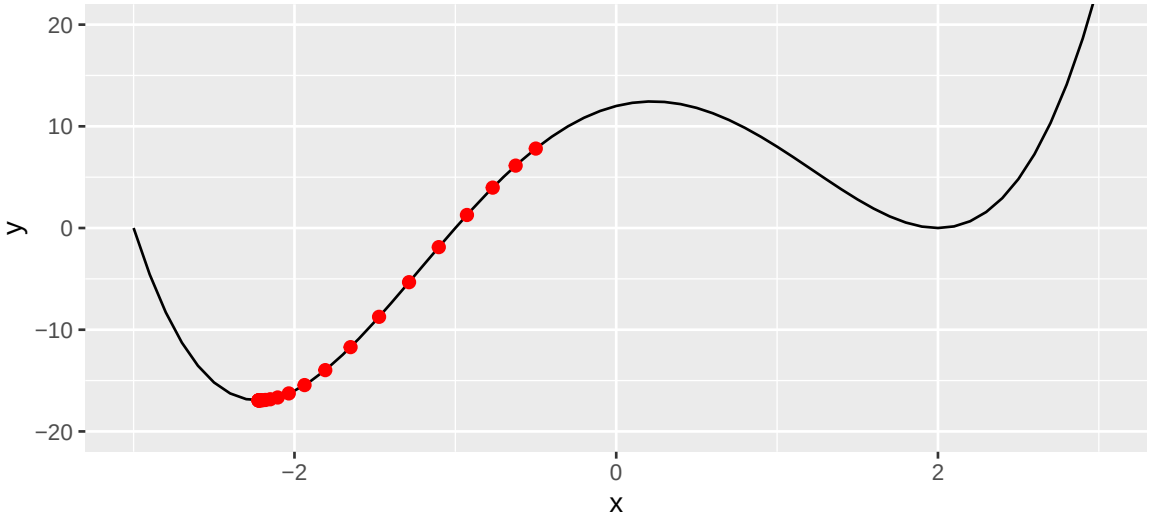

Figure 2.7: At the last step of the iteration process.

Now, let us run the same algorithm, but picking a larger value for the learning rate. Let us also increase the number of maximum iterations.

```
learning_rate <- 0.05
nb_max_iter <- 1000
tolerance <- 10^-5
# Starting value
x_1 <- -.5
# To keep track of the values through the iterations
x_1_values <- x_1
y_1_values <- f(x_1)
gradient_values <- NULL
intercept_values <- NULL

for(i in 1:nb_max_iter){
  # Steepest ascent:
  grad <- grad(func = f, x = c(x_1))

  intercept_value <- -grad*x_1 + f(x_1)
  # Keeping track
  gradient_values <- c(gradient_values, grad)
  intercept_values <- c(intercept_values, intercept_value)
```

```
  # Updating the value
  x_1 <- x_1 - learning_rate * grad
  y_1 <- f(x_1)

  # Keeping track
  x_1_values <- c(x_1_values, x_1)
  y_1_values <- c(y_1_values, y_1)

  # Stopping if no improvement (decrease of the cost function too small)
  if(abs(y_1_values[i] - y_1) < tolerance) break

}
```

```
i
```

```
## [1] 1000
```

```
ifelse(i < nb_max_iter,
       "The algorithm converged.",
       "The algorithm did not converge.")
```

```
## [1] "The algorithm did not converge."
```

```
df_plot <-
  tibble(x_1 = x_1_values[-length(x_1_values)],
         y = f(x_1),
         gradient = gradient_values,
         intercept = intercept_values
  )
df_plot
```

```
## # A tibble: 1,000 x 4
##       x_1       y gradient intercept
##     <dbl>   <dbl>    <dbl>     <dbl>
##   1 -0.5     7.81    12.5       14.1
##   2 -1.12   -2.29    18.6       18.6
##   3 -2.05  -16.4      6.35      -3.34
```

```
##   4 -2.37 -16.5      -6.60     -32.1
##   5 -2.04 -16.3       6.76      -2.52
##   6 -2.38 -16.4      -6.98     -33.0
##   7 -2.03 -16.2       7.12      -1.78
##   8 -2.38 -16.4      -7.32     -33.8
##   9 -2.02 -16.1       7.43      -1.15
##  10 -2.39 -16.3      -7.60     -34.5
## # ... with 990 more rows
```

```
ggplot() +
  geom_line(data = df, aes(x = x, y = y)) +
  geom_point(data = df_plot, aes(x = x_1, y= f(x_1)),
             colour = "red", size = .2) +
  coord_cartesian(ylim = c(-20, 20)) +
  labs(title = str_c("Step ", i))
```

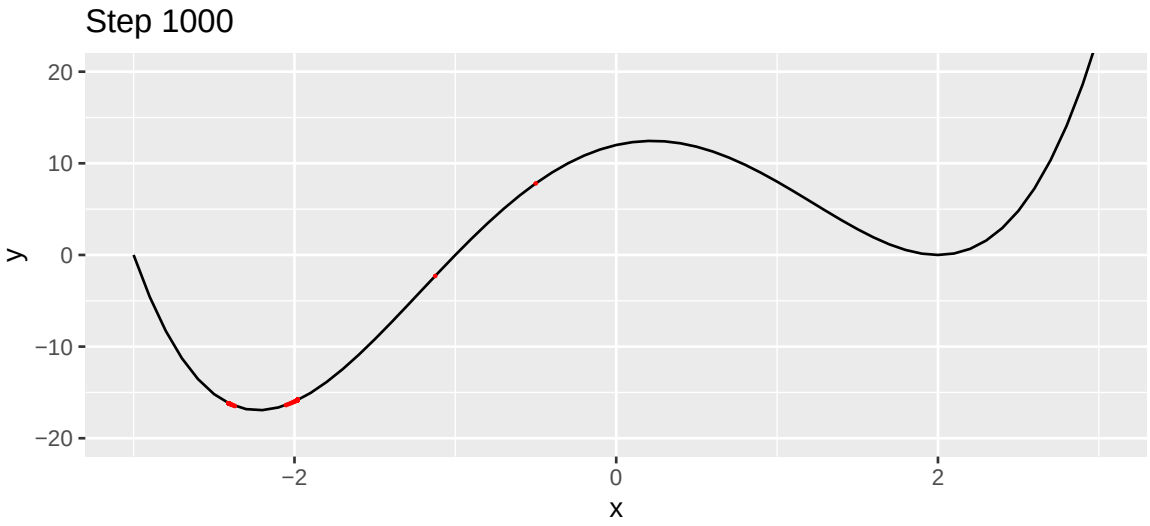

Figure 2.8: At the last step of the iteration process, with another starting value.

We can have a look at what happens iteratively with the following animated graph:

```
saveGIF({

  for(i in c(rep(1,5), 2:14, rep(15, 10))){
    p <-
```

```
      ggplot() +
      geom_line(data = df, aes(x = x, y = y)) +
      geom_point(data = df_plot %>% slice(i),
                 mapping = aes(x = x_1, y= f(x_1)),
                 colour = "red", size = 2) +
      geom_abline(data = df_plot %>% slice(i),
                  aes(slope = gradient, intercept = intercept),
                  colour = "blue") +
      coord_cartesian(ylim = c(-20, 20)) +
      labs(title = str_c("Step ", i))
    print(p)
  }

}, movie.name = "example_single_var_bounce.gif", interval = 0.5,
ani.width = 720, ani.height = 480)
```

The animated graph is not available in the PDF version of the hands-on session. Check to HTML version.

We jumped around the minimum and never reached it.

The algorithm is also sensitive to the starting point.

```
learning_rate <- 0.01
nb_max_iter <- 1000
tolerance <- 10^-5
# Starting value
x_1 <- .5
# To keep track of the values through the iterations
x_1_values <- x_1
y_1_values <- f(x_1)
gradient_values <- NULL
intercept_values <- NULL

for(i in 1:nb_max_iter){
  # Steepest ascent:
  grad <- grad(func = f, x = c(x_1))

  intercept_value <- -grad*x_1 + f(x_1)
```

```
  # Keeping track
  gradient_values <- c(gradient_values, grad)
  intercept_values <- c(intercept_values, intercept_value)

  # Updating the value
  x_1 <- x_1 - learning_rate * grad
  y_1 <- f(x_1)

  # Keeping track
  x_1_values <- c(x_1_values, x_1)
  y_1_values <- c(y_1_values, y_1)

  # Stopping if no improvement (decrease of the cost function too small)
  if(abs(y_1_values[i] - y_1) < tolerance) break

}
```

Let us check whether we converged:

```
i
```

```
## [1] 33
```

```
ifelse(i < nb_max_iter,
       "The algorithm converged.",
       "The algorithm did not converge.")
```

```
## [1] "The algorithm converged."
```

Yes. But let us look at where.

```
df_plot <-
  tibble(x_1 = x_1_values[-length(x_1_values)],
         y = f(x_1),
         gradient = gradient_values,
         intercept = intercept_values
```

```
  )
df_plot
```

```
## # A tibble: 33 x 4
##       x_1     y gradient intercept
##     <dbl> <dbl>    <dbl>     <dbl>
##   1 0.5    11.8     -4.50      14.1
##   2 0.545  11.6     -5.16      14.4
##   3 0.597  11.3     -5.89      14.8
##   4 0.656  10.9     -6.67      15.3
##   5 0.722  10.5     -7.49      15.9
##   6 0.797   9.87    -8.32      16.5
##   7 0.880   9.15    -9.12      17.2
##   8 0.972   8.28    -9.82      17.8
##   9 1.07    7.29   -10.4       18.4
##  10 1.17    6.20   -10.7       18.7
## # ... with 23 more rows
```

```
ggplot() +
  geom_line(data = df, aes(x = x, y = y)) +
  geom_point(data = df_plot, aes(x = x_1, y= f(x_1)), colour = "red") +
  coord_cartesian(ylim = c(-20, 20)) +
  labs(title = str_c("Step ", i))
```

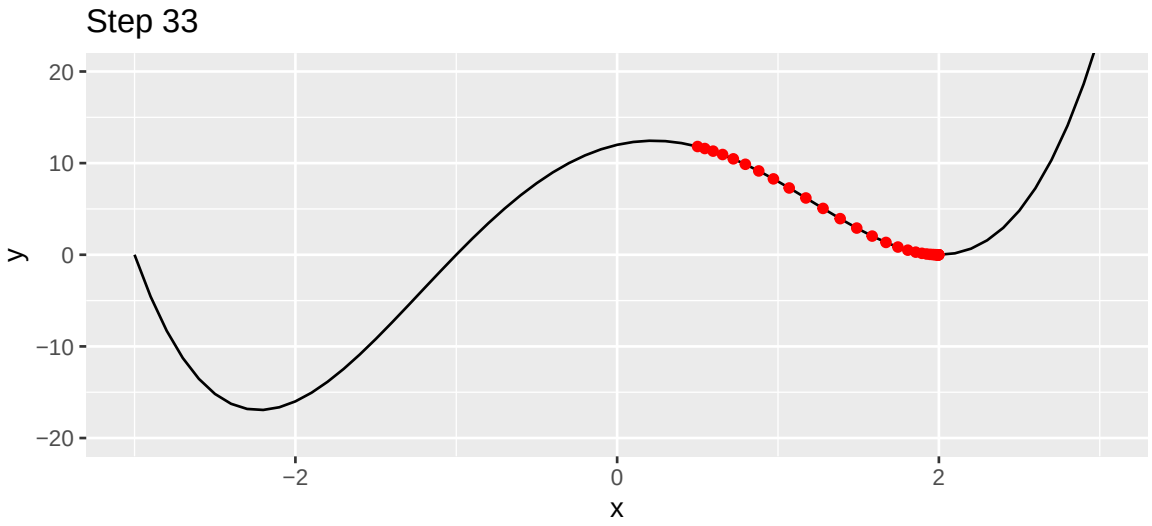

Figure 2.9: Ending up in a local minimum.

This time, we ended up in a local minimum.

Now let us increase the dimension of our problem, and move on to a function defined with two parameters. We will consider more afterwards, but then we will not be able to visualize as easily what happens using graphs.

2.1.3 Moving to Higher Dimensions Optimisation Problems

Let us consider the following data generating process: $f(x_1, x_2) = x_1^2 + x_2^2$.

```
x_1 <- x_2 <- seq(-2, 2, by = 0.3)
z_f <- function(x_1,x_2) x_1^2+x_2^2
z <- outer(x_1, x_2, z_f)
```

The representative surface of that function can be visualized as follows:

```
library(plot3D)
par(mar = c(1, 1, 1, 1))
flip <- 1 # 1 or 2
th = c(-300,120)[flip]
pmat <-
  persp3D(x = x_1, y = x_2, z = z, colkey=F, contour=T, ticktype = "detailed",
          asp = 1, phi = 30, theta = th, border = "grey10", alpha=.4,
          d = .8,r = 2.8,expand = .6,shade = .2,axes = T,box = T,cex = .1)
```

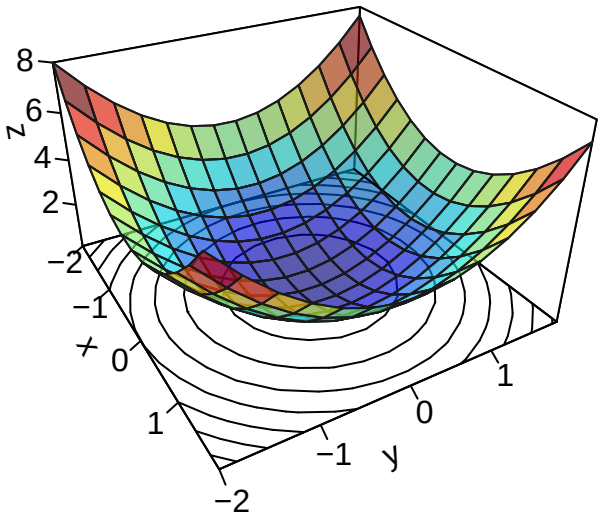

Figure 2.10: Surface of a function in $\mathbb{R}^2$.

Once again, we need to initialise the algorithm by picking starting values. Let us pick $\theta = (2, 2)$.

```
theta <- c(x_1 = 1.5, x_2 = 1.5)
```

Let us look at this point on the graph:

```
zz <- z_f(theta[["x_1"]], theta[["x_2"]])
new_point <- trans3d(theta[["x_1"]], theta[["x_2"]], zz,
                     pmat = pmat)


par(mar = c(1, 1, 1, 1))
flip <- 1 # 1 or 2
th = c(-300,120)[flip]
pmat <-
  persp3D(x = x_1, y = x_2, z = z, colkey=F, contour=T, ticktype = "detailed",
          asp = 1, phi = 30, theta = th, border = "grey10", alpha=.4,
          d = .8,r = 2.8,expand = .6,shade = .2,axes = T,box = T,cex = .1)
points(new_point,pch = 20,col = "red", cex=2)
```

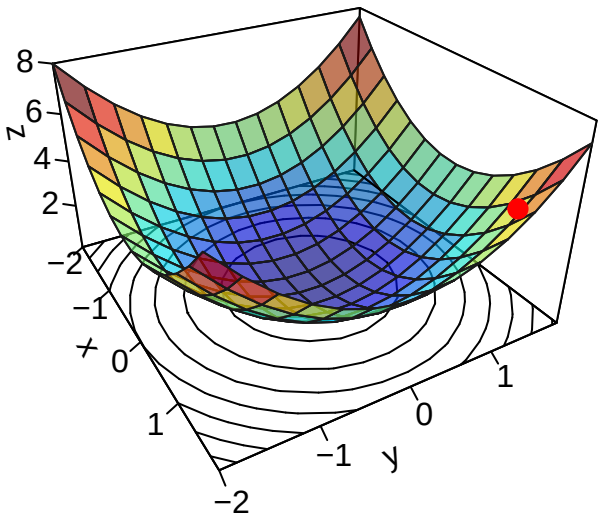

Figure 2.11: Starting point.

From that point, we need to decide the **direction** to go to and the **magnitude of the step** to take in that direction. The direction is obtained by computing the first derivative of the objective function $f(\cdot)$ with respect to each argument x_1 and x_2, at point θ. In other words, we need to

evaluate the gradient of the function at point θ.

$$\nabla f(\theta) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\theta) \\ \frac{\partial f}{\partial x_2}(\theta) \end{bmatrix}$$

The values will give us the steepest ascent. Once the learning rate is decided, we just need to update each argument by moving in the opposite direction of the steepest ascent. The updated value of the parameters after the end of the tth step will be as follows:

$$\begin{bmatrix} x_1^{(t+1)} \\ x_2^{(t+1)} \end{bmatrix} = \begin{bmatrix} x_1^{(t)} \\ x_2^{(t)} \end{bmatrix} - \eta \begin{bmatrix} \frac{\partial f}{\partial x_1}(x_1^{(t)}, x_2^{(t)}) \\ \frac{\partial f}{\partial x_2}(x_1^{(t)}, x_2^{(t)}) \end{bmatrix},$$

where $\begin{bmatrix} x_1^{(t+1)} \\ x_2^{(t+1)} \end{bmatrix}$ is the updated vector of parameters, $\begin{bmatrix} x_1^{(t)} \\ x_2^{(t)} \end{bmatrix}$ is the current value of the vector of parameters, $\eta \in \mathbb{R}^+$ is the learning rate, and $\begin{bmatrix} \frac{\partial f}{\partial x_1}(x_1^{(t)}, x_2^{(t)}) \\ \frac{\partial f}{\partial x_2}(x_1^{(t)}, x_2^{(t)}) \end{bmatrix}$ is the gradient of the function at point $\theta = \left(x_1^{(t)}, x_2^{(t)}\right)$.

In a more general context, when at a point $\theta \in \mathbb{R}^p$, at any step $t \leq 0$, the gradient descent algorithm tries to move in a direction $\delta\theta$ such that $\mathcal{L}\left(\theta^{(t)} + \delta\theta\right) < \mathcal{L}\left(\theta^{(t)}\right)$. The choice of $\delta\theta$ is made such that $\delta\theta = -\eta \cdot \nabla\mathcal{L}\left(\theta^{(t)}\right)$:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \nabla\mathcal{L}\left(\theta^{(t)}\right)$$

Note : the choice of $\delta\theta$ is made based on the first-order Taylor approximation. If $\mathcal{L} : \mathbb{R}^p \to \mathbb{R}$ is differentiable at point θ, for any small change $\delta\theta$, the best linear approximation to $\mathcal{L}$ is given by:

$$\mathcal{L}(\theta + \delta\theta) = \mathcal{L}(\theta) + \nabla\mathcal{L}(\theta)^\top \delta\theta + \mathcal{O}(\|\delta^2\theta\|)$$

We want to get :

$$\nabla\mathcal{L}(\theta)^\top \delta\theta < 0$$

$$\begin{aligned} &\mathcal{L}(\theta + \delta\theta) < \mathcal{L}(\theta) \\ \Leftrightarrow &\mathcal{L}(\theta) + \nabla\mathcal{L}(\theta)^\top \delta\theta < \mathcal{L}(\theta) \\ \Leftrightarrow &\nabla\mathcal{L}(\theta)^\top \delta\theta < 0. \end{aligned}$$

This dot product takes its minimum value when $\delta\theta = -\nabla\mathcal{L}(\theta)$. This gives the direction to go to. This does not tell us, however, the magnitude of the step that we should take.

Let us rewrite our function $f(\cdot)$ so that we can calculate its gradient by numerical approximation at a given point θ using `grad()` from {`numDeriv`}.

```
z_f_to_optim <- function(theta){
  x_1 <- theta[["x_1"]]
  x_2 <- theta[["x_2"]]
  x_1^2 + x_2^2
}
```

Let us set a learning rate:

```
learning_rate <- 10^-2
```

The steepest ascent can be obtained as follows:

```
grad <- grad(func = z_f_to_optim, x = theta)
grad
```

```
## [1] 3 3
```

The values can then be updated:

```
updated_x_1 <- theta[["x_1"]] - learning_rate * grad[1]
updated_x_2  <- theta[["x_2"]] - learning_rate * grad[2]
updated_theta <- c(x_1 = updated_x_1, x_2 = updated_x_2)
updated_theta
```

```
##   x_1   x_2
## 1.47  1.47
```

On the graph:

```
par(mar = c(1, 1, 1, 1))
flip <- 1 # 1 or 2
th = c(-300,120)[flip]
pmat <-
  persp3D(x = x_1, y = x_2, z = z, colkey=F, contour=T,
          ticktype = "detailed", asp = 1, phi = 30, theta = th,
          border = "grey10", alpha=.4, d = .8,r = 2.8,
          expand = .6,shade = .2,axes = T,box = T,cex = .1)
updated_zz <- z_f(updated_theta[["x_1"]], updated_theta[["x_2"]])
new_point_2 <- trans3d(updated_theta[["x_1"]],
                       updated_theta[["x_2"]],
                       updated_zz,
                       pmat = pmat)
points(new_point,pch = 20,col = "red", cex=2)
points(new_point_2,pch = 20,col = "darkgreen", cex=2)
```

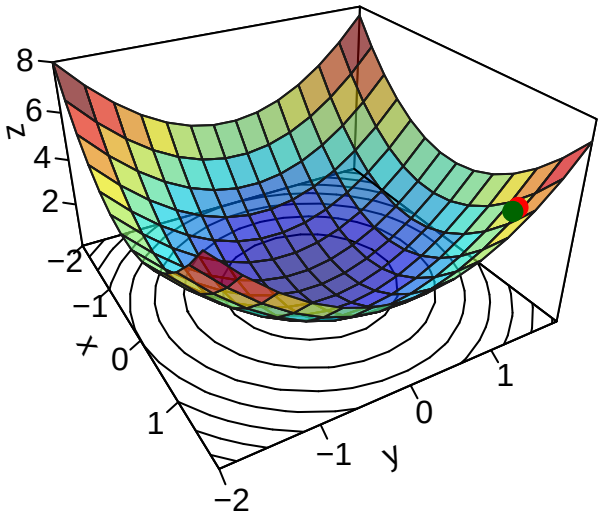

Figure 2.12: Updated value after the first iteration.

Then, we need to repeat the updating process. The full algorithm can be written this way:

```
learning_rate <- 10^-1
nb_max_iter <- 100
tolerance <- 10^-5

# Starting values
theta <- c(x_1 = 1.5, x_2 = 1.5)

# To keep track of what happens at each iteration
theta_values <- list(theta)
y_values <- z_f_to_optim(theta)

for(i in 1:nb_max_iter){
  # Steepest ascent
  grad <- grad(func = z_f_to_optim, x = theta)

  # Updating the parameters
  updated_x_1 <- theta[["x_1"]] - learning_rate * grad[1]
  updated_x_2  <- theta[["x_2"]] - learning_rate * grad[2]
  theta <- c(x_1 = updated_x_1, x_2 = updated_x_2)

  # Keeping track
```

```
  theta_values <- c(theta_values, list(theta))

  # Checking for improvement
  y_updated <- z_f_to_optim(theta)
  y_values <- c(y_values, y_updated)
  if(abs(y_values[i] - y_updated) < tolerance) break

}
```

Let us check at which iteration the algorithm stopped:

```
i
```

```
## [1] 28
```

```
ifelse(i < nb_max_iter,
       "The algorithm converged.",
       "The algorithm did not converge.")
```

```
## [1] "The algorithm converged."
```

Graphically:

```
par(mar = c(1, 1, 1, 1))
flip <- 1 # 1 or 2
th = c(-300,120)[flip]
pmat <-
  persp3D(x = x_1, y = x_2, z = z, colkey=F, contour=T,
          ticktype = "detailed", asp = 1, phi = 30, theta = th,
          border = "grey10", alpha=.4, d = .8,r = 2.8,
          expand = .6,shade = .2,axes = T,box = T,cex = .1)

xx <- map_dbl(theta_values, "x_1")
yy <- map_dbl(theta_values, "x_2")
zz <- y_values
new_point <- trans3d(xx,yy,zz,pmat = pmat)
```

```
lines(new_point,pch = 20,col = "red", cex=2, lwd=2)
points(new_point,pch = 20,col = "red", cex=2)
```

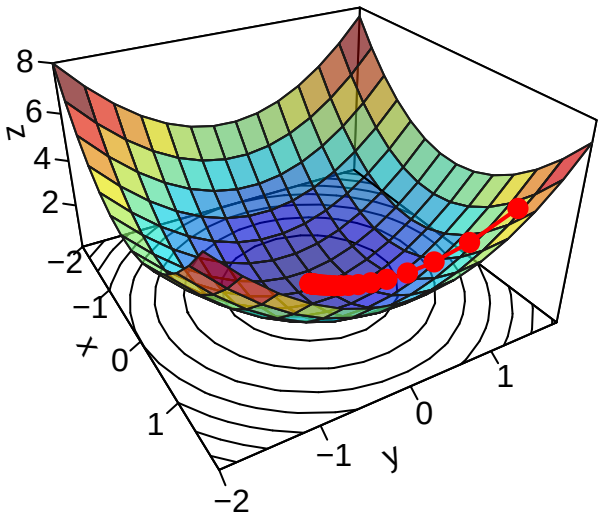

Figure 2.13: At the end of the iterative process.

To have a look, step by step:

```
library(animation)
saveGIF({

  for(j in c(rep(1,5), 2:(i-1), rep(i, 10))){

    par(mar = c(1, 1, 1, 1))
    flip <- 1 # 1 or 2
    th = c(-300,120)[flip]
    pmat <-
      persp3D(x = x_1, y = x_2, z = z, colkey=F, contour=T,
              ticktype = "detailed", asp = 1, phi = 30, theta = th,
              border = "grey10", alpha=.4, d = .8,r = 2.8,
              expand = .6,shade = .2,axes = T,box = T,cex = .1)

    xx <- map_dbl(theta_values, "x_1")[1:j]
    yy <- map_dbl(theta_values, "x_2")[1:j]
    zz <- y_values[1:j]
```

```
    new_point <- trans3d(xx,yy,zz,pmat = pmat)
    lines(new_point,pch = 20,col = "red", cex=2, lwd=2)
    points(new_point,pch = 20,col = "red", cex=2)
  }

}, movie.name = "descent_2D_sphere.gif", interval = 0.01,
ani.width = 720, ani.height = 480)
```

The animated graph is not available in the PDF version of the hands-on session. Check to HTML version.

With this example, we quickly converged to a solution. But the surface can be a bit more complex. Let us consider another data generating process, Mishra's Bird function:

$$f(x_1, x_2) = \sin(x_2) * \exp(1 - \cos(x_1))^2 + \cos(x_1) * \exp(1 - \sin(x_2))^2 + (x_1 - x_2)^2.$$

First, let us generate some data:

```
x_1 <- seq(-6.5, 0, by = 0.3)
x_2 <- seq(-10, 0, by = 0.3)
z_f <- function(x_1,x_2){
  sin(x_2)*exp(1-cos(x_1))^2 + cos(x_1)*exp(1-sin(x_2))^2 + (x_1-x_2)^2
}
z <- outer(x_1, x_2, z_f)
```

A graphical representation of the surface:

```
par(mar = c(1, 1, 1, 1))
flip <- 2 # 1 or 2
theta = c(-300,120)[flip]
pmat <-
  persp3D(x = x_1, y = x_2, z = z, colkey=F, contour=T,
          ticktype = "detailed", asp = 1, phi = 30, theta = th,
          border = "grey10", alpha=.4, d = .8,r = 2.8,
          expand = .6,shade = .2,axes = T,box = T,cex = .1)
```

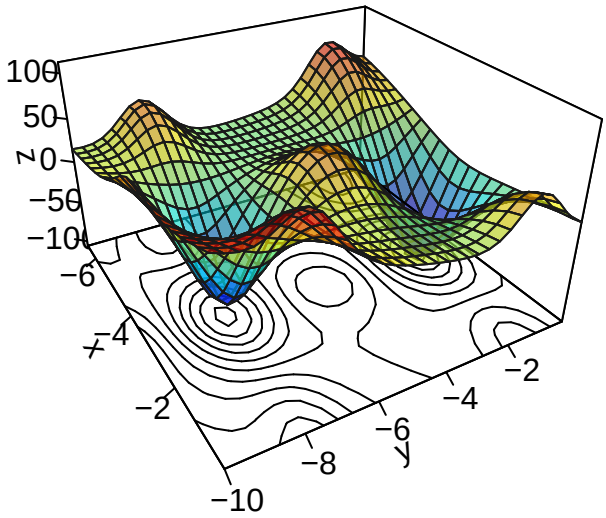

Figure 2.14: A more complex function in $\mathbb{R}^2$.

The function that needs to be optimized need to be rewritten so that the first argument is the vector of parameters over which minimisation is to take place.

```
z_f_to_optim <- function(theta){
  x_1 <- theta[1]
  x_2 <- theta[2]
  sin(x_2) * exp(1-cos(x_1))^2 + cos(x_1) * exp(1-sin(x_2))^2 +
    (x_1-x_2)^2
}
```

Let us create a function that uses the gradient descent algorithm to try find the minimum.

```
#' @param par Initial values for the parameters to be optimized over.
#' @param fn A function to be minimized, with first argument the vector
#'           of parameters over which minimisation is to take place.
#'           It should return a scalar result.
#' @param learning_rate Learning rate.
#' @param nb_max_iter The maximum number of iterations (default to 100).
#' @param tolerance The absolute convergence tolerance (default to 10^-5).
gradient_descent <- function(par, fn, learning_rate,
                             nb_max_iter = 100, tolerance = 10^-5){

  # To keep track of what happens at each iteration
```

```
  par_values <- list(par)
  y_values <- fn(par)

  for(i in 1:nb_max_iter){
    # Steepest ascent
    grad <- grad(func = fn, x = par)

    # Updating the parameters
    par <- par - learning_rate * grad

    # Keeping track
    par_values <- c(par_values, list(par))

    # Checking for improvement
    y_updated <- fn(par)
    y_values <- c(y_values, y_updated)
    rel_diff <- abs(y_values[i] - y_updated)
    if(rel_diff < tolerance) break

  }

  # Has the algorithm converged?
  convergence <- i < nb_max_iter | (rel_diff < tolerance)

  structure(
    list(
      par = par,
      value = y_updated,
      pars = do.call("rbind", par_values),
      values = y_values,
      convergence = convergence,
      nb_iter = i,
      nb_max_iter = nb_max_iter,
      tolerance = tolerance
    ))

}
```

Now this optimisation function can be called. Let us start at $\theta = (-6, -2)$, and try to find the minimum with a learning rate of 10^{-2} over at most 100 iterations.

```
res_optim <-
  gradient_descent(par = c(-6, -2), fn = z_f_to_optim,
                   learning_rate = 10^-2,
                   nb_max_iter = 100,
                   tolerance = 10^-5)
```

Let us check whether the algorithm converged:

```
res_optim$convergence
```

```
## [1] TRUE
```

```
res_optim$nb_iter
```

```
## [1] 41
```

The algorithm has converged. Let us look at the point we ended up with:

```
res_optim$par
```

```
## [1] -3.122755 -1.589316
```

```
res_optim$value
```

```
## [1] -106.7877
```

And graphically:

```
par(mar = c(1, 1, 1, 1))
pmat <-
  persp3D(x = x_1, y = x_2, z = z, colkey=F, contour=T,
          ticktype = "detailed", asp = 1, phi = 30, theta = 120,
          border = "grey10", alpha=.4, d = .8,r = 2.8,
          expand = .6,shade = .2,axes = T,box = T,cex = .1)
```

```
xx <- res_optim$pars[,1]
yy <- res_optim$pars[,2]
zz <- res_optim$values
new_point <- trans3d(xx,yy,zz,pmat = pmat)
lines(new_point,pch = 20,col = "red", cex=2, lwd=2)
points(new_point,pch = 20,col = "red", cex=2)
```

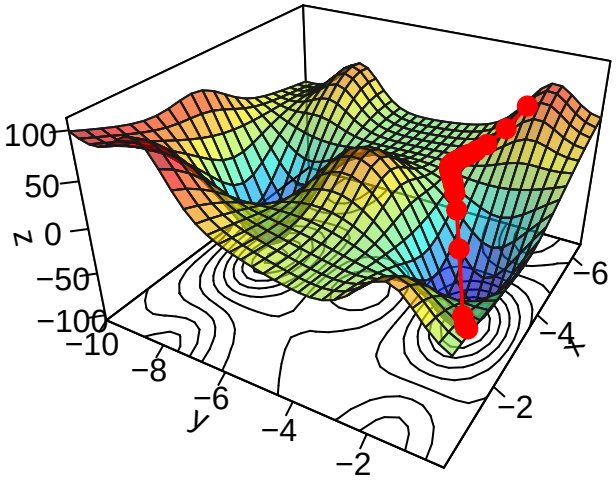

Figure 2.15: Interative process We end up in a local minimum.

```
saveGIF({
  for(j in c(rep(1,5), 2:(res_optim$nb_iter-1), rep(res_optim$nb_iter, 10))){

    par(mar = c(1, 1, 1, 1))
    pmat <-
      persp3D(x = x_1, y = x_2, z = z, colkey=F, contour=T,
              ticktype = "detailed", asp = 1, phi = 30, theta = 120,
              border = "grey10", alpha=.4, d = .8,r = 2.8,
              expand = .6, shade = .2,axes = T,box = T,cex = .1,
              main = str_c("Step ", j))

    xx <- res_optim$pars[1:j,1]
    yy <- res_optim$pars[1:j,2]
    zz <- res_optim$values[1:j]
    new_point <- trans3d(xx,yy,zz,pmat = pmat)
```

```
    lines(new_point,pch = 20,col = "red", cex=2, lwd=2)
    points(new_point,pch = 20,col = "red", cex=2)
  }

}, movie.name = "descent_3D_Mishra.gif", interval = 0.01,
ani.width = 720, ani.height = 480)
```

The animated graph is not available in the PDF version of the hands-on session. Check to HTML version.

We ended up in the minimum.

Another way to look at the gradient descent is through the following contour plot. At each iteration, we decide in which direction to go:

```
d <- tibble(x = res_optim$pars[,1],
            y = res_optim$pars[,2],
            z = res_optim$values)
contour2D(x=x_1, y=x_2, z=z, colkey=F, main="Contour plot",
          xlab="x_1", ylab="x_2")
points(x=d$x, y=d$y, t="p", pch=19, col = "red")
for(k in 2:(nrow(d+1)))
  segments(x0 = d$x[k-1], y0 = d$y[k-1],
           x1 = d$x[k], y1 = d$y[k], col= 'red')
```

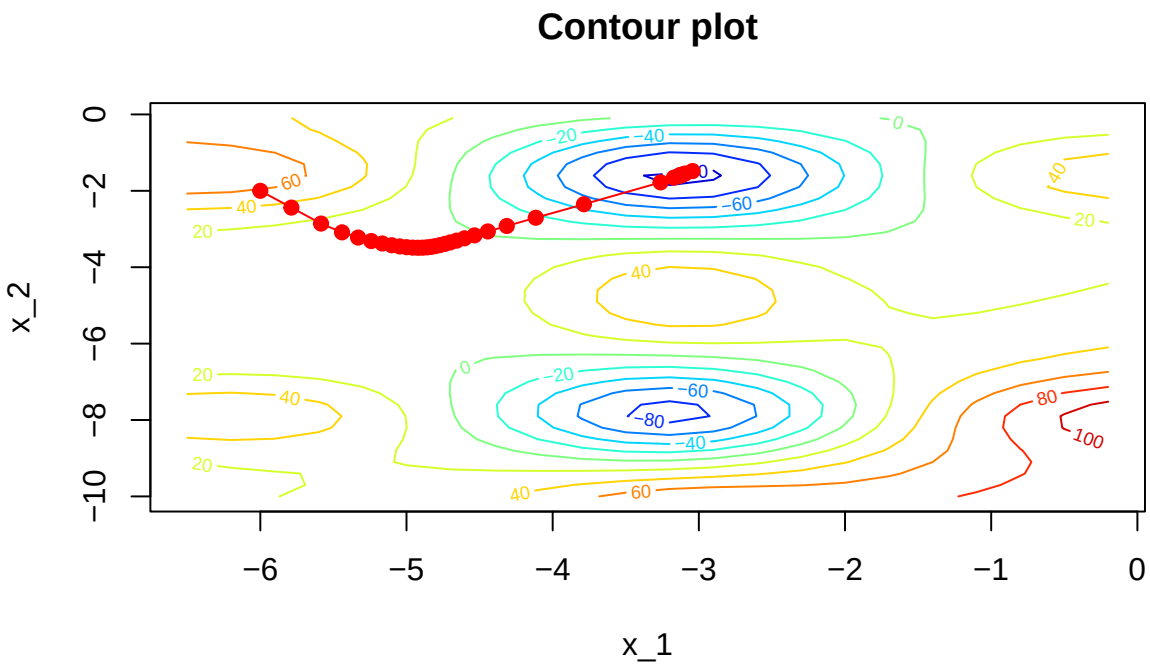

Figure 2.16: Another grapghical representation: contour plot.

Let us change the starting point to begin with $\theta = (-6, -4)$ (let us also increase the maximum number of iterations).

```
res_optim <-
  gradient_descent(par = c(-6, -4), fn = z_f_to_optim,
                   learning_rate = 10^-2,
                   nb_max_iter = 1000,
                   tolerance = 10^-5)
```

Let us check whether the algorithm converged:

```
res_optim$convergence
```

```
## [1] TRUE
```

```
res_optim$nb_iter
```

```
## [1] 141
```

The algorithm has also converged. Let us look at the point we ended up with:

```
par(mar = c(1, 1, 1, 1))
pmat <-
  persp3D(x = x_1, y = x_2, z = z, colkey=F, contour=T,
          ticktype = "detailed", asp = 1, phi = 30, theta = 120,
          border = "grey10", alpha=.4, d = .8,r = 2.8,
          expand = .6, shade = .2,axes = T,box = T,cex = .1)

xx <- res_optim$pars[,1]
yy <- res_optim$pars[,2]
zz <- res_optim$values
new_point <- trans3d(xx,yy,zz,pmat = pmat)
lines(new_point,pch = 20,col = "red", cex=2, lwd=2)
points(new_point,pch = 20,col = "red", cex=2)
```

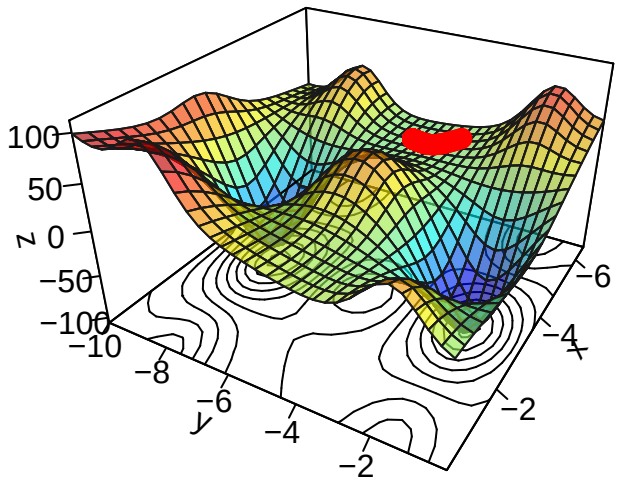

Figure 2.17: Getting stuck in a plateau.

We reached a local minimum.

```
d <- tibble(x = res_optim$pars[,1],
            y = res_optim$pars[,2],
```

```
            z = res_optim$values)
contour2D(x=x_1, y=x_2, z=z, colkey=F, main="Contour plot",
          xlab="x_1", ylab="x_2")
points(x=d$x, y=d$y, t="p", pch=19, col = "red")
for(k in 2:(nrow(d+1)))
  segments(x0 = d$x[k-1], y0 = d$y[k-1], x1 = d$x[k],
           y1 = d$y[k], col= 'red')
```

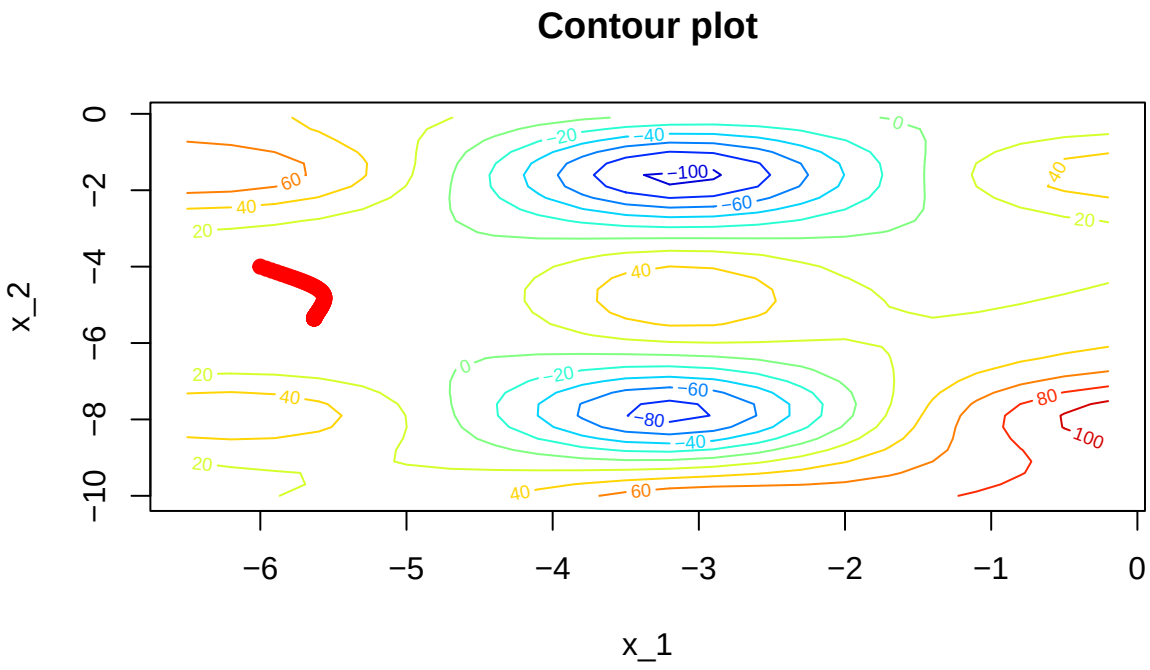

Figure 2.18: Contour plot: getting stuck in a plateau.

2.1.4 Case Study: Linear Regression

Let us generate some data.

$$y_i = 3x_i - 2 + \varepsilon_i, \quad i = 1, \ldots, n,$$

where ε is normally distributed with zero mean and variance $\sigma^2 = 4$.

```
set.seed(123)
# Number of observations
```

```
n <- 50
# x randomly drawn from a continuous uniform distribution with bounds [0,10]
x <- runif(min = 0, max = 10, n = n)
# Error term from Normal distribution with zero mean and variance 4
error <- rnorm(n = n, mean = 0, sd = 2)
# Response variable
beta_0 <- 3
beta_1 <- -2
y <- beta_0*x + beta_1 + error
```

Let us put the data in a table:

```
df <- tibble(x = x, y = y)
df
```

```
## # A tibble: 50 x 2
##         x       y
##     <dbl>   <dbl>
##  1 2.88    3.25
##  2 7.88   23.3
##  3 4.09   10.6
##  4 8.83   22.2
##  5 9.40   28.7
##  6 0.456   0.220
##  7 5.28   13.3
##  8 8.92   26.6
##  9 5.51   16.3
## 10 4.57   13.3
## # ... with 40 more rows
```

```
ggplot(data = df, aes(x = x, y = y)) + geom_point() +
  geom_abline(slope = 3, intercept = -2) +
  labs(title = "y = f(x)")
```

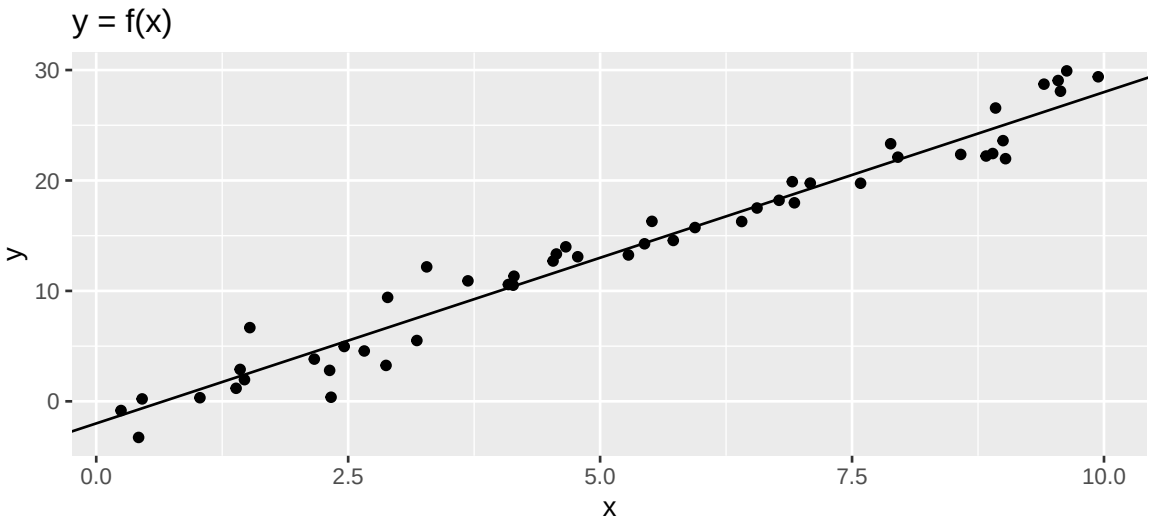

Figure 2.19: Data Generating Process and synthetic data.

Now, let us suppose that we do not know anymore the parameters β_0 and β_1. The only things we assume are that there exists a linear relationship between y and x and that the error term is normally distributed with zero mean and (unknown) variance σ^2.

In other words, we would like to estimate the following model:

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i, \quad i = 1, \ldots, n,$$

where $\varepsilon \, \mathcal{N}(0, \sigma^2)$, and where β_0, β_1 (and σ^2) are unknown and need to be estimated.

We would like to obtain estimates of β_0 and β_1 such that the loss function (our objective function) is the smallest. The loss function we will use is the mean squared error:

$$\mathcal{L}(\beta) = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \hat{y}_i)\right)^2,$$

where $\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i$, and where $\hat{\beta}_0$ and $\hat{\beta}_1$ are the estimates of β_0 and β_1, respectively.

We will use the gradient descent algorithm to estimate these parameters so as to minimise this loss function.

The function to optimise:

```
obj_function <- function(theta, y){
  y_pred <- theta[1] + theta[2]*x
  mean((y - y_pred)^2)
}
```

We need some starting values for β_0 and β_1:

```
beta <- c(0, 0)
```

Let us keep track of the updated values throughout the iterations:

```
beta_values <- beta
mse_values  <- NULL
```

We need to pick a learning rate:

```
learning_rate <- 10^-2
```

Let us set a maximum number of iterations:

```
nb_max_iter <- 1000
```

And an absolute tolerance:

```
abstol <- 10^-5
```

```
for(i in 1:nb_max_iter){
  # Predctions with the current values:
  y_pred <- beta[1] + beta[2]*x

  # Just for keeping track
  mse <- mean((y - y_pred)^2)
  mse_values <- c(mse_values, mse)

  gradient <- grad(func = obj_function, x = beta, y=y)

  # We could also use the exact expression here:
  # deriv_loss_beta <- -2/n * sum( y - y_pred )
  # deriv_loss_beta_0 <- -2/n * sum( x*(y - y_pred) )
```

```
  # Updating the value
  beta <- beta - learning_rate * gradient

  # Keeping track of the changes
  beta_values <- rbind(beta_values, beta)


  if(i>1){
    rel_diff <- abs(mse_values[i] - mse_values[i-1])
    if(rel_diff < abstol) break
  }

}
```

Has the algorithm converged?

```
print(str_c("Number of iterations: ", i))
```

```
## [1] "Number of iterations: 790"
```

```
convergence <- i < nb_max_iter | (rel_diff < abstol)
convergence
```

```
## [1] TRUE
```

The estimated values:

```
beta
```

```
## [1] -2.218623  3.066613
```

For comparison, the OLS estimates are as follows:

```
lm(y~x)
```

```
##
## Call:
## lm(formula = y ~ x)
##
## Coefficients:
## (Intercept)            x
##      -2.285        3.076
```

We can notice that the MSE quickly converges to the variance of the error:

```
ggplot(data = tibble(mse = mse_values) %>%
         mutate(iteration = row_number()) %>%
         filter(iteration > 1),
       aes(x = iteration, y = mse)) +
  geom_line()
```

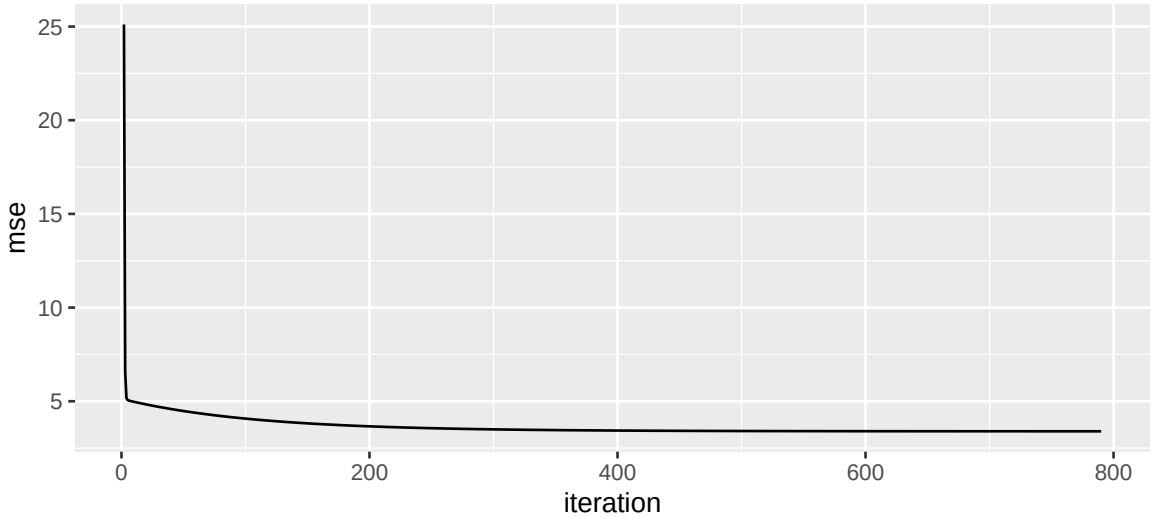

Figure 2.20: Quick convergence of the MSE to the variance of the error.

We can have a look at the updates throughout the iterative process:

```
as_tibble(beta_values, .name_repair = "minimal") %>%
  magrittr::set_colnames(c("beta_0", "beta_1")) %>%
```

```
  mutate(iteration = row_number()) %>%
  ggplot(data = ., aes(x = beta_0, y = beta_1)) +
  geom_point(aes(colour = iteration), size = .1)
```

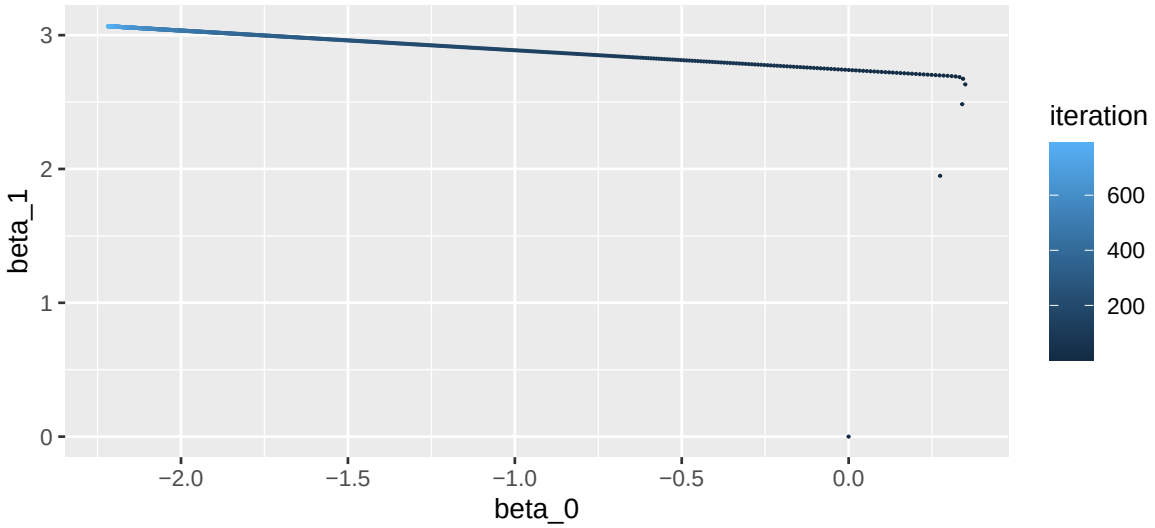

Figure 2.21: Updated values at each iteration.

2.2 Variants of the Gradient Descent Algorithm

So far, we have estimated the p parameters that minimise an objective function $\mathcal{L}(\theta)$, where θ is a vector of the p parameters to be estimated.

We have seen that the gradient descent algorithm updates the value of the ith parameter using the following rule:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \nabla\mathcal{L}(\theta^{(t)})$$

In the previous example, to compute the gradient of the objective function $\mathcal{L}$, we have used the **whole dataset**. The learning rate eta was a constant. In this section, we will consider different ways of updating the parameters. First, we will focus on the frequency of updates and on the samples used to update the parameters. Then, we will have a glance at ways used to make the learning rate vary along the iteration process.

Before jumping to those aspects, let us sum up how the gradient descent algorithm works:

Gradient Descent Algorithm

1. Randomly pick starting values for the parameters
2. Compute the gradient of the objective function at the current value of the parameters using all the observations from the training sample
3. Update the parameters
4. Repeat from step 2 until a fixed number of iteration or until convergence.

2.2.1 Frequency of Updates & Samples Used

2.2.1.1 Stochastic Gradient Descent

Actually, multiple ways can be used to compute the gradient of the objective function. Instead of updating the parameters using all the observations, the former can be updated using a single observation from the dataset at each iteration. Each sample observation is used in turn to evaluate the objective function and to update the parameters. Once all the observations have been used to update the parameters, we say that we have passed an **epoch**. The overall procedure in which a single observation (as opposed to the whole dataset) is used to update the parameters is called **Stochastic Gradient Descent** (SGD).

As opposed to Gradient Descent, Stochastic Gradient Descent do not require to train over the entire dataset, which may not be feasible depending on the size of the data at hand, or may be very slow. Imagine having a large dataset with a high number of features p and a large number of observations N. At each iteration, with Gradient Descent, we need to compute p first-order derivative for N observations. With Stochastic Gradient Descent, instead of computing the first-order derivative for all N observations, a single randomly drawn observation is used. The iteration is thus faster with Stochastic Gradient Descent. However, the update process becomes noisier and the algorithm converges at a lower rate. But the fact that the update process becomes noisier may not be a curse: it can allow us to avoid ending up in a local minimum.

As the update of the parameters is done for each observation, it is not possible to rely on vectorized or parallel implementation of this process.

To sum up, the algorithm works as follows:

Stochastic Gradient Descent Algorithm

1. Randomly pick starting values for the parameters
2. Select an observation
3. Compute the gradient of the objective function using the observation from step 2
4. Update the parameters
5. Repeat from step 2 until all the observations from the training sample have been used: this constitutes an epoch
6. Repeat the procedure from 2 to 5 to complete multiple epochs.

At iteration t, the parameters are updated using the ith observation:

$$\theta^{(t+1)} = \theta_i^{(t)} - \eta \cdot \nabla \mathcal{L}(\theta^{(t)}; X_i)$$

Let us apply this algorithm to estimate the parameters of a linear model. We can generate 1000 observations from the following process:

$$y_i = \beta_0 + \beta_1 x_{1,i} + \beta_2 x_{2,i} + \varepsilon_i, \quad i = 1, \dots, N$$

where x_1 and x_2 are randomly drawn from a $\mathcal{U}(0, 10)$ distribution and $\varepsilon \sim \mathcal{N}(0, 2)$.

Let us generate some data:

```
set.seed(123)
# Number of observations
n <- 1000
# x randomly drawn from a continuous uniform distribution with bounds [0,10]
x_1 <- runif(min = 0, max = 10, n = n)
x_2 <- runif(min = 0, max = 10, n = n)
# Error term from Normal distribution with zero mean and variance 4
error <- rnorm(n = n, mean = 0, sd = 2)
# Response variable
beta_0 <- 3
beta_1 <- -2
beta_2 <- .5
true_beta <- c(beta_0=beta_0, beta_1=beta_1, beta_2=beta_2)
y <- beta_0 + beta_1*x_1 + beta_2*x_2 + error
```

The objective function is the Mean Squared Error:

```
obj_function <- function(theta, y, X){
  y_pred <- X%*%theta
```

```
  mean((y - y_pred)^2)
}
```

We can construct the matrix of predictors as follows:

```
X <- cbind(rep(1, n), x_1, x_2)
colnames(X) <- c("Intercept", "x_1", "x_2")
head(X)
```

```
##      Intercept      x_1      x_2
## [1,]         1 2.875775 2.736227
## [2,]         1 7.883051 5.938669
## [3,]         1 4.089769 1.601848
## [4,]         1 8.830174 8.534302
## [5,]         1 9.404673 8.477392
## [6,]         1 0.455565 4.778868
```

We need some initial values for the vector of parameters:

```
beta <- c(1,1,1)
```

We can set the learning rate to 10^-2. We will only consider 10 epochs here,

```
learning_rate <- 10^-2
nb_epoch <- 20
```

To keep track of the process:

```
mse_values  <- NULL
```

We will compute the MSE after each epoch, on the whole dataset.

```
# pb <- txtProgressBar(min = 1, max=nb_epoch, style=3)
for(i_epoch in 1:nb_epoch){
  cat("\n----------\nEpoch: ", i_epoch, "\n")

  # Shuffle the order of observations
  index <- sample(1:n, size = n, replace=TRUE)
```

```
  for(i in 1:n){
    # The gradient is estimated using a single observation: the ith
    gradient <- grad(func = obj_function, x=beta,
                     y=y[index[i]], X = X[index[i],])
    # Updating the value
    beta <- beta - learning_rate * gradient
  }

  # Just for keeping track (not necessary to run the algorithm)
  # (Significantly slows down the algorithm)
  cost <- obj_function(beta, y, X)
  cat("MSE : ", cost, "\n")
  mse_values <- c(mse_values, cost)
  # End of keeping track


  # setTxtProgressBar(pb, i_epoch)
}
```

```
##
## ----------
## Epoch:  1
## MSE :  60.61677
##
## ----------
## Epoch:  2
## MSE :  14.27008
##
## ----------
## Epoch:  3
## MSE :  21.85282
##
## ----------
## Epoch:  4
## MSE :  4.317547
##
## ----------
## Epoch:  5
## MSE :  12.04292
##
## ----------
## Epoch:  6
## MSE :  7.704181
```

```
##
## ----------
## Epoch:  7
## MSE :  16.23794
##
## ----------
## Epoch:  8
## MSE :  6.016237
##
## ----------
## Epoch:  9
## MSE :  28.43145
##
## ----------
## Epoch:  10
## MSE :  10.1408
##
## ----------
## Epoch:  11
## MSE :  5.436552
##
## ----------
## Epoch:  12
## MSE :  8.876291
##
## ----------
## Epoch:  13
## MSE :  144.3951
##
## ----------
## Epoch:  14
## MSE :  7.522555
##
## ----------
## Epoch:  15
## MSE :  26.61209
##
## ----------
## Epoch:  16
## MSE :  5.29168
##
## ----------
## Epoch:  17
## MSE :  57.37272
##
## ----------
```

```
## Epoch:  18
## MSE :  4.390237
##
## ----------
## Epoch:  19
## MSE :  11.97248
##
## ----------
## Epoch:  20
## MSE :  5.225995
```

Here are the estimated parameters:

```
# True values:
true_beta
```

```
## beta_0 beta_1 beta_2
##    3.0   -2.0    0.5
```

```
# Estimates values:
(beta_sgd <- beta)
```

```
## [1]  2.6187446 -2.1165051  0.8039565
```

Looking at the MSE value at each epoch:

```
ggplot(data = tibble(epoch = 1:length(mse_values),
                     MSE = mse_values)) +
  geom_line(mapping = aes(x = epoch, y = MSE))
```

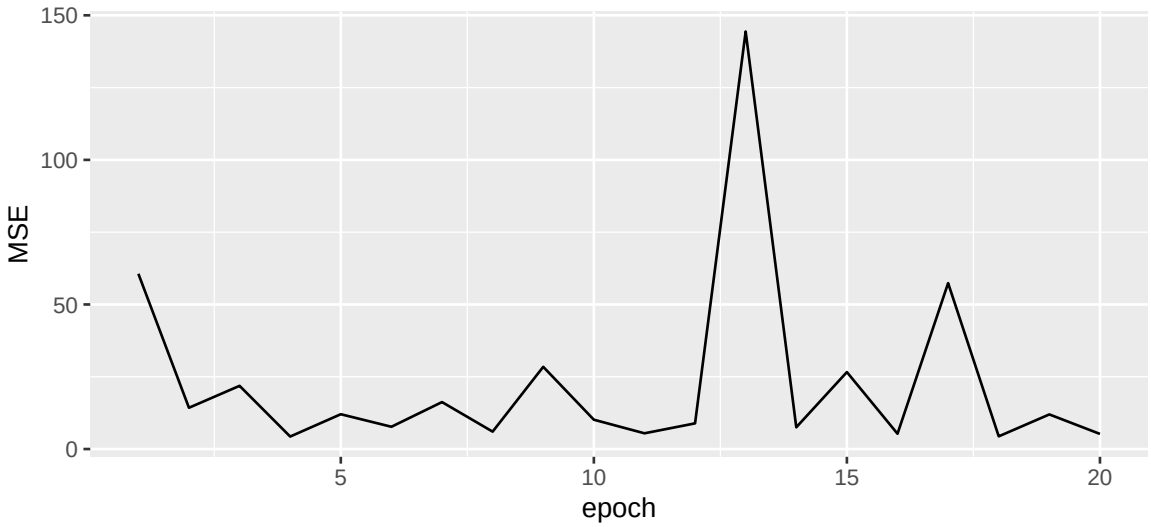

Figure 2.22: Singular Gradient Descent.

We can see that the MSE quickly falls but does not smoothly decreases with the epochs.

Let us create, for convenience, a simple function that performs the Stochastic Gradient Descent for a linear model:

```
#' Performs Stochastic Gradient Descent for a Linear Model.
#' @param par Initial values for the parameters.
#' @param fn A function to be minimized, with first argument the vector of
#'           parameters over which minimisation is to take place.
#'           It should return a scalar result.
#' @param y Target variable.
#' @param X Matrix of predictors.
#' @param learning_rate Learning rate.
#' @param nb_epoch Number of epochs.
#' @param silent If TRUE (default), progress information
#'               not printed in the console.
f_sgd <- function(par, fn, y, X, learning_rate=10^-2, nb_epoch=10, silent=TRUE){
  mse_values <- NULL
  for(i_epoch in 1:nb_epoch){
    if(!silent) cat("\n----------\nEpoch: ", i_epoch, "\n")
    n <- nrow(X)
    index <- sample(1:n, size = n, replace=TRUE)
```

```
    for(i in 1:n){
      gradient <- grad(func = fn, x=par, y=y[index[i]], X = X[index[i],])
      # Updating the value
      par <- par - learning_rate * gradient
    }

    # Just for keeping track (not necessary to run the algorithm)
    # Significantly slows down the algorith
    cost <- fn(par, y, X)
    if(!silent) cat("MSE : ", cost, "\n")
    mse_values <- c(mse_values, cost)
    # End of keeping track
  }
  structure(list(par = par, mse_values = mse_values,
                 nb_epoch = nb_epoch,
                 learning_rate = learning_rate))
}
```

It can then be applied as follows:

```
start_time_sgd <- Sys.time()
estim_sgd <- f_sgd(par = c(1,1,1), fn = obj_function, y = y, X = X,
                   silent=TRUE, nb_epoch = 20)
end_time_sgd <- Sys.time()
```

The time that elapsed to run through the 20 epochs:

```
# Time elapsed
end_time_sgd-start_time_sgd
```

```
## Time difference of 20.01073 secs
```

```
estim_sgd
```

```
## $par
## [1]  3.3837401 -1.4333884  0.5377486
##
## $mse_values
```

```
##  [1] 26.254430 20.623584  5.261876 14.536654  5.021757 36.114059
    15.058143
##  [8]  7.031129 19.488286  5.029290 21.690363 17.232837 14.928142
     9.019636
## [15] 17.868784  6.077928 16.709309 16.596630 11.279380 17.557011
##
## $nb_epoch
## [1] 20
##
## $learning_rate
## [1] 0.01
```

2.2.1.2 Batch Gradient Descent

Another variant of the Gradient Descent algorithm is the **Batch Gradient Descent**. Again, the gradient of the objective function is computed separately for each observation rather than on the whole dataset. In contrast with what is done with Stochastic Gradient Descent, the parameters are not updated after each observation. The average of the gradients computed for each observation is used to update the parameters only once at each epoch.

As there are fewer updates with this solution than with Stochastic Gradient Descent, the update process is less computationally expensive and less subject to noise. While a less noisy learning process can lead to more stable solutions, they also increase the risk of landing in a local minimum.

To sum up, the algorithm works as follows:

Batch Gradient Descent Algorithm

1. Randomly pick starting values for the parameters
2. For each observation, compute the gradient of the objective function
3. Compute the mean of the gradients computed in step 2
4. Update the parameters with the mean gradient from step 3
5. Repeat from step 2 a given number of times.

At iteration t, the parameters are updated as follows:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \frac{1}{n} \sum_{i=1}^{n} \nabla \mathcal{L}(\theta^{(t)}; X_i),$$

where n is the size of the training sample.

Let us use the same data as earlier, and use this time the Batch Gradient algorithm instead of the Stochastic Gradient algorithm.

We need starting values for the parameters:

```
beta <- c(1,1,1)
```

The learning rate and the number of epochs are the same:

```
learning_rate <- 10^-2
nb_epoch <- 20
```

And let us keep track of the MSE through the epochs:

```
mse_values  <- NULL
```

```
# pb <- txtProgressBar(min = 1, max=nb_epoch, style=3)
for(i_epoch in 1:nb_epoch){
  cat("\n----------\nEpoch: ", i_epoch, "\n")
  # For each observation, we need to compute the gradient
  gradients <- rep(0, ncol(X))
  for(i in 1:n){
    gradient_current <- grad(func = obj_function, x=beta, y=y[i], X = X[i,])
    gradients <- gradients+gradient_current
  }
  # Then we divide by the number of observations to get the average
  avg_gradients <- gradients/n

  # Updating the value
  beta <- beta - learning_rate * avg_gradients

  # Just for keeping track (not necessary to run the algorithm)
  # Significantly slows down the algorith
  cost <- obj_function(beta, y, X)
  cat("MSE : ", cost, "\n")
  mse_values <- c(mse_values, cost)
  # End of keeping track


  # setTxtProgressBar(pb, i_epoch)
}
```

```
##
## ----------
```

```
## Epoch:  1
## MSE :  30.67738
##
## ----------
## Epoch:  2
## MSE :  17.55995
##
## ----------
## Epoch:  3
## MSE :  13.37346
##
## ----------
## Epoch:  4
## MSE :  10.64122
##
## ----------
## Epoch:  5
## MSE :  8.778509
##
## ----------
## Epoch:  6
## MSE :  7.506044
##
## ----------
## Epoch:  7
## MSE :  6.636364
##
## ----------
## Epoch:  8
## MSE :  6.041603
##
## ----------
## Epoch:  9
## MSE :  5.634493
##
## ----------
## Epoch:  10
## MSE :  5.355469
##
## ----------
## Epoch:  11
## MSE :  5.163874
##
## ----------
## Epoch:  12
## MSE :  5.03196
```

```
##
## ----------
## Epoch:  13
## MSE :  4.940786
##
## ----------
## Epoch:  14
## MSE :  4.877425
##
## ----------
## Epoch:  15
## MSE :  4.833054
##
## ----------
## Epoch:  16
## MSE :  4.80165
##
## ----------
## Epoch:  17
## MSE :  4.779102
##
## ----------
## Epoch:  18
## MSE :  4.762605
##
## ----------
## Epoch:  19
## MSE :  4.750244
##
## ----------
## Epoch:  20
## MSE :  4.740712
```

The estimated values:

```
# True values:
true_beta
```

```
## beta_0 beta_1 beta_2
##    3.0   -2.0    0.5
```

```r
# Estimated values:
(beta_batch <- beta)
```

```
## [1]  0.8526963 -1.7735736  0.6632453
```

These codes can be wrapped up in a simple function:

```r
#' Performs Batch Gradient Descent for a Linear Model
#' @param par Initial values for the parameters.
#' @param fn A function to be minimized, with first argument the vector of
#'           parameters over which minimisation is to take place.
#'           It should return a scalar result.
#' @param y Target variable.
#' @param X Matrix of predictors.
#' @param learning_rate Learning rate.
#' @param nb_epoch Number of epochs.
#' @param silent If TRUE (default), progress information
#'               not printed in the console.
batch_gd <- function(par, fn, y, X, learning_rate=10^-2,
                     nb_epoch=10, silent=TRUE){
  mse_values  <- NULL
  n <- nrow(X)
  for(i_epoch in 1:nb_epoch){
    if(!silent) cat("\n----------\nEpoch: ", i_epoch, "\n----------")
    # For each observation in the batch, we need to compute the gradient
    gradients <- rep(0, ncol(X))
    for(i in 1:n){
      gradient_current <- grad(func = fn, x=par, y=y[i], X = X[i,])
      gradients <- gradients+gradient_current
    }
    # Then we divide by the number of observations to get the average
    avg_gradients <- gradients/n

    # Updating the value
    par <- par - learning_rate * avg_gradients

    # Just for keeping track (not necessary to run the algorithm)
    # Significantly slows down the algorithm
```

```
    cost <- fn(par, y, X)
    if(!silent) cat("MSE : ", cost, "\n")
    mse_values <- c(mse_values, cost)
    # End of keeping track
  }
  structure(list(par = par, mse_values = mse_values,
                 nb_epoch = nb_epoch,
                 learning_rate = learning_rate))
}
```

This function can be used as follows:

```
# To keep time
start_time_batch <- Sys.time()
estim_batch <-
  batch_gd(par = c(1,1,1), fn = obj_function, y = y, X = X, silent=TRUE,
           nb_epoch = 20)
end_time_batch <- Sys.time()
```

The time that has elapsed to run through the 20 epochs:

```
end_time_batch-start_time_batch
```

```
## Time difference of 20.70484 secs
```

We notice that this is not very different from the time used to estimate the parameters on 20 epochs with the Stochastic Gradient Descent algorithm: the number of predictors is very small in this example.

Looking at the MSE:

```
tibble(MSE = estim_batch$mse_values, epoch = 1:estim_batch$nb_epoch) %>%
  ggplot(data = ., mapping = aes(x=epoch, y = MSE)) +
  geom_line() +
  labs(x = "Epoch", y = "MSE")
```

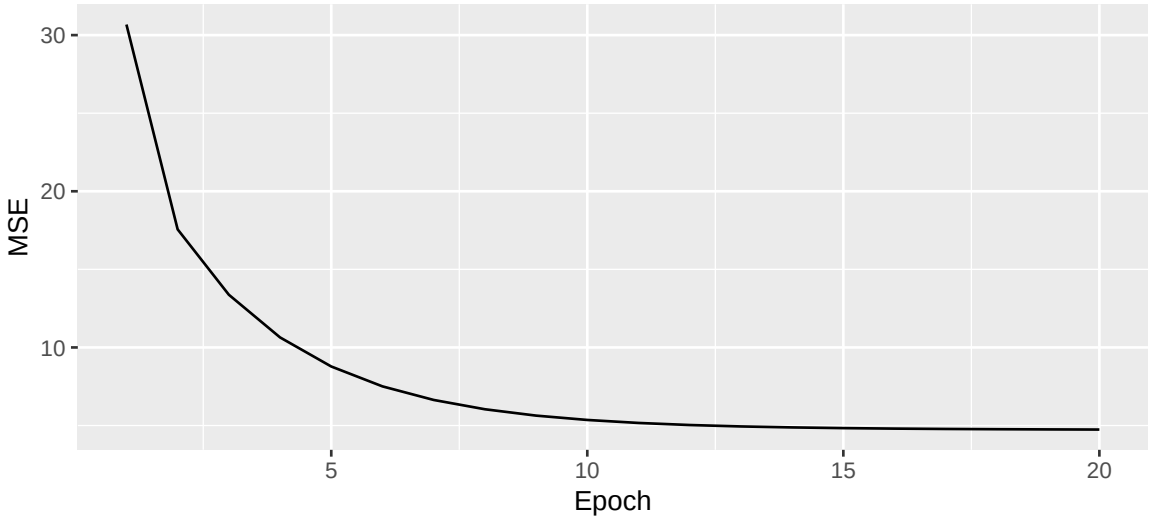

Figure 2.23: Batch Gradient Descent.

We can see that the decrease in the objective function is smoother with Batch Gradient Descend.

2.2.1.3 Mini-Batch Gradient Descent

Even if vectorised implementation can be used to fasten computing with Batch Gradient Descent (which is not possible with Stochastic Gradient Descent), the whole dataset is usually required to be loaded in memory, which can be time consuming.

Another approach, called **Mini-Batch Gradient Descent**, combines the idea of both Stochastic Gradient Descent and Batch Gradient Descent. In its first step, it consists in creating a batch of observations of smaller size than the entire dataset, called a mini-batch (usually, 64, 128, or 256 observations are used to create a mini batch). Then, the gradient of the objective function is calculated for each observation in the mini batch. The gradients are then averaged and used to update the parameters. A new iteration can then begin with a new mini-batch.

For a given mini-batch, the computations can be vectorised and does not require to have the entire dataset loaded in memory. However, the size of the mini-batches need to be decided on prior the algorithm is launched.

To sum up, the algorithm works as follows:

Mini-Batch Descent Algorithm

1. Randomly pick n observations from the training sample
2. For each observation, compute the gradient of the objective function
3. Compute the mean of the gradients computed in step 2
4. Update the parameters with the mean gradient from step 3
5. Repeat from step 1 a given number of times.

At iteration t, the parameters are updated as follows:

$$\theta^{(t+1)} = \theta_i^{(t)} - \eta \cdot \frac{1}{n} \sum_{i=1}^{n} \nabla \mathcal{L}(\theta^{(t)}; X_i),$$

where n is the size of the mini-batch.

Again, let us implement this algorithm with the linear model from earlier.

We need starting values for the parameters:

```
beta <- c(1,1,1)
```

Let us use the same learning rate as that was used with the Stochastic Gradient Descent algorithm and the same number of epochs.

```
learning_rate <- 10^-2
nb_epoch <- 20
```

We can select a number of observations per batch:

```
batch_size <- 250
```

Again, let us keep track of the MSE values after each epoch:

```
mse_values  <- NULL
```

Then we can use the following loop:

```
# pb <- txtProgressBar(min = 1, max=nb_epoch, style=3)
for(i_epoch in 1:nb_epoch){
  # cat("\n----------\nEpoch: ", i_epoch, "\n")
  # Randomly draw a batch
  index <- sample(1:n, size = batch_size, replace=TRUE)
```

```
  # For each observation in the batch, we need to compute the gradient
  gradients_batch <- rep(0, ncol(X))
  for(i in 1:batch_size){
    gradient_current <-
      grad(func = obj_function, x=beta, y=y[index[i]], X = X[index[i],])
    gradients_batch <- gradients_batch+gradient_current
  }
  # Then we divide by the number of observations to get the average
  avg_gradients_batch <- gradients_batch/batch_size

  # Updating the value
  beta <- beta - learning_rate * avg_gradients_batch

  # Just for keeping track (not necessary to run the algorithm)
  # Significantly slows down the algorithm
  cost <- obj_function(beta, y, X)
  mse_values <- c(mse_values, cost)
  # End of keeping track

  # setTxtProgressBar(pb, i_epoch)
}
```

The estimated values:

```
# True values:
true_beta
```

```
## beta_0 beta_1 beta_2
##    3.0   -2.0    0.5
```

```
# Estimated values
(beta_batch <- beta)
```

```
## [1]  0.8524989 -1.7747998  0.6444584
```

As for the Batch Gradient Descent, let us wrap these codes in a function:

```r
#' Performs Batch Gradient Descent for a Linear Model
#' @param par Initial values for the parameters.
#' @param fn A function to be minimized, with first argument the vector
#'           of parameters over which minimisation is to take place.
#'           It should return a scalar result.
#' @param y Target variable.
#' @param X Matrix of predictors.
#' @param learning_rate Learning rate.
#' @param nb_epoch Number of epochs.
#' @param batch_size Batch size.
#' @param silent If TRUE (default), progress information
#'               not printed in the console.
mini_batch_gd <- function(par, fn, y, X, learning_rate=10^-2, nb_epoch=10,
                          batch_size = 128, silent=TRUE){
  mse_values  <- NULL
  n <- nrow(X)
  for(i_epoch in 1:nb_epoch){
    if(!silent) cat("\n----------\nEpoch: ", i_epoch, "\n----------")
    # Randomly draw a batch
    index <- sample(1:n, size = batch_size, replace=TRUE)
    # For each observation in the batch, we need to compute the gradient
    gradients_batch <- rep(0, ncol(X))
    for(i in 1:batch_size){
      gradient_current <-
        grad(func = fn, x=par, y=y[index[i]], X = X[index[i],])
      gradients_batch <- gradients_batch+gradient_current
    }
    # Then we divide by the number of observations to get the average
    avg_gradients_batch <- gradients_batch/batch_size

    # Updating the value
    par <- par - learning_rate * avg_gradients_batch

    # Just for keeping track (not necessary to run the algorithm)
    # Significantly slows down the algorithm
    cost <- fn(par, y, X)
    if(!silent) cat("MSE : ", cost, "\n")
    mse_values <- c(mse_values, cost)
    # End of keeping track
```

```
  }
  structure(list(par = par, mse_values = mse_values,
                 nb_epoch = nb_epoch,
                 learning_rate = learning_rate,
                 batch_size = batch_size))
}
```

Let us run the Mini-Batch Gradient Descent algorithm multiple times, varying the number of observations in the mini-batches at each time:

```
# With 32 obs per mini-batch
start_time_mini_batch_32 <- Sys.time()
mini_batch_32 <- mini_batch_gd(par = c(1,1,1), fn = obj_function,
                               y = y, X = X,
                               silent=TRUE, nb_epoch = 20, batch_size = 32)
end_time_mini_batch_32 <- Sys.time()



# With 64 obs per mini-batch
start_time_mini_batch_64 <- Sys.time()
mini_batch_64 <- mini_batch_gd(par = c(1,1,1), fn = obj_function,
                               y = y, X = X, silent=TRUE,
                               nb_epoch = 20, batch_size = 64)
end_time_mini_batch_64 <- Sys.time()


# With 128 obs per mini-batch
start_time_mini_batch_128 <- Sys.time()
mini_batch_128 <- mini_batch_gd(par = c(1,1,1), fn = obj_function,
                                y = y, X = X, silent=TRUE,
                                nb_epoch = 20, batch_size = 128)
end_time_mini_batch_128 <- Sys.time()


# With 256 obs per mini-batch
start_time_mini_batch_256 <- Sys.time()
mini_batch_256 <- mini_batch_gd(par = c(1,1,1), fn = obj_function,
                                y = y, X = X, silent=TRUE,
```

```
                                  nb_epoch = 20, batch_size = 256)
end_time_mini_batch_256 <- Sys.time()
```

The estimated parameters:

```
# True values:
beta
```

```
## [1]  0.8524989 -1.7747998  0.6444584
```

```
# Estimated values:
mini_batch_32$par
```

```
## [1]  0.8851996 -1.7382412  0.7427221
```

```
mini_batch_64$par
```

```
## [1]  0.8647598 -1.7800043  0.6838670
```

```
mini_batch_128$par
```

```
## [1]  0.8689784 -1.7547041  0.6840733
```

```
mini_batch_256$par
```

```
## [1]  0.8557173 -1.7529742  0.6988836
```

Let us look at the time used to estimate the parameters in each situation. The greater the number of observations, the greater the time taken by the algorithm.

```
end_time_mini_batch_32-start_time_mini_batch_32
```

```
## Time difference of 0.8221748 secs
```

```
end_time_mini_batch_64-start_time_mini_batch_64
```

```
## Time difference of 1.153019 secs
```

```
end_time_mini_batch_128-start_time_mini_batch_128
```

```
## Time difference of 2.64806 secs
```

```
end_time_mini_batch_256-start_time_mini_batch_256
```

```
## Time difference of 6.198186 secs
```

> *Note*: if we pick a mini-batch size of 1, the Mini-Batch Gradient Descent algorithm is the same as the Batch Gradient Descent algorithm.

And let us have a look at the MSE along the epochs:

```
df_plot <-
  map_df(list(mini_batch_32, mini_batch_64, mini_batch_128, mini_batch_256),
         ~tibble(MSE = .$mse_values,
                 batch_size = .$batch_size,
                 epoch = 1:(.$nb_epoch)))

ggplot(data = df_plot, mapping = aes(x=epoch, y = MSE)) +
  geom_line(aes(colour = as.factor(batch_size))) +
  labs(x = "Epoch", y = "MSE") +
  scale_colour_discrete("Mini-Batch Size")
```

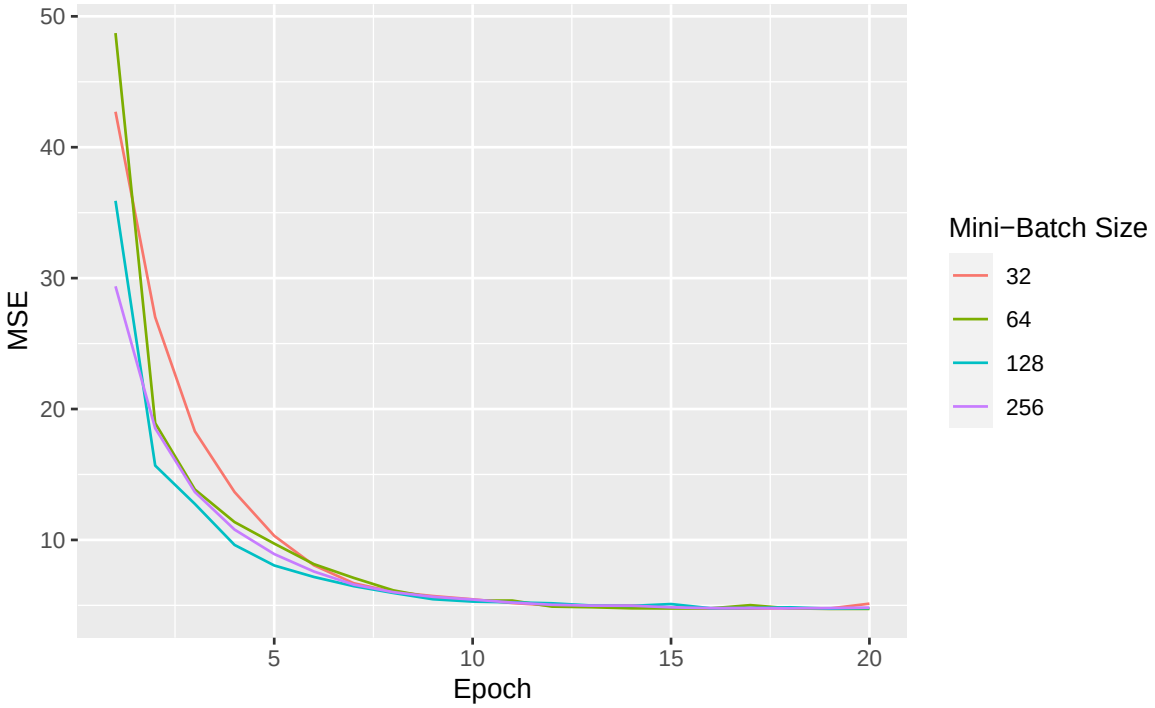

Figure 2.24: Mini-Batch Gradient Descent.

We note that the update process is smoother, less noisier as long as we increase the batch size.

To finish this section, let us have a look at the MSE over the epochs for the different algorithms we used, on the same graph:

```
df_plot <-
  tibble(MSE = estim_sgd$mse_values, epoch = 1:estim_sgd$nb_epoch,
         type = "Stochastic Gradient Descent") %>%
  bind_rows(
    tibble(MSE = estim_batch$mse_values, epoch = 1:estim_batch$nb_epoch,
           type = "Batch Gradient Descent")
  ) %>%
  bind_rows(
    map_df(list(mini_batch_32, mini_batch_64,
                mini_batch_128, mini_batch_256),
           ~tibble(MSE = .$mse_values,
                   epoch = 1:(.$nb_epoch),
                   type = .$batch_size,)) %>%
```

```
      mutate(type = str_c("Mini-Batch Gradient Descent (size=", type, ")"))
  )

ggplot(data = df_plot, mapping = aes(x=epoch, y = MSE)) +
  geom_line(mapping = aes(colour = type)) +
  labs(x = "Epoch", y = "MSE") +
  scale_colour_manual(values = c(
    "Stochastic Gradient Descent" = "#DDCC77",
    "Batch Gradient Descent" = "#117733",
    "Mini-Batch Gradient Descent (size=32)" = "#44AA99",
    "Mini-Batch Gradient Descent (size=64)" = "#88CCEE",
    "Mini-Batch Gradient Descent (size=128)" = "#DC267F",
    "Mini-Batch Gradient Descent (size=256)" = "#882255"
  )) +
  theme(legend.position = "bottom")
```

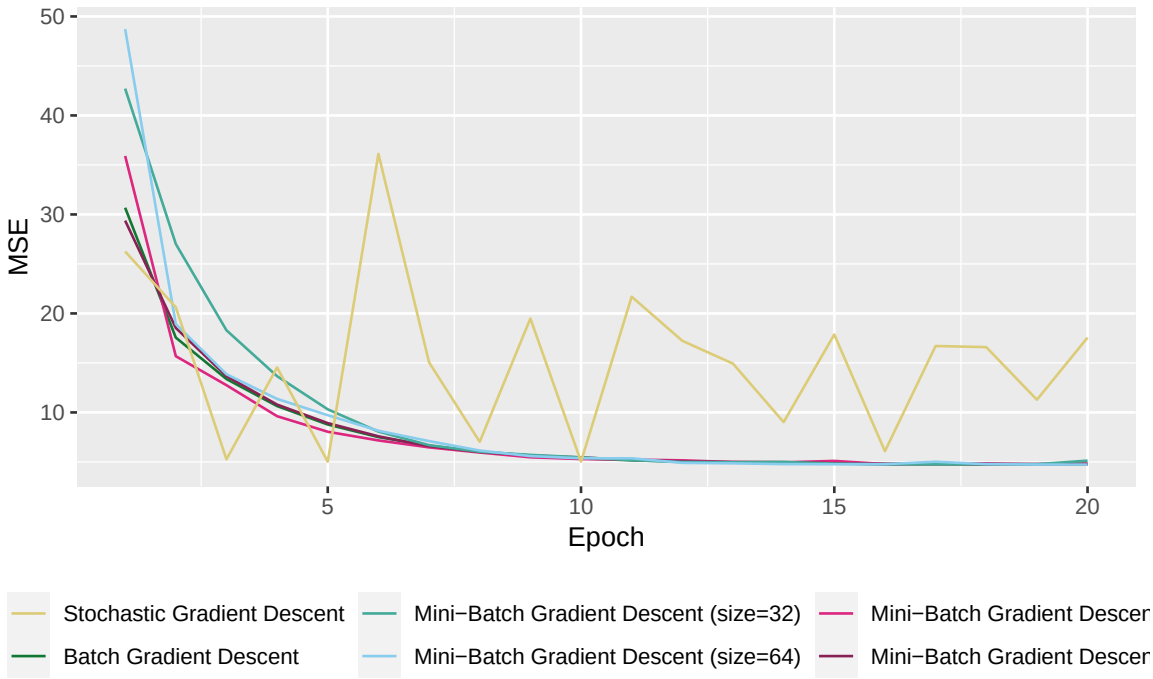

Figure 2.25: Optimisation with different algorithms.

2.2.2 Varying the Learning Rate

So far, we have considered a fixed learning rate η. The update rule for the p parameters of the objective function we used was the following:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \nabla \mathcal{L}\left(\theta^{(t)}\right).$$

The learning rate may change over the iteration process so that the update rule becomes:

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \cdot \nabla \mathcal{L}\left(\theta^{(t)}\right),$$

where η_t can be set in various ways.

2.2.2.1 Linear Decaying Rate

The learning rate can be set so that it decreases linearly with the number of iterations. In such a case, it is defined as follows:

$$\eta_t = \frac{\eta_t}{t+1}$$

2.2.2.2 Quadratic Decaying Rate

For a quadratically decaying learning rate:

$$\eta_t = \frac{\eta_t}{(t+1)^2}$$

2.2.2.3 Exponential Decaying Rate

For an exponential decay:

$$\eta_t = \eta_t \exp(-\beta t),$$

where $\beta > 0$.

2.3 Other Algorithms

There are many other algorithms and variants. I would like to sketch two other algorithms in this notebook: Newton's algorithm and the Coordinate Descent algorithm.

2.3.1 Newton's Method

When the function to be optimised is convex, doubly differentiable and takes its values in $\mathbb{R}^n$, it is possible to use the second-order derivative to redefine the learning rate.

Taylor's theorem states that if $\mathcal{L} : \mathbb{R}^p \to \mathbb{R}$ is twice-differentiable at point θ, for any small change $\delta\theta$, the best quadratic approximation to $\mathcal{L}$ is given by the second-order Taylor series:

$$\mathcal{L}(\theta + \delta\theta) = \mathcal{L}(\theta) + \nabla\mathcal{L}(\theta)^\top \delta\theta + \frac{1}{2}\delta\theta^\top \mathbf{H}\delta\theta + \mathcal{O}(\|\delta^3\theta\|),$$

with $\mathbf{H} = \nabla^2\mathcal{L}(\theta)$ the Hessian matrix
In a similar way as in the case of the best linear approximation, we need to take a step $\delta\theta$ such that :

$$\mathcal{L}(\theta + \delta\theta) < \mathcal{L}(\theta),$$

i.e., for which:

$$\delta\theta^\top \mathbf{H}\delta\theta < 0$$

With Newton's method, we will thus take a step along the gradient, and we will use the Hessian matrix to decide the step to take: by doing to, the rate at which we will go down the gradient will account for the convexity of the function.

Newton's Method

1. Randomly pick starting values for the parameters
2. Compute both the gradient and the Hessian of the objective function at the current value of the parameters using all the observations from the training sample
3. Update the parameters
4. Repeat from step 2 until a fixed number of iteration or until convergence.

At iteration t, the parameters are updated as follows:

$$\mathbf{H}^{(t)} = \nabla^2\mathcal{L}(\theta^{(t)})$$

$$\theta^{(t+1)} = \theta^{(t)} - (\mathbf{H}^{(t)})^{-1} \cdot \nabla\mathcal{L}(\theta^{(t)}),$$

While computing the second-order derivative can be fast if the expression of this function is simple, it can become computationally very expensive otherwise. The computation of the Hessian can also be very challenging when facing a large number of observations (n^2 computations are required for the second-order derivative).

Computing the inverse of the Hessian matrix is computationally expensive. The BFGS (Broyden Fletcher Goldfard Shanno) method avoids computing $\mathbf{H}^{-1}$ and instead estimates an approximation of the Hessian matrix.

Let us illustrate the method. Consider the following function:

$$f(x_1, x_2) = (x_1 - x_2)^4 + 2x_1^2 + x_2^2 - x_1 + 2x_2$$

```
x_1 <- seq(-10, 10, by = 0.3)
x_2 <- seq(-10, 10, by = 0.3)
z_f <- function(x_1,x_2) (x_1-x_2)^4 + 2*x_1^2 + x_2^2 - x_1 + 2*x_2
z_f_to_optim <- function(theta){
  x_1 <- theta[1]
  x_2 <- theta[2]
  (x_1-x_2)^4 + 2*x_1^2 + x_2^2 - x_1 + 2*x_2
}
z <- outer(x_1, x_2, z_f)
```

A graphical representation of this function can be obtained as follows:

```
par(mar = c(1, 1, 1, 1))
th = 150
pmat <-
  persp3D(x = x_1, y = x_2, z = z, colkey=F, contour=T,
          ticktype = "detailed", asp = 1, phi = 40, theta = th,
          border = "grey10", alpha=.4, d = .8,r = 2.8,
          expand = .6, shade = .2,axes = T,box = T,cex = .1)
```

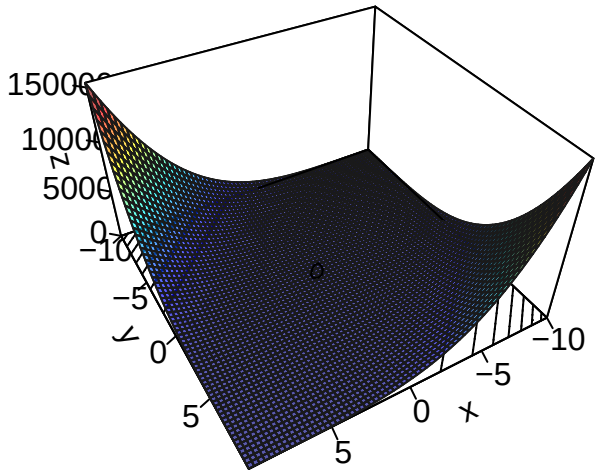

Figure 2.26: Surface of the illustrative function.

Let us pick some starting values:

```
theta <- c(-9, 9)
```

The Newton's Method quickly converges, let us pick a small maximum number of iteration.

```
nb_max_iter <- 20
```

Let us set a value for the absolute tolerance:

```
abstol <- 10^-5
```

At our starting point, the value of the function is:

```
(current_obj <- z_f_to_optim(theta))
```

```
## [1] 105246
```

Let us keep a track on our updated values for the vector of parameters:

```
theta_values <- NULL
```

Then we can make a loop to update iteratively the vector of parameters.

```
for(i in 1:nb_max_iter){
  gradient <- grad(func = z_f_to_optim, x = theta)
  H <- hessian(func = z_f_to_optim, x = theta)
  # Updating the parameters
  theta <- theta - t(solve(H) %*% gradient)
  new_obj <- z_f_to_optim(theta)
  # Keeping track
  theta_values <- rbind(theta_values, theta)

  if(abs(current_obj - new_obj) < abstol){
    break
  }else{
    current_obj <- new_obj
  }
}
```

The algorithm stopped after the following number of iterations:

```
i
```

```
## [1] 13
```

The algorithm tells us that the minimum is reached at the following point:

```
theta
```

```
##              [,1]       [,2]
## [1,] 0.03349047 -0.5669809
```

Let us have a look at the updates on a first graph:

```
par(mar = c(1, 1, 1, 1))
pmat <-
  persp3D(x = x_1, y = x_2, z = z, colkey=F, contour=T,
          ticktype = "detailed", asp = 1, phi = 40, theta = th,
          border = "grey10", alpha=.4, d = .8,r = 2.8,
          expand = .6, shade = .2,axes = T,box = T,cex = .1)
```

```
xx <- theta_values[,1]
yy <- theta_values[,2]
zz <- z_f(xx,yy)
new_point <- trans3d(xx,yy,zz,pmat = pmat)
lines(new_point,pch = 20,col = "red", cex=2)
points(new_point,pch = 20,col = "red", cex=2)
points(map(new_point, last),pch = 20,col = "green", cex=1.5)
```

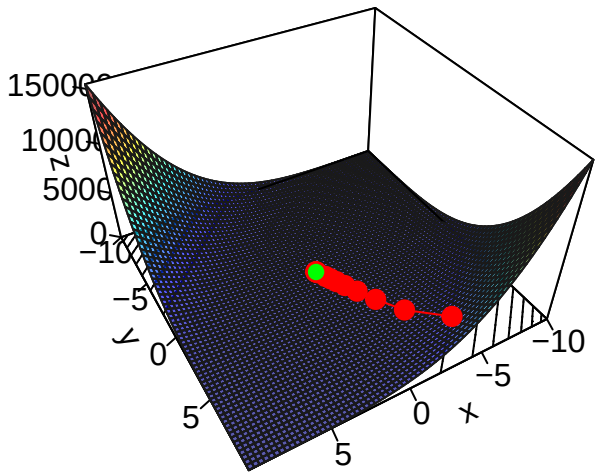

Figure 2.27: Newton's algorithm: steps of the iterative process.

And on a contour plot:

```
contour2D(x=x_1, y=x_2, z=z, colkey=F,
          main="Contour plot", xlab="x_1", ylab="x_2")
for(i in 1:(nrow(theta_values)-1)){
  segments(x0 = theta_values[i, 1], x1 = theta_values[i+1, 1],
           y0 = theta_values[i, 2], y1 = theta_values[i+1, 2],
           col = "red", lwd=2)
}
points(x=theta_values[,1], y=theta_values[,2], t="p",
       pch=19, col = "red")
points(x=theta_values[nrow(theta_values),1],
       y=theta_values[nrow(theta_values),2], t="p",
       pch=19, col = "green")
```

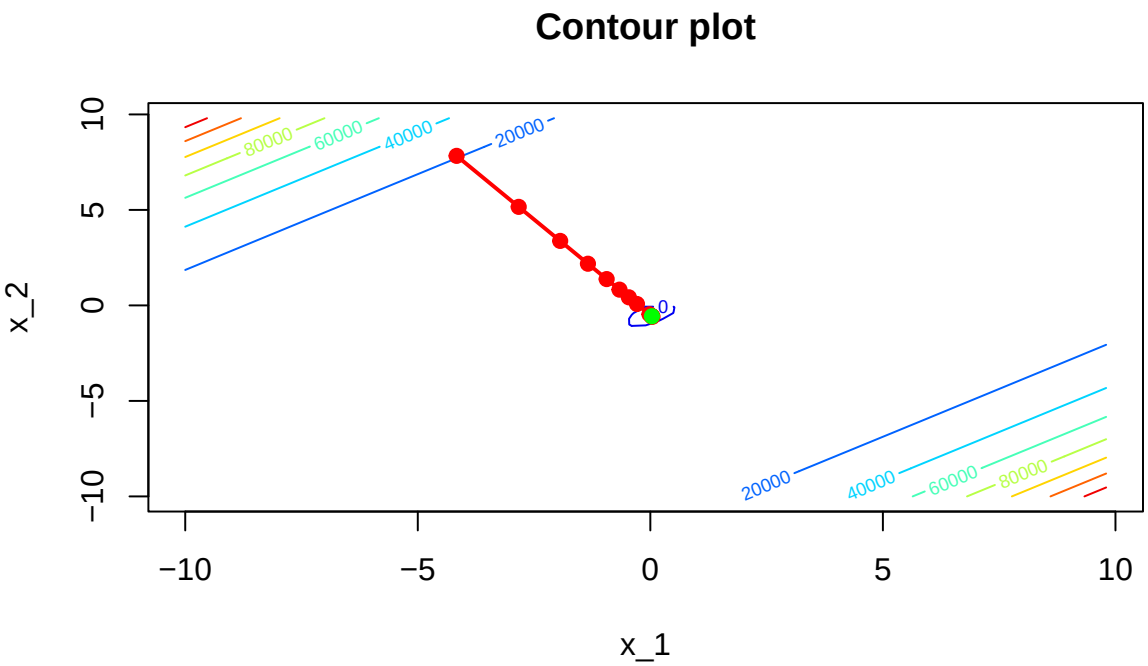

Figure 2.28: Newton's algorithm: contour plot of the iterative process.

Here, we converged quickly to the minimum, and the computation was really fast. When applying this algorithm to minimise the objective function of a supervised learning task using large datasets, computing the Hessian become way more costly.

To get more details on Newton's method, see Tibshirani (2019).

2.3.2 Coordinate Descent Algorithm

When trying to optimise a high-dimensional multivariate function, the calculation of each first derivative can quickly become very time consuming.

Intuitively, the n-dimensional optimisation problem can be seen as several small 1-dimensional optimisation problems. The basic idea is to try to minimise over a single dimension at each iteration, keeping all the values of the parameters constant.

More (technical/mathematical) details can be found in the slides titled "Coordinate Descent and Ascent Methods" from Nutini (2015) and in the slides "Optimisation et convexité 1, 2 and 3" from Charpentier (2020) (although the title is in French, the slides are in English, only the videos are in French).

If the function f is convex and differentiable, we can rely on the following theorem to find the minimum:

If $f : \mathbb{R}^n \to \mathbb{R}$ is convex, differentiable, then :

$$f(\mathbf{x}) \leq f\left(\mathbf{x} + \delta \rightarrow_i\right), \forall i \Rightarrow f(\mathbf{x}) = \min\{f\},$$

where $\rightarrow_i = (0, \ldots, 0, 1, 0, \ldots, 0) \in \mathbb{R}^n$.

In other words, if we find a point $\mathbf{x}$ such that $f(\mathbf{x})$ is minimised along each of the n coordinate axis, this point is a global minimiser.

We can thus try to find the minimum in each direction instead of looking directly at the problem in n dimensions. We will end up in the global minimum.

The algorithm works the following way:

Coordinate Descent Algorithm

1. Randomly pick starting values for the parameters
2. Select a dimension among the p (cyclic sampling, uniform sampling, ...)
3. Compute the first-order derivative of the objective function with respect to the ith parameter
4. Update the ith parameter
5. Repeat from step 2 until a fixed number of iteration or until convergence.

Let us first consider a smooth function to illustrate the method:

$$f(x_1, x_2) = x_1^2 + x_2^2 + x_1 x_2$$

Let us now generate some observations from that function:

```
library(plot3D)
library(numDeriv)
n <- 40
x_1 <- x_2 <- seq(-3, 3, length.out=n)
z_f <- function(x_1, x_2) x_1^2 + x_2^2 + x_1*x_2
z_f_to_optim <- function(theta)
  theta[1]^2 + theta[2]^2 + theta[1]*theta[2]
z <- outer(x_1, x_2, z_f)
```

We can visualise these observations on a 3D graph:

```
op <- par()
par(mar = c(1, 1, 1, 1))
flip <- 1
th <- 200
pmat <-
  persp3D(x = x_1, y = x_2, z = z, colkey=F, contour=T,
          ticktype = "detailed", xlab = "x_1", ylab = "x_2",
          zlab = "f(x_1, x_2)", asp = 1, phi = 30, theta = th,
          border = "grey10", alpha=.4, d = .8,r = 2.8,
          expand = .6, shade = .2, axes = T, box = T, cex = .1)
```

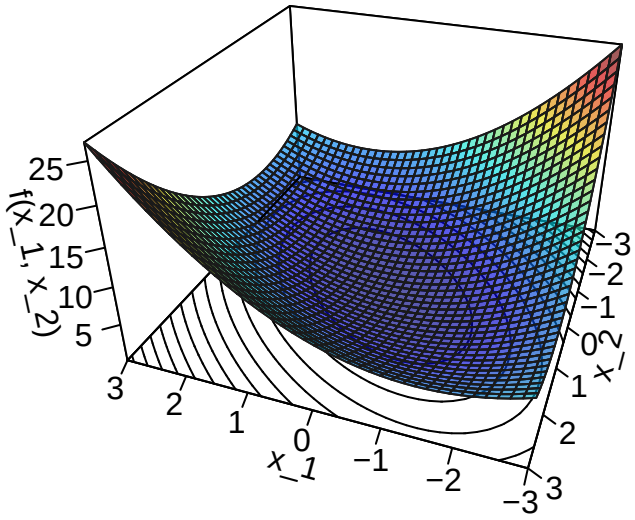

Figure 2.29: Surface of the illustrative spherical function.

Now, let us consider a starting point: $\theta = (2, 2.2)$

```
theta <- c(2, 2.2)
```

A contour plot can also be used to visualise the process:

```
contour(x_1, x_1, z, nlevels = 20, xlab = "x_1", ylab = "x_2")
points(theta[1], theta[2], pch = 19, cex = 2, col = "red")
```

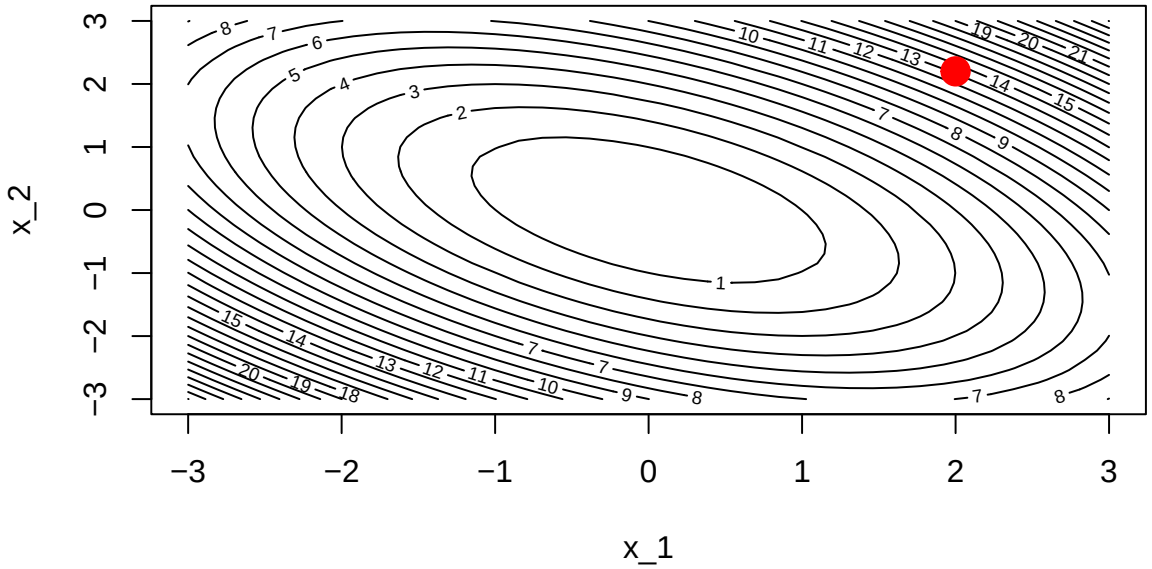

Figure 2.30: Starting point.

We need to minimise over a single dimension. For example, let us begin with the first dimension.

```
dim_i <- 1
```

The value of the parameter of the other dimensions (here only the second dimension) will be held fixed. We will only update the first dimension of the parameter.

```
contour(x_1, x_1, z, nlevels = 20, xlab = "x_1", ylab = "x_2")
abline(h = theta[2], lty=2, col = "blue", lwd=2)
points(theta[1], theta[2], pch = 19, cex = 2, col = "red")
```

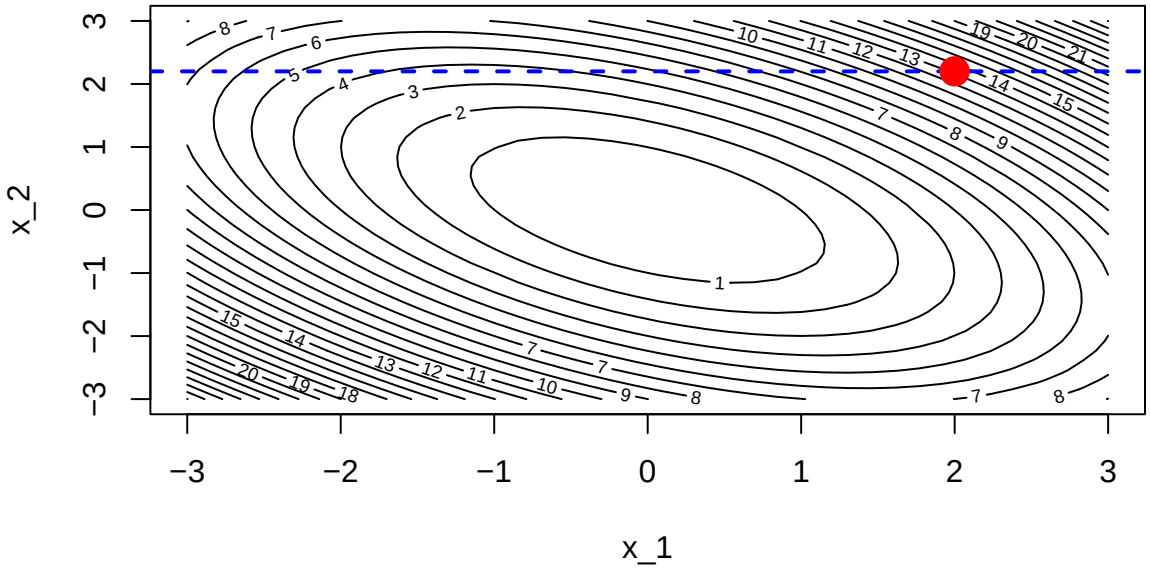

Figure 2.31: Optimisation in a single dimension (dashed blue line).

The first derivative of our function f with respect to x_1 writes:

$$\frac{\partial f}{\partial x_1}(x_1, x_2) = 2x_1 + x_2.$$

```
derivative_wrt_x1 <- function(theta){
  2*theta[1] + theta[2]
}
```

Evaluated at θ:

```
(grad_i <- derivative_wrt_x1(theta))
```

```
## [1] 6.2
```

Let us select a learning rate:

```
learning_rate <- 10^-1
```

The vector of parameters can then be updated:

```
theta_update <- theta
theta_update[dim_i] <- theta_update[dim_i] - learning_rate * grad_i
theta_update
```

```
## [1] 1.38 2.20
```

Let us keep track of the evolution of the values of θ.

```
theta_values <- rbind(theta, theta_update)
theta_values
```

```
##                [,1] [,2]
## theta          2.00  2.2
## theta_update 1.38  2.2
```

```
contour(x_1, x_1, z, nlevels = 20, xlab = "x_1", ylab = "x_2")
abline(h = theta[2], lty=2, col = "blue", lwd = 2)
points(theta[1], theta[2], pch = 19, cex = 2, col = "red")
segments(x0 = theta[1], x1 = theta_update[1],
         y0 = theta[2], y1 = theta_update[2], col = "red", lwd=2)
points(theta_update[1], theta_update[2], pch = 19, cex = 2, col = "red")
```

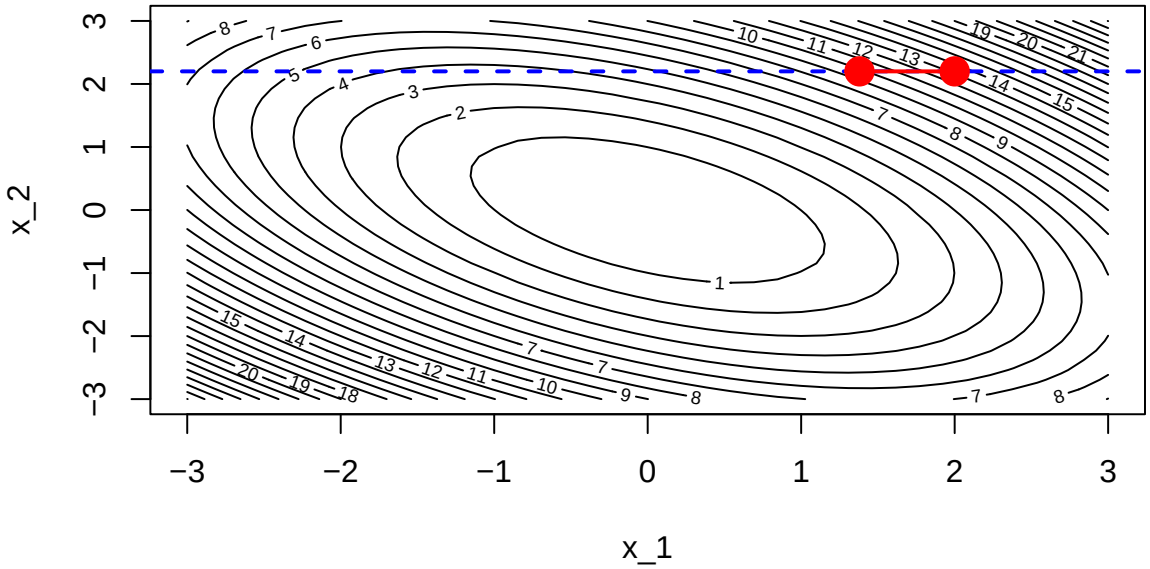

Figure 2.32: Updated value after the first step.

A new iteration can then begin. Let us now consider another dimension:

```
dim_i <- 2
```

This time, we will try to optimise on this second dimension only, keeping the values of the other dimension constant.

```
contour(x_1, x_1, z, nlevels = 20, xlab = "x_1", ylab = "x_2")
abline(v = theta_update[1], lty=2, col = "blue", lwd = 2)
points(theta[1], theta[2], pch = 19, cex = 2, col = "red")
segments(x0 = theta[1], x1 = theta_update[1],
         y0 = theta[2], y1 = theta_update[2], col = "red", lwd=2)
points(theta_update[1], theta_update[2], pch = 19,
       cex = 2, col = "red")
```

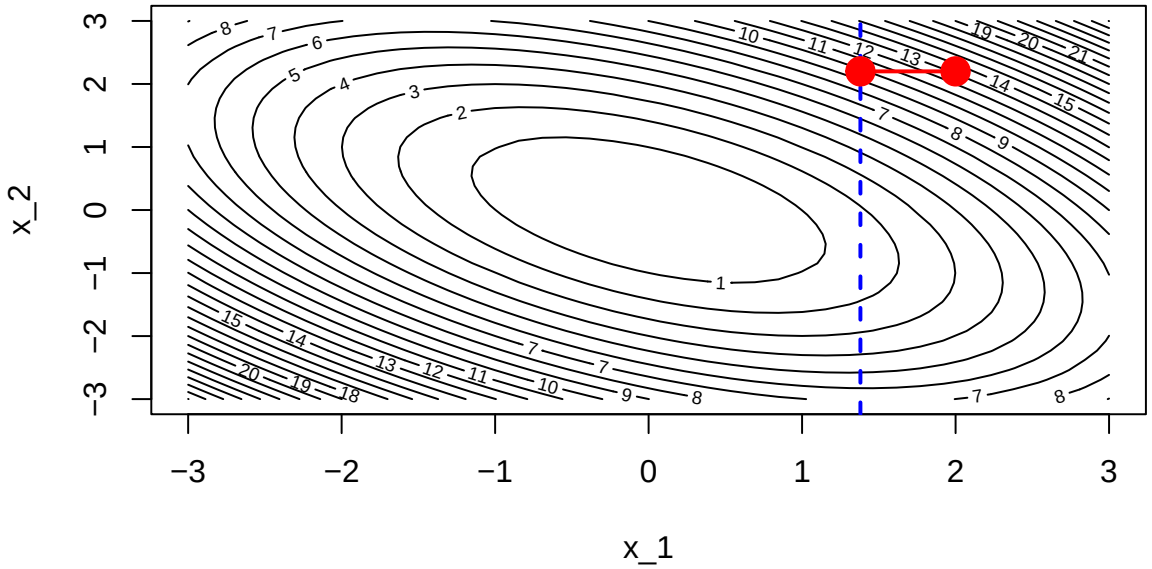

Figure 2.33: Optimisation in another dimension.

The first derivative of our function f with respect to x_2 writes:

$$\frac{\partial f}{\partial x_2}(x_1, x_2) = 2x_2 + x_1.$$

```
derivative_wrt_x2 <- function(theta){
  2*theta[2] + theta[1]
}
```

Evaluated at θ:

```
(grad_i <- derivative_wrt_x2(theta_update))
```

```
## [1] 5.78
```

The vector of parameters can then be updated:

```
theta_update[dim_i] <- theta_update[dim_i] - learning_rate * grad_i
theta_update
```

```
## [1] 1.380 1.622
```

Keeping track of the changes:

```
theta_values <- rbind(theta_values, theta_update)
theta_values
```

```
##                [,1]  [,2]
## theta          2.00 2.200
## theta_update   1.38 2.200
## theta_update   1.38 1.622
```

```
contour(x_1, x_1, z, nlevels = 20, xlab = "x_1", ylab = "x_2")
abline(v = theta_update[1], lty=2, col = "blue", lwd = 2)
for(i in 1:(nrow(theta_values)-1)){
  segments(x0 = theta_values[i, 1], x1 = theta_values[i+1, 1],
           y0 = theta_values[i, 2], y1 = theta_values[i+1, 2],
           col = "red", lwd=2)
}
points(theta_values[,1], theta_values[,2], pch = 19,
       cex = 2, col = "red")
```

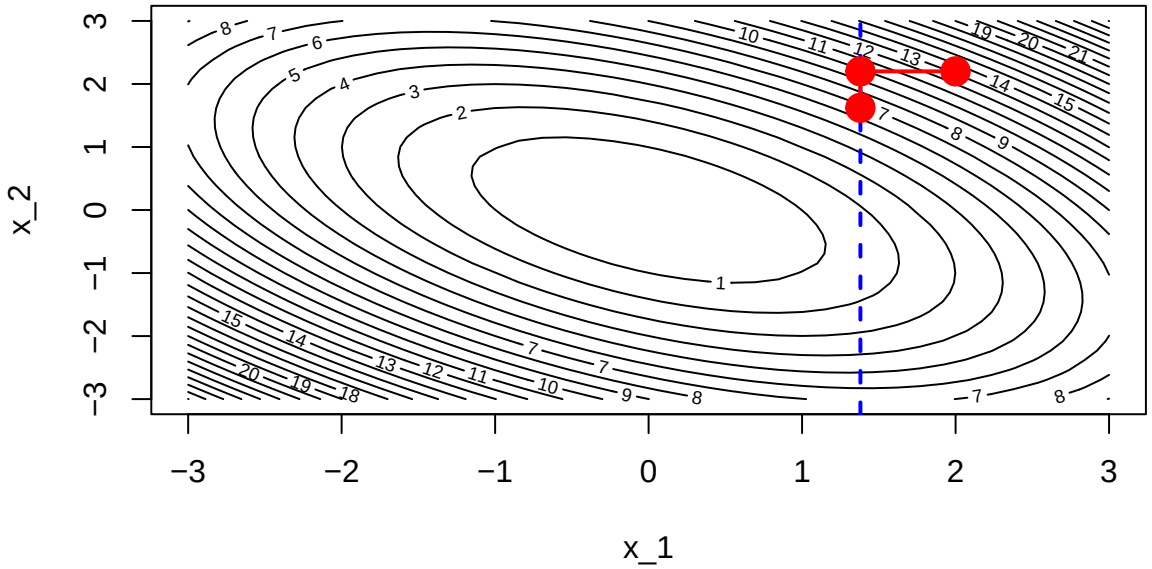

Figure 2.34: Updated value agter the second step.

Then we just need to iterate until a number of iterations is reached or until convergence.

Here is the full code:

```
# Starting values
theta <- c(2, 2.2)
learning_rate <- 10^-1
abstol <- 10^-5
nb_max_iter <- 100
z_current <- z_f_to_optim(theta)
# To keep track of what happens at each iteration
theta_values <- list(theta)
dims <- NULL

for(i in 1:nb_max_iter){

  nb_dim <- length(theta)
  # Cyclic rule to pick the dimension
  dim_i <- (i-1) %% nb_dim + 1

  # With uniform sampling
  # dim_i <- sample(x = seq_len(nb_dim), size = 1)
```

```
  # Steepest ascent
  if(dim_i == 1){
    grad_i <- derivative_wrt_x1(theta)
  }else{
    grad_i <- derivative_wrt_x2(theta)
  }

  # Updating the parameters
  theta_update <- theta
  theta_update[dim_i] <- theta_update[dim_i] - learning_rate * grad_i
  theta <- theta_update
  # To keep track of the changes
  theta_values <- c(theta_values, list(theta))
  dims <- c(dims, dim_i)

  # Checking for improvement
  z_updated <- z_f_to_optim(theta_update)
  if(abs(z_updated - z_current) < abstol) break
  z_current <- z_updated
}
```

The optimisation stopped at iteration:

```
i
```

```
## [1] 29
```

```
theta_values <- do.call("rbind", theta_values)
```

The final value for the parameter theta is:

```
theta
```

```
## [1] -0.03832741  0.07418966
```

Let us have a look at the path of the process:

```
contour(x_1, x_1, z, nlevels = 20, xlab = "x_1", ylab = "x_2")
abline(v = theta_update[1], lty=2, col = "blue", lwd = 2)
for(i in 1:(nrow(theta_values)-1)){
  segments(x0 = theta_values[i, 1], x1 = theta_values[i+1, 1],
           y0 = theta_values[i, 2], y1 = theta_values[i+1, 2],
           col = "red", lwd=2)
}
points(theta_values[,1], theta_values[,2], pch = 19,
       cex = .8, col = "red")
```

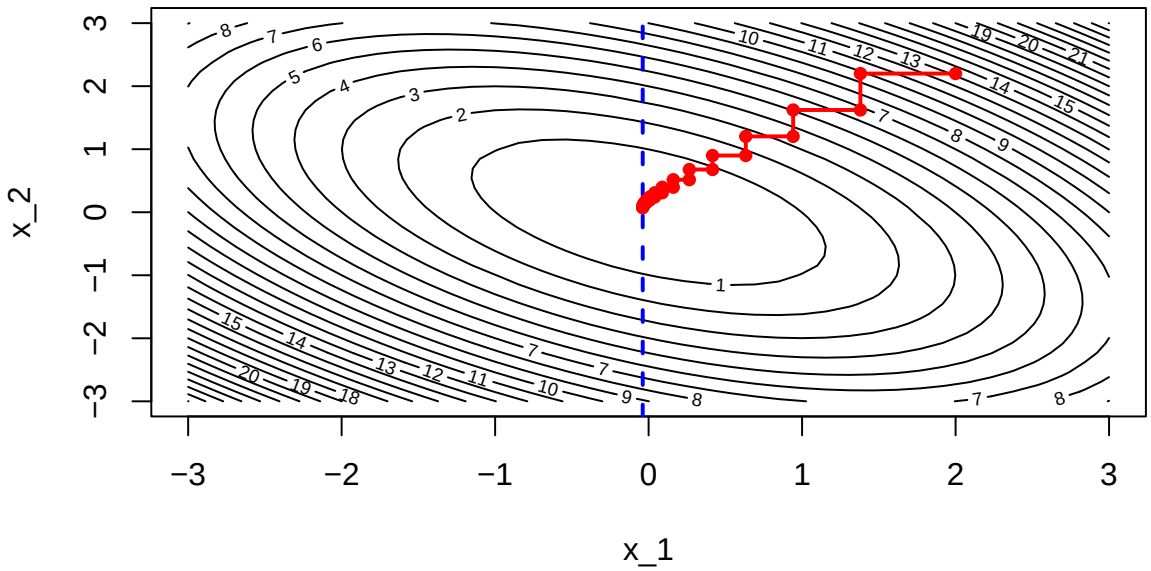

Figure 2.35: Coordinate descent algorithm: iterative process.

Or with the 3D graph:

```
saveGIF({

  for(j in c(rep(1,5), 2:(nrow(theta_values)-1), rep(nrow(theta_values), 10))){

    par(mar = c(1, 1, 1, 1))
    flip <- 1
    th <- 200
    pmat <-
```

```
      persp3D(x = x_1, y = x_2, z = z, colkey=F, contour=T,
              ticktype = "detailed", asp = 1, phi = 30, theta = th,
              border = "grey10", alpha=.4, d = .8,r = 2.8,
              expand = .6, shade = .2, axes = T, box = T, cex = .1,
              main = paste0("Step ", j))

    xx <- theta_values[1:j,1]
    yy <- theta_values[1:j,2]
    zz <- z_f(xx,yy)
    new_point <- trans3d(xx,yy,zz,pmat = pmat)
    lines(new_point,pch = 20,col = "red", cex=2, lwd=2)
    points(new_point,pch = 20,col = "red", cex=2)
  }

}, movie.name = "coordinate_descent_contour_3D.gif", interval = 0.01,
ani.width = 720, ani.height = 480)
```

The animated graph is not available in the PDF version of the hands-on session. Check to HTML version.

In this example, we have considered a cyclic sampling rule to decide the dimension to optimise over. Different rules exist. For example, one could use a uniform sampling rule.

2.3.2.1 When the Function to Optimize is not Differentiable in all Points

The coordinate descent algorithm will not be able to find the minimum of a non-differentiable function. The theorem we used in the case where f is convex AND differentiable cannot be used anymore.

It is however possible to find the minimum of some non-differentiable functions using a slightly modified version of the coordinate descent algorithm, if some conditions on the f function are met.

More precisely, if the function $f : \mathbb{R}^n \to \mathbb{R}$ is not differentiable at all points but can be written as follows:

$$f(\mathbf{x}) = g(\mathbf{x}) + \sum_{i=1}^{n} h_i(\mathbf{x}_i),$$

where g is convex and differentiable and where h_i is convex and non-differentiable, then:

$$f(\mathbf{x}) \leq f\left(\mathbf{x} + \delta \overrightarrow{\mathbf{e}}_i\right), \forall i \Rightarrow f(\mathbf{x}) = \min\{f\},$$

where $\overrightarrow{\mathbf{e}}_i = (0, \ldots, 0, 1, 0, \ldots, 0) \in \mathbb{R}^n$.

Let us consider the following function:

$$f(x_1, x_2) = \underbrace{x_1^2 + x_2^2}_{\text{convex and differentiable}} + \underbrace{|\, x_1 \,| + |\, x_2 \,|}_{\text{convex and non-differentiable}}$$

Let us visualise a graphical representation of this function:

```
n <- 25
x_1 <- x_2 <- seq(-3, 3, length.out=n)
z_f <- function(x_1, x_2) x_1^2+x_2^2 + abs(x_1) + abs(x_2)
z_f_to_optim <- function(theta)
  theta[1]^2+theta[2]^2 + abs(theta[1])+abs(theta[2])
z <- outer(x_1, x_2, z_f)
```

To compute the numerical partial first-order derivative with respect to the ith variable, we can create a function that will numerically estimate the value. Recall that the partial first-order derivative of function f is defined as:

$$\frac{\partial f}{\partial x_i}(\mathbf{x}) = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h},$$

where $\mathbf{x} = (x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n) \in \mathbb{R}^n$ and $\mathbf{e}_i = (0, \ldots, 0, 1, 0, \ldots, 0) \in \mathbb{R}^n$.

In R, we can define the following function to numerically estimate the first-order partial derivative:

```
#' Numerical partial first-order derivative of a function
#' @param par Initial values for the parameters
#' @param fn A function to be derived. It should return a scalar result.
#' @param dim Direction for the derivative (1 to compute the first derivative
#' with respect to the first parameter, 2 to compute the first derivative
#' with respect to the second parameter, etc.)
#' @param nb_dim number of dimensions
num_first_deriv <- function(par, fn, dim, nb_dim){
  h <- par[dim]*sqrt(10^-12)
  e <- rep(0, nb_dim) ; e[dim_i] <- 1
  (fn(par+h*e) - fn(par))/h
}
```

Then, we can run the coordinate descent algorithm:

```
# Starting values
theta <- c(2, 2.2)
learning_rate <- 10^-1
abstol <- 10^-6
nb_max_iter <- 500
z_current <- z_f_to_optim(theta)
# To keep track of what happens at each iteration
theta_values <- list(theta)
dims <- NULL


for(i in 1:nb_max_iter){

  nb_dim <- length(theta)
  # Cyclic rule to pick the dimension
  dim_i <- (i-1) %% nb_dim + 1

  # Partial derivative wrt to the dim_i axis
  grad_i <-
    num_first_deriv(par = theta, fn = z_f_to_optim,
                    dim = dim_i, nb_dim = nb_dim)

  # Updating the parameters
  theta_update <- theta
  theta_update[dim_i] <- theta_update[dim_i] - learning_rate * grad_i
  theta <- theta_update
  # To keep track of the changes
  theta_values <- c(theta_values, list(theta))
  dims <- c(dims, dim_i)

  # Checking for improvement
  z_updated <- z_f_to_optim(theta_update)
  if(abs(z_updated - z_current) < abstol) break
  z_current <- z_updated
}

theta
```

```
## [1] 0.05555447 0.05555593
```

```
theta_values <- do.call("rbind", theta_values)
```

Looking at the path followed by the updated parameters during the iterations:

```
contour(x_1, x_1, z, nlevels = 20, xlab = "x_1", ylab = "x_2")
for(i in 1:(nrow(theta_values)-1)){
  segments(x0 = theta_values[i, 1], x1 = theta_values[i+1, 1],
           y0 = theta_values[i, 2], y1 = theta_values[i+1, 2],
           col = "red", lwd=2)
}
points(theta_values[,1], theta_values[,2], pch = 19,
       cex = .8, col = "red")
```

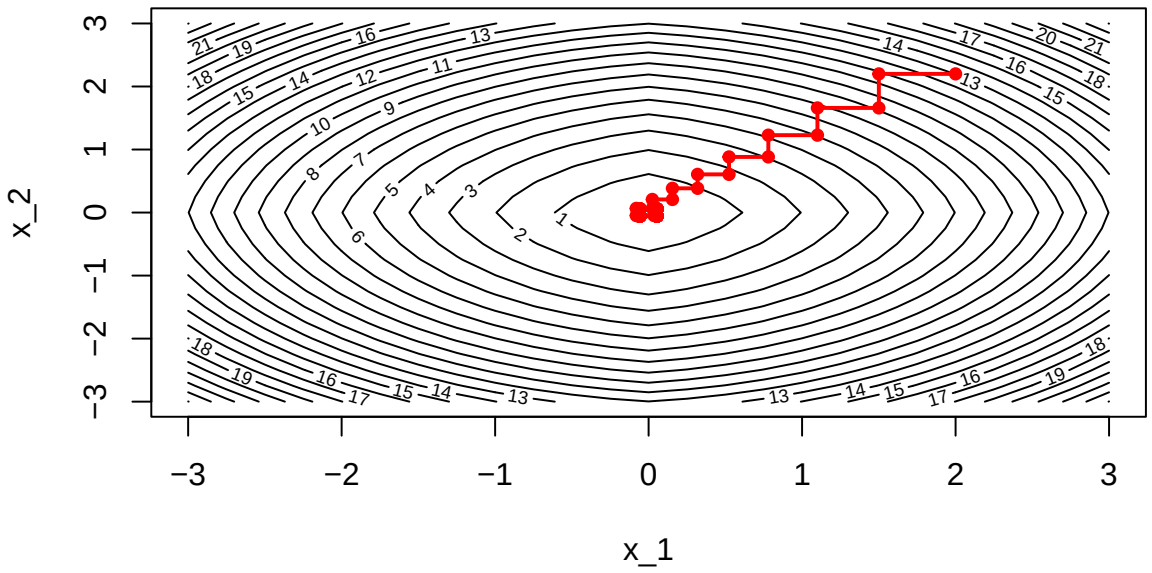

Figure 2.36: Coordinate descent: iterative process if the function is not differentiable in all points.

Chapter 3

Overfitting

This notebook illustrates the notion of overfitting through three examples. In the first example, the idea is to build a a classifier on some real data and to assess the predictive capabilities of the model on seen data, and then on unseen data. In the second example and third, the idea is to see how overfitting may lead us to select a hyperparameter of a machine learning model that may exhibit the best results on seen data but will produce a model that performs less well on unseen data.

Through these examples, the notion of cross validation will be recalled.

3.1 First Example: Default of Credit Card

Disclaimer: The idea of this session is taken from Demšar (2021).

In this first example, we will use a logistic regression to predict the default of credit card clients. The data used here come from Yeh and Lien (2009). They can be downloaded for free on the UCI Machine Learning Repository.

On the repository, the following information about the dataset is provided:

> This research aimed at the case of customers' default payments in Taiwan and compares the predictive accuracy of probability of default among six data mining methods. From the perspective of risk management, the result of predictive accuracy of the estimated probability of default will be more valuable than the binary result of classification - credible or not credible clients. Because the real probability of default is unknown, this study presented the novel "Sorting Smoothing Method" to estimate the real probability of default. With the real probability of default as the response variable (Y), and the predictive probability of default as the independent variable (X), the simple linear regression result (Y = A + BX) shows that the forecasting model

produced by artificial neural network has the highest coefficient of determination; its regression intercept (A) is close to zero, and regression coefficient (B) to one. Therefore, among the six data mining techniques, artificial neural network is the only one that can accurately estimate the real probability of default.

The **target variable** (y) is the default payment the next month (Yes = 1, No = 0). There are 23 features at hand.

- `LIMIT_BAL`: Amount of the given credit (NT dollar): it includes both the individual consumer credit and his/her family (supplementary) credit.
- `SEX`: Binary Coded Gender (1 = male, 2 = female)
- `EDUCATION`: Education (1 = graduate school; 2 = university; 3 = high school; 4 = others)
- `MARRIAGE`: Marital status (1 = married; 2 = single; 3 = others)
- `AGE`: Age (year)
- `PAY_0`, `PAY_1`, ..., `PAY_6`: History of past payment.
  - -1 = pay duly;
  - 1 = payment delay for one month;
  - 2 = payment delay for two months; ...
  - 8 = payment delay for eight months;
  - 9 = payment delay for nine months and above.

  Past monthly payment records (from April to September, 2005) tracked as follows:
  - `PAY_0`: repayment in September, 2005
  - `PAY_1`: repayment in August, 2005
  - ...
  - `PAY_6`: repayment in April, 2005
- `BILL_AMT1`, ..., `BILL_AMT6`: Amount of bill statement (NT dollar).
  - `BILL_AMT1`: in September, 2005
  - `BILL_AMT2`: in August, 2005
  - ...
  - `BILL_AMT6`: in April, 2005
- `PAY_AMT1`, ..., `PAY_AMT6`: Amount of previous payment (NT dollar).
  - `PAY_AMT1`: amount paid in September, 2005
  - `PAY_AMT2`: amount paid in August, 2005
  - ...
  - `PAY_AMT6`: amount paid in April, 2005

Let us load the data:

```
library(tidyverse)
library(readxl)
paste0(
  "https://egallic.fr/Enseignement/ML/ECB/",
  "data/default_credit_card_clients.xls") %>%
  download.file(destfile = "default_credit_card_clients.xls")

df <- read_excel("default_credit_card_clients.xls", skip = 1)
```

Let us rename the target variable:

```
df <-
  df %>%
  rename(y = `default payment next month`)
df
```

```
## # A tibble: 30,000 x 25
##       ID LIMIT_BAL   SEX EDUCATION MARRIAGE   AGE PAY_0 PAY_2
   PAY_3 PAY_4 PAY_5
##    <dbl>     <dbl> <dbl>     <dbl>    <dbl> <dbl> <dbl> <dbl> <
   dbl> <dbl> <dbl>
##  1     1     20000     2         2        1    24     2     2
   -1    -1    -2
##  2     2    120000     2         2        2    26    -1     2
       0     0     0
##  3     3     90000     2         2        2    34     0     0
       0     0     0
##  4     4     50000     2         2        1    37     0     0
       0     0     0
##  5     5     50000     1         2        1    57    -1     0
   -1     0     0
##  6     6     50000     1         1        2    37     0     0
       0     0     0
##  7     7    500000     1         1        2    29     0     0
       0     0     0
##  8     8    100000     2         2        2    23     0    -1
   -1     0     0
##  9     9    140000     2         3        1    28     0     0
       2     0     0
## 10    10     20000     1         3        2    35    -2    -2
   -2    -2    -1
## # ... with 29,990 more rows, and 14 more variables: PAY_6 <dbl>,
```

```
## #   BILL_AMT1 <dbl>, BILL_AMT2 <dbl>, BILL_AMT3 <dbl>, BILL_AMT4
    <dbl>,
## #   BILL_AMT5 <dbl>, BILL_AMT6 <dbl>, PAY_AMT1 <dbl>, PAY_AMT2 <
    dbl>,
## #   PAY_AMT3 <dbl>, PAY_AMT4 <dbl>, PAY_AMT5 <dbl>, PAY_AMT6 <dbl
    >, y <dbl>
```

3.1.1 Somme Summary Statistics on the Whole Dataset

Let us have a really quick glance at the dataset. For a more accurate estimate, data cleaning and feature engineering would be necessary. As this is not the point of this notebook, we will pretend not to have seen the issues that will be visible in the descriptive statistics.

Feel free to skip this part to get to the main point of the notebook.

The dimensions of the table are the following:

```
dim(df)
```

```
## [1] 30000    25
```

We note that the dataset is not balanced:

```
df %>%
  group_by(y) %>%
  count() %>%
  arrange(desc(n))
```

```
## # A tibble: 2 x 2
## # Groups:   y [2]
##       y     n
##   <dbl> <int>
## 1     0 23364
## 2     1  6636
```

Let us visualise this on a barplot:

```
df %>%
  mutate(y=factor(y, levels = c(1,0), labels = c("Yes", "No"))) %>%
```

```
  ggplot(data = .) +
  geom_bar(stat = "count", aes(y = y)) +
  labs(x = "Frequency", y = NULL) +
  theme(plot.title.position = "plot")
```

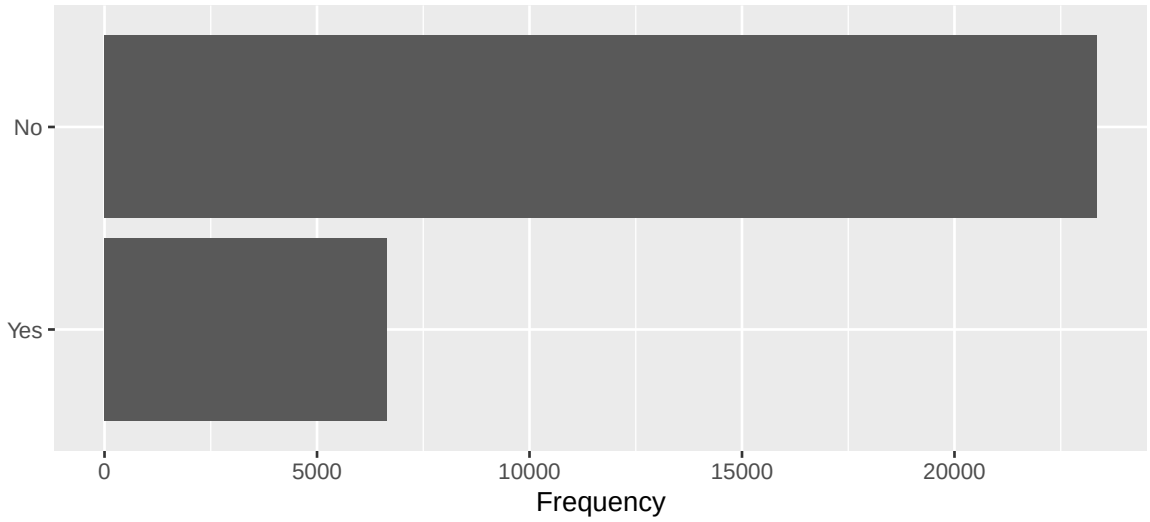

Figure 3.1: Default payment the next month.

We do not have thousands of predictors here, so we can have a little fun with the descriptive statistics. Let us create a table that associates a label to each variable and provides its type (numeric, qualitative or ordinal).

```
c(
  "y", "Default payment the next month", "qualitative",
  "ID", "ID number", "numeric",
  "LIMIT_BAL", "Amount of the given credit", "numeric",
  "SEX", "Gender", "qualitative",
  "EDUCATION", "Education", "ordinal",
  "MARRIAGE", "Marital status", "qualitative",
  "AGE", "Age", "numeric",
  "PAY_0", "Repayment status September", "qualitative",
  "PAY_2", "Repayment status August", "qualitative",
  "PAY_3", "Repayment status July", "qualitative",
  "PAY_4", "Repayment status June", "qualitative",
  "PAY_5", "Repayment status May", "qualitative",
```

```
  "PAY_6", "Repayment status April", "qualitative",
  "BILL_AMT1", "Bill amount September", "numeric",
  "BILL_AMT2", "Bill amount August", "numeric",
  "BILL_AMT3", "Bill amount July", "numeric",
  "BILL_AMT4", "Bill amount June", "numeric",
  "BILL_AMT5", "Bill amount May", "numeric",
  "BILL_AMT6", "Bill amount April", "numeric",
  "PAY_AMT1", "Amount previous payment September", "numeric",
  "PAY_AMT2", "Amount previous payment August", "numeric",
  "PAY_AMT3", "Amount previous payment July", "numeric",
  "PAY_AMT4", "Amount previous payment June", "numeric",
  "PAY_AMT5", "Amount previous payment May", "numeric",
  "PAY_AMT6", "Amount previous payment April", "numeric") %>%
  matrix(ncol = 3, byrow = TRUE) %>%
  as_tibble() %>%
  magrittr::set_colnames(c("variable", "label", "type")) ->
  variable_names_df
```

We can look at the values of our predictors depending on the response variable. Let us perform some basic parametric tests to see whether the distribution of each feature differs depending on the response variable.

```
#' format_p_value
#' Format the p-value so that it displays "<10^{i}", if i <= 3
#' rounds the value with 2 digits otherwise
#' @param x p-value to format
#' x <- 0.000133613 ; x <- 0.0011
format_p_value <- function(x){

  if(x < 10^-3){
    resul <- str_c("**< 10^{-3}**")
  }else if(x > 10^-3){
    if(x < 0.05){
      resul <- str_c("**", round(x, 3), "**")
    }else{
      resul <- round(x, 3)
    }
  }else{
    resul <- "**0.001**"
```

```
  }

  resul
}
```

We can create a rather big function to create a table with summary statistics, depending on the type of each variable and on the response variable.

```
#' Compute descriptive statistics on `df` grouped by `grouping_var`
#' depending on the type of the variable of interest
#' (qualitative, ordinal, numerical)
#' Then performs test of equality of mean/proportion on the sub-samples.
#'
#' @param variable (string) name of the variable of interest
#' @param grouping_var (string) name of the qualitative variable
#'                     that defines groups
#' @param variable_names table with three columns:
#'                       1- variable name, 2- label, 3-type
sample_diff <- function(variable, df,
                        grouping_var,
                        variable_names = variable_names_tlm){

  label_variable <-
    variable_names %>% filter(variable == !!variable) %>%
    magrittr::extract2("label")
  type <-
    variable_names %>% filter(variable == !!variable) %>%
    magrittr::extract2("type")

  if(type %in% c("qualitative", "ordinal")){

    whole_sample <-
      df %>%
      filter(!is.na(!!!syms(variable)), !is.na(!!!syms(grouping_var))) %>%
      mutate(label = label_variable) %>%
      select(label, !!!variable) %>%
      group_by(label, !!!syms(variable)) %>%
      summarise(n = n(), .groups = "drop")
```

```
    nb_tot <- sum(whole_sample$n)

    whole_sample <-
      whole_sample %>%
      mutate(pct = n / sum(n),
             pct = (round(100*pct, 2)) %>% str_c("(", ., "%)")) %>%
      unite(`Whole Sample`, n, pct, sep = " ")

    subsamples <-
      df %>%
      filter(!is.na(!!!syms(variable)), !is.na(!!!syms(grouping_var))) %>%
      mutate(label = label_variable) %>%
      select(label, !!!variable, !!!grouping_var) %>%
      group_by(label, !!!syms(variable), !!!(syms(grouping_var))) %>%
      summarise(n = n(), .groups = "drop") %>%
      group_by(!!!syms(grouping_var)) %>%
      mutate(pct = n / sum(n),
             pct = (round(100*pct, 2)) %>% str_c("(", ., "%)")) %>%
      unite(value, n, pct, sep = " ")

    res <-
      whole_sample %>%
      left_join(subsamples) %>%
      ungroup() %>%
      spread_(key_col = grouping_var,  value_col = "value",
              fill = "0 (0.00%)") %>%
      mutate(label = ifelse(
        row_number() == 1,
        yes = str_c(label,
                    " (n = ", format(nb_tot, big.mark = ","), ")"),
        no = label)) %>%
      unite(label, label, !!!variable, sep = " ")

  }else{

    whole_sample <-
      df %>%
      filter(!is.na(!!!syms(variable)), !is.na(!!!syms(grouping_var))) %>%
      mutate(label = label_variable) %>%
```

```
    select(label, !!!variable)

nb_tot <- nrow(whole_sample)

whole_sample <-
  whole_sample %>%
  group_by(label) %>%
  summarise(mean = mean(!!!syms(variable), na.rm = TRUE) %>%
              formatC(format = "e", digits = 2),
            sd = sd(!!!syms(variable), na.rm = TRUE) %>%
              formatC(format = "e", digits = 2),
            .groups = "drop") %>%
  mutate(sd = str_c("(+/-", sd, ")")) %>%
  unite(`Whole Sample`, mean, sd, sep = " ")


subsamples <-
  df %>%
  filter(!is.na(!!!syms(variable)), !is.na(!!!syms(grouping_var))) %>%
  mutate(label = label_variable) %>%
  select(label, !!!variable, !!!grouping_var) %>%
  group_by(label, !!!syms(grouping_var)) %>%
  summarise(mean = mean(!!!syms(variable)) %>%
              formatC(format = "e", digits = 2),
            sd = sd(!!!syms(variable)) %>%
              formatC(format = "e", digits = 2),
            .groups = "drop") %>%
  mutate(sd = str_c("(+/-", sd, ")")) %>%
  unite(value, mean, sd, sep = " ")

res <-
  whole_sample %>%
  left_join(subsamples) %>%
  ungroup() %>%
  spread_(key_col = grouping_var,  value_col = "value", fill = "0 (+/-0.00)") %>%
  mutate(label = ifelse(
    row_number() == 1,
    yes = str_c(label,
                " (n = ", format(nb_tot, big.mark = ","), ")"),
```

```
        no = label))
  }

  if(type == "qualitative"){
    chisq_test <- chisq.test(df %>% magrittr::extract2(grouping_var),
                             df %>% magrittr::extract2(variable))
    p_value <- chisq_test$p.value
  }else if(type == "ordinal"){
    kruskal_test <- kruskal.test(str_c(grouping_var, "~", variable) %>%
                                   as.formula(), data = df)
    p_value <- kruskal_test$p.value
  }else{
    # Numerical
    form <- str_c(variable, " ~ ", grouping_var) %>% as.formula()
    anov <- aov(form, data = df)
    anov_summary <-  anov %>% summary() %>% .[[1]]
    p_value <- anov_summary$`Pr(>F)`[1]
  }

  # Adding stars for p-values
  p_value <- format_p_value(p_value)

  resul <- res %>%
    mutate(`p value` = c(rep("", nrow(res)-1), p_value))
  resul
}# End of sample_diff()
```

Let us apply this function to all our predictors:

```
df_summary_stats <-
  variable_names_df$variable[!variable_names_df$variable%in%c("y", "ID")]%>%
  map_df(sample_diff, df = df,
         grouping_var = "y", variable_names = variable_names_df)
```

Now we can print the summary table:

```
library(kableExtra)
df_summary_stats %>%
  kableExtra::kable(caption = "Driving habits before and after the claim",
```

```
                  format = "latex", longtable = TRUE) %>%
kableExtra::kable_classic(full_width = F, html_font = "Cambria") %>%
kableExtra::kable_styling(
  bootstrap_options = c("striped", "hover", "condensed", "responsive"),
  font_size = 6)
```

Table 3.1: Driving habits before and after the claim

label	Whole Sample	0	1	p value
Amount of the given credit (n = 30,000)	1.67e+05 (+/-1.30e+05)	1.78e+05 (+/-1.32e+05)	1.30e+05 (+/-1.15e+05)	**< 10^{-3}**
Gender (n = 30,000) 1	11888 (39.63%)	9015 (38.59%)	2873 (43.29%)	
Gender 2	18112 (60.37%)	14349 (61.41%)	3763 (56.71%)	**< 10^{-3}**
Education (n = 30,000) 0	14 (0.05%)	14 (0.06%)	0 (0.00%)	
Education 1	10585 (35.28%)	8549 (36.59%)	2036 (30.68%)	
Education 2	14030 (46.77%)	10700 (45.8%)	3330 (50.18%)	
Education 3	4917 (16.39%)	3680 (15.75%)	1237 (18.64%)	
Education 4	123 (0.41%)	116 (0.5%)	7 (0.11%)	
Education 5	280 (0.93%)	262 (1.12%)	18 (0.27%)	
Education 6	51 (0.17%)	43 (0.18%)	8 (0.12%)	**< 10^{-3}**
Marital status (n = 30,000) 0	54 (0.18%)	49 (0.21%)	5 (0.08%)	
Marital status 1	13659 (45.53%)	10453 (44.74%)	3206 (48.31%)	
Marital status 2	15964 (53.21%)	12623 (54.03%)	3341 (50.35%)	
Marital status 3	323 (1.08%)	239 (1.02%)	84 (1.27%)	**< 10^{-3}**
Age (n = 30,000)	3.55e+01 (+/-9.22e+00)	3.54e+01 (+/-9.08e+00)	3.57e+01 (+/-9.69e+00)	**0.016**
Repayment status September (n = 30,000) -2	2759 (9.2%)	2394 (10.25%)	365 (5.5%)	
Repayment status September -1	5686 (18.95%)	4732 (20.25%)	954 (14.38%)	
Repayment status September 0	14737 (49.12%)	12849 (54.99%)	1888 (28.45%)	
Repayment status September 1	3688 (12.29%)	2436 (10.43%)	1252 (18.87%)	
Repayment status September 2	2667 (8.89%)	823 (3.52%)	1844 (27.79%)	
Repayment status September 3	322 (1.07%)	78 (0.33%)	244 (3.68%)	
Repayment status September 4	76 (0.25%)	24 (0.1%)	52 (0.78%)	
Repayment status September 5	26 (0.09%)	13 (0.06%)	13 (0.2%)	
Repayment status September 6	11 (0.04%)	5 (0.02%)	6 (0.09%)	
Repayment status September 7	9 (0.03%)	2 (0.01%)	7 (0.11%)	
Repayment status September 8	19 (0.06%)	8 (0.03%)	11 (0.17%)	**< 10^{-3}**
Repayment status August (n = 30,000) -2	3782 (12.61%)	3091 (13.23%)	691 (10.41%)	
Repayment status August -1	6050 (20.17%)	5084 (21.76%)	966 (14.56%)	
Repayment status August 0	15730 (52.43%)	13227 (56.61%)	2503 (37.72%)	
Repayment status August 1	28 (0.09%)	23 (0.1%)	5 (0.08%)	
Repayment status August 2	3927 (13.09%)	1743 (7.46%)	2184 (32.91%)	
Repayment status August 3	326 (1.09%)	125 (0.54%)	201 (3.03%)	
Repayment status August 4	99 (0.33%)	49 (0.21%)	50 (0.75%)	
Repayment status August 5	25 (0.08%)	10 (0.04%)	15 (0.23%)	
Repayment status August 6	12 (0.04%)	3 (0.01%)	9 (0.14%)	
Repayment status August 7	20 (0.07%)	8 (0.03%)	12 (0.18%)	
Repayment status August 8	1 (0%)	1 (0%)	0 (0.00%)	**< 10^{-3}**
Repayment status July (n = 30,000) -2	4085 (13.62%)	3328 (14.24%)	757 (11.41%)	
Repayment status July -1	5938 (19.79%)	5012 (21.45%)	926 (13.95%)	
Repayment status July 0	15764 (52.55%)	13013 (55.7%)	2751 (41.46%)	
Repayment status July 1	4 (0.01%)	3 (0.01%)	1 (0.02%)	
Repayment status July 2	3819 (12.73%)	1850 (7.92%)	1969 (29.67%)	
Repayment status July 3	240 (0.8%)	102 (0.44%)	138 (2.08%)	
Repayment status July 4	76 (0.25%)	32 (0.14%)	44 (0.66%)	
Repayment status July 5	21 (0.07%)	9 (0.04%)	12 (0.18%)	
Repayment status July 6	23 (0.08%)	9 (0.04%)	14 (0.21%)	
Repayment status July 7	27 (0.09%)	5 (0.02%)	22 (0.33%)	
Repayment status July 8	3 (0.01%)	1 (0%)	2 (0.03%)	**< 10^{-3}**
Repayment status June (n = 30,000) -2	4348 (14.49%)	3511 (15.03%)	837 (12.61%)	
Repayment status June -1	5687 (18.96%)	4783 (20.47%)	904 (13.62%)	
Repayment status June 0	16455 (54.85%)	13439 (57.52%)	3016 (45.45%)	

Repayment status June 1	2 (0.01%)	1 (0%)	1 (0.02%)	
Repayment status June 2	3159 (10.53%)	1506 (6.45%)	1653 (24.91%)	
Repayment status June 3	180 (0.6%)	70 (0.3%)	110 (1.66%)	
Repayment status June 4	69 (0.23%)	23 (0.1%)	46 (0.69%)	
Repayment status June 5	35 (0.12%)	17 (0.07%)	18 (0.27%)	
Repayment status June 6	5 (0.02%)	3 (0.01%)	2 (0.03%)	
Repayment status June 7	58 (0.19%)	10 (0.04%)	48 (0.72%)	
Repayment status June 8	2 (0.01%)	1 (0%)	1 (0.02%)	**< 10^{-3}**
Repayment status May (n = 30,000) -2	4546 (15.15%)	3651 (15.63%)	895 (13.49%)	
Repayment status May -1	5539 (18.46%)	4642 (19.87%)	897 (13.52%)	
Repayment status May 0	16947 (56.49%)	13752 (58.86%)	3195 (48.15%)	
Repayment status May 2	2626 (8.75%)	1203 (5.15%)	1423 (21.44%)	
Repayment status May 3	178 (0.59%)	65 (0.28%)	113 (1.7%)	
Repayment status May 4	84 (0.28%)	33 (0.14%)	51 (0.77%)	
Repayment status May 5	17 (0.06%)	7 (0.03%)	10 (0.15%)	
Repayment status May 6	4 (0.01%)	1 (0%)	3 (0.05%)	
Repayment status May 7	58 (0.19%)	10 (0.04%)	48 (0.72%)	
Repayment status May 8	1 (0%)	0 (0.00%)	1 (0.02%)	**< 10^{-3}**
Repayment status April (n = 30,000) -2	4895 (16.32%)	3914 (16.75%)	981 (14.78%)	
Repayment status April -1	5740 (19.13%)	4765 (20.39%)	975 (14.69%)	
Repayment status April 0	16286 (54.29%)	13217 (56.57%)	3069 (46.25%)	
Repayment status April 2	2766 (9.22%)	1365 (5.84%)	1401 (21.11%)	
Repayment status April 3	184 (0.61%)	66 (0.28%)	118 (1.78%)	
Repayment status April 4	49 (0.16%)	18 (0.08%)	31 (0.47%)	
Repayment status April 5	13 (0.04%)	6 (0.03%)	7 (0.11%)	
Repayment status April 6	19 (0.06%)	5 (0.02%)	14 (0.21%)	
Repayment status April 7	46 (0.15%)	8 (0.03%)	38 (0.57%)	
Repayment status April 8	2 (0.01%)	0 (0.00%)	2 (0.03%)	**< 10^{-3}**
Bill amount September (n = 30,000)	5.12e+04 (+/-7.36e+04)	5.20e+04 (+/-7.36e+04)	4.85e+04 (+/-7.38e+04)	**< 10^{-3}**
Bill amount August (n = 30,000)	4.92e+04 (+/-7.12e+04)	4.97e+04 (+/-7.10e+04)	4.73e+04 (+/-7.17e+04)	**0.014**
Bill amount July (n = 30,000)	4.70e+04 (+/-6.93e+04)	4.75e+04 (+/-6.96e+04)	4.52e+04 (+/-6.85e+04)	**0.015**
Bill amount June (n = 30,000)	4.33e+04 (+/-6.43e+04)	4.36e+04 (+/-6.43e+04)	4.20e+04 (+/-6.44e+04)	0.079
Bill amount May (n = 30,000)	4.03e+04 (+/-6.08e+04)	4.05e+04 (+/-6.06e+04)	3.95e+04 (+/-6.14e+04)	0.242
Bill amount April (n = 30,000)	3.89e+04 (+/-5.96e+04)	3.90e+04 (+/-5.95e+04)	3.83e+04 (+/-5.96e+04)	0.352
Amount previous payment September (n = 30,000)	5.66e+03 (+/-1.66e+04)	6.31e+03 (+/-1.80e+04)	3.40e+03 (+/-9.54e+03)	**< 10^{-3}**
Amount previous payment August (n = 30,000)	5.92e+03 (+/-2.30e+04)	6.64e+03 (+/-2.53e+04)	3.39e+03 (+/-1.17e+04)	**< 10^{-3}**
Amount previous payment July (n = 30,000)	5.23e+03 (+/-1.76e+04)	5.75e+03 (+/-1.87e+04)	3.37e+03 (+/-1.30e+04)	**< 10^{-3}**
Amount previous payment June (n = 30,000)	4.83e+03 (+/-1.57e+04)	5.30e+03 (+/-1.67e+04)	3.16e+03 (+/-1.12e+04)	**< 10^{-3}**
Amount previous payment May (n = 30,000)	4.80e+03 (+/-1.53e+04)	5.25e+03 (+/-1.61e+04)	3.22e+03 (+/-1.19e+04)	**< 10^{-3}**
Amount previous payment April (n = 30,000)	5.22e+03 (+/-1.78e+04)	5.72e+03 (+/-1.88e+04)	3.44e+03 (+/-1.35e+04)	**< 10^{-3}**

3.1.2 Fitting the Model

Let us take a sample of the data such that the number of predictors is about the same order of magnitude as the number of rows. We will create a dataset with only 50 observations here.

```
set.seed(134)
# Number of observations
n <- 50
df_first_example <-
  sample_n(df %>% filter(y == 0), size = n/2, replace= FALSE) %>%
  bind_rows(
    sample_n(df %>% filter(y == 1), size = n/2, replace= FALSE)
  ) %>%
  select(-ID)
```

We will use a logistic regression to try to predict the credit card default clients. Please note that we do it very naively here, without considering, for example, that some data are categorical.

Let us use all the 23 predictors in the model.

```
reg <- glm(y ~ ., data = df_first_example, family = "binomial")
```

For convenience, we can define two functions:

1. `confusion_table`: a function that constructs the confusion table
2. `compute_metric_confusion`: a function that computes errors metrics from the confusion matrix

```
#' Get the confusion table
#' @param observed vector of observed values
#' @param predicted vector of predicted values
confusion_table <- function(observed, predicted){
  confusion_matrix <- table(observed, predicted)
  confusion_matrix_prop <- prop.table(confusion_matrix)
  confusion_matrix_print <- confusion_matrix
  for(i in 1:ncol(confusion_matrix_print)){
    for(j in 1:nrow(confusion_matrix_print)){
      confusion_matrix_print[i,j] <-
        str_c(confusion_matrix_print[i,j], " (",
              confusion_matrix_prop[i,j], "%)")
    }
  }
  metrics <- compute_metric_confusion(confusion_matrix)
  list(confusion_matrix = confusion_matrix,
       confusion_matrix_print = confusion_matrix_print,
       metrics = metrics)
}
```

```
#' Computes errors metrics from the confusion matrix
#' @param confusion_matrix Confusion matrix
compute_metric_confusion <- function(confusion_matrix){
  true_negative <- confusion_matrix["0", "0"]
  if(is.null(true_negative)) true_negative <- 0

  false_positive <- confusion_matrix["0", "1"]
  if(is.null(false_positive)) false_positive <- 0
```

```
  false_negative <- confusion_matrix["1", "0"]
  if(is.null(false_negative)) false_negative <- 0

  true_positive <- confusion_matrix["1", "1"]
  if(is.null(true_positive)) true_positive <- 0

  error_rate <- (false_positive+false_negative)/sum(confusion_matrix)

  # True positive rate
  TPR <- true_positive / (true_positive + false_negative)
  # False positive rate
  FPR <- false_positive / (false_positive + true_negative)
  # True negative rate
  TNR <- true_negative / (true_negative + false_positive)
  # False negative rate
  FNR <- false_negative / (false_negative + true_positive)
  # Overall error rate
  OER <- (false_negative + false_positive) /
    (false_negative + false_positive + true_positive + true_negative)

  list(`Erorr rate` = error_rate,
       `True positive rate` = TPR,
       `False positive rate` = FPR,
       `True negative rate` = TNR,
       `False negative rate` = FNR,
       `Overall error rate` = OER
  )
}
```

Now, let us have a look at the predictive capabilities of our model:

```
# Predictions:
threshold <- .5
pred_prob <- predict(reg)
pred <- ifelse(pred_prob>.5, 1, 0)
```

We can easily obtain the confusion matrix and some error metrics with our function:

```
confusion_matrix <- confusion_table(df_first_example$y, pred)
```

Here is the confusion matrix:

```
confusion_matrix$confusion_matrix_print
```

```
##          predicted
## observed 0          1
##        0 23 (0.46%) 2 (0.04%)
##        1 6 (0.12%)  19 (0.38%)
```

And the metrics:

```
confusion_matrix$metrics
```

```
## $`Erorr rate`
## [1] 0.16
##
## $`True positive rate`
## [1] 0.76
##
## $`False positive rate`
## [1] 0.08
##
## $`True negative rate`
## [1] 0.92
##
## $`False negative rate`
## [1] 0.24
##
## $`Overall error rate`
## [1] 0.16
```

With 50 observations and 23 regressors, we do not make that many mistakes.

3.1.3 Randomly Assigning the Classes

Now, let us try something. We will keep the observations unchanged, excepts for the value of the target value. The latter will be randomly drawn. We will store the new target variable in a column named `y_new` in the data table.

```
set.seed(123)
df_first_example <-
  df_first_example %>%
  mutate(y_new = sample(c(0, 1), size = n(), replace = TRUE))
```

```
df_first_example$y
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
    1 1 1 1 1 1 1 1
## [39] 1 1 1 1 1 1 1 1 1 1 1 1
```

```
df_first_example$y_new
```

```
##  [1] 0 0 0 1 0 1 1 1 0 0 1 1 1 0 1 0 1 0 0 0 0 1 0 0 0 0 1 1 0 1
    0 1 0 1 1 0 0 0
## [39] 0 1 0 1 1 0 0 0 0 1 0 0
```

```
# Half of the "observed" values have been modified here
sum(df_first_example$y == df_first_example$y_new)
```

```
## [1] 25
```

Let us train the model again, this time on this randomly drawn response:

```
reg_2 <-
  glm(y_new ~ ., data = df_first_example %>% select(-y),
      family = "binomial")
```

And we can now have a look at the predicting capabilities of the model:

```
pred_prob_2 <- predict(reg_2)
pred_2 <- ifelse(pred_prob_2>.5, 1, 0)
conf_matrix_2 <- confusion_table(df_first_example$y_new, pred_2)
conf_matrix_2$confusion_matrix_print
```

```
##          predicted
## observed 0          1
##        0 25 (0.5%) 5 (0.1%)
##        1 6 (0.12%) 14 (0.28%)
```

```
conf_matrix_2$metrics
```

```
## $`Erorr rate`
## [1] 0.22
##
## $`True positive rate`
## [1] 0.7
##
## $`False positive rate`
## [1] 0.1666667
##
## $`True negative rate`
## [1] 0.8333333
##
## $`False negative rate`
## [1] 0.3
##
## $`Overall error rate`
## [1] 0.22
```

The accuracy of the model is still quite good... Maybe this is due to our sampling? Let us repeat this random switching of the "true" class. At each iteration, we will store the results in a list.

```
random_fit <- function(){
  df_first_example <-
    df_first_example %>%
    mutate(y_new = sample(c(0, 1), size = n(), replace = TRUE))

  # Let us train the model again, this time on this randomly drawn response
  reg_2 <-
    glm(y_new ~ ., data = df_first_example %>% select(-y),
        family = "binomial")

  pred_prob_2 <- predict(reg_2)
  pred_2 <- ifelse(pred_prob_2>.5, 1, 0)
```

```
  conf_matrix_2 <- confusion_table(df_first_example$y_new, pred_2)
  conf_matrix_2
}

results_random <- vector(mode = "list", length = 100)
# pb <- txtProgressBar(min = 1, max = 100, style = 3)
for(i in 1:100){
  results_random[[i]] <- random_fit()
  # setTxtProgressBar(pb, i)
}
```

The overall error rates of each of the 100 repetitions can be accessed the following way:

```
error_rates <-
  map(results_random, "metrics") %>%
  map_dbl("Overall error rate")
```

Here are some summary statistics of these error rates:

```
summary(error_rates)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.0750  0.1600  0.1386  0.2000  0.3600
```

Let us visualise those with a boxplot:

```
ggplot(data = tibble(error_rate = error_rates)) +
  geom_boxplot(aes(x =error_rate)) +
  labs(x = "Overall error rate")
```

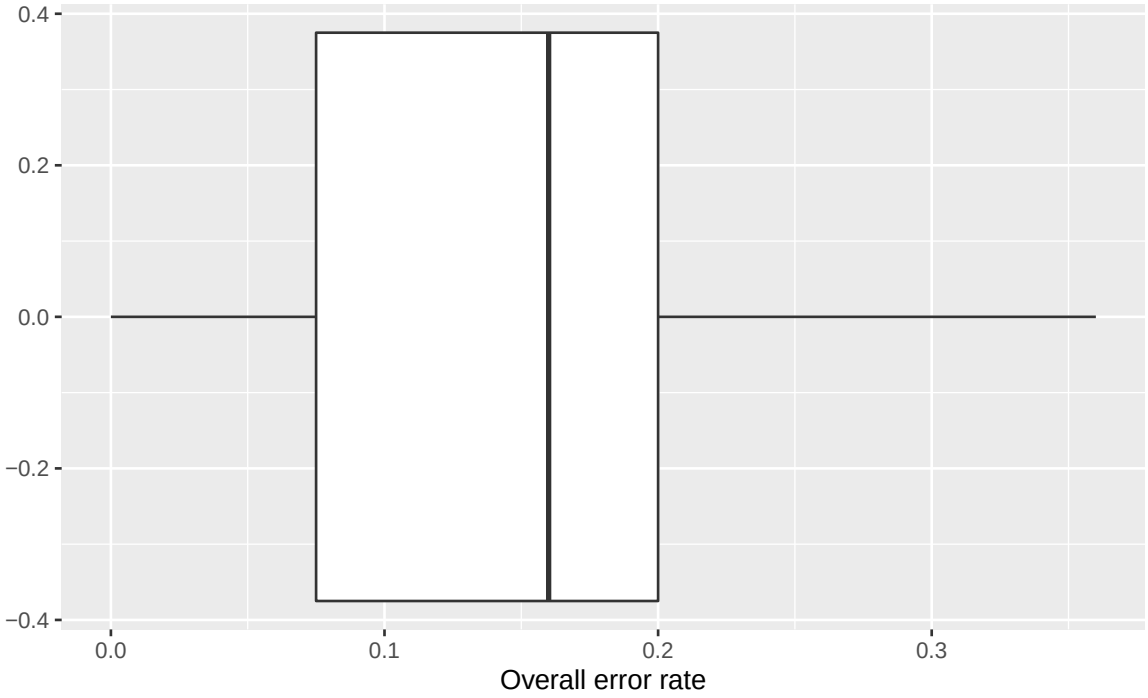

Figure 3.2: Overall error rate over 100 repetitions.

What these error show is that the model is able to "memorize" the data and is able to reproduce not so badly the response variable. We face here some overfitting. With a more flexible model, the error rate could even be equal to zero.

3.1.4 Performances on Unseen Data

Let us go back to our previous regression and let us use our model to make predictions on unseen data.

First, we need some unseen data:

```
df_test <-
  sample_n(df %>% filter(y == 0), size = n/2, replace= FALSE) %>%
  bind_rows(
    sample_n(df %>% filter(y == 1), size = n/2, replace= FALSE)
  ) %>%
  select(-ID)
```

The predictions can be obtained as follows:

```
pred_prob_test <- predict(reg, newdata=df_test)
pred_test <- ifelse(pred_prob_test>.5, 1, 0)
```

The confusion matrix on these unseen data is obtained with the following instructions:

```
confusion_matrix_test <- confusion_table(df_test$y, pred_test)
confusion_matrix_test$confusion_matrix_print
```

```
##          predicted
## observed 0          1
##        0 17 (0.34%) 8 (0.16%)
##        1 16 (0.32%) 9 (0.18%)
```

And here are some metrics obtained thanks to this confusion matrix:

```
confusion_matrix_test$metrics
```

```
## $`Erorr rate`
## [1] 0.48
##
## $`True positive rate`
## [1] 0.36
##
## $`False positive rate`
## [1] 0.32
##
## $`True negative rate`
## [1] 0.68
##
## $`False negative rate`
## [1] 0.64
##
## $`Overall error rate`
## [1] 0.48
```

The predictive ability of the model worsened.

Could it be bad luck?

```
set.seed(123)
# Sample of size 200
n <- 200
df_first_example_2 <-
  sample_n(df %>% filter(y == 0), size = n/2, replace= FALSE) %>%
  bind_rows(
    sample_n(df %>% filter(y == 1), size = n/2, replace= FALSE)
  ) %>%
  select(-ID)
```

Splitting into train/test to evaluate the capacity of the model to generalise its results on unseen data:

```
n_train <- round(.8*nrow(df_first_example_2))
ind_train <- sample(1:nrow(df_first_example_2),
                    size = n_train, replace = FALSE)

df_train <- df_first_example_2 %>% slice(ind_train)
df_test <- df_first_example_2 %>% slice(-ind_train)
```

The dimensions of the training and of the testing datasets:

```
dim(df_train)
```

```
## [1] 160  24
```

```
dim(df_test)
```

```
## [1] 40 24
```

Let us fit the model on the training set:

```
reg_train <- glm(y ~ ., data = df_train, family = "binomial")
```

The predictions on the training dataset (in-sample predictions):

```
pred_train_prob <- predict(reg_train)
pred_train <- ifelse(pred_train_prob>.5, 1, 0)
```

And the confusion matrix:

```
confusion_matrix_train <- confusion_table(df_train$y, pred_train)
confusion_matrix_train$confusion_matrix_print
```

```
##          predicted
## observed 0             1
##        0 68 (0.425%)   14 (0.0875%)
##        1 41 (0.25625%) 37 (0.23125%)
```

```
confusion_matrix_train$metrics
```

```
## $`Erorr rate`
## [1] 0.34375
##
## $`True positive rate`
## [1] 0.474359
##
## $`False positive rate`
## [1] 0.1707317
##
## $`True negative rate`
## [1] 0.8292683
##
## $`False negative rate`
## [1] 0.525641
##
## $`Overall error rate`
## [1] 0.34375
```

Let us see our performances on the test set:

```
pred_test_prob <- predict(reg_train, newdata = df_test)
pred_test <- ifelse(pred_test_prob>.5, 1, 0)
confusion_matrix_test <- confusion_table(df_test$y, pred_test)
confusion_matrix_test$confusion_matrix_print
```

```
##          predicted
## observed 0          1
##        0 12 (0.3%) 6 (0.15%)
##        1 12 (0.3%) 10 (0.25%)
```

```
confusion_matrix_test$metrics
```

```
## $`Erorr rate`
## [1] 0.45
##
## $`True positive rate`
## [1] 0.4545455
##
## $`False positive rate`
## [1] 0.3333333
##
## $`True negative rate`
## [1] 0.6666667
##
## $`False negative rate`
## [1] 0.5454545
##
## $`Overall error rate`
## [1] 0.45
```

We will repeat this procedure 100 times. To make things easier, we can define a function that randomly constitutes a training and a testing datasets from `df_first_example_2`, fits the model and then returns the confusion matrices as well as the goodness of fit metrics.

```
train_test_estim <- function(){

  n_train <- round(.8*nrow(df_first_example_2))
  ind_train <-
    sample(1:nrow(df_first_example_2), size = n_train, replace = FALSE)

  df_train <- df_first_example_2 %>% slice(ind_train)
  df_test <- df_first_example_2 %>% slice(-ind_train)

  reg_train <- glm(y ~ ., data = df_train, family = "binomial")
  # Predictions on the train set
```

```
  pred_train_prob <- predict(reg_train)
  pred_train <- ifelse(pred_train_prob>.5, 1, 0)
  # On the test set
  pred_test_prob <- predict(reg_train, newdata = df_test)
  pred_test <- ifelse(pred_test_prob>.5, 1, 0)

  # Confusion matrices
  confusion_matrix_train <- confusion_table(df_train$y, pred_train)
  confusion_matrix_test <- confusion_table(df_test$y, pred_test)

  list(confusion_matrix_train = confusion_matrix_train,
       confusion_matrix_test = confusion_matrix_test)

}
```

The function can be applied over a loop. At each iteration, we can store the results.

```
results_random_2 <- vector(mode = "list", length = 100)
# pb <- txtProgressBar(min = 1, max = 100, style = 3)
for(i in 1:100){
  results_random_2[[i]] <- train_test_estim()
  # setTxtProgressBar(pb, i)
}
```

The overall error rates obtained on both sets over the 100 iterations can be obtained as follows:

```
error_rates_train_2 <-
  map(results_random_2, "confusion_matrix_train") %>%
  map("metrics") %>%
  map_dbl("Overall error rate")

error_rates_test_2 <-
  map(results_random_2, "confusion_matrix_test") %>%
  map("metrics") %>%
  map_dbl("Overall error rate")
```

```
summary(error_rates_train_2)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
```

```
##   0.2437   0.3047   0.3250   0.3225   0.3438   0.4000
```

```
summary(error_rates_test_2)
```

```
##     Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.3000  0.3750  0.4500  0.4427  0.5000  0.6250
```

Let us have a look at these overall error rates on boxplots:

```
ggplot(data = tibble(error_rate = c(error_rates_train_2,error_rates_test_2),
                     sample = rep(c("Train", "Test"), each = 100))) +
  geom_boxplot(aes(x =error_rate, y = sample)) +
  labs(x = "Overall Error rate", y = "Sample")
```

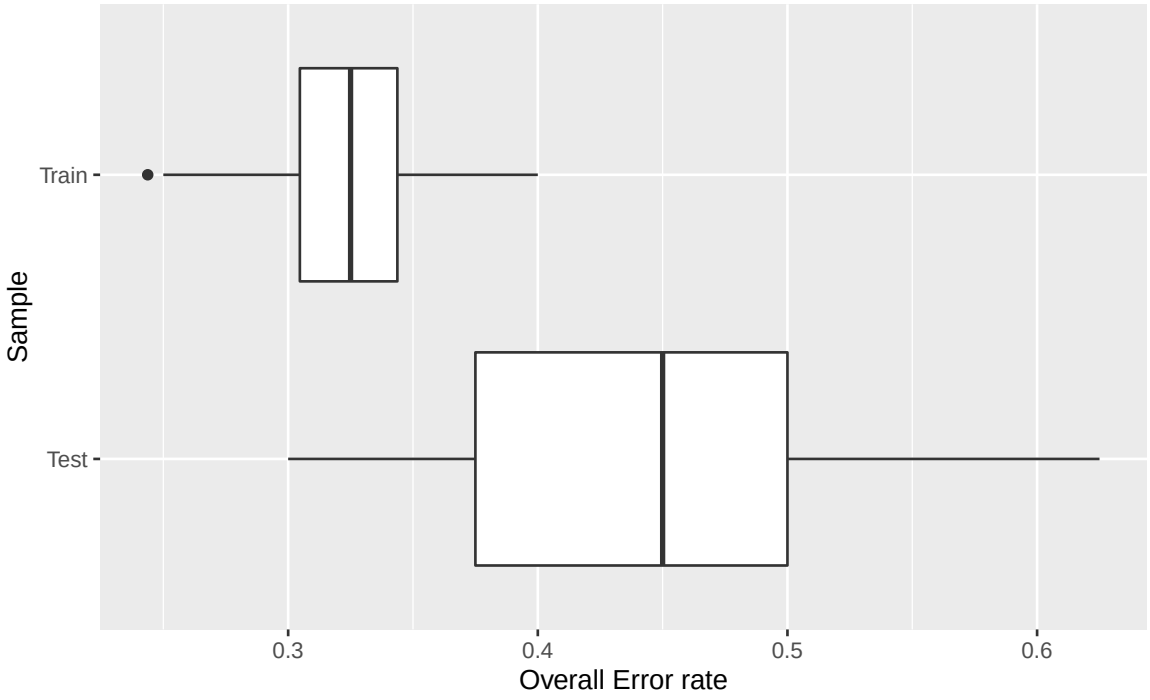

Figure 3.3: Model performance on 100 random draws of the data.

We can see that it was not bad luck in this case. The model tends to make better predictions on the training set and its predictive capacities tend to be worsen on the testing set.

To have an idea of how good the model may perform on unseen data, we can proceed as we just did, but we can also use cross validation.

3.1.5 K-fold Cross Validation

We will use K-fold Cross validation to assess the predictive capacities of the model. We will use a value of $k = 5$.

K-fold Cross Validation

- Consider a dataset with n training observations. This set of observations can be divided into k subsets of roughly the same size. Each subset is called a *fold*.
- In each k-fold, the fitting procedure is performed on the $k - 1$ folds and evaluated on the kth fold.
- The error metric is computed at each iteration
- Once each of the k-fold has served as an evaluation set, we can compute the average of the error metrics (the cross-validation error).

Choice of K

- The choice of the number of folds is not straightforward.
  - Relatively small values of k lead to larger training samples, which may result in more bias in the estimation of the true surface.
  - Relatively high values of k lead to less bias in the estimation of the true surface, but they also lead to a higher variance of the estimated test error.
- In the end, it depends on the size and structure of the dataset.
- In practice, we often pick k = 3, k = 5 or k = 10.

Here, let us use $k = 5$.

```
k <- 5
```

We can define a small function that will split a vector into k parts. We will feed this function with the index of rows.

```
#' Splits a vector into folds
#' @param x vector of observations to be splitted
#' @param k number of desired folds
split_into_folds <- function(x,k)
  split(x, cut(seq_along(x), k, labels = FALSE))
```

Here is an example of how this function works:

```
split_into_folds(seq(1, 20), k=3)
```

```
## $`1`
## [1] 1 2 3 4 5 6 7
##
## $`2`
## [1]  8  9 10 11 12 13
##
## $`3`
## [1] 14 15 16 17 18 19 20
```

Here, we will make a stratified sampling to make sure we end with approximatively the same distribution of the target variable as that of the dataset.

```
(classes <- unique(df_first_example_2$y))
```

```
## [1] 0 1
```

```
folds <- vector(mode = "list", length = length(classes))
for(i in 1:length(classes)){
  class <- classes[i]
  # row number of obs with target equal to !!class
  # shuffle the obs
  ind_class <-
    sample(which(df_first_example_2$y == class), replace=FALSE)
  # split into k folds
  folds[[i]] <- split_into_folds(ind_class, k=k)

}

folds <-
  map(1:k, ~map(folds, .)) %>%
  map(unlist) %>%
  map(sample)
```

Here are the row indices in each fold:

```
folds
```

```
## [[1]]
##  [1]   3 168 139  35  69   1 146 175 160  73  79  60  41 195  18
     2  33 108 133
## [20] 138 169 156 131 118  68 123  10  32  50  19 122  47 121  80
    162  81  89 101
## [39] 171 141
##
## [[2]]
##  [1]  29  20  30  52 116  88 184 183  54  37  77 158 127 113  16
    180 173 164 157
## [20]  96  56  57 196 117  92 155  82  40  75  26 153 163  48 192
    149  43  14 161
## [39] 190 142
##
## [[3]]
##  [1]  93 150 167 194 103 147 191 106  66  84 107  53 176 159 186
    177  34 181  87
## [20]  62  13  17 144 135 197   9 200  67  31 193  51  85   4 145
     90  70   8   7
## [39] 136  25
##
## [[4]]
##  [1] 102 109  22  46 182  63 125 115 104 165  64  98  28 198   5
     86 132 114  39
## [20] 119 129  71  45  42  97  94  78 124  61  99  21 111 179 112
     23 170 187  83
## [39] 126 178
##
## [[5]]
##  [1]  15 137 174 185 120 148  49  55  11  74  44 100 110  27 154
     95 189  65  76
## [20] 199 140 143  24 128 152 130 166 151  72  38  58   6 134  59
     12 188  91  36
## [39] 105 172
```

We will consider each fold iteratively and make the estimation of the modle on the remaing folds. At each iteration, we will store the overall error rates.

```
error_rates <- NULL
```

Here is the loop:

```
for(current_fold in 1:k){
  # All the obs but those of the kth fold
  ind_obs_current <- unlist(folds[-current_fold])
  df_train_current <- df_first_example_2 %>% slice(ind_obs_current)
  df_test_current <- df_first_example_2 %>% slice(-ind_obs_current)

  # Fitting the model
  reg_train <- glm(y ~ ., data = df_train_current, family = "binomial")
  # Predictions on the train set
  pred_train_prob <- predict(reg_train)
  pred_train <- ifelse(pred_train_prob>.5, 1, 0)
  # On the test set
  pred_test_prob <- predict(reg_train, newdata = df_test_current)
  pred_test <- ifelse(pred_test_prob>.5, 1, 0)

  # Confusion matrices
  confusion_matrix_train <- confusion_table(df_train_current$y, pred_train)
  confusion_matrix_test <- confusion_table(df_test_current$y, pred_test)

  error_rate_train <- confusion_matrix_train$metrics$`Overall error rate`
  error_rate_test <- confusion_matrix_test$metrics$`Overall error rate`
  error_rates[[current_fold]] <-
    list(error_rate_train = error_rate_train,
       error_rate_test = error_rate_test)
}
```

The overall error rate we obtained at each iteration is:

```
map_dbl(error_rates, "error_rate_train")
```

```
## [1] 0.33750 0.31250 0.30625 0.34375 0.31250
```

We can calculate the mean of this error:

```
map_dbl(error_rates, "error_rate_train") %>% mean()
```

```
## [1] 0.3225
```

And on the test set:

```
map_dbl(error_rates, "error_rate_test")
```

```
## [1] 0.400 0.450 0.525 0.475 0.450
```

```
map_dbl(error_rates, "error_rate_test") %>% mean()
```

```
## [1] 0.46
```

This mean error gives us a pretty good idea of the average error we will make with unseen data when the model is made into production.

3.1.6 Leave-one-out Cross Validation

Now, let us turn to leave-one-out cross validation.

Leave one out cross validation

- Leave one out cross validation is a k-fold cross validation where $k = n$, *i.e.*, the number of folds equals the number of training examples.
- The idea is to leave one observation out and then perform the fitting procedure on all remaining data. - Then, iterate on each data point.

- Each fitting procedure yields an estimation. It is then possible to average the results to get the error metric.
- While this procedure reduces the bias, as it uses all data points, it may be time consuming.
- In addition, the estimations may be influenced by outliers.

This procedure can be implemented as follows:

```
# Leave one out CV
error_rates_loocv <- NULL

for(i in 1:nrow(df_first_example_2)){
  df_train_current <- df_first_example_2 %>% slice(-i)
  # Single obs:
```

```
  df_test_current <- df_first_example_2 %>% slice(i)

  # Fitting the model
  reg_train <- glm(y ~ ., data = df_train_current, family = "binomial")
  # Predictions on the train set
  pred_train_prob <- predict(reg_train)
  pred_train <- ifelse(pred_train_prob>.5, 1, 0)
  # On the test set
  pred_test_prob <- predict(reg_train, newdata = df_test_current)
  pred_test <- ifelse(pred_test_prob>.5, 1, 0)

  avg_error_train <- mean(!pred_train == df_train_current$y)
  avg_error_test <- mean(!pred_test == df_test_current$y)

  error_rates_loocv[[i]] <-
    list(avg_error_train = avg_error_train,
         avg_error_test = avg_error_test)

}
```

We can have a look at the performances of the model on the training set:

```
map_dbl(error_rates_loocv, "avg_error_train")
```

```
##  [1] 0.3517588 0.3467337 0.3567839 0.3567839 0.3517588 0.3467337
    0.3467337
##  [8] 0.3467337 0.3517588 0.3517588 0.3366834 0.3417085 0.3517588
    0.3467337
## [15] 0.3467337 0.3366834 0.3417085 0.3517588 0.3417085 0.3417085
    0.3266332
## [22] 0.3567839 0.3517588 0.3618090 0.3517588 0.3567839 0.3567839
    0.3567839
## [29] 0.3567839 0.3517588 0.3567839 0.3517588 0.3316583 0.3467337
    0.3366834
## [36] 0.3517588 0.3517588 0.3517588 0.3517588 0.3567839 0.3467337
    0.3517588
## [43] 0.3366834 0.3266332 0.3618090 0.3417085 0.3567839 0.3517588
    0.3216080
## [50] 0.3316583 0.3417085 0.3517588 0.3467337 0.3567839 0.3517588
    0.3517588
## [57] 0.3567839 0.3517588 0.3165829 0.3467337 0.3467337 0.3417085
    0.3517588
```

```
## [64] 0.3517588 0.3517588 0.3567839 0.3517588 0.3517588 0.3517588
   0.3618090
## [71] 0.3517588 0.3417085 0.3517588 0.3417085 0.3467337 0.3467337
   0.3467337
## [78] 0.3467337 0.3618090 0.3567839
##  [ reached getOption("max.print") -- omitted 120 entries ]
```

```
map_dbl(error_rates_loocv, "avg_error_train") %>% mean()
```

```
## [1] 0.3501005
```

And on the testing set:

```
map_dbl(error_rates_loocv, "avg_error_test")
```

```
##  [1] 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0
   0 0 1 1 1 0 0 0
## [39] 0 0 1 0 1 1 0 0 0 1 1 1 1 0 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0 0
   0 1 0 0 1 0 1 0
## [77] 1 0 0 0
##  [ reached getOption("max.print") -- omitted 120 entries ]
```

```
map_dbl(error_rates_loocv, "avg_error_test") %>% mean()
```

```
## [1] 0.465
```

Once again, this average give us an idea of how good (or bad) our model will perform on unseen data. If the model performs greatly on the training dataset but very poorly on the testing dataset, then we face overfitting.

3.2 Second Example: Selling Price of Cars

In this second example, we will train a machine learning model so that it can predict the selling price of cars. We will use a random forest algorithm to perform this supervised task. We will try different values for one of the hyperparameters of the model, namely the number of variables randomly sampled as candidates at each split (`mtry`).

In the next hands-on session, we will talk more about random forests. For now, let us just take it as a "black box" for which we need to select a value of one of its hyperparameters. In a nutshell, the higher the value for `mtry`, the more flexible the model.

The data we will rely on give the selling price of cars (in rupees) and provide details on the characteristics of the car. These were downloaded from Kaggle[1] and cleaned (you can download the raw dataset on Kaggle and clean it yourself if you prefer).

The columns of the dataset are the following:

Table 3.2: Variables of the dataset used for the second example.

Variable	Description
`name`	name of the car
`year`	year in which the car was bought
`selling_price`	price the owner wants to sell the car at (in thousand rupees)
`km_driven`	distance completed by the car in km
`fuel`	fuel type of the car
`seller_type`	tells if car is sold by individual or dealer
`transmission`	Gear transmission of the car (Automatic/Manual)
`owner`	number of previous owners
`mileage`	mileage of the car
`engine`	engine capacity of the car
`max_power`	max power of engine
`torque`	torque of the car
`seats`	number of seats in the car

The data can be loaded as follows:

```
cars_df <- read_csv("https://egallic.fr/Enseignement/ML/ECB/data/cars.csv")
```

There are 8128 rows and 12 in the dataset.

The target variable is `selling_price`, *i.e.*, the price of the car:

```
summary(cars_df$selling_price)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
```

[1] https://www.kaggle.com/nehalbirla/vehicle-dataset-from-cardekho

```
##    30.0   255.0   450.0   638.3   675.0 10000.0
```

```
sd(cars_df$selling_price)
```

```
## [1] 806.2534
```

Let us have a quick glance at the correlations for numerical variables:

```
cars_df %>%
  select(where(is_double)) %>%
  cor(use = "complete.obs")
```

```
##                      year selling_price   km_driven    mileage
       engine
## year           1.000000000    0.41230156 -0.42854848  0.3303059
   0.0182631
## selling_price  0.412301558    1.00000000 -0.22215848 -0.1198457
   0.4556818
## km_driven     -0.428548483   -0.22215848  1.00000000 -0.1743984
   0.2060307
## mileage        0.330305862   -0.11984571 -0.17439836  1.0000000
   -0.5661751
## engine         0.018263100    0.45568180  0.20603073 -0.5661751
   1.0000000
## max_power      0.226597796    0.74967378 -0.03815852 -0.3621186
   0.7039745
## seats         -0.007923033    0.04161669  0.22725939 -0.4462056
   0.6111034
##                 max_power        seats
## year           0.22659780 -0.007923033
## selling_price  0.74967378  0.041616694
## km_driven     -0.03815852  0.227259388
## mileage       -0.36211858 -0.446205571
## engine         0.70397453  0.611103386
## max_power      1.00000000  0.191999183
## seats          0.19199918  1.000000000
```

The proportions of each category for discrete variables can be obtained as follows:

```
cars_df %>%
  select(selling_price,where(~is.character(.x) | is.factor(.x)), -name) %>%
  pivot_longer(cols = -selling_price) %>%
  group_by(name, value) %>%
  count() %>%
  group_by(name) %>%
  mutate(prop = round(100*n / sum(n), 2)) %>%
  arrange(name, desc(prop))
```

```
## # A tibble: 14 x 4
## # Groups:   name [4]
##    name         value                     n  prop
##    <chr>        <chr>                 <int> <dbl>
##  1 fuel         Diesel                 4402 54.2
##  2 fuel         Petrol                 3631 44.7
##  3 fuel         CNG                      57  0.7
##  4 fuel         LPG                      38  0.47
##  5 owner        First Owner            5289 65.1
##  6 owner        Second Owner           2105 25.9
##  7 owner        Third Owner             555  6.83
##  8 owner        Fourth & Above Owner    174  2.14
##  9 owner        Test Drive Car            5  0.06
## 10 seller_type  Individual             6766 83.2
## 11 seller_type  Dealer                 1126 13.8
## 12 seller_type  Trustmark Dealer        236  2.9
## 13 transmission Manual                 7078 87.1
## 14 transmission Automatic              1050 12.9
```

Some categories only appear a very restricted number of time. Let us make some changes.

```
cars_df <-
  cars_df %>%
  mutate(
    fuel = forcats::fct_lump_min(fuel, min = 100, other_level = "Other"),
    owner = forcats::fct_recode(owner, "First Owner" = "Test Drive Car"))
```

There are some missing values:

```
cars_df %>%
  summarise(across(everything(), ~sum(is.na(.)))) %>%
```

```
  data.frame() %>%
  t()
```

```
##                [,1]
## name              0
## year              0
## selling_price     0
## km_driven         0
## fuel              0
## seller_type       0
## transmission      0
## owner             0
## mileage           0
## engine          221
## max_power       216
## seats           221
```

Let us use some quick and dirty method here: we will just replace the `NA` values with the avrage observed in the data. This should require more serious investigation.

Let us also make some feature engineering to augment the dataset, *i.e.*, let us add transformed explanatory variables to the dataset:

```
cars_df <-
  cars_df %>%
  # replace NA values with average in data
  # (this method is really dirty and should be discussed)
  mutate(
    engine    = replace_na(engine, mean(engine, na.rm=TRUE)),
    max_power = replace_na(max_power, mean(max_power, na.rm=TRUE)),
    seats     = replace_na(seats, mean(seats, na.rm=TRUE)),
    mileage   = ifelse(mileage < 0, yes = NA, no = mileage),
    mileage   = replace_na(mileage, mean(mileage, na.rm=TRUE))
  ) %>%
  mutate(selling_price_log = log(selling_price),
         km_driven_log     = log(km_driven),
         mileage_log       = log(mileage+1),
         engine_log        = log(engine),
         max_power_log     = log(max_power+1))
```

To fasten the computations in this session, let us only keep 500 observations.

```
set.seed(456)
cars_df <- cars_df %>% sample_n(1000)
```

We will use the `randomForest()` function from {randomForest}. The library thus needs to be loaded.

```
library(randomForest)
```

Let us split the data into two samples: 80% of the observations in a training sample, and the remaining 20% in a validation sample.

```
set.seed(456)
n_train <- round(.8*nrow(cars_df))
ind_train <- sample(1:nrow(cars_df), size = n_train, replace = FALSE)
df_train <- cars_df %>% slice(ind_train)
df_valid <- cars_df %>% slice(-ind_train)
```

Let us check how many observations we have in each samples:

```
dim(df_train)
```

```
## [1] 800  17
```

```
dim(df_valid)
```

```
## [1] 200  17
```

3.2.1 Predicting the Price with a Random Forest

Let us train a random forest on the training data with the following hyperparameters for the algorithm:

- `ntree=50`: we will grow 50 trees
- `mtry=4`: at each split in a tree, we will sample only 4 candidate variables
- `nodesize=10` : the Minimum size of terminal nodes is set to 10 (there will be no less than 10 observations in the terminal nodes)

```
set.seed(456)
mtry <- 4

model_1 <-
  randomForest::randomForest(
    selling_price ~ year + fuel + km_driven +
      seller_type + transmission + owner + mileage +
      engine + max_power + seats,
    data=df_train, importance=TRUE,
    ntree=50, mtry=mtry, nodesize = 10)
```

The estimated values can be obtained as follows:

```
predicted_train_1 <- predict(model_1, newdata = df_train)
```

Using those predictions, we can compute the Mean Squared Error (MSE):

```
resid_train_1 <- df_train$selling_price - predicted_train_1
mse_train_1 <- mean(resid_train_1^2)
mse_train_1
```

```
## [1] 16141.57
```

Let us pick another value for `mtry` a higher value can make the model more flexible and could thus reduce the MSE.

```
set.seed(456)
mtry <- 7
model_2 <-
  randomForest::randomForest(
    selling_price ~ year + fuel + km_driven +
      seller_type + transmission + owner + mileage +
      engine + max_power + seats,
    data=df_train, importance=TRUE,
    ntree=50, mtry=mtry, nodesize = 10)
```

Using this second model, let us compute the residuals and then the MSE:

```
predicted_train_2 <- predict(model_2, newdata = df_train)
resid_train_2     <- df_train$selling_price - predicted_train_2
(mse_train_2 <- mean(resid_train_2^2))
```

```
## [1] 16106.57
```

The MSE has indeed decreased. Let us compute the MSE for both models, but this time ondata which were not used to train the model.

```
predicted_test_1  <- predict(model_1, newdata = df_valid)
predicted_test_2  <- predict(model_2, newdata = df_valid)
resid_test_1      <- df_valid$selling_price - predicted_test_1
resid_test_2      <- df_valid$selling_price - predicted_test_2
mse_test_1 <- mean(resid_test_1^2)
mse_test_2 <- mean(resid_test_2^2)
```

The MSE computed with the errors made on data never seen by the model are:

```
mse_test_1 # less flexible model
```

```
## [1] 21387.4
```

```
mse_test_2 # more flexible model
```

```
## [1] 22276.27
```

Let us recall the MSE computed on the train set:

```
mse_train_1
```

```
## [1] 16141.57
```

```
mse_train_2
```

```
## [1] 16106.57
```

Compared with the less flexible model (the first one), the more flexible model yielded a lower MSE on the training set, but a higher MSE on unseen data. We may suspect some overfitting here.

Let us have a broader look at how the MSE varies as long as we make the model more or less flexible. To do so, we will simply use a loop. At each iteration, we will train the random forest on the same data, but we will make the number of variables randomly sampled as candidates at each split vary.

```
set.seed(123)
mtry <- c(2,3,4,5,6,7,8)
# pb <- txtProgressBar(min = 1, max = length(mtry), style = 3)
mse <- NULL
for(j in 1:length(mtry)){

  model <-
    randomForest::randomForest(
      selling_price ~ year + fuel + km_driven +
        seller_type + transmission + owner + mileage +
        engine + max_power + seats,
      data=df_train, importance=TRUE,
      ntree=50, mtry=mtry[j], nodesize = 10)

  predicted_train <- predict(model, newdata = df_train)
  predicted_test  <- predict(model, newdata = df_valid)
  resid_train     <- df_train$selling_price - predicted_train
  resid_test      <- df_valid$selling_price - predicted_test

  mse_train <- mean(resid_train^2)
  mse_test  <- mean(resid_test^2)

  mse <- mse %>%
    bind_rows(tibble(mse_train = mse_train, mse_test = mse_test,
                     mtry = mtry[j]))
  # setTxtProgressBar(pb, j)
}
```

```
mse
```

```
## # A tibble: 7 x 3
##   mse_train mse_test  mtry
##       <dbl>    <dbl> <dbl>
## 1    27188.   30378.     2
## 2    18281.   25414.     3
## 3    16088.   19389.     4
## 4    18224.   27619.     5
## 5    14252.   23608.     6
## 6    14278.   24330.     7
## 7    14439.   24171.     8
```

```
mse %>%
  pivot_longer(cols = -mtry,
               names_to = "sample", values_to = "MSE") %>%
  mutate(sample = factor(sample, levels = c("mse_train", "mse_test"),
                         labels = c("Train", "Test"))) %>%
  ggplot(data = ., mapping = aes(x = mtry, y = MSE)) +
  geom_line(mapping = aes(colour = sample)) +
  labs(
    x = "Number of variables randomly sampled as candidates at each split",
    y = "MSE")
```

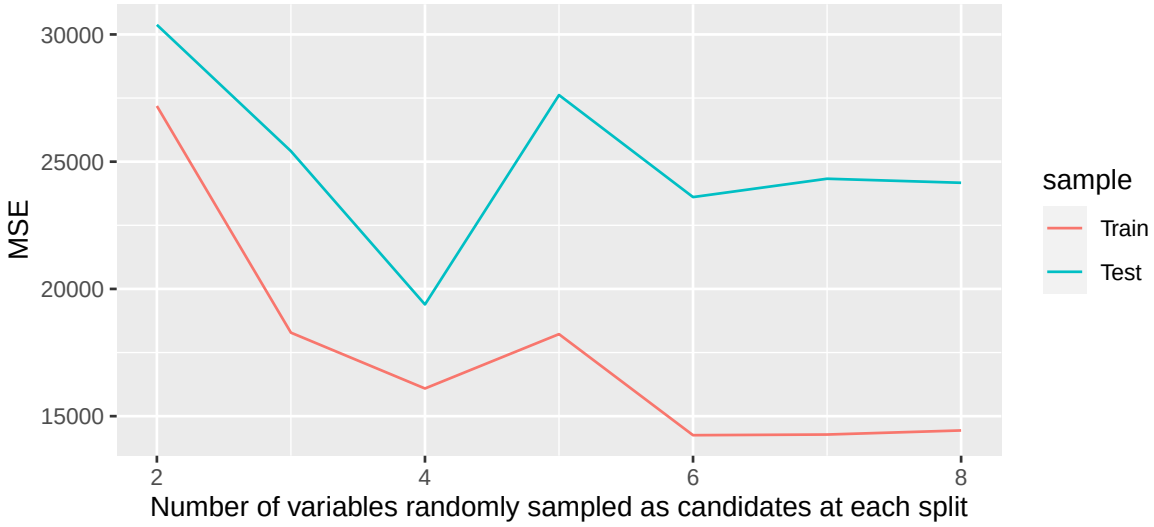

Figure 3.4: Mean Squared Error on Train/Test data..

This graph illustrates multiple things:

1. for each value of `mtry`, the MSE is relatively higher on the test sample
2. as long as the hyperparameter `mtry` increases, *i.e.*, as long as the flexibility of the model increases, the mean squared error computed on the training sample diminishes
3. this is not the case for unseen data: increasing the flexibility of the model may yield better predictions up to a certain flexibility. At some point, however, the model can learn characteristics that are specific to the test sample but which cannot be generalised afterwards: this is overfitting.

3.2.2 Cross-validation to Select the Hyperparameters

In the previous example, we have seen that the predictive capacities of the model are subject to the hyperparameters. This raises the question of the choice of value for these hyperparameters. To do so, a common practice is to perform a **grid search** and to select the values that yield the best results on unseen data. The performances of the model are assessed using cross-validation. Let us have a look at an example.

Let us only create a very simple grid, in which we will, once again, make a single hyperparameter vary. We will see in the next hands-on session how to perform a slightly more sophisticated grid search.

We will assess the model predictive capacities using k-fold cross-validation, with $k = 5$. The predictive capacities will be evaluated using the MSE.

```
nb_folds <- 5
```

As in the first example of this notebook, let us define a function that assign the elements of a vector (we will use the row numbers) to one of the k folds.

```
#' Splits a vector into folds
#' @param x vector of observations to be splitted
#' @param k number of desired folds
split_into_folds <- function(x,k) split(x, cut(seq_along(x), k, labels = FALSE))
```

Let us assign each observation to a fold:

```
set.seed(123)
fold_ind <- split_into_folds(sample(1:nrow(df_train)), nb_folds)
```

We will keep track of the MSE through the iterative process:

```
mse <- NULL
```

For each value of the hyperparameter `mtry`, we will train the random forest k times. At each of the k times, the k-th fold will be left aside to train the machine learning algorithm and will only be used to assess the performances of the model.

```
mtry <- c(2,3,4,5,6,7,8)
for(j in 1:length(mtry)){
  for(k in 1:nb_folds){
    df_train_current <-
      df_train %>%
      slice(unlist(fold_ind[-k]))

    df_test_current <-
      df_train %>%
      slice(fold_ind[[k]])

    model <-
      randomForest::randomForest(
        selling_price ~ year + fuel + km_driven +
          seller_type + transmission + owner + mileage +
          engine + max_power + seats,
```

```
        data=df_train, importance=TRUE,
        ntree=50, mtry=mtry[j], nodesize = 10)

    predicted_train <- predict(model, newdata = df_train_current)
    predicted_test  <- predict(model, newdata = df_test_current)
    resid_train     <- df_train_current$selling_price - predicted_train
    resid_test      <- df_test_current$selling_price - predicted_test

    mse_train <- mean(resid_train^2)
    mse_test  <- mean(resid_test^2)

    mse <- mse %>%
      bind_rows(tibble(mse_train = mse_train, mse_test = mse_test,
                       k = k, mtry = mtry[j]))

  }
}
```

For each value of `mtry`, we have obtained k MSE measures evaluated on the training set and k MSE measures evaluated on the fold left aside (test set). The average MSE for each value of `mtry`, both for the train and the test sets can be computed as follows:

```
mse_summary <-
  mse %>%
  group_by(mtry) %>%
  summarise(
    mse_train = mean(mse_train),
    mse_test = mean(mse_test)
  )
mse_summary
```

```
## # A tibble: 7 x 3
##    mtry mse_train mse_test
##   <dbl>     <dbl>    <dbl>
## 1     2    26458.   29510.
## 2     3    18376.   17647.
## 3     4    16029.   16240.
## 4     5    15167.   13525.
## 5     6    15704.   18081.
## 6     7    15427.   15310.
```

```
## 7     8    14845.   14689.
```

If we only look at the MSE on the train sample, we would be tempted to select a value of 5 for the number of variables to sample from at each split.

```
mse_summary %>%
  arrange(mse_train)
```

```
## # A tibble: 7 x 3
##    mtry mse_train mse_test
##   <dbl>     <dbl>    <dbl>
## 1     8    14845.   14689.
## 2     5    15167.   13525.
## 3     7    15427.   15310.
## 4     6    15704.   18081.
## 5     4    16029.   16240.
## 6     3    18376.   17647.
## 7     2    26458.   29510.
```

But looking at the MSE computed on unseen data, a value of 6 may be a better choice.

```
mse_summary %>%
  arrange(mse_test)
```

```
## # A tibble: 7 x 3
##    mtry mse_train mse_test
##   <dbl>     <dbl>    <dbl>
## 1     5    15167.   13525.
## 2     8    14845.   14689.
## 3     7    15427.   15310.
## 4     4    16029.   16240.
## 5     3    18376.   17647.
## 6     6    15704.   18081.
## 7     2    26458.   29510.
```

3.3 Third Example: Choice of Lambda in Lasso Regression

Emmanuel Flachaire showed in his slides yesterday that the Lasso regression can be written as follows:

$$\text{minimise}_{\alpha,\beta} \sum_{i=1}^{n}(y_i - \alpha - \boldsymbol{X}_i\boldsymbol{\beta})^2 + \lambda \sum_{j=1}^{p}|\beta_j|.$$

He explained that λ can be selected by cross validation. A pre-built routine in R does it for us (`cv.glmnet()` from {glmnet}). In this third example, we will use cross validation to select the value for λ. We will compare the performances of the model on unseen data by contrasting two situations:

1. one in which λ will be selected without cross-validation
2. another one in which λ will be selected using cross-validation.

First, let us load the data from {ISLR} (see James et al. (2021), chapter 6):

```
library(tidyverse)
library(ISLR)
# Removing NA values
Hitters <- na.omit(Hitters)
```

Let us create the matrix of predictors x and the target variable y (Salary).

```
x <- model.matrix(Salary ~., Hitters)[,-1]
head(x)
```

```
##                   AtBat Hits HmRun Runs RBI Walks Years CAtBat
   CHits CHmRun
## -Alan Ashby         315   81     7   24  38    39    14   3449
   835     69
## -Alvin Davis        479  130    18   66  72    76     3   1624
   457     63
## -Andre Dawson       496  141    20   65  78    37    11   5628
   1575    225
## -Andres Galarraga   321   87    10   39  42    30     2    396
   101     12
##                   CRuns CRBI CWalks LeagueN DivisionW PutOuts
   Assists Errors
## -Alan Ashby         321  414    375       1          1     632
         43     10
## -Alvin Davis        224  266    263       0          1     880
         82     14
## -Andre Dawson       828  838    354       1          0     200
         11      3
## -Andres Galarraga    48   46     33       1          0     805
         40      4
```

```
##                     NewLeagueN
## -Alan Ashby                  1
## -Alvin Davis                 0
## -Andre Dawson                1
## -Andres Galarraga            1
##  [ getOption("max.print") est atteint -- 2 lignes omises ]
```

```
y <- Hitters$Salary
head(y)
```

```
## [1] 475.0 480.0 500.0  91.5 750.0  70.0
```

We will choose λ by fitting models on a training dataset. The performances of the model will be compared on a validation set:

```
n_train <- round(.8*nrow(x))
ind_train <- sample(1:nrow(x), size = n_train, replace=FALSE)
x_train <- x[ind_train,]
x_valid <- x[-ind_train,]
y_train <- y[ind_train]
y_valid <- y[-ind_train]
```

Different values of lambda will be tested:

```
grid <- 10^seq(2, -2, length = 100)
grid
```

```
##  [1] 100.00000000  91.11627561  83.02175681  75.64633276
   68.92612104
##  [6]  62.80291442  57.22367659  52.14008288  47.50810162
   43.28761281
## [11]  39.44206059  35.93813664  32.74549163  29.83647240
   27.18588243
## [16]  24.77076356  22.57019720  20.56512308  18.73817423
   17.07352647
## [21]  15.55676144  14.17474163  12.91549665  11.76811952
   10.72267222
## [26]   9.77009957   8.90215085   8.11130831   7.39072203
   6.73415066
```

```
## [31]   6.13590727   5.59081018   5.09413801   4.64158883
   4.22924287
## [36]   3.85352859   3.51119173   3.19926714   2.91505306
   2.65608778
## [41]   2.42012826   2.20513074   2.00923300   1.83073828
   1.66810054
## [46]   1.51991108   1.38488637   1.26185688   1.14975700
   1.04761575
## [51]   0.95454846   0.86974900   0.79248290   0.72208090
   0.65793322
## [56]   0.59948425   0.54622772   0.49770236   0.45348785
   0.41320124
## [61]   0.37649358   0.34304693   0.31257158   0.28480359
   0.25950242
## [66]   0.23644894   0.21544347   0.19630407   0.17886495
   0.16297508
## [71]   0.14849683   0.13530478   0.12328467   0.11233240
   0.10235310
## [76]   0.09326033   0.08497534   0.07742637   0.07054802
   0.06428073
##  [ reached getOption("max.print") -- omitted 20 entries ]
```

A function that computes the mean squared error can be defined:

```
#' Cpmputes the Mean Squared Error
#' @param observed vector of observed data
#' @param predicted vector of predicted values, same length as `observed`
compute_mse <- function(observed, predicted){
  mean((observed - predicted)^2)
}
```

3.3.1 First Method: Without Cross-Validation

First, we will select a value of λ without using cross-validation.

We need to loop over the different values of λ. At each iteration, we will store the value of lambda we used and the computed MSE (calculated on the training set). As we will pick the value of λ based on the performances of the model on already seen data, there is a risk of overfitting.

The list `mse_lasso` will contain the MSE computed for each value of λ.

```
mse_lasso <- vector(mode = "list", length = length(grid))
```

We need to load {glmnet} to estimate the model:

```
library(glmnet)
```

And the loop:

```
for(i in 1:length(grid)){
  lambda <- grid[i]
  lasso_m_current <- glmnet(x = x_train, y = y_train,
                            alpha=1, lambda = lambda, standardize = TRUE)
  predictions_current <- predict(lasso_m_current, newx = x_train)
  mse_lasso[[i]] <- tibble(lambda = lambda, mse = compute_mse(y_train, predictions_current))
}
```

Let us bind all the tables from the list in a single table, and order the observations by descending values of the MSE:

```
mse_lasso <-
  bind_rows(mse_lasso) %>%
  arrange(mse)
mse_lasso
```

```
## # A tibble: 100 x 2
##    lambda     mse
##     <dbl>   <dbl>
##  1 0.0278 102263.
##  2 0.0254 102263.
##  3 0.0231 102263.
##  4 0.0305 102263.
##  5 0.0210 102263.
##  6 0.0192 102263.
##  7 0.0335 102263.
##  8 0.0175 102263.
##  9 0.0159 102263.
## 10 0.0368 102263.
## # ... with 90 more rows
```

The selected value of λ, with such a technique, is the one for which the MSE is the lowest, *i.e.*,

the first one in our sorted table:

```
best_lambda_1 <- mse_lasso$lambda[1]
```

We can plot the MSE as a function of λ:

```
ggplot(data = mse_lasso, mapping = aes(x = lambda, y = mse)) +
  geom_line()
```

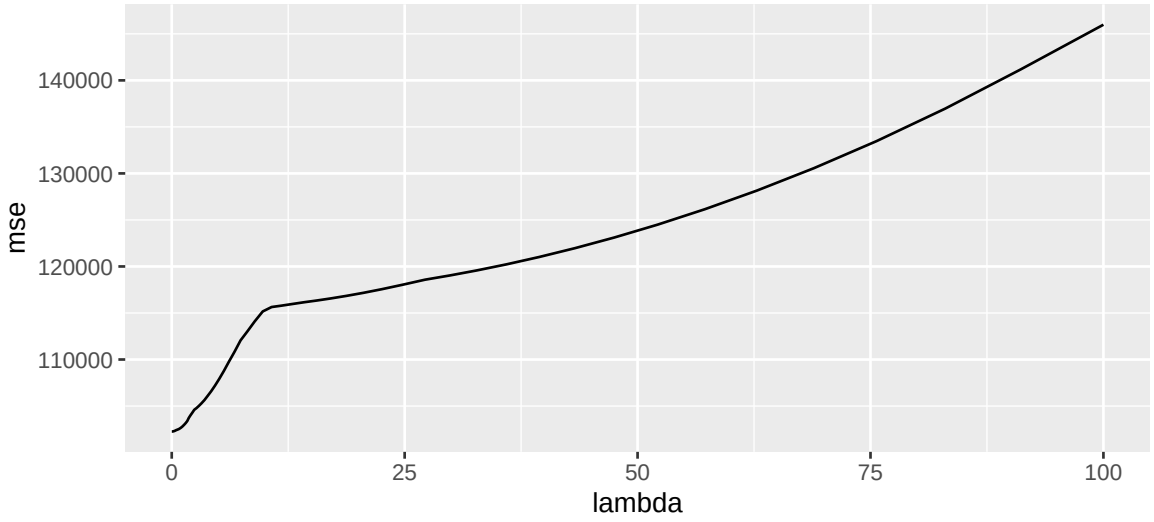

Figure 3.5: Mean Squared Error depending on the value of lambda.

3.3.2 Second Method: With Cross-Validation

To avoid selecting the value of λ by assessing the capabilities of the model to reproduce already seen data, we will use k-fold cross validation, with $k = 10$.

```
nb_folds <- 10
```

Let us assign a fold to each data from the training set:

```
n <- nrow(x_train)
folds <- sample(rep(1:nb_folds, length=n))
```

As in the first situation, we will use a loop to select the value of λ. First, we need to loop on the different folds, and for each fold, we need to train the model on the different values for λ (using

a second loop). At each time, we need to store the value of λ, the index of the k-th fold used to compute the MSE, and the computed MSE.

```
mse <- vector(mode = "list", length = nb_folds)
```

```
for(k in 1:nb_folds){
  ind_current <- folds != k

  # Train set
  x_train_current <- x_train[ind_current,]
  y_train_current <- y_train[ind_current]
  # Test set
  x_test_current <- x_train[-ind_current,]
  y_test_current <- y_train[-ind_current]

  # Looping over the values of lambda
  mse_fold <- vector(mode = "list", length = length(grid))
  for(i in 1:length(grid)){
    lambda <- grid[i]
    # Estimating on the training set
    lasso_m_current <- glmnet(x = x_train_current, y = y_train_current,
                              alpha=1, lambda = lambda, standardize = TRUE)
    # Predictions on the test set
    pred_test <- predict(lasso_m_current, x_test_current)
    # MSE
    mse_test_current <- compute_mse(y_test_current, pred_test)

    # Storing the results
    mse_fold[[i]] <-
      tibble(nb_folds = k, lambda = lambda, mse_test = mse_test_current)
  }
  mse[[k]] <- bind_rows(mse_fold)

}
mse <- bind_rows(mse)
```

The resulting table looks as follows:

```
mse
```

```
## # A tibble: 1,000 x 3
```

```
##    nb_folds lambda mse_test
##       <int>  <dbl>    <dbl>
##  1        1  100    146138.
##  2        1   91.1  142031.
##  3        1   83.0  137975.
##  4        1   75.6  134597.
##  5        1   68.9  131781.
##  6        1   62.8  129434.
##  7        1   57.2  127478.
##  8        1   52.1  125846.
##  9        1   47.5  124485.
## 10        1   43.3  123348.
## # ... with 990 more rows
```

Let us compute the average and standard deviation of MSE on the k folds:

```
mse_summary <-
  mse %>%
  group_by(lambda) %>%
  summarise(
    mse_sd_test = sd(mse_test),
    mse_test = mean(mse_test)
  )
```

What is the value of lambda that gives the lowest MSE? To answer this question, we just need to sort the table by descending values of the calculated MSE, and keep the first row.

```
best_lambda_cv <-
  mse_summary %>%
  arrange(mse_test) %>%
  slice(1)
```

Let us consider the values of λ within a 1-standard error interval:

```
best_lambda_cv <-
  best_lambda_cv %>%
  mutate(mse_test_one_sd = mse_test+mse_sd_test)

best_lambda_candidates <-
  mse_summary %>%
  filter(mse_test <= best_lambda_cv$mse_test_one_sd) %>%
```

```
  arrange(desc(lambda))

best_lambda_candidates
```

```
## # A tibble: 61 x 3
##    lambda mse_sd_test mse_test
##     <dbl>       <dbl>    <dbl>
##  1   2.66        1303.  106955.
##  2   2.42        1255.  106668.
##  3   2.21        1220.  106446.
##  4   2.01        1228.  106220.
##  5   1.83        1225.  105998.
##  6   1.67        1249.  105797.
##  7   1.52        1279.  105633.
##  8   1.38        1335.  105478.
##  9   1.26        1384.  105363.
## 10   1.15        1444.  105274.
## # ... with 51 more rows
```

Among the candidates, let us pick the highest:

```
best_lambda_2 <- best_lambda_candidates$lambda[1]
best_lambda_2
```

```
## [1] 2.656088
```

Just for the record, we can plot the MSE calculated as a function of λ:

```
mse_summary %>%
  ggplot(data = mse_summary, mapping = aes(x = lambda, y = mse_test)) +
  geom_line() +
  labs(x = "Lambda", y = "MSE")
```

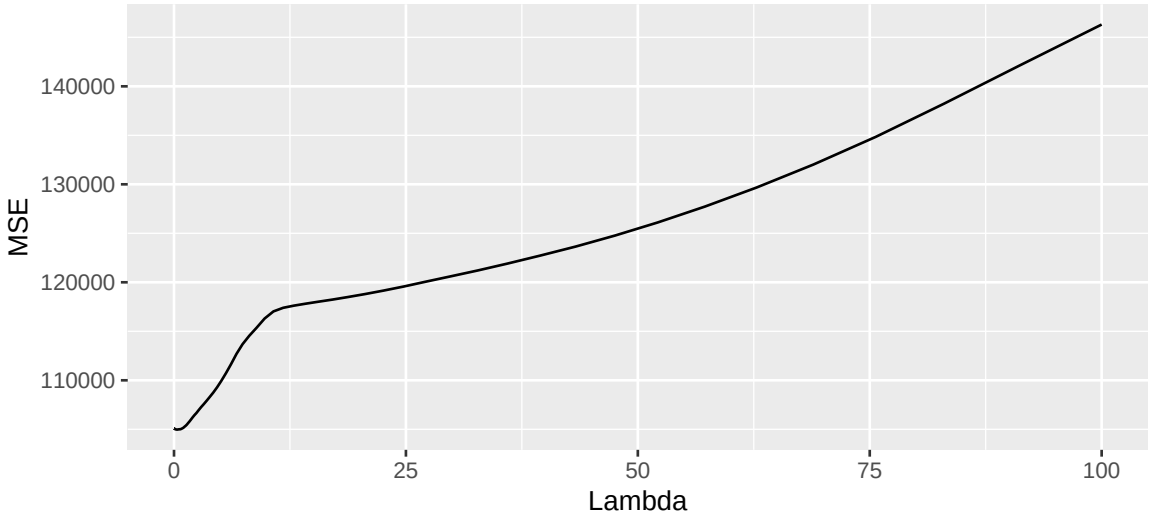

Figure 3.6: Average MSE computed on the left-aside fold.

3.3.3 Comparing the Capacities of the Models on Unseen Data

We have two different values for λ:

1. one computed without using cross-validation
2. another computer with cross-validation

```
# Model where the value of lambda was selected based on
# the MSE computed using all the observations at hand
lasso_model_1 <-
  glmnet(x = x_train, y = y_train,
         alpha=1, lambda = best_lambda_1, standardize = TRUE)
```

```
# Model where the value of lambda was selected using cross-validation
lasso_model_2 <-
  glmnet(x = x_train, y = y_train,
         alpha=1, lambda = best_lambda_2, standardize = TRUE)
```

Let us predict the values on the validation set for both models:

```
pred_valid_1 <- predict(lasso_model_1, x_valid)
mse_valid_1 <- compute_mse(y_valid, pred_valid_1)
```

Now we can compute the MSE in both cases:

```
pred_valid_2 <- predict(lasso_model_2, x_valid)
mse_valid_2 <- compute_mse(y_valid, pred_valid_2)
```

```
mse_valid_1
```

```
## [1] 58511.33
```

```
mse_valid_2
```

```
## [1] 57736.12
```

One can argue that this result may be due to chance. We can thus make some simulations to reproduce the exact same steps a great number of times.

3.3.4 Repeating the Comparison 100 times

We will repeat the following process 100 times :

1. create a training and a validation step
2. select λ without cross-validation on training the model on the training sample
3. select λ with cross-validation on training the model on the different folds
4. fit the models with the two different values of λ
5. compute the MSE on the validation set for both models

Please note that this procedure takes a few minutes on a standard computer. We could use parallel computing to fasten it.

```
nb_repeat <- 100
mse_simulation <- vector(mode = "list", length = nb_repeat)

# pb <- txtProgressBar(min = 1, max = nb_repeat, style = 3)
for(i_repeat in 1:nb_repeat){

  # Training and validation sets
  n_train <- round(.8*nrow(x))
  ind_train <- sample(1:nrow(x), size = n_train, replace=FALSE)
  x_train <- x[ind_train,]
  x_valid <- x[-ind_train,]
```

```
y_train <- y[ind_train]
y_valid <- y[-ind_train]

# 1. Lambda estimated without cross-validation
mse_lasso <- vector(mode = "list", length = length(grid))
for(i in 1:length(grid)){
  lambda <- grid[i]
  lasso_m_current <- glmnet(x = x_train, y = y_train,
                            alpha=1, lambda = lambda, standardize = TRUE)
  predictions_current <- predict(lasso_m_current, newx = x_train)
  mse_lasso[[i]] <-
    tibble(lambda = lambda,
           mse = compute_mse(y_train, predictions_current))
}

mse_lasso <-
  bind_rows(mse_lasso) %>%
  arrange(mse)

best_lambda_1 <- mse_lasso$lambda[1]

#2. Lambda estimated with cross-validation
folds <- sample(rep(1:nb_folds, length=n))

mse <- vector(mode = "list", length = nb_folds)
for(k in 1:nb_folds){
  ind_current <- folds != k

  # Train set
  x_train_current <- x_train[ind_current,]
  y_train_current <- y_train[ind_current]
  # Test set
  x_test_current <- x_train[-ind_current,]
  y_test_current <- y_train[-ind_current]

  # Looping over the values of lambda
  mse_fold <- vector(mode = "list", length = length(grid))
  for(i in 1:length(grid)){
    lambda <- grid[i]
```

```
    # Estimating on the training set
    lasso_m_current <-
      glmnet(x = x_train_current, y = y_train_current,
             alpha=1, lambda = lambda, standardize = TRUE)
    # Predictions on the test set
    pred_test <- predict(lasso_m_current, x_test_current)
    # MSE
    mse_test_current <- compute_mse(y_test_current, pred_test)

    # Storing the results
    mse_fold[[i]] <-
      tibble(nb_folds = k, lambda = lambda, mse_test = mse_test_current)
  }
  mse[[k]] <- bind_rows(mse_fold)

}
mse <- bind_rows(mse)

# Compute the average and standard deviation of MSE on the k folds
mse_summary <-
  mse %>%
  group_by(lambda) %>%
  summarise(
    mse_sd_test = sd(mse_test),
    mse_test = mean(mse_test)
  )

# What is the value of lambda that gives the lowest MSE?
best_lambda_cv <-
  mse_summary %>%
  arrange(mse_test) %>%
  slice(1) %>%
  mutate(mse_test_one_sd = mse_test+mse_sd_test)

# Lambdas within 1 standard error
best_lambda_candidates <-
  mse_summary %>%
  filter(mse_test <= best_lambda_cv$mse_test_one_sd) %>%
  arrange(desc(lambda))
```

```
  # Among the candidates, let us pick the highest
  best_lambda_2 <- best_lambda_candidates$lambda[1]

  # 3. Performance on the validation set

  # Model where the value of lambda was selected based on
  # the MSE computed using all the observations at hand
  lasso_model_1 <-
    glmnet(x = x_train, y = y_train,
           alpha=1, lambda = best_lambda_1, standardize = TRUE)

  # Model where the value of lambda was selected using cross-validation
  lasso_model_2 <-
    glmnet(x = x_train, y = y_train,
           alpha=1, lambda = best_lambda_2, standardize = TRUE)


  # Predictions:
  pred_valid_1 <- predict(lasso_model_1, x_valid)
  mse_valid_1 <- compute_mse(y_valid, pred_valid_1)

  pred_valid_2 <- predict(lasso_model_2, x_valid)
  mse_valid_2 <- compute_mse(y_valid, pred_valid_2)


  mse_simulation[[i_repeat]] <-
    tibble(i = i_repeat, mse_valid_1 = mse_valid_1, mse_valid_2)

  # setTxtProgressBar(pb, i_repeat)
}
mse_simulation <- bind_rows(mse_simulation)
```

We end up with a table with the MSE for the 100 simulations:

```
mse_simulation
```

```
## # A tibble: 100 x 3
##         i mse_valid_1 mse_valid_2
##     <int>       <dbl>       <dbl>
##  1      1     153647.     131441.
```

```
##  2     2     137854.     116347.
##  3     3     222221.     220751.
##  4     4      76973.      79704.
##  5     5      55429.      50502.
##  6     6     105757.      91559.
##  7     7     103680.     101132.
##  8     8     164501.     148567.
##  9     9     129571.     125734.
## 10    10     137936.     149992.
## # ... with 90 more rows
```

Let us look at how many replications yielded lower MSE on the validation set when λ was selected by cross-validation:

```
mse_simulation %>%
  mutate(lower_with_cv = mse_valid_2 < mse_valid_1) %>%
  group_by(lower_with_cv) %>%
  count()
```

```
## # A tibble: 2 x 2
## # Groups:   lower_with_cv [2]
##   lower_with_cv     n
##   <lgl>         <int>
## 1 FALSE            29
## 2 TRUE             71
```

And we can also compute some descriptive statistics:

```
mse_simulation %>%
  pivot_longer(cols = -i) %>%
  group_by(name) %>%
  summarise(mean = mean(value),
            sd = sd(value),
            q_1 = quantile(value, probs = .25),
            med = quantile(value, probs = .5),
            q_3 = quantile(value, probs = .75)
            )
```

```
## # A tibble: 2 x 6
##   name           mean     sd     q_1     med     q_3
##   <chr>         <dbl>  <dbl>   <dbl>   <dbl>   <dbl>
```

```
## 1 mse_valid_1 130271. 43112. 100508. 134385. 157273.
## 2 mse_valid_2 126686. 43244.  93671. 123173. 156620.
```

And lastly, we can look at the distribution of the MSE with boxplots:

```
mse_simulation %>%
  pivot_longer(cols = -i, names_to = "sample", values_to = "MSE") %>%
  mutate(sample = factor(sample, levels = c("mse_valid_1", "mse_valid_2"),
                         labels = c("Without CV", "With CV"))) %>%
  ggplot(data = ., mapping = aes(y = MSE)) +
  geom_boxplot(mapping = aes(fill = sample)) +
  scale_fill_discrete("Choice of lambda")
```

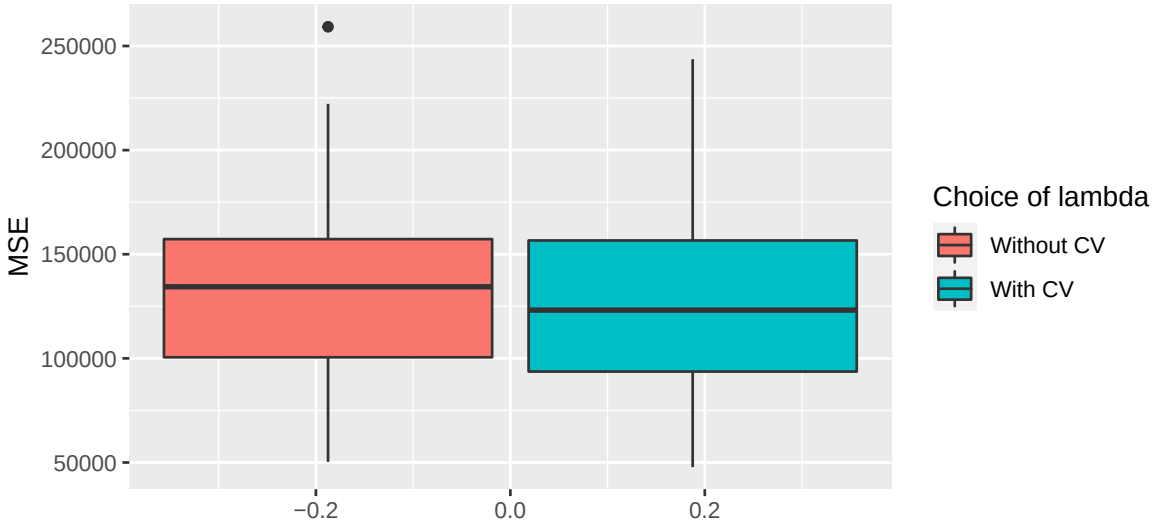

Figure 3.7: MSE on the replications depending on how lambda was selected.

Chapter 4

Trees

This chapter presents some of the tree methods used for classification or regression problems. After presenting some data that will be used to illustrate the methods, it shows how decision trees work. Then, it present two ensemble methods based on decision trees: bagging and random forests. It is built using two main references: James et al. (2021), Boehmke and Greenwell (2019).

4.1 Data Used in the Notebook

To illustrate how to use tree-based methods, we will rely on Seoul bike sharing demand data set (Sathishkumar, Jangwoo, and Yongyun 2020 ; Sathishkumar and Yongyun 2020) freely available on the UCI Machine Learning Repository.

The data give the number of bicycles rented each hour from December 1st, 2017 to November 30th, 2018. It contains 8,760 observations. Some characteristics are made available on each day:

- `date`: the date (day/month/year)
- `Rented Bike Count`: number of bicycles rented
- `Hour`: hour
- `Temperature(°C)`: temperature in Celcius degrees
- `Humidity(%)`: percentage of humidity
- `Wind speed (m/s)`: wind speed in metres per second
- `Visibility (10m)`: visibility at 10 metre
- `Dew point temperature(°C)`: dew point temperature in Celcius degrees, i.e., temperature to which air must be cooled to become saturated with water vapour
- `Solar Radiation (MJ/m2)`: solar radiation in megajoules per square metre
- `Rainfall(mm)`: rainfall in millimetres
- `Snowfall (cm)`: snowfall in centimetres
- `Seasons`: season (Spring, Summer, Automn, Winter)

- `Holiday`: holiday (Holiday, No Holiday)
- `Functioning Day`: functioning day of the bicycle rental service (Yes, No)

We will need to use many functions from the packages of the tidyverse environment.

```
library(tidyverse)
```

A copy of the dataset is available on my website. The CSV file can directly be loaded in R as follows:

```
url <- "https://egallic.fr/Enseignement/ML/ECB/data/SeoulBikeData.csv"
bike <- read_csv(url, locale = locale(encoding = "latin1"))
```

There are a few days during which the rental service is not functioning:

```
table(bike$`Functioning Day`)
```

```
##
##   No  Yes
##  295 8465
```

For simplicity, let us remove the few observations for which the service is not functioning.

```
bike <-
  bike %>%
  filter(`Functioning Day` == "Yes") %>%
  select(-`Functioning Day`)
```

The name of the variables is not convenient at all to work with. Let us rename the variables:

```
bike <-
  bike %>%
  rename(
    "date" = `Date`,
    "rented_bike_count" = `Rented Bike Count`,
    "hour" = `Hour`,
    "temperature" = `Temperature(°C)`,
    "humidity" = `Humidity(%)`,
    "wind_speed" = `Wind speed (m/s)`,
    "visibility" = `Visibility (10m)`,
```

```
    "dew_point_temperature" = `Dew point temperature(°C)`,
    "solar_radiation" = `Solar Radiation (MJ/m2)`,
    "rainfall" = `Rainfall(mm)`,
    "snowfall" = `Snowfall (cm)`,
    "seasons" = `Seasons`,
    "holiday" = `Holiday`
  )
```

There may be some seasonality in the data depending on: the hour, the day of the week, or the month of the year. While the hour is already given in the `hour` variable, the other component are not. Let us transform the `date` column in a date format. Then, the month and the day of the week can easily be extracted using functions from {lubidate}. We will provide the month name and the name of the week day in English. Depending on our settings, the functions `month()` and `wday()` from {lubridate} may give different outputs. We will thus make sure to set the time locale to English. The name of the locale is system dependent: on Unix, we can use `"en_US"`, while on Windows, we can use `"english_us"`.

```
loc_time_english <-
  ifelse(.Platform$OS.type == "unix", "en_US", "english_us")
```

```
library(lubridate)
```

The new variables can be created as follows:

```
bike <-
  bike %>%
  mutate(
    date = dmy(date),
    year = year(date),
    month = month(date, label = TRUE, locale = loc_time_english),
    month = factor(as.character(month),
                   levels = c("Jan", "Feb", "Mar", "Apr",
                              "May", "Jun", "Jul", "Aug",
                              "Sep", "Oct", "Nov", "Dec")),
    week_day = wday(date, label = TRUE, locale = loc_time_english),
    week_day = factor(as.character(week_day),
                      levels = c("Mon", "Tue", "Wed",
                                 "Thu", "Fri", "Sat", "Sun")),
    seasons = factor(seasons, levels = c("Spring", "Summer",
```

```
                                          "Autumn", "Winter"))
  )
```

Now, let us have a look at some summary statistics. First, we shall consider our target variable, *i.e.*, the hourly number of bikes rented:

```
summary(bike$rented_bike_count)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     2.0   214.0   542.0   729.2  1084.0  3556.0
```

On average, there are 729 bikes rented per hour in Seoul over the considered period (December 2017 to November 2018).

We can have an idea of the distribution by plotting a histogram:

```
ggplot(data = bike, mapping = aes(x = rented_bike_count)) +
  geom_histogram(colour = "black") +
  labs(x = "Rented bike count", y = "Count")
```

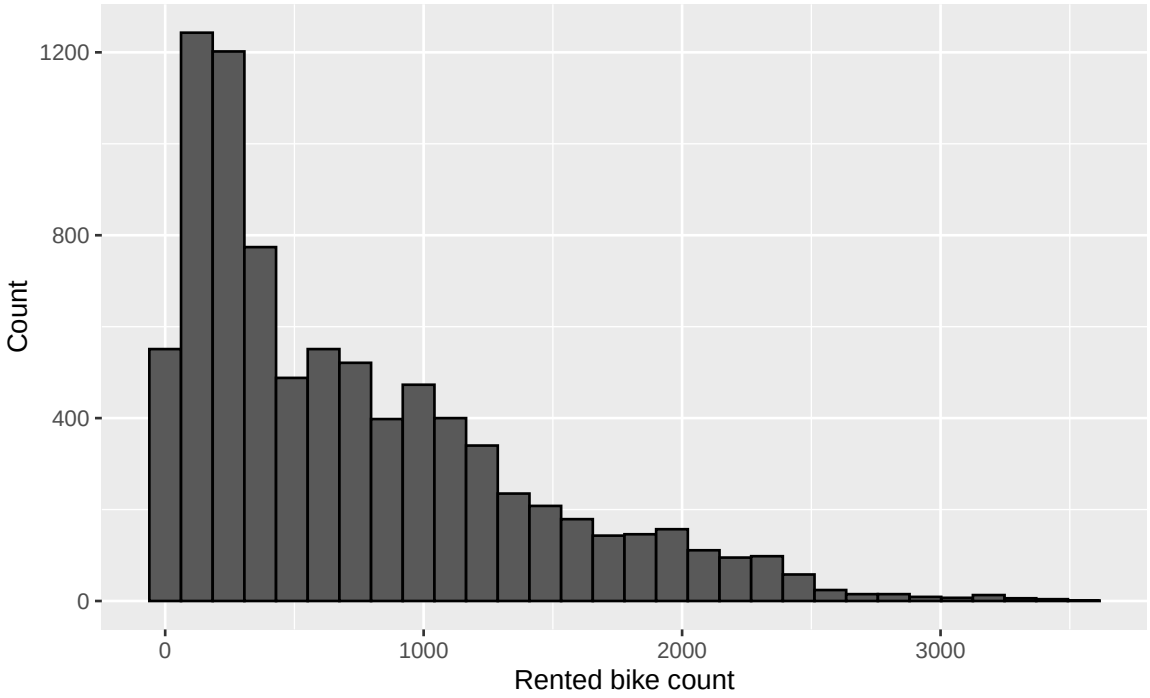

Figure 4.1: Distribution of rented bike count.

The distribution is skewed. Let us look at the distribution of the target variable depending on the month:

```
ggplot(data = bike, mapping = aes(x = rented_bike_count)) +
  geom_histogram(colour = "black") +
  labs(x = "Rented bike count", y = "Count") +
  facet_wrap(~month)
```

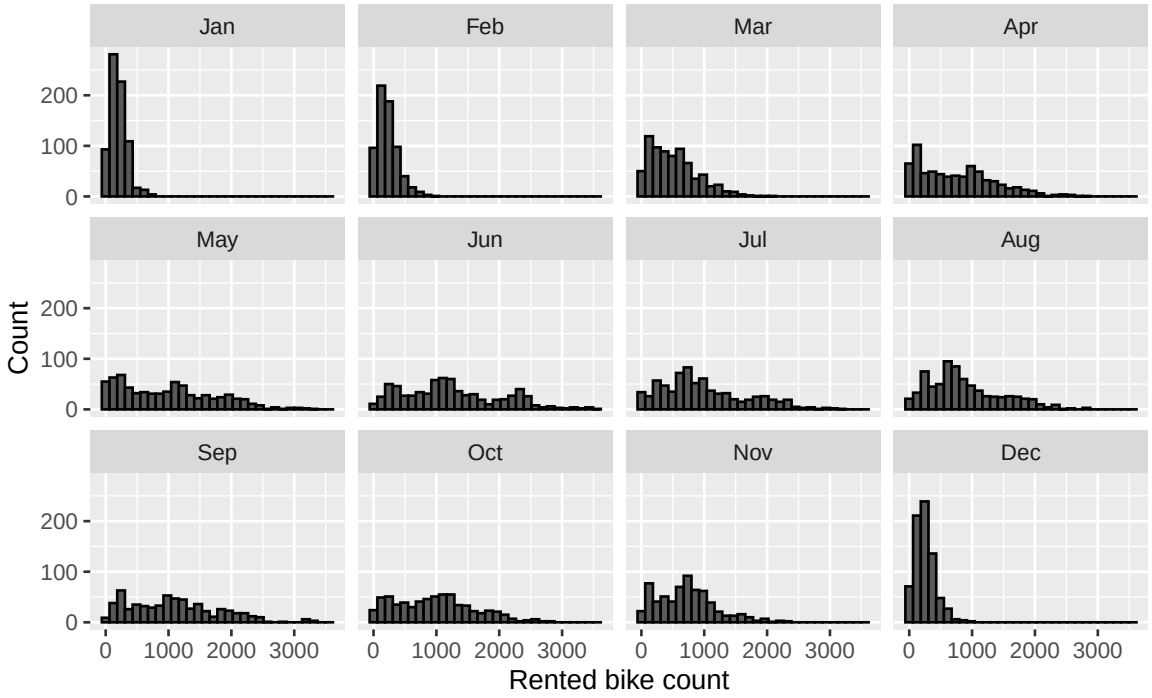

Figure 4.2: Distribution of rented bike count by month.

In cold months (December, January, February), the distribution appears to be concentrated around low values. There seems to be monthly seasonality here.

Let us look at the distribution of the number of bikes rented depending on the weekday:

```
ggplot(data = bike, mapping = aes(x = rented_bike_count)) +
  geom_histogram(colour = "black") +
  labs(x = "Rented bike count", y = "Count") +
  facet_wrap(~week_day)
```

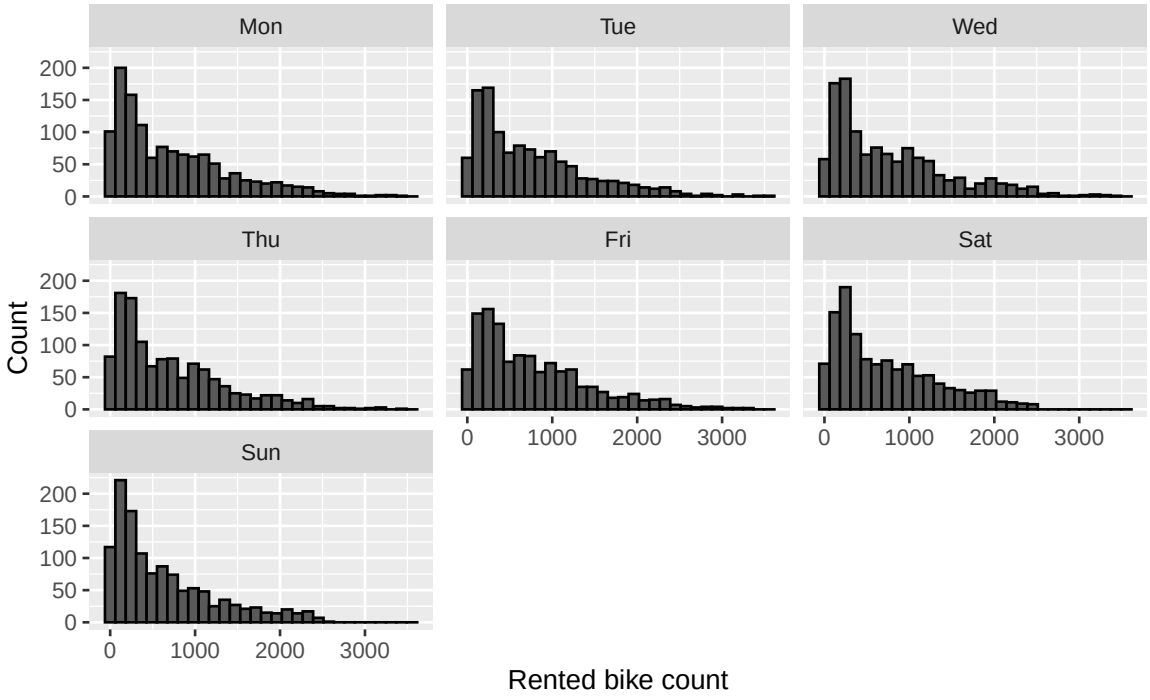

Figure 4.3: Distribution of rented bike count by weekday.

To the naked eye, there does not seem to be a specific link between the day of the week and the number of bicycles rented.

Somme summary statistics depending on the weekday can be obtained using the `tableby()` function from {arsenal}.

```
library(arsenal)
tableby_control <- tableby.control(
  numeric.stats=c("Nmiss", "meansd", "median", "q1q3")
)
tab <- tableby(week_day~rented_bike_count, data = bike,
               control = tableby_control)
summary(tab, text = NULL) %>%
  kableExtra::kable(
    caption = "Rented bike count depending on the week days.", booktabs = T,
    format = "latex", longtable = TRUE) %>%
  kableExtra::kable_classic(html_font = "Cambria") %>%
  kableExtra::kable_styling(
```

```
    bootstrap_options = c("striped", "hover", "condensed",
                          "responsive", "scale_down"),
    font_size = 5)
```

Table 4.1: Rented bike count depending on the week days.

	Mon (N=1248)	Tue (N=1152)	Wed (N=1200)	Thu (N=1200)	Fri (N=1224)	Sat (N=1217)	Sun (N=1224)	Total (N=8465)	p value
rented_bike_count									< 0.001
Mean (SD)	730.563 (664.961)	745.309 (646.550)	769.963 (674.288)	718.332 (639.037)	776.417 (654.013)	727.602 (600.706)	637.413 (605.013)	729.157 (642.351)	
Median	533.000	571.500	570.500	538.000	600.000	560.000	424.000	542.000	
Q1, Q3	186.000, 1099.000	227.750, 1083.500	225.000, 1133.250	210.000, 1068.000	262.750, 1142.000	230.000, 1095.000	170.750, 943.000	214.000, 1084.000	

The ANOVA test result reported in the previous table lead us to think that the mean is not the same across all weekdays. We notice that the number of rented bikes is relatively lower on Sundays and relatively higher on Fridays.

To have an idea of the distribution of the target variable conditional on the hour, we can use polar coordinates:

```
bike %>%
  group_by(hour) %>%
  summarise(rented_bike_count = mean(rented_bike_count)) %>%
  ggplot(data = ., aes(x = hour, y = rented_bike_count)) +
  geom_bar(stat = "identity") +
  coord_polar() +
  labs(x = NULL, y = "Rented bike count") +
  scale_x_continuous(breaks = seq(0,24, by = 2))
```

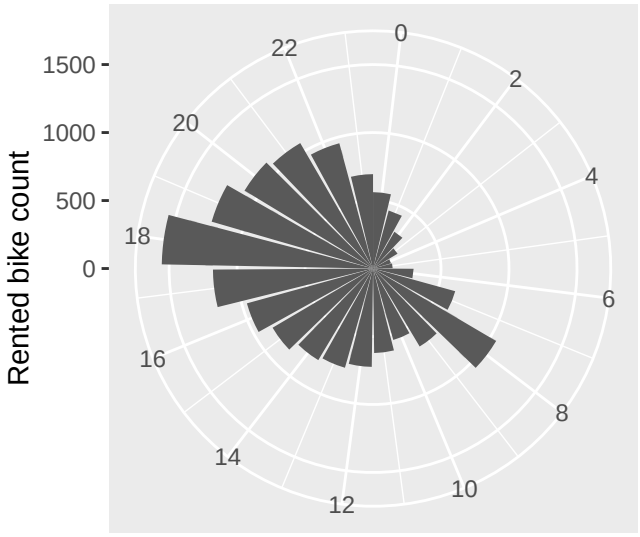

Figure 4.4: Rented bike count per hour.

As one could expect, the number of bikes rented during the night are much lower that during the day. Two peaks are observed: around 8am and around 6pm, *i.e.*, at the beginning and end of the working day.

We can also look at the distribution of the number of rented bikes depending on the seasons:

```
bike %>%
  group_by(hour, seasons) %>%
  summarise(rented_bike_count = mean(rented_bike_count)) %>%
  ggplot(data = ., aes(x = hour, y = rented_bike_count)) +
  geom_bar(stat = "identity", mapping = aes(fill = seasons)) +
  coord_polar() +
  labs(x = NULL, y = "Rented bike count") +
  scale_x_continuous(breaks = seq(0,24, by = 2)) +
  facet_wrap(~seasons) +
  scale_fill_manual("Seasons",
                    values = c("Spring" = "#004D40", "Summer" = "#FFC107",
                               "Autumn" = "#D81B60", "Winter" = "#1E88E5"))
```

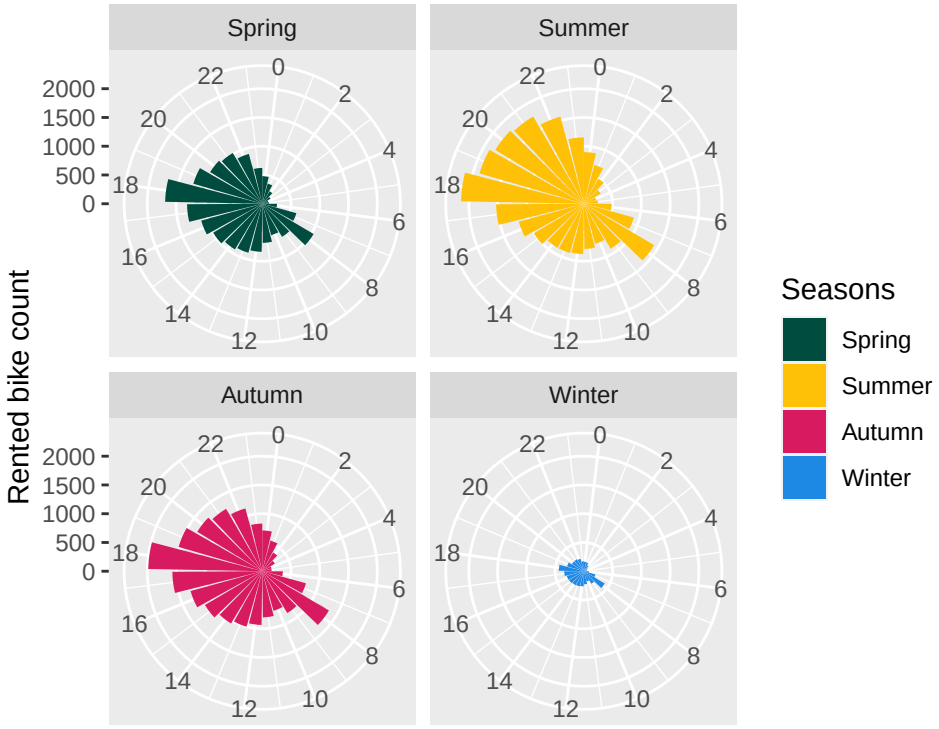

Figure 4.5: Rented bike count per hour and per season.

As previously seen, the number of rented bikes is much lower during Winter. The peaks at 8am and 6pm are observed regardless the seasons.

Now, using scatter plots, let us explore the relationship between the number of bikes rented and each weather variable. Let us first create each plot and store everyone of them in a different object.

```
library(cowplot)
p_temp <-
  ggplot(data = bike,
         mapping = aes(x = temperature, y = rented_bike_count)) +
  geom_point(alpha = .3) +
  geom_smooth() +
  labs(x = "Temperature (°C)", y = NULL,
       title = "Temperature vs humidity")

p_humidity <-
  ggplot(data = bike,
         mapping = aes(x = humidity, y = rented_bike_count)) +
```

```
  geom_point(alpha = .3) +
  geom_smooth() +
  labs(x = "Humidity (%)", y = NULL,
       title = "Rented count vs humidity")

p_wind <-
  ggplot(data = bike,
         mapping = aes(x = wind_speed, y = rented_bike_count)) +
  geom_point(alpha = .3) +
  geom_smooth() +
  labs(x = "Wind speed (m/s)", y = NULL,
       title = "Rented count vs wind speed")

p_visibility <-
  ggplot(data = bike,
         mapping = aes(x = visibility, y = rented_bike_count)) +
  geom_point(alpha = .3) +
  geom_smooth() +
  labs(x = "Visibility (10m)", y = NULL,
       title = "Rented count vs visibility")

p_dew_point <-
  ggplot(data = bike,
         mapping = aes(x = dew_point_temperature, y = rented_bike_count)) +
  geom_point(alpha = .3) +
  geom_smooth() +
  labs(x = "Dew point temperature (°C)", y = NULL,
       title = "Rented count vs dew point temperature")

p_solar_radiation <-
  ggplot(data = bike,
         mapping = aes(x = solar_radiation, y = rented_bike_count)) +
  geom_point(alpha = .3) +
  geom_smooth() +
  labs(x = "Solar Radiation (MJ/m2)", y = NULL,
       title = "Rented count vs solar radiation")


p_rainfall <-
```

```
  ggplot(data = bike,
         mapping = aes(x = rainfall, y = rented_bike_count)) +
  geom_point(alpha = .3) +
  geom_smooth() +
  labs(x = "Rainfall (mm)", y = NULL,
       title = "Rented count vs rainfall")

p_snowfall <-
  ggplot(data = bike,
         mapping = aes(x = snowfall, y = rented_bike_count)) +
  geom_point(alpha = .3) +
  geom_smooth() +
  labs(x = "Snowfall (cm)", y = NULL,
       title = "Rented count vs snowfall")
```

Then, using the `plot_grid()` function from {cowplot}, we can plot all these graphs on a single figure:

```
cowplot::plot_grid(p_temp, p_humidity, p_wind,
                   p_visibility, p_dew_point, p_rainfall,
                   p_snowfall, ncol = 2)
```

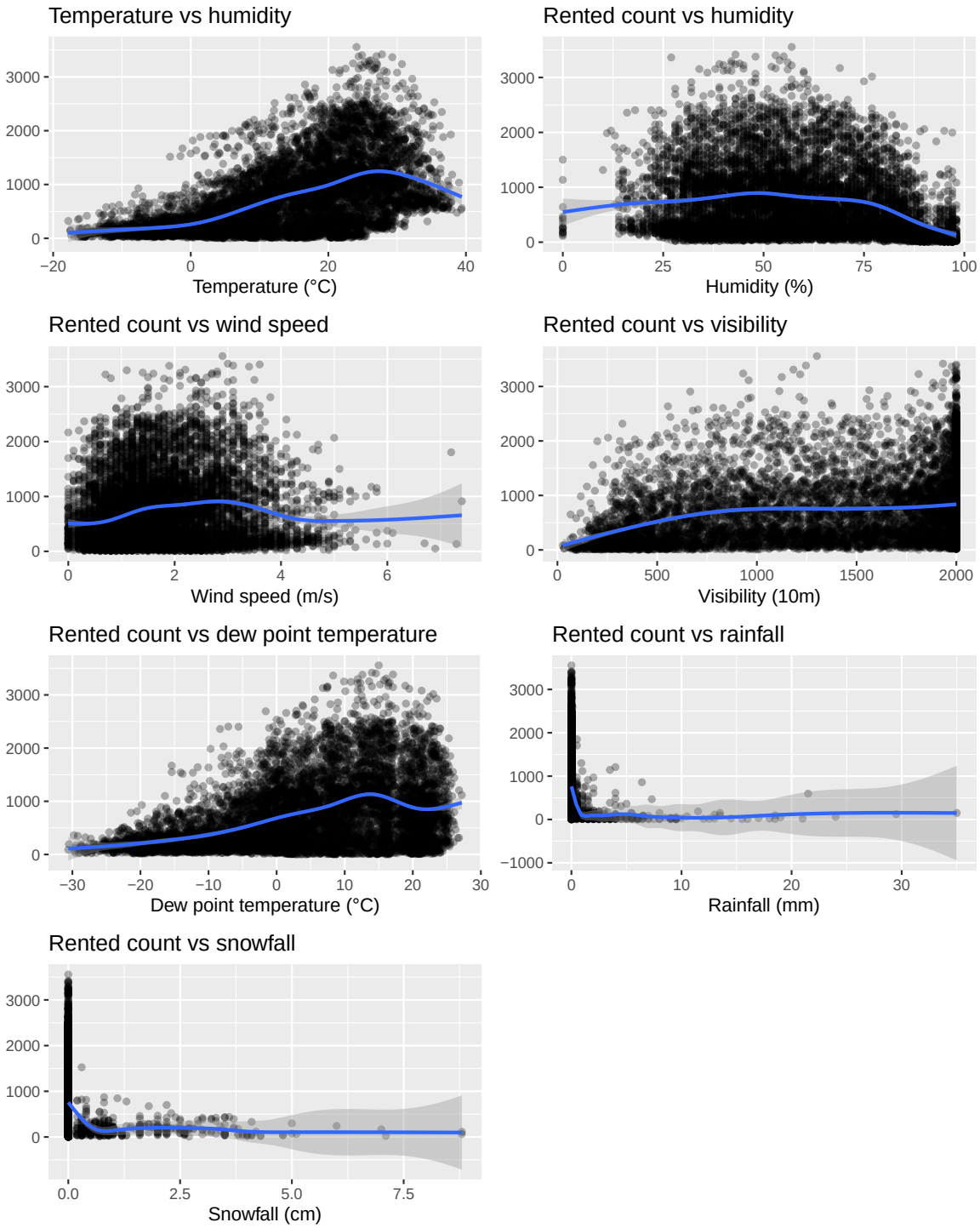

Figure 4.6: Relationship between the rented count and the numeric predictors.

The plot show nonlinear relationships between the weather and the number of rented bikes. One should note that these graphs do not allow to look at interaction effects.

Is there any correlation between the numerical variables of the dataset? Let us have a look.

```
correl_mat <-
  bike %>%
  select_if(is.numeric) %>%
  cor() %>%
  round(2)
correl_mat
```

```
##                         rented_bike_count  hour temperature
   humidity wind_speed
## rented_bike_count                    1.00  0.43        0.56
   -0.20       0.13
## hour                                 0.43  1.00        0.12
   -0.24       0.29
## temperature                          0.56  0.12        1.00
   0.17      -0.04
## humidity                            -0.20 -0.24        0.17
   1.00      -0.34
## wind_speed                           0.13  0.29       -0.04
   -0.34       1.00
## visibility                           0.21  0.10        0.03
   -0.55       0.18
## dew_point_temperature                0.40  0.00        0.91
   0.54      -0.18
##                         visibility dew_point_temperature
   solar_radiation rainfall
## rented_bike_count             0.21                  0.40
              0.27    -0.13
## hour                          0.10                  0.00
              0.14     0.01
## temperature                   0.03                  0.91
              0.35     0.05
## humidity                     -0.55                  0.54
   -0.46     0.24
## wind_speed                    0.18                 -0.18
              0.33    -0.02
## visibility                    1.00                 -0.18
              0.15    -0.17
## dew_point_temperature        -0.18                  1.00
              0.10     0.13
##                           snowfall  year
```

```
## rented_bike_count         -0.15  0.23
## hour                      -0.02  0.00
## temperature               -0.22  0.38
## humidity                   0.11  0.04
## wind_speed                 0.00  0.00
## visibility                -0.12  0.05
## dew_point_temperature     -0.15  0.33
##  [ getOption("max.print") est atteint -- 4 lignes omises ]
```

It may be more convenient to construct a correlation plot (we can do so here as the number of predictors is not too large).

```
library(corrplot)
corrplot.mixed(correl_mat, order = 'AOE')
```

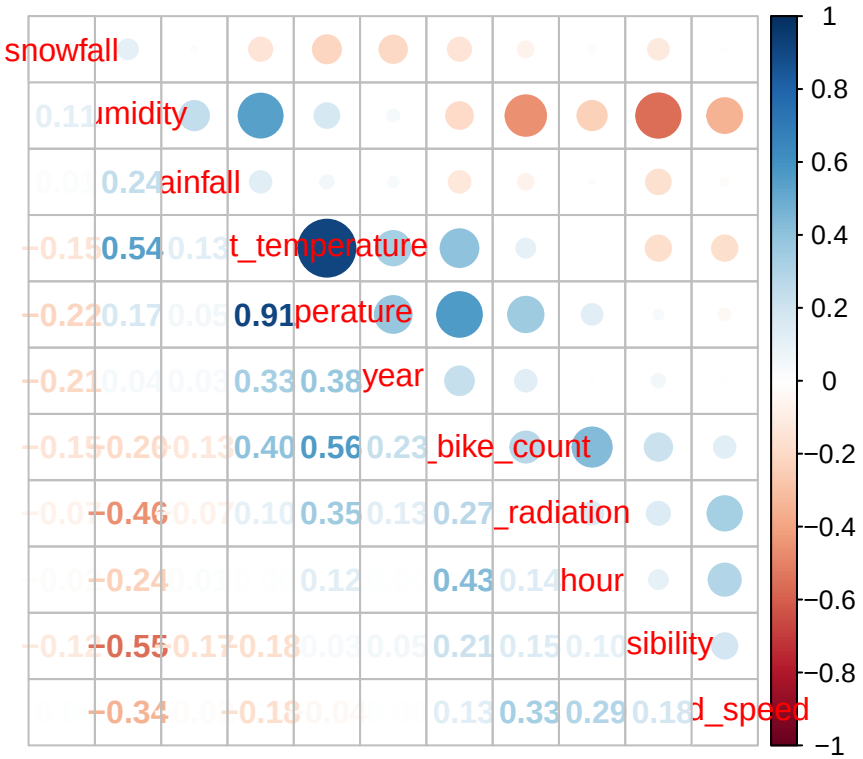

Figure 4.7: Correlation plot.

A nice summary table can also be constructed.

```
tableby_control <- tableby.control(
  numeric.stats=c("Nmiss", "meansd", "range", "median", "q1q3")
)
tab <- tableby(~., data = bike, control = tableby_control)
summary(tab, text = NULL) %>%
  kableExtra::kable(caption = "Summary statistics.", booktabs = T,
                    format = "latex", longtable = TRUE) %>%
  kableExtra::kable_classic(full_width = F, html_font = "Cambria") %>%
  kableExtra::kable_styling(
    bootstrap_options = c("striped", "hover", "condensed", "responsive"),
    font_size = 6)
```

Table 4.2: Summary statistics.

	Overall (N=8465)
date	
Median	2018-05-28
Range	2017-12-01 - 2018-11-30
rented_bike_count	
Mean (SD)	729.157 (642.351)
Range	2.000 - 3556.000
Median	542.000
Q1, Q3	214.000, 1084.000
hour	
Mean (SD)	11.507 (6.921)
Range	0.000 - 23.000
Median	12.000
Q1, Q3	6.000, 18.000
temperature	
Mean (SD)	12.771 (12.104)
Range	-17.800 - 39.400
Median	13.500
Q1, Q3	3.000, 22.700
humidity	
Mean (SD)	58.147 (20.485)
Range	0.000 - 98.000
Median	57.000
Q1, Q3	42.000, 74.000
wind_speed	
Mean (SD)	1.726 (1.034)
Range	0.000 - 7.400
Median	1.500
Q1, Q3	0.900, 2.300
visibility	
Mean (SD)	1433.873 (609.051)
Range	27.000 - 2000.000
Median	1690.000
Q1, Q3	935.000, 2000.000
dew_point_temperature	
Mean (SD)	3.945 (13.242)
Range	-30.600 - 27.200
Median	4.700

Q1, Q3	-5.100, 15.200
solar_radiation	
Mean (SD)	0.568 (0.868)
Range	0.000 - 3.520
Median	0.010
Q1, Q3	0.000, 0.930
rainfall	
Mean (SD)	0.149 (1.126)
Range	0.000 - 35.000
Median	0.000
Q1, Q3	0.000, 0.000
snowfall	
Mean (SD)	0.078 (0.444)
Range	0.000 - 8.800
Median	0.000
Q1, Q3	0.000, 0.000
seasons	
Spring	2160 (25.5%)
Summer	2208 (26.1%)
Autumn	1937 (22.9%)
Winter	2160 (25.5%)
holiday	
Holiday	408 (4.8%)
No Holiday	8057 (95.2%)
year	
Mean (SD)	2017.912 (0.283)
Range	2017.000 - 2018.000
Median	2018.000
Q1, Q3	2018.000, 2018.000
month	
Jan	744 (8.8%)
Feb	672 (7.9%)
Mar	744 (8.8%)
Apr	696 (8.2%)
May	720 (8.5%)
Jun	720 (8.5%)
Jul	744 (8.8%)
Aug	744 (8.8%)
Sep	624 (7.4%)
Oct	665 (7.9%)
Nov	648 (7.7%)
Dec	744 (8.8%)
week_day	
Mon	1248 (14.7%)
Tue	1152 (13.6%)
Wed	1200 (14.2%)
Thu	1200 (14.2%)
Fri	1224 (14.5%)
Sat	1217 (14.4%)
Sun	1224 (14.5%)

For the sake of the illustration, let us create a binary outcome variable. Let us imagine that, for example, the service is too costly if the number of bikes rented per day is below 300. In such a case, we might want to train a classifier which will predict whether or not the number of bikes rented per day will be below or above that threshold.

```
bike <-
  bike %>%
  mutate(y_binary = ifelse(rented_bike_count < 300, "Low", "High"))
```

Let us look at the number and proportions of observations in each category:

```
table(bike$y_binary)
```

```
##
## High  Low
## 5526 2939
```

```
prop.table(table(bike$y_binary))
```

```
##
##      High       Low
## 0.6528057 0.3471943
```

We can also recreate the summary table depending on the binary outcome variable:

```
tableby_control <- tableby.control(
  numeric.stats=c("Nmiss", "meansd", "range", "median", "q1q3")
)
tab <- tableby(y_binary~., data = bike, control = tableby_control)
summary(tab, text = NULL) %>%
  kableExtra::kable(
    caption = "Summary statistics depending on the binary response variable.",
    booktabs = T,
    format = "latex", longtable = TRUE) %>%
  kableExtra::kable_classic(full_width = F, html_font = "Cambria") %>%
  kableExtra::kable_styling(
    bootstrap_options = c("striped", "hover", "condensed", "responsive"),
    font_size = 6)
```

Table 4.3: Summary statistics depending on the binary response variable.

	High (N=5526)	Low (N=2939)	Total (N=8465)	p value

date				< 0.001
Median	2018-07-05	2018-02-18	2018-05-28	
Range	2017-12-01 - 2018-11-30	2017-12-01 - 2018-11-30	2017-12-01 - 2018-11-30	
rented_bike_count				< 0.001
Mean (SD)	1036.016 (597.662)	152.191 (82.792)	729.157 (642.351)	
Range	300.000 - 3556.000	2.000 - 299.000	2.000 - 3556.000	
Median	912.000	155.000	542.000	
Q1, Q3	563.000, 1358.750	79.000, 222.500	214.000, 1084.000	
hour				< 0.001
Mean (SD)	12.940 (6.533)	8.812 (6.827)	11.507 (6.921)	
Range	0.000 - 23.000	0.000 - 23.000	0.000 - 23.000	
Median	14.000	6.000	12.000	
Q1, Q3	8.000, 18.000	4.000, 14.000	6.000, 18.000	
temperature				< 0.001
Mean (SD)	17.460 (10.024)	3.954 (10.678)	12.771 (12.104)	
Range	-17.800 - 39.400	-17.500 - 31.100	-17.800 - 39.400	
Median	18.900	1.800	13.500	
Q1, Q3	9.725, 25.200	-4.200, 11.300	3.000, 22.700	
humidity				< 0.001
Mean (SD)	55.503 (18.326)	63.119 (23.227)	58.147 (20.485)	
Range	0.000 - 98.000	0.000 - 98.000	0.000 - 98.000	
Median	55.000	64.000	57.000	
Q1, Q3	42.000, 69.000	43.000, 84.000	42.000, 74.000	
wind_speed				< 0.001
Mean (SD)	1.769 (0.992)	1.645 (1.105)	1.726 (1.034)	
Range	0.000 - 7.400	0.000 - 7.300	0.000 - 7.400	
Median	1.600	1.300	1.500	
Q1, Q3	1.000, 2.400	0.800, 2.300	0.900, 2.300	
visibility				< 0.001
Mean (SD)	1497.753 (556.808)	1313.765 (680.812)	1433.873 (609.051)	
Range	66.000 - 2000.000	27.000 - 2000.000	27.000 - 2000.000	
Median	1745.000	1523.000	1690.000	
Q1, Q3	1062.000, 2000.000	606.500, 1994.000	935.000, 2000.000	
dew_point_temperature				< 0.001
Mean (SD)	7.670 (11.123)	-3.059 (14.048)	3.945 (13.242)	
Range	-29.800 - 27.200	-30.600 - 26.000	-30.600 - 27.200	
Median	8.700	-3.900	4.700	
Q1, Q3	-0.100, 16.775	-14.900, 7.600	-5.100, 15.200	
solar_radiation				< 0.001
Mean (SD)	0.780 (0.968)	0.168 (0.406)	0.568 (0.868)	
Range	0.000 - 3.520	0.000 - 2.460	0.000 - 3.520	
Median	0.280	0.000	0.010	
Q1, Q3	0.000, 1.420	0.000, 0.040	0.000, 0.930	
rainfall				< 0.001
Mean (SD)	0.020 (0.364)	0.391 (1.819)	0.149 (1.126)	
Range	0.000 - 21.500	0.000 - 35.000	0.000 - 35.000	
Median	0.000	0.000	0.000	
Q1, Q3	0.000, 0.000	0.000, 0.000	0.000, 0.000	
snowfall				< 0.001
Mean (SD)	0.016 (0.194)	0.193 (0.691)	0.078 (0.444)	
Range	0.000 - 4.100	0.000 - 8.800	0.000 - 8.800	
Median	0.000	0.000	0.000	
Q1, Q3	0.000, 0.000	0.000, 0.000	0.000, 0.000	
seasons				< 0.001
Spring	1502 (27.2%)	658 (22.4%)	2160 (25.5%)	
Summer	1889 (34.2%)	319 (10.9%)	2208 (26.1%)	
Autumn	1570 (28.4%)	367 (12.5%)	1937 (22.9%)	
Winter	565 (10.2%)	1595 (54.3%)	2160 (25.5%)	
holiday				< 0.001
Holiday	180 (3.3%)	228 (7.8%)	408 (4.8%)	

No Holiday	5346 (96.7%)	2711 (92.2%)	8057 (95.2%)	
year				< 0.001
Mean (SD)	2017.958 (0.201)	2017.826 (0.379)	2017.912 (0.283)	
Range	2017.000 - 2018.000	2017.000 - 2018.000	2017.000 - 2018.000	
Median	2018.000	2018.000	2018.000	
Q1, Q3	2018.000, 2018.000	2018.000, 2018.000	2018.000, 2018.000	
month				< 0.001
Jan	157 (2.8%)	587 (20.0%)	744 (8.8%)	
Feb	174 (3.1%)	498 (16.9%)	672 (7.9%)	
Mar	481 (8.7%)	263 (8.9%)	744 (8.8%)	
Apr	485 (8.8%)	211 (7.2%)	696 (8.2%)	
May	536 (9.7%)	184 (6.3%)	720 (8.5%)	
Jun	637 (11.5%)	83 (2.8%)	720 (8.5%)	
Jul	633 (11.5%)	111 (3.8%)	744 (8.8%)	
Aug	619 (11.2%)	125 (4.3%)	744 (8.8%)	
Sep	519 (9.4%)	105 (3.6%)	624 (7.4%)	
Oct	542 (9.8%)	123 (4.2%)	665 (7.9%)	
Nov	509 (9.2%)	139 (4.7%)	648 (7.7%)	
Dec	234 (4.2%)	510 (17.4%)	744 (8.8%)	
week_day				< 0.001
Mon	795 (14.4%)	453 (15.4%)	1248 (14.7%)	
Tue	765 (13.8%)	387 (13.2%)	1152 (13.6%)	
Wed	787 (14.2%)	413 (14.1%)	1200 (14.2%)	
Thu	778 (14.1%)	422 (14.4%)	1200 (14.2%)	
Fri	867 (15.7%)	357 (12.1%)	1224 (14.5%)	
Sat	815 (14.7%)	402 (13.7%)	1217 (14.4%)	
Sun	719 (13.0%)	505 (17.2%)	1224 (14.5%)	

Now that we are a bit more familiar with the data, let us dig into the core subject of this notebook.

4.2 Training and Test Sets

Let us create a training dataset and a test dataset. We will put 80% of the first observations in the training set and the remaining 20% in the test set. Although we will not explore the time series aspect of the data at first, let us prepare the training and test sets such that the data in the test set will be those at the end of the sample period.

First, we need to make sure that the table is sorted by ascending values of date and then by ascending values of hour:

```
bike <-
  bike %>%
  arrange(date, hour)
```

Then the two sets can be created:

```
n_train <- round(.8*nrow(bike))

# Training set
```

```
df_train <-
  bike %>%
  slice(1:n_train)

# Test set
df_test <-
  bike %>%
  slice(-(1:n_train))
```

Let us look at the dimensions:

```
dim(df_train)
```

```
## [1] 6772   17
```

```
dim(df_test)
```

```
## [1] 1693   17
```

4.3 Decision Trees

In this first part, we will try to predict the number of rented bikes with a regression tree and then we will try to predict the binary outcome variable using a classification tree. The method we will used is called Classification and Regression Trees (CART) and was introduced in L. Breiman et al. (1984).

We will rely on {rpart} to build the tree and on {rpart.plot} to create graphical illustrations (whenever it is possible to do so).

```
library(rpart)
library(rpart.plot)
```

4.3.1 Regression Trees

To predict the number of hourly rented bikes, the predictor space will be segmented into a number of simple regions. To create these segments, the algorithm creates a series of binary splits. The data are recursively split into terminal nodes (leaves). Let us illustrate this with our data.

```
part_tree <-
  rpart(rented_bike_count ~.,
        data = df_train %>% select(-y_binary, -date),
        method = "anova")
```

The rules that were learnt by the algorithm are the following

```
part_tree
```

```
## n= 6772
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##   1) root 6772 2825639000  687.5492
##     2) temperature< 12.15 3131  199783800  285.6675
##       4) temperature< 5.65 2383   64996140  228.1637 *
##       5) temperature>=5.65 748  101804000  468.8650 *
##     3) temperature>=12.15 3641 1685319000 1033.1390
##       6) hour< 15.5 2365  518621400  748.7455
##        12) solar_radiation< 0.205 1084  114108200  460.0489
##          24) hour>=1.5 801   61201670  362.2322 *
##          25) hour< 1.5 283   23550280  736.9081 *
##        13) solar_radiation>=0.205 1281  237713700  993.0445
##          26) month=Mar,Apr,Jul,Aug 723   92311480  850.5007 *
##          27) month=May,Jun,Sep 558  111677400 1177.7380 *
##       7) hour>=15.5 1276  620889500 1560.2470
##        14) rainfall>=0.05 109   19285900  259.2018 *
##        15) rainfall< 0.05 1167  399864100 1681.7670
##          30) hour>=22.5 137   11867380 1124.7230 *
##          31) hour< 22.5 1030  339831500 1755.8590
##            62) month=Mar,Apr,Aug 392   86211860 1483.6530 *
##            63) month=May,Jun,Jul,Sep 638  206727700 1923.1080
##             126) humidity>=80.5 42   10854010 1015.6900 *
##             127) humidity< 80.5 596  158853500 1987.0540 *
```

They can be visualised using a tree:

```
rpart.plot(part_tree)
```

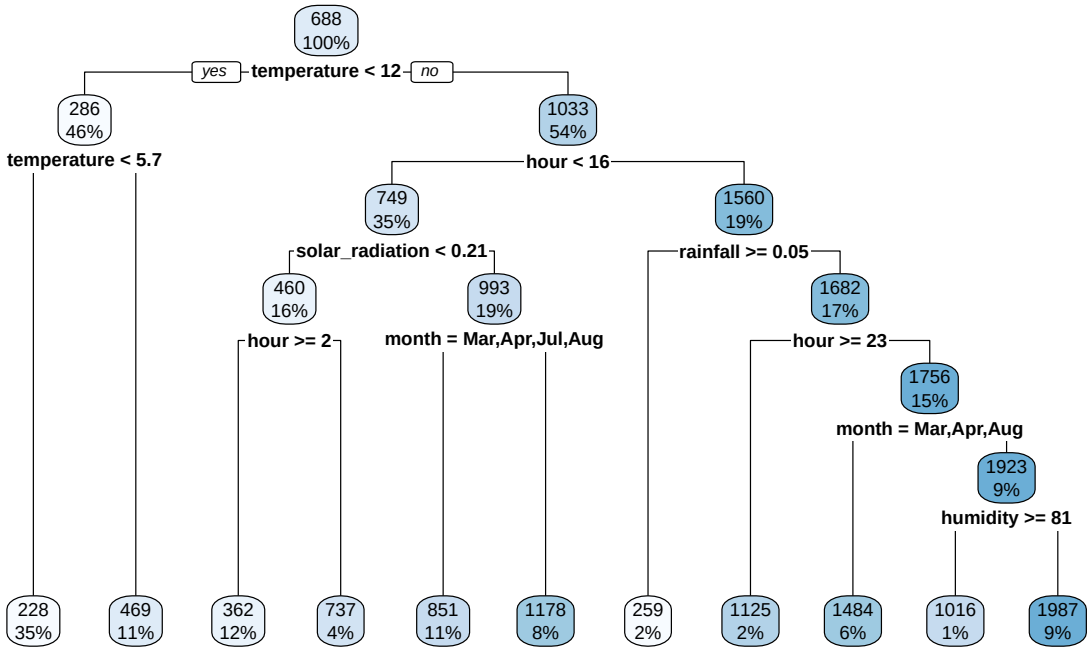

Figure 4.8: A first decision tree.

The first rule appears on top of the tree. Among all the tested separating rules, the one involving the variable `temperature` with a threshold of 12 is the one that minimises the residual sum of square (RSS). In other words, this couple of variable/cutoff value is the one that minimises deviations to the mean (variances) in each resulting leaf.

On the root node (top of the graph), the bubble shows two values:

- `688`: this corresponds to the average value of the response variable for the observations in the node (here, the average for all observations)
- `100%`: this percentage corresponds to the number of observation from the whole dataset that can be found in that node.

```
mean(df_train$rented_bike_count)
```

```
## [1] 687.5492
```

The first variable used to perform a split is `temperature`. Observations with a value for the variable `temperature` strictly lower than 12.15 (46%) will go to the left leaf. The others (the remaining 54%) will go to the right leaf.

Among the observations for which `temperature` is strictly lower than 12.15, the average of the response variable is equal to 286 (once rounded):

```
df_train %>%
  filter(temperature < 12.15) %>%
  summarise(mean = mean(rented_bike_count))
```

```
## # A tibble: 1 x 1
##    mean
##   <dbl>
## 1  286.
```

In that node, another split is performed, using the `temperature` variable once again, and the cutoff value that lead to the lowest RSS within that node is 5.65. Hence, observations for which the temperature is lower than 12.15 and lower than 5.65 (35% of all observations of the dataset) will go to the left leaf, while those for which the the temperature is lower than 12.15 and greater than 5.65 (11% of all observations of the dataset) will go to the right leaf.

```
df_train %>%
  filter(temperature < 12.15) %>%
  filter(temperature < 5.65) %>%
  count() %>%
  mutate(prop = n / nrow(df_train))
```

```
## # A tibble: 1 x 2
##       n  prop
##   <int> <dbl>
## 1  2383 0.352
```

We end up in a final node. The predicted value for such observations will be the average of the response variable in this node:

```
df_train %>%
  filter(temperature < 12.15) %>%
  filter(temperature < 5.65) %>%
  summarise(pred = mean(rented_bike_count))
```

```
## # A tibble: 1 x 1
##    pred
```

```
##    <dbl>
## 1  228.
```

And on the right node:

```
df_train %>%
  filter(temperature < 12.15) %>%
  filter(temperature > 5.65) %>%
  count() %>%
  mutate(prop = n / nrow(df_train))
```

```
## # A tibble: 1 x 2
##       n  prop
##   <int> <dbl>
## 1   748 0.110
```

```
df_train %>%
  filter(temperature < 12.15) %>%
  filter(temperature > 5.65) %>%
  summarise(pred = mean(rented_bike_count))
```

```
## # A tibble: 1 x 1
##    pred
##   <dbl>
## 1  469.
```

Let us detail the **process of selecting the variable/cutoff pair.** We can create a simple function to help compute the residual sum of squares:

```
compute_rss <- function(observed, predicted){
  sum((observed-predicted)^2)
}
```

Before making any split, the RSS is equal to:

```
rss_init <-
  compute_rss(df_train$rented_bike_count,
```

```
              mean(df_train$rented_bike_count))
rss_init
```

```
## [1] 2825639089
```

Let us consider the variable `temperature` used to partition the data, and a threshold of 12.15. Note that this variable is numerical. We will illustrate next how to procede with categorical data.

```
variable_split <- "temperature"
threshold <- 12.15
```

We split the data into two subsamples:

1. one for which `temperature` is below 12.15
2. another one for which `temperature` is greater than or equal to 12.15.

In each subset, we compute the average of the response variable: this will be the prediction made in the node:

```
tmp_data <-
  df_train %>%
  mutate(below_t = temperature < 12.15) %>%
  select(rented_bike_count, below_t) %>%
  group_by(below_t) %>%
  mutate(pred = mean(rented_bike_count))

tmp_data
```

```
## # A tibble: 6,772 x 3
## # Groups:   below_t [2]
##    rented_bike_count below_t  pred
##                <dbl> <lgl>   <dbl>
##  1               254 TRUE     286.
##  2               204 TRUE     286.
##  3               173 TRUE     286.
##  4               107 TRUE     286.
##  5                78 TRUE     286.
##  6               100 TRUE     286.
##  7               181 TRUE     286.
##  8               460 TRUE     286.
##  9               930 TRUE     286.
## 10               490 TRUE     286.
```

```
## # ... with 6,762 more rows
```

We can then compute the RSS :

```
rss_after_split <-
  compute_rss(tmp_data$rented_bike_count,
              tmp_data$pred)
rss_after_split
```

```
## [1] 1885102486
```

The improvement due to the split, in terms of percentage deviation of the RSS is given by:

```
-(rss_after_split - rss_init)/rss_init
```

```
## [1] 0.332858
```

It is the value reported in the `split` element of the object returned by `rpart()`:

```
head(part_tree$splits)
```

```
##                          count ncat   improve index       adj
## temperature               6772   -1 0.3328580 12.15 0.0000000
## month                     6772   12 0.2719021  1.00 0.0000000
## seasons                   6772    4 0.2395872  2.00 0.0000000
## dew_point_temperature     6772   -1 0.2132190  3.55 0.0000000
## hour                      6772   -1 0.1572311  6.50 0.0000000
## dew_point_temperature        0   -1 0.9162729  5.25 0.8189077
```

Now, let us wrap-up this code in a function, so that it is easy to make the threshold vary:

```
#' @param data data frame
#' @param variable_to_predict name of the variable to predict
#' @param variable_split name of the variable used to create the partitions
#' @param threshold threshold used to create the two partitions
rss_split_numeric <-
```

```
  function(data, variable_to_predict,
           variable_split, threshold){
    tmp_data <-
      data %>%
      mutate(below_t = !!sym(variable_split) < threshold) %>%
      select(!!sym(variable_to_predict), below_t) %>%
      group_by(below_t) %>%
      mutate(pred = mean(!!sym(variable_to_predict)))

    compute_rss(tmp_data[[variable_to_predict]], tmp_data[["pred"]])
  }
```

An example with the same variable (`temperature`) and threshold (12.15) as earlier:

```
rss_split_numeric(
  data = df_train, variable_to_predict = "rented_bike_count",
  variable_split = "temperature", threshold = 12.15)
```

```
## [1] 1885102486
```

All that we need to to is to make the threshold vary. For example, we can consider values ranging from the minimum of the splitting variable to its maximum, so that 1000 different threshold values are tested:

```
number_of_cuts <- 1000
thresholds <-
  seq(min(df_train$temperature), max(df_train$temperature),
      length.out = number_of_cuts)
```

Then, we can loop over these threshold values. At each iteration, we can store the RSS in a vector.

```
rss_tmp <- rep(NA, number_of_cuts)
for(i in 1:number_of_cuts){
  rss_tmp[i] <-
    rss_split_numeric(
      data = df_train, variable_to_predict = "rented_bike_count",
      variable_split = "temperature", threshold = thresholds[i])
}
```

After a few seconds, once the loop has ended, we can look at the threshold value for which the RSS was the lowest:

```
thresholds[which.min(rss_tmp)]
```

```
## [1] 12.14555
```

> We find the same value as that shown on top of the tree!

If we compute the average value of the number of rented bikes in each partition:

```
df_train %>%
  mutate(below_t = temperature < thresholds[which.min(rss_tmp)]) %>%
  select(rented_bike_count, below_t) %>%
  group_by(below_t) %>%
  summarise(pred = mean(rented_bike_count),
            n = n()) %>%
  ungroup() %>%
  mutate(prop = round(100*n/sum(n)))
```

```
## # A tibble: 2 x 4
##   below_t  pred     n  prop
##   <lgl>   <dbl> <int> <dbl>
## 1 FALSE   1033.  3641    54
## 2 TRUE     286.  3131    46
```

> We get the values shown on top of the children nodes.

Then, we need to consider **all variables as the variable used to partition the data** (not only temperature). We will not do it here, but I am sure that you understood the process of selecting the splitting rule. What you may be now wondering, is how to proceed if the variable used to partition the data is not numerical, but categorical. In such a case, we just need to consider the combination of binary splits. For example, let us consider the variable `month`. There are $l = 12$ months in the dataset, so that makes $2^{(l-1)} - 1 = 2047$ possible binary splits:

- (Jan) vs (Feb or Mar or Apr or May or Jun or Jul or Aug or Sep or Oct or Nov or Dec)
- (Jan or Feb) vs (Mar or Apr or May or Jun or Jul or Aug or Sep or Oct or Nov or Dec)

- (Jan or Mar) vs (Feb or Apr or May or Jun or Jul or Aug or Sep or Oct or Nov or Dec)
- ...
- (Jan or Feb or Mar) vs (Apr or May or Jun or Jul or Aug or Sep or Oct or Nov or Dec)
- ...
- (Jan or Feb or Mar or Apr or May or Jun or Jul or Aug or Sep or Oct or Nov) vs (Dec).

We can define a simple function that will give us all possible ways of creating unique binary partitions from a set of classes.

```
#' @param classes vector of names of classes
possible_splits <- function(classes){
  l <- length(classes)
  if(l>2){
    resul <- resul <- rep(NA, 2^(l-1) - 1)
    i <- 1
    for(k in 1:(l-1)){
      tmp <- combn(c(classes), k)
      for(j in 1:ncol(tmp)){
        if(tmp[1, j] != classes[1]) break
        current <- str_c(tmp[,j], collapse = ",")
        resul[i] <- str_c(tmp[,j], collapse = ",")
        i <- i+1
      }
    }
  }else{
    # If only two classes
    resul <- classes[1]
  }
  resul
}
```

This function just needs to be fed with a vector of classes. For example, if the classes are `A`, `B`, `C`, `D`, there will be $2^{(4-1)} - 1 = 7$ unique ways to partition the data into two areas.

```
possible_splits(c("A", "B", "C", "D"))
```

```
## [1] "A"     "A,B"   "A,C"   "A,D"   "A,B,C" "A,B,D" "A,C,D"
```

Let us illustrate how the partition is done on the tree by moving to one of the specific node: the one concerning observations for which temperature is greater than or equal to 12.15, the hour is

lower than 15.5, the solar radiation is greater than or equal to 0.205.

Recall the plot:

```
rpart.plot(part_tree)
```

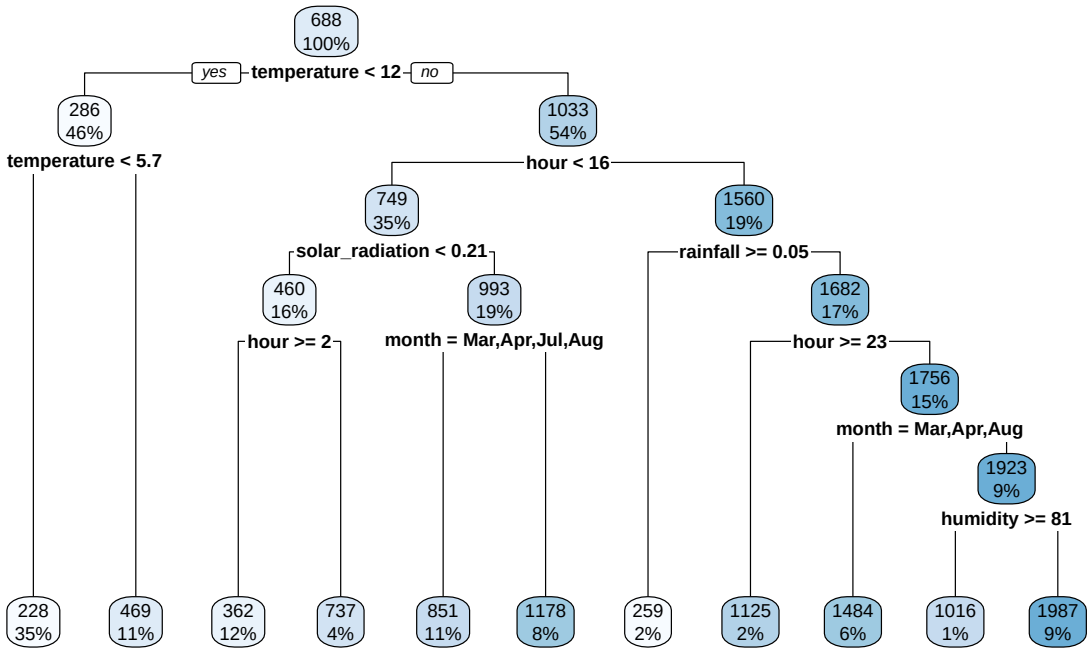

Figure 4.9: The same first decision tree.

```
df_tmp <-
  df_train %>%
  filter(temperature >= 12.15 & hour < 15.5 & solar_radiation >= 0.205)
```

The next partition, according to the graph, is done by separating observations for which the month is either March, April, July, or August from other observations. Let us try to retrieve such a result by hand.

At this node, we are left with 1281 observations:

```
nrow(df_tmp)
```

```
## [1] 1281
```

The different levels (or classes) of the `month` variable are the following:

```
classes <- as.character(unique(df_tmp$month))
classes
```

```
## [1] "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
```

We note that there are less than 12 months in the dataset at this point. The number of classes is thus lower and the number of possible splits will thus be much smaller than 2047.

We can use our little function to obtain the list of possible splits:

```
list_possible_splits <- possible_splits(classes)
head(list_possible_splits)
```

```
## [1] "Mar"     "Mar,Apr" "Mar,May" "Mar,Jun" "Mar,Jul" "Mar,Aug"
```

There are actually 7 classes, so there are 63 possible splits:

```
length(list_possible_splits)
```

```
## [1] 63
```

Let us consider the following split:

```
list_possible_splits[2]
```

```
## [1] "Mar,Apr"
```

All we need to do is to create a variable that will state whether the month is either in March or in April, *i.e.*, either in one of the elements of the following vector:

```
str_split(list_possible_splits[2], ",")[[1]]
```

```
## [1] "Mar" "Apr"
```

Then, we can group the observations by this dummy variable and compute the average of the response variable in each subset. This average will be the prediction made at this node, if we use (March or April) as the decision rule.

```
tmp_data <-
  df_tmp %>%
  mutate(var_in_categ = month %in%
           str_split(list_possible_splits[2],",")[[1]])%>%
  select(rented_bike_count, var_in_categ) %>%
  group_by(var_in_categ) %>%
  mutate(pred = mean(rented_bike_count))
```

We can check how many observations are in each subset:

```
table(tmp_data$var_in_categ)
```

```
##
## FALSE  TRUE
##  1065   216
```

And we need to compute the RSS:

```
compute_rss(tmp_data[["rented_bike_count"]], tmp_data[["pred"]])
```

```
## [1] 235207726
```

Now, let us wrap-up this in a function, so that it can be used over a loop where the decision rule is changed at each iteration:

```
#' @param data data frame
#' @param variable_to_predict name of the variable to predict
#' @param variable_split name of the variable used to create the partitions
#' @param split_rule vector of classes used to create the two partitions
rss_split_categ <-
  function(data, variable_to_predict,
           variable_split, split_rule){
    tmp_data <-
      data %>%
```

```
      mutate(var_in_categ = !!sym(variable_split) %in% split_rule) %>%
      select(!!sym(variable_to_predict), var_in_categ) %>%
      group_by(var_in_categ) %>%
      mutate(pred = mean(!!sym(variable_to_predict)))

    compute_rss(tmp_data[[variable_to_predict]], tmp_data[["pred"]])
  }
```

With the same rule as that used in the example, the function `rss_split_categ()` can be used as follows:

```
rss_split_categ(df_tmp,
                variable_to_predict = "rented_bike_count",
                variable_split = "month",
                split_rule = str_split(list_possible_splits[2], ",")[[1]])
```

```
## [1] 235207726
```

Let us loop over all the possible splits, and store the RSS at each iteration.

```
rss_tmp <- rep(NA, length(list_possible_splits))
for(i in 1:length(list_possible_splits)){
  rss_tmp[i] <-
    rss_split_categ(
      df_tmp,
      variable_to_predict = "rented_bike_count",
      variable_split = "month",
      split_rule = str_split(list_possible_splits[i], ",")[[1]])
}
```

The RSS is at its lowest for this variable with the following rule:

```
list_possible_splits[which.min(rss_tmp)]
```

```
## [1] "Mar,Apr,Jul,Aug"
```

Good news, we obtain the same result as that provided by the `rpart()` function! We can also

check that the predictions in each subset and the proportion of observations are the same as that obtained with `rpart()`.

```
df_tmp %>%
  mutate(var_in_categ = month %in%
           str_split(list_possible_splits[which.min(rss_tmp)],",")[[1]])%>%
  select(rented_bike_count, var_in_categ) %>%
  group_by(var_in_categ) %>%
  summarise(pred = mean(rented_bike_count),
            n = n()) %>%
  mutate(prop = round(100*n/nrow(df_train)))
```

```
## # A tibble: 2 x 4
##   var_in_categ  pred     n  prop
##   <lgl>        <dbl> <int> <dbl>
## 1 FALSE        1178.   558     8
## 2 TRUE          851.   723    11
```

```
rpart.plot(part_tree)
```

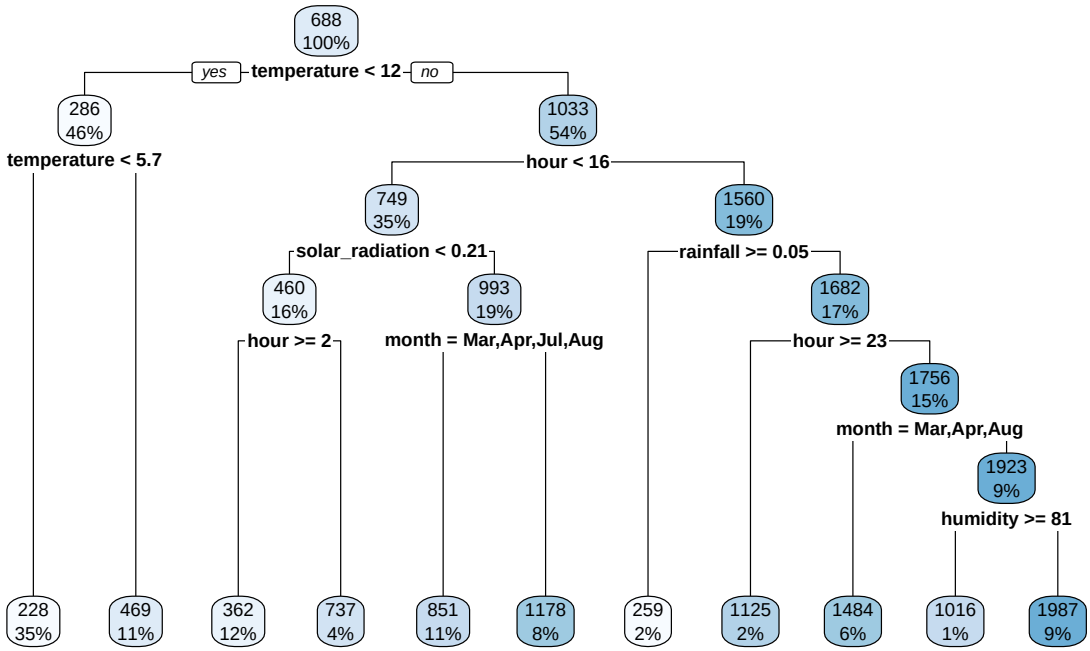

Figure 4.10: The same (again) first decision tree.

To sum up, at each node, each variable is tested as a candidate to segment the input space into two parts. Multiple splits are tested for each variables. The final choice is the variable and the splitting rule leading to the lowest RSS value.
Once a split is done, the splitting process can be repeated in each partition. This process goes on recursively until some stopping criteria. All the variables are again considered as candidates, even the one that was used in the previous split.

4.3.2 Stopping the Recursive Splitting Process

The question that can then be asked is: what criteria should be used to decide to stop the recursive partitioning process?

There are multiple ways of stopping the algorithm. These ways are controlled by the arguments of the function `rpart.control()` from {rpart}. Some of them are the following:

- `maxdepth`: we can set the maximum depth of the tree, with the root node counted as depth 0
- `minsplit`: we can decide that in order to make a split, there should be at least a minimum number of observations in a node ; if it is not the case, the split should not be attempted
- `minbucket`: we can also decide that a split should not be performed if it leads to a leaf node with too few observations (by default, this value is set in R as 1/3 of `minsplit`)
- `cp`: in the case of regression trees with anova splitting, if the split leads to an increase in the R-squared less than `cp` compared to the previous value, then the split should not be done. Setting `cp=0` will lead to overlooking this criterion as a stopping criterion.

These arguments can be directly provided to the `rpart()` function. In the following example, a split will be attempted if there is at least 1,000 observation in the node for which the split is performed, and if it leads to partitions in which at least 500 observations are left in the leaf node. If attempting the split respects these two conditions, we will overlook the fact that the R-squared should be increased at the next step, by setting `cp=-1`.

```
part_tree <-
  rpart(rented_bike_count ~.,
        data = df_train %>% select(-y_binary, -date),
        method = "anova",
        minsplit = 1000,
        minbucket = 500,
```

```
        cp=0)
rpart.plot(part_tree)
```

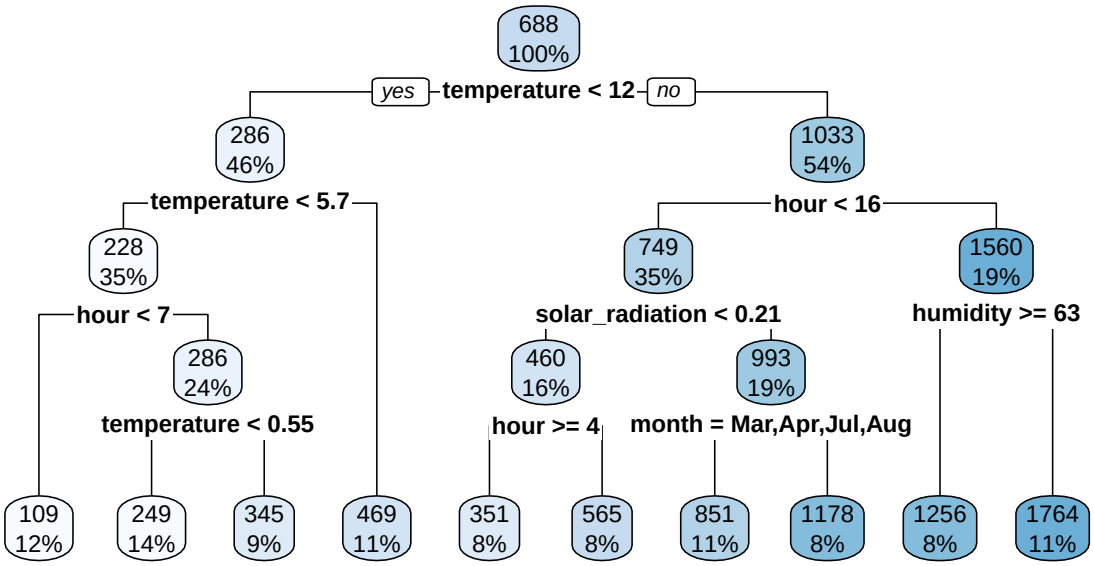

Figure 4.11: Decision tree where splits are made if there is at least 1000 obs. in the node and if the number of obs. in the resulting leaves are at least 500.

If now we allow splits to be done with the same conditions but only if they lead to an increase in the R-squared value of 0.01 from one step to the next:

```
part_tree <-
  rpart(rented_bike_count ~.,
        data = df_train %>% select(-y_binary, -date),
        method = "anova",
        minsplit = 1000,
        minbucket = 500,
        cp=.01)
rpart.plot(part_tree)
```

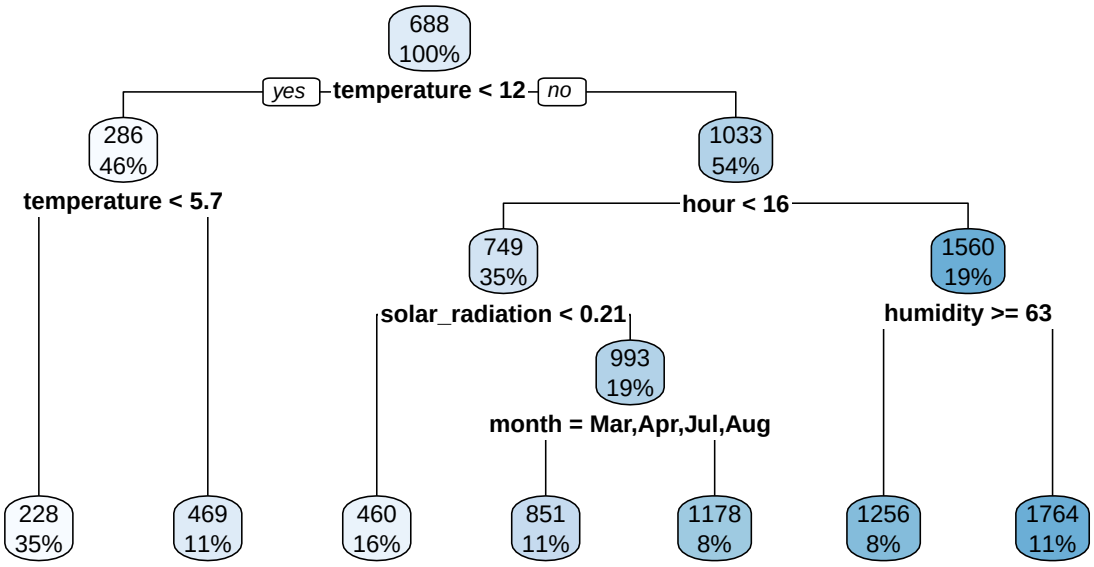

Figure 4.12: Growing the tree stopped earlier as we imposed restrictions on the improvement needed to make split.

It can be noted that the tree is a bit shallower.

The tree grown has a depth of 4. Let us constrain the tree to have a maximum depth of only 3.

```
part_tree <-
  rpart(rented_bike_count ~.,
        data = df_train %>% select(-y_binary, -date),
        method = "anova",
        maxdepth = 3,
        cp = 0)
rpart.plot(part_tree)
```

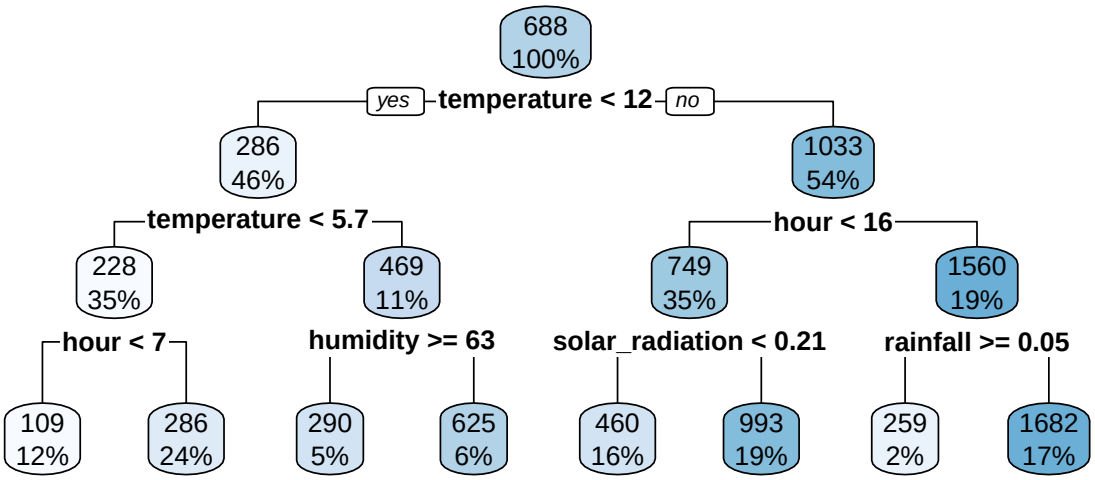

Figure 4.13: Decision tree when constraining its depth.

The algorithm stopped earlier.

Let us spent a bit more time with the two parameters controlling the maximum depth of the tree and the minimum number of observations in a node for a split to be attempted. We can easily give the intuition that the higher the depth of the tree, the higher the risk of overfitting. In a same way, the lower the minimum number of observations in a node for a split to be attempted, the higher the risk of overfitting. Let us give an illustration with synthetic data. Let us assume that we observe 100 observations drawn from the following process:

$$y_i = .1x_i^3 - x_i^2 + x_i + \varepsilon_i, \quad \varepsilon \sim \mathcal{N}(0, 4).$$

```
set.seed(123)
n <- 100
x <- runif(n=n, min=0, max=10)
eps <- rnorm(n, 0, 4)
f <- function(x) .1*x^3-1*x^2+x
y <- f(x)+eps
```

Let us plot the observed values and show with a red dotted line the expected value of data drawn from this data generating process.

```
df_sim <- tibble(x=x, y=y)
ggplot(data = df_sim, mapping = aes(x = x, y = y)) +
  geom_line(data = tibble(x=seq(0, 10, by = .1), y=f(x)),
            size = 1.1, linetype = "dashed", colour = "#AA2F2F") +
  geom_point()
```

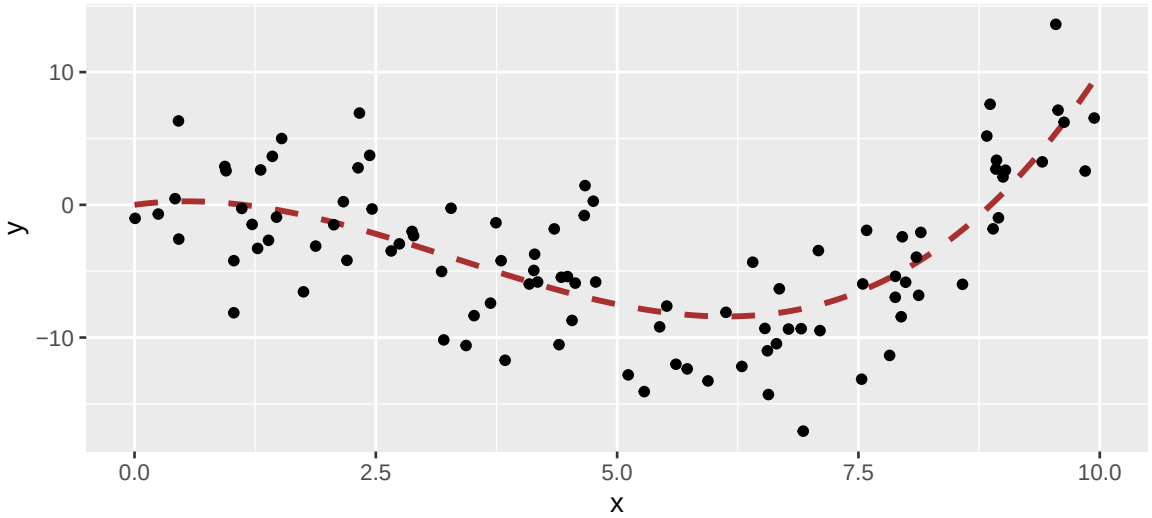

Figure 4.14: Generating Data Process and generated data.

Now, let us train a regression tree on the observed values. Let us limit the depth of the tree to 1.

```
part_tree <-
  rpart(y ~ x,
        data = df_sim,
        method = "anova",
        maxdepth = 1,
        cp = 0
  )
rpart.plot(part_tree)
```

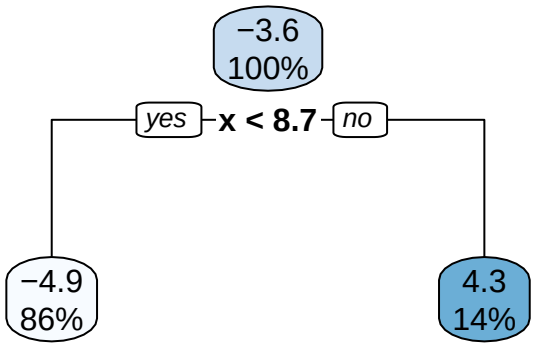

Figure 4.15: Decision tree built on the synthetic data with a maximum depth of 1.

The decision boundary is such that if x is lower than 8.7, the predicted value will be -4.9; otherwise it will be 4.3.

```
ggplot(mapping = aes(x = x, y = y)) +
  geom_line(data = tibble(x=seq(0, 10, by = .1), y=f(x)),
            size = 1.1, linetype = "dashed", colour = "#AA2F2F") +
  geom_point(data = df_sim) +
  geom_segment(data = tibble(x = c(-Inf, 8.7),
                             xend = c(8.7, Inf),
                             y = c(-4.9, 4.3),
                             yend = c(-4.9, 4.3)),
               mapping = aes(x=x, y=y, xend=xend, yend=yend),
               colour = "#005A8B", size = 1.5)
```

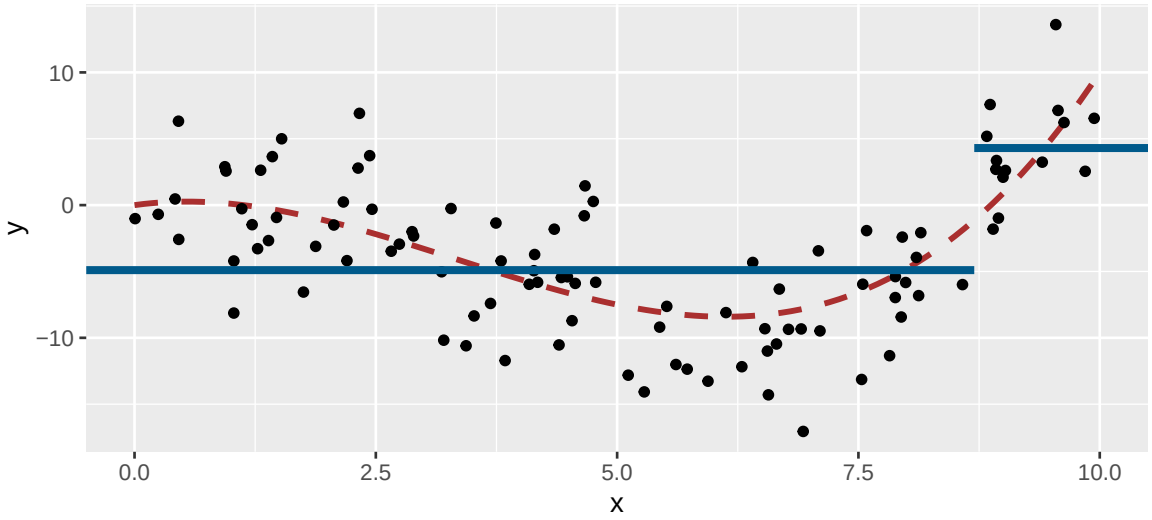

Figure 4.16: Decision boundary of the grown tree.

Now let us vary the `maxdepth` and `minsplit` parameters over a loop. We can create a function that will return the decision boundaries (to be precise, it is not exactly the exact decision boundaries that is returned, but a very close approximation).

```
get_pred_part <- function(max_depth, min_split){
  part_tree <-
    rpart(y ~ x,
          data = df_sim,
          method = "anova",
          minsplit = min_split,
          maxdepth = max_depth,
          cp = 0
    )

  tibble(
    x = seq(0, 10, by = .1),
    y = predict(part_tree, newdata = tibble(x=seq(0, 10, by = .1))),
    max_depth = max_depth,
    min_split = min_split
  )

}
```

We will loop over all the combinations that can be made up with the values for `maxdepth` and `minsplit` that we define, and apply the `get_pred_part()` at each iteration.

```
predicted_vals <-
  list(
    max_depth = c(1, 3, 5, 10),
    min_split = c(2, 10, 20)) %>%
  cross() %>%
  map_df(purrr::lift(get_pred_part))
predicted_vals
```

```
## # A tibble: 1,212 x 4
##         x     y max_depth min_split
##     <dbl> <dbl>     <dbl>     <dbl>
##  1    0   -4.92         1         2
##  2    0.1 -4.92         1         2
##  3    0.2 -4.92         1         2
##  4    0.3 -4.92         1         2
##  5    0.4 -4.92         1         2
##  6    0.5 -4.92         1         2
##  7    0.6 -4.92         1         2
##  8    0.7 -4.92         1         2
##  9    0.8 -4.92         1         2
## 10    0.9 -4.92         1         2
## # ... with 1,202 more rows
```

```
library(ggtext)
ggplot(mapping = aes(x = x, y = y)) +
  geom_line(data = tibble(x=seq(0, 10, by = .1), y=f(x)),
            size = 1.1, linetype = "dashed", colour = "#AA2F2F") +
  geom_point(data = df_sim) +
  geom_step(data = predicted_vals %>%
              mutate(max_depth = factor(
                max_depth,
                levels = sort(unique(predicted_vals$max_depth)),
                labels = str_c("Max depth: ",
                               sort(unique(predicted_vals$max_depth)))),
                min_split = factor(
                  min_split,
                  levels = sort(unique(predicted_vals$min_split),
```

```
                                  decreasing = TRUE),
                  labels = str_c("Min node size: ",
                                 sort(unique(predicted_vals$min_split),
                                      decreasing = TRUE)))
              ),
            colour = "#005A8B", size = 1.5) +
facet_grid(max_depth~min_split) +
labs(title = str_c(
  "Change in **<span style='color:#005A8B'>",
  "decision boundaries</span>** induced by parameter variations")
) +
theme(
  plot.title = element_markdown(lineheight = 1.1),
  plot.title.position = "plot",
  legend.text = element_markdown(size = 11)
)
```

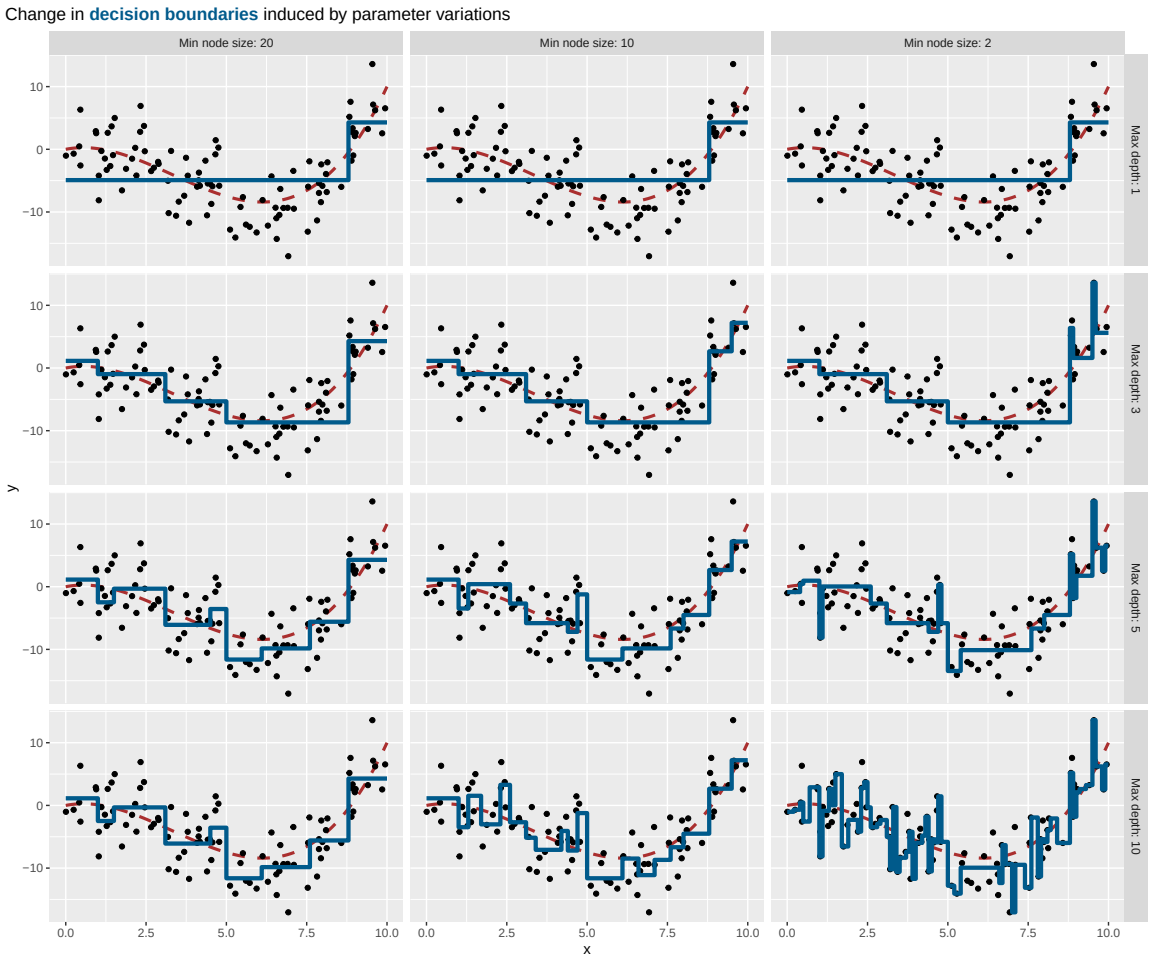

Figure 4.17: Varying the parameters affect the decision boundary and may lead to overfitting.

In the previous graph, we clearly note that:

- if the maximum depth of the tree that is grown is 1, then the minimum node size does not play any role
- as long as we increase the maximum depth, the boundary decision gets closer to the points: there is a risk of overfitting
- as long as the number of minimum observations in a node for a split to be attempted decreases, the boundary decision gets closer to the points: there is once again a risk of overfitting.

4.3.2.1 Pruning

With the previous example, we saw that smaller trees lead to lower variance. They also lead to better interpretation. On the other hand, this is done at the expense of a little bias. We might be tempted to rely on growing small trees, by stropping the algorithm as soon as the decrease in the SSR due to the current split falls below some threshold. But this is too short-sided, as a poor split could be followed by a very good split.

Another strategy, called **tree pruning**, consists in growing a large tree, and then **prune** it back to keep only a subtree. To determine the level at which pruning is carried out, we can proceed by cross-validation. But instead of considering all trees, only a subset of those are considered. The cross-validation process is used to select a penalization parameter (λ). As explained in Emmanuel Flachaire's course, we put a prize to pay for having a tree with many terminal nodes J, or regions,

$$\min \sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \overline{y}_{R_j})^2 + \lambda J.$$

With a value of $\lambda = 0$, the subtree we end up with is the large tree we grew in the first place. As long as we increase the value of λ, having many terminal nodes J become costlier.

In {rpart}, the number of folds for the cross-validation is provided through the `xval` argument of the `rpart.control()` function. It can also be given directly to the `rpart()` function.

Let us perform a 10-fold cross validation with the Seoul bike data (this is actually done by default when calling the `rpart()` function).

```
set.seed(123)
large_tree <-
  rpart(rented_bike_count ~.,
        data = df_train %>% select(-y_binary, -date),
        method = "anova",
        control = rpart.control(xval = 10, cp = -1, minbucket = 20))

# Number of nodes
nodes_large_tree <- as.numeric(rownames(large_tree$frame))
max(rpart:::tree.depth(nodes_large_tree))
```

```
## [1] 16
```

We can plot the results of the cross-validation:

```
plotcp(large_tree)
```

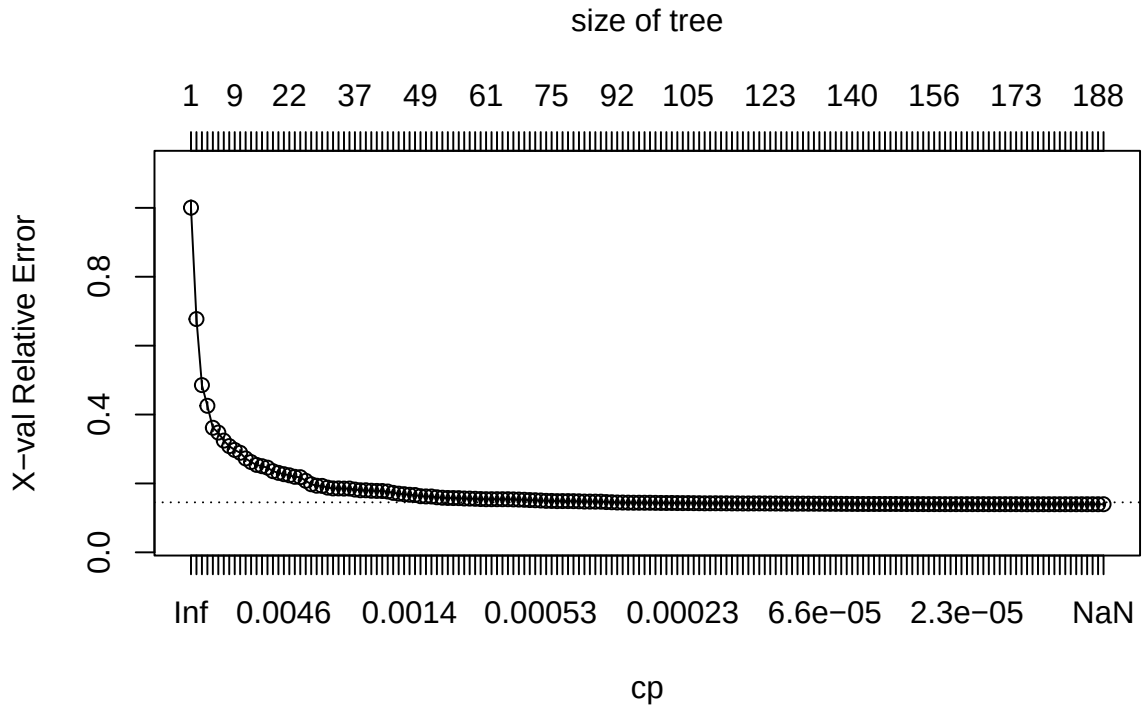

Figure 4.18: Relative error depending on the complexity parameter (10-fold cross-validation results), for the Seoul bike data.

The results of the cross-validation are stored in the `cptable` attribute of the result. The column `CP` (cost-complexity) corresponds to different values of λ, and the column `xerror` gives the resulting error.

```
head(large_tree$cptable)
```

```
##           CP nsplit rel error    xerror        xstd
## 1 0.33285801      0 1.0000000 1.0005565 0.021514547
## 2 0.19316259      1 0.6671420 0.6771662 0.015698709
## 3 0.07139607      2 0.4739794 0.4855051 0.012615119
## 4 0.05903069      3 0.4025833 0.4253927 0.010949168
## 5 0.01704578      4 0.3435527 0.3615918 0.009915513
## 6 0.01659517      5 0.3265069 0.3476690 0.009469073
```

Let us find the minimum value of the cost-parameter λ:

```
min_val <-
  as_tibble(large_tree$cptable) %>%
  arrange(xerror) %>%
  slice(1)

min_val
```

```
## # A tibble: 1 x 5
##            CP nsplit `rel error` xerror    xstd
##         <dbl>  <dbl>       <dbl>  <dbl>   <dbl>
## 1 0.00000162    185      0.0973  0.140 0.00544
```

Then, we can consider all the values of the cost-parameter within a 1-standard error deviation from that value:

```
candidates <-
  as_tibble(large_tree$cptable) %>%
  filter( (xerror > min_val$xerror-min_val$xstd) &
            (xerror < min_val$xerror+min_val$xstd)) %>%
  arrange(desc(CP))
candidates
```

```
## # A tibble: 90 x 5
##           CP nsplit `rel error` xerror    xstd
##        <dbl>  <dbl>       <dbl>  <dbl>   <dbl>
##  1 0.000314     91       0.106  0.145 0.00546
##  2 0.000311     92       0.106  0.145 0.00546
##  3 0.000295     93       0.106  0.144 0.00546
##  4 0.000295     94       0.105  0.144 0.00546
##  5 0.000294     95       0.105  0.144 0.00546
##  6 0.000286     96       0.105  0.144 0.00546
##  7 0.000285     97       0.104  0.144 0.00546
##  8 0.000269     98       0.104  0.144 0.00546
##  9 0.000262     99       0.104  0.143 0.00546
## 10 0.000259    100       0.104  0.143 0.00545
## # ... with 80 more rows
```

Among those candidates, we can pick up the values that results in the smallest subtree:

```
cp_val <-
  candidates %>%
  slice(1) %>%
  magrittr::extract2("CP")
cp_val
```

```
## [1] 0.0003139774
```

Then, the tree can be pruned accordingly with this cost-complexity value, using the `prune()` function from {rpart}:

```
pruned_tree <- prune(large_tree, cp=cp_val)
```

Let us check the depth of the resulting tree:

```
nodes_pruned_tree <- as.numeric(rownames(pruned_tree$frame))
max(rpart:::tree.depth(nodes_pruned_tree))
```

```
## [1] 12
```

And for comparison, let us grow a smaller tree:

```
small_tree <-
  rpart(rented_bike_count ~.,
        data = df_train %>% select(-y_binary, -date),
        method = "anova",
        maxdepth = 2,
        cp = 0,
        xval = 0)

# Number of nodes
nodes_small_tree <- as.numeric(rownames(small_tree$frame))
max(rpart:::tree.depth(nodes_small_tree))
```

```
## [1] 2
```

Let us compute the MSE on the test set:

```
pred_test_small <- predict(small_tree, newdata = df_test)
pred_test_large <- predict(large_tree, newdata = df_test)
pred_test_pruned <- predict(pruned_tree, newdata = df_test)

compute_mse <- function(observed, predicted){
  mean((observed-predicted)^2)
}

mse_small_tree <-
  compute_mse(observed = df_test$rented_bike_count,
              predicted = pred_test_small)
mse_large_tree <-
  compute_mse(observed = df_test$rented_bike_count,
              predicted = pred_test_large)
mse_pruned_tree <-
  compute_mse(observed = df_test$rented_bike_count,
              predicted = pred_test_pruned)

mse <- scales::number(c(mse_small_tree, mse_large_tree,
                        mse_pruned_tree))
names(mse) <- c("Small", "Large", "Pruned")
mse
```

```
##      Small      Large     Pruned
## "295 291" "164 097" "168 395"
```

Here, pruning the tree unfortunately produced a higher MSE in the test set. Let us have another look at pruning with our synthetic dataset this time.

Let us draw some new observations from the same data generating process.

```
set.seed(234)
x_new <- runif(n=n, min=0, max=10)
eps <- rnorm(n, 0, 4)
y_new <- f(x)+eps
df_sim_new <- tibble(x=x_new, y=y_new)
ggplot(mapping = aes(x = x, y = y)) +
  geom_line(data = tibble(x=seq(0, 10, by = .1), y=f(x)),
```

```
            colour = "black", size = 1.1, linetype = "dashed") +
  geom_point(data = df_sim %>% mutate(sample = "Train") %>%
               bind_rows(
                 df_sim_new %>% mutate(sample = "Test")
               ),
             mapping = aes(colour = sample)) +
  scale_colour_manual("Sample",
                      values = c("Train" = "#D55E00", "Test" = "#009E73"))
```

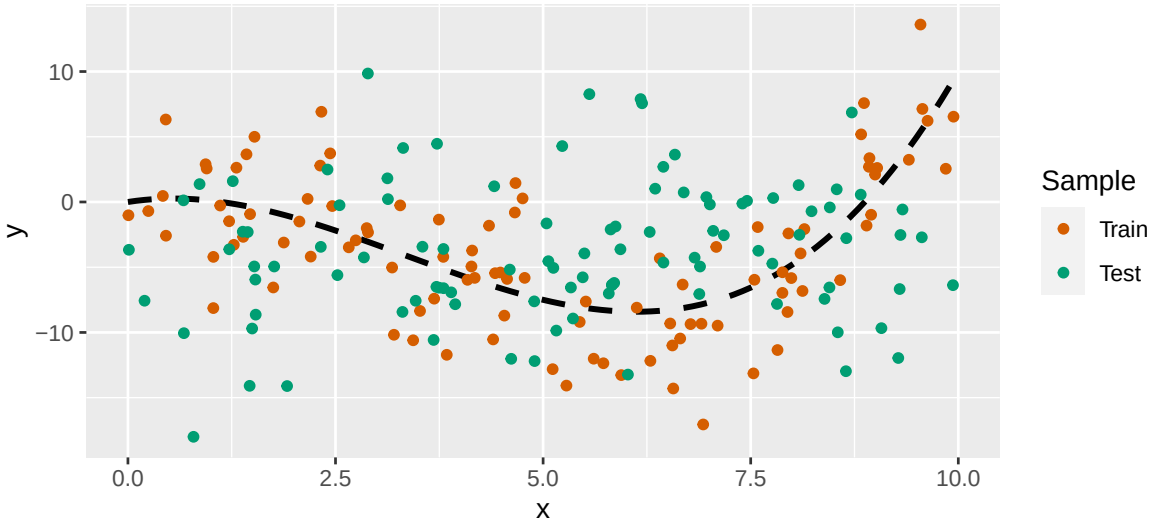

Figure 4.19: Synthetic data.

Let us build a large tree using the training sample:

```
part_tree_large <-
  rpart(y ~ x,
        data = df_sim,
        method = "anova",
        control = rpart.control(
          xval = 10,
          cp = -1,
          minbucket = 5)
  )
rpart.plot(part_tree_large)
```

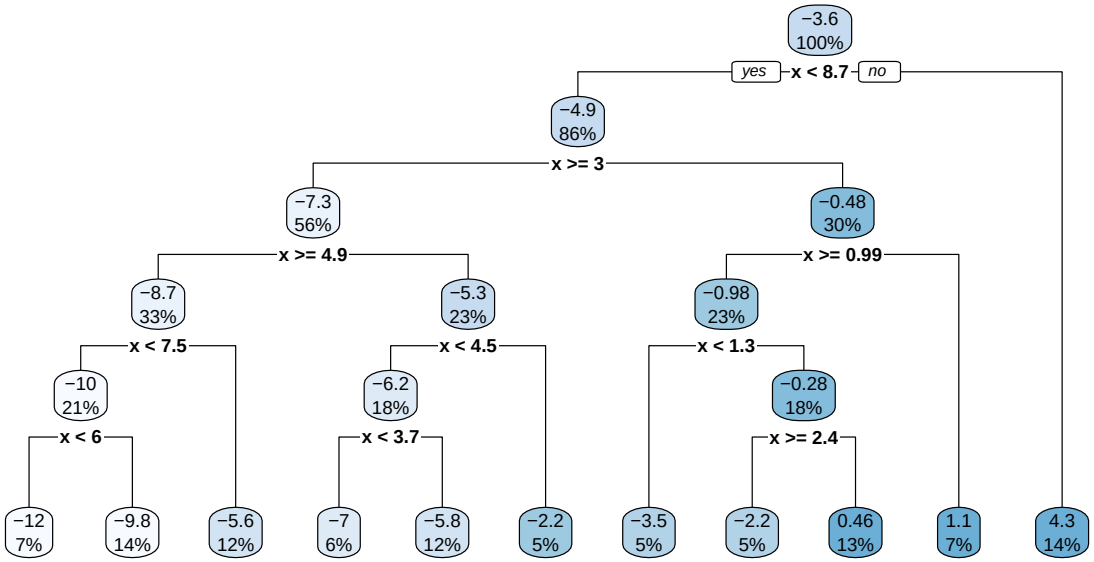

Figure 4.20: Unpruned tree.

The cost-parameter can be chosen thanks to the 10-fold cross-validation that was performed:

```
plotcp(part_tree_large)
```

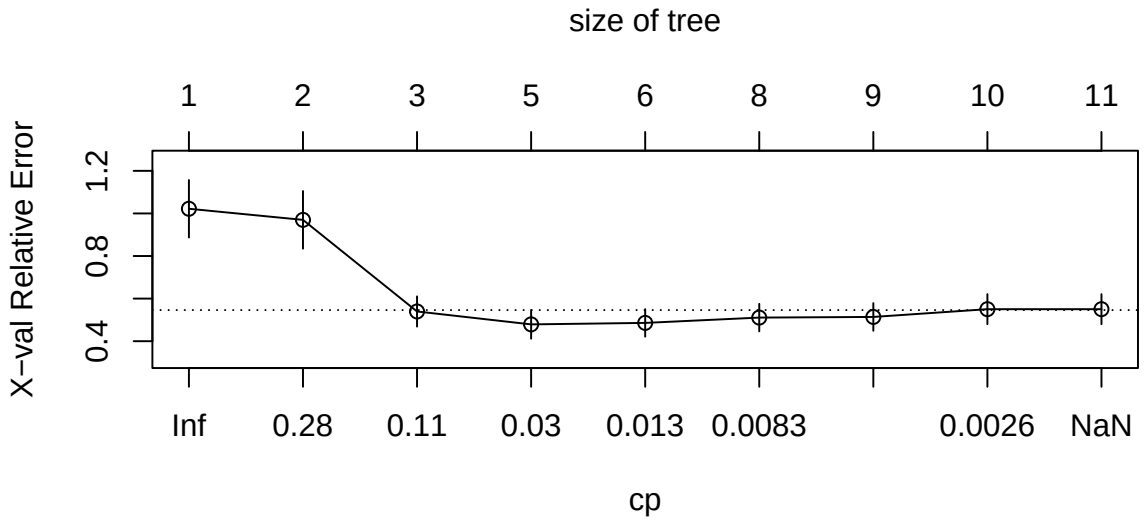

Figure 4.21: Relative error depending on the complexity parameter, for the synthetic data.

The same procedure to select λ as that used on the Seoul bike data can be done. In a first step,

we select the value that minimises the error:

```
min_val <-
  as_tibble(part_tree_large$cptable) %>%
  arrange(xerror) %>%
  slice(1)
min_val
```

```
## # A tibble: 1 x 5
##       CP nsplit `rel error` xerror   xstd
##    <dbl>  <dbl>       <dbl>  <dbl>  <dbl>
## 1 0.0185      4       0.344  0.479 0.0673
```

Then, we consider all values within a 1-standard error:

```
candidates <-
  as_tibble(part_tree_large$cptable) %>%
  filter( (xerror > min_val$xerror-min_val$xstd) &
            (xerror < min_val$xerror+min_val$xstd)) %>%
  arrange(desc(CP))
candidates
```

```
## # A tibble: 5 x 5
##        CP nsplit `rel error` xerror   xstd
##     <dbl>  <dbl>       <dbl>  <dbl>  <dbl>
## 1 0.0482       2       0.440  0.539 0.0711
## 2 0.0185       4       0.344  0.479 0.0673
## 3 0.00927      5       0.325  0.486 0.0647
## 4 0.00749      7       0.307  0.511 0.0646
## 5 0.00429      8       0.299  0.514 0.0649
```

And finally, we pick the values among the candidates that produces the shallowest tree:

```
cp_val <-
  candidates %>%
  slice(1) %>%
  magrittr::extract2("CP")
cp_val
```

```
## [1] 0.04824351
```

We can then prune the large tree:

```
pruned_tree <- prune(part_tree_large, cp=cp_val)
rpart.plot(pruned_tree)
```

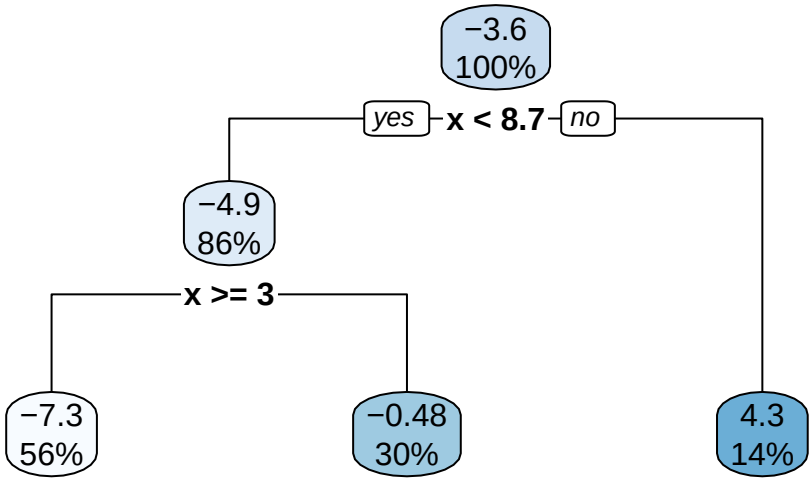

Figure 4.22: Pruned tree, synthetic data.

The depth of the pruned tree is indeed smaller:

```
nodes_pruned_tree <- as.numeric(rownames(pruned_tree$frame))
max(rpart:::tree.depth(nodes_pruned_tree))
```

```
## [1] 2
```

Let us visualise the boundaries of each tree:

```
boundaries <-
  tibble(
    x = seq(0, 10, by = .1),
    y = predict(part_tree_large,
                newdata = tibble(x=seq(0, 10, by = .1)))
  )
boundaries_pruned <-
  tibble(
    x = seq(0, 10, by = .1),
```

```
    y = predict(pruned_tree,
                newdata = tibble(x=seq(0, 10, by = .1)))
  )

ggplot(mapping = aes(x = x, y = y)) +
  geom_line(data = tibble(x=seq(0, 10, by = .1), y=f(x)),
            size = 1.1, linetype = "dashed", colour = "#AA2F2F") +
  geom_point(data = df_sim %>% mutate(sample = "Train") %>%
               bind_rows(
                 df_sim_new %>% mutate(sample = "Test")
               ),
             mapping = aes(shape = sample)) +
  geom_step(data = boundaries %>% mutate(model = "Large Tree") %>%
              bind_rows(
                boundaries_pruned %>% mutate(model = "Pruned Tree")
              ),
            mapping = aes(colour = model),
            # colour = "#005A8B",
            size = 1.5) +

  labs(title = str_c(
    "Decision boundaries for the **<span style='color:#E69F00'>",
    "large</span>** and for the **<span style='color:#56B4E9'>",
    "pruned</span>** trees")
  ) +
  scale_colour_manual("Boundary",
                      values = c("Large Tree" = "#E69F00",
                                 "Pruned Tree" = "#56B4E9")) +
  scale_shape_discrete("Dataset") +
  theme(
    plot.title = element_markdown(lineheight = 1.1),
    plot.title.position = "plot",
    legend.text = element_markdown(size = 11)
  )
```

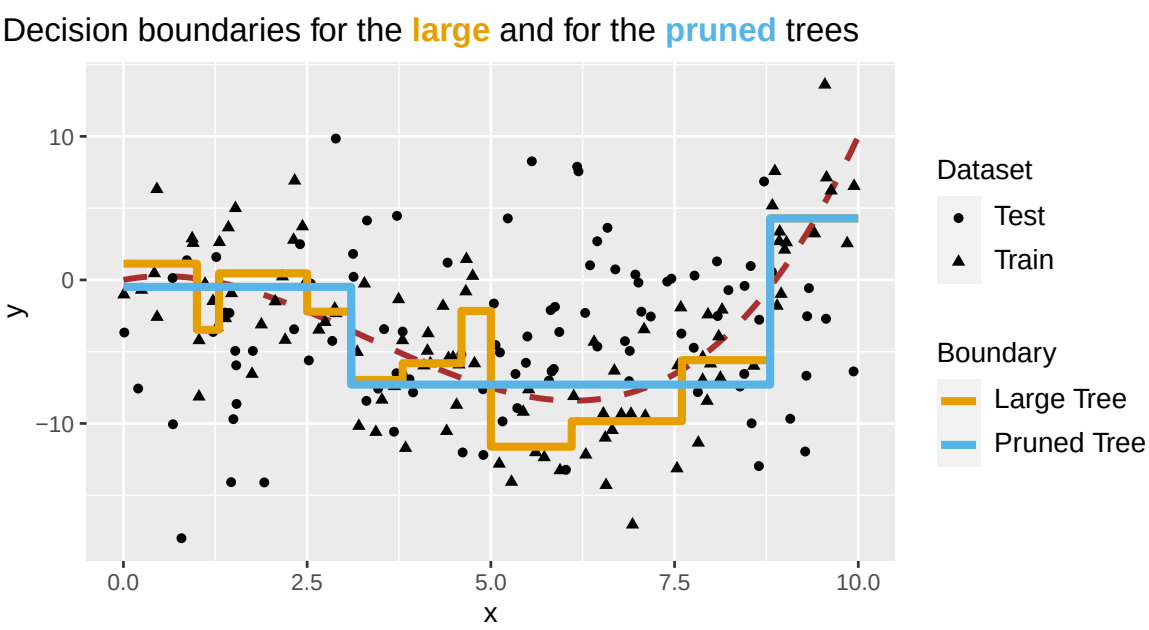

Figure 4.23: Decision boundaries are different after the tree was pruned.

Now, using these boundaries, let us compute the MSE on the unseen data (test set):

```
pred_test_large <- predict(part_tree_large, newdata = df_sim_new)
pred_test_pruned <- predict(pruned_tree, newdata = df_sim_new)

compute_mse <- function(observed, predicted){
  mean((observed-predicted)^2)
}


mse_large_tree <-
  compute_mse(observed = df_sim_new$y, predicted = pred_test_large)
mse_pruned_tree <-
  compute_mse(observed = df_sim_new$y, predicted = pred_test_pruned)

mse <- scales::number(c(mse_large_tree, mse_pruned_tree))
names(mse) <- c("Large", "Pruned")
print("MSE : \n")
```

```
## [1] "MSE : \n"
```

```
mse
```

```
##  Large Pruned
##   "65"   "47"
```

We avoided overfitting by pruning the tree and ended-up with a lower MSE on the test.

4.3.3 Classification Trees

So far, we have considered a **numerical** target/response variable. Let us now consider the case where the predictor is **categorical**. This implies two changes:

1. the criterion used to select the variable/cutoff pairs can no longer be the RSS

- we can use another metric, such as the **classification error rate** for example
- or the **Gini index**, or the **Entropy**

2. the prediction made for each partition is no longer the average of the response variable

- we can rely on a voting rule based on the proportions of the categories: for example, the proportions of each class can considered as the probabilities that the observations of a partition belong to each corresponding class.

Instead of using the classification error, it is more common to either use the **Gini impurity index** or the **entropy** to select the pair variable/cutoff that will be used to make a split.

The **Gini impurity index** at some node $\mathcal{N}$, as reminded in Emmanuel Flachaire's course, is given by:

$$G(\mathcal{N}) = \sum_{k=1}^{K} p_k(1-p_k) = 1 - \sum_{k=1}^{K} p_k^2,$$

where p_k is the fraction of observations (or *training samples*) labeled with class k in the node. If all the p_k are close to 0, or to 1, the Gini impurity index has a small value: in such cases, there will be mostly observations from the same class, the node will be homogeneous. The Gini impurity index thus gives an idea of the total variance across the K classes in a node.
Entropy is defined as follows:

$$E(\mathcal{N}) = -\sum_{k=1}^{K} p_k \log(p_k)$$

If the p_k are all near 0 or near 1, the entropy will also be near 0.
After a split into two leaves $\mathcal{N}_L$ and $\mathcal{N}_R$, the Gini impurity index becomes:

$$G(\mathcal{N}_L, \mathcal{N}_R) = p_L G(\mathcal{N}_L) + p_R G(\mathcal{N}_R),$$

where p_L and p_R are the proportion of observations in $\mathcal{N}_L$ and $\mathcal{N}_R$
In a similar fashion, the entropy becomes:

$$E(\mathcal{N}_L, \mathcal{N}_R) = p_L E(\mathcal{N}_L) + p_R E(\mathcal{N}_R).$$

Splits can be done as long as they substantially decrease impurity, *i.e.*, when

$$\Delta = G(\mathcal{N}) - G(\mathcal{N}_L, \mathcal{N}_R) > \epsilon,$$

where ϵ is a threshold value set by the user.
The choice of the pair variable/cutoff can be done so as to select the one that minimises the impurity, *i.e.*, the one that maximises Δ.

Let us look at an example on the Seoul bike data. First, let us look at how some little changes need to be made when calling the `rpart()` function.

With **regression trees**, the argument `method` was set to `"anova"`. Now that the variable of interest is categorical, we need to change the value to `"class"`. By default, the Gini impurity index is used as the splitting index. If we want to use entropy, we can feed the argument `parms` with a list with the element `split` equal to `"class"`.

Let us build a tree to predict our binary variable (recall that it is equal to `"High"` if the number

of hourly bikes is greater than 300 and "Low" otherwise). Let us make sure that at least 20 observations are in a node before a split is attempted (`minsplit = 20`) and that there are at least 7 observations in each terminal leaves (`minbucket = 7`).

```
classif_tree_gini <-
  rpart(y_binary ~.,
        data = df_train %>% select(-rented_bike_count, -date),
        method = "class",
        minsplit = 20,
        minbucket = 7,
        parms = list(split = "gini"))
```

The tree can be visualised thanks to the `rpart.plot()` function, as in the case of a regression tree. The argument `extra` allows to change the type of information reported on the graph. Here, since we have a binary response variable, the value is automatically set to 106 if not specified differently. It is made of two parts:

- 6: the probability of the second class only is reported
- +100: the percentage of observations in the node is also added

```
rpart.plot(classif_tree_gini, extra = 106)
```

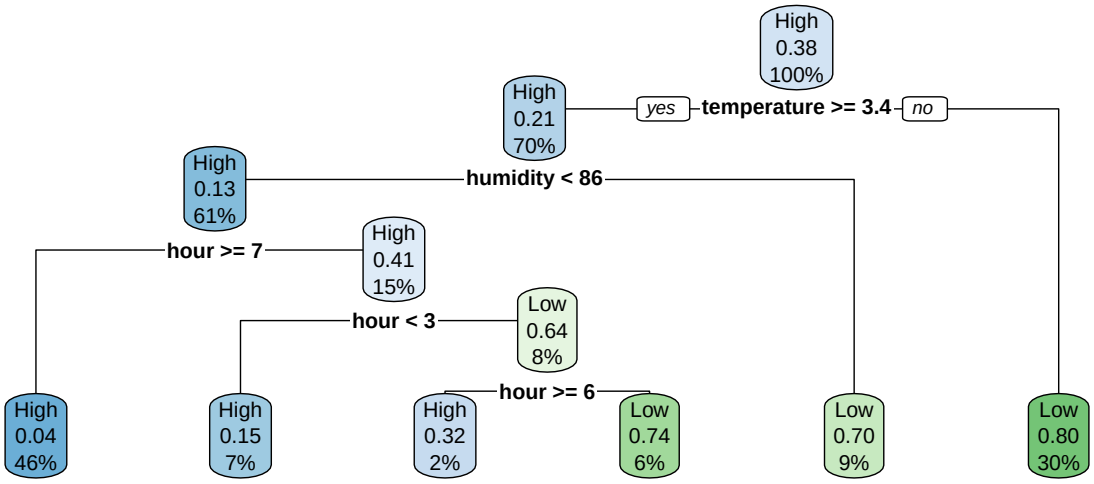

Figure 4.24: A first classification tree grown on Seoul bike data.

Here, at the top of the tree where 100% of observations are in the node, we read on the graph that the probability to be predicted as the second class (`Low`) is 0.38. This corresponds to the percentage of `Low` observations in the training dataset.

```
prop.table(table(df_train$y_binary))
```

```
##
##      High       Low
## 0.6151802 0.3848198
```

If the temperature is greater than or equal to 3.35 (we go to the left), we are left with 70% of the observations. In the resulting node, the probability to be classified as the second class (Low) is estimated at 0.21. This means that 21% of the observations in that node are labelled as Low:

```
df_train %>%
  filter(temperature >= 3.35) %>%
  group_by(y_binary) %>%
  count() %>%
  ungroup() %>%
  mutate(prop = round(n/sum(n), 2))
```

```
## # A tibble: 2 x 3
##   y_binary      n  prop
##   <chr>     <int> <dbl>
## 1 High       3754  0.79
## 2 Low         990  0.21
```

This concerns 70% of the observations:

```
df_train %>%
  group_by(temperature < 3.35) %>%
  count() %>%
  ungroup() %>%
  mutate(prop = n / sum(n))
```

```
## # A tibble: 2 x 3
##   `temperature < 3.35`      n  prop
##   <lgl>                 <int> <dbl>
## 1 FALSE                  4744 0.701
## 2 TRUE                   2028 0.299
```

If the temperature is strictly lower than 3.35 (we go to the right), which concerns the remaining 30% of the observations, there are 20% of observations label with the first class (`High`) and 80%with the second (`Low`):

```
df_train %>%
  filter(temperature < 3.35) %>%
  group_by(y_binary) %>%
  count() %>%
  ungroup() %>%
  mutate(prop = round(n/sum(n), 2))
```

```
## # A tibble: 2 x 3
##   y_binary     n  prop
##   <chr>    <int> <dbl>
## 1 High       412   0.2
## 2 Low       1616   0.8
```

This is a terminal leaf node, so any observation that falls in it, the predicted class will be that of the most frequent class in that node. With this tree, if an observation has `temperature >=3.35`:

- the probability that the number of bikes is of class "High" returned by the model will be 0.2
- the probability that the number of bikes is of class "Low" returned by the model will be 0.8.

When calling the plot, we set `extra=106`. As $106 \mod 100 = 6$, only the probability of the second class is repord. Here is the list of available values that can be used (this list is extracted from the help page of the `rpart()` function):

- `1`: the number of observations that fall in the node
- `2`: the classification rate at the node (number of correct classifications and the number of observations in the node)
- `3`: misclassification rate at the node (number of incorrect classifications and the number of observations in the node)
- `4`: probability per class of observations in the node (conditioned on the node, sum across a node is 1)
- `5`: like 4 but don't display the fitted class
- `6`: probability of the second class only. Useful for binary responses (the one we used)
- `7`: like 6 but don't display the fitted class
- `8`: probability of the fitted class
- `9`: probability relative to all observations – the sum of these probabilities across all leaves is 1. This is in contrast to the options above, which give the probability relative to observations falling in the node – the sum of the probabilities across the node is 1
- `10`: like 9 but display the probability of the second class only. (Useful for binary responses).
- `11`: like 10 but don't display the fitted class

```
rpart.plot(classif_tree_gini, extra = 102)
```

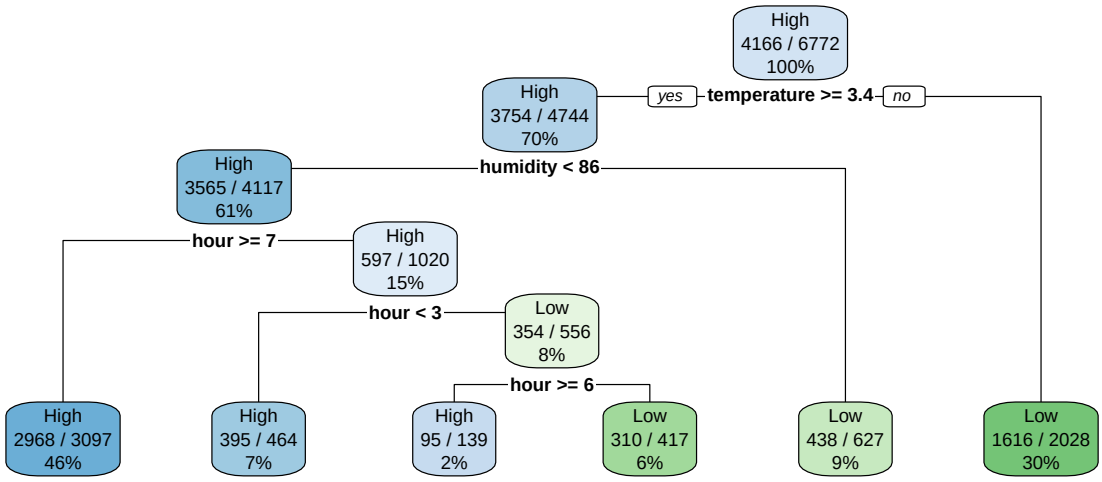

Figure 4.25: Showing the classification rate at the node.

If we want to use entropy instead of the Gini impurity index as the splitting index, all we need to do is to change the value of the element `split` in the list given to the `parms` argument so that it becomes `"information"` instead of `"gini"`.

```
classif_tree_entropy <-
  rpart(y_binary ~.,
        data = df_train %>% select(-rented_bike_count, -date),
        method = "class",
        parms = list(split = "information"))
rpart.plot(classif_tree_entropy)
```

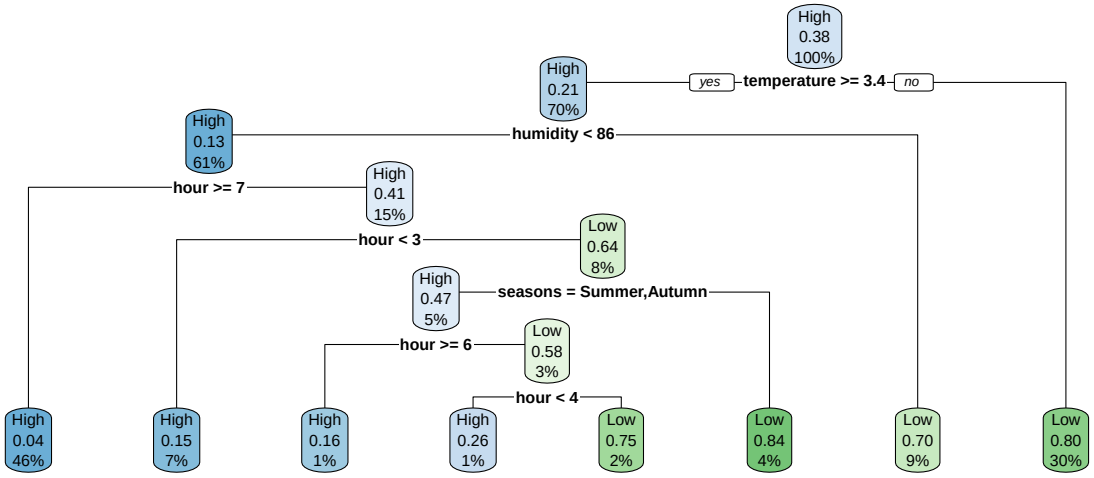

Figure 4.26: Classification tree build using entropy instead of gini to measure impurity index.

Similarly to what we did with regression trees, let us have a closer look at how the splitting rule is performed. Let us consider `temperature` as the splitting variable, and let us set a cutoff value of 10.

```
variable_split <- "temperature"
threshold <- 10
```

Let us consider the root as the current node:

```
current_node <- df_train
```

The proportion of each class, p_k, can be computed as follows:

```
prop_current_node <-
  current_node %>%
  group_by(y_binary) %>%
  count() %>%
  ungroup() %>%
  mutate(p_k = n / sum(n))
prop_current_node
```

```
## # A tibble: 2 x 3
##   y_binary     n   p_k
##   <chr>    <int> <dbl>
## 1 High      4166 0.615
```

```
## 2 Low        2606 0.385
```

The Gini impurity index in that current node is:

```
gini_node <-
  prop_current_node %>%
  mutate(g_k = p_k*(1-p_k)) %>%
  magrittr::extract2("g_k") %>%
  sum()
gini_node
```

```
## [1] 0.4734671
```

Let us define a function that computes the Gini impurity index on a given node:

```
#' @param data_node tibble/data.frame with the data of the node
#' @param target name of the target variable (discrete variable)
compute_gini <- function(data_node, target){

  prop_current_node <-
    data_node %>%
    group_by(!!sym(target)) %>%
    count() %>%
    ungroup() %>%
    mutate(p_k = n / sum(n))

  prop_current_node %>%
    mutate(g_k = p_k*(1-p_k)) %>%
    magrittr::extract2("g_k") %>%
    sum()
}
```

Which can be used this way:

```
compute_gini(current_node, "y_binary")
```

```
## [1] 0.4734671
```

Let us split the data into two partitions, depending on the splitting variable and the tested cutoff:

```
left_node <-
  current_node %>%
  filter(temperature < threshold)

right_node <-
  current_node %>%
  filter(temperature >= threshold)
```

The proportions of observations in each node p_L and p_R can be computed as follows:

```
p_L <- nrow(left_node) / nrow(current_node)
p_R <- nrow(right_node) / nrow(current_node)
cat(str_c("Prop left: ", p_L, "\n", "Prop right: ", p_R))
```

```
## Prop left: 0.427495569994093
## Prop right: 0.572504430005907
```

The Gini impurity coefficient in left and right nodes:

```
gini_left <- compute_gini(left_node, "y_binary")
gini_right <- compute_gini(right_node, "y_binary")

c(gini_left, gini_right)
```

```
## [1] 0.4353491 0.2749603
```

The weighted average of the Gini impurity indices:

```
gini_l_r <- p_L*gini_left + p_R * gini_right
gini_l_r
```

```
## [1] 0.3435258
```

Let us recap:

- the Gini impurity index in the current node is equal to 0.4734671

- the Gini impurity index after the split is equal to 0.3435258

Hence, the decrease in impurity (Δ) after the split is equal to:

```
gini_node - gini_l_r
```

```
## [1] 0.1299412
```

Equivalently, the split allowed to reduce the impurity by -0.27%:

```
(gini_l_r - gini_node)/gini_node
```

```
## [1] -0.2744462
```

Let us wrap-up the previous code in a function:

```
gini_split <- function(data_node, variable_to_predict,
                       variable_split, threshold,
                       minbucket){

  if(is.numeric(data_node[[variable_split]])){
    left_node <-
      data_node %>%
      filter(!!sym(variable_split) < threshold)

    right_node <-
      data_node %>%
      filter(!!sym(variable_split) >= threshold)
  }else{
    left_node <-
      data_node %>%
      filter(!!sym(variable_split) %in% threshold)

    right_node <-
      data_node %>%
      filter( !( !!sym(variable_split) %in% threshold ))
  }

  # If there is less than a given number of obs in the leaves
```

```
  # warning
  warning_min_bucket <-
    any( table(left_node[[variable_to_predict]]) < minbucket) |
    any( table(right_node[[variable_to_predict]]) < minbucket)

  # Proportions of obs in the resulting leaves
  p_L <- nrow(left_node) / nrow(current_node)
  p_R <- nrow(right_node) / nrow(current_node)

  # Gini in leaves
  gini_left <- compute_gini(left_node, "y_binary")
  gini_right <- compute_gini(right_node, "y_binary")

  gini_l_r <- p_L*gini_left + p_R * gini_right

  list(gini_l_r = gini_l_r, warning_min_bucket = warning_min_bucket)
}
```

This function can then be applied as follows:

```
gini_split(
  data_node = df_train, variable_to_predict = "y_binary",
  variable_split = "temperature", threshold = 10, minbucket = 7)
```

```
## $gini_l_r
## [1] 0.3435258
##
## $warning_min_bucket
## [1] FALSE
```

Let us consider multiple cutoffs:

```
number_of_cuts <- 1000
thresholds <-
  seq(min(df_train$temperature), max(df_train$temperature),
      length.out = number_of_cuts)
```

We can loop over these values of cutoff.

```
gini_leaves <-
  map(thresholds, ~gini_split(
    data_node = df_train, variable_to_predict = "y_binary",
    variable_split = "temperature", threshold = ., minbucket = 7))
```

The resulting values of $G(\mathcal{N}_L, \mathcal{N}_R)$ can be stored in a table:

```
gini_leaves_df <-
  tibble(
    g_leaves = map_dbl(gini_leaves, "gini_l_r"),
    warning_min_bucket = map_lgl(gini_leaves, "warning_min_bucket"),
    threshold = thresholds) %>%
  mutate(delta = gini_node - g_leaves,
         warning_min_bucket = factor(
           warning_min_bucket, levels = c(TRUE, FALSE),
           labels = c("Not permitted", "Permitted")))
gini_leaves_df
```

```
## # A tibble: 1,000 x 4
##    g_leaves warning_min_bucket threshold     delta
##       <dbl> <fct>                 <dbl>     <dbl>
##  1    0.473 Permitted             -17.8 0
##  2    0.473 Not permitted         -17.7 0.0000437
##  3    0.473 Not permitted         -17.7 0.0000437
##  4    0.473 Not permitted         -17.6 0.0000437
##  5    0.473 Not permitted         -17.6 0.0000437
##  6    0.473 Not permitted         -17.5 0.0000437
##  7    0.473 Not permitted         -17.5 0.0000704
##  8    0.473 Not permitted         -17.4 0.000158
##  9    0.473 Not permitted         -17.3 0.000158
## 10    0.473 Not permitted         -17.3 0.000158
## # ... with 990 more rows
```

When the threshold lead to a split in which there were too few observations (less than 7) in a leaf, we labelled the result as "Not permitted" in the `warming_min_bucket` column. The improvement obtained with each tested cutoff value can be visualised on a graph:

```
ggplot(data = gini_leaves_df,
       mapping = aes(x = threshold, y = delta)) +
  geom_point(mapping = aes(colour = warning_min_bucket)) +
```

```
  scale_colour_manual(
    "Threshold",
    values = c("Permitted" = "blue", "Not permitted" = "red")) +
  labs(x = "Temperature threshold") +
  theme(plot.title.position = "plot")
```

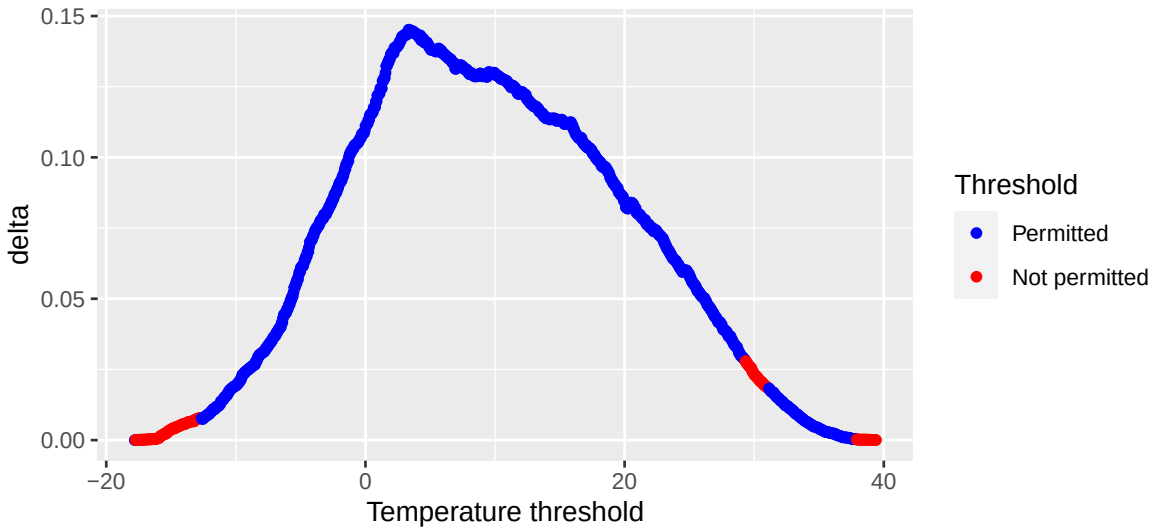

Figure 4.27: Decrease in the impurity design.

```
gini_leaves_df %>%
  filter(warning_min_bucket == "Permitted") %>%
  arrange(desc(delta))
```

```
## # A tibble: 852 x 4
##    g_leaves warning_min_bucket threshold delta
##       <dbl> <fct>                   <dbl> <dbl>
##  1    0.328 Permitted                3.33 0.145
##  2    0.328 Permitted                3.39 0.145
##  3    0.329 Permitted                3.44 0.145
##  4    0.329 Permitted                3.50 0.145
##  5    0.329 Permitted                3.61 0.145
##  6    0.329 Permitted                3.67 0.145
##  7    0.329 Permitted                3.73 0.144
##  8    0.329 Permitted                3.79 0.144
##  9    0.329 Permitted                3.56 0.144
## 10    0.330 Permitted                3.84 0.144
```

```
## # ... with 842 more rows
```

The best threshold for the split made with `temperature` is the one, among the permitted solutions, that leads to the highest impurity decrease:

```
gini_l_r_temperature <-
  gini_leaves_df %>%
  filter(warning_min_bucket == "Permitted") %>%
  arrange(desc(delta))
gini_l_r_temperature
```

```
## # A tibble: 852 x 4
##    g_leaves warning_min_bucket threshold delta
##       <dbl> <fct>                  <dbl> <dbl>
##  1    0.328 Permitted               3.33 0.145
##  2    0.328 Permitted               3.39 0.145
##  3    0.329 Permitted               3.44 0.145
##  4    0.329 Permitted               3.50 0.145
##  5    0.329 Permitted               3.61 0.145
##  6    0.329 Permitted               3.67 0.145
##  7    0.329 Permitted               3.73 0.144
##  8    0.329 Permitted               3.79 0.144
##  9    0.329 Permitted               3.56 0.144
## 10    0.330 Permitted               3.84 0.144
## # ... with 842 more rows
```

We obtain a value very close to that reported in the graph (the specific 3.35 value was not tested in our sequence of cutoffs).

> To be exhaustive, we should consider all other variables, and not only `temperature`, to select the best pair of variable/cutoff to perform the split.

If we look at the value reported in the `improve` column of the `splits` element returned by the `rpart()` function, we actually get $n_{\mathcal{I}} \times \Delta$, where $n_{\mathcal{I}}$ is the number of observations in the current node.

```
head(classif_tree_gini$splits)
```

```
##                       count ncat     improve index       adj
## temperature            6772    1 982.9149704  3.35 0.0000000
```

```
## seasons                 6772    4 793.1418098  1.00 0.0000000
## month                   6772   12 793.1418098  2.00 0.0000000
## dew_point_temperature   6772    1 539.9467740  2.65 0.0000000
## hour                    6772    1 434.9381343  6.50 0.0000000
## seasons                    0    4   0.9208506  3.00 0.7357002
```

Compared with:

```
nrow(current_node) * (gini_node-gini_l_r_temperature$g_leaves[1])
```

```
## [1] 982.915
```

More explanations on the rows of that table will be provided afterwards in this section and in the variable importance section.

Let us just give a few examples of other **competitor variables**. The element `splits` returned by the `rpart()` function contains, among other things, some information relative to those competitor variables. In our example, the first five rows of this `split` table report the improvement in the fit depending on the variable used to make the split (the value reported concerns the best cutoff for each variable tested).

```
head(classif_tree_gini$splits, 5)
```

```
##                        count ncat  improve index adj
## temperature             6772    1 982.9150  3.35   0
## seasons                 6772    4 793.1418  1.00   0
## month                   6772   12 793.1418  2.00   0
## dew_point_temperature   6772    1 539.9468  2.65   0
## hour                    6772    1 434.9381  6.50   0
```

Let us recall that the value in the `improve` column correspond to $n_{\mathcal{I}} \times \Delta$. The Gini impurity index in the root node is:

```
(gini_initial <- compute_gini(df_train, "y_binary"))
```

```
## [1] 0.4734671
```

If we use the temperature variable as the splitting one, the increase in the goodness of fit (as measured by the average of the Gini impurity index computed on the two resulting leaves) can be obtained with the following code:

```
gini_split_temperature <-
  gini_split(
    data_node = df_train, variable_to_predict = "y_binary",
    variable_split = "temperature", threshold = 3.35, minbucket = 7)

nrow(df_train) * (gini_initial - gini_split_temperature$gini_l_r)
```

```
## [1] 982.915
```

For the variable seasons:

```
gini_split_seasons <-
  gini_split(
    data_node = df_train, variable_to_predict = "y_binary",
    variable_split = "seasons", threshold = c("Winter"), minbucket = 7)

nrow(df_train) * (gini_initial - gini_split_seasons$gini_l_r)
```

```
## [1] 793.1418
```

For the variable month:

```
gini_split_month <-
  gini_split(
    data_node = df_train, variable_to_predict = "y_binary",
    variable_split = "month", threshold = c("Jan","Feb","Dec"), minbucket = 7)

nrow(df_train) * (gini_initial - gini_split_month$gini_l_r)
```

```
## [1] 793.1418
```

and so on.

4.3.4 Variable Importance

For a given node, the variable that is eventually used to perform the split is called a **primary variable**. We have seen that the choice of the primary split is done using a metric that computes the goodness of fit (for example, the Gini Index or the Impurity index in the case of a classification tree).

To compute the importance of a variable, we can also consider the **surrogate variables**. These variables are used to try to handle missing values. Once a primary split has been obtained, another variable can be used to try to obtain a similar assignment in the left and right nodes. If there are missing values when we use the primary split, then a surrogate split can be used to make the prediction. The surrogate variable may thus be relatively important if they are able to reproduce more or less the same results as the split obtained with the primary variable.

Let us have a quick look at how to measure the capacity of a surrogate variable to reproduce the same results as that obtained with a primary variable. To do so, we depart from what is explained in Section 3.4 of the vignette of {rpart}.

Once again, let us focus on the root node.

```
current_node <- df_train
```

The primary variable for that split, is `temperature`, and the best cutoff is `3.35`. Let us use the variable `seasons` as a surrogate. Among all the possible combinations of splitting rules for seasons, the one that gives the best results in terms of error rate is `"Winter"`.

```
tmp <-
  current_node %>%
  mutate(
    # Primary split
    leave_primary = case_when(
      temperature < 3.35 ~ "left prim.",
      temperature >= 3.35 ~ "right prim."),
    # Surrogate split
    leave_surrogate = case_when(
      seasons == "Winter" ~ "left surr.",
      seasons != "Winter" ~ "right surr."
    ))
```

Let us look at the results:

```
confusion <- table(tmp$leave_primary, tmp$leave_surrogate, exclude = NULL)
confusion
```

```
##
##                 left surr. right surr.
##    left prim.         1826          202
##    right prim.         334         4410
```

Using the surrogate, 1826 + 4410 = 6236 out of the 6772 observations are sent to the correct direction. This corresponds to an average proportion of correctly sent observations of 0.9208506.

```
correct_direction <- (confusion["left prim.", "left surr."] + confusion["right prim.", "right
correct_direction
```

```
## [1] 6236
```

```
# Proportion
correct_direction/sum(confusion)
```

```
## [1] 0.9208506
```

The majority rule gets 4744 correct:

```
maj_rule_correct <- sum(confusion["right prim.", ])
maj_rule_correct
```

```
## [1] 4744
```

Hence, the adjusted agreement is:

```
adjust <- (correct_direction - maj_rule_correct) / (sum(confusion) - maj_rule_correct)
adjust
```

```
## [1] 0.7357002
```

The relative importance of a variable is computed by accounting for when it appears as a primary variable, and also when it appears as a surrogate variable.

The composite metrics can be accessed through the `variable.importance` element of the result returned by the `rpart()` function.

```
classif_tree_gini$variable.importance
```

```
##           temperature                 month               seasons
##             989.14552             723.13074             723.13074
## dew_point_temperature                  hour              humidity
##             681.60244             372.01312             371.05659
##                  year              snowfall              rainfall
##             244.27473             179.32867             148.96535
##       solar_radiation            visibility            wind_speed
##              88.03473              51.16607              34.21947
```

When we call the `summary()` function on the result of the `rpart()` function, we get rounded values of the relative importance of the variables:

```
round(100 * classif_tree_gini$variable.importance /
        sum(classif_tree_gini$variable.importance))
```

```
##           temperature                 month               seasons
##                    21                    16                    16
## dew_point_temperature                  hour              humidity
##                    15                     8                     8
##                  year              snowfall              rainfall
##                     5                     4                     3
##       solar_radiation            visibility            wind_speed
##                     2                     1                     1
```

The only thing that remains is to understand how those values are obtained. To that end, we need to better understand the content of the `split` element that is returned by the `rpart()` function. In this table, primary variables, competitor variables (variables that would have produce a relatively close result in term of increase in the goodness of fit), and surrogate variables are given in a jumble.

```
splits_df <-
  as_tibble(classif_tree_gini$splits, rownames = "variable")
splits_df
```

```
## # A tibble: 43 x 6
##    variable              count  ncat improve   index   adj
##    <chr>                 <dbl> <dbl>   <dbl>   <dbl> <dbl>
##  1 temperature            6772     1 983.       3.35 0
##  2 seasons                6772     4 793.       1    0
##  3 month                  6772    12 793.       2    0
##  4 dew_point_temperature  6772     1 540.       2.65 0
##  5 hour                   6772     1 435.       6.5  0
##  6 seasons                   0     4   0.921    3    0.736
##  7 month                     0    12   0.921    4    0.736
##  8 dew_point_temperature     0     1   0.907   -3.85 0.688
##  9 year                      0     1   0.775 2018.   0.249
## 10 snowfall                  0    -1   0.755    0.05 0.182
## # ... with 33 more rows
```

We can use the results of table contained in the element `frame` of the result returned by the `rpart()` function. More specifically, the columns `ncompete` and `nsurrogate` tell us how many competitor variables and how many surrogate variables are reported in the table in the `splits` element, just after the primary variable.

```
head(classif_tree_gini$frame)
```

```
##            var    n   wt  dev yval  complexity ncompete
   nsurrogate     yval2.V1
## 1  temperature 6772 6772 2606    1 0.462010744         4
   5 1.000000e+00
## 2     humidity 4744 4744  990    1 0.095548734         4
   3 1.000000e+00
## 4         hour 4117 4117  552    1 0.029163469         4
   4 1.000000e+00
## 8       <leaf> 3097 3097  129    1 0.000000000         0
   0 1.000000e+00
## 9         hour 1020 1020  423    1 0.029163469         4
   5 1.000000e+00
##        yval2.V2     yval2.V3     yval2.V4     yval2.V5 yval2.
   nodeprob
## 1  4.166000e+03 2.606000e+03 6.151802e-01 3.848198e-01   1.000000
   e+00
```

```
## 2  3.754000e+03 9.900000e+02 7.913153e-01 2.086847e-01   7.005316
   e-01
## 4  3.565000e+03 5.520000e+02 8.659218e-01 1.340782e-01   6.079445
   e-01
## 8  2.968000e+03 1.290000e+02 9.583468e-01 4.165321e-02   4.573243
   e-01
## 9  5.970000e+02 4.230000e+02 5.852941e-01 4.147059e-01   1.506202
   e-01
##  [ getOption("max.print") est atteint -- ligne 1 omise ]
```

We can thus create a vector that can be added to the table that contains the information on the splits.

```
# Position of the primary variables in `frame`
ind_primary <- which(classif_tree_gini$frame$var != "<leaf>")

type_var_df <-
  classif_tree_gini$frame %>%
  slice(ind_primary) %>%
  select(ncompete, nsurrogate)

type_var <- NULL
for(i in 1:nrow(type_var_df)){
  type_var <- c(type_var, "primary",
                rep("competitor", type_var_df[i, "ncompete"]),
                rep("surrogate", type_var_df[i, "nsurrogate"])
  )
}
type_var
```

```
##  [1] "primary"    "competitor" "competitor" "competitor" "
   competitor"
##  [6] "surrogate"  "surrogate"  "surrogate"  "surrogate"  "
   surrogate"
## [11] "primary"    "competitor" "competitor" "competitor" "
   competitor"
## [16] "surrogate"  "surrogate"  "surrogate"  "primary"    "
   competitor"
## [21] "competitor" "competitor" "competitor" "surrogate"  "
   surrogate"
## [26] "surrogate"  "surrogate"  "primary"    "competitor" "
   competitor"
```

```
## [31] "competitor" "competitor" "surrogate"  "surrogate"  "
   surrogate"
## [36] "surrogate"  "surrogate"  "primary"    "competitor" "
   competitor"
## [41] "competitor" "competitor" "surrogate"
```

This vector can then be added in `splits_df`:

```
splits_df <- splits_df %>% mutate(type_var = type_var)
splits_df
```

```
## # A tibble: 43 x 7
##    variable              count  ncat improve   index   adj
   type_var
##    <chr>                 <dbl> <dbl>   <dbl>   <dbl> <dbl> <chr>
##  1 temperature            6772     1 983.       3.35 0
   primary
##  2 seasons                6772     4 793.       1    0
   competitor
##  3 month                  6772    12 793.       2    0
   competitor
##  4 dew_point_temperature  6772     1 540.       2.65 0
   competitor
##  5 hour                   6772     1 435.       6.5  0
   competitor
##  6 seasons                   0     4   0.921    3    0.736
   surrogate
##  7 month                     0    12   0.921    4    0.736
   surrogate
##  8 dew_point_temperature     0     1   0.907   -3.85 0.688
   surrogate
##  9 year                      0     1   0.775 2018.   0.249
   surrogate
## 10 snowfall                  0    -1   0.755    0.05 0.182
   surrogate
## # ... with 33 more rows
```

Let us focus once again on the variable `temperature`.

```
variable_of_interest <- "temperature"
```

Each primary split for this variable is credited with the improvement value:

```
var_imp <-
  splits_df %>%
  filter(variable == variable_of_interest, type_var == "primary") %>%
  magrittr::extract2("improve") %>%
  sum()
var_imp
```

```
## [1] 982.915
```

And each surrogate split where `temperature` is used gets `split$adj` times the primary split's value. We thus need to add a column that gives the primary split's value for each surrogate variable.

```
splits_df <-
  splits_df %>%
  mutate(primary_split_value =
           ifelse(type_var == "primary", improve, NA)) %>%
  fill(primary_split_value) %>%
  mutate(primary_split_value =
           ifelse(type_var != "surrogate", NA, primary_split_value))
splits_df
```

```
## # A tibble: 43 x 8
##    variable         count  ncat improve   index   adj type_var
   primary_split_va~
##    <chr>            <dbl> <dbl>   <dbl>   <dbl> <dbl> <chr>
                  <dbl>
##  1 temperature       6772     1 983.       3.35 0     primary
                   NA
##  2 seasons           6772     4 793.       1    0     competit~
                  NA
##  3 month             6772    12 793.       2    0     competit~
                  NA
##  4 dew_point_temp~   6772     1 540.       2.65 0     competit~
                  NA
##  5 hour              6772     1 435.       6.5  0     competit~
                  NA
##  6 seasons              0     4   0.921    3    0.736 surrogate
                 983.
##  7 month                0    12   0.921    4    0.736 surrogate
                 983.
##  8 dew_point_temp~      0     1   0.907   -3.85 0.688 surrogate
                 983.
```

```
##  9 year                0      1   0.775 2018.   0.249 surrogate
                  983.
## 10 snowfall            0     -1   0.755    0.05 0.182 surrogate
                  983.
## # ... with 33 more rows
```

We can compute the contribution of the variable `temperature` each time it is a surrogate variable to the importance metric:

```
var_imp_surrogate <-
  splits_df %>%
  filter(variable == variable_of_interest,
         type_var == "surrogate") %>%
  mutate(imp_contrib = primary_split_value*adj) %>%
  magrittr::extract2("imp_contrib") %>%
  sum()
var_imp_surrogate
```

```
## [1] 6.230553
```

When adding the two parts, we obtain the importance measure of the `temperature` variable:

```
var_imp+var_imp_surrogate
```

```
## [1] 989.1455
```

To compute the importance of all variables at once:

```
splits_df %>%
  mutate(importance = ifelse(type_var == "primary", improve, 0),
         importance = ifelse(
           type_var == "surrogate",
           importance+primary_split_value*adj, importance)) %>%
  group_by(variable) %>%
  summarise(importance = sum(importance)) %>%
  arrange(desc(importance))
```

```
## # A tibble: 12 x 2
##    variable               importance
##    <chr>                       <dbl>
##  1 temperature                 989.
##  2 month                       723.
##  3 seasons                     723.
##  4 dew_point_temperature       682.
##  5 hour                        372.
##  6 humidity                    371.
##  7 year                        244.
##  8 snowfall                    179.
##  9 rainfall                    149.
## 10 solar_radiation              88.0
## 11 visibility                   51.2
## 12 wind_speed                   34.2
```

We get the same values as those reported in the result of `rpart()` (although the values are rounded in the tibble):

```
classif_tree_gini$variable.importance
```

```
##           temperature                 month               seasons
##             989.14552             723.13074             723.13074
## dew_point_temperature                  hour              humidity
##             681.60244             372.01312             371.05659
##                  year              snowfall              rainfall
##             244.27473             179.32867             148.96535
##       solar_radiation            visibility            wind_speed
##              88.03473              51.16607              34.21947
```

We can plot those values using a barplot:

```
as_tibble(classif_tree_gini$variable.importance,
          rownames = "variable") %>%
  mutate(rel_importace = 100*value / sum(value)) %>%
  ggplot(data = .,
         mapping = aes(x = rel_importace,
                       y = fct_reorder(variable, rel_importace))) +
  geom_bar(stat = "identity") +
  labs(x = "relative importance", y = NULL) +
  theme(plot.title.position = "plot")
```

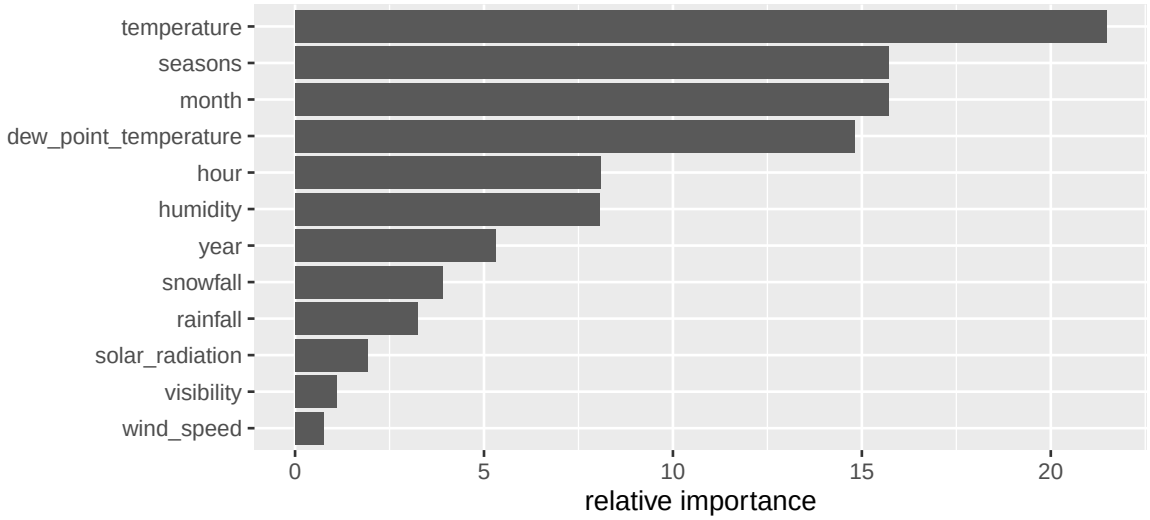

Figure 4.28: Variable importance for the Classification Tree.

4.4 Ensemble Methods

As explained in James et al. (2021), the predictive performances of trees are not very competitive with other machine learning algorithm. In addition, small changes in the data can lead to large changes in the final estimated tree. A way to overcome these issues is to rely on aggregating multiple decision trees. The basic idea is to build **weak learners**, *i.e.*, simple models that will provide predictions that may be mediocre but are consistently better than an random guessing. These weak learners can then be combined together: this forms an **ensemble method**.

This section will present three methods: bagging, random forest, and then boosting. Each of these methods are build using regression trees or classification trees as weak learners.

4.4.1 Bagging

Bagging (bootstrap aggregating) consists in generating multiple bootstrap from the data, estimate a tree model on each sample (building weak learners), and then averaging the predictions made by the weak learners to form new predictions. They were introduced in Leo Breiman (1996).

The new predictions are made as follows:

- in the case of a regression problem, the average of the values predicted by the weak learners can be calculated.

- in the case of a classification problem, a majority vote can be used or the estimated probabilities for each class can be averaged.

Proceeding this way offers the advantage to reduce the variance of the model. A known issue with regression trees is that they are sensitive so small changes in the data and therefore suffer from high variance. If another variable is used to make a split, or if a cutoff is changed, the output can be greatly modified. This issue can be reduced thanks to bootstrapping. Aggregating the results will lower the variance. From this, we understand that if we want the bootstrapping and aggregation to be efficient in lowering the variance, the weak learners need to be high variance models (otherwise, they would yield approximately the same results regardless of the changes made, and taking the average of values which do not vary much would not greatly diminish the variance).

If we want to get an unbiased prediction, with low variance, it may be a good idea to use weak learners with low bias. We can therefore use deep trees. Even if those deep trees will suffer from high variance, the bagging procedure will lower this variance.

A disadvantage of bagging compared to a regression or classification tree is that the results are no longer interpretable using a decision tree.

Let us illustrate how the method works with a regression problem. We use the same synthetic data that was used with the regression trees.

```
set.seed(123)
n <- 100
x <- runif(n=n, min=0, max=10)
eps <- rnorm(n, 0, 4)
f <- function(x) .1*x^3-1*x^2+x
y <- f(x)+eps
df_sim <- tibble(x=x, y=y)
```

We will create $B = 100$ bootstrap resamples of the data. On each sample, we will build an unpruned regression tree (cp=0).

```
n <- nrow(df_sim)
nb_bagg <- 100
baggin_models <- vector(mode = "list", length = nb_bagg)
for(i in 1:nb_bagg){

  ind_x <- sample(1:n, size = n, replace = TRUE)
  df_tmp <- df_sim %>% slice(ind_x)
```

```
  cart <-
    rpart(y ~ x,
          data = df_tmp,
          method = "anova",
          control = rpart.control(cp = 0)
    )

  baggin_models[[i]] <- cart

}
```

Each element of the object baggin_models contains a weak learner:

```
baggin_models[[1]]
```

```
## n= 100
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 100 3845.46700  -4.0083810
##    2) x< 8.465409 89 2476.34100  -5.2637170
##      4) x>=3.036704 60  940.14170  -7.6988470
##        8) x< 7.538915 44  752.24400  -8.4066460
##         16) x>=5.029507 24  291.07670  -9.9336230
##           32) x< 6.665854 14  108.26800 -10.7958400 *
##           33) x>=6.665854 10  157.82980  -8.7265200 *
##         17) x< 5.029507 20  338.05560  -6.5742730
##           34) x< 3.99258 10  107.92920  -9.0906140 *
##           35) x>=3.99258 10  103.48700  -4.0579310 *
##        9) x>=7.538915 16  105.23620  -5.7524020 *
##      5) x< 3.036704 29  444.28740  -0.2255172
##       10) x>=1.637487 12   77.95662  -1.9062080 *
##       11) x< 1.637487 17  308.50700   0.9608528 *
##    3) x>=8.465409 11   94.10378   6.1484350 *
```

Let us create a function that will make predictions on each weak learner on some data. The predictions for a given point obtained with the weak learners will then be averaged.

```
predict_bagged <- function(baggin_models, newdata){
  map(baggin_models, ~predict(., newdata = newdata)) %>%
```

```
    bind_cols(
      .name_repair = ~ vctrs::vec_as_names(
        ..., repair = "unique", quiet = TRUE)) %>%
    rowMeans()
}
```

For simplicity, let us create some new data ranging from 0 to 10:

```
new_data <- tibble(x = seq(0, 10, by = .1))
```

The predictions can then easily be calculated:

```
pred_bagging <- predict_bagged(baggin_models, new_data)
head(pred_bagging)
```

```
## [1] -0.1701479 -0.1701479 -0.1701479 -0.1701479 -0.1701479
    -0.2358646
```

Let us plot the observed values and the predictions obtained with bagging:

```
df_plot <-
  new_data %>%
  mutate(pred = pred_bagging,
         expected = f(x))

ggplot(mapping = aes(x = x, y = y)) +
  geom_point(data = df_sim) +
  geom_line(data = df_plot %>% pivot_longer(-x, values_to = "y"),
            mapping = aes(colour = name),
            size = 1.1)
```

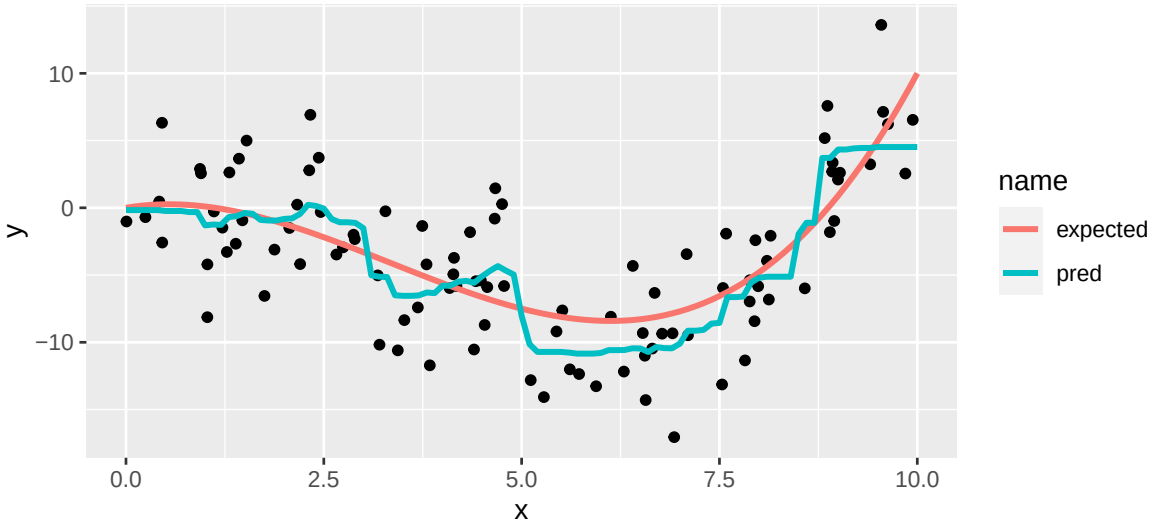

Figure 4.29: Bagging with 100 bootstrap training sets.

4.4.1.1 Number of Trees

What happens when more trees are grown? To have an idea, let us make the number of trees vary. A little function that samples from the data and then grow an unpruned tree may help.

```
train_weak_learner <- function(data, formula, method, control){
  n <- nrow(data)
  ind_x <- sample(1:n, size = n, replace = TRUE)
  df_tmp <- data[ind_x,]
  cart <- rpart(formula = formula,
                data = df_tmp,
                method = method,
                control = control)
  cart
}
```

This function can then be used to grow many trees. We will call it 500 times so that 500 unpruned trees are grown. Here, we only have a few observations an a single predictor, so growing a tree is really fast. But with bigger data, it may be interesting to grow the trees using **parallel computations**. We will do so here, to present an easy way to implement parallel computations.

First, some libraries need to be loaded:

```
library(foreach)
library(doSNOW)
```

We will use all the cores of the machine except for one.

```
ncl <- parallel::detectCores()-1
(cl <- makeCluster(ncl))
```

```
## groupe de processus socket avec 7 noeuds sur l'hôte 'localhost'
```

```
registerDoSNOW(cl)
```

Then, a parallel loop can be defined this way:

```
baggin_models_2 <- foreach(
  i = 1:500,
  .packages=c("rpart")
  ) %dopar% {
  train_weak_learner(
    data = df_sim,
    formula = y~x,
    method="anova",
    control = rpart.control(cp = 0))
}
```

Once we are finished, we can close the workers.

```
stopCluster(cl)
```

We are left with a list of 500 elements, each containing a week learner.

```
class(baggin_models_2) ; length(baggin_models_2)
```

```
## [1] "list"
```

```
## [1] 500
```

Let us make predictions on our sequence of values ranging from 0 to 1 (`new_data`). Those predictions will be made using either :

- 10 trees
- 75 trees
- 250 trees
- 500 trees

```
df_plot_2 <-
  map(c(10, 75, 250, 500), ~add_column(
  new_data,
  nb_bagg = as.character(.),
  pred = predict_bagged(baggin_models_2[1:.], new_data))) %>%
  bind_rows()
```

The results can be visualized this way:

```
ggplot() +
  geom_line(
    data = tibble(x=seq(0, 10, by = .1), y=f(x)),
    mapping = aes(x = x, y = y),
    colour = "black", size = 1.1, linetype = "dashed") +
  geom_point(data = df_sim,
             mapping = aes(x = x, y = y)) +
  geom_line(data = df_plot_2, size = 1.1,
            mapping = aes(x = x, y = pred, colour = nb_bagg)) +
  scale_colour_discrete("Number of weak learners") +
  theme(plot.title.position = "plot")
```

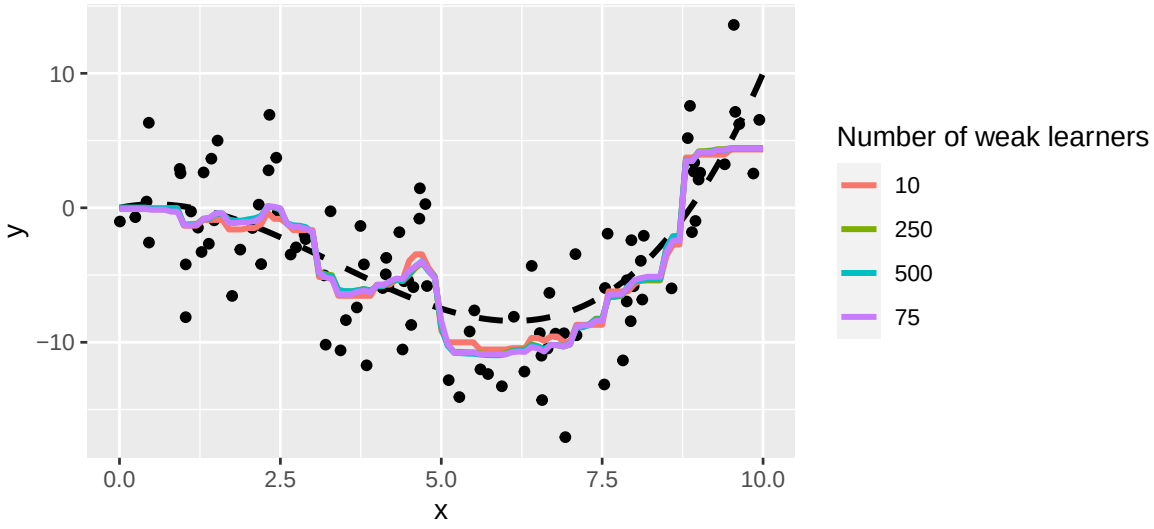

Figure 4.30: Predictions made from the aggregations.

It can be noted that increasing the number of trees smoothes the prediction surface.

What is the effect of increasing the number of trees on the predictive capacities of the model? We can explore this with simulation – again.

First, we need to get some out of sample data.

```
set.seed(234)
x_new <- runif(n=n, min=0, max=10)
eps <- rnorm(n, 0, 4)
y_new <- f(x)+eps
df_sim_new <- tibble(x=x_new, y=y_new)
```

We will compute the Mean Squared Error (MSE) on these unseen data. We will do this by varying the number of weak learners, in order to visualise the evolution of the MSE as a function of the number of trees.

```
mse_test_bagging <-
  map(1:500, ~add_column(
  df_sim_new,
  nb_bagg = .,
  pred = predict_bagged(baggin_models_2[1:.], df_sim_new))) %>%
  map(~mutate(., mse = (y-pred)^2)) %>%
  map_df(~summarise(., nb_bagg=unique(nb_bagg), mse = mean(mse)))
```

Then, the results can be plotted.

```
ggplot(data = mse_test_bagging,
       mapping = aes(x = nb_bagg, y = mse)) +
  geom_line() +
  labs(x = "Number of trees (weak learners)", y = "MSE") +
  theme(plot.title.position = "plot")
```

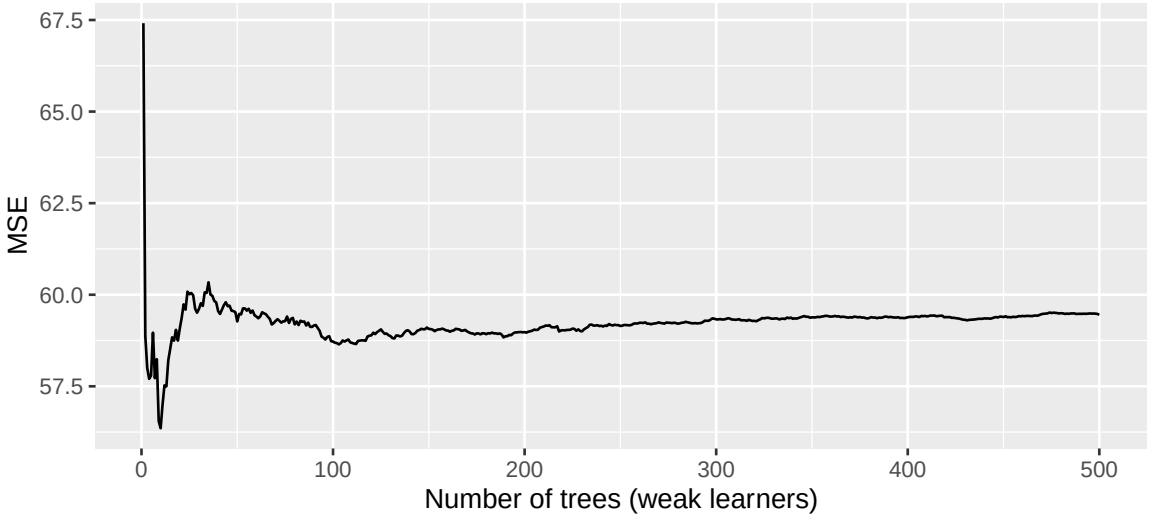

Figure 4.31: MSE vs number of trees.

We can note two things from this graph:

- after a certain number of trees used is reached, the MSE seems to stabilise: here, we could use around 150 trees only
- using more trees than what would be enough does not lead to overfitting.

4.4.1.2 Out-of-bag estimations

We just had a look at the performance of our models using the validation set approach. Actually, we could have a look at how the model performs on unseen data differently. When a bootstrap sample is used to create a tree, on average, only two third of the observations end up in the sample. Hence, the remaining observations can be used to make out-of-sample predictions. The conventional term for these out-of-sample observations is: **out-of-bag** observations. Averaging

the predicted value of an observation over the learners in which it was out-of-bag gives an **out-of-bag prediction**. Doing it for all of the observations of the sample allows us to compute the **overall out-of-bag MSE**.

> As the out-of-bag predictions are made on unseen data, the out-of-bag MSE is computed using unseen data and can therefore be used to assess the goodness of fit of the model.

This can easily be implemented with a function. After the unpruned tree is grown, all we need to do is to use it to make predictions on the left-out data. We need to keep track on which data it concerns, though.

```
train_weak_learner <- function(data, formula, method, control){
  n <- nrow(data)
  ind_x <- sample(1:n, size = n, replace = TRUE)
  df_tmp <- data[ind_x,]
  cart <- rpart(formula = formula,
                data = df_tmp,
                method = method,
                control = control)

  # oob predictions
  oob_pred <- predict(cart, newdata = data[-ind_x,])
  names(oob_pred) <- seq(1,n)[-ind_x]

  list(estim = cart, oob_pred = oob_pred)
}
```

Let us use a loop to build 500 trees.

```
library(doSNOW)
ncl <- parallel::detectCores()-1
(cl <- makeCluster(ncl))
```

```
## groupe de processus socket avec 7 noeuds sur l'hôte 'localhost'
```

```
registerDoSNOW(cl)

baggin_models_3 <- foreach(
```

```
  i = 1:500,
  .packages=c("rpart")) %dopar% {

  train_weak_learner(
    data = df_sim,
    formula = y~x,
    method="anova",
    control = rpart.control(cp = 0))
}

stopCluster(cl)
```

The out-of-bag predictions can the be computed:

```
oob_pred <-
  map(baggin_models_3, "oob_pred") %>%
  map_df(~as_tibble(., rownames = "id")) %>%
  bind_rows() %>%
  mutate(id = as.numeric(id)) %>%
  rename(pred = value) %>%
  group_by(id) %>%
  summarise(nb_seen = n(),
            pred = mean(pred))
oob_pred
```

```
## # A tibble: 100 x 3
##       id nb_seen    pred
##    <dbl>   <int>   <dbl>
##  1     1     192  -1.62
##  2     2     161  -6.02
##  3     3     202  -5.47
##  4     4     181   2.52
##  5     5     194   4.42
##  6     6     179  -0.985
##  7     7     170 -10.3
##  8     8     181   3.95
##  9     9     187 -11.3
## 10    10     174  -4.68
## # ... with 90 more rows
```

Just for the sake of it, we can check how many times each observations has been used as an out-of-bag observation over the 500 trees:

```
summary(oob_pred$nb_seen)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   157.0   175.8   183.0   182.2   189.0   207.0
```

The out-of-bag MSE can then be computed:

```
df_sim %>%
  mutate(id = row_number()) %>%
  left_join(oob_pred) %>%
  mutate(error_sq = (y-pred)^2) %>%
  summarise(mse = mean(error_sq))
```

```
## # A tibble: 1 x 1
##     mse
##   <dbl>
## 1  15.6
```

Now, let us make the number of tree vary and look how it affects the out-of-bag MSE. Again, a helper function may be convenient.

```
#' Computes the OOB MSE using `nb_tree` trees
#' @param nb_tree number of trees used (weak learners)
compute_oob_mse_nb_trees <- function(nb_tree){
  oob_pred <-
    map(baggin_models_3[1:nb_tree], "oob_pred") %>%
    map_df(~as_tibble(., rownames = "id")) %>%
    bind_rows() %>%
    mutate(id = as.numeric(id)) %>%
    rename(pred = value) %>%
    group_by(id) %>%
    summarise(nb_seen = n(),
              pred = mean(pred))

  df_sim %>%
    mutate(id = row_number()) %>%
```

```
    left_join(oob_pred, by = "id") %>%
    mutate(error_sq = (y-pred)^2) %>%
    summarise(mse = mean(error_sq, na.rm=TRUE)) %>%
    mutate(nb_tree = nb_tree)
}
```

Let us compute the out-of-bag MSE on models that use from 1 to 300 trees. With a too small number of trees, as all the observations will not necessarily be left-out, computing the mean would result in a NA value, unless we specify that NA values should be discarded before computing the average.

```
oob_mse_nb_trees <- map_df(1:300, compute_oob_mse_nb_trees)
```

Let us visualise the results:

```
ggplot(data = oob_mse_nb_trees,
       mapping = aes(x = nb_tree, y = mse)) +
  geom_line() +
  labs(x = "Number of trees grown (weak learners", y = "MSE") +
  theme(plot.title.position = "plot")
```

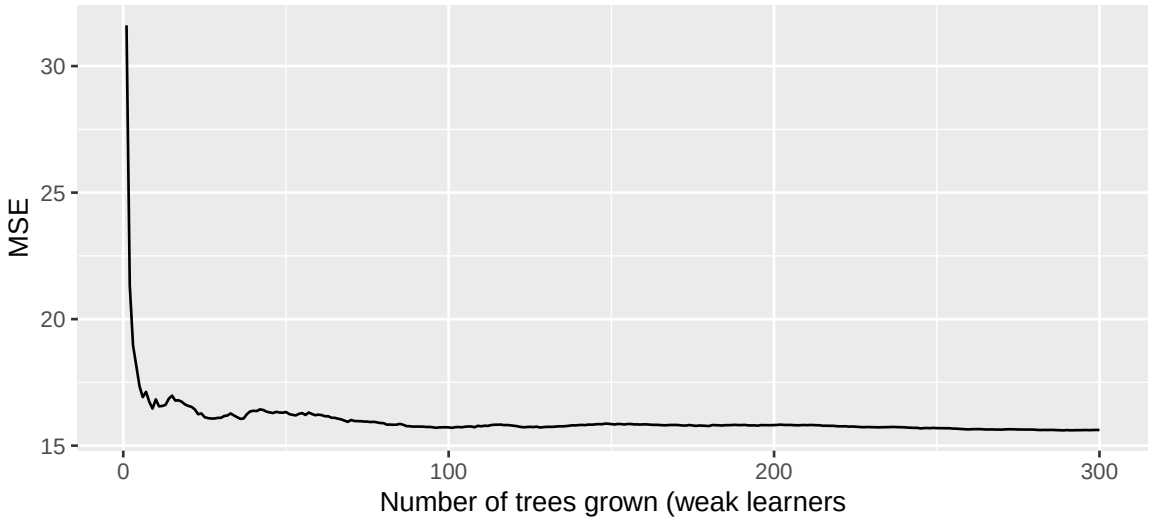

Figure 4.32: Out-of-bag MSE - Bagging.

As in the validation approach, we are able to see with this graph that after a certain number of

trees, the error no longer decreases.

4.4.1.3 Variable Importance Measures

To assess the variable importance with bagging, we can proceed in a similar fashion to what we have done with regression or classification trees.

For each tree, we can compute the decrease in the RSS in the case of a regression problem, or the improvement in the Gini impurity index (or Entropy) in the case of a classification problem, for each variable. Then, these values can be averaged over the trees. Again, if the value is large, it hints that the variable is an important predictor.

We can go back to our bike data to illustrate the method. First, we need to grow many unpruned trees.

```
library(doSNOW)
ncl <- parallel::detectCores()-1
(cl <- makeCluster(ncl))
```

```
## groupe de processus socket avec 7 noeuds sur l'hôte 'localhost'
```

```
registerDoSNOW(cl)

bagging_models_bike <- foreach(
  i = 1:200,
  .packages=c("rpart", "dplyr")
  ) %dopar% {

  train_weak_learner(
    data = df_train %>% select(-y_binary, -date),
    formula = rented_bike_count~.,
    method="anova",
    control = rpart.control(cp = 0))
}

stopCluster(cl)
```

Then, we can extract the variable importance measures from all these trees, and average the values over the trees:

```
variable_importance <-
  map(bagging_models_bike, "estim") %>%
  map("variable.importance") %>%
  bind_rows() %>%
  colMeans()
```

The relative measure can be computed as follows:

```
variable_importance_df <-
  as_tibble(100*variable_importance/sum(variable_importance),
            rownames = "variable")
variable_importance_df
```

```
## # A tibble: 14 x 2
##    variable                value
##    <chr>                   <dbl>
##  1 temperature            18.0
##  2 hour                   15.1
##  3 dew_point_temperature  14.7
##  4 month                  14.4
##  5 seasons                11.2
##  6 solar_radiation         7.00
##  7 humidity                6.89
##  8 rainfall                3.41
##  9 year                    2.94
## 10 wind_speed              2.40
## 11 week_day                1.91
## 12 visibility              1.97
## 13 holiday                 0.0892
## 14 snowfall                0.0670
```

And we can visualise it using a barplot.

```
ggplot(data = variable_importance_df,
       mapping = aes(x = value, y = fct_reorder(variable, value))) +
  geom_bar(stat="identity")+
  labs(x = NULL, y = NULL) +
  theme(plot.title.position = "plot")
```

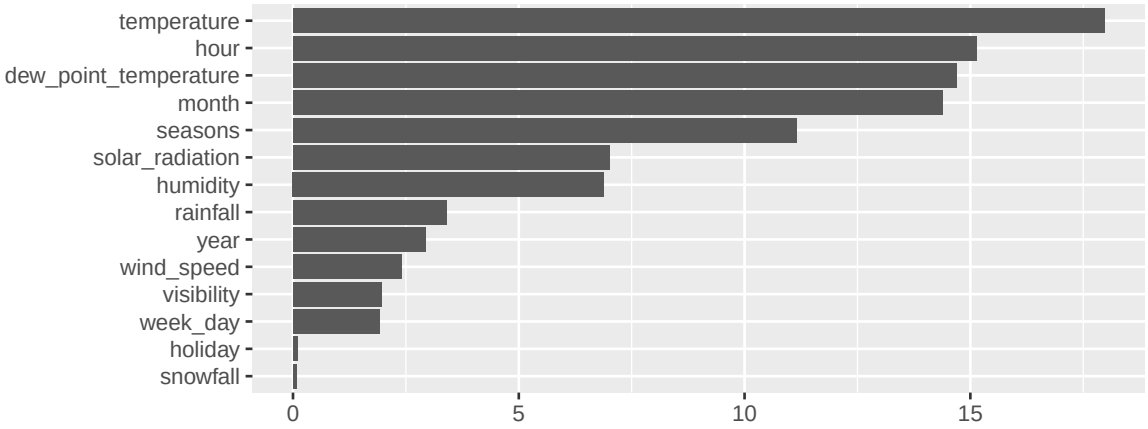

Figure 4.33: Variable relative importance.

4.4.1.4 Pre-built Function

Rather than using the previous codes which are not optimised for fast running, we can use the `bagging()` function of {ipred}. Having unpacked the method, it is easy to understand the use of this function. First, the package {ipred} can be loaded:

```
library(ipred)
```

The number of trees grown is controlled by the argument `nbagg`. To obtain out-of-bat estimate of the error rate, the `coob` argument needs to be set to `TRUE`. The argument `control` can be fed with a list of values for the different parameters we want for the weak learners. We want to build unpruned trees, so we set `cp=0`. Then we can play with the other parameters, such as the minimum number of split `minsplit` (but this may result in pruned trees), the minimum observations in the terminal leaves `minbucket`, the number of competitor variable we want to retain in the output `maxcompete`, the number of surrogate splits retained in the output `maxsurrogate`, etc. Let us just set `cp=0` here.

```
mod_bag <- bagging(
  formula = rented_bike_count ~ .,
  data = df_train %>% select(-y_binary, -date),
  nbagg = 100,
  coob = TRUE,
  control = rpart.control(cp = 0)
)
```

The trees are stored in the element `mtrees`

```
class(mod_bag$mtrees) ; length(mod_bag$mtrees)
```

```
## [1] "list"
```

```
## [1] 100
```

If we look at the first element of `mtrees`, we can notice that it contains two elements:

- `bindx`: the indices of the observations used to grow the tree (row number of the initial data)
- `btree`: the estimated weak learner

```
names(mod_bag$mtrees[[1]])
```

```
## [1] "bindx" "btree"
```

```
head(mod_bag$mtrees[[1]]$bindx)
```

```
## [1] 4411  188 1348 1931 4869 2808
```

The out-of-bag error is stored in the `err` element. The reported value depends on the type of problem:

- for regression problems, it is the root mean squared error (RMSE)
- for classification problems, it is the misclassification rate.

```
mod_bag$err
```

```
## [1] 175.0919
```

The variable importance can be obtained using the methodology we used. But we can also rely on a very convenient package (which will be used afterwards with random forests): {caret}. This package offers a function called `varImp()`.

```
library(caret)
variable_importance <- varImp(mod_bag)
variable_importance
```

```
##                          Overall
## dew_point_temperature 66.062358
## holiday                2.583719
## hour                  55.000256
## humidity              60.226333
## month                 44.683457
## rainfall               7.881722
## seasons               13.489418
## snowfall               5.235853
## solar_radiation       41.959113
## temperature           77.962421
## visibility            53.343140
## week_day              69.950691
## wind_speed            46.141727
## year                   6.792069
```

And once again the values can be plotted on a barplot:

```
ggplot(data = as_tibble(variable_importance, rownames = "variable"),
       mapping = aes(x = Overall, y = fct_reorder(variable, Overall))) +
  geom_bar(stat="identity")+
  labs(x = NULL, y = NULL) +
  theme(plot.title.position = "plot")
```

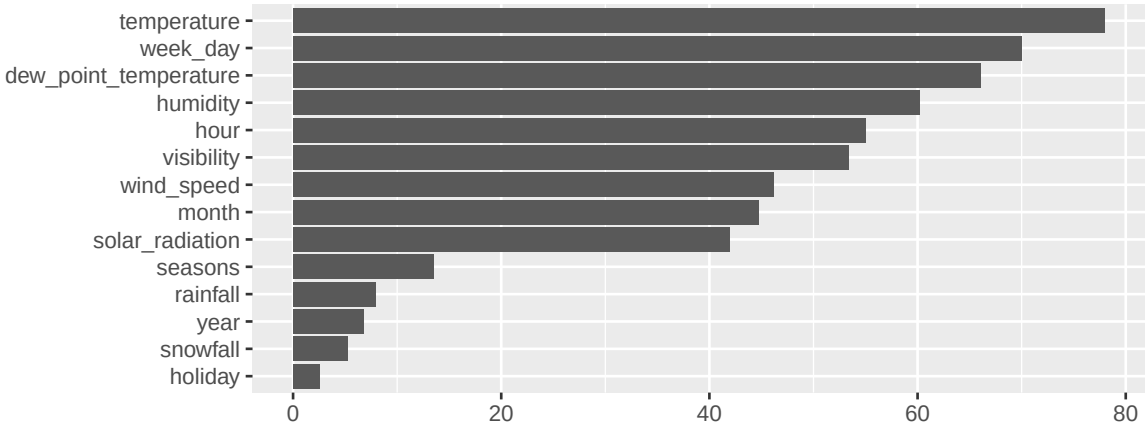

Figure 4.34: Variable relative importance.

4.4.2 Random Forests

The idea behind random forest is very similar to what is done with bagging. Many decision trees are build using bootstrapped training samples, but instead of considering all the variables as candidates each time a split is attempted, only a random sample of the explanatory variables is considered (Leo Breiman 2001). Proceeding this way allows to **decorrelate the trees.** In fact, with bagging, the trees grown are not completely independent.

If wee look at the top of the first trees grown using bagging, we can notice that the variables used to make the first splits is often the same, and the cutoff is also often very similar, despite the fact that the observations were randomly drawn to create bootstrap training samples:

```
tree_1_pruned <- prune(mod_bag$mtrees[[1]]$btree, cp=.1)
tree_2_pruned <- prune(mod_bag$mtrees[[2]]$btree, cp=.1)
tree_3_pruned <- prune(mod_bag$mtrees[[3]]$btree, cp=.1)
tree_4_pruned <- prune(mod_bag$mtrees[[4]]$btree, cp=.1)

op <- par()
par(mfrow = c(2,2))
rpart.plot(tree_1_pruned, main = "Tree 1")
rpart.plot(tree_2_pruned, main = "Tree 2")
rpart.plot(tree_3_pruned, main = "Tree 3")
rpart.plot(tree_4_pruned, main = "Tree 4")
```

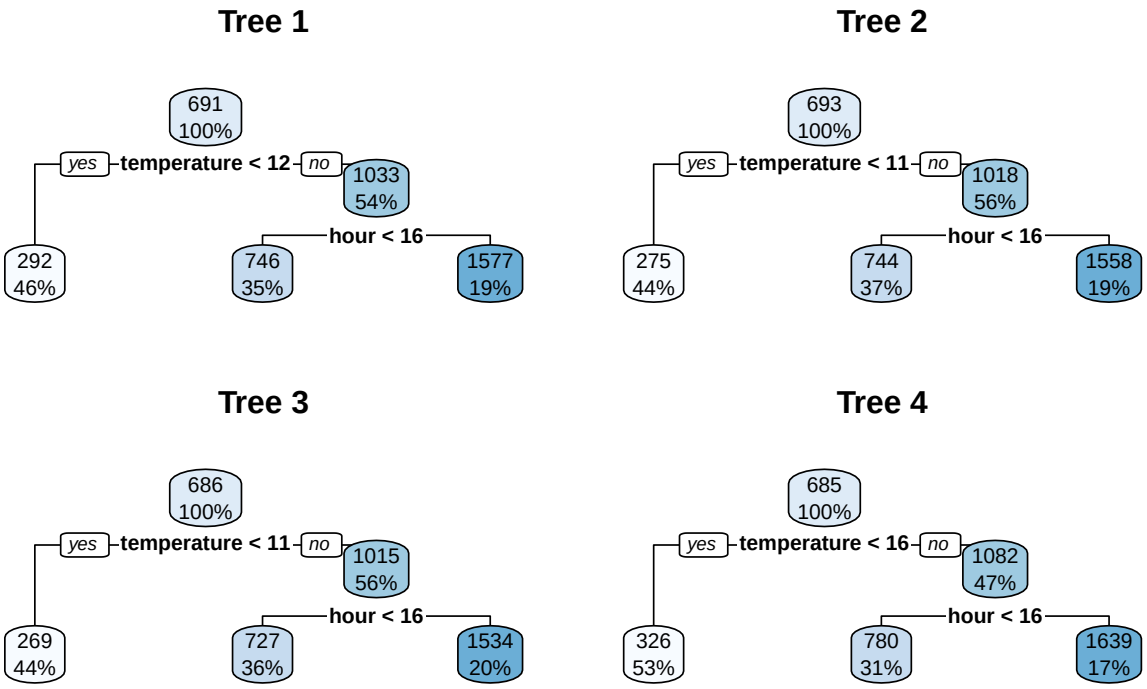

Figure 4.35: Trees may be correlated when use bagging.

```
par(op)
```

By sampling m variables among the p predictors so that the former are considered as competitors to make a split, random forests overcome this pitfall. Averaging the predictions made on decorellated trees will produce better results, with less variance.

Default values for the number of variables randomly samples as competitors/candidates at each split are usually :

- $m = p/3$ for regression problems
- $m = \sqrt{p}$ for classification problems.

4.4.2.1 A First Example with {randomForest}

The package {randomForest} provides a function called `randomForest()` which implements Breiman's random forest algorithm for classification and regression. Let us see how to use that function. First, the package can be loaded.

```
library(randomForest)
```

The `formula` and the `data` arguments can once again be used. Other arguments allow us to tune the algorithm:

- `ntree`: number of trees to grow
- `mtry`: number of variables randomly sampled as candidates at each split.
- `nodesize`: minimum size of terminal nodes. The default value is:
  - 1 for classification problems
  - 5 for regressio problems
- `maxnodes`: maximum number of terminal nodes trees in the forest can have. If not given, trees are grown to the maximum possible (subject to limits by `nodesize`). If set larger than maximum possible, a warning is issued.

Let us grow 200 trees and set the number of variables randomly sampled as candidates at each split (m) to 5, *i.e.*, the square root of our number of predictors. We will define `nodesize` and `maxnodes` with their default values.

```
mod_rf <-
  randomForest(
  formula = rented_bike_count ~ .,
  data = df_train %>% select(-y_binary, -date),
  ntree = 200,
  mtry = round(ncol(select(df_train, -y_binary, -date))/3),
  nodesize = 5,
  maxnodes = NULL
)
```

The out-of-bag MSE computed using 1 to 100 trees are stored in the `mse` elements of the model:

```
df_plot_mse_oob_rf <-
  tibble(mse = mod_rf$mse) %>%
  mutate(nb_tree = row_number())
df_plot_mse_oob_rf
```

```
## # A tibble: 200 x 2
##       mse nb_tree
##     <dbl>   <int>
##  1 95175.       1
##  2 88586.       2
##  3 82743.       3
```

```
##  4 70951.        4
##  5 66796.        5
##  6 62666.        6
##  7 59989.        7
##  8 56013.        8
##  9 53375.        9
## 10 51505.       10
## # ... with 190 more rows
```

For comparison, let us compute the MSE obtained on a test sample. We can use the `predict()` function from {randomForest} to get the predicted values for all the trees, using the out-of-bag sample (to my understanding, it is not possible to directly ask for the predictions made using a single tree).

```
predictions_test_tree_wise <-
  predict(mod_rf, newdata = df_test, predict.all = TRUE)
```

The `individual` element of the returned object contains a matrix with the predicted values. Each column corresponds to the estimated values of a single tree.

```
dim(predictions_test_tree_wise$individual)
```

```
## [1] 1693  200
```

All we need to do is to compute the MSE using up to `nb_tree`:

```
#' Computes the MSE using `nb_tree` trees
compute_mse_trees <- function(nb_tree){
  pred <- predictions_test_tree_wise$individual[,1:nb_tree]
  if(nb_tree > 1) pred <- rowMeans(pred)
  tibble(mse = mean((df_test$rented_bike_count - pred)^2),
         nb_tree = nb_tree)
}
```

```
df_plot_mse_test_rf <- map_df(1:200, compute_mse_trees)
df_plot_mse_test_rf
```

```
## # A tibble: 200 x 2
##         mse nb_tree
```

```
##       <dbl>   <int>
##  1 336998.        1
##  2 246233.        2
##  3 232757.        3
##  4 194906.        4
##  5 167375.        5
##  6 163933.        6
##  7 161718.        7
##  8 161246.        8
##  9 158161.        9
## 10 157843.       10
## # ... with 190 more rows
```

For the sake of comparison, let us compare these MSE to that obtained with bagging.

```
mod_bag <- bagging(
  formula = rented_bike_count ~ .,
  data = df_train %>% select(-y_binary, -date),
  nbagg = 200,
  coob = TRUE,
  control = rpart.control(cp = 0)
)
```

Let us first compute the out-of-bag MSE.

```
#' Computes out-ob-bag predictions for the ith tree
compute_oob_bagging <- function(i, model, data){
  n <- nrow(data)
  current_tree <- model$mtrees[[i]]$btree
  ind_x_current <- model$mtrees[[i]]$bindx
  oob_pred <- predict(current_tree, newdata = data[-ind_x_current,])
  as_tibble(oob_pred) %>%
    mutate(ind_x = seq(1,n)[-ind_x_current], tree = i)
}

oob_bagging <-
  map_df(1:length(mod_bag$mtrees),
         ~compute_oob_bagging(.,mod_bag, df_train))
oob_bagging
```

```
## # A tibble: 497,933 x 3
```

```
##     value ind_x  tree
##     <dbl> <int> <int>
##   1 390.      10     1
##   2 439.      14     1
##   3 405       17     1
##   4 412.      18     1
##   5 418       20     1
##   6 342.      24     1
##   7 277       25     1
##   8 248.      27     1
##   9  74.9     31     1
## 10 306.      35     1
## # ... with 497,923 more rows
```

Based on those out-of-bags predictions, we can compute the out-of-bag overall MSE depending on how many trees are considered:

```
#' Computes out-of-bag MSE using out-of-bag predictions
#' from `nb_tree` trees
compute_mse_oob_ntree <- function(nb_tree){

  oob_pred <-
    oob_bagging %>%
    filter(tree <= nb_tree) %>%
    group_by(ind_x) %>%
    summarise(oob_pred = mean(value))

  df_train %>%
    mutate(ind_x = row_number()) %>%
    left_join(
      oob_pred,
       by = "ind_x"
    ) %>%
    mutate(error_sq = (rented_bike_count-oob_pred)^2) %>%
    summarise(mse = mean(error_sq, na.rm=TRUE)) %>%
    mutate(nb_tree = nb_tree)

}

df_plot_mse_oob_bagging <-
  map_df(1:length(mod_bag$mtrees), compute_mse_oob_ntree)
```

```
df_plot_mse_oob_bagging
```

```
## # A tibble: 200 x 2
##       mse nb_tree
##     <dbl>   <int>
##  1 51984.       1
##  2 53239.       2
##  3 47821.       3
##  4 46944.       4
##  5 44856.       5
##  6 43811.       6
##  7 43563.       7
##  8 42060.       8
##  9 39683.       9
## 10 39366.      10
## # ... with 190 more rows
```

Then, let us finally compute the MSE on a test set. Let us define a helper function that computed the predictions on the test set for the ith tree grown.

```
#' Computes predictions for the ith tree
#' (helper function)
compute_pred_tree <- function(i, model, newdata){
  current_tree <- model$mtrees[[i]]$btree
  pred <- predict(current_tree, newdata = newdata)
  as_tibble(pred, rownames = "ind_x") %>%
    mutate(tree = i)
}

pred_test_bagging <-
  map_df(1:length(mod_bag$mtrees),
         ~compute_pred_tree(., mod_bag, df_test))
```

The MSE on the test set:

```
#' Compute the MSE on a test set (`df_test`)
#' @param nb_tree number of trees used to compute the MSE
compute_mse_bagging_ntree <- function(nb_tree){

  pred <-
```

```
    pred_test_bagging %>%
    mutate(ind_x = as.numeric(ind_x)) %>%
    filter(tree <= nb_tree) %>%
    group_by(ind_x) %>%
    summarise(oob_pred = mean(value))

  df_train %>%
    mutate(ind_x = row_number()) %>%
    left_join(
      pred,
      by = "ind_x"
    ) %>%
    mutate(error_sq = (rented_bike_count-oob_pred)^2) %>%
    summarise(mse = mean(error_sq, na.rm=TRUE)) %>%
    mutate(nb_tree = nb_tree)

}
df_plot_mse_test_bagging <-
  map_df(1:length(mod_bag$mtrees), compute_mse_bagging_ntree)
df_plot_mse_test_bagging
```

```
## # A tibble: 200 x 2
##         mse nb_tree
##       <dbl>   <int>
##   1 633941.        1
##   2 594158.        2
##   3 580209.        3
##   4 595718.        4
##   5 601718.        5
##   6 606807.        6
##   7 602178.        7
##   8 599957.        8
##   9 605989.        9
##  10 613597.       10
## # ... with 190 more rows
```

We now have:

- `df_plot_mse_oob_bagging`: out-of-bag MSE for bagging
- `df_plot_mse_test_bagging`: MSE computed on the test set for bagging
- `df_plot_mse_oob_rf`: out-of-bag MSE for random forest

- `df_plot_mse_test_rf`: MSE computed on the test set for random forest

Let us plot those errors:

```
df_plot <-
  df_plot_mse_oob_bagging %>%
  mutate(type = "OOB", model = "Bagging") %>%
  bind_rows(
    df_plot_mse_test_bagging %>%
      mutate(type = "Test", model = "Bagging")
  ) %>%
  bind_rows(
    df_plot_mse_oob_rf %>%
      mutate(type = "OOB", model = "Random Forests")
  ) %>%
  bind_rows(
    df_plot_mse_test_rf %>%
      mutate(type = "Test", model = "Random Forests")
  )

ggplot(data = df_plot,
       mapping = aes(x = nb_tree, y = mse)) +
  geom_line(mapping = aes(colour = model)) +
  labs(x = "Number of trees", y = "Error (MSE)") +
  theme(plot.title.position = "plot") +
  facet_wrap(~type, scales = "free_y", ncol = 2)
```

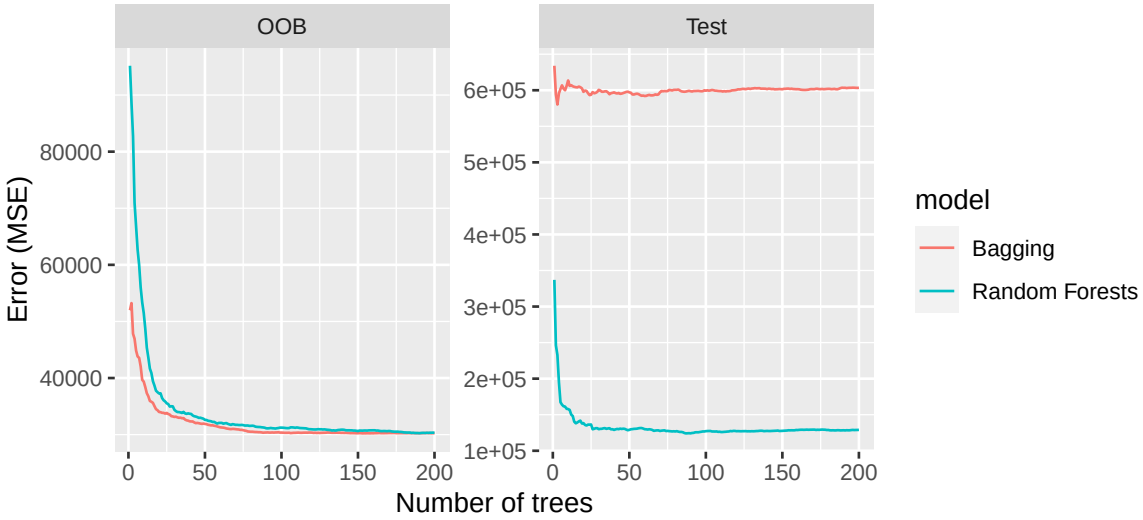

Figure 4.36: Goodness of fit depending on the number of trees.

We note that the OOB error are lower than the errors computed on the test set.
The errors obtained with a Random Forest are usually lower than their counterpart obtained with bagging (which is not the case here!).

4.4.2.2 Varying the hyperparameters

We have just seen in the last graph that the error of the model seem to stabilize after a certain number of trees. If the number of trees is too large, it causes no overfitting, it just takes longer to estimate the model.

4.4.2.2.1 Varying `mtry` Let us make the hyperparameter `mtry` that controls the number of variables sampled from the predictors to perform a split. We will consider values ranging from 1 to 5. In each case, we will grow a random forest, and then look at the out-of-bag error depending on the number of trees grown.

```
library(doSNOW)
ncl <- parallel::detectCores()-1
(cl <- makeCluster(ncl))
```

```
## groupe de processus socket avec 7 noeuds sur l'hôte 'localhost'
```

```
registerDoSNOW(cl)

# With a progress bar
# pb <- txtProgressBar(min=1, max = 5, style = 3)
# progress <- function(n) setTxtProgressBar(pb, n)
# opts <- list(progress = progress)

nb_predictors <- ncol(df_train %>% select(-y_binary, -date))

mod_rf_mtry <- foreach(
  mtry=1:5,
  .packages=c("randomForest", "dplyr")
  # .options.snow = opts) %dopar% { # With probressbar
  ) %dopar% {
    randomForest(
      formula = rented_bike_count ~ .,
      data = df_train %>% select(-y_binary, -date),
      ntree = 200,
      mtry = mtry,
      nodesize = 5,
      maxnodes = NULL
    )
  }
# close(pb)
stopCluster(cl)
```

From the results, we can extract the out-of-bag MSE:

```
df_plot_mtry <-
  map(mod_rf_mtry, "mse") %>%
  map_df(~tibble(mse=.) %>%
           mutate(nb_tree = row_number()), .id = "mtry")
df_plot_mtry
```

```
## # A tibble: 1,000 x 3
##    mtry      mse nb_tree
##    <chr>   <dbl>   <int>
##  1 1     234790.       1
##  2 1     198114.       2
##  3 1     170895.       3
##  4 1     181261.       4
```

```
##   5 1      175626.        5
##   6 1      177187.        6
##   7 1      167673.        7
##   8 1      167372.        8
##   9 1      169075.        9
##  10 1      168488.       10
## # ... with 990 more rows
```

And then we can plot the results:

```
ggplot(data = df_plot_mtry,
       mapping = aes(x = nb_tree, y = mse)) +
  geom_line(mapping = aes(colour = mtry)) +
  labs(x = "Number of trees", y = "Error (MSE)")
```

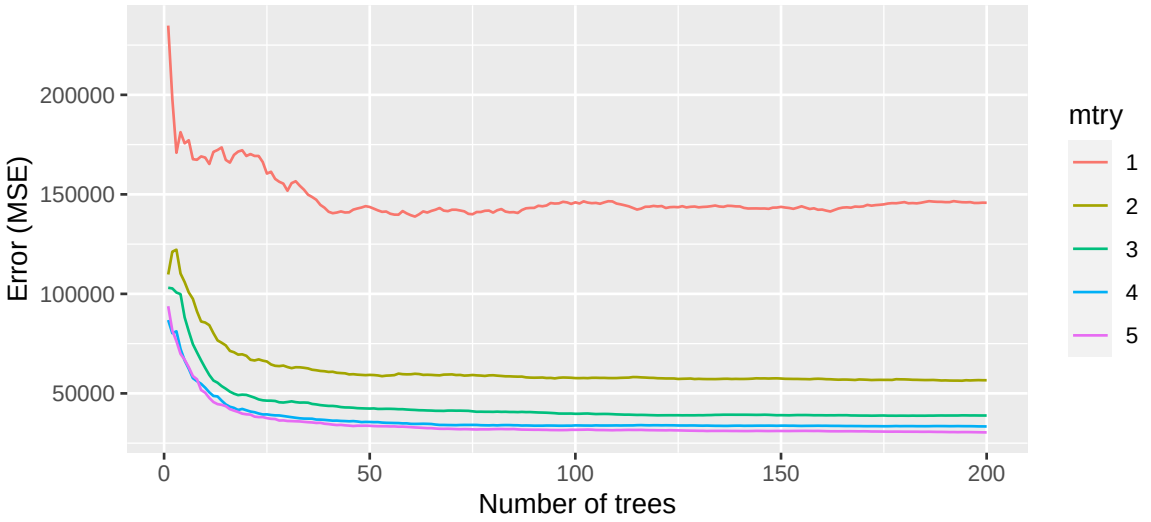

Figure 4.37: Varying the number of variables samples to perform a split.

Recall that taking only a random sample of m variables among the p available to perform a split allows us to decorrelate the trees. Boehmke and Greenwell (2019) notes that in the presence of a lot of noise in the data, when there are few predictors, high values for `mtry` allow the selection of variables with a stronger signal. On the other hand, when there are many relevant predictors, low values of `mtry` offer better performance. In practice, it is therefore necessary to test multiple values in order to select the one that offers the most interesting results in terms of error reduction.

We will see in a subsequent section how to create a **grid search** that allows to **fine tune** the model. In other words, we will create a matrix that will contain combinations of different values for the hyperparameters of the model (a grid) and we will iterate over that grid to try each possible combination contained in it.

4.4.2.2.2 Varying `nodesize` Let us now consider varying the minimum size of terminal nodes. Recall that increasing the numner of nodesize leads to smaller trees (which thus fastens the computation).

```
library(doSNOW)
ncl <- parallel::detectCores()-1
(cl <- makeCluster(ncl))
```

```
## groupe de processus socket avec 7 noeuds sur l'hôte 'localhost'
```

```
registerDoSNOW(cl)

# pb <- txtProgressBar(min=1, max = 4, style = 3)
# progress <- function(n) setTxtProgressBar(pb, n)
# opts <- list(progress = progress)

nb_predictors <- ncol(df_train %>% select(-y_binary, -date))


mod_rf_nodesize <- foreach(
  nodesize = c(5, 10, 15, 20),
  .packages=c("randomForest", "dplyr")
  # .options.snow = opts # progressbar
  ) %dopar% {
    randomForest(
```

```
      formula = rented_bike_count ~ .,
      data = df_train %>% select(-y_binary, -date),
      ntree = 200,
      mtry = round(nb_predictors/3),
      nodesize = nodesize,
      maxnodes = NULL
    )
  }
# close(pb)
stopCluster(cl)
```

The out-of-bag errors can once again be extracted from the different models:

```
df_plot_nodesize <-
  map(mod_rf_nodesize, "mse") %>%
  map_df(~tibble(mse=.) %>% mutate(nb_tree = row_number()), .id = "ind_nodesize") %>%
  mutate(ind_nodesize = as.numeric(ind_nodesize)) %>%
  left_join(
    tibble(nodesize = c(5, 10, 15, 20)) %>% mutate(ind_nodesize = row_number()),
    by = "ind_nodesize"
  )
```

```
df_plot_nodesize
```

```
## # A tibble: 800 x 4
##    ind_nodesize    mse nb_tree nodesize
##           <dbl>  <dbl>   <int>    <dbl>
##  1            1 83334.       1        5
##  2            1 83699.       2        5
##  3            1 79142.       3        5
##  4            1 71100.       4        5
##  5            1 67064.       5        5
##  6            1 62475.       6        5
##  7            1 58955.       7        5
##  8            1 55024.       8        5
##  9            1 54036.       9        5
## 10            1 50384.      10        5
## # ... with 790 more rows
```

And then we can plot the results:

```
ggplot(data = df_plot_nodesize,
         mapping = aes(x = nb_tree, y = mse)) +
  geom_line(mapping = aes(colour = factor(nodesize))) +
  labs(x = "Number of trees", y = "Error (MSE)") +
  scale_colour_discrete("nodesize")
```

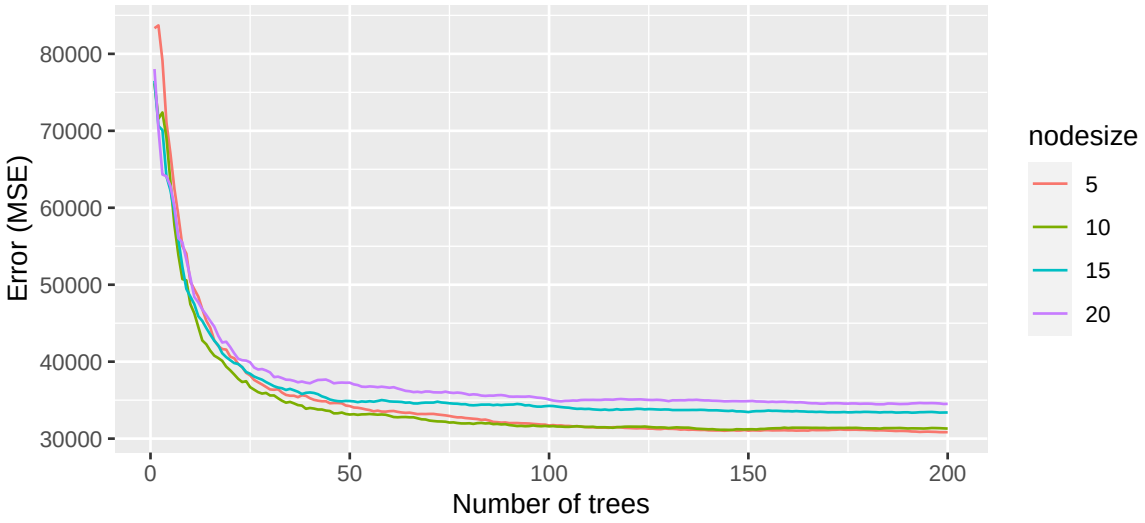

Figure 4.38: Varying the minimum size of terminal nodes.

Once again, we need to loop over different values to try to find the one that will produce the best results.

4.4.2.2.3 Varying `maxnodes`

Lastly, let us consider the argument `maxnodes` that controls the maximum number of terminal nodes trees in the forest can have. If the value is not provided, as was the case in all our other examples, trees are grown to the maximum possible (subject to limits by `nodesize`).

```
library(doSNOW)
ncl <- parallel::detectCores()-1
(cl <- makeCluster(ncl))
```

```
## groupe de processus socket avec 7 noeuds sur l'hôte 'localhost'
```

```
registerDoSNOW(cl)

# pb <- txtProgressBar(min=1, max = 6, style = 3)
# progress <- function(n) setTxtProgressBar(pb, n)
# opts <- list(progress = progress)

nb_predictors <- ncol(df_train %>% select(-y_binary, -date))

mod_rf_maxnodes <- foreach(
  max_nodes = c(5, 10, 20, 30, 40, 50),
  .packages=c("randomForest", "dplyr")
  # .options.snow = opts # with progressbar
  ) %dopar% {
    randomForest(
      formula = rented_bike_count ~ .,
      data = df_train %>% select(-y_binary, -date),
      ntree = 500,
      mtry = round(nb_predictors/3),
      nodesize = 5,
      maxnodes = max_nodes
    )
  }
# close(pb)
stopCluster(cl)
```

Let us extract the out-of-bag MSE:

```
df_plot_maxnodes <-
  map(mod_rf_maxnodes, "mse") %>%
  map_df(~tibble(mse=.) %>%
           mutate(nb_tree = row_number()), .id = "ind_maxnodes") %>%
  mutate(ind_maxnodes = as.numeric(ind_maxnodes)) %>%
  left_join(
    tibble(maxnodes = c(5, 10, 20, 30, 40, 50)) %>%
      mutate(ind_maxnodes = row_number()),
    by = "ind_maxnodes"
  )
df_plot_maxnodes
```

```
## # A tibble: 3,000 x 4
```

```
##    ind_maxnodes      mse nb_tree maxnodes
##           <dbl>    <dbl>   <int>    <dbl>
##  1            1 271863.        1        5
##  2            1 230301.        2        5
##  3            1 207807.        3        5
##  4            1 209513.        4        5
##  5            1 203316.        5        5
##  6            1 197045.        6        5
##  7            1 197162.        7        5
##  8            1 194583.        8        5
##  9            1 193653.        9        5
## 10            1 190162.       10        5
## # ... with 2,990 more rows
```

```
df_plot_maxnodes
```

```
## # A tibble: 3,000 x 4
##    ind_maxnodes      mse nb_tree maxnodes
##           <dbl>    <dbl>   <int>    <dbl>
##  1            1 271863.        1        5
##  2            1 230301.        2        5
##  3            1 207807.        3        5
##  4            1 209513.        4        5
##  5            1 203316.        5        5
##  6            1 197045.        6        5
##  7            1 197162.        7        5
##  8            1 194583.        8        5
##  9            1 193653.        9        5
## 10            1 190162.       10        5
## # ... with 2,990 more rows
```

And plot the results:

```
ggplot(data = df_plot_maxnodes,
       mapping = aes(x = nb_tree, y = mse)) +
  geom_line(mapping = aes(colour = factor(maxnodes))) +
  labs(x = "Number of trees", y = "Error (MSE)") +
  scale_colour_discrete("maxnodes")
```

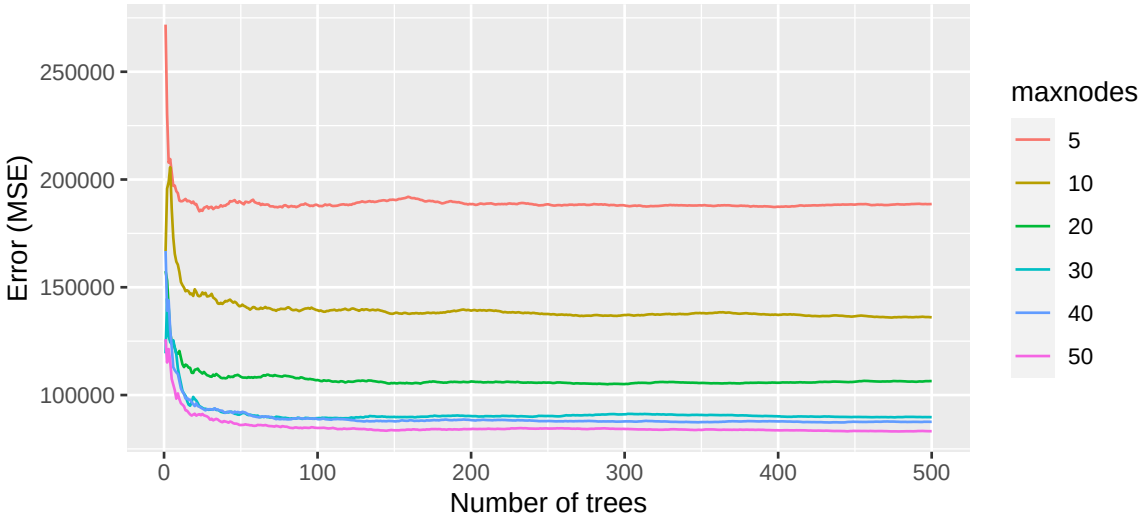

Figure 4.39: Varying the maximum number of terminal nodes trees.

When the number of maximum number of terminal nodes is restricted, it leads to shallower trees. Recall that using unpruned trees creates more sensible weak learners.

4.4.2.3 Variable importance

To get an idea of the importance of the variables, it is possible to rely on the same measures as before, namely the average total reduction in measurement error contributed by each variable. However, Terence Parr, Kerem Turgutlu, Christopher Csiszar, and Jeremy Howard warn us that this default method to compute variable importance is biased (see their notebook). Quoting them:

> We've known for years that this common mechanism for computing feature importance is biased; i.e. it tends to inflate the importance of continuous or high-cardinality categorical variables.

Another measure is widely used with random forests. It is based on the permutation of variables. The procedure is as follows:

- For tree $t \in \{1, \ldots, \text{ntree}\}$, using the out-of-bag sample, compute the goodness of fit (*e.g.*, the MSE for regression problems, or the accuracy for classification problems) : $r^{(t)}$
  - For each predictor $j \in \{1, \ldots, p\}$:
    * generate a new matrix of predictors by permuting predictor j in the matrix of predictors $\mathbf{x}$
    * compute the goodness of fit using the predictions made with the prermuted data: $r_j^{(t)}$
    * compute the permutation variable importance for the tree as either:
      · $\text{PI}_j^{(t)} = r_j^{(t)} - r^{(t)}$
      · $\text{PI}_j^{(t)} = \frac{r_j^{(t)}}{r^{(t)}}$.
- Once the tree-wise permutation importance is estimated for all trees t, for each predictor $j \in \{1, \ldots, p\}$:
  - compute the forest-wise permutation importance as the average over all tree-wise permutation importance:

$$\text{PI}_j = \frac{\sum_{t=1}^{\text{ntree}} \text{PI}_j^{(t)}}{\text{ntree}}.$$

The idea behind this procedure is as follows, as indicated in Debeer and Strobl (2020). If the response variable y and an explanatory variable $\mathbf{x}_j$ have a structural dependence, adding noise to the variable $\mathbf{x}_j$ should destroy this structural dependence. Once the noise is added, the splits made from $\mathbf{x}_j$ become random splits. The contribution of the variable to the decrease in the error criterion/impurity measure (MSE or Gini criterion) should therefore be diminished or even lost. Swapping the values of $\mathbf{x}_j$ is a way of noising this variable. Comparing the goodness of fit of the tree (and the random forest) before and after the noising allows us to quantify the importance of variable $\mathbf{x}_j$ in predicting the response y: if the permutation had no effect, the variable $\mathbf{x}_j$ does not matter. If, on the other hand, the permutation results in a decrease in the predictive capacity of the model, then the variable $\mathbf{x}_j$ is important for predicting y.

In a first place, we need an estimated random forest. As we will need it for the sake of the illustration, we will keep the out-of-bag errors in the resulting object, setting `keep.inbag = TRUE`. We will also specifically ask for the default importance metric to be return by setting `importance = TRUE`.{R}.

```
mod_rf <-
  randomForest(
    formula = rented_bike_count ~ .,
    data = df_train %>% select(-y_binary, -date),
```

```
    ntree = 100,
    mtry = round(ncol(select(df_train, -y_binary, -date))/3),
    nodesize = 5,
    maxnodes = NULL,
    keep.inbag = TRUE,
    importance = TRUE
  )
```

Let us define (one again) a function to compute the Mean Squarred Error (we face a regression problem here).

```
compute_mse <- function(observed, predicted){
  mean((observed-predicted)^2)
}
```

Let us illustrate how the method works with a single tree and a single noised-up predictor in a first place. We will extend the methodology to get an overall measure afterwards. Let us focus on the first weak learner:

```
tree_number <- 1
```

We want to compute the MSE based on the out-of-bag sample for that tree. To get the out-of-bag observations from the first tree, we can proceed as follows:

```
ind_x_oob <-
  mod_rf$inbag[, tree_number][mod_rf$inbag[, tree_number] == 0] %>%
  names() %>%
  as.numeric()

df_oob <- df_train[ind_x_oob, ]
```

Let us focus on the first predictor:

```
predictor_names <- attr(mod_rf$terms, "term.labels")
j <- 1
predictor_names[j]
```

```
## [1] "hour"
```

We can use the `predict()` function from {randomForest} as we did previously.

```
predictions_init <-
  predict(mod_rf, newdata = df_oob, predict.all = TRUE)
```

Then we can extract only the predicted values from the first tree, and compute the initial MSE, *i.e.*, the MSE computed without noising up the first predictor.

```
predictions_init_t <- predictions_init$individual[, tree_number]
mse_init_t <- compute_mse(df_oob$rented_bike_count, predictions_init_t)
mse_init_t
```

```
## [1] 78511.54
```

Then, using the `sample()` fuction, we can noise-up the first predictor.

```
df_oob_noised_j <- df_oob %>%
  mutate(!!sym(predictor_names[j]) :=
           sample(!!sym(predictor_names[j]), replace=FALSE))
```

We can have a glance at the first observations before and after the shuffling:

```
cbind(init = df_oob[[predictor_names[j]]],
      noised = df_oob_noised_j[[predictor_names[j]]]) %>%
  head()
```

```
##       init noised
## [1,]     0      8
## [2,]     1      9
## [3,]     4      1
## [4,]     9     20
## [5,]    10     22
## [6,]    21     21
```

The out-of-bag MSE with this noised-up data can then be computed:

```
predictions_xj <-
  predict(mod_rf, newdata = df_oob_noised_j, predict.all = TRUE)
predictions_noised_j_t <-
```

```
  predictions_xj$individual[, tree_number]
mse_noised_j_t <-
  compute_mse(df_oob_noised_j$rented_bike_count, predictions_noised_j_t)
mse_noised_j_t
```

```
## [1] 350665.2
```

```
cbind(mse_before = mse_init_t, mse_after = mse_noised_j_t)
```

```
##      mse_before mse_after
## [1,]   78511.54  350665.2
```

The MSE greatly rose. But we need to see if this happens with all the trees, and not just with the first one.

Let us define a function that recreate the out-of-bag data for a given tree:

```
get_oob_data <- function(tree_number, model){
  # Indices of out-of-bag observations
  ind_x_oob <-
    mod_rf$inbag[, tree_number][model$inbag[, tree_number] == 0] %>%
    names() %>%
    as.numeric()

  # oob data
  df_train[ind_x_oob, ]
}
```

Now, we can wrap-up the previous code to expand the computation to all trees and all variables:

```
#' Computes tree-wise permutation importance
#' @param tree_number index of the t-th tree
#' @param model object of class randomForest
compute_pi_tree <- function(tree_number, model){

  # Out-of-bag data for tree `tree_number`
  df_oob <- get_oob_data(tree_number, model)
```

```
  # tree-wise oob MSE before noising-up predictors
  predictions_init <- predict(model, newdata = df_oob, predict.all = TRUE)
  predictions_init_t <- predictions_init$individual[, tree_number]
  mse_init_t <- compute_mse(df_oob$rented_bike_count, predictions_init_t)

  predictor_names <- attr(mod_rf$terms, "term.labels")

  # Looping over indices of predictors
  mse_noised_t <- rep(NA, length(predictor_names))
  for(j in seq_along(predictor_names)){
    # Noising-up data
    df_oob_noised_j <- df_oob %>%
      mutate(!!sym(predictor_names[j]) := sample(
        !!sym(predictor_names[j]),
        replace=FALSE))

    # tree-wise oob MSE after noising-up jth predictor
    predictions_xj <-
      predict(model, newdata = df_oob_noised_j, predict.all = TRUE)
    predictions_noised_j_t <-
      predictions_xj$individual[, tree_number]
    mse_noised_j_t <-
      compute_mse(df_oob_noised_j$rented_bike_count, predictions_noised_j_t)
    mse_noised_t[[j]] <- mse_noised_j_t
  }

  # tree-wise permutation importance
  tree_wise_pi <- mse_noised_t-mse_init_t
  names(tree_wise_pi) <- predictor_names
  tree_wise_pi
}
```

The tree-wise permutation importance of the variables computed over the first tree can be obtained as follows:

```
permutation_importance_t <- compute_pi_tree(tree_number = 1, model = mod_rf)
permutation_importance_t
```

```
##                hour         temperature            humidity
##        2.766762e+05        2.947449e+05        5.654866e+04
```

```
##            wind_speed              visibility dew_point_temperature
##          7.344090e+03            5.761622e+03          1.604150e+04
##       solar_radiation                rainfall              snowfall
##          1.111862e+05            2.516485e+04          5.721013e+02
##               seasons                 holiday                    year
##          1.649556e+04            1.017557e+03         -8.320239e-02
##                 month                week_day
##          2.768299e+04            4.095305e+04
```

The tree-wise permutation importance values for all trees can then be obtained by looping over the trees:

```
tree_wise_pi <-
  map(1:mod_rf$ntree, ~compute_pi_tree(., model = mod_rf))
```

The results can be put in a table:

```
tree_wise_pi_df <-
  tree_wise_pi %>%
  map_df(~as_tibble(., rownames = "variable"), .id="tree")
tree_wise_pi_df
```

```
## # A tibble: 1,400 x 3
##    tree  variable                value
##    <chr> <chr>                   <dbl>
##  1 1     hour                  281244.
##  2 1     temperature           286672.
##  3 1     humidity               62440.
##  4 1     wind_speed              4406.
##  5 1     visibility              4968.
##  6 1     dew_point_temperature  14472.
##  7 1     solar_radiation       126512.
##  8 1     rainfall               28872.
##  9 1     snowfall                 739.
## 10 1     seasons                15761.
## # ... with 1,390 more rows
```

Lastly, the forest-wise permutation importance of each variable can be calculated:

```
permutation_importance <-
  tree_wise_pi_df %>%
  group_by(variable) %>%
  summarise(permutation_importance = mean(value)) %>%
  arrange(desc(permutation_importance))
```

These can be visualised over a barplot:

```
ggplot(data = permutation_importance,
       mapping = aes(x = permutation_importance,
                     y = fct_reorder(variable, permutation_importance))) +
  geom_bar(stat = "identity") +
  labs(x = "Permutation importance", y = NULL)
```

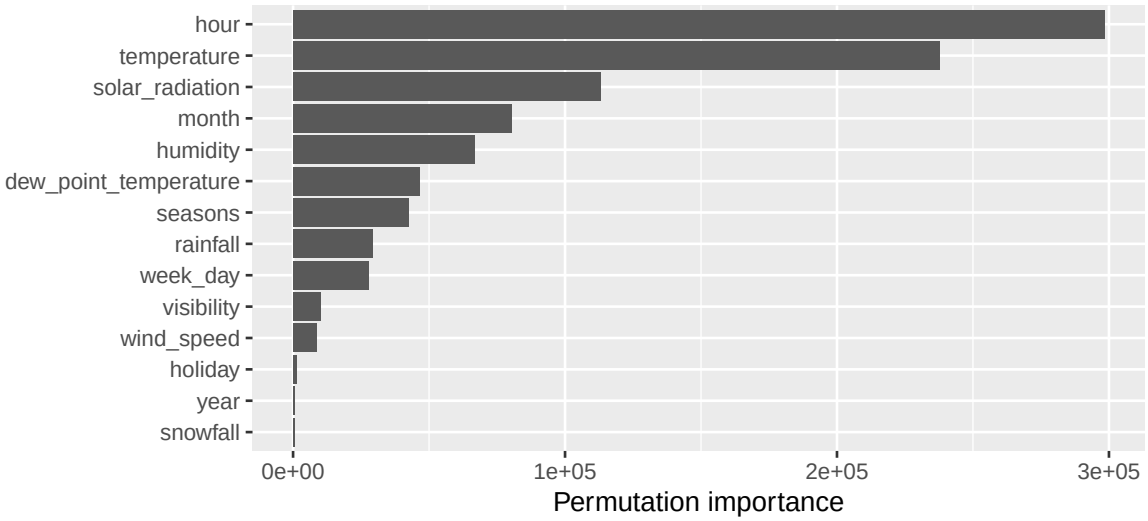

Figure 4.40: Tree-wise permutation importance.

For comparison, we can have a loop at the variable importance as computed with the standard method:

```
ggplot(data = as_tibble(mod_rf$importance, rownames = "variable"),
       mapping = aes(x = IncNodePurity,
                     y = fct_reorder(variable, IncNodePurity))) +
  geom_bar(stat = "identity") +
```

```
  labs(x = "Variable importance (Increase in node Purity)", y = NULL)
```

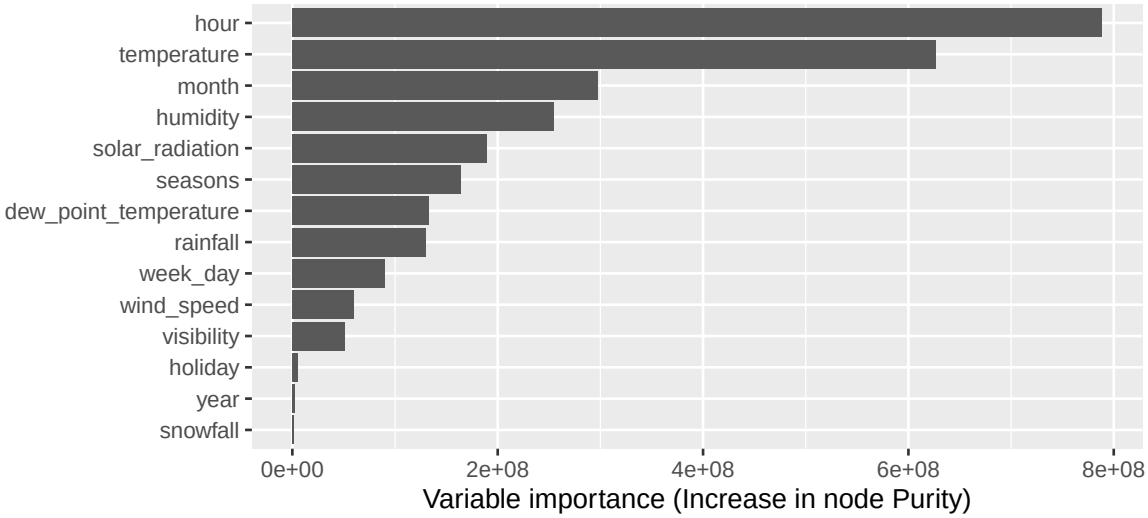

Figure 4.41: Variable importance: default method computed using the increase in node purity.

We note a few changes.

Obviously, to obtain the forest-wise permutation importance, there already exists a pre-build function in {randomForest}: `importance()`.

```
# Permutation importance
imp <- importance(mod_rf, type=1, scale = F)

ggplot(data = as_tibble(imp, rownames = "variable"),
       mapping = aes(x = `%IncMSE`,
                     y = fct_reorder(variable, `%IncMSE`))) +
  geom_bar(stat = "identity") +
  labs(x = "Permutation importance (% Increase in MSE)", y = NULL)
```

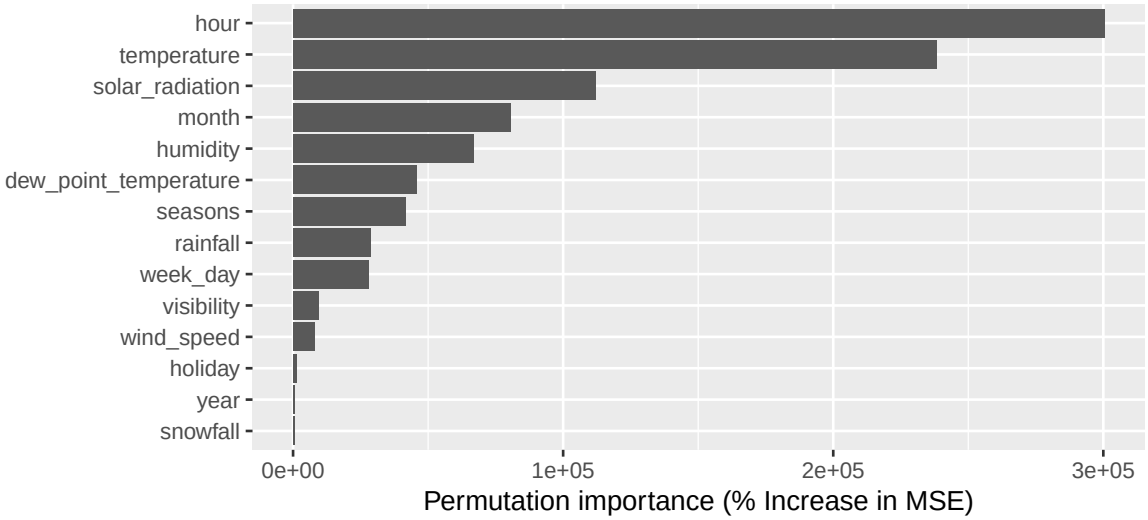

Figure 4.42: Permutation importance returned by randomForest: percent increase in MSE.

4.4.2.4 Fine Tuning

We have seen that some hyperparameters of the random forest can be changed. Tweaking those hyperparameters may affect greatly the predictive capacities of the model.

We have seen that when we increase the number of trees to be grown, the MSE stabilises. Having too many trees does not cause overfitting. Boehmke and Greenwell (2019) suggests the following rule of thumb to pick a first value for the number of trees to be grown: 10 times the number of predictors.

In our Seoul bikes dataset, the number of predictors is:

```
number_predictors <-
  df_train %>% select(-y_binary, -date) %>% ncol()
number_predictors
```

```
## [1] 15
```

Let us grow some random forests by exploring different values for:

- `mtry`: number of variables randomly sampled as candidates at each split
- `nodesize`: minimum size of terminal nodes
- `maxnodes`: maximum number of terminal nodes trees in the forest can have.

We make the grid here short on purpose: this is just an example, we want to avoid too long computing times.

```
hyperparameters_grid <-
  expand_grid(
    ntree = c(number_predictors*10, number_predictors*20),
    mtry = c(3,5),
    nodesize = c(3, 5, 10),
    maxnodes = c(-1, 10, 50, 100, 200)
  )
hyperparameters_grid
```

```
## # A tibble: 60 x 4
##    ntree  mtry nodesize maxnodes
##    <dbl> <dbl>    <dbl>    <dbl>
##  1   150     3        3       -1
##  2   150     3        3       10
##  3   150     3        3       50
##  4   150     3        3      100
##  5   150     3        3      200
##  6   150     3        5       -1
##  7   150     3        5       10
##  8   150     3        5       50
##  9   150     3        5      100
## 10   150     3        5      200
## # ... with 50 more rows
```

```
library(doSNOW)
ncl <- parallel::detectCores()-1
(cl <- makeCluster(ncl))
```

```
## groupe de processus socket avec 7 noeuds sur l'hôte 'localhost'
```

```
registerDoSNOW(cl)

# Progress bar
# pb <- txtProgressBar(min=1, max = nrow(hyperparameters_grid), style = 3)
# progress <- function(n) setTxtProgressBar(pb, n)
# opts <- list(progress = progress)
```

```
mse_search <- foreach(
  i=1:nrow(
    hyperparameters_grid),
  .packages=c("randomForest", "dplyr")
  # .options.snow = opts # with progressbar
  ) %dopar% {

    current_hyper <- hyperparameters_grid[i,]

    mod_rf_tmp <-
      randomForest(
        formula = rented_bike_count ~ .,
        data = df_train %>% select(-y_binary, -date),
        ntree = current_hyper$ntree,
        mtry = current_hyper$mtry,
        nodesize = current_hyper$nodesize,
        maxnodes = current_hyper$maxnodes
      )

    current_hyper %>% mutate(mse = mean(mod_rf_tmp$mse))
  }
# close(pb)
stopCluster(cl)
```

Of the hyperparameters explored, those whose combination produces the best results in terms of out-of-bag overall MSE are those listed in the first row of the table below:

```
mse_search <-
  bind_rows(mse_search) %>%
  arrange(mse)
mse_search
```

```
## # A tibble: 60 x 5
##    ntree  mtry nodesize maxnodes    mse
##    <dbl> <dbl>    <dbl>    <dbl>  <dbl>
##  1   300     5        5      200 55951.
##  2   300     5       10      200 56288.
##  3   300     5        3      200 56675.
##  4   150     5       10      200 57414.
##  5   150     5        5      200 58106.
```

```
##  6   150     5        3      200 59403.
##  7   300     5       10      100 67833.
##  8   150     5       10      100 68076.
##  9   300     3       10      200 68533.
## 10   300     3        3      200 68976.
## # ... with 50 more rows
```

The "best" random forest can then be grown again:

```
best_hyper_param <- mse_search %>% slice(1)

mod_rf_best <-
  randomForest(
    formula = rented_bike_count ~ .,
    data = df_train %>% select(-y_binary, -date),
    ntree = best_hyper_param$ntree,
    mtry = best_hyper_param$mtry,
    nodesize = best_hyper_param$nodesize,
    maxnodes = best_hyper_param$maxnodes
  )
```

```
pred_rf_best <- predict(mod_rf_best, newdata = df_test)
mse_rf_best <-
  compute_mse(observed = df_test$rented_bike_count,
              predicted = pred_rf_best)

pred_bagging <- predict(mod_bag, newdata = df_test)
mse_bagging <-
  compute_mse(observed = df_test$rented_bike_count,
              predicted = pred_bagging)

mse_bagging
```

```
## [1] 118766.9
```

```
mse_rf_best
```

```
## [1] 138780.4
```

Chapter 5

Support Vector Machines

This chapter presents another type of classifiers known as **support vector machines** (SVM). It corresponds to the 9th chapter of James et al. (2021).

We will cover:

- The **maximal margin classifier**, which requires the classes of the response variable to be separable by a **linear boundary**.
- The **support vector classifier** which is a generalization of the maximal margin classifier and allows an boundary that does not perfectly separate the classes.
- The **support vector machines** which further allow for non-linear boundaries.

5.1 Maximal Margin Classifier

5.1.1 Hyperplane

We will use the notion of a **separating hyperplane** in what follows. Hence, a little detour on recalling what a hyperplane is seems fairly reasonable.

In a p-dimensional space, a **hyperplane** is a flat affine subspace of dimension $p - 1$ (a subspace whose dimension is one less than that of its ambiant space).

Let us consider an example, in two dimensions, where a hyperplane is simply a line.

```
library(tidyverse)
x <- seq(0,1, by = .1)
y <- 2*x+1
df <- tibble(x = x, y = y)
ggplot(data = df, aes(x = x, y = y)) +
```

```
  geom_line()+
  theme(text = element_text(size=6), legend.position = "right") +
  xlab(expression(x[1])) + ylab(expression(x[2]))
```

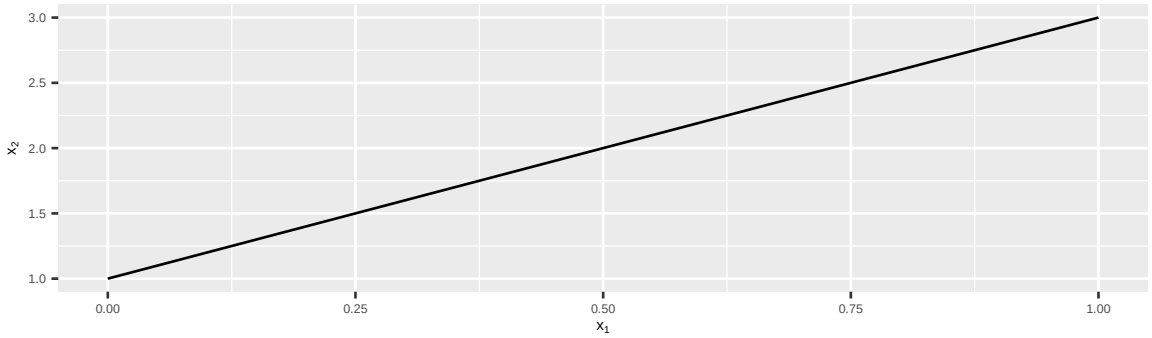

Figure 5.1: In two dimensions ($p = 2$), a hyperplane is a one-dimensional subspace, a line.

In three dimension, a hyperplane is a flat two-dimensional subspace, *i.e.*, a plane.

```
x <- seq(-10, 10, length= 30)
y <- x
f <- function(x, y){
  2*x+4*y
}
z <- outer(x, y, f)
z[is.na(z)] <- 1
persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue",
      ltheta = 120, shade = 0.15, ticktype = "detailed",
      xlab = "x_1", ylab = "x_2", zlab = "x_3=2x+4y"
)
```

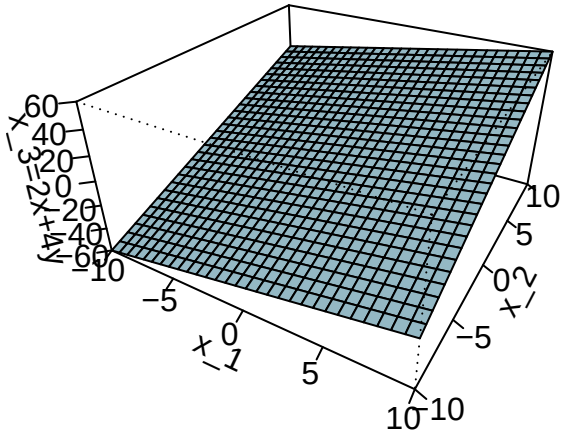

Figure 5.2: In three dimensions ($p = 3$), a hyperplane is a flat two-dimensional subspace ($p = 2$), a plane.

In two dimensions, for parameters β_0, β_1 and β_2, the following equation defines the hyperplane:

$$\beta_0 + \beta_1 \mathbf{x}_1 + \beta_2 \mathbf{x}_2 = 0 \tag{5.1.1}$$

Any point whose coordinates for which Eq. (5.1.1) holds is a point on the hyperplane.

In a p-dimension ambiant space, an hyperplace is defined for parameters β_0, β_1, ..., β_p by the following equation:

$$\beta_0 + \beta_1 \mathbf{x}_1 + \ldots + \beta_p \mathbf{x}_p = 0 \tag{5.1.2}$$

Any point whose coordinates are given in a vector of length p, *i.e.*, $\mathbf{X} = \begin{bmatrix} \mathbf{x_1} & \ldots & \mathbf{x}_p \end{bmatrix}^\top$ for which Eq. (5.1.2) holds is a point on the hyperplane.

If a point does not satisfy Eq. (5.1.2), then it lies either in one side or another side of the hyperplane, *i.e.*,

$$\begin{aligned} \beta_0 + \beta_1 \mathbf{x}_1 + \ldots + \beta_p \mathbf{x}_p > 0 \text{ or} \\ \beta_0 + \beta_1 \mathbf{x}_1 + \ldots + \beta_p \mathbf{x}_p < 0 \end{aligned} \tag{5.1.3}$$

Hence, the hyperplane can be viewed as a subspace that divides a p-dimensional space in two halves.

```
x_1 <- seq(-2,2, by = .1)
f <- function(x) (1*x+.1)*0.5
x_2 <- f(x_1)
grid <- expand.grid(x_1=seq(-1.05,1.05, by = .1),
                    x_2=seq(-1.05,1.05, by = .1)) %>%
  tbl_df() %>%
  mutate(colour = ifelse(x_1 + 0.1 - 2*x_2 > 0, "magenta", "blue"))
df <- tibble(x = x_1, y = x_2)

ggplot(data = df, aes(x = x_1, y = x_2)) +
  geom_polygon(data = tibble(x = c(-2, 2, 2), y = c(f(-2), f(2), f(-2))),
               aes(x=x, y = y), fill = "magenta", alpha = .2) +
  geom_polygon(data = tibble(x = c(-2, -2, 2), y = c(f(-2), f(2), f(2))),
               aes(x=x, y = y), fill = "blue", alpha = .2) +
  geom_point(data = grid, aes(x = x_1, y = x_2, colour = colour)) +
  geom_line() +
  theme(legend.position = "none") +
  coord_equal(xlim = c(0, 1), ylim = c(0,1)) +
  xlab(expression(x[1])) + ylab(expression(x[2])) +
  scale_colour_manual(
    NULL, values = c("magenta" = "#D41159", "blue"="#1A85FF"))
```

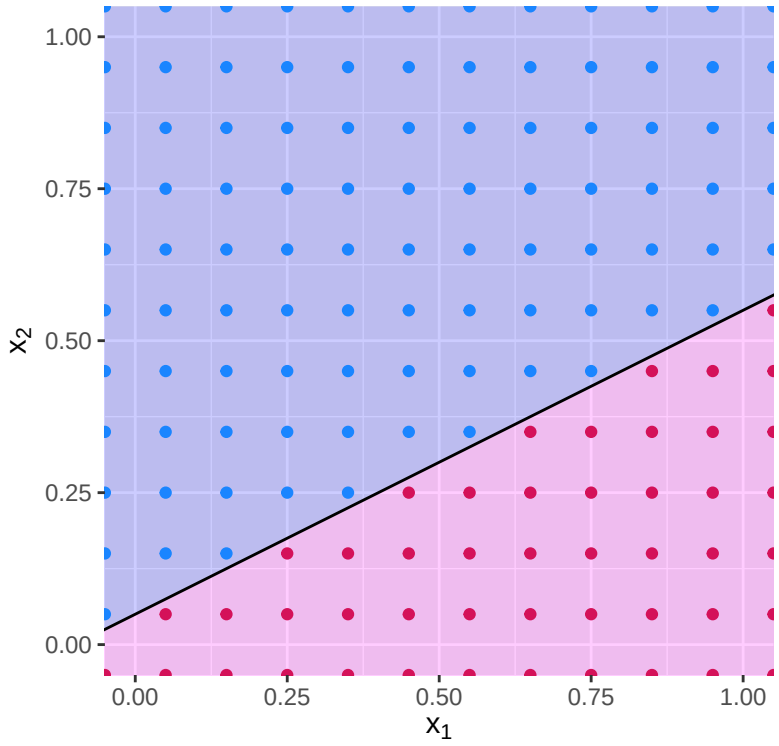

Figure 5.3: Hyperplane $x_1 - 2x_2 + 0.1$. The blue region corresponds to the set of points for which $x_1 - 2x_2 + 0.1 > 0$, the red region corresponds to the set of points for which $x_1 - 2x_2 + 0.1 < 0$.

Let us consider a simplified situation to begin with the **classification** problem.

Suppose that we have a set of n observations $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, where the response variable can take two values $\{\text{class } 1, \text{class } 2\}$, depending on the relationship with the p predictors.

In a first simplified example, let us assume that it is possible to construct a **separating hyperplane** that separates **perfectly** all observations.

In such a case, the hyperplane is such that:

$$\begin{cases} \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p > 0 & \text{if } y_i = \text{class } 1 \\ \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p < 0 & \text{if } y_i = \text{class } 2 \end{cases} \tag{5.1.4}$$

For convenience, as it is often the case in classification problem with a binary outcome, the response variable y can be coded as 1 and -1 (for class 1 and class 2, respectively). In that case,

the hyperplane has the property that, for all observations $i = 1, \ldots, n$:

$$y_i(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip}) > 0 \tag{5.1.5}$$

Let us illustrate this with simulated data in two dimensions, that are perfectly separable by a line.

```
df <-
  tibble(x_1 = c(5,7,9,12, 13,1,2,4,6,8),
         x_2 = c(1,1,4,7,6,3,1,5,9,7),
         y = c(rep(-1, 5), rep(1, 5))) %>%
  mutate(colour = ifelse(y == -1, yes ="blue", no = "magenta"))
```

The data can be visualised as follows:

```
ggplot() +
  geom_point(data = df,
             mapping = aes(x = x_1,
                           y = x_2, colour = colour)) +
  scale_colour_manual(
    NULL,
    values = c("blue" = "#1A85FF", "magenta" = "#D41159"))
```

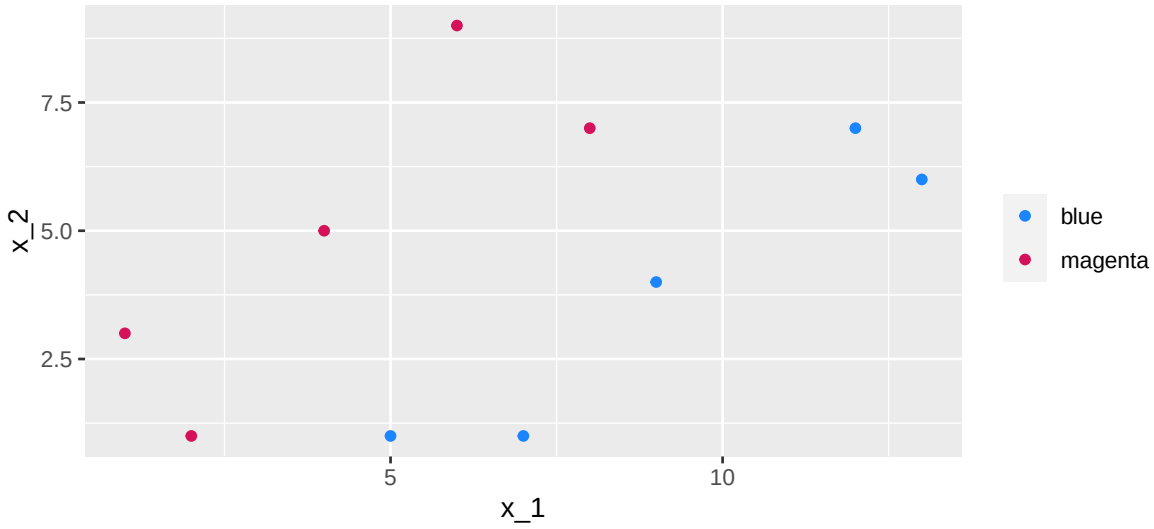

Figure 5.4: Data in two dimensions with two classes, where a separating line can perfectly separate the data.

There exists an infinite number of values for the slope and the intercept so that a line separates

the data into two distinct parts, one in which all the blue points will be grouped, and another one in which all the magenta points will be grouped. Some values can be picked, for example:

```
slope <- 1.4 ; intercept <- -5
```

This results in the following separating line:

```
ggplot() +
  geom_point(data = df,
             mapping = aes(x = x_1,
                           y = x_2, colour = colour)) +
  scale_colour_manual(
    NULL,
    values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
  geom_abline(slope = slope, intercept = intercept) +
  coord_equal(xlim = c(0, 15), ylim = c(0,10)) +
  theme(legend.position = "none")
```

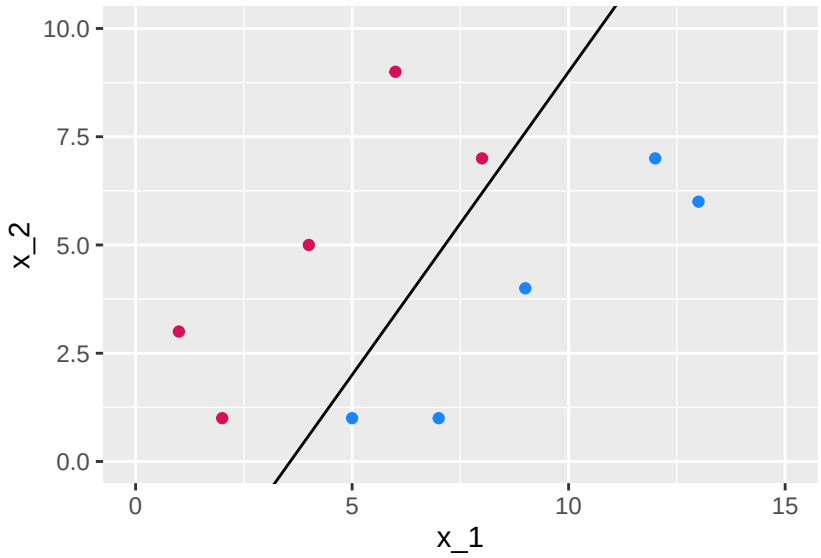

Figure 5.5: A first line of equation $x_2 = 1.4x_1 - 5$ that perfectly separates the data.

But other values might work just fine:

```
slope_2 <- 1 ; intercept_2 <- -2.5
```

```
ggplot() +
  geom_point(data = df,
             mapping = aes(x = x_1,
                           y = x_2, colour = colour)) +
  scale_colour_manual(
    NULL, values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
  geom_abline(slope = slope, intercept = intercept) +
  geom_abline(slope = slope_2, intercept = intercept_2, colour="#FE6100") +
  geom_label(
    data = tibble(x_1 =8, x_2=9, lab = "x_2 = 1.4x_1 - 5"),
    mapping = aes(x = x_1, y = x_2, label = lab), fill = "black",
    colour = "white") +
  geom_label(
    data = tibble(x_1 = 10, x_2=6, lab = "x_2 = x_1 - 2.5"),
    mapping = aes(x = x_1, y = x_2, label = lab), fill = "#FE6100",
    colour = "black") +
  coord_equal(xlim = c(0, 15), ylim = c(0,10))
```

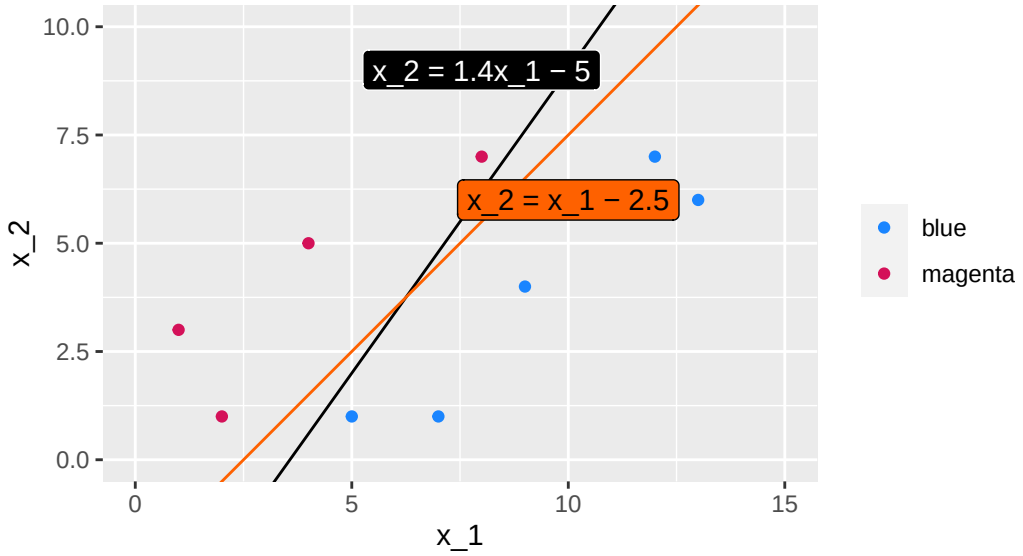

Figure 5.6: Two lines of equations $x_2 = 1.4x_1 - 5$ and $x_2 = x_1 - 2.5$ that perfectly separate the data.

The question is: how can values for the slope and intercept be found using an algorithm, in an optimal way?

5.1.2 Margin

As usual, we would like to decide among the possible values, what is the **optimal choice**, regarding some criterion.

A solution consists in computing the distance from each observation to a given separating hyperplane. The distance which is the smallest is called the **margin**. The objective is to select the separating hyperplane for which the **margin is the farthest from the observations**, *i.e.*, to select the **maximal margin hyperplane**.

This is known as **the maximal margin hyperplane**.

Let us visualise a margin, for the two different values of intercept and slope that were used earlier. To do so, the Euclidean distance of each point to the separating line needs to be computed.

```
df_tmp <-
  df %>%
  mutate(distance_margin =
           abs(slope*x_1 - x_2 + intercept) /
           sqrt(slope^2+(-1)^2))
df_tmp
```

```
## # A tibble: 10 x 5
##       x_1   x_2     y colour  distance_margin
##     <dbl> <dbl> <dbl> <chr>              <dbl>
##  1     5     1    -1 blue               0.581
##  2     7     1    -1 blue               2.21
##  3     9     4    -1 blue               2.09
##  4    12     7    -1 blue               2.79
##  5    13     6    -1 blue               4.18
##  6     1     3     1 magenta            3.84
##  7     2     1     1 magenta            1.86
##  8     4     5     1 magenta            2.56
##  9     6     9     1 magenta            3.25
## 10     8     7     1 magenta            0.465
```

Once these distances are calculated, the minimum needs to be kept: this define the margin, conditional on the separating line that was used.

```
min_margin <-
  df_tmp %>%
  arrange(distance_margin) %>%
  slice(1) %>%
  magrittr::extract2("distance_margin")
min_margin
```

```
## [1] 0.4649906
```

If we want to plot the margin, we need to further compute the intercept of the lines that define it.

```
dist_intercept <- min_margin/sin(pi/2-atan(slope))
```

The margin obtained with a slope of 1.4 and an intercept of -5 can be visualised this way:

```
ggplot() +
  geom_point(data = df,
             mapping = aes(x = x_1, y = x_2, colour=colour)) +
  scale_colour_manual(
    NULL, values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
  geom_abline(slope = slope, intercept = intercept, colour = "#FE6100") +
  coord_equal(xlim = c(0, 15), ylim = c(0,10)) +
  geom_abline(slope = slope, intercept = intercept-dist_intercept,
              colour = "black", linetype = "dashed") +
  geom_abline(slope = slope, intercept = intercept+dist_intercept,
              colour = "black", linetype = "dashed") +
  theme(legend.position = "none")
```

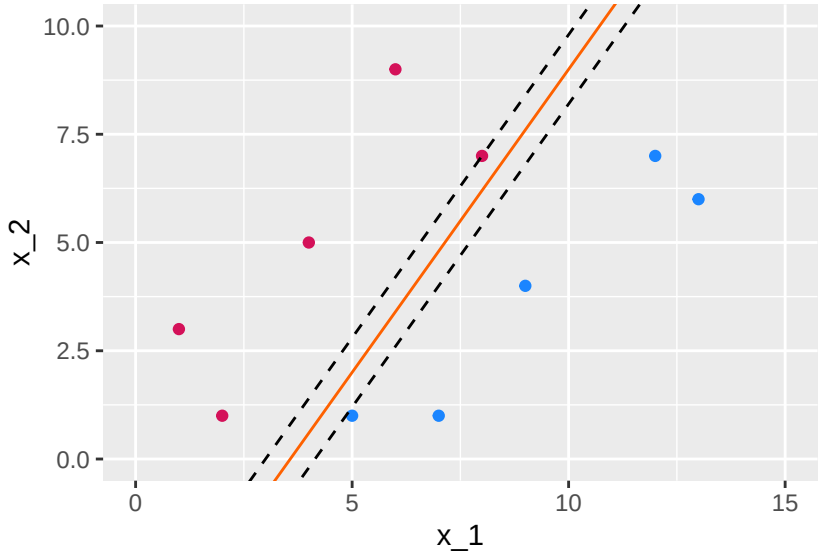

Figure 5.7: Margin obtained using the following separating hyperplane: $x_2 = 1.4x_1 - 5$.

Now, let us get the margin with a slope of 1 and an intercept of -2.5:

```
df_tmp <-
  df %>%
  mutate(distance_margin =
           abs(slope_2*x_1 - x_2 + intercept_2) /
           sqrt(slope_2^2+(-1)^2))
min_margin <-
  df_tmp %>%
  arrange(distance_margin) %>%
  slice(1) %>%
  magrittr::extract2("distance_margin")
min_margin
```

```
## [1] 1.06066
```

Let us look at the margin graphically:

```
dist_intercept <- min_margin/sin(pi/2-atan(slope_2))
ggplot() +
  geom_point(data = df,
```

```
            mapping = aes(x = x_1, y = x_2, colour=colour)) +
scale_colour_manual(
  NULL, values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
geom_abline(slope = slope_2, intercept = intercept_2, colour = "black") +
coord_equal(xlim = c(0, 15), ylim = c(0,10)) +
geom_abline(slope = slope_2, intercept = intercept_2-dist_intercept,
            colour = "black", linetype = "dashed") +
geom_abline(slope = slope_2, intercept = intercept_2+dist_intercept,
            colour = "black", linetype = "dashed") +
theme(legend.position = "none")
```

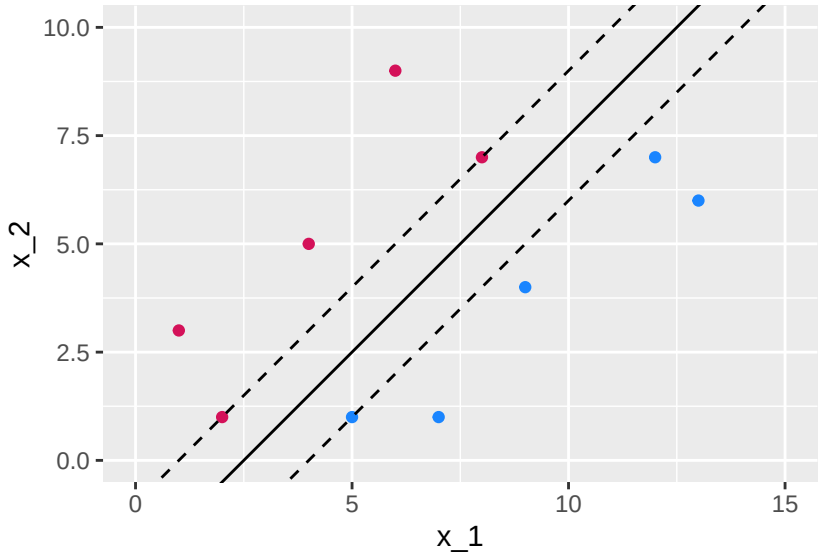

Figure 5.8: Margin obtained using the following separating hyperplane: $x_2 = x_1 - 2.5$.

The obtained margin is relatively higher in the second case. The idea is to try to find the maximal. This boils down to an optimization program.

The optimization problem is the following:

$$\max_{\beta_0,\beta_1,\ldots,\beta_p,M} M \tag{5.1.6}$$

$$\text{s.t.} \sum_{j=1}^{p} \beta_j^2 = 1 \tag{5.1.7}$$

$$y_i(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip}) \geq M, \forall i = 1, \ldots, n \tag{5.1.8}$$

Constraint (5.1.8) ensures that each obs. is on the correct side of the hyperplane.

Constraint (5.1.7) ensures that the perpendicular distance from the ith observation to the hyperplane is given by:

$$y_i(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip})$$

Together, constraints (5.1.7) and (5.1.8) impose that each observation is on the correct side of the hyperplane and at least a distance M from the hyperplane.

We will see how this optimization problem can be implemented in R in a moment. For now, let us admit that the maximum margin is the one obtained with the values 1 for the slope and -2.5 for the intercept.

```
ggplot() +
  geom_point(data = df,
             mapping = aes(x = x_1, y = x_2, colour=colour)) +
  geom_point(data =
               tibble(
                 x_1 = c(5,2,8),
                 x_2 = c(1,1,7),
                 colour = c("blue", "magenta", "magenta")),
             mapping = aes(x = x_1, y = x_2, fill = colour),
             colour="black", size = 4, shape = 21) +
  scale_colour_manual(
    NULL, values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
  scale_fill_manual(
    NULL,values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
  geom_abline(slope = slope_2, intercept = intercept_2, colour = "black") +
  geom_abline(slope = slope_2, intercept = intercept_2-dist_intercept,
              colour = "black", linetype = "dashed") +
  geom_abline(slope = slope_2, intercept = intercept_2+dist_intercept,
              colour = "black", linetype = "dashed") +
  coord_equal(xlim = c(0, 15), ylim = c(0,10)) +
  theme(legend.position = "none")
```

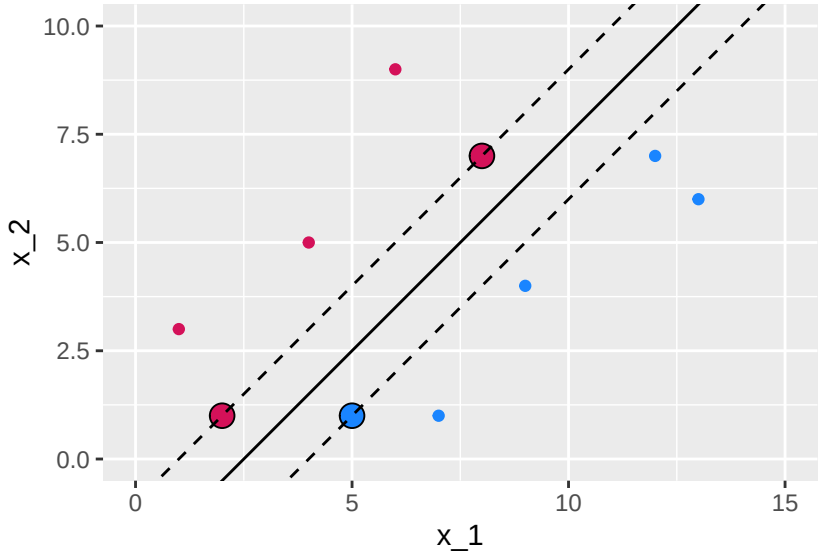

Figure 5.9: Maximum margin classifier for a perfectly separable binary outcome variable.

In the graph above, 3 observations from the training set that are equidistant from the maximal margin hyperplane were highlighted. These points are known as the **support vectors:**

- they are vectors in p-dimensional space
- they "support" the maximal margin hyperplane (if they move, the maximal margin hyperplane also moves).

For any other points, if they move but stay outside the boundary set by the margin, this does not affect the separating hyperplane. So the observations that fall in top of the fences are called **support vector** because they directly determine where the fences will be located.

In our example, the maximal margin hyperplane only depends on three points, but this is not a general result. The number of support vectors can vary according to the data.

Once the maximum margin is known, **classification** follows directly:

- cases that fall on one side of the maximal margin hyperplane are labelled as one class
- cases that fall on the other side of the maximal margin hyperplane are labelled as the other class.

The classification rule that follows from the decision boundary is known as **hard thresholding.**

So far, we were in a simplified situation in which it is possible to find a perfectly separable hyperplane. In reality, data are not always that cooperative, in that:

- there is no maximal margin classifier (the set of values are no longer linearly separable)
- the optimization problem gives no solution with $M > 0$.

In such cases, we can allow some number of observations to violate the rules so that they can lie on the wrong side of the margin boundaries. We can develop a hyperplane that *almost* separates the classes. The generalization of the maximal margin classifier to the non-separable case is known as the **support vector classifier**.

5.2 Support Vector Classifiers

Let us consider the case in which finding a maximal margin classifier is either **not possible or not desirable**.

A maximal margin classifier may not be desired as:

- its margins can be too narrow and therefore lead to relatively higher generalization errors
- the maximal margin hyperplane may be too sensitive to a change in a single observation.

As it is usually the case in statistical methods, a trade-off thus arises: here it consists in trading some accuracy in the classification for more robustness in the results.

Let us add some data (four observations) to the synthetic dataset at hand:

```
new_data <-
  tibble(x_1 = c(7, 10, 5, 8.5),
         x_2 = c(3.5, 9.5, 2.3, 3.5),
         y = c(1, -1, -1, 1),
         colour = c("magenta", "blue", "blue", "magenta"))

df_2 <-
  df %>%
  bind_rows(new_data) %>%
  mutate(class = factor(colour, levels = c("blue", "magenta")))
```

Let us plot the data on the previous graph.

```
ggplot() +
  geom_point(data = df_2,
             mapping = aes(x = x_1, y = x_2, colour=colour)) +
  geom_point(data = new_data,
             mapping = aes(x = x_1, y = x_2, fill = colour),
             colour="black", size = 4, shape = 21) +
  scale_colour_manual(
```

```
  NULL, values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
scale_fill_manual(
  NULL, values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
geom_abline(slope = slope_2, intercept = intercept_2, colour = "black") +
geom_abline(slope = slope_2, intercept = intercept_2-dist_intercept,
            colour = "black", linetype = "dashed") +
geom_abline(slope = slope_2, intercept = intercept_2+dist_intercept,
            colour = "black", linetype = "dashed") +
coord_equal(xlim = c(0, 15), ylim = c(0,10)) +
theme(legend.position = "none")
```

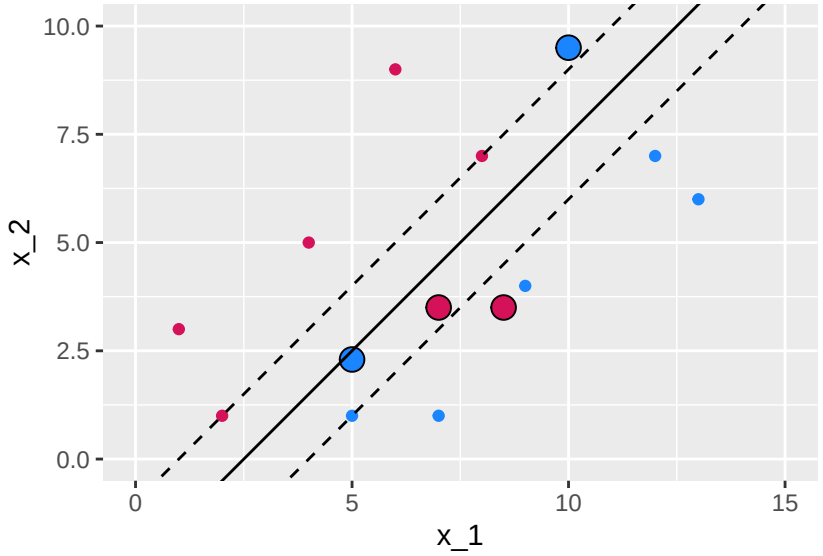

Figure 5.10: The new observations violate the previous margin.

As can be noted, the now observations that were added lie between the fences of the previous margin. In addition, the two magenta points lie in the **wrong side of the hyperplane.** One of the blue point (the one on top) is also on the wrong side of the hyperplane, while the second blue point (at the bottom) is on the **correct side of the hyperplane but on the wrong side of the margin.**

The **support vector classifier** will allow some points to violate the buffer zone. The optimization

problem becomes:

$$\max_{\beta_0,\beta_1,\ldots,\beta_p,\varepsilon_1,\ldots,\varepsilon_n,M} M \tag{5.2.1}$$

$$\text{s.t. } \sum_{j=1}^{p} \beta_j^2 = 1, \tag{5.2.2}$$

$$y_i(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip}) \geq M(1 - \varepsilon_t), \forall i = 1, \ldots, \tag{5.2.3}$$

$$n) \tag{5.2.4}$$

$$\varepsilon_i \geq 0, \sum_{i=1}^{n} \varepsilon_i \leq C, \tag{5.2.5}$$

where C is a nonnegative **tuning** parameter, M is the width of the margin.

$\varepsilon_1, \ldots, \varepsilon_n$ allow individual observations to lie on the wrong side of the margin or the hyperplane, they are called **slack variables** (more details on those are provided below).

Once the optimization problem is solved, the **classification** follows instantly by looking at **which side of the hyperplane** the observation lies. For a new observation x_0, the classification is based on the sign of $\beta_0 + \beta_1 x_0 + \ldots + \beta_p x_0$.

The slack variable ε_i indicates where the ith observation is located relative to both the hyperplane and the margin:

- $\varepsilon_i = 0$: the ith observation is on the correct side of the margin
- $\varepsilon_i > 0$: the ith observation is on the wrong side of the margin
- $\varepsilon_i > 1$: the ith observation is on the wrong side of the hyperplane

The tuning parameter C from Eq. (5.2.5) reflects a measure of **how permissive** we were when the margin was maximized, as it bounds the sum of ε_i. Setting a value of C to 0 implies that we do not allow any observations from the training sample to lie in the wrong side of the hyperplane. The optimization problem then boils down to that of the maximum margin (if the two classes are perfectly separable). If the value of C is greater than 0, then no more than C observations can be on the wrong side of the hyperplane:

- $\varepsilon_i > 0$ for each observation that lies on the wrong side of the hyperplane
- $\sum_{i=1}^{n} \varepsilon_i \leq C$

So, the idea of the support vector classifier can be viewed as maximizing the width of the buffer zone conditional on the slack variables. But the distance of some slack variables to the boundary can vary from one observation to another. The sum of these distances can then be viewed as a measure of how permissive we were when the margin was maximized.

- the **more permissive**, the larger the sum, the **higher the number of support vectors**, and the easier to locate a separating hyperplane within the margin
- but on the other hand, being more permissive can lead to a **higher bias** as we introduce **more misclassifications**.

Let us illustrate how the method works in R. We will use the function `svm()` from the package {e1071}.

It should be noted that the optimization problem that `svm()` addresses is written differently (see Karatzoglou, Meyer, and Hornik (2006)). The cost parameter of the function `svm()` allows us to control the penalty paid by the SVM for missclassifying a training point:

- a high cost value will be used to missclassify as few observations as possible
- a low cost (strictly greater than 0) will allow for more observations to be missclassified (the prediction function will be simpler).

With our initial data (perfectly separable date), let us set a high cost in the function, corresponding to penalise a lot missclassifications made with the training observations. We will therefore obtain a hard-margin. We will set `cost = 1000`. Here, we also set `kernel = "linear"` to use a linear kernel. More details on kernels will be provided in the next section, to consider non linear decision boundaries.

Before using the `svm()` function, let us create a factor variable in our initial dataset:

```
df <- df %>% mutate(class=factor(colour))
```

```
library(e1071)
svm_fit <- svm(
  class~x_1+x_2,
  data = df,
  cost = 1000,
  # Using a linear kernel
  kernel = "linear",
  # Not scaling the variables
  # Scaling: transforming y and X to get 0 mean and unit variance
  scale = FALSE)
svm_fit
```

```
##
## Call:
```

```
## svm(formula = class ~ x_1 + x_2, data = df, cost = 1000, kernel =
    "linear",
##     scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  1000
##
## Number of Support Vectors:  3
```

Only tree observations define the support vector. Their index can be accessed as follows:

```
svm_fit$index
```

```
## [1]  1  7 10
```

Which corresponds to those points:

```
df_support_vector <-
  tibble(df %>% slice(svm_fit$index))
df_support_vector
```

```
## # A tibble: 3 x 5
##     x_1   x_2     y colour  class
##   <dbl> <dbl> <dbl> <chr>   <fct>
## 1     5     1    -1 blue    blue
## 2     2     1     1 magenta magenta
## 3     8     7     1 magenta magenta
```

The negative intercept is in the element `rho` from the returned object. Hence, the coefficient β_0 can be computed this way:

```
beta_0 <- -svm_fit$rho
beta_0
```

```
## [1] -1.666317
```

The coefficient β_1,can be obtained as follows:

```
beta_1 <- sum(svm_fit$coefs * df$x_1[svm_fit$index])
beta_1
```

```
## [1] 0.6664919
```

And β_2:

```
beta_2 <- sum(svm_fit$coefs * df$x_2[svm_fit$index])
beta_2
```

```
## [1] -0.6664919
```

We can thus deduct the values for the intercept in our two dimensional case:

```
intercept <- -beta_0/beta_2
intercept
```

```
## [1] -2.500131
```

And the slode:

```
slope <- -beta_1/beta_2
slope
```

```
## [1] 1
```

```
ggplot() +
  geom_point(data = df,
             mapping = aes(x = x_1, y = x_2, colour=colour)) +
  geom_point(data = df_support_vector,
             mapping = aes(x = x_1, y = x_2, fill = colour),
             colour="black", size = 4, shape = 21) +
  scale_colour_manual(
```

```
    NULL, values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
  geom_abline(slope = slope, intercept = intercept, colour = "black") +
  coord_equal(xlim = c(0, 15), ylim = c(0,10)) +
  geom_abline(intercept = -beta_0/beta_2, slope = -beta_1/beta_2,
              colour = "black") +
  geom_abline(slope = slope, intercept = (-beta_0-1)/beta_2,
              colour = "black", linetype = "dashed") +
  geom_abline(slope = slope, intercept = (-beta_0+1)/beta_2,
              colour = "black", linetype = "dashed") +
  theme(legend.position = "none")
```

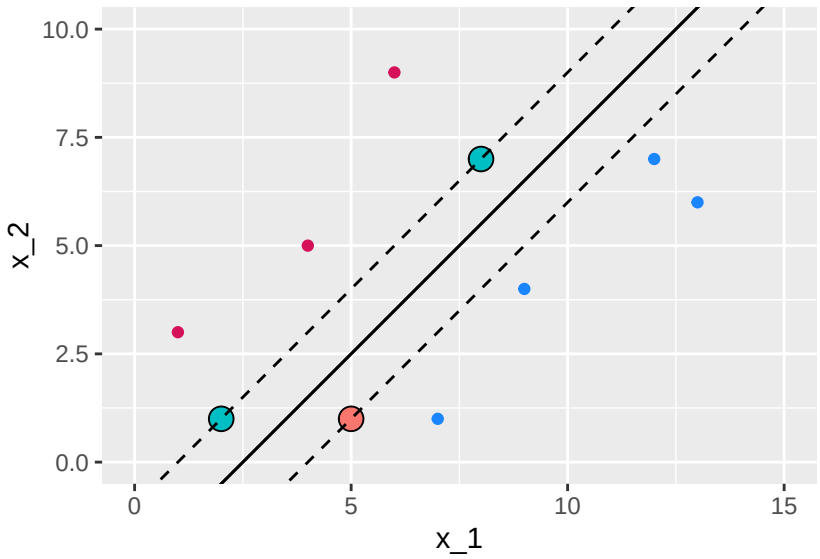

Figure 5.11: Maximal margin classifier for the initial dataset with linearly separable observation.

Now, let us simulate some new data that are not perfectly linearly separable. First, we begin with perfectly separable data:

```
set.seed(1234)
x_1 <- rbeta(n=5000, shape1 = 1, shape2 = 1)
x_2 <- rbeta(n=5000, shape1 = 1, shape2 = 1)
df <- tibble(x_1 = x_1, x_2 = x_2)

df <-
  df %>%
```

```
  mutate(
    flag = (0.5* x_1 + x_2 - 0.6 > 0 & 3*x_1 + x_2 - 1.5 < 0) |
      (3*x_1 + x_2 - 1.5 > 0 & 0.5* x_1 + x_2 - 0.6 < 0) |
      (3*x_1 + x_2 - 1.5 > 0 & 1.5* x_1 + x_2 - 1 < 0)) %>%
  filter(!flag) %>%
  mutate(
    class = ifelse(3*x_1 + x_2 - 1.5 > 0 & 1.5*x_1 + x_2 - 1 > 0 &
                     0.5*x_1 + x_2 - 0.6 > 0,
                   yes = "blue", no = "magenta")) %>%
  group_by(class) %>%
  sample_n(200) %>%
  ungroup()
```

And then, let us create some data that will be on the wrong side of the separating line.

```
df_new <-
  df %>%
  mutate(
    flag = (0.5* x_1 + x_2 - 0.6 > 0 & 3*x_1 + x_2 - 1.5 < 0) |
      (3*x_1 + x_2 - 1.5 > 0 & 0.5* x_1 + x_2 - 0.6 < 0) |
      (3*x_1 + x_2 - 1.5 > 0 & 1.5* x_1 + x_2 - 1 < 0)) %>%
  filter(!flag) %>%
  mutate(
    class = ifelse(3*x_1 + x_2 - 1.5 > 0 & 1.5*x_1 + x_2 - 1 > 0 &
                     0.5*x_1 + x_2 - 0.6 > 0,
                   yes = "magenta", no = "blue")) %>%
  group_by(class) %>%
  sample_n(20) %>%
  ungroup()
```

These two datasets can be merged:

```
df_2 <-
  df %>%
  bind_rows(df_new) %>%
  mutate(class = factor(class)) %>%
  sample_frac(1)
```

Let us have a look:

```
ggplot() +
  geom_point(data = df_2,
             mapping = aes(x = x_1, y = x_2, colour = class)) +
  scale_colour_manual(
    NULL,
    values = c("blue" = "#1A85FF", "magenta" = "#D41159"))
```

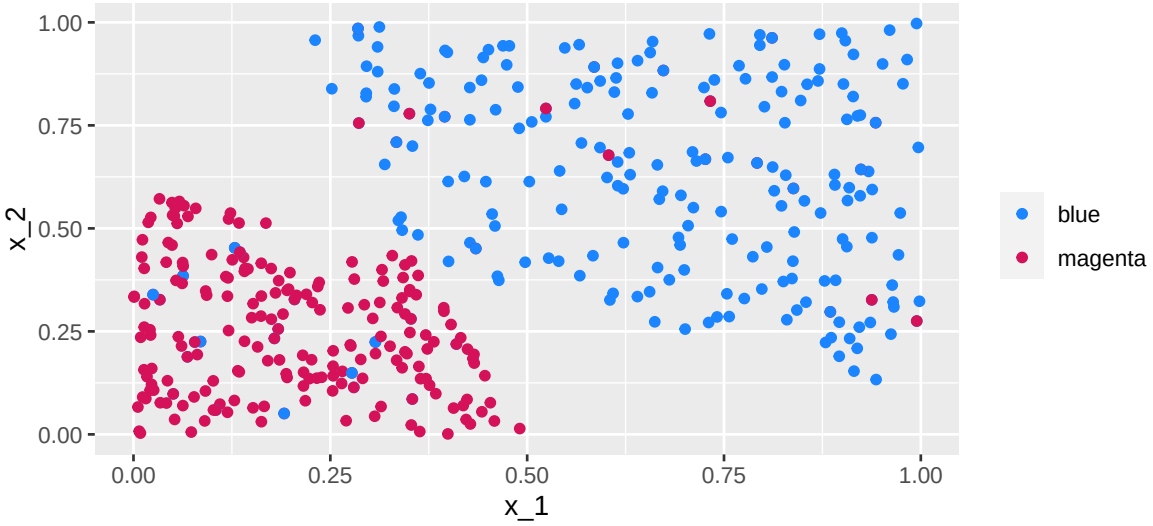

Figure 5.12: Simulated almost linearly-separable data.

Data are no longer linearly separable (but they are almost linearly separable), hence it will not be possible to obtain a hard-margin. The cost parameter can however be used to allow more or less missclassified training observations. Let us begin with a value of 100 :

```
svm_fit_2 <- svm(
  formula = class~x_1+x_2,
  data = df_2,
  cost = 100,
  # Using a linear kernel
  kernel = "linear",
  # Not scaling the variables
  # Scaling: transforming y and X to get 0 mean and unit variance
  scale = FALSE)
svm_fit_2
```

```
##
## Call:
## svm(formula = class ~ x_1 + x_2, data = df_2, cost = 100, kernel
   = "linear",
##     scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  100
##
## Number of Support Vectors:  142
```

There are 142 points that define the support vector. Let us visualise the soft-margin. First we need to compute some elements for the plot to be made. Let us make a function to make things more compact.

```
plot_results <- function(svm_result){
  df_support_vector <-
    tibble(df_2 %>% slice(svm_result$index))
  beta_0 <- -svm_result$rho
  beta_1 <- sum(svm_result$coefs * df_2$x_1[svm_result$index])
  beta_2 <- sum(svm_result$coefs * df_2$x_2[svm_result$index])
  intercept <- -beta_0/beta_2
  slope <- -beta_1/beta_2
  f <- function(x) x*slope+intercept
  ggplot() +
    geom_polygon(
      data = tibble(x = c(-2, 2, 2), y = c(f(-2), f(2), f(-2))),
      aes(x=x, y = y), fill = "blue", alpha = .2) +
    geom_polygon(
      data = tibble(x = c(-2, -2, 2), y = c(f(-2), f(2), f(2))),
      aes(x=x, y = y), fill = "magenta", alpha = .2) +
    geom_point(data = df_2, aes(x = x_1, y = x_2, colour=class)) +
    scale_colour_manual(
      NULL, values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
    scale_fill_manual(
      NULL, values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
    geom_abline(intercept = -beta_0/beta_2, slope = -beta_1/beta_2,
```

```
                colour = "black") +
    geom_abline(slope = slope, intercept = (-beta_0-1)/beta_2,
                colour = "black", linetype = "dashed") +
    geom_abline(slope = slope, intercept = (-beta_0+1)/beta_2,
                colour = "black", linetype = "dashed") +
    coord_equal(xlim = c(0, 1), ylim = c(0,1)) +
    theme(legend.position = "none")
}
```

The results can be plotted:

```
plot_results(svm_fit_2)
```

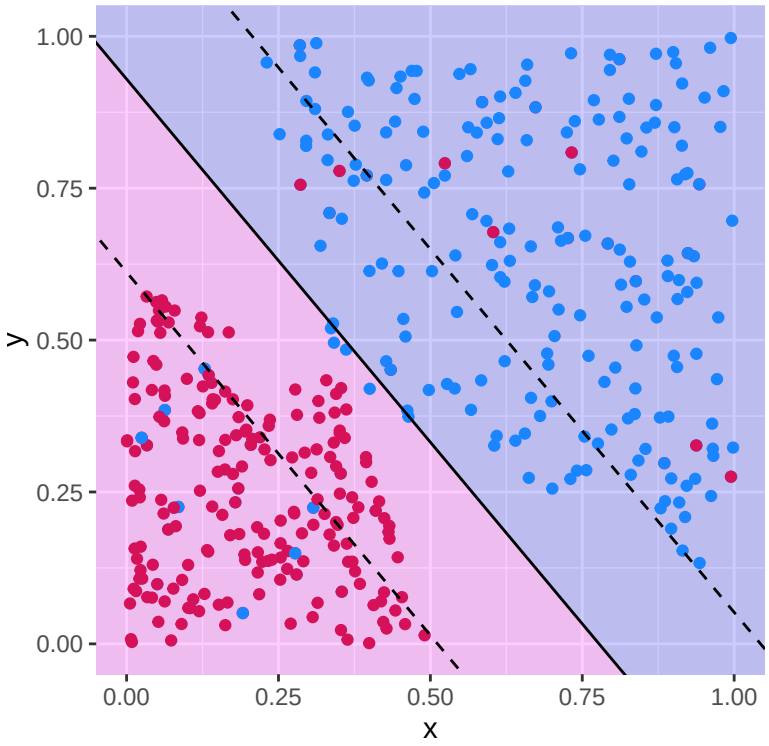

Figure 5.13: Support vector classifier with $c = 100$.

The confusion table:

```
confusion_t <-
  table(observed = df_2$class,
        predicted = predict(svm_fit_2, df_2))
confusion_t
```

```
##           predicted
## observed  blue magenta
##   blue     195      25
##   magenta   20     200
```

There are thus 45 missclassified observations.

```
1 - sum(diag(confusion_t)) / sum(confusion_t)
```

```
## [1] 0.1022727
```

Let us consider a smaller value for the cost parameter of the `svm()` function. Recall that reducing the value of this argument should allow more observations to be missclassified.

```
svm_fit_3 <- svm(
  formula = class~x_1+x_2,
  data = df_2,
  cost = .1,
  # Using a linear kernel
  kernel = "linear",
  # Not scaling the variables
  # Scaling: transforming y and X to get 0 mean and unit variance
  scale = FALSE)
svm_fit_3
```

```
##
## Call:
## svm(formula = class ~ x_1 + x_2, data = df_2, cost = 0.1, kernel
    = "linear",
##     scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
```

```
##  SVM-Kernel:  linear
##        cost:  0.1
##
## Number of Support Vectors:  267
```

The results can be plotted:

```
plot_results(svm_fit_3)
```

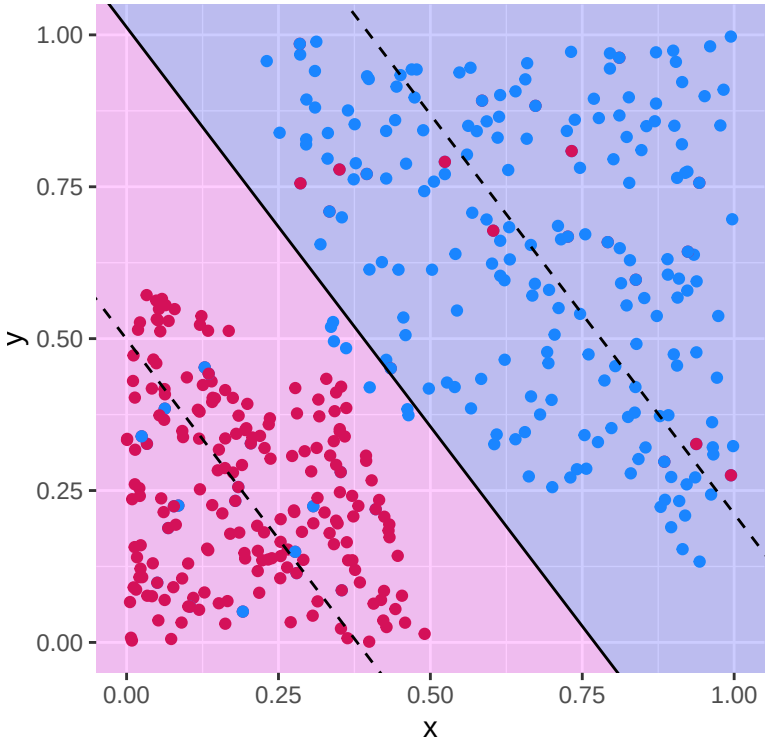

Figure 5.14: Support vector classifier with $c = .1$.

The confusion table:

```
confusion_t <-
  table(observed = df_2$class,
        predicted = predict(svm_fit_3, df_2))
confusion_t
```

```
##           predicted
## observed  blue magenta
##    blue      193      27
##    magenta    20     200
```

There are thus 47 missclassified observations.

```
1 - sum(diag(confusion_t)) / sum(confusion_t)
```

```
## [1] 0.1068182
```

It can be noted that the decision boundary has changed when the cost parameter was modified.

The choice of the cost parameter can be obtained using cross validation, thanks to the `tune()` function from {e1071}. A grid of values need to be provided.

```
values_cost <- 10^seq(3, -2, length = 50)
```

Then the `tune()` can be used:

```
tune_out <- tune(svm,
                 train.x = class~x_1+x_2,
                 data=df_2,
                 kernel="linear",
                 ranges=list(cost=values_cost))
```

By default, 10-fold cross validation is used. The function `tune.control()` passed on to the argument `tunecontrol` of the `tune()` function allows to change this.

The results:

```
tune_out
```

```
##
## Parameter tuning of 'svm':
##
```

```
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##  1000
##
## - best performance: 0.1022727
```

The cost parameter that provided the best results (with respect to the the classification error):

```
tune_out$best.parameters
```

```
##   cost
## 1 1000
```

The best model is returned (hence, there is no need to evaluate it a second time).

```
tune_out$best.model
```

```
##
## Call:
## best.tune(method = svm, train.x = class ~ x_1 + x_2, data = df_2,
##     ranges = list(cost = values_cost), kernel = "linear")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  1000
##
## Number of Support Vectors:  142
```

The confusion table:

```
confusion_t <-
  table(observed = df_2$class, predicted = predict(tune_out$best.model, df_2))
confusion_t
```

```
##          predicted
## observed  blue magenta
```

```
##    blue      195       25
##    magenta    20      200
```

There are thus 45 missclassified observations.

```
1 - sum(diag(confusion_t)) / sum(confusion_t)
```

```
## [1] 0.1022727
```

```
plot_results(tune_out$best.model)
```

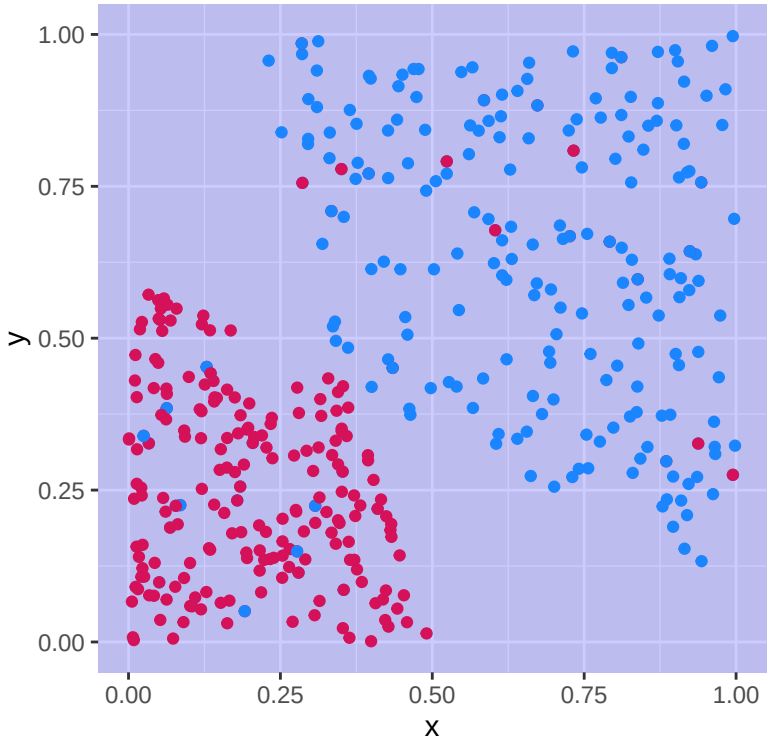

Figure 5.15: Support vector classifier with the cost parameter obtained by cross-validation.

5.3 Support Vector Machines

In this section, we will look at a solution to classification problems when the classes are **not linearly separable**. The basic idea is to **convert a linear classifier** into a classifier that produced **non-linear decision boundaries.**

Let us consider the following illustrative example. Let us generate some data.

```
set.seed(1234)
x_1 <- runif(n = 90, 0, 10)
x_2 <- 2*x_1+3 + rnorm(n = length(x_1))

df <- tibble(x_1 = x_1, x_2 = x_2) %>%
  arrange(x_1)
```

Now, let us assign a class to each observation.

```
df <-
  df %>%
  slice(16:30) %>%
  mutate(class = "blue") %>%
  bind_rows(
    df %>%
      slice(31:60) %>%
      mutate(class = "magenta")
  ) %>%
  bind_rows(
    df %>%
      slice(61:75) %>%
      mutate(class = "blue")
  ) %>%
  mutate(class = factor(class))
```

The data can be visualised:

```
ggplot(data = df, aes(x = x_1, y = x_2, colour = class)) +
  geom_point() +
  scale_colour_manual(
    NULL, values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
  coord_equal(xlim = c(1, 10), ylim = c(5,20)) +
  theme(legend.position = "none")
```

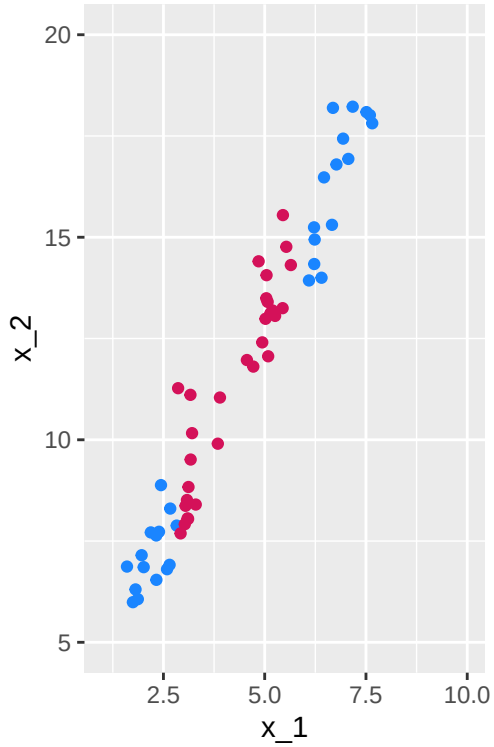

Figure 5.16: Non linearly-separable data.

Let us try to fit a support vector classifier:

```
svm_fit_lin <-
  svm(class~x_1+x_2, data = df, kernel = "linear", scale = FALSE)
```

```
beta_0 <- -svm_fit_lin$rho
beta_1 <- sum(svm_fit_lin$coefs * df$x_1[svm_fit_lin$index])
beta_2 <- sum(svm_fit_lin$coefs * df$x_2[svm_fit_lin$index])

intercept <- -beta_0/beta_2
slope = -beta_1/beta_2

f <- function(x) slope*x+intercept
```

```
ggplot() +
```

```
geom_polygon(
  data = tibble(x = c(-2, 12, 12), y = c(f(-2), f(12), f(-2))),
  aes(x=x, y = y), fill = "magenta", alpha = .2) +
geom_polygon(
  data = tibble(x = c(-2, -2, 12), y = c(f(-2), f(12), f(12))),
  aes(x=x, y = y), fill = "blue", alpha = .2) +
geom_point(data = df, aes(x = x_1, y = x_2, colour=class)) +
scale_colour_manual(
  NULL, values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
scale_fill_manual(
  NULL, values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
geom_abline(intercept = -beta_0/beta_2, slope = -beta_1/beta_2,
            colour = "black") +
geom_abline(slope = slope, intercept = (-beta_0-1)/beta_2,
            colour = "black", linetype = "dashed") +
geom_abline(slope = slope, intercept = (-beta_0+1)/beta_2,
            colour = "black", linetype = "dashed") +
coord_equal(xlim = c(1, 10), ylim = c(5,20)) +
theme(legend.position = "none")
```

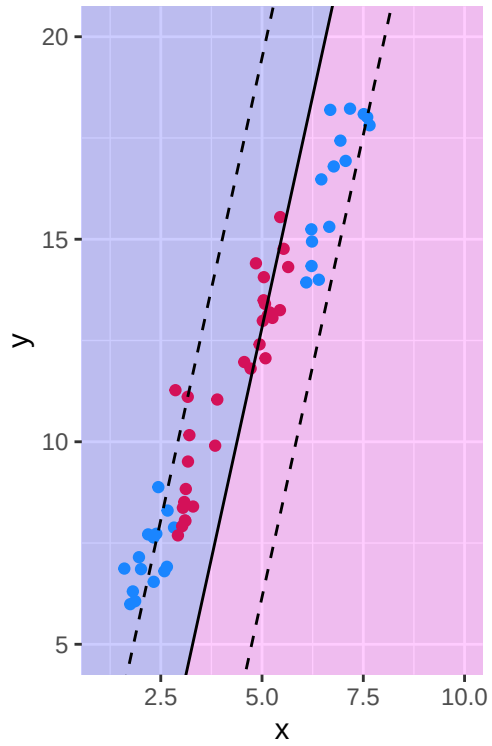

Figure 5.17: Decision boundary of the Support Vector Classifier trained on non linearly-separable data.

Obviously, the results give poor predictive performances.

To account for **non-linear boundaries**, it is possible to add more dimensions to the observation space, either by adding polynomial functions of the predictors, or by adding interaction terms between the predictors. However, as the number of predictors is enlarged, the computations become harder...

The **support vector machine** allows to enlarge the number of predictors while keeping efficient computations. The idea of the **support vector machine** is to fit a separating hyperplane in a space with a higher dimension than the predictor space. Instead of using the set of predictors, the idea is to use a kernel.

The solution of the optimization problem given by Eq. (5.2.1) to Eq. (5.2.5) involves only the **inner products** of the observations. The inner product of two observations x_1 and x_2 is given by $\langle x_1, x_2 \rangle = \sum_{j=1}^{p} x_{1j} x_{2j}$.

The **linear support vector classifier** can be represented as:

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i \langle x, x_i \rangle, \tag{5.3.1}$$

where the n parameters α_i need to be estimated, as well as the parameter β_0. This requires to compute all the $\binom{n}{2}$ inner products $\langle x_i, x_{i'} \rangle$ between all pairs of training observations.

We can see in Eq. (6.1.2) that if we want to evaluate the function f for a new point x_0, we need to compute the inner product between x_0 and each of the points x_i from the training sample. If a point x_i from the training sample **is not from the set $\mathcal{S}$ of the support vectors**, then it can be shown that α_i is equal to zero.

Hence, Eq. (6.1.2) eventually writes:

$$f(x) = \beta_0 + \sum_{i \in \mathcal{S}} \alpha_i \langle x, x_i \rangle, \tag{5.3.2}$$

thus reducing the computational effort to perform when evaluating f.

Now, rather than using the actual inner product $\langle x_i, x_{i'} \rangle = \sum_{j=1}^{p} x_{ij} x_{i'j}$ when it needs to be computed, let us assume that we replace it with a **generalization** of the inner product, following some functional form K known as a **kernel**: $K(x_i, x_{i'})$.

A kernel will compute the similarities of two observations. For example, if we pick the following functional form:

$$K(x_i, x_{i'}) = \sum_{j=1}^{p} x_{ij} x_{i'j}, \tag{5.3.3}$$

it leads back to the **support vector classifier** (or the **linear kernel**).

5.3.1 Polynomial Kernel

We can use a non-linear kernel, for example a **polynomial kernel** of degree d:

$$K(x_i, x_{i'}) = \left(1 + \sum_{j=1}^{p} x_{ij} x_{i'j}\right)^d, \tag{5.3.4}$$

If we do so, the decision boundary will be more flexible. The functional form of the classifier becomes:

$$f(x) = \beta_0 + \sum_{i \in \mathcal{S}} \alpha_i K(x, x_i) \tag{5.3.5}$$

an is such a case, when the support vector classifier is combined with a non-linear kernel, the resulting classifier is known as a **support vector machine**.

In R, with the function `svm()`, all we need to do is to specify the argument `kernel` with the desired kernel. For the polynomial kernel, we need to set `kernel = "polynomial"`. The degree of the polynomial is set through the `degree` argument.

For example, with a 10th order polynomial kernel:

```
degree <- 10
svm_fit_poly <-
  svm(class~x_1+x_2, data = df, kernel = "polynomial",
      degree = degree,
      scale = FALSE, cost=1000, coef0=1)
```

To visualise the decision boundary, we can create a grid and then use the `predict()` method to predict the class on each point of the grid.

```
grid <-
  expand_grid(x_1=seq(0, 11, length.out = 300),
              x_2=seq(4, 21, length.out = 300)) %>%
  tbl_df()
```

The predicted class:

```
grid$pred <- predict(svm_fit_poly, newdata = grid)
head(grid)
```

```
## # A tibble: 6 x 3
##      x_1   x_2 pred
##    <dbl> <dbl> <fct>
## 1      0  4    blue
## 2      0  4.06 blue
## 3      0  4.11 blue
## 4      0  4.17 blue
## 5      0  4.23 blue
## 6      0  4.28 blue
```

```
ggplot() +
  geom_tile(data = grid, aes(x = x_1, y = x_2, fill = pred), alpha = .2) +
  geom_point(data = df, aes(x = x_1, y = x_2, colour=class)) +
```

```
scale_colour_manual(
  NULL, values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
scale_fill_manual(
  NULL, values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
coord_equal(xlim = c(1, 10), ylim = c(5,20)) +
theme(legend.position = "none") +
geom_contour(data = grid %>%
               mutate(pre_val = as.numeric(pred)),
             mapping = aes(x = x_1, y = x_2, z=pre_val),
             breaks = 1.5, colour = "black", linetype = "solid")
```

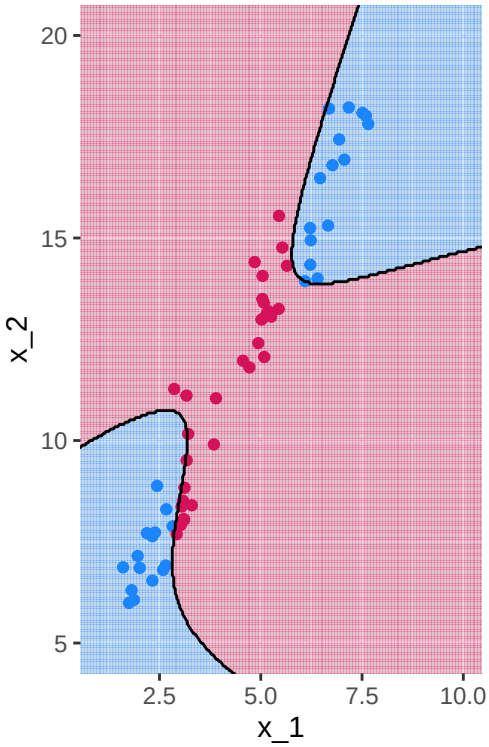

Figure 5.18: Polynomial Kernel, degree 10.

Let us use 10-fold cross-validation to select the couple degree/cost among tested values in a grid search:

```
hyperparam_grid <- list(
  cost = 10^seq(3, -2, length = 50),
  degree = seq(2, 10, by = 2)
)
```

Then the `tune()` can be used, again:

```
tune_out <- tune(svm,
                 train.x = class~x_1+x_2,
                 data=df,
                 kernel="polynomial",
                 ranges=hyperparam_grid)
```

The results of the cross-validation:

```
tune_out
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##       cost degree
##   29.47052      2
##
## - best performance: 0
```

And the selected hyperparameters:

```
tune_out$best.parameters
```

```
##         cost degree
## 16 29.47052      2
```

The confusion table:

```
confusion_t <-
  table(observed = df$class, predicted = predict(tune_out$best.model, df))
confusion_t
```

```
##           predicted
## observed   blue magenta
##    blue      30       0
##    magenta    0      30
```

There are thus 0 missclassified observations.

```
1 - sum(diag(confusion_t)) / sum(confusion_t)
```

```
## [1] 0
```

The predicted class for each observation on our grid of points:

```
grid$pred <- predict(tune_out$best.model, newdata = grid)
```

```
ggplot() +
  geom_tile(data = grid, aes(x = x_1, y = x_2, fill = pred), alpha = .2) +
  geom_point(data = df, aes(x = x_1, y = x_2, colour=class)) +
  scale_colour_manual(
    NULL, values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
  scale_fill_manual(
    NULL, values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
  coord_equal(xlim = c(1, 10), ylim = c(5,20)) +
  theme(legend.position = "none") +
  geom_contour(data = grid %>%
                 mutate(pre_val = as.numeric(pred)),
               mapping = aes(x = x_1, y = x_2, z=pre_val),
               breaks = 1.5, colour = "black", linetype = "solid")
```

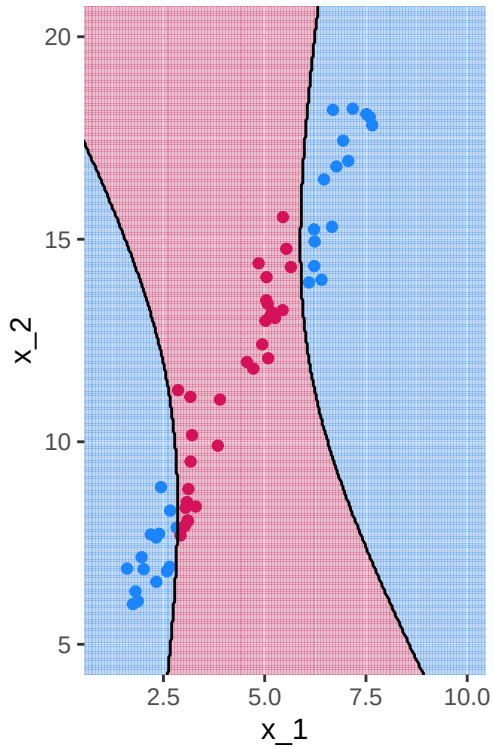

Figure 5.19: Polynomial Kernel, degree 2.

5.3.2 Radial kernel

Other kernels are possible, such as the **radial kernel**:

$$K(x_i, x_{i'}) = \exp\left(-\gamma \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2\right), \tag{5.3.6}$$

where γ is a positive constant that accounts for the smoothness of the decision boundary (and also controls the variance of the model):

- very large values lead to fluctuating decision boundaries that accounts for high variance (and may lead to overfitting)
- small values lead to smoother boundaries and low variance.

If a test observation x_0 is far (considering the Euclidean distance) from a training observation x_i:

- $\sum_{j=1}^{p}(x_{0j} - x_{ij})^2$ will be large
- hence $K(x_0, x_i) = \exp\left(-\gamma \sum_{j=1}^{p}(x_{0j} - x_{ij})^2\right)$ will be really small
- hence x_i will play no role in $f(x_0)$

So, observations far from x_0 will play no role in its predicted class: the radial kernel therefore has a *local* behaviour.

Once again, in R, we need to specify the `kernel` argument of the `svm()` function, and provide a value to the argument `gamma`.

```
gamma <- .1
svm_fit_poly <-
  svm(class~x_1+x_2, data = df, kernel = "radial", gamma = .1,
      scale = FALSE, cost=1000)
```

Again, let us define a grid of points for which the predictions will be made using the SVM, to visually observe the decision boundaries.

```
grid <-
  expand_grid(
    x_1=seq(0, 11, length.out = 300),
    x_2=seq(4, 21, length.out = 300)) %>%
  tbl_df()

grid$pred <- predict(svm_fit_poly, newdata = grid)
```

The decision boundaries can be plotted:

```
ggplot() +
  geom_tile(data = grid,
            mapping = aes(x = x_1, y = x_2, fill = pred),
            alpha = .2) +
  geom_point(data = df,
             mapping = aes(x = x_1, y = x_2, colour=class)) +
  scale_colour_manual(
    NULL, values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
  scale_fill_manual(
    NULL, values = c("blue" = "#1A85FF", "magenta" = "#D41159")) +
```

```
coord_equal(xlim = c(1, 10), ylim = c(5,20)) +
theme(legend.position = "none") +
geom_contour(data = grid %>%
               mutate(pre_val = as.numeric(pred)),
             mapping = aes(x = x_1, y = x_2, z=pre_val),
             breaks = 1.5, colour = "black", linetype = "solid")
```

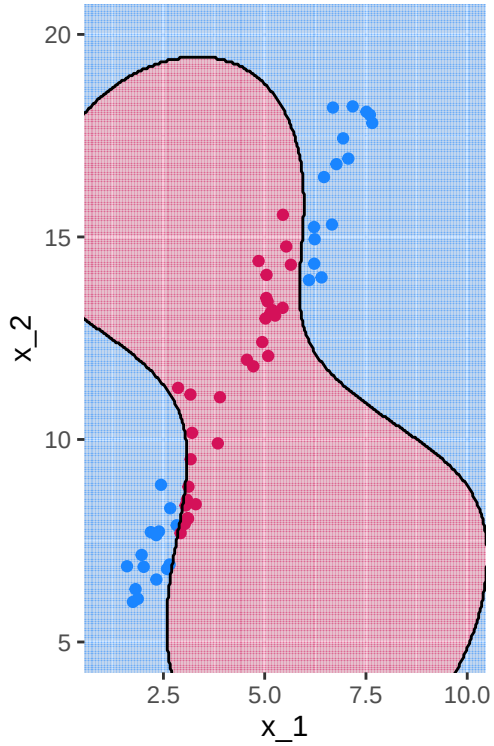

Figure 5.20: Radial Kernel, $\gamma = .1$.

Chapter 6

Deep Learning

When we use linear regression, logistic regression, or SVM what we do can be viewed as using a "simple" **two-layer architecture**:

- first layer: the input data
- second layer: the output data

In a nutshell, the idea behind deep learning is to :

- extract **linear combinations of the inputs** as derived predictors
- use a **combination of simple non-linear functions** on these predictors to predict the output.

Instead of a two-layer architecture, deep learning allows for more layers. Deep learning is built on a **combination of neural networks**, which, when combined, form a **deep neural network**.

Different types of architecture exist for neural networks, among which:

- the multilayer perceptrons (MLP). This is the oldest and simplest technique. The input data goes through the nodes and exit on the output nodes.
- the convolutional neural networks (CNN) [not part of this notebook]. They are based on using convolutions in place of general matrix multiplication in at least one of the layers. Convolutional neural networks are used to process data that has a grid-like topology (*e.g.*, time series in 1D or images in 2D)
- the recurrent neural networks (RNN). With these neural networks, the output of a layer is saved and fed back to the input. At each step, each neuron remembers some information from the previous step. This is useful for sequential data (*e.g.*, text or time series).

This notebook will first present densely connected networks and then mode to recurrent neural networks. It is build using the following main references: James et al. (2021), Chollet and Allaire (2018), Goodfellow et al. (2016), and Ng and Katanforoosh (2018).

Throughout the notebook, the functions from the tidyverse environment will be needed.

```
library(tidyverse)
```

6.1 Neural Networks

A neural network takes an output y and some input data $\mathbf{x}$ with p predictors and n observations. It builds a nonlinear function $f(\mathbf{x})$ to predict the output y.

6.1.1 Neural Network with a Single Hidden Layer

To make things clear, let us rely on the excellent example provided in James et al. (2021). It consists in looking at a feed-forward neural network with $p = 4$ predictors (features), a single hidden layer with $S = 5$ units.

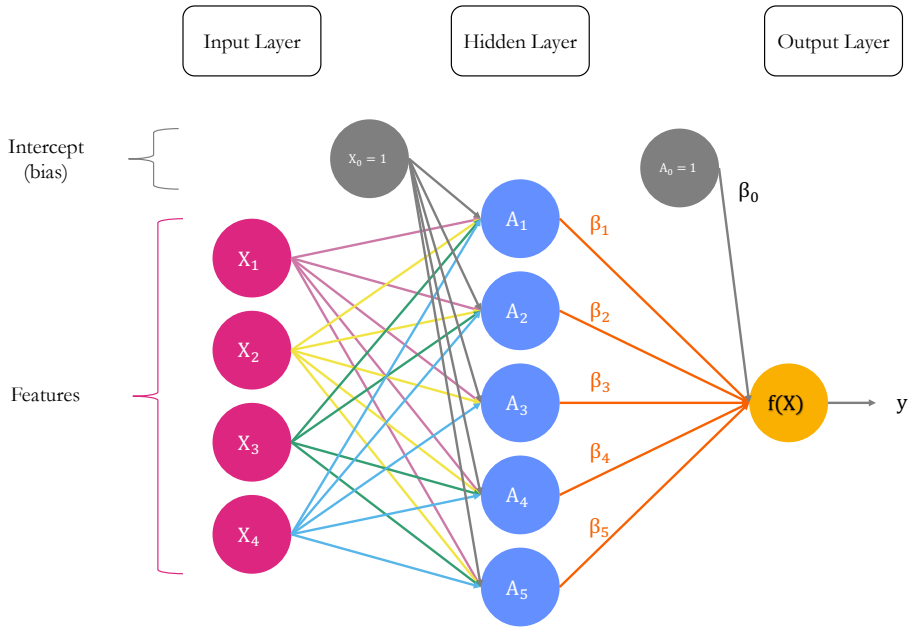

Figure 6.1: A simple Neural Network with 4 predictors and a single hidden layer.

The nonlinear function can be written as follows:

$$f(\mathbf{x}) = \beta_0 + \sum_{s=1}^{S} \beta_s h_s(\mathbf{x}) \tag{6.1.1}$$

$$= \beta_0 + \sum_{s=1}^{S} \beta_s g\left(w_{s_0} + \sum_{j=1}^{p} w_{sj}\mathbf{x_j}\right) \tag{6.1.2}$$

This function is build in two steps. First, each activation (or neuron) A_s, $s = 1, \ldots, S$ in the hidden layer are computed as a linear combination of the p features and an intercept (also known as a neuron bias):

$$A_s = h_s(\mathbf{x_j}) = g_s\left(w_{s_0} + \sum_{j=1}^{p} w_{sj}\mathbf{x_j}\right).$$

The s-th activation is a function $h_s(\cdot)$ of:

- the input $\mathbf{x_j}$
- weighted by a vector of **connection weights** $w_{10}, w_{11}, w_{12}, \ldots, w_{1p}$ between unit s and the m inputs (here $m = p = 4$)
- and an intercept parameter w_{s_0} (also known as a neuron bias)
- associated to an activation function $g_s(\cdot)$ (more details on the functional form of the activation function will be provided after).

The second step in building the nonlinear function $f(\cdot)$ consists in making a linear combination of all the activation functions:

$$f(\mathbf{x}) = \beta_0 + \sum_{s=1}^{S} \beta_s A_s$$

The parameters $\beta_0, \ldots, \beta_k$ need to be estimated, as well as the other parameters w_{10}, w_{11}, w_{12}, ..., w_{1p}, w_{20}, ..., w_{Sp}.

To estimate these parameters, we seek to minimize a non convex cost function. It can be done by computing the gradient of the neural network (this is known as **backpropagation of the gradient**). The cost function may be, for example, the Mean Squared Error in a regression problem.

Different activation functions can be chosen. Let us define those in R.

- the **identify function**: $g(z) = z$
  - $\mathbb{R} \to \mathbb{R}$

- the **sigmoid/logistic function:** $g(z) = \frac{1}{1+\exp(-z)}$
  - $\mathbb{R} \to [0, 1]$

```
sigmoid_f <- function(z) 1/(1+exp(-z))
```

- the **softmax function:** $g(z)_j = \left(\frac{\exp(z_j)}{\sum_{k=1}^{K} \exp z_k}\right)$, for $j \in \{1, \ldots, K\}$, where K is the number of elements of the input vector
  - Generalization of the logistic function
  - $\mathbb{R} \to [0, 1]^K$
- the **hyperbolic tangent function:** $g(z) = \tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)} = \frac{\exp(2z)-1}{\exp(2z)+1}$
  - $\mathbb{R} \to [-1, 1]$

```
hyperbolic_tan_f <- function(z) (exp(2*z) - 1)/(exp(2*z)+1)
```

- the **hard threshold function:** $g_\beta(z) = 1_{(x \geq \beta)}$
  - $\mathbb{R} \to [0, 1]$

```
hard_threshold_f <- function(z, beta) as.numeric(z >= beta)
```

- the **R**ectified **L**inear **U**nit function (ReLU): $g(z) = \max(0, z)$
  - $\mathbb{R} \to [0, max(z)]$, allows to predict positive values. The ReLu function is continuous but not derivable.

```
relu_f <- function(z) pmax(0, z)
```

Let us look at their graphical representation

```
x <- seq(-2, 2, length.out = 1000)
df <- tibble(
  x = x,
  identity = x,
  sigmoid = sigmoid_f(x),
  tanh = hyperbolic_tan_f(x),
  threshold = hard_threshold_f(x, beta = 1),
  ReLU = relu_f(x))
```

```
df <-
  df %>%
  gather(type, value, -x) %>%
  mutate(type = factor(
    type,
    levels = c("identity", "sigmoid",
               "tanh", "threshold", "ReLU"),
    labels = c("Identity", "Sigmoid",
               "Tanh", "Threshold (beta=1)", "ReLU")))

ggplot(data = df, aes(x = x, y = value, colour = type)) +
  geom_line(size = 1.1) +
  labs(x = "z", y = "f(z)") +
  scale_colour_manual(
    NULL, values = c("Identity" = "#648FFF",
                     "Sigmoid" = "#785EF0",
                     "Tanh" = "#DC267F",
                     "Threshold (beta=1)" = "#FE6100",
                     "ReLU" = "#FFB000"))
```

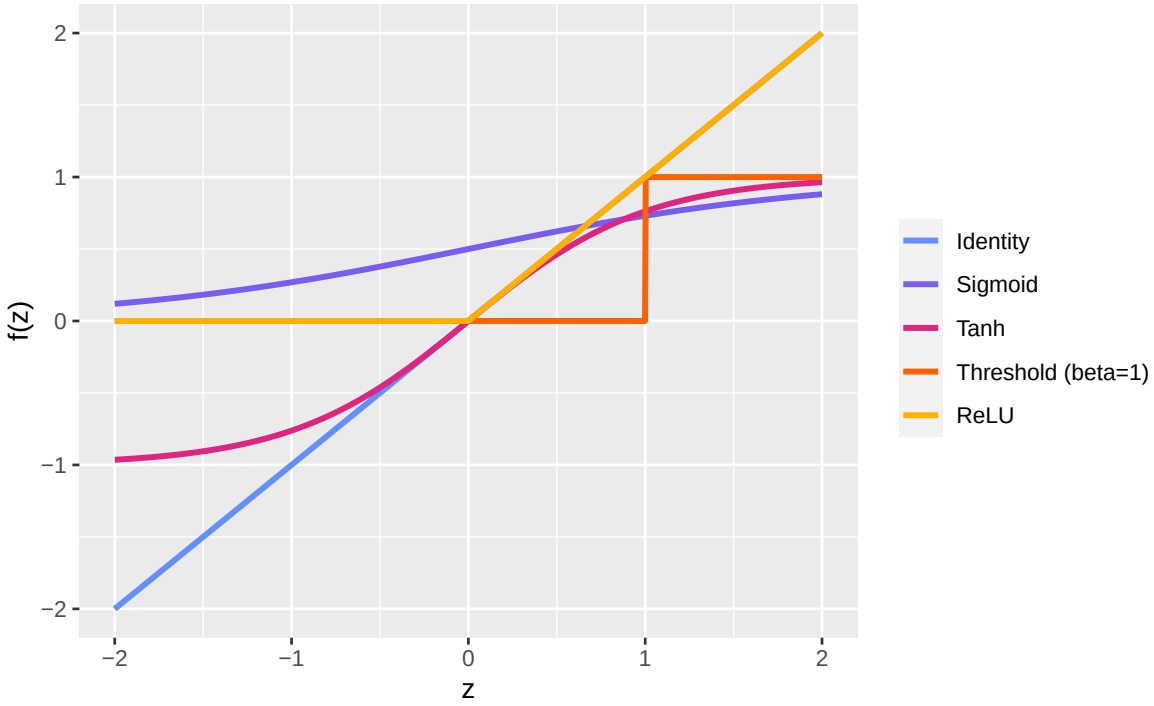

Figure 6.2: Different activation functions.

Figure 6.1, illustrates that after passing through the hidden layer and in particular having been fed to each of the 5 neurons, the model creates 5 new variables which are then fed to the next layer (the output layer in the figure). The activation function in the output layer depends on the **domain of the response function.**

- for example, if the variable y is real valued, the identity function can used
- on the other hand, if the variable y is a binary variable, we would like to obtain a probability of being classified as the reference. In that case, a sigmoid function can be used.

The **number of neurons in the output layer** are picked depending on the type of the variable to predict:

- if it is **numerical**, the output layer contains a single neuron
- if the variable that needs to be predicted is **categorical** with more than two classes, the output layer must contain as many neurons as the output variable contains classes. The activation function of each of the neurons in the output layer can then be picked as the sigmoid function to obtain a probability of being classified in each category.

6.1.2 Multilayer Perception

The **Multilayer perceptron** is a structure composed of several **hidden layers** of neurons/units. We will use ℓ to denote the ℓth layer among the L layers of the neural network. The output of the ith neuron of the ℓth layer becomes the input of the $\ell + 1$ layer.

The **activation function** is the same for all units in a given layer. Once again, in the last layer, the choice of the activation function depends on the domain of the variable to predict and the number of units depends on the type of the data.

While Hornik, Stinchcombe, and White (1989) has shown that a **feed-forward** 3-layers (only one hidden) Neural Network with a finite number of neurons, with the same activation for each neuron in the hidden layer and the identity activation function for the output layer can approximate any bounded and regular function from $\mathbb{R}^p$ to $\mathbb{R}$. This **universal approximation theorem** therefore states that simple neural networks can represent a wide variety of functions. But, as stated by Goodfellow et al. (2016) :

> "we are not guaranteed that the training algorithm will be able to learn that function."

The optimisation algorithm may not be able to find the value of the parameters that corresponds to the desired function. Besides, there is a risk of overfitting:

> "the layer may be infeasibly large and may fail to learn and generalize correctly"

Adding more hidden layers with a modest number of neurons in each should make the learning task easier, as stated in James et al. (2021).

We can illustrate how a multilayer perceptron works by being original (or not) and by taking the example of the famous MNIST dataset (Modified National Institute of Standards) database (Lecun et al. 1998). It contains a training set of 60,000 examples of handwritten digits (0 to 9) and a test set of 10,000 examples. In this database, the digits have been size-normalized and centered to fit into a 28×28 pixel bounding box. Each pixel of each 28×28 image have a grayscale value in the range $0 : 255$.

The data can be accessed through the package {keras}. Keras is a deep learning framework which allows us to build and train deep-learning models. We will use the R interface to Keras (https://keras.rstudio.com/). To be able to use Keras, not only do we need to install the {keras} package, but we also need to install a backend tensor engine. We can, for instance, install TensorFlow.

We can first install the {tensorflow} package:

```
install.packages("tensorflow")
```

Then the {keras} package:

```
install.packages("keras")
```

Once the package is installed, it can be loaded.

```
library(keras)
```

Then, TensorFlow can be installed:

```
tensorflow::install_tensorflow()
```

And can be configured:

```
tensorflow::tf_config()
```

Then, we need to install the core Keras library (we only do it once). If no python environment is found, we may agree to install Miniconda.

```
install_keras()
```

> On one of my computers running on MacOS Big Sur version 11.6, with R 4.0.5, I had an error during the installation ("error code 1"). I used the following instructions to solve the problem:
>
> ```
> unlink(reticulate::miniconda_path(), recursive = TRUE)
> reticulate::install_miniconda(path = reticulate::miniconda_path(),
>                               update = TRUE, force = FALSE)
> keras::install_keras()
> ```

If our computer is equipped with an NVIDIA GPU (Graphics Processing Unit) and our system configured with CUDA and cuDNN libraries, we can install the GPU-based version of the back-end engine. Otherwise, we can stick with the CPU-based (Central Processing Unit) version. The installation needs to be done only once.

```
# install_keras(tensorflow = "gpu")
```

Let us load the MNIST dataset:

```
mnist <- dataset_mnist()
```

It is a list of two arguments: the train set, and the test set. The train set contains $n = 60,000$ images (observations/training examples). They are stored as an array. Recall that each image is made of 28x28 pixels. Hence, each example has $p = 28 \times 28 = 784$ predictors/features.

```
x_train <- mnist$train$x
dim(x_train)
```

```
## [1] 60000    28    28
```

There are $10,000$ observations in the test set:

```
x_test <- mnist$test$x
dim(x_test)
```

```
## [1] 10000    28    28
```

The corresponding labels (true values), y:

```
y_train <- mnist$train$y
head(y_train)
```

```
## [1] 5 0 4 1 9 2
```

And for the test set:

```
y_test <- mnist$test$y
head(y_test)
```

```
## [1] 7 2 1 0 4 1
```

Another way to assign the content of the elements of the list `mnist` to the four objects (`x_train`, `y_train`, `x_test`, and `y_test`) is to use the **multi-assignment operator %<-%** from {zeallot} (the operator is available if {keras} is loaded).

```
c(c(x_train, y_train), c(x_test, y_test)) %<-% mnist
```

There is approximatively 10% of each of the 10 different classes (0, 1, 2, ..., 9):

```
round(100*prop.table(table(y_train)),1)
```

```
## y_train
##    0    1    2    3    4    5    6    7    8    9
##  9.9 11.2  9.9 10.2  9.7  9.0  9.9 10.4  9.8  9.9
```

We can visualize on a graph how the first example looks like. It is labelled as 5:

```
y_train[1]
```

```
## [1] 5
```

Let us create a function to change a training example with its 784 characteristics back to a matrix of 28x28 pixels, and then, let us define another small function to plot this matrix.

```
rotate <- function(x) t(apply(x, 2, rev))
convert_example_tibble <- function(example){
    rotate(example) %>%
    reshape2::melt(c("xx", "yy"),
                   value.name = "grey_scale") %>%
  as_tibble()
}

plot_example <- function(example){
  ggplot(data = convert_example_tibble(example),
       mapping = aes(x = xx, y = yy, fill = grey_scale))+
  geom_tile() +
  coord_equal() +
```

```
  scale_fill_gradient("Grey Scale", low = "white", high = "black")
}
```

```
plot_example(x_train[1,,])
```

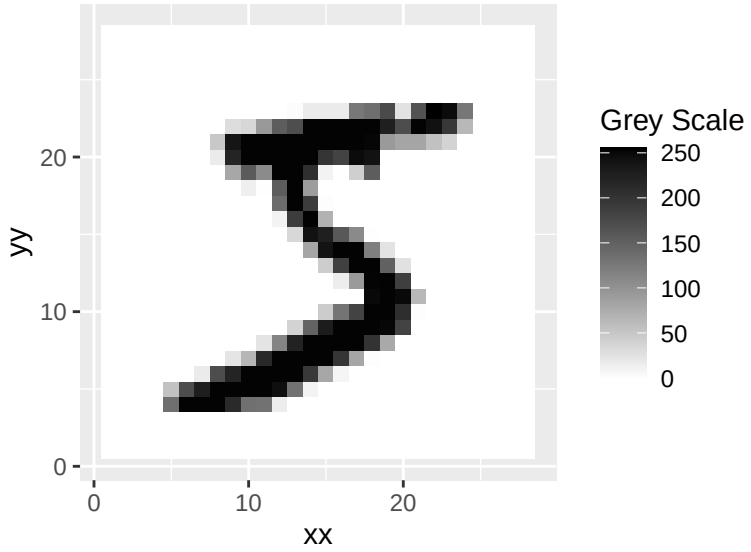

Figure 6.3: The first observation in the training set is a 5.

The first 10 observations from MNIST training set :

```
map_df(1:10, ~convert_example_tibble(x_train[.,,]), .id = "ind") %>%
  mutate(ind = as.numeric(ind)) %>%
  left_join(tibble(y = y_train[1:10], ind = 1:10),
            by = "ind") %>%
  mutate(lab = str_c("ex. ", ind, ": ", y)) %>%
  ggplot(data = .,
       mapping = aes(x = xx, y = yy, fill = grey_scale))+
  geom_tile() +
  coord_equal() +
  scale_fill_gradient("Grey Scale", low = "white", high = "black") +
  facet_wrap(~lab, ncol = 5) +
  theme(axis.text = element_blank(), axis.ticks = element_blank()) +
  labs(x = NULL, y = NULL)
```

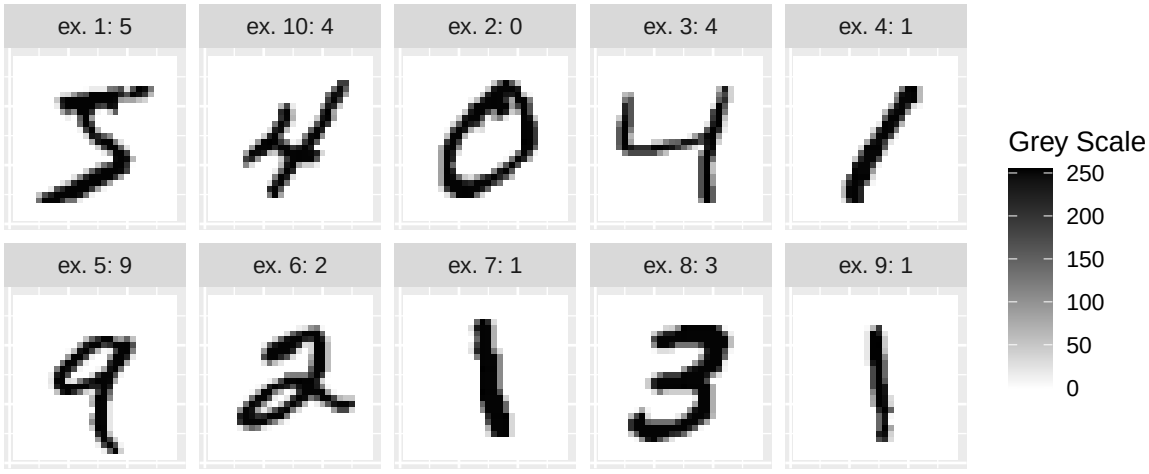

Figure 6.4: The first 10 observations from the MNIST dataset.

Figure 6.5 shows a diagram explaining the architecture of a multilayer perceptron suitable to classify the observations from the MNIST dataset. It has two hidden layers.

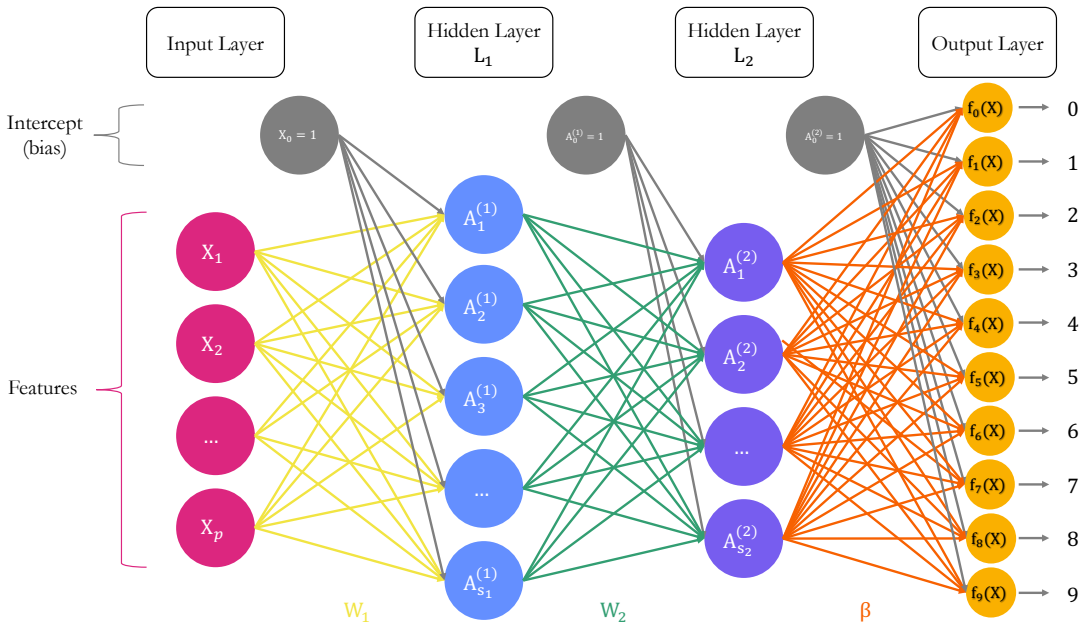

Figure 6.5: A Neural Network with p predictors, two hidden layer, and 10 outputs.

Input Layer

The first layer is the input layer. It contains 784 neurons ($p = 784$ pixels for an observation.

First Hidden Layer

The first hidden layer is made of S_1 units/neurons. The s-th activation writes, for $s = 1, \ldots, S_1$:

$$A_s^{(1)} = h_s^{(1)}(\mathbf{x}) \tag{6.1.3}$$

$$= g\left(w_{s_0}^{(1)} + \sum_{j=1}^{p} w_{sj}^{(1)} \mathbf{x}_j\right) \tag{6.1.4}$$

Each unit applies a nonlinear transformation to a linear combination of the data fed as input. The input of the first hidden layer are the observations: the 784 features, and an intercept/bias. The output of the S_1 activations can then be used as inputs for the next layer. In total, there are $(784 + 1) \times S_1$ weights to estimate in the matrix W_1 of weights.

Second Hidden Layer

The second hidden layer is make of S_2 units. The s-th activation writes, for $s = 1, \ldots, S_2$:

$$A_s^{(2)} = h_s^{(2)}(\mathbf{x}) \tag{6.1.5}$$

$$= g\left(w_{s_0}^{(2)} + \sum_{k=1}^{S_1} w_{sk}^{(2)} A_k^{(1)}\right) \tag{6.1.6}$$

Each unit receives the output of the previous layer as an input as well as an intercept. It then applies a nonlinear transformation function g to a linear combination of the output of the previous layer and an intercept. The output of the S_2 units can then be given as input to the next layer. In total, there are $(S_1 + 1) \times S_2$ weights to estimate in the matrix W_2 of weights.

Output Layer

The last layer is the output layer. As we wish to build a classifier for $M = 10$ classes, the last output needs to contain M = 10 units. In a first step, 10 different models will be constructed, using a linear combination of the input provided by the previous layer, *i.e.*, the S_2 values and an intercept. Hence, for model m, with $m = 0, 1, \ldots, 9$

$$Z_m = \beta_{0,m} + \sum_{s=1}^{S_2} \beta_{s,m} h_s^{(2)}(\mathbf{x}) \tag{6.1.7}$$

$$= \beta_{0,m} + \sum_{s=1}^{S_2} \beta_{s,m} A_s^{(2)} \tag{6.1.8}$$

Each unit receives the output of the previous layer as an input as well as an intercept. It computes a linear combination of these outputs of the previous layer and this intercept. In total, there are $(S_2 + 1) \times M$ weights to estimate in the matrix β of weights.

Then, in a second step, an activation function can be applied. Here, as we wish to obtain probabilities associated to each of the classes (0, 1, ... 9), the activation function is not applied to each unit separately. We would like each unit to return the probability that an observation is of class m, *i.e.*, for $m = 0, 1, \ldots, 9$), we would like to obtain:

$$f_m(\mathbf{x}) = \mathbb{P}(y = m|\mathbf{x}). \tag{6.1.9}$$

A way to do so is to use a **softmax activation function:**

$$f_m(\mathbf{x}) = \frac{\exp(Z_m)}{\sum_{s=0}^{9} \exp(Z_s)}. \tag{6.1.10}$$

6.1.2.1 Practice With Keras: classifier

Let us use Keras to create the architecture. We will use the `keras_model_sequential()` function to create a linear stack of layers. Our neural network will be made of two hidden layers with $S_1 = 256$ and $S_2 = 128$ units in the first and in the second layers, respectively. These layers will be **densely connected** (or, **fully connected**): each output from the previous layer is connected to each input in the current layer. To create a layer, we can use the `layer_dense()` function. The number of units in a later are specified with the `units` argument, while the activation function is specified using the `activation` argument (if no activation is specified, the identity function is applied).

Recall that each observation/training example is, in our case, a 28×28 matrix, stored in an array. We will use the `array_reshape()` function from {reticulate} to reshape the observation to obtain a matrix with each row corresponding to an observation and each column to a predictor.

```
x_train <- array_reshape(x_train, dim = c(nrow(x_train), 784))
x_test <- array_reshape(x_test, dim = c(nrow(x_test), 784))
```

Before feeding the data to the model, we need to preprocess it so that the values are scaled down to $[0, 1]$. This helps the algorithm to converge to a solution. (The weights used are randomly chosen with small values at the beginning of the optimisation iterative process. Scaling the input – and output if it is a numeric variable – will reduce the problem complexity of the problem that the neural network tries to solve). Here, as the pixels correspond to grayscale intensity (0: black, 255:white), we can simply divide the values by the maximum and will get values between 0 and 1.

```
x_train <- x_train/255
```

The response variable is for now numerical. Let us transform it to a qualitative (factor) variable. Since the target has 10 classes, we will get a matrix with 10 columns, each containing a dummy variable. We thus convert a vector of integers to a binary class matrix. This process is referred to as one-hot encoding.

```
y_train <- to_categorical(y_train)
```

We also need to apply the same transformation to the test set:

```
x_test <- x_test/255
y_test <- to_categorical(y_test)
```

Lastly, just before we set up the architecture of the model, let us set some observations aside to observe the performances of the model during the iterative training process. The model will train on some data (train set), we will monitor its performances on unseen data (validation set). We do this, because if we change the structure of the model, we need to be able to know which model performs best. The test data will then be used to know how the "best" model may perform when used on entirely new data. In a way, the data in the validation set may not serve to train the model, but they are used to fine tune it. They are thus not entirely *unseen*.

```
n_valid <- round(.2*nrow(x_train))
ind_partial_valid <-
  sample(1:nrow(x_train), size = n_valid, replace =  FALSE)

x_train_partial <- x_train[-ind_partial_valid, ]
x_validation <- x_train[ind_partial_valid, ]

y_train_partial <- y_train[-ind_partial_valid,]
y_validation <- y_train[ind_partial_valid,]
```

```
dim(x_train_partial)
```

```
## [1] 48000   784
```

```
dim(x_validation)
```

```
## [1] 12000   784
```

```
dim(y_train_partial)
```

```
## [1] 48000    10
```

```
dim(y_validation)
```

```
## [1] 12000    10
```

Here is our sequential model:

```
model_architecture <-
  keras_model_sequential(name = "MNIST_Model") %>%
  # First hidden layer
  layer_dense(units = 256, activation = "relu",
              input_shape = 784) %>%
  # Second hidden layer
  layer_dense(units = 128, activation = "relu") %>%
  # Output layer
  layer_dense(units = 10, activation = 'softmax')
```

Let us have a look at the informations provided:

```
model_architecture
```

```
## Model
## Model: "MNIST_Model"
```

```
##
    ---------------------------------------------------------------------------

##  Layer (type)                 Output Shape
                      Param #
##
    ===========================================================================

##  dense_2 (Dense)              (None, 256)
                      200960
##
##  dense_1 (Dense)              (None, 128)
                      32896
##
##  dense (Dense)                (None, 10)
                      1290
##
##
    ===========================================================================

## Total params: 235,146
## Trainable params: 235,146
## Non-trainable params: 0
##
    ---------------------------------------------------------------------------
```

The first layer is the input data: there is 60,000 training examples, but we have not yet fed them to the model. In the first hidden layer (`dense_2`) L_1, the number of parameters to estimate is $(784+1) \times 256 = 200,960$. In the second hidden layer (`dense_1`) $L2$, the number of parameters to estimate is $(256+1) \times 128 = 32,896$. In the output layer (`dense`), the number of parameters to estimate is $(128+1) \times 10 = 1,290$.

Once the architecture of the model is set, we still need to decide:

1. which **objective function to use** (which loss function)
2. which **optimizer** to use
3. which **metrics** to evaluate during training and testing.

These three aspects are set through the `compile()` from {generics}. Let us use the following:

1. **loss**: since we want to build a classifier (the response variable/target is qualitative), we can use the cross entropy (the model will seek to minimise the negative multinomial log-likelihood)

$$-\sum_{i=1}^{60,000}\sum_{m=0}^{9} = y_{i,m}\log(f_m(x_i))$$

2. **optimizer**: we will use the default optimizer, called "rmsprop," which is an extension of gradient descent
    - if we want to specifically set the learning rate and the decay to values, let us say 0.01 and 10^-6, respectively, we can set the optimizer to :

```
optimizer_rmsprop(lr=0.01, decay = 1e-6)
```

3. **metrics**: the model will evaluate the accuracy through the iterations.

Let us do it (the changes are made in-place, we do not make a new assignment with `<-`):

```
model_architecture %>%
  compile(
    loss = "categorical_crossentropy",
    optimizer = "rmsprop",
    metrics = c("accuracy")
  )
```

It is possible to use a user-defined loss function, and/or metrics.
To define a custom metric function, it is possible to use the `custom_metric()` function from {keras}. The syntax is the following:

```
custom_metric(name, metric_fn)
```

where `name` is the desired name of the metric (a string), and `metric_fn` is a function with the following signature `function(y_true, y_pred)`, that accepts tensors.
However, there is a high chance that the metric we want to use has already been coded (mean squarred error, accuracy, auc, crossentropy, cosine similarity, ...)
To get the list of all the ready-to-use loss functions, we can type `?loss-functions` in the console.

Now that we are all set, the neural network can be trained, using the `fit()` function. We can specify the number of epochs (we will only use 10 here) as well as the size of the batches (we will set it to 128). Recall from the first hands-on session that increasing the size of the batches leads to less noisy optimisation process (but fasten the computation). During the training, it is possible to follow the loss and the desired metrics on the validation sample: we just need to provide the validation set through a list to the `validation_data` argument.

```
mnist_model_hist <-
  model_architecture %>%
  fit(x_train_partial, y_train_partial,
      epochs = 20, batch_size = 128,
      validation_data = list(x_validation, y_validation))
```

During the training, after each epoch, the loss function is evaluated and the result is printed in the console as well as plotted on a graph to monitor the process. As we also asked the accuracy to be reported, it is computed after each epoch and reported both on the console and on a graph too.

We assigned the output returned by `fit()` to an object we named `mnist_model_hist`. This way, we can access the different metrics computed during the training process.

```
mnist_model_hist$metrics
```

```
## $loss
##  [1] 0.294749498 0.115315005 0.076535478 0.054404959 0.041911248
    0.031840410
##  [7] 0.024246907 0.018707141 0.015080354 0.012119832 0.009452516
    0.007689864
## [13] 0.006182983 0.005007274 0.005078920 0.004429060 0.003973258
    0.003519077
## [19] 0.002576303 0.002296944
##
## $accuracy
##  [1] 0.9130625 0.9658750 0.9762292 0.9831458 0.9873750 0.9902500
    0.9926667
##  [8] 0.9942709 0.9952708 0.9961458 0.9968750 0.9974583 0.9978750
    0.9983959
## [15] 0.9984583 0.9986042 0.9988125 0.9987292 0.9991041 0.9991875
##
## $val_loss
##  [1] 0.15497054 0.11522171 0.08932213 0.09704802 0.09755370
    0.10156651
##  [7] 0.08734521 0.08598512 0.09692394 0.09851355 0.12780029
    0.11920659
## [13] 0.12321968 0.14193113 0.13232668 0.14073463 0.13415202
    0.14239880
## [19] 0.16715638 0.16219296
##
## $val_accuracy
##  [1] 0.9535000 0.9659167 0.9726667 0.9704167 0.9716667 0.9729167
    0.9765834
```

```
##  [8] 0.9760000 0.9762500 0.9773333 0.9742500 0.9760833 0.9780833
    0.9743333
## [15] 0.9771667 0.9757500 0.9780000 0.9778333 0.9756666 0.9775000
```

```
plot(mnist_model_hist)
```

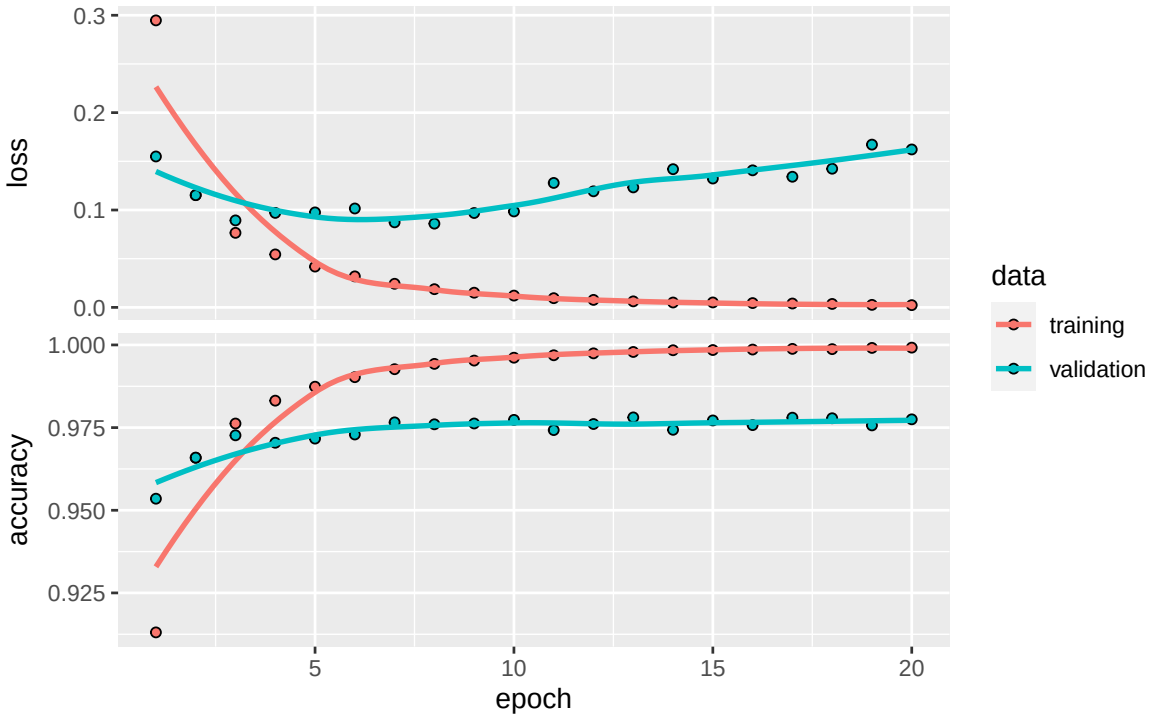

Figure 6.6: Accuracy and loss after each epoch, both on the training and validation sets.

Just for fun, we can also create our own graph:

```
as_tibble(mnist_model_hist$metrics) %>%
  mutate(epoch = row_number()) %>%
  pivot_longer(cols = -epoch) %>%
  mutate(data = ifelse(str_detect(name, "^val_"), "validation", "training"),
         name = str_remove(name, "^val_")) %>%
  ggplot(data = .,
         mapping = aes(x = epoch, y = value, colour = data)) +
  geom_line() +
```

```
  geom_point() +
  scale_colour_manual("Sample",
                      values = c("validation" = "#E69F00",
                                 "training" = "#0072B2")) +
  facet_wrap(~name, scales = "free_y", ncol = 1)
```

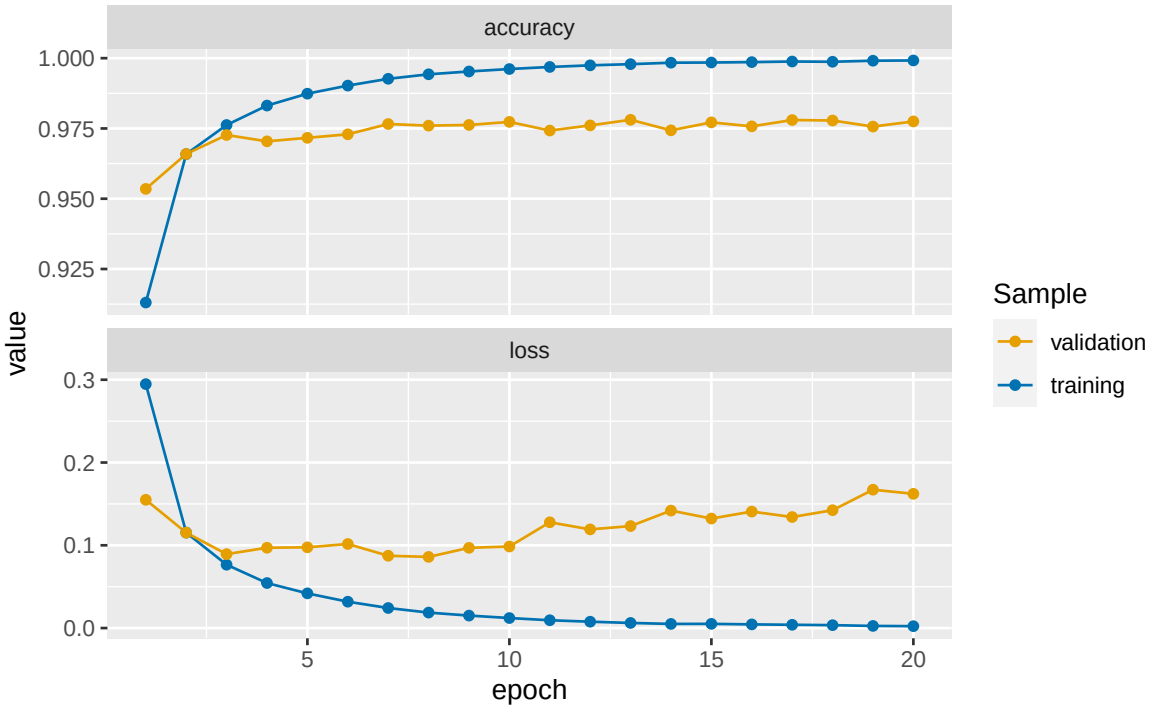

Figure 6.7: Accuracy and loss after each epoch, custom made graph.

We reached an accuracy 99.92%.

We notice that after the third epoch, the accuracy in the training set becomes higher than that of the validation set: we therefore face overfitting. We could then stop the training earlier to avoid overfitting and still have fair predictive capacities.

Let us have a look at the performances of the model on the test set. To do so, we can use the `evaluate()` function.

```
performances_test <- model_architecture %>%
  evaluate(x_test, y_test)
```

```
performances_test
```

```
##     loss accuracy
## 0.149107 0.979500
```

The accuracy in the test set is 97.95%.

To use the estimated model to make predictions, the function `predict()` can be used.

```
mnist_pred_test <- model_architecture %>% predict(x_test)
```

As we have a multi-class target variable (10 classes), the model returned probabilities of belonging to each class. Hence, when we applied the `predict()` function on the test sample with $10,000$ observations, we obtain a matrix with $10,000$ rows and 10 columns.

For the first observation from the test sample, the predicted probabilities are the following:

```
mnist_pred_test[1,]
```

```
##  [1] 5.216950e-22 5.249442e-25 1.699038e-18 6.831150e-16 2.110166
    e-27
##  [6] 5.990859e-20 3.190818e-32 1.000000e+00 1.109502e-25 7.978255
    e-18
```

While the observed value is:

```
y_test[1, ]
```

```
##  [1] 0 0 0 0 0 0 0 1 0 0
```

The first observation from the test set, if we predict the class based on the highest estimated probability is correctly predicted by the model.

We can make sure that the sum of the predicted probabilities, thanks to the softmax function used in the output layer, sum up to 1.

```
sum(mnist_pred_test[1,])
```

```
## [1] 1
```

Let us have a look at the predictive capacities for each categories. To do so, we first need to obtain the predicted class for each observation. We will predict the class of an observation based on the highest probability estimated by the model among the k classes.

```
mnist_pred_test_class <-
  as_tibble(mnist_pred_test, .name_repair = "unique") %>%
  mutate(pred_class = max.col(.),
         pred_class = seq(0,9)[pred_class]) %>%
  magrittr::extract2("pred_class")
```

The observed class:

```
mnist_test_class <-
  y_test %>%
  as_tibble(.name_repair = "unique") %>%
  mutate(y_class = max.col(.),
         y_class = seq(0,9)[y_class]) %>%
  magrittr::extract2("y_class")
```

The confusion table writes:

```
table(observed = mnist_test_class,
      predicted = mnist_pred_test_class)
```

```
##          predicted
## observed    0    1    2    3    4    5    6    7    8    9
##        0  974    0    0    2    0    0    1    1    2    0
##        1    0 1128    1    1    0    0    2    1    2    0
##        2    4    8  995    4    2    0    2    8    9    0
##        3    0    0    5  983    0    8    1    5    2    6
##        4    2    0    2    0  964    0    3    3    1    7
##        5    2    0    0    6    3  872    4    1    3    1
```

```
##         6    4    3    1    1    9    2  936    0    2    0
##         7    0    6    5    1    1    0    0 1011    1    3
##  [ getOption("max.print") est atteint -- 2 lignes omises ]
```

A graph showing the percentage of correct predictions can be made. To that end, let us obtain the percentage of correct predictions for each class.

```
table_pct_pred <-
  tibble(
  observed = factor(mnist_test_class, levels = 0:9),
  predicted = factor(mnist_pred_test_class, levels = 0:9)
) %>%
  group_by(observed) %>%
  mutate(n_k = sum(n())) %>%
  group_by(observed, predicted, n_k) %>%
  count() %>%
  mutate(pred_pct = n / n_k) %>%
  ungroup() %>%
  select(observed, predicted, pred_pct)
table_pct_pred
```

```
## # A tibble: 73 x 3
##    observed predicted pred_pct
##    <fct>    <fct>        <dbl>
##  1 0        0          0.994
##  2 0        3          0.00204
##  3 0        6          0.00102
##  4 0        7          0.00102
##  5 0        8          0.00204
##  6 1        1          0.994
##  7 1        2          0.000881
##  8 1        3          0.000881
##  9 1        6          0.00176
## 10 1        7          0.000881
## # ... with 63 more rows
```

If we use this table directly, there will be missing cases. Let us complete it with 0 values when there are 0 predictions made for a given class when the true class is of class k.

```
classes_mnist <-
  levels(factor(mnist_test_class, levels = 0:9))
df_plot_pred_mnist <-
  expand_grid(
  observed = classes_mnist,
  predicted = classes_mnist) %>%
  left_join(
    table_pct_pred
  ) %>%
  mutate(pred_pct = replace_na(pred_pct, 0))
```

The graph showing the percentage of predicted classes for each observed class can then be plotted.

```
ggplot(data = df_plot_pred_mnist,
       mapping = aes(x = predicted, y = observed,
                     fill = pred_pct,
                     label = round(pred_pct, 2))) +
  geom_tile() +
  geom_text(color = "white") +
  scale_fill_gradient("% of\n correct pred.",
                      low = "#1E88E5", high = "#D81B60") +
  labs(x = "Predicted", y = "Observed") +
  theme(panel.background = element_blank())
```

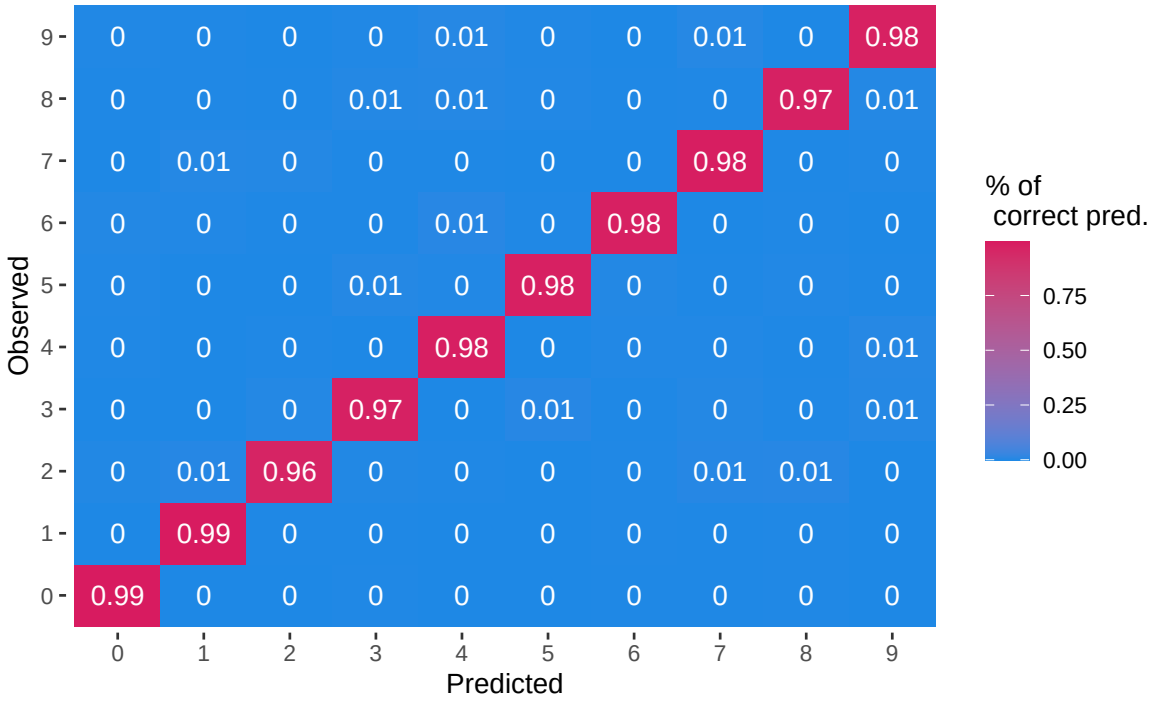

Figure 6.8: Percentage of predicted classes for each observed class.

Another way of looking at the predictive capacities consists in looking at the percentage of correctly predicted observation per class.

```
table_pct_pred_2 <-
  tibble(
  observed = factor(mnist_test_class, levels = 0:9),
  predicted = factor(mnist_pred_test_class, levels = 0:9)
) %>%
  mutate(correct = observed == predicted) %>%
  group_by(observed) %>%
  summarise(pct_correct = sum(correct) / n()) %>%
  mutate(pct_incorrect = 1-pct_correct) %>%
  pivot_longer(cols = -observed) %>%
  mutate(name = factor(name,
                       levels = c("pct_correct", "pct_incorrect"),
                       labels = c("Correctly predicted",
                                  "Incorrect predicted")))
```

The following graph can then be plotted.

```
ggplot(data = table_pct_pred_2,
       mapping = aes(x = name, y = observed,
                     fill = value,
                     label = round(value, 2))) +
  geom_tile() +
  geom_text(color = "white") +
  scale_fill_gradient("% of\n correct pred.",
                      low = "#1E88E5", high = "#D81B60") +
  labs(x = "Accuracy of prediction", y = "Observed") +
  theme(panel.background = element_blank())
```

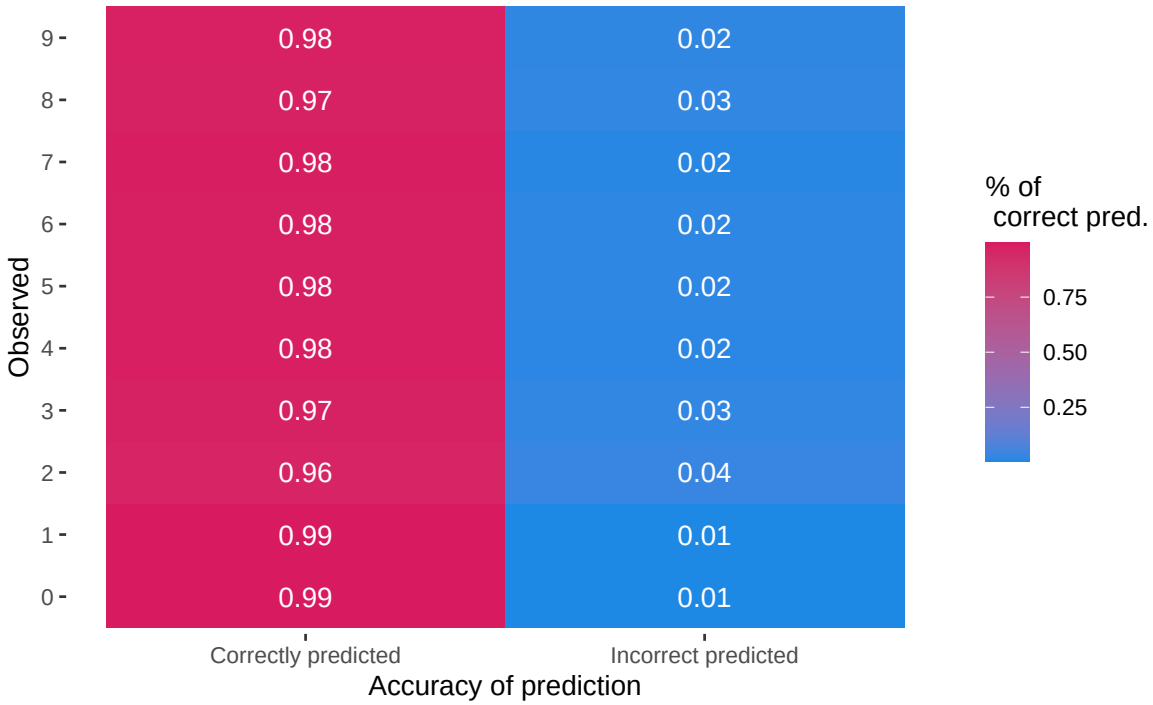

Figure 6.9: Percentage of correctly or incorrectly predicted observations by observed class.

The output layer is made of 10 units. Let us have a look at what happens if we just change one thing : the number of units in one of the hidden layers, such that it becomes very small compared to the number of classes to be predicted. We will here set the number of units in the second layer to only 2.

```
model_architecture_2 <-
  keras_model_sequential(name = "MNIST_Model_2") %>%
  # First hidden layer
  layer_dense(units = 256, activation = "relu",
              input_shape = 784) %>%
  # Second hidden layer
  layer_dense(units = 2, activation = "relu") %>%
  # Output layer
  layer_dense(units = 10, activation = 'softmax')
```

This is the only change. We will use the same optimizer and the same loss function.

```
model_architecture_2 %>%
  compile(
    loss = "categorical_crossentropy",
    optimizer = "rmsprop",
    metrics = c("accuracy")
  )
```

Let us train the model over 20 epochs and a batch size of 128, as earlier.

```
mnist_model_2_hist <-
  model_architecture_2 %>%
  fit(x_train_partial, y_train_partial,
      epochs = 20, batch_size = 128,
      validation_data = list(x_validation, y_validation))
mnist_model_2_hist
```

We clearly note that the accuracy of the model dropped. This is because the information was too compressed when going through the second layer.

```
as_tibble(mnist_model_hist$metrics) %>%
  mutate(model = "Initial model",
         epoch = row_number()) %>%
  bind_rows(as_tibble(mnist_model_2_hist$metrics) %>%
              mutate(model = "Second model",
                     epoch = row_number())) %>%
  pivot_longer(cols = -c(epoch, model)) %>%
  mutate(data = ifelse(str_detect(name, "^val_"),
```

```
                      "validation", "training"),
       name = str_remove(name, "^val_")) %>%
ggplot(data = .,
       mapping = aes(x = epoch, y = value, colour = data,
                     linetype = model)) +
geom_line() +
geom_point() +
scale_colour_manual("Sample",
                    values = c("validation" = "#E69F00",
                               "training" = "#0072B2")) +
facet_wrap(~name, scales = "free_y", ncol = 1) +
labs(y = NULL)
```

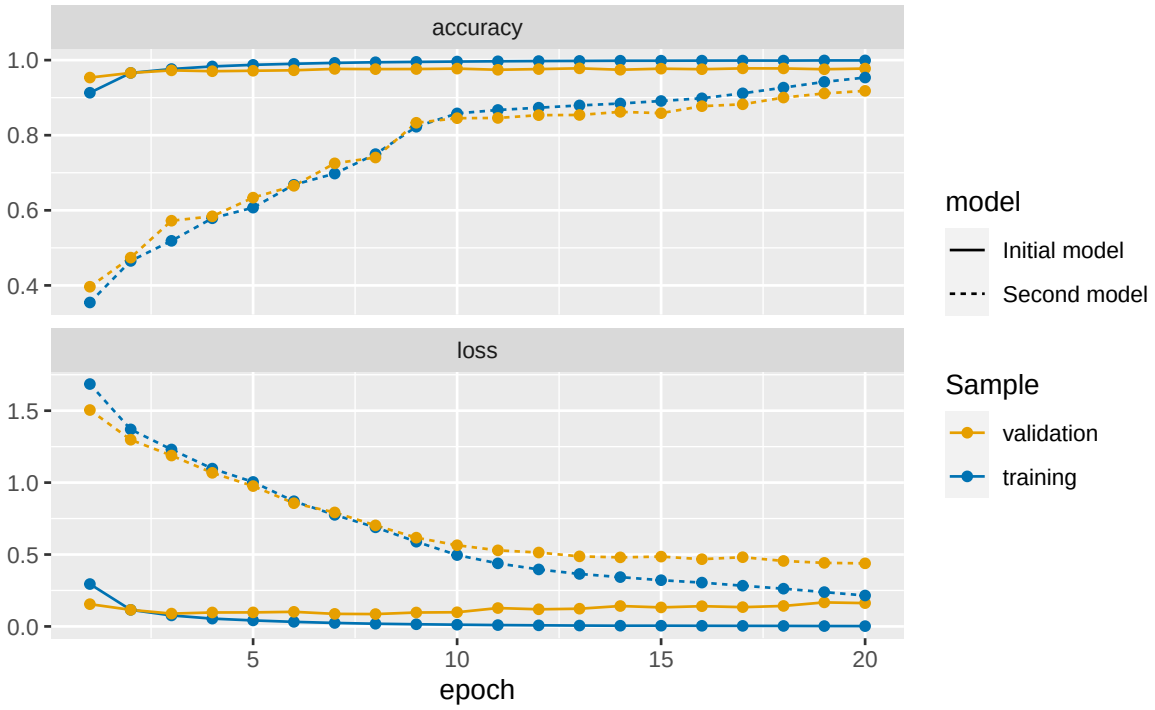

Figure 6.10: Accuracy and loss after each epoch, custom made graph.

To save the model using HDF5 files, {keras} provides the function `save_model_hdf5()`.

```
if(!dir.exists("deep_learning_models"))
  dir.create("deep_learning_models")
model_architecture_2 %>%
  save_model_hdf5("deep_learning_models/mnist_model.h5")
```

A model saved in a HDF5 file can then be loaded using the `load_model_hdf5()` function:

```
model_architecture_2 <-
  load_model_hdf5("deep_learning_models/mnist_model.h5")
```

To destroy the current TensorFlow graph and create a new one, the function `k_clear_session()` can be used. This is useful to avoid clutter from previous models or layers.

```
k_clear_session()
```

6.1.2.2 Practice With Keras: Regression

To illustrate how to build and estimate a neural network in a regression context, let us use the Seoul bike data from the second hands-on session.

```
load(url("http://egallic.fr/Enseignement/ML/ECB/data/bike.rda"))
```

We have created new variable in the data during the second hands-on session. However, we still need to pre-process the data a little bit. While some functions such as `lm()`, `svm()` or `glm()` automatically create dummy variables in the design matrix whenever it contains a qualitative variable, this is not the case with the functions we use in {keras}. We thus need to manually transform categorical variables to dummy variables. We can use the `dummy_cols()` function from {fastDummies} to create dummy variables from `character` or `factors`. Here is a simple example:

```
library(fastDummies)
tibble(x = c("A", "B", "C"),
       y = factor(c("mon", "tue", "wed")),
       z = 1:3) %>%
  # Creating dummies for chr and fct
  dummy_cols(
    remove_first_dummy = TRUE,
    remove_selected_columns = TRUE)
```

```
## # A tibble: 3 x 5
```

```
##        z   x_B   x_C y_tue y_wed
##    <int> <int> <int> <int> <int>
## 1      1     0     0     0     0
## 2      2     1     0     1     0
## 3      3     0     1     0     1
```

By setting `remove_first_dummy = TRUE`, we make sure to remove the first dummy of each categorical variable (to avoid colinearity), and by setting `remove_selected_columns = TRUE` we remove the original column from the table.

Let us do it with the bike dataset:

```
x_data <-
  bike %>%
  select(-date, -rented_bike_count, -y_binary) %>%
  dummy_cols(
    remove_first_dummy = TRUE,
    remove_selected_columns = TRUE)
```

The variable to predict (the target) is the number of bikes rented each hour:

```
y_data <- bike$rented_bike_count
```

Let us create the training and the test sets. For the moment, let us ignore the fact that bicycle rental data can be regarded as time-series. We will discuss sequentiality in the data later.

```
n_train <- round(.8*nrow(x_data))
set.seed(123)
ind_train <- sample(1:nrow(x_data), size = n_train, replace = FALSE)
```

The train set:

```
x_train <- x_data[ind_train, ]
y_train <- y_data[ind_train]
dim(x_train)
```

```
## [1] 6772   31
```

```
length(y_train)
```

```
## [1] 6772
```

And the test set:

```
x_test <- x_data[-ind_train, ]
y_test <- y_data[-ind_train]
dim(x_test)
```

```
## [1] 1693   31
```

```
length(y_test)
```

```
## [1] 1693
```

Let us scale the data, by removing to each variable x_j, $j = 1, \ldots, p$ its mean $\overline{x}_j$ and by dividing the result by its standard deviation σ_{x_k}: $\frac{x_j - \overline{x}_j}{\sigma_{x_j}}$. We thus need to compute the mean and standard deviation of each variable in the **training test**:

```
mean_train <- apply(x_test, 2, mean)
std_train <- apply(x_test, 2, sd)
```

Then, the data from both the train and test set can be scaled:

```
x_train <- scale(x_train, center = mean_train, scale = std_train)
x_test <- scale(x_test, center = mean_train, scale = std_train)
```

> Note that the data from the **test** set were scaled using the means and standard deviations computed on the train set.

Let us create a function that defines the model depending on the train data. As the number of observation is modest, we will only consider a small network with three hidden layers. We will monitor the Mean Absolute Error (MAE) :

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \widehat{y}_i|.$$

```
model_structure_bike <- function(train_data) {
  model <-
    keras_model_sequential() %>%
    # First hidden layer
    layer_dense(units = 128, activation = "relu",
                input_shape = dim(train_data)[[2]]) %>%
    # Second hidden layer
    layer_dense(units = 128, activation = "relu") %>%
    # Third hidden layer
    layer_dense(units = 64, activation = "relu") %>%
    # Output layer
    layer_dense(units = 1)

  model %>%
    compile(
    optimizer = "rmsprop",
    loss = "mse",
    metrics = c("mae")
) }
```

We will use 5-fold cross-validation to evaluate the neural network. We do so because we will need to adjust some parameters of the model to find one that suits us (we can change the number of hidden layers and their number of respective units, the number of epochs, the batch size, and so on). We therefore need to compare the models with each other.

```
nb_folds <- 5
```

Here are the assignment of the folds:

```
set.seed(123)
folds <- sample(rep(1:nb_folds, length=nrow(x_train)))
```

Let us set the number of epoch to 10 and the batch size to 1. Recall from the first hands-on session that setting the batch size to 1 means that during an epoch, each observation will be used in turn to update the parameters of the models.

```
num_epochs <- 10
batch_size <- 32
```

```
loss_and_metrics <- NULL
```

By using `verbose = 0` in the functions `fit()` and `evaluate()`, the model is trained silently, without either plotting the loss and MAE nor printing the values in the console.

```
for(k in 1:nb_folds){
  # cat(str_c("\nFold number: ", k, "\n"))

  ind_current <- folds != k

  # Train set
  x_train_partial <- x_train[which(ind_current),]
  y_train_partial <- y_train[which(ind_current)]

  # Validation set
  x_train_valid <- x_train[which(!ind_current),]
  y_train_valid <- y_train[which(!ind_current)]

  # Building the model
  model <- model_structure_bike(x_train_partial)

  # Training the model on the training set
  model %>%
    fit(x_train_partial, y_train_partial,
        epochs = num_epochs, batch_size = batch_size, verbose = 0)

  # Evaluating its performances on test data
  results_current <- model %>%
    evaluate(x_train_valid, y_train_valid, verbose = 0)

  # Store loss and metrics
  loss_and_metrics <-
    loss_and_metrics %>% bind_rows(
      as_tibble(t(results_current)) %>%
        mutate(fold = k)
    )
```

```
}
```

The computed metrics with thus version of the neural network:

```
loss_and_metrics
```

```
## # A tibble: 5 x 3
##       loss    mae  fold
##      <dbl> <dbl> <int>
## 1 131442.   243.      1
## 2 140303.   255.      2
## 3 123807.   239.      3
## 4 119251.   234.      4
## 5 120723.   243.      5
```

The average of the the Mean Average Error over the folds is:

```
mean(loss_and_metrics$mae)
```

```
## [1] 242.6967
```

That means that on average, our predictions are off by NA bikes. This is quite high, when compared to the distribution of the target variable.

```
summary(y_train)
```

```
##     Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      2.0   214.0   539.5   730.0  1087.0  3556.0
```

Let us train the model over more epochs to see if it yields better results. Let us train it over 50 epochs.

```
num_epochs <- 800
batch_size <- 32
```

```
loss_and_metrics_2 <- vector(mode = "list", length = nb_folds)
```

Warning: the following code takes about 15 minutes to run on a standard computer.

This time, let us keep track of the training history as in the first example with the MNIST dataset.

```
library(tictoc)

results <- vector(mode = "list", length = nb_folds)

tic()
for(k in 1:nb_folds){
  cat(str_c("\nFold number: ", k, "\n"))

  ind_current <- folds != k

  # Train set
  x_train_partial <- x_train[which(ind_current),]
  y_train_partial <- y_train[which(ind_current)]

  # Validation set
  x_train_valid <- x_train[which(!ind_current),]
  y_train_valid <- y_train[which(!ind_current)]

  # Building the model
  model <- model_structure_bike(x_train_partial)

  # Training the model on the training set
  history <-
    model %>%
    fit(x_train_partial, y_train_partial,
        validation_data = list(x_train_valid, y_train_valid),
        epochs = num_epochs, batch_size = batch_size, verbose = 0)

  # Evaluating its performances on test data
  results_current <- model %>%
    evaluate(x_train_valid, y_train_valid, verbose = 0)

    results[[k]] <- results_current

  # Store loss and metrics
```

```
  loss_and_metrics_2[[k]] <- history
}
toc()
```

Let us compute, for each epoch, the average of the 5-folds MAE both on the train and on the validation samples:

```
mae_mean_folds <-
  map(loss_and_metrics_2, "metrics") %>%
  map("val_mae") %>%
  map_df(~as_tibble(.) %>% mutate(epoch = row_number()),
         .id = "fold") %>%
  mutate(sample = "validation") %>%
  bind_rows(
    map(loss_and_metrics_2, "metrics") %>%
      map("mae") %>%
      map_df(~as_tibble(.) %>% mutate(epoch = row_number()),
             .id = "fold") %>%
      mutate(sample = "train")
  ) %>%
  group_by(epoch, sample) %>%
  summarise(mae = mean(value), .groups = "drop")
mae_mean_folds
```

```
## # A tibble: 1,600 x 3
##    epoch sample       mae
##    <int> <chr>      <dbl>
##  1     1 train       440.
##  2     1 validation  300.
##  3     2 train       281.
##  4     2 validation  269.
##  5     3 train       263.
##  6     3 validation  264.
##  7     4 train       255.
##  8     4 validation  255.
##  9     5 train       250.
## 10     5 validation  258.
## # ... with 1,590 more rows
```

Let us have a look at the metrics over the epochs:

```
ggplot(data = mae_mean_folds,
       mapping = aes(x = epoch, y = mae, colour = sample)) +
  geom_line() +
  scale_colour_manual("Sample",
                      values = c("validation" = "#E69F00",
                                 "train" = "#0072B2")) +
  labs(x = "Epoch", y = "MAE (validation sample)")
```

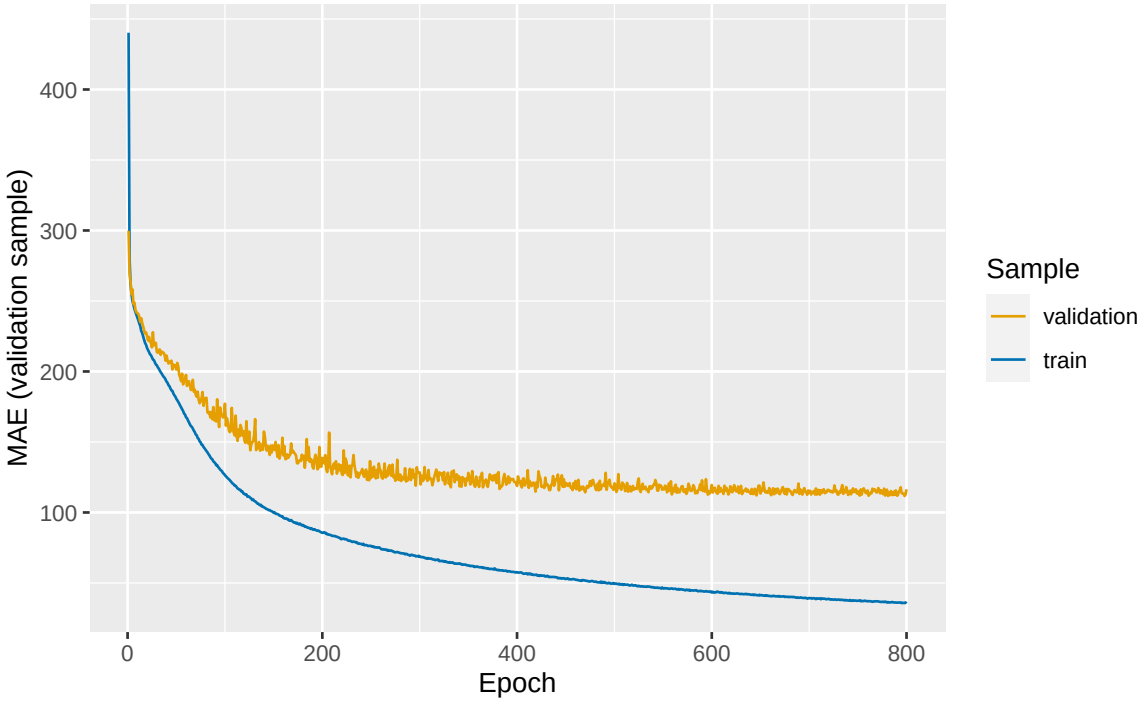

Figure 6.11: Mean Absolute Error (MAE) computed on the validation samples, over the epochs

The model overfits. The MAE seems to be stable after around 400 epochs. To avoid overfitting while keeping the number of observations constant, we can reduce both the number of layers and the number of units in them. We could also add weight regularization and add dropout. More on those aspects will be presented below.

Trying to adjust the model to find an architecture and hyperparameters that provides good results takes some time (this process is . It is possible to use a grid as in the second hands-on session to loop over different configurations of the network. After we have found a model that suits us, we can train a final one, which can then be used to make predictions on unseen data.

Warning: the following code takes about 5 minutes to run on a standard machine.

Let us say our final model is the following one:

```
num_epochs <- 400
batch_size <- 32
model <- model_structure_bike(x_train)
model %>%
    fit(x_train, y_train,
        epochs = num_epochs, batch_size = batch_size, verbose = 0)
```

When estimating the final model, we use the hyperparameters that lead to the best results, and we use the whole training dataset to train the algorithm.

The predictive abilities of the model on unseen data:

```
results_test <- model %>% evaluate(x_test, y_test)
results_test
```

```
##        loss         mae
## 31469.1484   105.7719
```

6.1.2.3 Regularisation techniques

6.1.2.3.1 Weight Regularization Recall Emanuelle Flachaire's course about penalised regressions such as Lasso, Ridge, or Elastic Net. In those regressions, the model is penalised for having too many variables. This is done through the introduction of a constraint in the model equation (and therefore in the loss function to be minimised).

The same idea of regularisation has been introduced in neural networks, to put some constraints on the complexity of the network, to avoid overfitting. These constraints are applied to the weights to force them to take only small values. Lasso (L1 norm) or Ridge (L2 norm/weight decay) regularisation on weights can be used.

Weight regularisation to the neural network with Keras is done by specifying the `layer_dense()` function with the argument `kernel_regularizer`. For example, for a L1 norm, we can use `kernel_regularizer = regularizer_l1(l = 0.01)`, while for a L2 norm, we can use `kernel_regularizer = regularizer_l2(l = 0.01)`.

To illustrate how regularisation techniques help to avoid overfitting, we will create a network that overfits. The risk of overfitting increases with the complexity of the model (controlled by

the number of hidden layers and their respective number of units) and with the number of epochs the model is trained over.

Consider, again, the Seoul bike data. We will build a classifier to try to predict the binary variable `y_binary` that indicates whether the number of bikes rented per hours is belo 300 ("Low") or not ("High").

```
x_data <-
  bike %>%
  select(-date, -rented_bike_count, -y_binary) %>%
  dummy_cols(
    remove_first_dummy = TRUE,
    remove_selected_columns = TRUE)

# 1 if "Low", 0 otherwise
# y_data <- to_categorical(ifelse(bike$y_binary == "Low", yes = 1, no = 0))
y_data <- ifelse(bike$y_binary == "Low", yes = 1, no = 0)
```

The first 80% observations are kept for the train set and the remaining for the test set.

```
n_train <- round(.8*nrow(x_data))
set.seed(123)
ind_train <- sample(1:nrow(x_data), size = n_train, replace = FALSE)
```

The training test is thus:

```
x_train <- x_data[ind_train, ]
y_train <- y_data[ind_train]
dim(x_train)
```

```
## [1] 6772   31
```

```
length(y_train)
```

```
## [1] 6772
```

And the test set:

```
x_test <- x_data[-ind_train, ]
y_test <- y_data[-ind_train]
dim(x_test)
```

```
## [1] 1693   31
```

```
length(y_test)
```

```
## [1] 1693
```

The predictors can be scaled as previously, by demeaning with the values from the train set and dividing by the standard deviation, for each predictor.

```
mean_train <- apply(x_test, 2, mean)
std_train <- apply(x_test, 2, sd)
x_train <- scale(x_train, center = mean_train, scale = std_train)
x_test <- scale(x_test, center = mean_train, scale = std_train)
```

The predictive capacities of the model will be assessed on unseen data using 3-fold cross validation. The folds are assigned randomly:

```
nb_folds <- 3
set.seed(123)
folds <- sample(rep(1:nb_folds, length=nrow(x_train)))
```

Let us define a function constructing the architecture of the model, depending on the regularisation parameter of the L2 regularisation.

```
model_structure_bike_l2 <- function(train_data, l=0.01) {
  model <-
    keras_model_sequential() %>%
    # First hidden layer
    layer_dense(units = 32, activation = "relu",
                input_shape = dim(train_data)[[2]],
                kernel_regularizer = regularizer_l2(l)) %>%
    # Second hidden layer
```

```
    layer_dense(units = 64, activation = "relu",
                kernel_regularizer = regularizer_l2(l)) %>%
    # Third hidden layer
    layer_dense(units = 32, activation = "relu",
                kernel_regularizer = regularizer_l2(l)) %>%
    # Output layer
    layer_dense(units = 1, activation = "sigmoid")

  model %>%
    compile(
      optimizer = "rmsprop",
      loss = "binary_crossentropy",
      metrics = c("accuracy",
                  metric_true_positives(name = "tp"),
                  metric_true_negatives(name = "tn"),
                  metric_false_positives(name = "fp"),
                  metric_false_negatives(name = "fn"))
    )
}
```

Notice that the `metrics` argument of the `compile()` function takes a vector of arguments. Along with the accuracy, the number of true positives, true negatives, false positives, and false negatives will be reported, both for the train and the validation sets at each epoch, for each step of the k-fold cross-validation.

Let us train a model on 300 epochs, with a batch size of 512 observations.

```
num_epochs <- 300
batch_size <- 512

metrics_regul <-
  vector(mode = "list", length = nb_folds)

for(k in 1:nb_folds){
  cat(str_c("\nFold number: ", k, "\n"))

  ind_current <- folds != k
```

```
  # Train set
  x_train_partial <- x_train[which(ind_current),]
  y_train_partial <- y_train[which(ind_current)]

  # Validation set
  x_train_valid <- x_train[which(!ind_current),]
  y_train_valid <- y_train[which(!ind_current)]

  # Building the model
  model_with_regul <- model_structure_bike_l2(x_train_partial, l = 0.01)

  # Training the model on the training set
  history <-
    model_with_regul %>%
    fit(x_train_partial, y_train_partial,
        validation_data = list(x_train_valid, y_train_valid),
        epochs = num_epochs, batch_size = batch_size, verbose = 0)

  # Store loss and metrics
  metrics_regul[[k]] <- history
}
```

For comparison, a model without regularisation can be trained. First, a function that creates the architecture of the model can be defined.

```
model_structure_bike <- function(train_data) {
  model <-
    keras_model_sequential() %>%
    # First hidden layer
    layer_dense(units = 32, activation = "relu",
                input_shape = dim(train_data)[[2]]) %>%
    # Second hidden layer
    layer_dense(units = 64, activation = "relu") %>%
    # Third hidden layer
    layer_dense(units = 32, activation = "relu") %>%
    # Output layer
    layer_dense(units = 1, activation = "sigmoid")

  model %>%
```

```
    compile(
      optimizer = "rmsprop",
      loss = "binary_crossentropy",
      metrics = c("accuracy",
                  metric_true_positives(name = "tp"),
                  metric_true_negatives(name = "tn"),
                  metric_false_positives(name = "fp"),
                  metric_false_negatives(name = "fn"))
    )
}
```

Then, the predictive abilities of the model can be assessed on the validation sets during the 3-fold cross-validation.

```
num_epochs <- 500
batch_size <- 512

metrics_without_regul <-
  vector(mode = "list", length = nb_folds)


for(k in 1:nb_folds){
  cat(str_c("\nFold number: ", k, "\n"))

  ind_current <- folds != k

  # Train set
  x_train_partial <- x_train[which(ind_current),]
  y_train_partial <- y_train[which(ind_current)]

  # Validation set
  x_train_valid <- x_train[which(!ind_current),]
  y_train_valid <- y_train[which(!ind_current)]

  # Building the model
  model <- model_structure_bike(x_train_partial)

  # Training the model on the training set
  history <-
```

```
    model %>%
    fit(x_train_partial, y_train_partial,
        validation_data = list(x_train_valid, y_train_valid),
        epochs = num_epochs, batch_size = batch_size, verbose = 0)

  # Store loss and metrics
  metrics_without_regul[[k]] <- history
}
```

Let us define a function to extract the metric values from the history stored during the k-fold cross-validation process. This function will return the average over the k folds for each epoch, in a tibble.

```
#' @param k_fold_history list with the training history during the k-fold CV
#' @param metric (string) name of the metric to extract from the history
history_to_tibble <- function(k_fold_history, metric){
  map(k_fold_history, "metrics") %>%
    # Metrics on the validation sets
    map(str_c("val_",metric)) %>%
    map_df(~as_tibble(.) %>% mutate(epoch = row_number()),
           .id = "fold") %>%
    mutate(sample = "validation") %>%
    bind_rows(
      map(k_fold_history, "metrics") %>%
        map(metric) %>%
        map_df(~as_tibble(.) %>% mutate(epoch = row_number()),
               .id = "fold") %>%
        mutate(sample = "train")
    ) %>%
    group_by(epoch, sample) %>%
    summarise(!!sym(metric) := mean(value), .groups = "drop") %>%
    mutate(sample = factor(sample, levels = c("validation", "train")))
}
```

Let us apply this function to the list containing the history of the training without regulation, and then with a L2 regularization.

```
df_metrics_regul <-
  history_to_tibble(metrics_without_regul, "accuracy") %>%
```

```
  mutate(type = "Without regularisation") %>%
  bind_rows(
    history_to_tibble(metrics_regul, "accuracy") %>%
      mutate(type = "L2 regularisation (l=0.01)")
  )

df_metrics_regul
```

```
## # A tibble: 1,200 x 4
##    epoch sample     accuracy type
##    <int> <fct>         <dbl> <chr>
##  1     1 train         0.685 Without regularisation
##  2     1 validation    0.737 Without regularisation
##  3     2 train         0.769 Without regularisation
##  4     2 validation    0.789 Without regularisation
##  5     3 train         0.800 Without regularisation
##  6     3 validation    0.807 Without regularisation
##  7     4 train         0.814 Without regularisation
##  8     4 validation    0.816 Without regularisation
##  9     5 train         0.823 Without regularisation
## 10     5 validation    0.822 Without regularisation
## # ... with 1,190 more rows
```

Then we can look at the results:

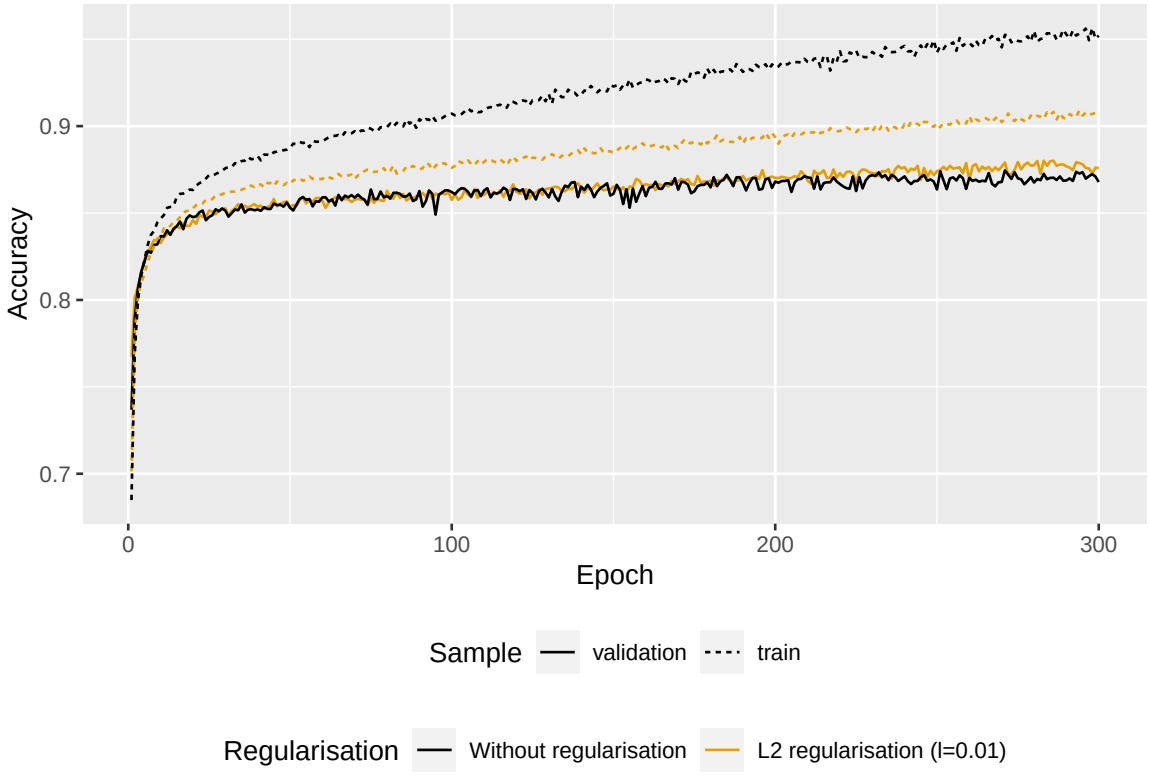

Figure 6.12: Accuracy with or without regularisation.

We note that the model with L2 regularisation is less prone to overfitting than the model where no penalisation on the weights was applied.

6.1.2.3.2 Dropout Consider a layer with its S_1 units: it outputs S_1 values. When dropout is applied to a layer, some of its outputs are randomly set to 0: they are **dropped out**. During the training process, a fraction of the outputs of the layer to which dropout is applied are set to zero: this fraction is called the **dropout rate**.

This corresponds to assigning to each unit of the layer a probability of being dropped out, during each iterations of the training phase. Usually, the droupout rate rate of a hidden layer is set to 0.5.

During the test phase, none of the outputs of the units will be set to 0. Instead, the output of the layer, at the time of the test, is scaled to take into account that the outputs of some of the units have been randomly set to 0.

It is therefore a technique that adds noise, in order to prevent the model from learning certain particularities that are specific to the training data and that cannot then be generalised to the whole data set.

Let us look at a simple example with a small neural network with. Then, we can look at how to perform dropout in R.

Assume that we have only 4 explanatory variables, 2 hidden layers (the first one with 5 unites and the second with 4 unit), and that we want to predict a numerical output variable. Let us apply dropout regularisation only on the second hidden layer. The corresponding architecture of the network is depicted in Figure 6.13.

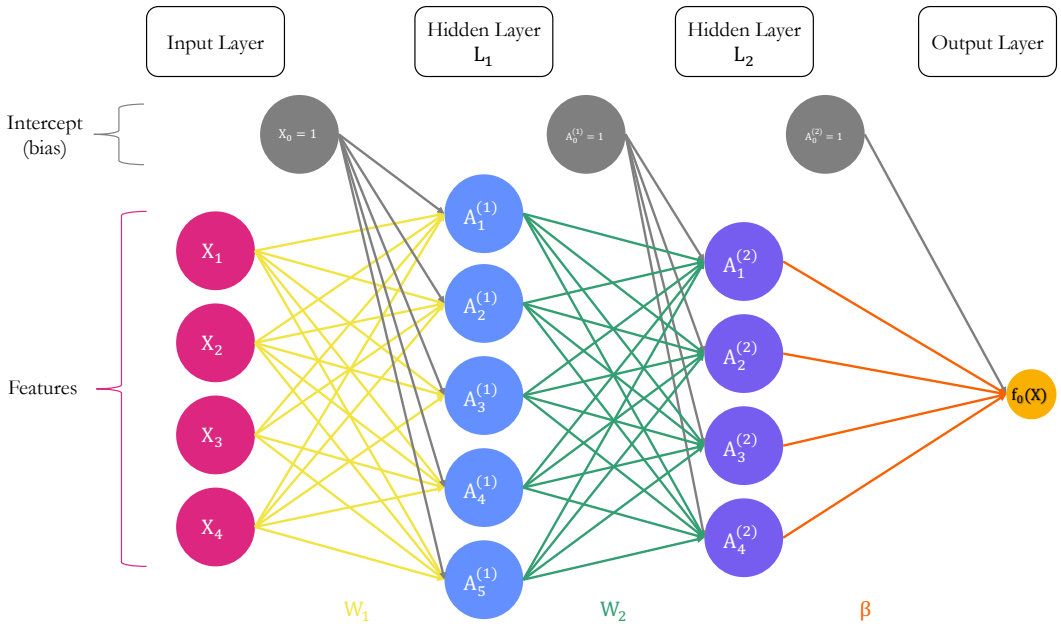

Figure 6.13: Estimated parameters of the model before applying the droupout.

Now, let us say that we apply dropout regularisation to the first and second hidden layers, using a dropout rate of .5. For each layer, on average, .5 of the units will be dropped out at each iteration. Assume that for our current example, the 2nd and 4th unit of the first hidden layer were randomly picked to be dropped out, as well as the first two units of the second hidden layer. The situation is illustrated in 6.14.

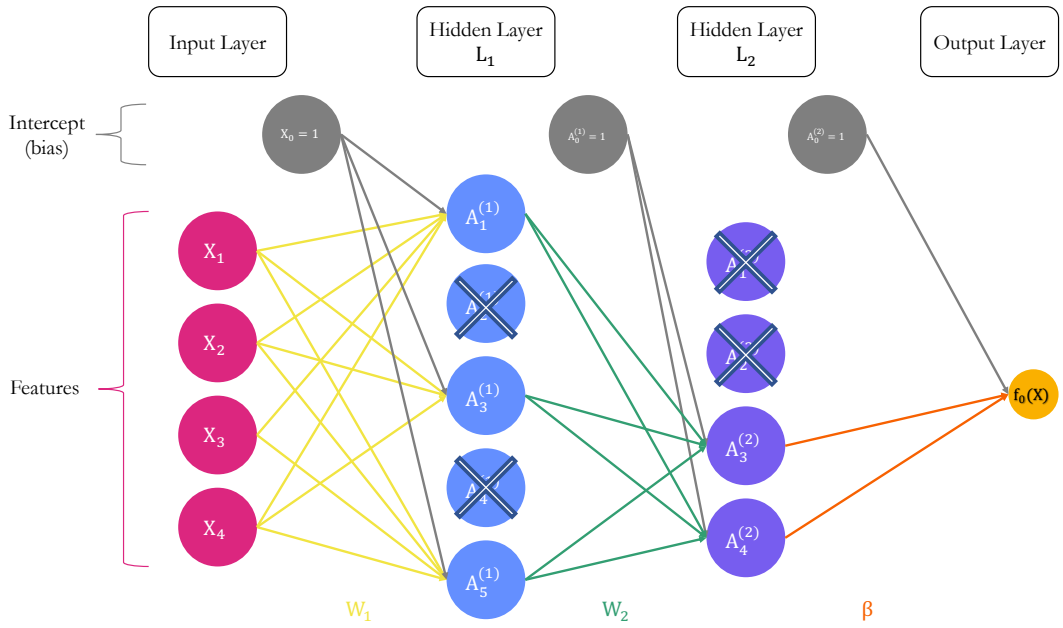

Figure 6.14: Dropout regularisation: each unit of the hidden layer has a probability to be dropped out.

One of the most used techniques to implement this dropout regularisation is called **inverted dropout**. Let us use some R codes to illustrate how it is applied. Let us define the weight matrices as well as the matrix of coefficients β (we assume the following values obtained from previous iteration of the backward propagation algorithm).

Let us simulate some data.

```
set.seed(123)
x <- rnorm(4)
```

Assume that our weight matrix W_1 is:

```
W_1 <- matrix(round(rnorm(20), 2), ncol = 5)
colnames(W_1) <- str_c("L1 Unit ", 1:5)
W_1
```

```
##      L1 Unit 1 L1 Unit 2 L1 Unit 3 L1 Unit 4 L1 Unit 5
## [1,]      0.13     -0.69      0.40      0.50     -1.07
## [2,]      1.72     -0.45      0.11     -1.97     -0.22
## [3,]      0.46      1.22     -0.56      0.70     -1.03
```

```
## [4,]      -1.27       0.36       1.79      -0.47      -0.73
```

The weight matrix W_2:

```
W_2 <- matrix(round(rnorm(20), 2), ncol = 4)
colnames(W_2) <- str_c("L2 Unit ", 1:4)
W_2
```

```
##       L2 Unit 1 L2 Unit 2 L2 Unit 3 L2 Unit 4
## [1,]      -0.63       1.25       0.82      -0.38
## [2,]      -1.69       0.43       0.69      -0.69
## [3,]       0.84      -0.30       0.55      -0.21
## [4,]       0.15       0.90      -0.06      -1.27
## [5,]      -1.14       0.88      -0.31       2.17
```

And the β matrix:

```
beta <- c(0.23, -0.10, 0.13, 0.02)
```

Then the bias:

```
bias_0 <- rnorm(5)
bias_1 <- rnorm(4)
bias_2 <- rnorm(1)

bias_0
```

```
## [1]  1.2079620 -1.1231086 -0.4028848 -0.4666554  0.7799651
```

```
bias_1
```

```
## [1] -0.08336907  0.25331851 -0.02854676 -0.04287046
```

```
bias_2
```

```
## [1] 1.368602
```

Each unit of the first hidden layer makes a linear combination of the inputs, using the current weights.

```
linear_comb_layer_1 <- x %*% W_1 + bias_0
linear_comb_layer_1
```

```
##      L1 Unit 1 L1 Unit 2 L1 Unit 3 L1 Unit 4  L1 Unit 5
## [1,]  1.366655  1.294207 -1.399061 0.7645134 -0.2266276
```

Then, let us say we use a ReLU function for the first hidden layer. The output value of each unit is then:

```
A_k_layer_1 <- relu_f(linear_comb_layer_1)
A_k_layer_1
```

```
## [1] 1.3666550 1.2942066 0.0000000 0.7645134 0.0000000
```

The linear combination made in the second layer, by each unit:

```
linear_comb_layer_2 <- A_k_layer_1 %*% W_2 + bias_1
linear_comb_layer_2
```

```
##      L2 Unit 1 L2 Unit 2 L2 Unit 3 L2 Unit 4
## [1,] -3.016894  3.206208  1.939242 -2.426134
```

Assume we also use a ReLU function for the second hidden layer. The output value of each unit is then:

```
A_k_layer_2 <- relu_f(linear_comb_layer_2)
A_k_layer_2
```

```
## [1] 0.000000 3.206208 1.939242 0.000000
```

Lastly, assuming the identity function as the activation function in the output layer, the prediction is:

```
pred <- A_k_layer_2 %*% beta + bias_2
pred
```

```
##           [,1]
## [1,] 1.300083
```

Now, let us apply inverted dropout regularisation. If we want a dropout rate of .2 (and not .5 as previously), it means that the proportion of the units we want to keep is equal to .8.

```
keep_rate <- .8
```

The choice of which units is made silent or not, is actually made through the weights. A random draw from a Bernoulli distribution with parameter $p = .8$ is made for each of the weights.

```
kept_units_L1 <-
  matrix(rbinom(n=length(W_1), size = 1, prob = keep_rate), ncol = ncol(W_1))
kept_units_L1
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    1    1    1    1
## [2,]    1    0    0    1    0
## [3,]    0    1    1    1    1
## [4,]    1    1    1    1    1
```

The same applies for the second hidden layer:

```
kept_units_L2 <-
  matrix(rbinom(n=length(W_2), size = 1, prob = keep_rate), ncol = ncol(W_2))
kept_units_L2
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    0    1
## [2,]    1    1    1    0
## [3,]    1    1    1    1
## [4,]    0    0    1    1
## [5,]    1    1    1    1
```

The matrix of weights W_1 then becomes:

```
W_1_dropout <- W_1 * kept_units_L1
W_1_dropout
```

```
##       L1 Unit 1 L1 Unit 2 L1 Unit 3 L1 Unit 4 L1 Unit 5
## [1,]       0.13      -0.69       0.40       0.50      -1.07
## [2,]       1.72       0.00       0.00      -1.97       0.00
## [3,]       0.00       1.22      -0.56       0.70      -1.03
## [4,]      -1.27       0.36       1.79      -0.47      -0.73
```

These weight still need some adjustments: they need to be scaled. Recall from earlier that the k-th activation in the second layer writes:

$$A_k^{(2)} = g\left(w_{k_0}^{(2)} + \sum_{s=1}^{S_1} w_{ks}^{(2)} A_s^{(1)}\right)$$

As on average, a proportion of weights equal to the probability for a unit to be kept is forced to be zero, the expected value of $\sum_{s=1}^{S_1} w_{ks}^{(2)} A_s^{(1)}$) will be lowered by the dropout rate (*i.e.*, 20% in our example). A way to avoid this diminished expected value consists in dividing the weights by the theoretical proportion of kept units:

```
W_1_dropout <- W_1_dropout/keep_rate
W_1_dropout
```

```
##       L1 Unit 1 L1 Unit 2 L1 Unit 3 L1 Unit 4 L1 Unit 5
## [1,]     0.1625   -0.8625     0.5000     0.6250   -1.3375
## [2,]     2.1500    0.0000     0.0000    -2.4625    0.0000
## [3,]     0.0000    1.5250    -0.7000     0.8750   -1.2875
## [4,]    -1.5875    0.4500     2.2375    -0.5875   -0.9125
```

The activations then become:

```
A_k_layer_1_dropout <- relu_f(x %*% W_1_dropout + bias_0)
A_k_layer_1_dropout
```

```
## [1] 0.510071 1.769061 0.000000 1.072306 0.000000
```

The matrix of weights W_2 becomes:

```
W_2_dropout <- W_2 * kept_units_L2 / keep_rate
W_2_dropout
```

```
##       L2 Unit 1 L2 Unit 2 L2 Unit 3 L2 Unit 4
## [1,]   -0.7875    1.5625    0.0000   -0.4750
## [2,]   -2.1125    0.5375    0.8625    0.0000
## [3,]    1.0500   -0.3750    0.6875   -0.2625
## [4,]    0.0000    0.0000   -0.0750   -1.5875
## [5,]   -1.4250    1.1000   -0.3875    2.7125
```

The activations of the second layer are then:

```
A_k_layer_2_dropout <- relu_f(A_k_layer_1_dropout %*% W_2_dropout + bias_1)
A_k_layer_2_dropout
```

```
## [1] 0.000000 2.001175 1.416845 0.000000
```

Then the prediction becomes:

```
pred_dropout <- A_k_layer_2_dropout %*% beta + bias_2
pred_dropout
```

```
##          [,1]
## [1,] 1.352675
```

Then a backward pass can be made to compute gradient and then the parameters can be updated. At each iteration of the algorithm, the units that are dropped out are randomly selected. This whole process creates noise in the in the output values of the layers. The model generalises better and thus avoids capturing specificities of the learning sample.

> When making predictions with the trained network, no units are dropped out.

To use dropout regularisation with Keras is set with the `layer_dropout()` function from {keras}. This function needs to be applied **right before** the `layer_dense()` function. The dropout rate is specified through the `rate` argument.

```
model_structure_bike_dropout <- function(train_data) {
  model <-
    keras_model_sequential() %>%
    # First hidden layer
    layer_dense(units = 32, activation = "relu",
                input_shape = dim(train_data)[[2]]) %>%
    # Second hidden layer
    layer_dropout(rate = .4) %>%
    layer_dense(units = 64, activation = "relu") %>%
    # Third hidden layer
    layer_dropout(rate = .4) %>%
    layer_dense(units = 32, activation = "relu") %>%
    # Output layer
    layer_dense(units = 1, activation = "sigmoid")

  model %>%
    compile(
      optimizer = "rmsprop",
      loss = "binary_crossentropy",
      metrics = c("accuracy",
                  metric_true_positives(name = "tp"),
                  metric_true_negatives(name = "tn"),
                  metric_false_positives(name = "fp"),
                  metric_false_negatives(name = "fn"))
    )
}
```

The model can be trained using 3-fold cross validation and a dropout rate of .2 applied to the second and third hidden layers. As earlier, the model is trained over 300 epochs, using a batch size of 512 observations.

```
num_epochs <- 300
batch_size <- 512

metrics_dropout <-
  vector(mode = "list", length = nb_folds)

for(k in 1:nb_folds){
  cat(str_c("\nFold number: ", k, "\n"))
```

```
  ind_current <- folds != k

  # Train set
  x_train_partial <- x_train[which(ind_current),]
  y_train_partial <- y_train[which(ind_current)]

  # Validation set
  x_train_valid <- x_train[which(!ind_current),]
  y_train_valid <- y_train[which(!ind_current)]

  # Building the model
  model_with_dropout <- model_structure_bike_dropout(x_train_partial)

  # Training the model on the training set
  history <-
    model_with_dropout %>%
    fit(x_train_partial, y_train_partial,
        validation_data = list(x_train_valid, y_train_valid),
        epochs = num_epochs, batch_size = batch_size, verbose = 0)

  # Store loss and metrics
  metrics_dropout[[k]] <- history
}
```

Let us apply the function we previously defined (`history_to_tibble()`) to the list containing the history of the training with the dropout regulation

```
df_metrics_regul <-
  history_to_tibble(metrics_without_regul, "accuracy") %>%
  mutate(type = "Without regularisation") %>%
  bind_rows(
    history_to_tibble(metrics_regul, "accuracy") %>%
      mutate(type = "L2 regularisation (l=0.01)")
  ) %>%
  bind_rows(
    history_to_tibble(metrics_dropout, "accuracy") %>%
      mutate(type = "Dropout regularisation (rate=.4)")
  )
```

Then we can look at the results:

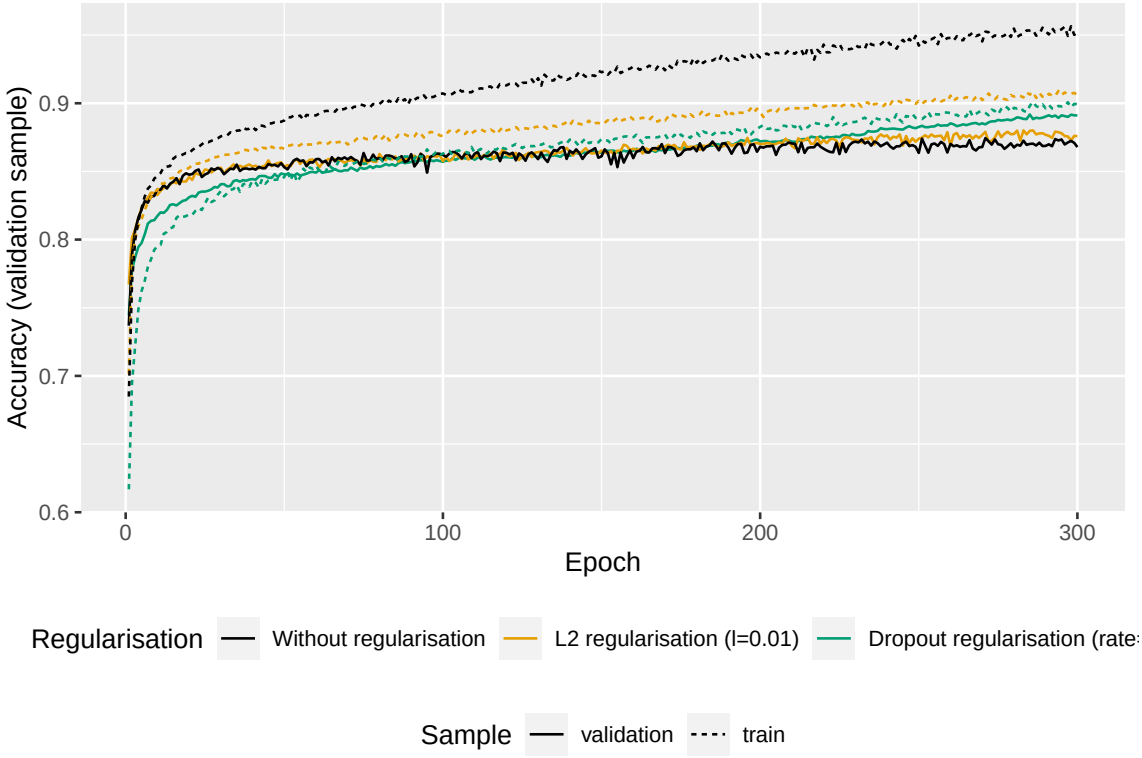

Figure 6.15: Accuracy with or without regularisation (L2 or dropout)

We note that using regularisation techniques, the model is less prone to overfitting.

6.2 Recurrent Neural Networks

This section presents models that can be used to predict sequence data. The focus will be made on **time series** here.

Densely connected networks as those presented earlier have no memory. In contrast, recurrent neural networks will process each training example as a sequence, it will loop over the elements of the sequence before turning to another training example. During the loop, the network will iterate over the elements of the training example (for time series, these elements will be the lagged values).

The input given the the network is a 2D tensor. The first dimension of the tensor is the **time**

step, while the second is made of the predictors. The network loops over the time steps. At each time step ℓ, it considers the input values at ℓ as well as some **state values** at time ℓ. The state values correspond to what the network has seen over the previous iterations of the current sequence (which is initialised to 0 for the first time step).

Each observation/example can be represented as a sequence of L points in time, so $X_t = \{L^L X_t, \ldots, L^2 X_t, LX_t\}$, where L is the lag operator such that $LX_t = X_{t-1}$, $L^2 X_t = X_{t-2}$, and so on. The output Y is the point value to be predicted.

Figure 6.16 illustrates how a recurrent neural network operates, considering a time series with 3 lags accounted for (*i.e.*, a sequence of length 3), where $\mathbf{X} = \{L^3\mathbf{X}_t, L^2\mathbf{X}_t, L\mathbf{X}_t\}$, an output Y to predict, and a hidden-layer sequence $\{As\}_1^S = \{A_1, A_2, \ldots, A_S\}$. Each $\mathbf{X}_\ell$ is a vector $[\mathbf{X}_{\ell,1}, \mathbf{X}_{\ell,2}, \ldots, \mathbf{X}_{\ell,p}]$.

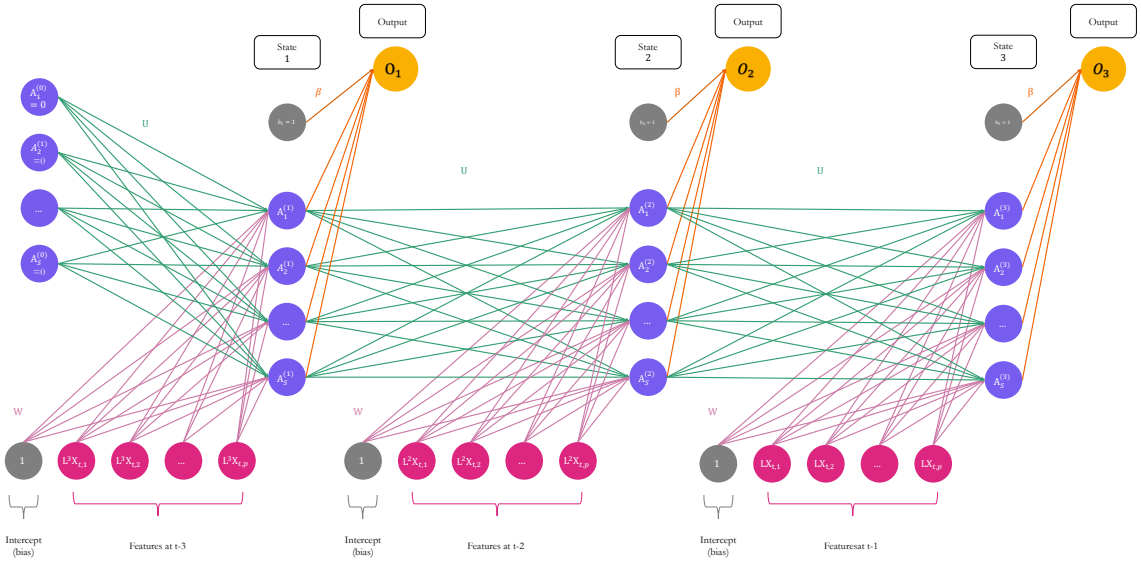

Figure 6.16: A simple recurrent neural network which processes sequences of length 3.

At each time step ℓ, the activations $A_1^{(\ell)}, A_2^{(\ell)}, \ldots, A_S^{(\ell)}$ are updated using:

- a linear combination of the previous activations $A_1^{(\ell-1)}, A_2^{(\ell-1)}, \ldots, A_S^{(\ell-1)}$ weighted by a matrix of weights $\mathbf{U}$
- a linear combination of the inputs $\mathbf{X}_\ell$ weighted by a matrix of weights $\mathbf{W}$
- and a bias (intercept) w_{s_0}.

The s-th activation in time step ℓ, with $k = 1, \ldots, S$ writes:

$$A_s^{(\ell)} = g\left(w_{s_0} + \sum_{j=1}^{p} w_{kj}\mathbf{X}_{\ell,j} + \sum_{s=1}^{S} u_{ks}A_s^{(\ell-1)}\right), \tag{6.2.1}$$

where $A_s^{(0)} = 0$ for all $s = 1, \ldots, S$.

Then, each activations at time step ℓ is fed to the output layer:

$$O_\ell = f\left(\beta_0 + \sum_{s=1}^{S} \beta_s A_s^{(\ell)}\right),$$

where $f(\cdot)$ is an activation function (the identity function for a quantitative response).

Figure 6.17 shows a more compact schematic of the same architecture as that shown in Figure 6.16.

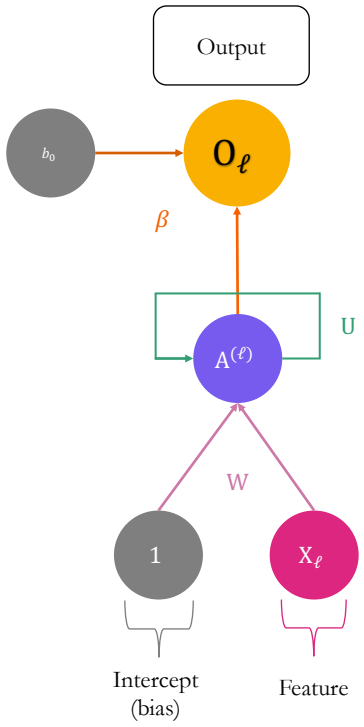

Figure 6.17: The more compact schematic of the simple recurrent neural network which processes sequences of length 3.

For a regression problem, the loss function to be optimised is the residual sum of squares:

$$\sum_{i=1}^{n} \left(y_i - o_i^{(L)}\right)^2,$$

for $i = 1, \ldots, n$, where $o_i^{(L)}$ is the output for the i-th observation (the prediction).

Observe that only the last output O^L is used in the minimisation problem.

6.2.1 Practice with Keras

Let us return to the Seoul bike dataset. Up until now, we have ignored the fact that these data can be processed as time series. Let us follow the example on weather data provided in the 6th chapter of Chollet and Allaire (2018) and adapt it to the Seoul bike data.

First, as noticed by one of the participants during the second hands-on session, the hour variable could be considered in a more intelligent way than was done with the trees. Let us create two variables from it:

- one indicating whether is evening (from 7pm to 9pm), night (from 10pm to 6am), day (7am to 6pm)
- another one indicating whether each observation is recorded at 8 am (morning peek), 6pm (afternoon peak), or outside peak periods.

Other feature engineering could be considered, of course.

```
bike <-
  bike %>%
  mutate(day_moment = case_when(
    hour > 18 & hour <= 22 ~ "evening",
    hour > 22 | hour <= 6 ~ "night",
    hour > 6 & hour <= 18 ~ "day",
    TRUE ~ "problem"
  )) %>%
  mutate(day_peak = case_when(
    hour == 18 ~ "evening",
    hour == 8 ~ "morning",
    TRUE ~ "none"
  ))
```

Let us have a look at the distribution of those new variables:

```
prop.table(table(bike$day_moment))
```

```
##
##       day   evening     night
## 0.5004135 0.1668045 0.3327820
```

```
prop.table(table(bike$day_peak))
```

```
##
##    evening    morning       none
## 0.04170112 0.04170112 0.91659776
```

The data spans from 2017-12-01 to 2018-11-30:

```
range(bike$date)
```

```
## [1] "2017-12-01" "2018-11-30"
```

As the training, validation and test samples will be constituted while keeping the notion of temporality, it is not possible to use the variable indicating the months as a qualitative variable. There are not enough observations to include this variable in the model. The `hour` variable will also be discarded from the features, as two other variables were just created to avoid using the former as a numerical variable. (It could have been possible to just create dummy 11 variables from the `hour` variable instead.)

Let us remove the unnecessary variables from the sample, and in the mean time, let us create dummy variables for all qualitative variables.

```
data_bikes <-
  bike %>%
  select(-date, -y_binary, -month, -seasons, -hour) %>%
  dummy_cols(
    remove_first_dummy = TRUE,
    remove_selected_columns = TRUE)
```

6.2.1.1 Preparing Training, Validation and Test Datasets

The whole dataset can be split into three parts:

- the **train set:** from the first observation to the five-thousandth (`1:5000`)
- the **validation set:** from the five-and-a-half thousandth to the six-thousand-and-five-hundredth (`5001:6500`)
- the **test set:** the remainder

```
ind_train <- 1:5000
ind_valid <- 5001:6500
```

Let us store in an object the index of the column of the target variable in the dataset:

```
ind_target <- which(colnames(data_bikes) == "rented_bike_count")
ind_target
```

```
## [1] 1
```

The observations need to be scaled, as in the previous examples, to make the optimisation problem look for solutions in a less complex dimension.

```
train_data <- data_bikes[ind_train, -ind_target]
```

In order to get a better idea of the model's capabilities when exposed to unseen data, the mean and standard deviation for scaling each explanatory variable should be calculated on the training data.

```
(mean <- apply(train_data, 2, mean))
```

```
##            temperature               humidity             wind_speed
##               7.659700              55.265200               1.864140
##             visibility dew_point_temperature       solar_radiation
##            1334.842400              -1.634220               0.537718
##               rainfall               snowfall                   year
##               0.126340               0.106920            2017.851200
##    holiday_No Holiday           week_day_Tue           week_day_Wed
##               0.942400               0.144000               0.139200
##          week_day_Thu           week_day_Fri           week_day_Sat
##               0.139200               0.145600               0.144000
##          week_day_Sun    day_moment_evening      day_moment_night
##               0.144000               0.166400               0.334200
```

```
##      day_peak_morning          day_peak_none
##              0.041600               0.916800
```

```
(std <- apply(train_data, 2, sd))
```

```
##           temperature              humidity            wind_speed
##            11.1084170            21.2841883             1.0904797
##            visibility dew_point_temperature       solar_radiation
##           639.5248880            12.2944269             0.8582797
##              rainfall              snowfall                  year
##             1.0224565             0.4750508             0.3559268
##    holiday_No Holiday          week_day_Tue          week_day_Wed
##             0.2330088             0.3511248             0.3461897
##          week_day_Thu          week_day_Fri          week_day_Sat
##             0.3461897             0.3527400             0.3511248
##          week_day_Sun    day_moment_evening      day_moment_night
##             0.3511248             0.3724766             0.4717572
##      day_peak_morning          day_peak_none
##             0.1996933             0.2762119
```

It is important to check that no standard deviation is exactly equal to 0, which could be the case if a dummy variable takes a single value in the training set and another one in the validation or the test sets. Any variable for which the standard deviation computed on the training set should therefore be removed from the features.

The data can then be scaled:

```
data_bikes_x <- data_bikes[, -ind_target]
data_bikes_x <- scale(data_bikes_x, center = mean, scale = std)
```

Let us scale the target variable a bit differently, as it is count data:

```
max_target <- max(data_bikes[[ind_target]])
target_bikes <- data_bikes[[ind_target]]/max_target
```

Let us add the target variable:

```
data_bikes <- cbind(target_bikes, data_bikes_x)
ind_target <- 1
```

To update the parameters of the model, we will rely on batches of training data. The samples will be generated on the fly. To do so, the approach provided in the 6th chapter of Chollet and Allaire (2018) will be used. Such an approach is particularly useful when the data at hand is huge and cannot be stored in memory: instead of loading all the training examples, they are loaded only when sampled to be included in a batch.

As previously stated, we will feed to model with 2D tensors. The first dimension will be the time step ℓ while the second dimension will be the matrix of predictors at the corresponding time step ℓ.

Let us say that we want to:

- make predictions 24 hours ahead: we will set a parameter named `delay` to `24`, as the unit of time in the bike dataset is the hour
- rely on the previous 3-days data: we will set a parameter named `lookback` to `3*24`
- sample the observations every `steps=1` time steps.

We will use a batch size of 32.

```
lookback <- 3*24
step <- 1
delay <- 24
batch_size <- 32
```

The function that generates samples expects some indices to specify where in the whole dataset to sample from. This allows us to generate batches of observations for the training sample, for the validation sample, and for the test sample. Note, however, that we will also define another function to generate all the examples from the test sample with another function, to look at the predicted values.

```
#' WARNING: as variable `i_gen` is updated in the
#' global environment you should make sure that no variable
#' with the same name is used in your code.
#' Otherwise, some conflicts may appear.
#' Function from Chollet & Allaire (2018)
#' @param data Data frame with all the observations in rows
#' @param lookback Number of lags
#' @param delay Horizon for the prediction
```

```
#' @param min_index,max_index Index of the first and last
#'                   examples in `data` in which sampling
#'                    is to take place
#' @param shuffle Should the rows of the dataset from which
#'                samples are drawn be randomly swapped before
#'                sampling?
#' @param batch_size Number of samples per batch
#' @param step Period, in time steps, at which samples are drawn
#'             for daily data, if `step=3`, samples are drawn
#'             every 3 days
#' @param ind_target col index of the target variable in `data`
generator <-
  function(data, lookback, delay, min_index, max_index,
           shuffle = FALSE, batch_size = 128, step = 6,
           ind_target) {
    if (is.null(max_index))
      max_index <- nrow(data) - delay - 1

    i_gen <- min_index + lookback
    function() {
      if (shuffle) {
        rows <- sample(c((min_index+lookback):max_index),
                       size = batch_size)
      } else {
        if (i_gen + batch_size >= max_index)
          i_gen <<- min_index + lookback
        rows <- c(i_gen:min(i_gen+batch_size, max_index))
        i_gen <<- i_gen + length(rows)
      }
      samples <- array(0, dim = c(length(rows),
                                  lookback / step,
                                  dim(data)[[-1]]))
      targets <- array(0, dim = c(length(rows)))
      # targets_j <- array(0, dim = c(length(rows)))
      for (j in 1:length(rows)) {
        indices <- seq(rows[[j]] - lookback, rows[[j]],
                       length.out = dim(samples)[[2]])
        samples[j,,] <- data[indices,]
        targets[[j]] <- data[rows[[j]] + delay, ind_target]
```

```
        # targets_j[[j]] <- rows[[j]] + delay
      }
      list(samples, targets)
    }
}
```

The `generator()` function returns a list with two elements:

1. `samples`: the observations sampled, stored in an array of dimensions [`batch_size`, `lookback`, `number_of_predictors`], where `number_of_predictors` is, as you surely already guessed, the number of predictors
2. `targets`: the vector of the corresponding targets for the `batch_size` examples drawn.

Let us define an object that will draw observations from the training set when it will be called during the training of the network:

```
train_gen <- generator(
  data = data_bikes,
  lookback = lookback,
  delay = delay,
  min_index = 1,
  max_index = max(ind_train),
  shuffle = TRUE,
  step = step,
  batch_size = batch_size,
  ind_target = ind_target
)
```

The same can be done for the objects that will draw observations from the validation set, and for the test set:

```
val_gen <- generator(
  data = data_bikes,
  lookback = lookback,
  delay = delay,
  min_index = min(ind_valid),
  max_index = max(ind_valid),
  shuffle = FALSE,
  step = step,
  batch_size = batch_size,
```

```
  ind_target = ind_target
)

test_gen <- generator(
  data = data_bikes,
  lookback = lookback,
  delay = delay,
  min_index = max(ind_valid)+1,
  max_index = NULL,
  shuffle = FALSE,
  step = step,
  batch_size = batch_size,
  ind_target = ind_target
)
```

You may have noticed that for `val_gen()` and `test_gen()`, the argument `shuffle` was set to `FALSE`. We can thus have an idea of how many steps are needed to draw from these functions in order to see the entire validation and entire set, respectively.

```
val_steps <-
  (max(ind_valid)-min(ind_valid)-lookback)/batch_size
val_steps
```

```
## [1] 44.59375
```

```
test_steps <-
  (nrow(data_bikes)-(max(ind_valid)+1)-lookback)/batch_size
test_steps
```

```
## [1] 59.125
```

As mentioned earlier, we can also create a function to create the complete set of tensors from the training sample:

```
generate_data_full <-
  function(data, lookback, delay, min_index, max_index,
           step = 1, ind_target) {
```

```
    rows_min <- max(min_index, lookback)
    rows_max <- max_index-delay
    rows <- rows_min:rows_max
    function(){
      samples <-
        array(0, dim = c(length(rows),
                         lookback / step,
                         dim(data)[[-1]]))
      targets <- array(0, dim = c(length(rows)))
      for (j in 1:length(rows)) {
        indices <-
          seq(rows[[j]] - (lookback-1), rows[[j]],
              length.out = dim(samples)[[2]])
        samples[j,,] <- data[indices,]
        targets[[j]] <- data[rows[[j]]+delay, ind_target]
      }
      list(samples, targets)
    }
  }
```

This `generate_data_full()` works in the same spirit as the previous generation function, except that it does not perform sampling.

The positions of the values in the training sample can be stored:

```
ind_test <- (last(ind_valid)+1):nrow(data_bikes)
```

Let us also create an object that will contain the `generate_data_full()` function and will return a list with two elements:

1. `samples`: the test examples, stored in an array of dimensions [`n_test`,`lookback`, `number_of_predictors`], where `n_test` corresponds to the number of observations in the test sample minus `delay`.
2. `targets`: the corresponding values of the target variable.

```
data_test <-
  generate_data_full(
    data = data_bikes,
    lookback = lookback,
    delay = delay,
```

```
    min_index = ind_test[1],
    max_index = last(ind_test),
    step = 1,
    ind_target = ind_target
  )
```

Let us create the tensors containing the all the observations from the test sample:

```
c(x_data_test, y_data_test) %<-% data_test()
```

There are 1941 observations and 21 predictors, for which the past 72 values are stored (from earliest to latest date).

```
dim(x_data_test)
```

```
## [1] 1941   72   21
```

There are 1941 corresponding target values:

```
dim(y_data_test)
```

```
## [1] 1941
```

6.2.1.2 Naive Benchmark

Assume that the number of bikes rented per hour is a continuous variable (which is debatable, since it is actually count data). Further assume that the number of bikes rented is periodical with a daily period. Then you can assume that the number of bikes rented at the same time the next day will be identical to the number of bikes rented now. Under these asumption, what would the mean absolute error be?

```
#' Computes the average of MAE obtained on batches drawn
#' from the validation sample.
#' The number of batches drawn is `val_steps`
#' Function from Chollet & Allaire (2018)
evaluate_naive_method <- function() {
  batch_maes <- rep(NA, val_steps)
```

```
  for (step in 1:val_steps) {
    # Gen. batches of data from the validation sample
    c(samples, targets) %<-% val_gen()
    # Take the last observed values for each example
    # in the batch, i.e., target_{t-1}
    preds <- samples[, dim(samples)[[2]], ind_target]
    # CMAE using target_{t-1} as the predicted value
    mae <- mean(abs(preds - targets))
    batch_maes[step] <- mae
  }
  # Avg of MAE obtained over the different batches
  mean(batch_maes)
}
batch_mae <- evaluate_naive_method()
batch_mae
```

```
## [1] 0.0748107
```

The average absolute error, under these hypotheses would therefore be equal to 266.03:

```
batch_mae*max_target
```

```
## [1] 266.0269
```

Can a recurrent neural network beat this? Easier said than done.

6.2.1.3 A Densely Connected Model

In a first step, let us try to predict the number of rented bikes with a simple densely connected model. A recurrent neural network will be trained after and the predictive capacities on the validation sets can be compared.

Let us use a single hidden layer with 16 units. The data will be flatten using the `layer_flatten()` function. Hence, the lagged values will be used as predictors, but the model will not consider that the data are sequential. As the target variable is numerical with values in $\mathbb{R}^+$, let us use a ReLU activation in the output layer.

```
model_benchmark <-
  keras_model_sequential() %>%
  layer_flatten(
    input_shape = c(lookback / step, dim(data_bikes)[-1])) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 1, activation = "relu")
```

The loss function that will be used is the MAE.

```
model_benchmark %>% compile(
  optimizer = "rmsprop",
  loss = "mae"
)
```

The model can be trained over 50 epochs. At each epoch, 60 batches of data will be drawn from the generator. The MAE will also be computed on `val_steps`.{R}=44.59375 validation samples drawn from the generator.

```
history_benchmark <-
  model_benchmark %>%
  fit(
    train_gen,
    steps_per_epoch = 60,
    epochs = 50,
    validation_data = val_gen,
    validation_steps = val_steps
  )
```

The results can be saved:

```
model_benchmark %>%
  save_model_hdf5(
    "deep_learning_models/bikes_model_benchmark.h5")
save(history_benchmark,
     file =
       "deep_learning_models/history_benchmark.rda")
```

Let us have a look at the MAE computed both on the training and the validation sets during the training.

```
plot(history_benchmark)
```

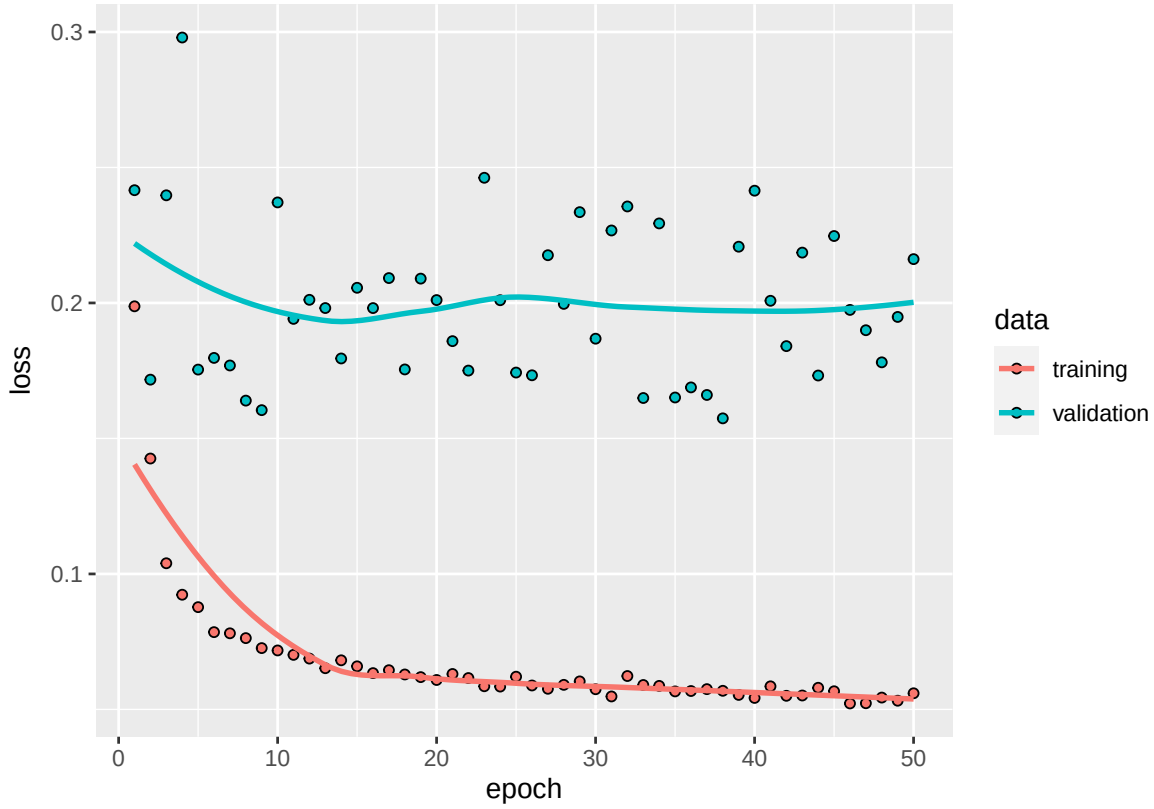

Figure 6.18: Training and validation MAE on bike data, with the densely connected network.

The results obtained with both the training and validation samples are quite poor. In addition, the model overfits.

The MAE on the test set:

```
mae_nn <-
  evaluate(model_benchmark, x=x_data_test, y=y_data_test)
mae_nn*max_target
```

```
##      loss
## 506.5381
```

We have poor performances on the test set here.

The predictions on the test set can be obtained as follows (recall the `x_data_test` was created earlier):

```
preds_nn <- predict(model_benchmark, x = x_data_test)
```

Let us have a look at the predicted values hour by hour over the test sample.

```
ind_in_data <- ind_test[(delay+1):length(ind_test)]
df_test_pred_nn <-
  tibble(obs = bike$rented_bike_count[ind_in_data],
         pred = preds_nn[,1]*max_target) %>%
  mutate(date = bike$date[ind_in_data],
         hour = bike$hour[ind_in_data],
         hm = lubridate::ymd_h(str_c(date, " ", hour))
         ) %>%
  mutate(month = lubridate::month(date))
```

Let us plot the results:

```
df_test_pred_nn %>%
  pivot_longer(cols = c(obs, pred),
               names_to = "type", values_to = "bike") %>%
  ggplot(data = .,
         mapping = aes(x = hm, y = bike,
                       colour = type, linetype = type)) +
  geom_line() +
  facet_wrap(~month, scale = "free", ncol=1) +
  labs(x = "Date", y = "Number of rented bikes") +
  scale_colour_manual(
    NULL,
    values = c("obs" = "#648FFF", "pred" = "#DC267F")) +
  scale_linetype_discrete(NULL)
```

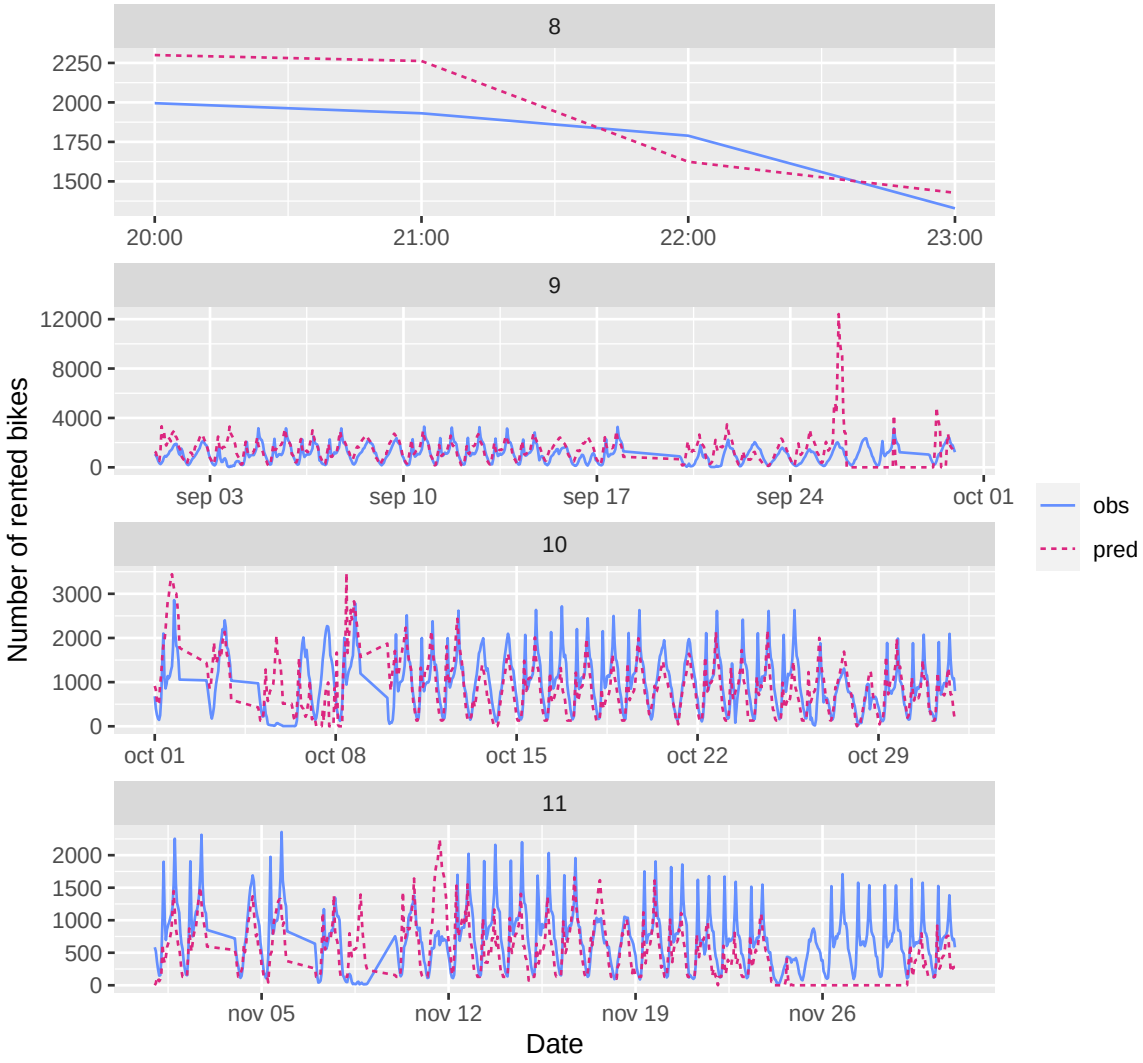

Figure 6.19: Predictions for the densely connected network, by month.

We can also look at the residuals:

```
df_test_pred_nn %>%
  mutate(residuals = obs-pred) %>%
  ggplot(data = ., mapping = aes(x=hm, y = residuals)) +
  geom_line() +
  geom_hline(yintercept = 0, colour = "#648FFF",
             linetype = "dashed") +
```

```
  labs(x = "Date", y = "Residuals")
```

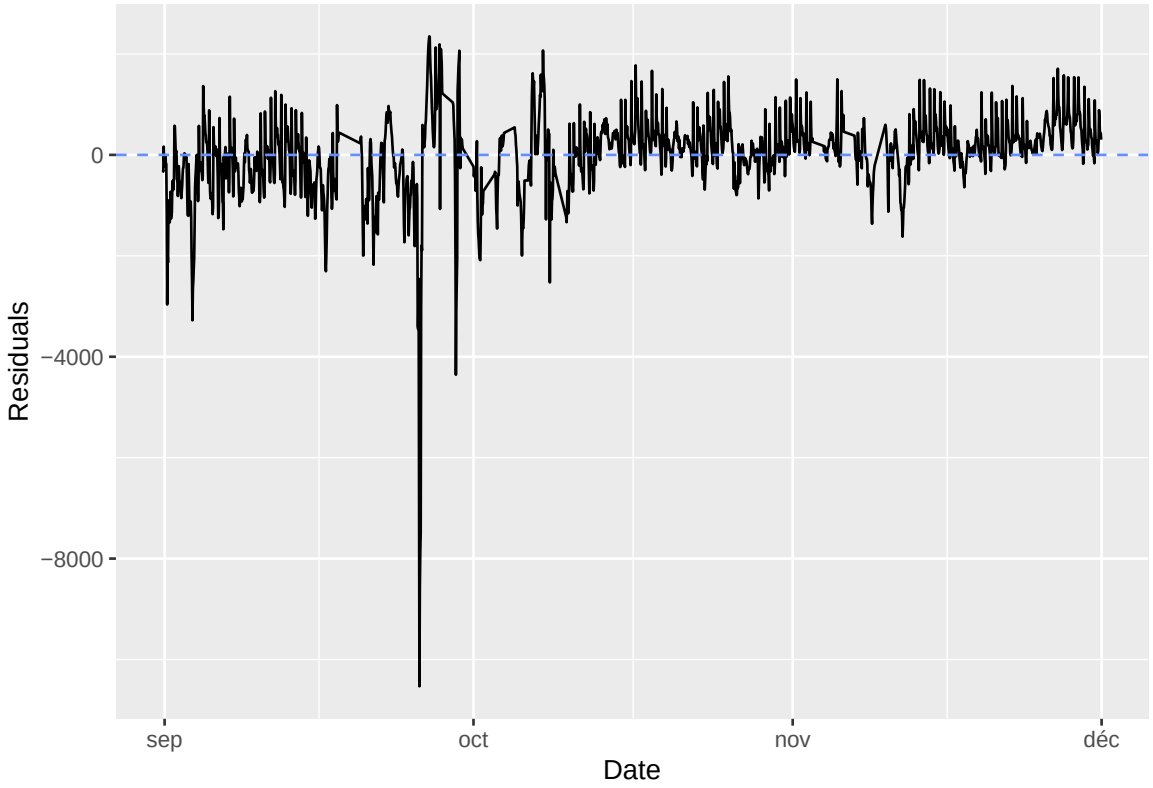

Figure 6.20: Residuals for the densely connected network, estimations made on the test sample.

6.2.1.4 Recurrent Network without Dropout

Now, let us consider the data as sequential and build our first recurrent network. We will use the GRU layer (Gated recurrent unit), as in the example from Chollet and Allaire (2018), and as in Emanuelle Flachaire's course.

```
model_rnn <-
  keras_model_sequential() %>%
  layer_gru(
    units = 16,
    input_shape = list(NULL, dim(data_bikes)[[-1]])) %>%
  layer_dense(units = 1, activation = "relu")
```

The MAE will once again be the loss function being optimised:

```
model_rnn %>%
  compile(
    optimizer = "rmsprop",
    loss = "mae"
  )
```

The model can be trained over 50 epochs, drawing 60 batches per epoch. At each epoch, 44.59375 batch samples will be drawn from the validation set, and the MAE will be computed on those samples.

```
history_rnn <-
  model_rnn %>%
  fit(
    train_gen,
    steps_per_epoch = 60,
    epochs = 50,
    validation_data = val_gen,
    validation_steps = val_steps
  )
```

Once the model is estimated, it can be saved.

```
model_rnn %>% save_model_hdf5("deep_learning_models/model_rnn.h5")
save(history_rnn, file = "deep_learning_models/history_rnn.rda")
```

Let us have a look at the MAE over the epochs.

```
plot(history_rnn)
```

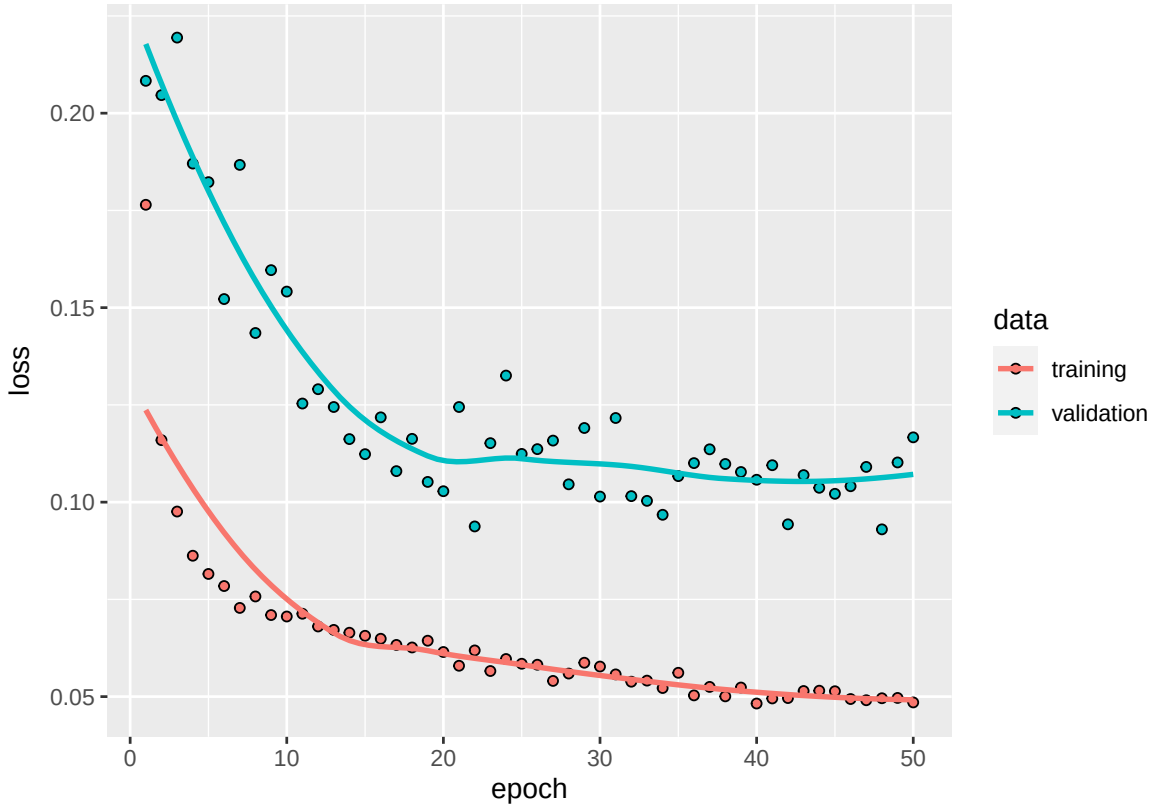

Figure 6.21: Training and validation MAE on bike data, with the first recurrent network.

The performances seem almost correct here (maybe a bit more epochs could lead to lower MAE on the validation set: the algorithm does not seem to have converged yet). We note that the model overfits.

The performances of the model on the test data can be assessed as well:

```
mae_rnn <- evaluate(model_rnn, x=x_data_test, y=y_data_test)
mae_rnn*max_target
```

```
##      loss
## 280.2893
```

Not so bad.

The predictions using that model can be obtained as follows:

```
preds_rnn <- predict(model_rnn, x=x_data_test)
```

Let us put those predicted values in a tibble:

```
ind_in_data <- ind_test[(delay+1):length(ind_test)]
df_test_pred_rnn <-
  tibble(obs = bike$rented_bike_count[ind_in_data],
         pred = preds_rnn[,1]*max_target) %>%
  mutate(date = bike$date[ind_in_data],
         hour = bike$hour[ind_in_data],
         hm = lubridate::ymd_h(str_c(date, " ", hour))
         ) %>%
  mutate(month = lubridate::month(date))
```

The predicted values and the observed values on the test sample can be plotted:

```
df_test_pred_rnn %>%
  pivot_longer(cols = c(obs, pred),
               names_to = "type", values_to = "bike") %>%
  ggplot(data = .,
         mapping = aes(x = hm, y = bike,
                       colour = type, linetype = type)) +
  geom_line() +
  facet_wrap(~month, scale = "free", ncol=1) +
  labs(x = "Date", y = "Number of rented bikes") +
  scale_colour_manual(
    NULL,
    values = c("obs" = "#648FFF", "pred" = "#DC267F")) +
  scale_linetype_discrete(NULL)
```

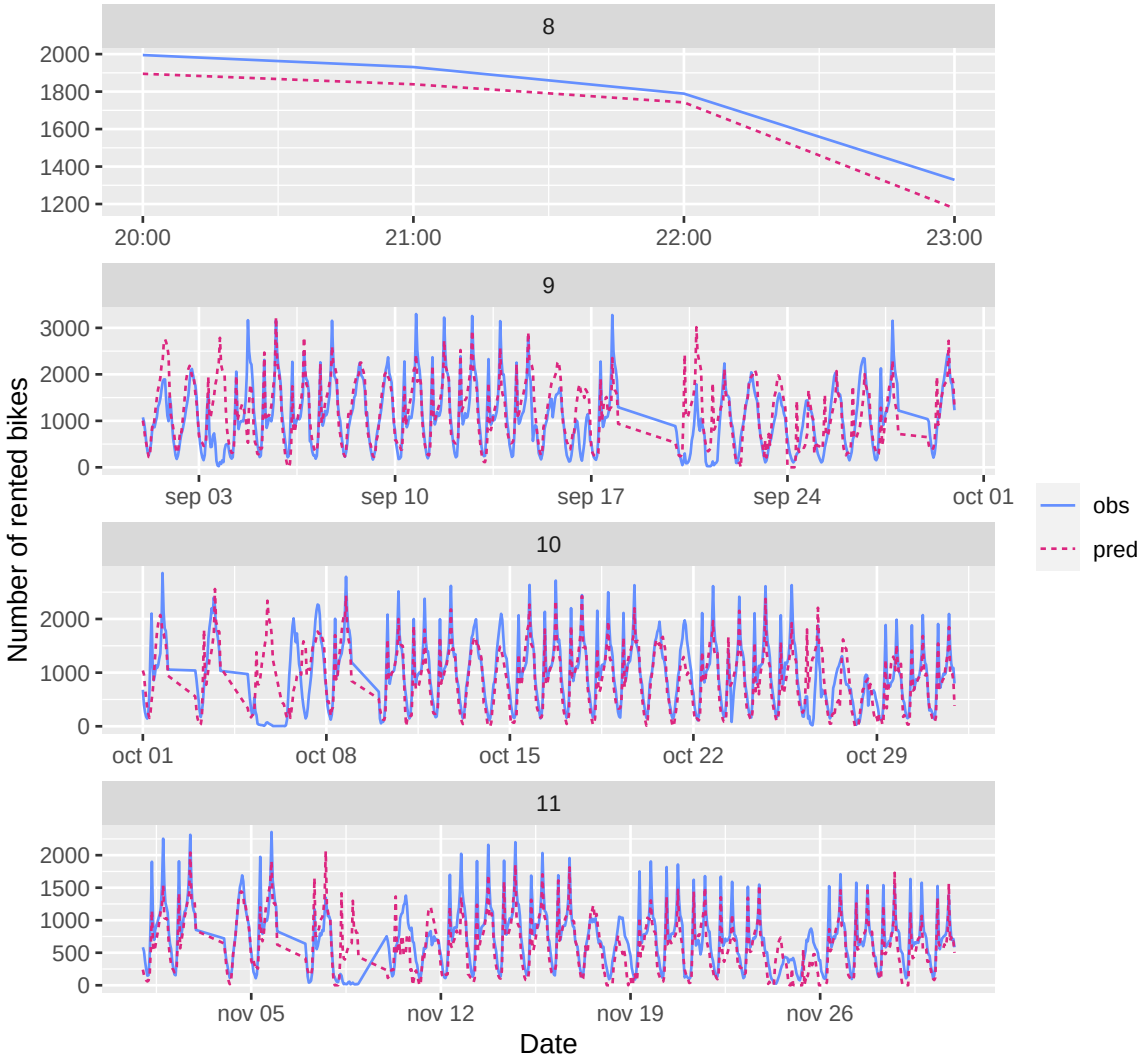

Figure 6.22: Predictions for the first recurrent network, by month.

Let us look at the residuals:

```
df_test_pred_rnn %>%
  mutate(residuals = obs-pred) %>%
  ggplot(data = ., mapping = aes(x=hm, y = residuals)) +
  geom_line() +
  geom_hline(yintercept = 0, colour = "#648FFF",
             linetype = "dashed") +
```

```
  labs(x = "Date", y = "Residuals")
```

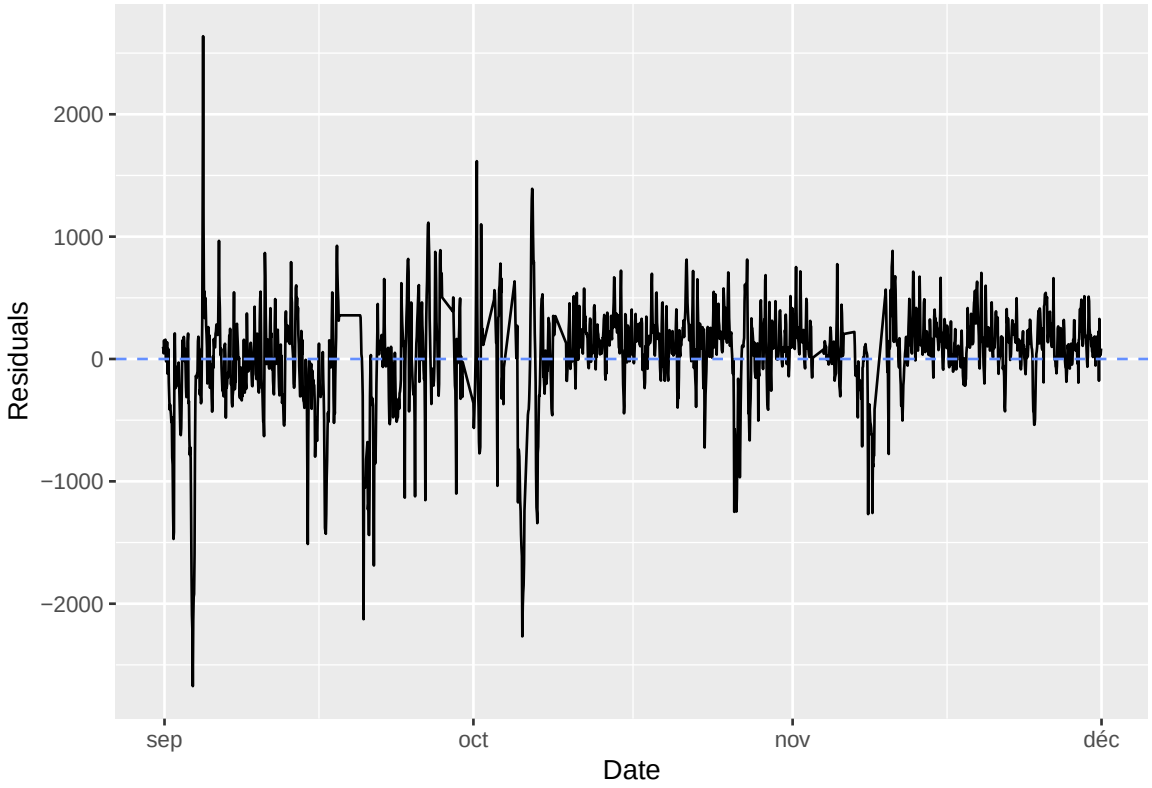

Figure 6.23: Residuals with the first recurrent network.

The model tends to underestimate the number of rented bikes at the end og the period. Having more data to capture seasonnal patterns over the year may help.

6.2.1.5 Recurrent Network with Dropout

We observe some overfitting with the previous recurrent network. Let us try to fight it using dropout. As explained in Chollet and Allaire (2018), with recurrent neural networks, the activations that are temporarily set to 0 should be the same for every time step (to propagate the learning error through time). Such a mechanisme is already in use in Keras. The argument `dropout` of a layer controls the dropout rate applied to this layer. In addition, a dropout rate can be applied to the recurrent activations of a layer (the $A_s^{(\ell-1)}$): a temporally constant mask should be used. The dropout rate of the recurrent activations is controlled by the argument `recurrent_dropout` of the layer.

```
model_rnn_drop <-
  keras_model_sequential() %>%
  layer_lstm(
    units = 16,
    dropout = 0.3,
    recurrent_dropout = 0.4,
    input_shape = list(NULL, dim(data_bikes)[[-1]])) %>%
  layer_dense(units = 1, activation = "relu")
```

Once again, the loss function is the MAE:

```
model_rnn_drop %>% compile(
  optimizer = "rmsprop",
  loss = "mae"
)
```

Networks for which regularisation methods are applied usually need mode epochs to converge. A rule of thumb is to aim about twice the number of epochs.

```
history_rnn_drop <-
  model_rnn_drop %>%
  fit(
    train_gen,
    steps_per_epoch = 60,
    epochs = 100,
    validation_data = val_gen,
    validation_steps = val_steps
  )
```

The trained model can be saved:

```
model_rnn_drop %>%
  save_model_hdf5("deep_learning_models/model_rnn_drop.h5")
save(history_rnn_drop,
     file = "deep_learning_models/history_rnn_drop.rda")
```

Let us have a look at the evolution of the MAE over the training process:

```
plot(history_rnn_drop)
```

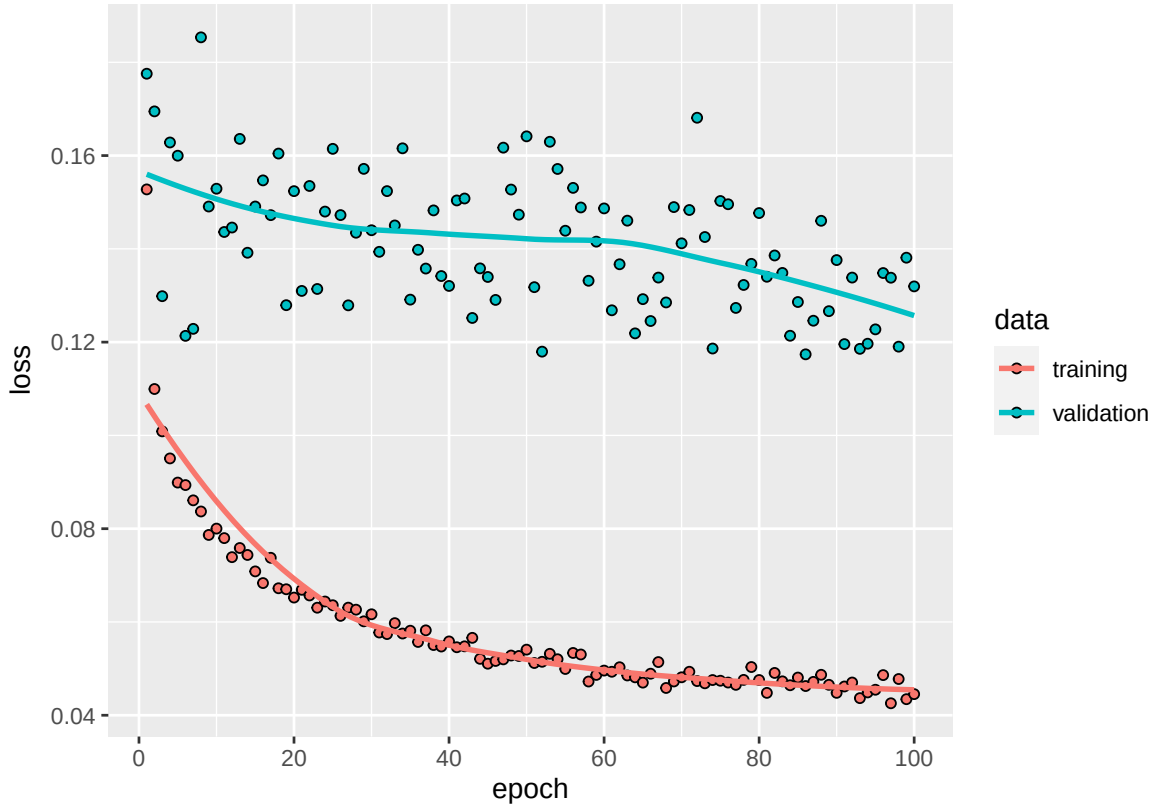

Figure 6.24: Training and validation MAE on bike data, with the regularised recurrent network.

The model still overfits and the predictions are not very good.

The MAE on the test set:

```
mae_rnn_drop <-
  evaluate(model_rnn_drop,
           x=x_data_test, y=y_data_test)
mae_rnn_drop*max_target
```

```
##      loss
## 314.8936
```

Let us make some prediction using the model:

```
preds_rnn_drop <-
  predict(model_rnn_drop, x = x_data_test)
```

Let us put those predicted values in a tibble:

```
ind_in_data <- ind_test[(delay+1):length(ind_test)]
df_test_pred_rnn_drop <-
  tibble(obs = bike$rented_bike_count[ind_in_data],
         pred = preds_rnn_drop[,1]*max_target) %>%
  mutate(date = bike$date[ind_in_data],
         hour = bike$hour[ind_in_data],
         hm = lubridate::ymd_h(str_c(date, " ", hour))
         ) %>%
  mutate(month = lubridate::month(date))
```

The predicted values and the observed values on the test sample can be plotted:

```
df_test_pred_rnn_drop %>%
  pivot_longer(cols = c(obs, pred),
               names_to = "type", values_to = "bike") %>%
  ggplot(data = .,
         mapping = aes(x = hm, y = bike,
                       colour = type, linetype = type)) +
  geom_line() +
  facet_wrap(~month, scale = "free", ncol=1) +
  labs(x = "Date", y = "Number of rented bikes") +
  scale_colour_manual(
    NULL,
    values = c("obs" = "#648FFF", "pred" = "#DC267F")) +
  scale_linetype_discrete(NULL)
```

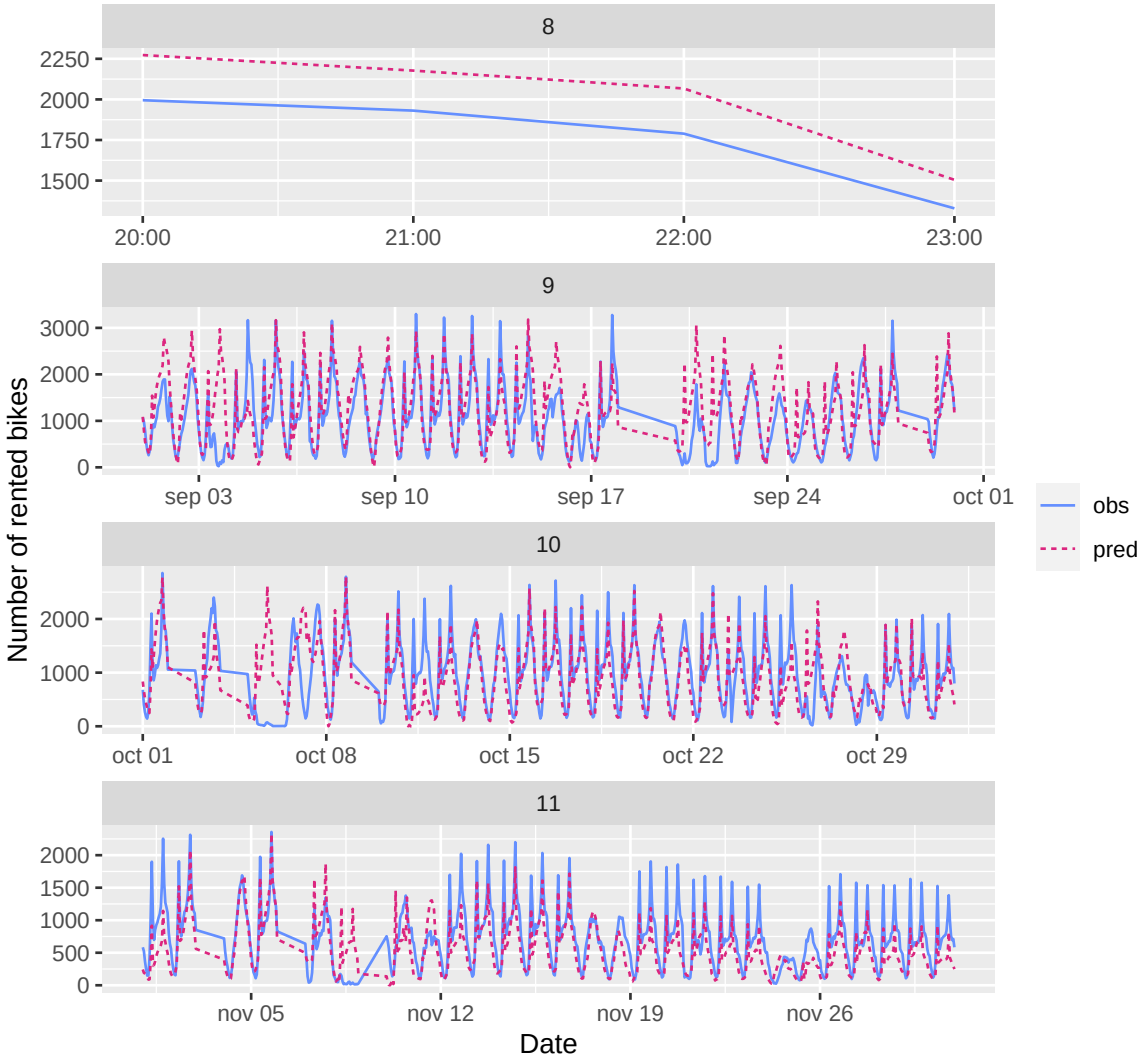

Figure 6.25: Predictions for the regularised recurrent network, by month.

The model still underestimate the needs at the end of the period.

```
df_test_pred_rnn_drop %>%
  mutate(residuals = obs-pred) %>%
  ggplot(data = ., mapping = aes(x=hm, y = residuals)) +
  geom_line() +
  geom_hline(yintercept = 0, colour = "#648FFF",
             linetype = "dashed") +
```

```
labs(x = "Date", y = "Residuals")
```

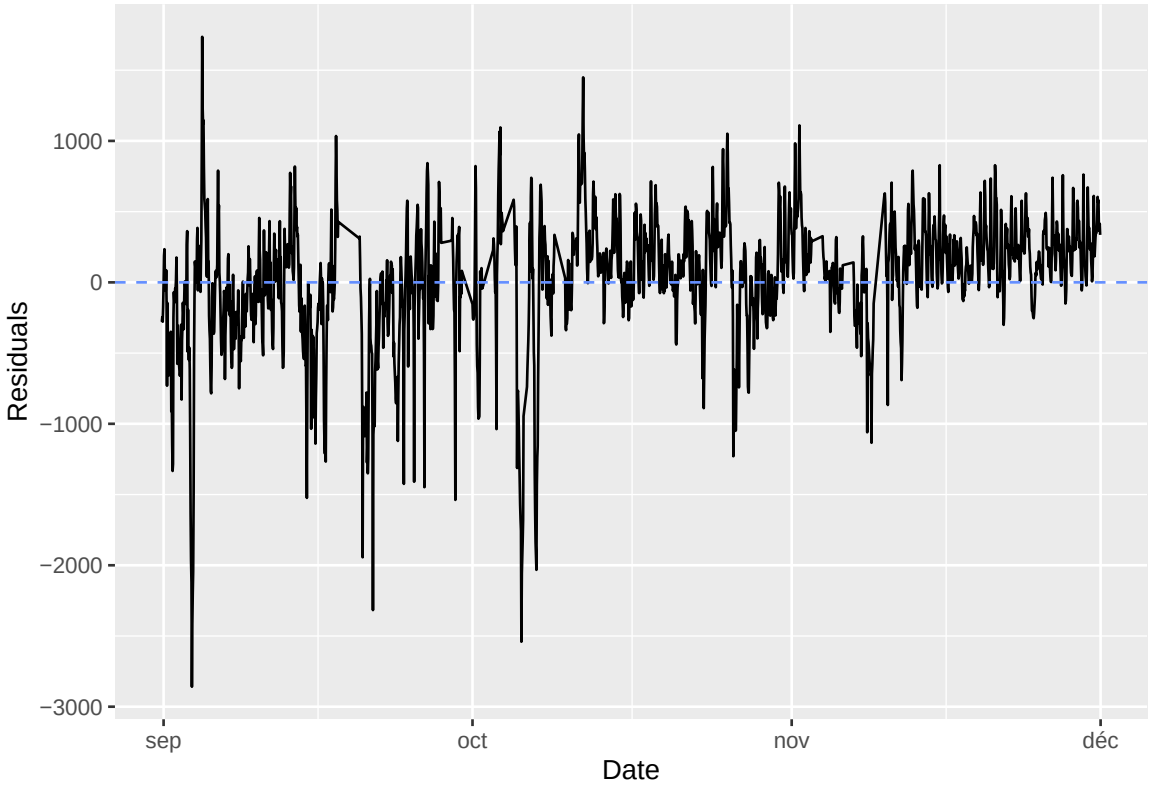

Figure 6.26: Residuals with the regularised recurrent network.

The models that were trained need to be fine tuned to get better results. But to better account for the cyclical components of the data over the year, multiple years are needed.

Now it is your turn: compare the performances obtained with a neural network and those obtained with an ARIMA model.

References

Boehmke, Brad, and Brandon Greenwell. 2019. *Hands-on Machine Learning with r*. Chapman; Hall/CRC. https://doi.org/10.1201/9780367816377.

Breiman, L., J. Friedman, C. J. Stone, and R. A. Olshen. 1984. *Classification and Regression Trees*. Taylor & Francis.

Breiman, Leo. 1996. "Bagging Predictors." *Machine Learning* 24 (2): 123–40. https://doi.org/10.1007/bf00058655.

———. 2001. "Random Forests." *Machine Learning* 45 (1): 5–32. https://doi.org/10.1023/a:1010933404324.

Charpentier, Arthur. 2020. "Act6100 Analyse Des Données En Actuariat." https://github.com/freakonometrics/ACT6100.

Chollet, François, and J. J. Allaire. 2018. *Deep Learning with R*. Manning Publications.

Debeer, Dries, and Carolin Strobl. 2020. "Conditional Permutation Importance Revisited." *BMC Bioinformatics* 21 (1). https://doi.org/10.1186/s12859-020-03622-2.

Demšar, Blaž, Janez AND Zupan. 2021. "Hands-on Training about Overfitting." *PLOS Computational Biology* 17 (3): 1–19. https://doi.org/10.1371/journal.pcbi.1008671.

Goodfellow, Ian, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep Learning*. Vol. 1. MIT press Cambridge.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. 1989. "Multilayer Feedforward Networks Are Universal Approximators." *Neural Networks* 2 (5): 359–66. https://doi.org/10.1016/0893-6080(89)90020-8.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2021. *An Introduction to Statistical Learning*. 2nd ed. Springer.

Karatzoglou, Alexandros, David Meyer, and Kurt Hornik. 2006. "Support Vector Machines inR." *Journal of Statistical Software* 15 (9). https://doi.org/10.18637/jss.v015.i09.

Lecun, Y., L. Bottou, Y. Bengio, and P. Haffner. 1998. "Gradient-Based Learning Applied to Document Recognition." *Proceedings of the IEEE* 86 (11): 2278–2324. https://doi.org/10.1109/5.726791.

Ng, Andrew, and Kian Katanforoosh. 2018. "Deep Learning." http://cs229.stanford.edu/notes/cs229-notes-deep_learning.pdf.

Nutini, Julie. 2015. "Coordinate Descent and Ascent Methods." https://www.cs.ubc.ca/labs/lci/mlrg/slides/mlrg_CD.pdf.

Sathishkumar, V. E., Park Jangwoo, and Cho Yongyun. 2020. "Using Data Mining Techniques for Bike Sharing Demand Prediction in Metropolitan City." *Computer Communications* 153 (March): 353–66. https://doi.org/10.1016/j.comcom.2020.02.007.

Sathishkumar, V. E., and Cho Yongyun. 2020. "A Rule-Based Model for Seoul Bike Sharing Demand Prediction Using Weather Data." *European Journal of Remote Sensing* 53 (sup1): 166–83. https://doi.org/10.1080/22797254.2020.1725789.

Tibshirani, Ryan. 2019. "Convex Optimization Course." https://www.stat.cmu.edu/~ryantibs/convexopt/.

Yeh, I-Cheng, and Che-hui Lien. 2009. "The Comparisons of Data Mining Techniques for the Predictive Accuracy of Probability of Default of Credit Card Clients." *Expert Systems with Applications* 36 (2): 2473–80. https://doi.org/10.1016/j.eswa.2007.12.020.