Machine Learning Supplementary Material

Linus Lach

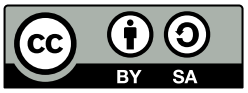
2025-10-01

Contents

Preface

This exercise manuscript supplements the lecture notes provided for Prof. Dr. Yarema Okhrin's lecture *Machine Learning* at the University of Augsburg. Please note that this is still a work in progress and subject to change.

The idea to develop such a manuscript stems from Albert Rapp who also helped me starting this project.



What this manuscript is

The manuscript intends to provide more context to different areas usually neglected in lecture and exercise sessions. The exercise sessions can especially suffer from an imbalance between repeating the theoretical aspects of the lecture and applying the concepts thoroughly. Moreover, this manuscript is comprehensive, containing every exercise and solution presented in the exercise sessions. The solutions will be more detailed than the ones presented in the in-person sessions, which can help if the goal is to partake in the exam.

What this manuscript isn't

This manuscript generally lacks is the interaction between the students and lecturers which I believe is an important aspect of the in-person exercise sessions.

1 Important R concepts

Most exercises involve working with the statistical software R. Since a comprehensive recap of R would go beyond the scope of this lecture, we only review libraries and concepts we frequently need throughout the exercises. For a comprehensive introduction to R, see YARDS by Dr. Albert Rapp.

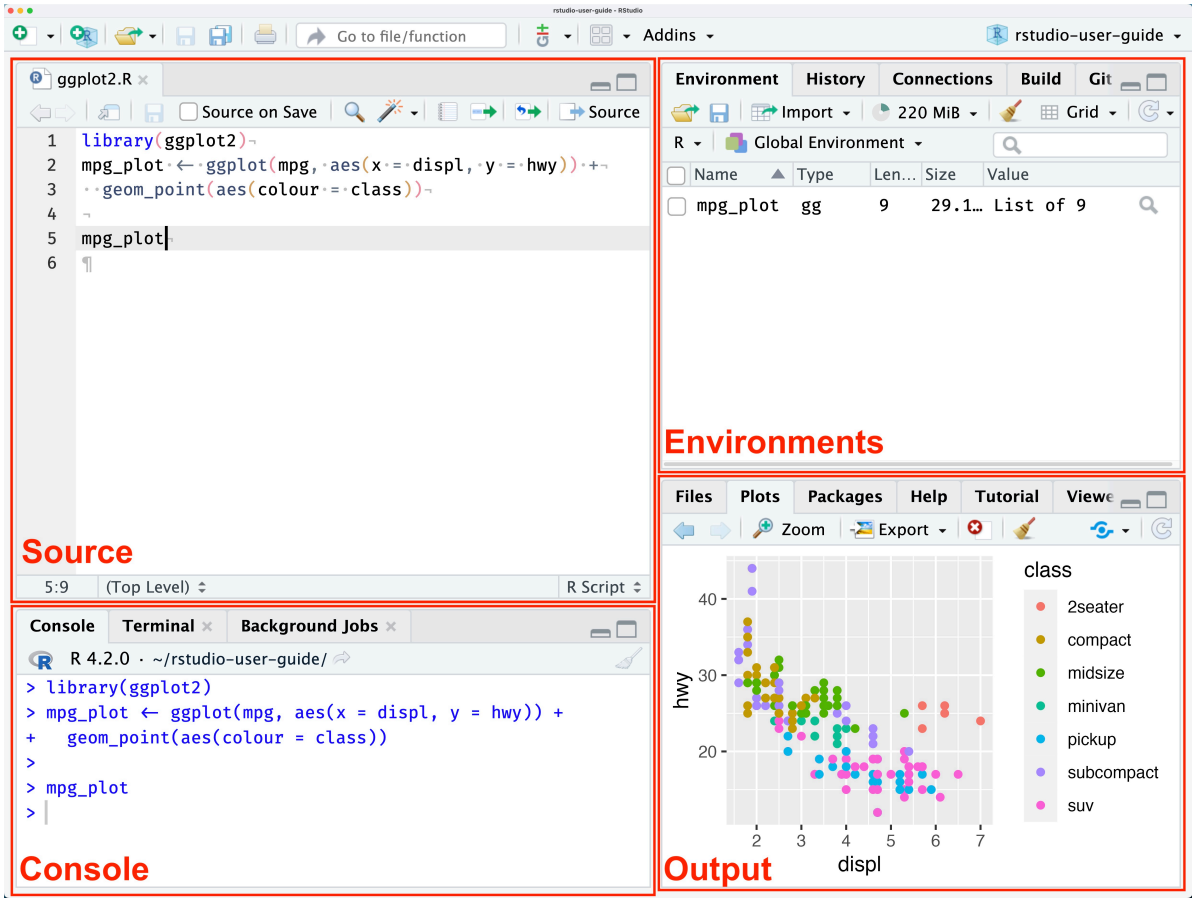
1.1 Setting up R

1.1.1 Installing R, RStudio, and getting started with Quarto

Before starting the revision, we need to ensure that R and RStudio are installed. The installation process for both R and RStudio is straightforward and user-friendly. While R (the programming language) comes with a preinstalled graphical user interface, we will use the integrated development environment RStudio instead, as it looks more appealing and provides some features RGui lacks. It is important to note that in order to get RStudio to work, R needs to be installed first.

- R for Windows can be downloaded here.
- R for MacOS/Unix based systems can be downloaded here
- RStudio can be downloaded here.

After successfully installing R and RStudio, starting the latter should open a window that somewhat looks like the following (cf. RStudio User Guide).



The *Source* pane displays a `.R` file named `ggplot2.R`. While `.R` files are the standard file format for R scripts used for programming in R, we will use Quarto documents (files ending with `.qmd`). Quarto labels itself as the

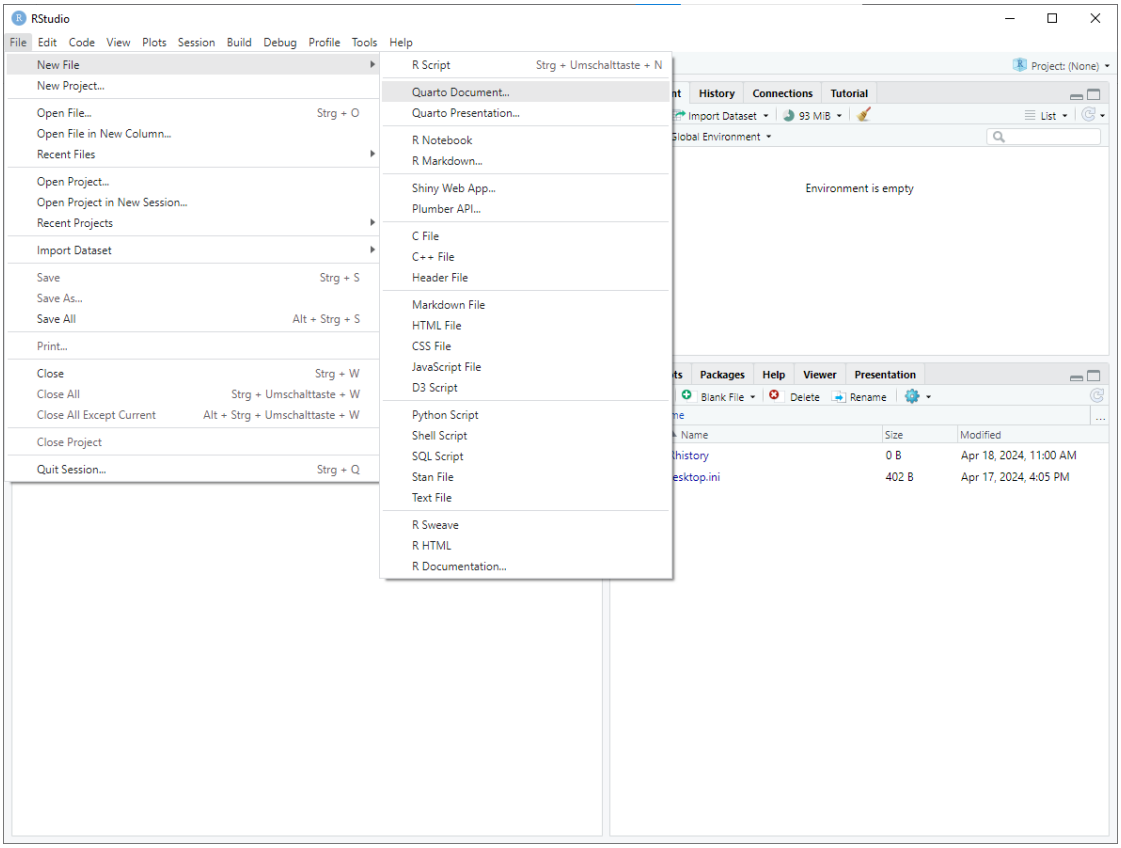
> next-generation version of R Markdown,

meaning that a Quarto document allows

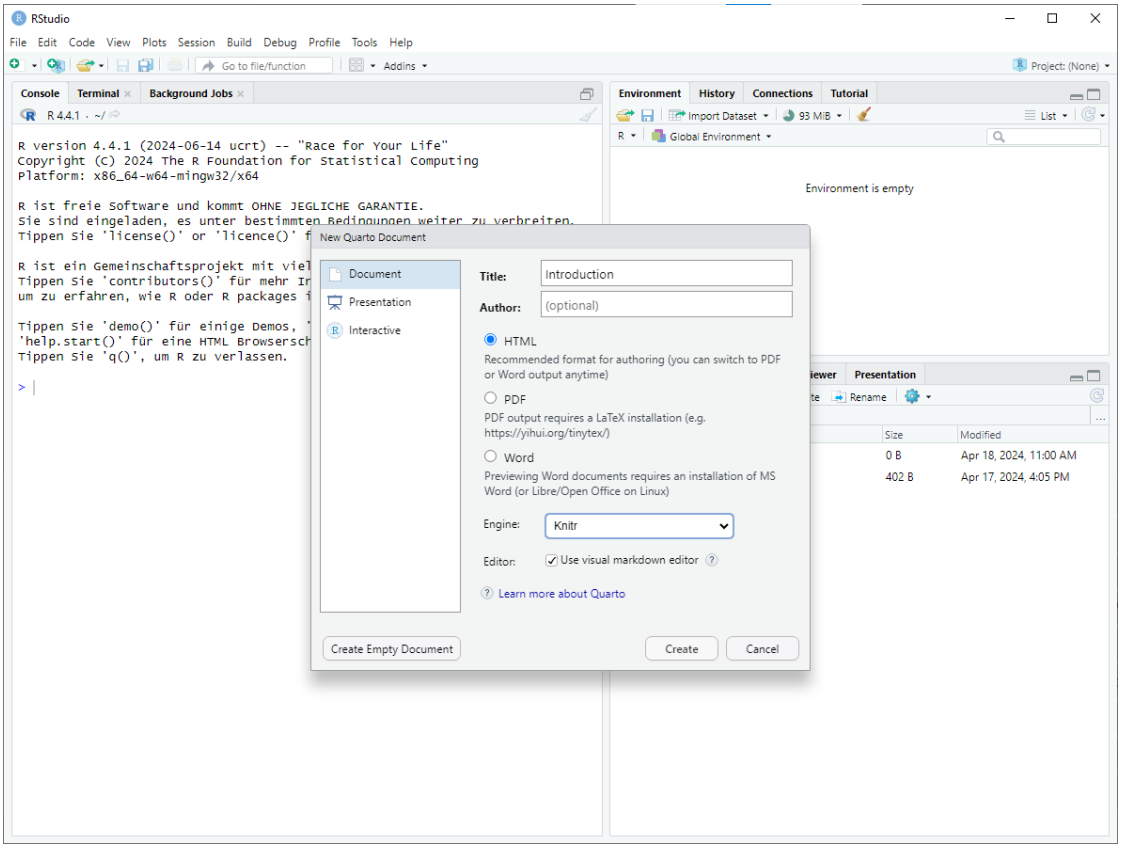
> weaving together narrative text and code to produce elegantly formatted output as documents, web pages, books, and more.

Quarto is preinstalled in RStudio, so creating such documents is relatively simple.

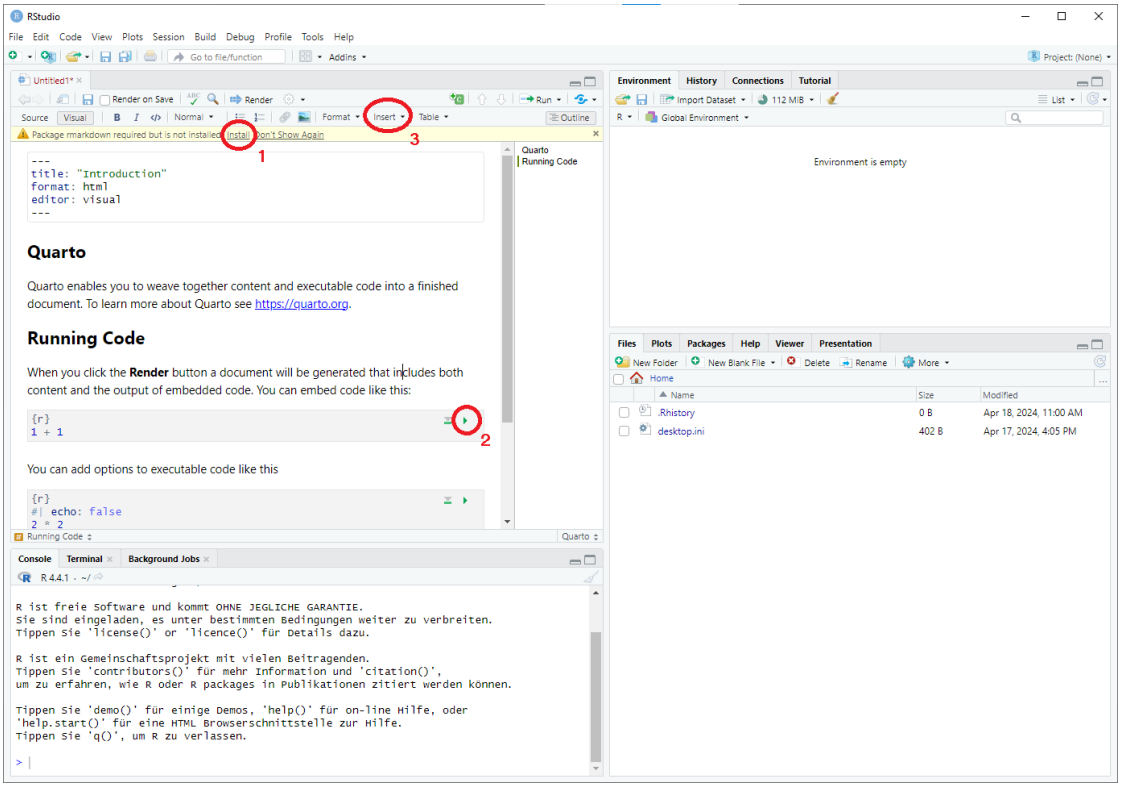
1. click on `New File -> Quarto Document...` in the `File` tab



2. Once the `New Quarto Document` window opens, you can modify the title and specify the output format. For the sake of simplicity, we will use the HTML output format. However, you could also choose PDF if you have a LaTeX installation on your device. Clicking on the `create` button will create a new Quarto Document.



3. The source pane now displays a sample Quarto document that can be modified. You might have to install `rmarkdown` (cf. mark 1). The Quarto document can then be modified by clicking on the respective sections. R Code cells can be executed by clicking on the green arrow (cf. mark 2). To insert new R code cells between paragraphs, click on the `Insert` tab and select `Executable Cell -> R` (cf. mark 3).



This should be enough to get you started with Quarto Documents. For further reading, I recommend the Quarto Guide.

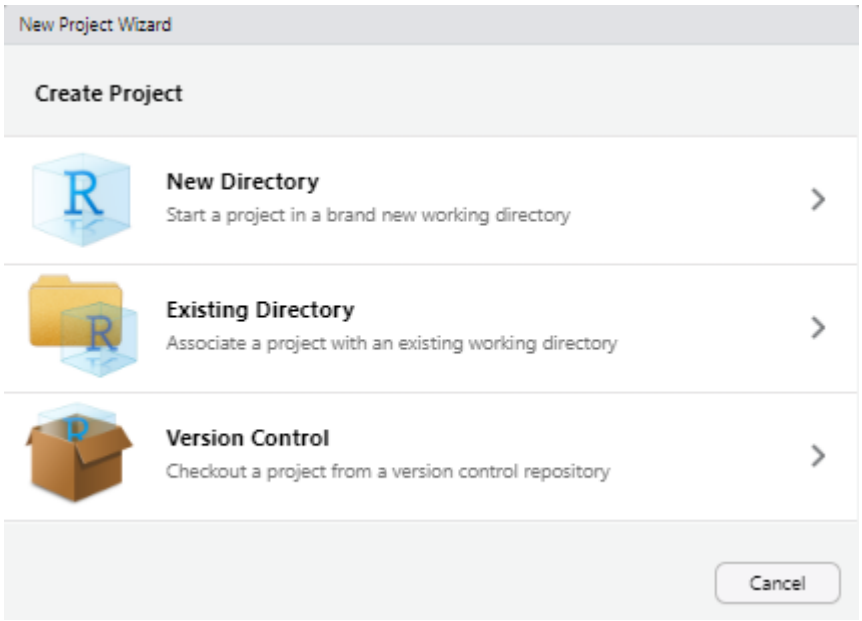
> **Tip**
>
> 1. While solving the (programming) exercises, you can create a new paragraph using `#` for a new chapter to make your document more readable. Additionally, you can simply create a section using `##`.
> 2. Writing down some details on how you approached the programming exercises in a paragraph above or below can help you understand your approach when repeating the exercises later.

1.1.2 Working directories and projects

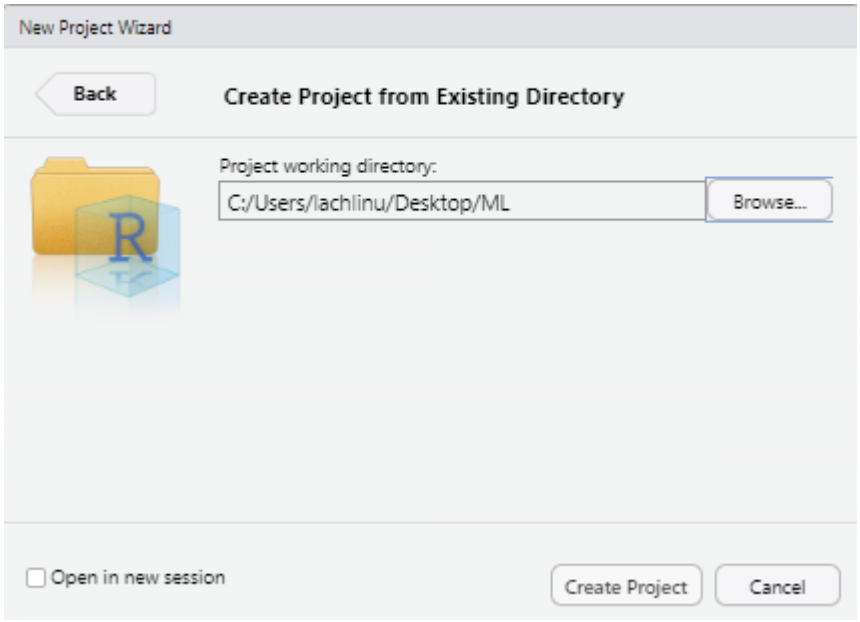
After opening RStudio, execute the `getwd()` command in the console pane, which returns the current working directory. The working directory displays the directory of the R process. For example, if the return value of the `getwd()` command is `C:\Users\lachlinu\Desktop\ML`, then R can access any file in the `ML` directory. One way to change the working directory is using the `setwd()` command, which changes the current working directory. Manually changing

the directory in every `.qmd` document might become tedious after a while, so a more practical alternative is setting up a project. RStudio projects allow the creation of an individual working directory for multiple contexts. For example, you might use R not only for solving Machine Learning exercises but also for your master's thesis. Then, setting up two different projects will help you organize working directories and workspaces for each project individually. To set up a project for this course

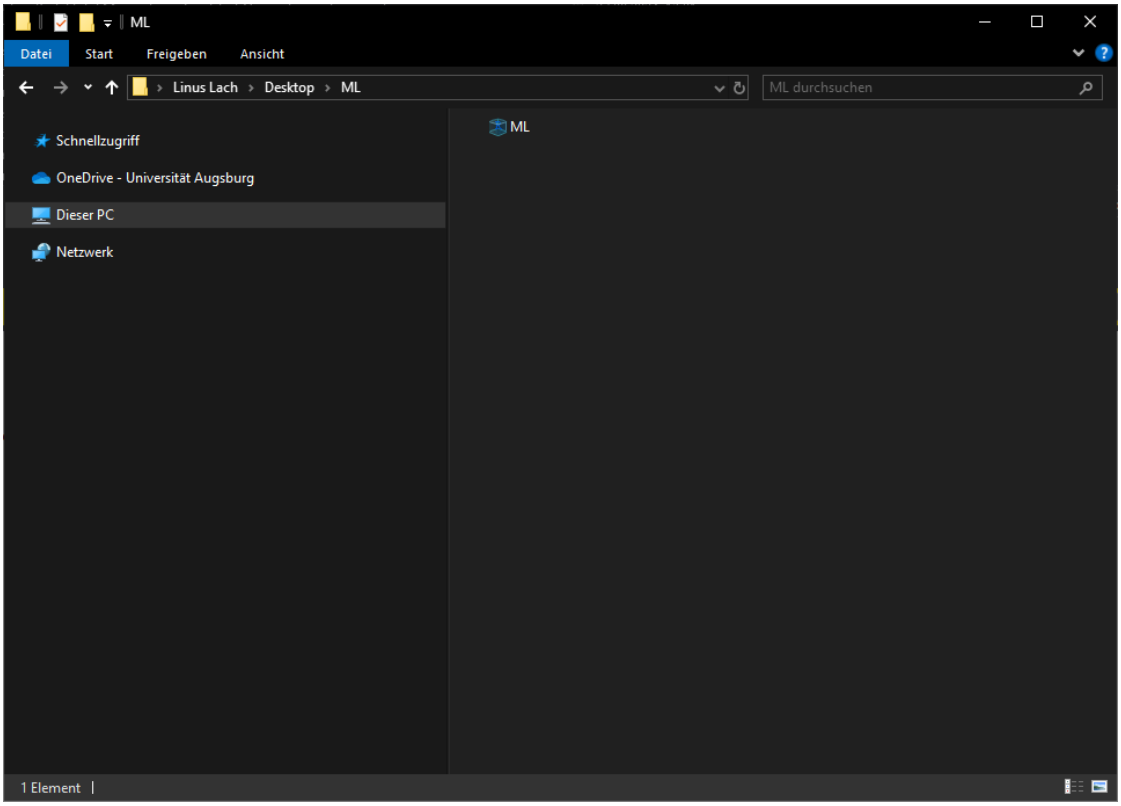
1. Go to the `File` tab and select `New Project...`.



2. Choose `Existing Directory` and navigate to the directory in which you want to create the project in and click the `Create Project` button.



3. You can now open the project by double clicking on the icon which should open a new RStudio window.



Once the project is opened, running the `getwd()` command in the console pane returns its path. Any file in this directory can be directly accessed without specifying the preceding path.

1.2 Help, my code doesn't do what I expect it to do!

Debugging, also known as finding out why code does not behave as expected, plays a substantial role in programming. Even after years of practice, occasional bugs will occur. There are several more or less effective approaches to resolve problems that might occur in the context of this course, with debugging being a key focus.

1. A simple Google search can do wonders. Most questions that arise in the context of this lecture have probably been asked on boards like Stack Overflow. Alternatively, the search results might contain websites like DataCamp or GeeksforGeeks, which will most likely answer your question as well.
2. Using a large language model like ChatGPT. When it comes to debugging or coding questions in general, large language models are helpful. Enter your question as a prompt and the answer will likely help you. However, when using less popular libraries, the

answers might contain hallucinations that make it much more difficult to resolve your problem.

3. The internal `help` function, while it might be considered old-school, is a highly effective troubleshooting approach. It can provide valuable insights when a function doesn't behave as expected. If a function doesn't behave as expected, typing `?function_name` in the console pane opens the respective help page. The `Arguments` section of the help page explains the different input parameters of a function. The `Value` section describes what the function intends to return. Examples of how to use a function are provided in the last section of the help.

4. If each of the steps above fails, you can also ask questions via mail. When reporting a problem via mail, it's crucial to provide some context and the steps you've tried to solve it yourself. This information is invaluable in understanding the issue and providing an effective solution. Also, as part of the learning process, try solving the problems first since that is one of the most essential skill sets to develop.

1.3 Working with datasets

In the context of machine learning, data builds the foundation. Whether you want to predict the weather, future stock prices, or the base rent of a potential rental apartment, without high-quality data, even the most advanced models fail. However, the reality is that data we gather from the web, servers, or spreadsheets is often far from pristine. For example, missing values could be encoded as `NA` (Not Available), `NaN` (Not a Number), `NULL`, or simply by an empty string `" "`. That is why knowing your way around basic data manipulation is essential.

1.3.1 Importing data

To manipulate data, we first need to import it. R has quite a few preinstalled data sets; however, I prefer data sets that are either more complex, related to everyday life, or just more fun to explore. This course will provide most of the data in `.csv` or `.txt` files. Before we can start manipulating data, it's essential to import it correctly. This ensures that the data is in the right format for further analysis.

Consider the Netflix Movies and TV Shows data set, which can be downloaded from the data science community website Kaggle or directly below.

Download Netflix Data

Once you have downloaded the data and placed it in your current working directory, you can import it using the `read.csv` command:

```
data_netflix <- read.csv("netflix_titles.csv")
```

> **Pro Tip**
>
> To maintain structure in your project folder, it is advisable to create a separate directory for the data and import it from there. For example, if the project directory contains a `data` directory with the `netflix_title.csv` file inside, it can be imported using
>
> ```
> data_netflix <- read.csv("data/netflix_titles.csv")
> ```

1.3.2 Essential functions and libraries

One of the most versatile and essential collection of libraries in the context of data science with R is the `{tidyverse}` library, which includes

- `{ggplot}` for creating beautiful graphics,
- `{dplyr}` for data manipulation,
- `{stringr}` for working with strings, and
- `{tibble}` for storing data effectively.

An excellent in-depth introduction to the tidyverse called R for Data Science is freely available online if that piques your interest. This introduction focuses on a few core functions that will be useful throughout the course. As every other library, `{tidyverse}` can be attached using the `library` function once it has been installed:

```
install.packages("tidyverse")
library(tidyverse)
```

Once the library has been added, every function contained is available. For example, the `glimpse` function can be used on the `data_netflix` data set to get a short overview of the data types and contents:

```
glimpse(data_netflix)
```

```
Rows: 8,807
Columns: 12
$ show_id      <chr> "s1", "s2", "s3", "s4", "s5", "s6", "s7", "s8", "s9", "s1~
$ type         <chr> "Movie", "TV Show", "TV Show", "TV Show", "TV Show", "TV ~
$ title        <chr> "Dick Johnson Is Dead", "Blood & Water", "Ganglands", "Ja~
$ director     <chr> "Kirsten Johnson", "", "Julien Leclercq", "", "", "Mike F~
```

```
$ cast         <chr> "", "Ama Qamata, Khosi Ngema, Gail Mabalane, Thabang Mola~
$ country      <chr> "United States", "South Africa", "", "", "India", "", "",~
$ date_added   <chr> "September 25, 2021", "September 24, 2021", "September 24~
$ release_year <int> 2020, 2021, 2021, 2021, 2021, 2021, 2021, 1993, 2021, 202~
$ rating       <chr> "PG-13", "TV-MA", "TV-MA", "TV-MA", "TV-MA", "TV-MA", "PG~
$ duration     <chr> "90 min", "2 Seasons", "1 Season", "1 Season", "2 Seasons~
$ listed_in    <chr> "Documentaries", "International TV Shows, TV Dramas, TV M~
$ description  <chr> "As her father nears the end of his life, filmmaker Kirst~
```

`Rows: 8,807` means that the data set has 8,807 entries, and `Columns: 12` means that the data set has 12 variables, respectively. The first column presents the variable names, their data types, and some initial values, providing a clear structure to the data set. We can already see that except for one variable (`release_year`), every other variable is of type `chr`, which stands for *character* or *string*.

1.3.2.1 Filtering, grouping, and summarizing data sets

Functions frequently encountered while working with data are `filter`, `group_by`, and `summarise`. Let's say we want to find out, according to the data set, how many movies and series were released in each year following 2010. Now, if we were to tackle this problem without the `{tidyverse}` framework, our code might look a little something like this:

```r
netflix_filtered <- data_netflix[data_netflix$release_year > 2010, ]
result <- aggregate(rep(1, nrow(netflix_filtered)),
                    by = list(netflix_filtered$release_year),
                    FUN = sum)
colnames(result) <- c("release_year", "n")
result
```

```
   release_year    n
1          2011  185
2          2012  237
3          2013  288
4          2014  352
5          2015  560
6          2016  902
7          2017 1032
8          2018 1147
9          2019 1030
10         2020  953
11         2021  592
```

or this:

```
netflix_filtered <- data_netflix[data_netflix$release_year > 2010, ]
result <- as.data.frame(table(netflix_filtered$release_year))
colnames(result) <- c("release_year", "n")
result
```

```
   release_year    n
1          2011  185
2          2012  237
3          2013  288
4          2014  352
5          2015  560
6          2016  902
7          2017 1032
8          2018 1147
9          2019 1030
10         2020  953
11         2021  592
```

The first code cell seems much more complicated than the second, yet it returns the same result. However, things can be simplified even more using the `{dplyr}` library that is contained in the `{tidyverse}`:

```
1 netflix_filtered <- data_netflix %>%
2   filter(release_year>2010) %>%
3   group_by(release_year) %>%
4   summarise(n= n())
```

Let us break down the code snippet above:

1. In line `1` we use the pipe operator `%>%`. It is part of the `{magrittr}` package and forwards an object into a function of call expression. Figuratively, a pipe does precisely what is expected: channel an object from one and to another. In this case, the pipe operator `%>%` passes the `data_netflix` data set into the `filter` function.
2. In line `2` the `filter` function selects a subset of a data set that satisfies a given condition. Here, the condition is that the movie or series' release year should be after `2010`, which is indicated by the `>` condition.

> **Note**
>
> Without the pipe operator, the first and second line can be merged into
>
> ```
> filter(data_netflix,release_year>2010)
> ```
>
> however, concatenating multiple functions causes the code to be unreadable and should thus be avoided.

3. Results of the filtering procedure are then passed to the `group_by` function via the pipe operator again. The `group_by` function converts the underlying data set into a grouped one where operations can be performed group-wise. In the third line of the code cell, the `group_by` function is applied to the `release_year` variable, meaning that the data set now contains a group for every release year.

4. This can be seen as a pre-processing step for the `summarise` function applied in the following line. The `summarise` function creates a new data set based on the functions passed as arguments. These functions are applied to every group created by the previous step. In the example above, the function applied is `n()`, which returns the group size. Thus, setting `n=n()` as the argument creates a new column named `n`, which contains the number of samples within each group.

1.3.3 Mutating data sets

Besides filtering, grouping, and summarizing, another important concept is mutating the data, i.e., modifying the content of the data set.

The `mutate` function either creates new columns or modifies existing columns based on the passed column names and functions that are passed. It is helpful for modifying data and their types, creating new variables based on existing ones, and removing unwanted variables. In the example below, the `mutate` function is used to modify the variable `date_added`, create a new variable called `is_show` and delete the variable `type`.

```
data_netflix <- data_netflix %>%
  mutate(date_added = mdy(date_added),
         is_show = if_else(type=="TV Show",TRUE,FALSE),
         type = NULL
         )
```

1. In the first line of the code cell, the data set `data_netflix` is specified to be overwritten by its mutated version. Overwriting a data set is achieved by reassigning the `data_netlifx` object to the output of the pipe concatenation of the following code lines.

2. The original `data_netflix` data set is passed into the mutate function in the second line. Here, the variable `date_written` is overwritten by the output of the `mdy` function with argument `date_added`. The `mdy` function is a function in the `{lubridate}` library that transforms dates stored in strings to `date` objects that are easier to handle. Note that we can directly pass column names into functions as we have previously passed the data set into the `mutate` function using the `%>%` operator.
3. In the third line, a new variable `is_show` is created, which takes the value `TRUE`, if the type of an entry in the data set is `"TV Show"` and `FALSE` if it is not. The `if_else` function achieves this.
4. Setting the `type` variable to `NULL` effectively removes it from the data set.

> **Note**
>
> Assigning values in functions is achieved by using the `=` symbol. Assigning new variables outside of functions can also be done with the `=` symbol, but it is rarely used and except for some pathological cases there is no difference. However, most R users prefer assigning environment variables using `<-` which does not work in function calls.

> **Pro Tip**
>
> In the previous code cell a line break was added after the `%>%` and each argument in the `mutate` function for readability purposes. The code also works without adding the line breaks, but it can get messy fast:
>
> ```
> data_netflix <- data_netflix %>% mutate(date_added = mdy(date_added),...)
> ```

1.3.4 Factor Variables

An important data type that can handle both ordinal (data with some notion of order) and nominal data are so-called `factor` variables.

Consider the following toy data set containing seven people with corresponding age groups and eye colors.

```
data_example <- tibble(
 names = c("Alice", "Bob", "Charlie", "Diana", "Eve", "Frank", "Grace"),
 age_groups = c("18-25", "<18", "26-35", "36-45", "18-25", "60+", "26-35"),
 eye_color = c("Blue", "Brown", "Green", "Hazel", "Brown", "Blue", "Green")
)

data_example
```

```
# A tibble: 7 x 3
  names   age_groups eye_color
  <chr>   <chr>      <chr>
1 Alice   18-25      Blue
2 Bob     <18        Brown
3 Charlie 26-35      Green
4 Diana   36-45      Hazel
5 Eve     18-25      Brown
6 Frank   60+        Blue
7 Grace   26-35      Green
```

Since the variable `age_group` only specifies a range of ages, it does not make sense to encode them as integers rather than ordinal variables. The' mutate' function can encode age groups as ordinal variables. This involves setting the `age_groups` variable to a factor with levels and labels. Levels specify the order of the values, and labels can be used to rename these categories.

```
data_example <- data_example %>%
  mutate(
    age_groups = factor(
      age_groups,
      levels = c("<18", "18-25", "26-35", "36-45", "60+"),
      labels = c("child","adult","adult","adult","senior"),
      ordered = TRUE
    )
  )
```

1. Similar to the previous example, we should specify that we overwrite the `data_example` data set with a mutated version.
2. The `mutate` function is applied to the `age_groups` variable.
3. Setting `age_groups = factor(age_groups, ...)` converts the `age_groups` column into a (so far unordered) factor, allowing for specific levels (categories) and labels.
4. `levels = c("<18", "18-25",...)` specifies the predefined levels for the age groups.
5. `ordered=TRUE` specifies that the age groups are ordered according to the specified levels.
6. Last but not least, `labels = c("child", "adult", ...)` specifies the labels that replace the numeric age groups. For instance, <18 is labeled as `"child"`, the ranges 18-25, 26-35, and 36-45 are labeled as `"adult"`, and 60+ is labeled as `"senior"`.

Similarly, the variable `eye_color` can also be converted to a nominal factor variable:

```
data_example <- data_example %>%
  mutate(
    eye_color = factor(eye_color)
  )
```

To confirm that the variable `age_group` is indeed ordered, we can print the levels in ascending. The `levels`-function does exactly that:

```
levels(data_example$age_groups)
```

```
[1] "child"  "adult"  "senior"
```

Note, that we used the `$` selector since the `levels`-function can't extract the levels from a list of factors:

```
data_example %>%
  select(age_groups) %>%
  levels()
```

```
NULL
```

By checking the types of the respective objects, we can confirm our suspicion:

```
data_example %>%
  select(age_groups) %>%
  typeof()
```

```
[1] "list"
```

```
data_example$age_groups %>%
  typeof()
```

```
[1] "integer"
```

To extract the raw values from the list, we can use the `pluck()`-function:

```
data_example %>%
  select(age_groups) %>%
  pluck("age_groups") %>%
  typeof()
```

```
[1] "integer"
```

By passing the column name, either as a string or the position as an integer (2 in this case), we can extract the raw values. Checking the data type subsequently confirms that we can now apply the `levels()` function:

```
data_example %>%
  select(age_groups) %>%
  pluck("age_groups") %>%
  levels()
```

```
[1] "child"  "adult"  "senior"
```

1.4 Visualizing data with ggplot

Another critical aspect of data science and machine learning is graphical storytelling. Describing an algorithm strictly using mathematical notation or exploring a data set using descriptive and inductive statistics alone can make it challenging to understand the message. While R offers some base functions for creating graphics, this course primarily uses the library `{ggplot2}`. A comprehensive introduction to `{ggplot2}` can be found in Hadley Wickham's book Elegant Graphics for Data Analysis. A short summary can be found below.

For the following example, we will use the `netflix_filtered` data set (see Section 1.3.2.1)

A graphic created with `{ggplot2}` consists of the following three base components:

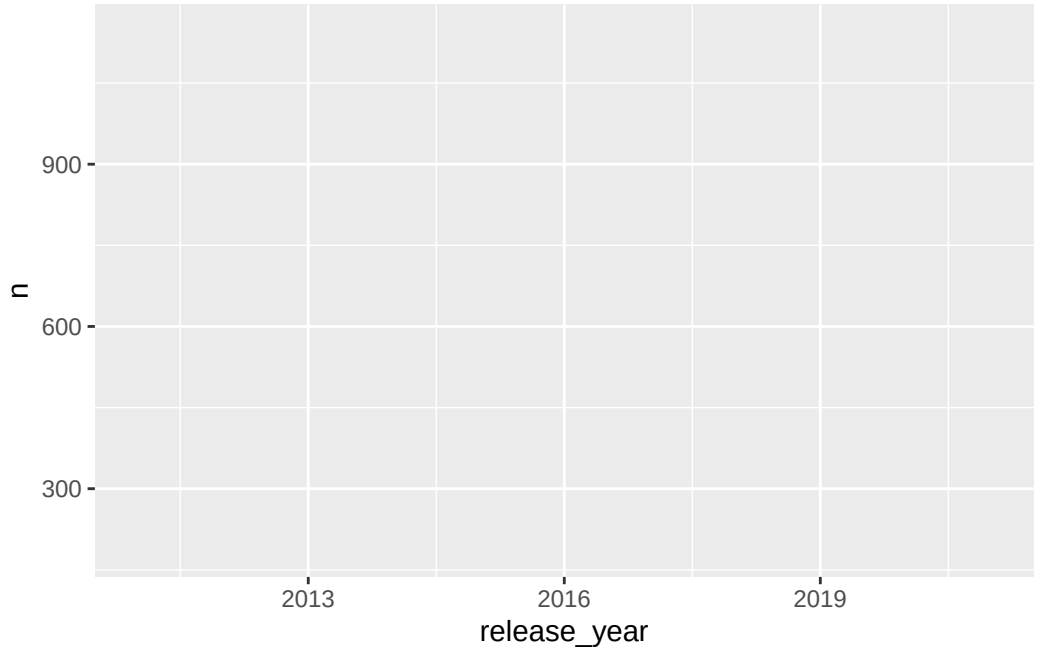
1. The data itself.

   ```
   ggplot(data = netflix_filtered)
   ```

Note, that the plot does not show any axis, ticks, and variables.

2. A set of *aesthetics mappings* that describe how variables in the data are mapped to visual properties.

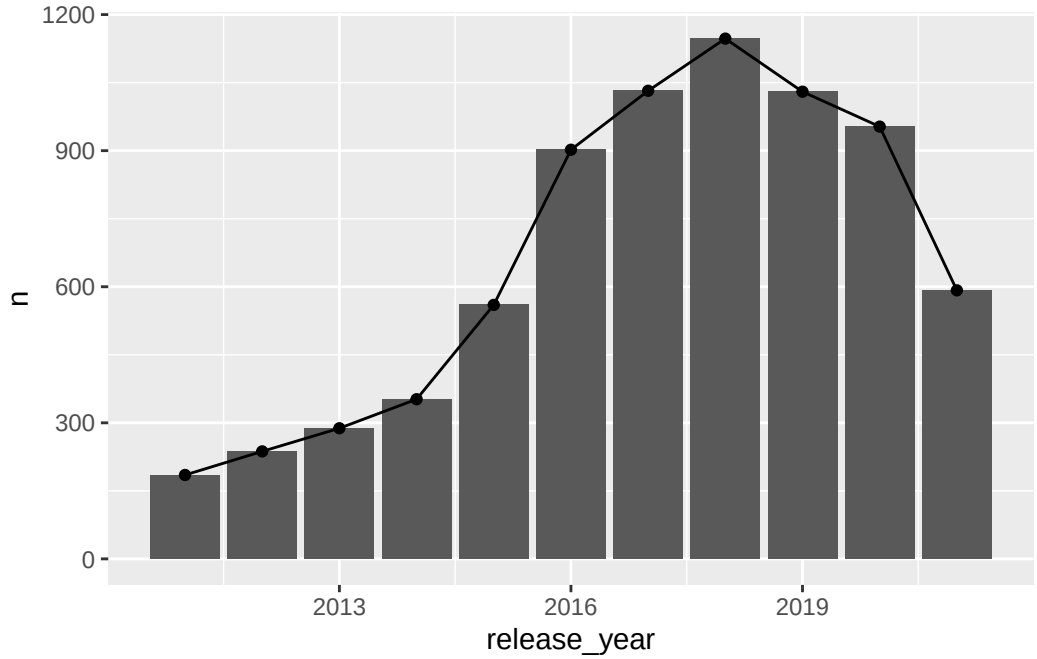
```
ggplot(aes(x=release_year, y=n), data = netflix_filtered)
```



Using the `aes` function, we have specified that the release year should be mapped to the x-axis, and n to the y-axis.

3. Lastly, the *geom*-layer (component) describes how each observation in the data set is represented.

```
ggplot(aes(x=release_year, y=n), data = netflix_filtered)+
  geom_col()+
  geom_point()+
  geom_line()
```



Compared to the previous two code cells, a lot is going on here. So, let us break it down.

1. The plus at the end of line `1` is used to **add** another layer.
2. `geom_col` adds a column chart to the canvas, creating columns starting at `0` and ending at n. Then, `+` indicates that another layer is added.
3. `geom_point` represents the data as points on the plane, i.e., an x and y-coordinate. The `+` indicates that yet another layer is added afterward.
4. Lastly, the `geom_line` function adds a line connecting each data point with the one following.

Pro Tips

1. As before, the data set can also directly be piped into the `ggplot` function:

```
netflix_filtered %>%
ggplot(aes(x=release_year, y=n))+
 geom_col()+
 geom_point()+
 geom_line()
```

2. By changing the order of the layers, you can specify which layer should be added first and last. In this example, since `geom_col` was added first and every other layer is placed on top of the column plot.

There are a lot more functions and settings that can be applied to each function. A selection of those is discussed in the exercises.

1.5 Exercises

Throughout the exercises, we will work with the Credit Card Customers data set that can either be downloaded using the provided link or the button below.

Download BankChurners

The data set consists of 10,127 entries that represent individual customers of a bank including but not limited to their age, salary, credit card limit, and credit card category.

1.5.1 Statistical Data Exploration and Manipulation

We will start by getting a feeling for the data and performing some basic data manipulation steps.

Exercise 1.1. Import the data set and use the `glimpse` function to generate a summary of the data set.

Exercise 1.2. Assume that the data set has been imported and saved as an object called `credit_info`. Explain the following code snippet both syntactically and semantically. *Hint: Use the help function for any function you do not know.*

```
credit_info %>%
  select_if(is.character) %>%
  sapply(table)
```

```
$Attrition_Flag

Attrited Customer Existing Customer
             1627              8500

$Gender

   F    M
5358 4769

$Education_Level
```

```
      College     Doctorate      Graduate   High School Post-Graduate
         1013           451          3128          2013           516
   Uneducated       Unknown
         1487          1519

$Marital_Status

Divorced  Married   Single  Unknown
     748     4687     3943      749

$Income_Category

       $120K +    $40K - $60K    $60K - $80K   $80K - $120K Less than $40K
           727           1790           1402           1535           3561
       Unknown
          1112

$Card_Category

    Blue     Gold Platinum   Silver
    9436      116       20      555
```

Exercise 1.3. Overwrite the variables `Income_Category` and `Education_Level` into ordered factors. When setting the levels for each group, set `"Unknown"` as the lowest level. Use this cleaned data set for the remaining exercises.

Exercise 1.4. Group the data set by income category and find out each group's `mean` and `median` credit limit.

Exercise 1.5. Which income group has the highest mean credit limit?

Exercise 1.6. Use the following code snippet to modify the data set by incorporating it into the `mutate` function. The snippet converts all `"Unknown"` values contained in character or factor columns into `NA` values, which are easier to handle.

```
across(where(~ is.character(.) | is.factor(.)),
       ~na_if(.,"Unknown"))
```

Exercise 1.7. Apply the `na.omit()` function to the data set to remove all samples in the data set that contain `NA` values. How many samples have been removed in total?

Sometimes, we only want to infer results for specific subgroups. The Blue Credit Card is the most common type of credit card. Gaining insights for this particular group allows us to retrieve information that might be useful in later analyses.

Exercise 1.8. Find out how many customers have a Blue credit card.

Exercise 1.9. Create a new data set `credit_info_blue` containing all customers that hold a Blue credit card.

Exercise 1.10. Find the number of female customers holding the Blue Card who are, at most, 40 years old and have a credit limit above 10,000 USD.

1.5.2 Visual Data Exploration

Exercise 1.11. We now want to explore some of the demographics in our data set. Create a histogram for the age of the customers using the `geom_histogram` function. Note that only one variable is required for the aesthetics to create a histogram.

Exercise 1.12. Using the default parameters in the `geom_histogram` function, the message "`stat_bin()` using `bins = 30`. Pick better value with `binwidth`." is displayed. Modify the code so that each age gets its own bin.

Exercise 1.13. Now that the histogram looks more organized, we want to add more information. For example, by setting the `fill` option to `Gender`, we can create two overlapping histograms showing the number of male and female customers within each age group.

Exercise 1.14. Instead of visualizing the `Gender` as in the plot above, we now want to analyze the continuous variable `Credit_Limit`. Therefore, instead of a histogram, use the `geom_density` function that plots an estimate of the underlying probability density.

Exercise 1.15. The histograms and density plots only provide limited insight into the demographics and customer status as it is relatively complex to figure out the proportions of each group. To take this one step further, consider the following histogram, which shows the `Education_Level` within every bin.

```
ggplot(data = credit_info_clean, aes(Customer_Age, fill = Education_Level))+
  geom_histogram(binwidth = 1)
```

We can use the `geom_histogram` function and the `facet_wrap` function, which generates a subplot for each group. Apply the `facet_wrap` function to create a subplot for each education level.

1.5.3 Loss functions

In future exercises, different loss functions will be deployed to measure how far some regression results deviate from actual values. This exercise, therefore, briefly discusses the advantages and disadvantages of some loss functions and introduces them in R.

Data-wise, we will consider the `credit_info` dataset and a simple linear model that is used to predict each customer's credit limit.

The following Code snippet reads the unmodified data, removes the features `Total_Revolving_Bal` and `Avg_Open_To_Buy` and trains a linear model with target variable `Credit_Limit` on all the remaining features. It's important to note that the model is intentionally kept simple for demonstrative purposes, making it easier for you to grasp and apply the concepts.

Copy the snippet into your own notebook and run it. *Hint: You might have to change the path in the* `read.csv` *function to your specified data path (Exercise 2.1) and install the libraries that are attached.*

```r
library(tidymodels)
library(yardstick)

credit_info <- read.csv("data/BankChurners.csv")

model_linear_data <- credit_info %>%
  select(-c(Total_Revolving_Bal,Avg_Open_To_Buy))

model_linear_res <- linear_reg() %>%
  fit(Credit_Limit ~., data = model_linear_data) %>%
  augment(model_linear_data)
```

The object `model_linear_res` now contains our model's original data set and predictions. Do not worry if you do not understand every line in the snippet above. We will consider training models in future exercises more thoroughly.

1.5.3.1 MAE Loss

The first loss function we explore is the **M**ean **A**bsolute **E**rror (MAE) loss defined as

$$\text{MAE} := \text{MAE}(y, \hat{y}) := \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|,$$

where $y = (y_1, ..., y_n)$ are target values and $\hat{y} = (\hat{y_1}, ..., \hat{y_n})$ are estimates of the target values.

Exercise 1.16. Briefly explain how the `MAE` loss can be interpreted regarding the target features scale.

Exercise 1.17. The `mae` loss is a function in the `{yardstick}` library. If not already done, install the `{yardstick}` library and read the `help` function of the `mae` function. Then, apply it tot the `model_linear_res` data set and interpret the result.

1.5.3.2 (R)MSE

Another widely used loss function is the (**R**oot)**M**ean**S**quare**E**rror. It is defined as

$$\text{RMSE} := \text{RMSE}(y, \hat{y}) := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$
$$\text{MSE} := \text{MSE}(y, \hat{y}) := \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Exercise 1.18. Repeat the exercise Exercise 1.16 and Exercise 1.17 for the RMSE and MSE.

1.6 Solutions

Solution 1.1 (Exercise 2.1).

```
credit_info <- read.csv("data/BankChurners.csv")
```

Solution 1.2 (Exercise 2.2).

```
credit_info %>%
  select_if(is.character) %>%
  sapply(table)
```

```
$Attrition_Flag

Attrited Customer Existing Customer
             1627               8500

$Gender

   F    M
5358 4769

$Education_Level

      College     Doctorate      Graduate   High School Post-Graduate
         1013           451          3128          2013           516
   Uneducated       Unknown
         1487          1519

$Marital_Status
```

```
Divorced  Married   Single  Unknown
     748     4687     3943      749

$Income_Category

       $120K +    $40K - $60K    $60K - $80K   $80K - $120K Less than $40K
           727           1790           1402           1535           3561
       Unknown
          1112

$Card_Category

    Blue     Gold Platinum   Silver
    9436      116       20      555
```

1. In the first line, the data set `credit_info` is passed to the following line.
2. The `credit_info` data set is passed into the `select_if` function that selects columns of the data set based on some condition passed in the arguments. In this case, the condition is the `is.character` function, that checks, whether a column is of type `chr`. The results are then piped into the following line.
3. In the third line, the selected columns are passed into the `sapply` function, that applies a given function column wise to a data set and returns the resulting data set. Here, the `table` function is applied generating a contingency table of the counts for each column.

Solution 1.3 (Exercise 1.3).

```
credit_info_clean <-credit_info %>%
  mutate(Income_Category = factor(Income_Category,
                                  levels = c("Unknown","Less than $40K",
                                             "$40K - $60K","$60K - $80K",
                                             "$80K - $120K","$120K +"),
                                  ordered = TRUE),
         Education_Level = factor(Education_Level,
                                  levels = c("Unknown","Uneducated",
                                             "High School","College",
                                             "Graduate", "Post-Graduate",
                                             "Doctorate")
                                  )
         )
```

Solution 1.4 (Exercise 1.4).

```
credit_info_clean %>%
  group_by(Income_Category) %>%
  summarise(
    meanlim = mean(Credit_Limit),
    medlim = median(Credit_Limit)
  )
```

```
# A tibble: 6 x 3
  Income_Category meanlim medlim
  <ord>              <dbl>  <dbl>
1 Unknown            9517.   6380
2 Less than $40K     3754.   2766
3 $40K - $60K        5462.   3682
4 $60K - $80K       10759.   7660
5 $80K - $120K      15810.  12830
6 $120K +           19717.  18442
```

Solution 1.5 (Exercise 1.5).

```
credit_info_clean %>%
  group_by(Income_Category) %>%
  summarise(
    mean_group = mean(Credit_Limit)
  )
```

```
# A tibble: 6 x 2
  Income_Category mean_group
  <ord>                <dbl>
1 Unknown              9517.
2 Less than $40K       3754.
3 $40K - $60K          5462.
4 $60K - $80K         10759.
5 $80K - $120K        15810.
6 $120K +             19717.
```

Unsurprisingly, the highest income category also has the highest mean credit limit (19,717 USD).

Solution 1.6 (Exercise 1.6).

```
credit_info_clean <- credit_info_clean %>%
  mutate(across(
    where(~ is.character(.) | is.factor(.)),
    ~ na_if(., "Unknown")
  ))
```

Solution 1.7 (Exercise 1.7).

```
nrow_old <- nrow(credit_info_clean)

credit_info_clean <- credit_info_clean %>%
  na.omit()

glue::glue("{nrow_old-nrow(credit_info_clean)} samples were removed.")
```

```
3046 samples were removed.
```

Solution 1.8 (Exercise 1.8).

```
credit_info_clean %>%
  group_by(Card_Category) %>%
  summarise(n=n())
```

```
# A tibble: 4 x 2
  Card_Category     n
  <chr>         <int>
1 Blue           6598
2 Gold             81
3 Platinum         11
4 Silver          391
```

Solution 1.9 (Exercise 1.9).

```
credit_info_blue <- credit_info_clean %>%
  filter(Card_Category == "Blue")
```

Solution 1.10 (Exercise 1.10).

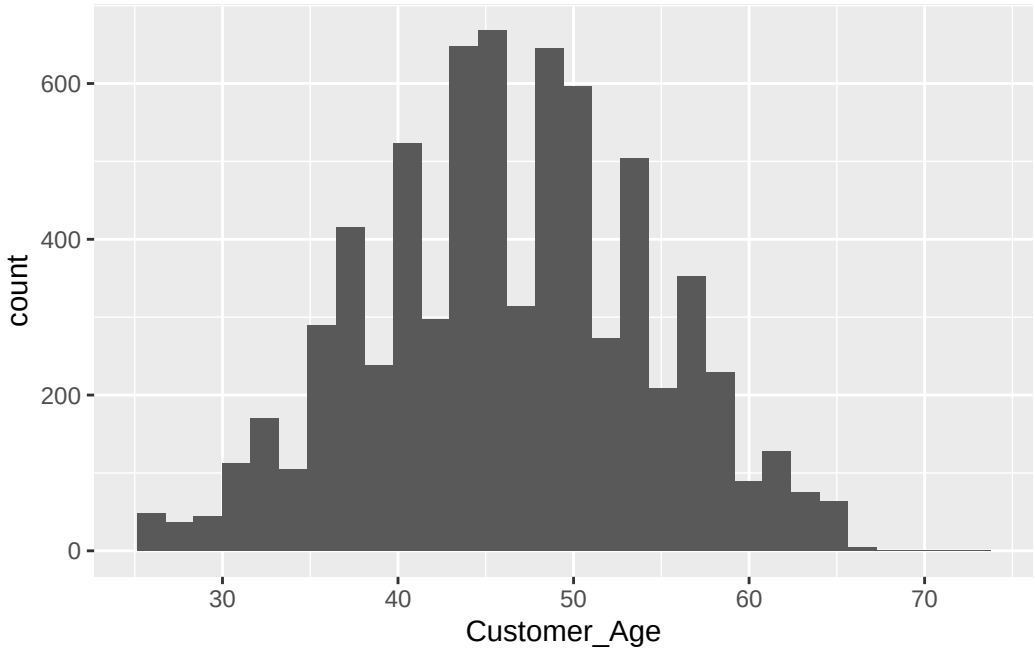
```
credit_info_blue %>%
  filter(Gender == "F" &
           Customer_Age <= 40 &
           Credit_Limit > 10000) %>%
  count()
```

```
  n
1 7
```

Solution 1.11 (Exercise 1.11).

```
ggplot(data = credit_info_clean, aes(Customer_Age))+
  geom_histogram()
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Solution 1.12 (Exercise 1.12).

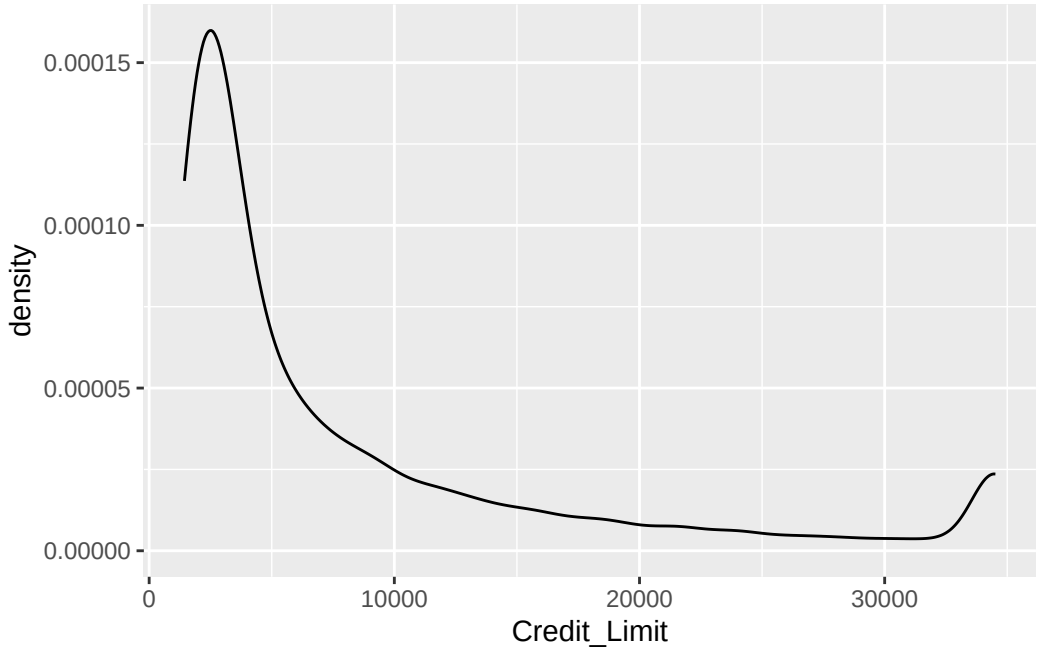
```
ggplot(data = credit_info_clean, aes(Customer_Age))+
  geom_histogram(binwidth = 1)
```

Solution 1.13 (Exercise 1.13).

```
ggplot(data = credit_info_clean, aes(Customer_Age, fill = Gender))+
  geom_histogram(binwidth = 1)
```

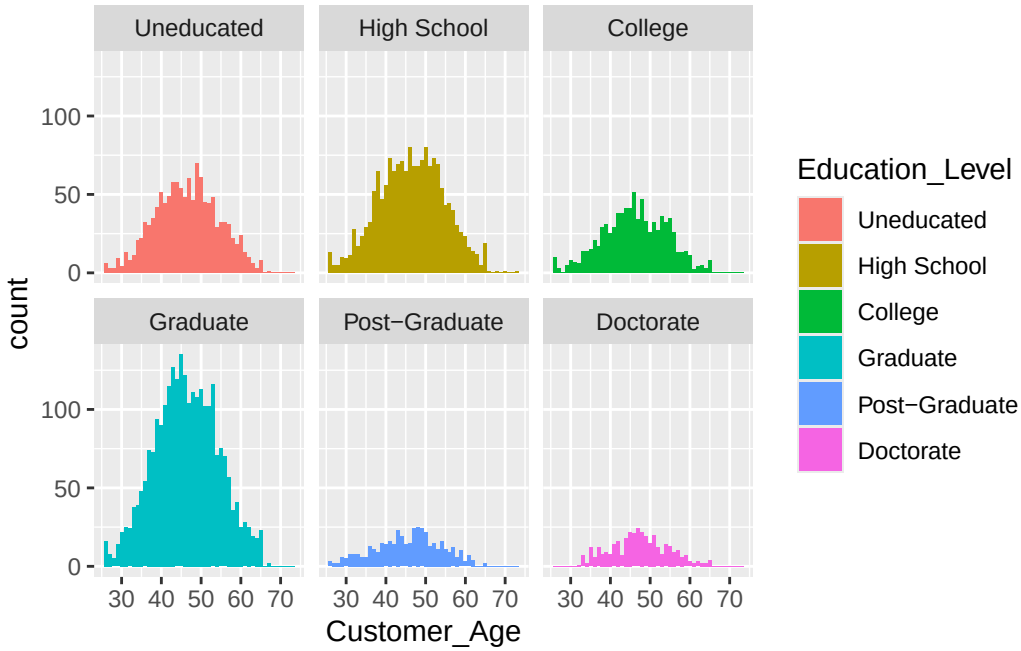
Solution 1.14 (Exercise 1.14).

```
ggplot(data = credit_info_clean, aes(Credit_Limit))+
  geom_density()
```



Solution 1.15 (Exercise 1.15).

```
ggplot(data = credit_info_clean,
       aes(Customer_Age, fill = Education_Level)
       )+
  geom_histogram(binwidth = 1) +
  facet_wrap("Education_Level")
```



Solution 1.16 (Exercise 1.16). The Mean Absolute Error (MAE) loss can be interpreted in terms of the scale of the target features because it directly measures the average absolute difference between predicted and actual target values. Thus, if the target variable is on a large scale (e.g., thousands), MAE will also be large. Conversely, for small target values, the MAE will be correspondingly smaller. This makes MAE sensitive to the scale of the target features, and it is essential to normalize or scale data if different features or targets are on very different scales to ensure the MAE provides meaningful comparisons across models or data sets.

Solution 1.17 (Exercise 1.17). The `model_linear_res` data set contains the `.pred` column, where predictions of the linear model are saved. We can use the predictions and the outcome variable `Credit_Limit` to calculate the MAE.

```
model_linear_res %>% mae(.pred,Credit_Limit)
```

```
# A tibble: 1 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 mae     standard       4114.
```

An MAE of 4114 indicates that on average, the predicted credit limit of a customer deviates 4114 USD.

Solution 1.18 (Exercise 1.18).

1. Similar to the MAE loss, the RMSE can be interpreted in terms of the scale of the target features. It also measures the average difference between the observed and predicted values, but its unique feature is that it emphasizes outliers more. Greater distances are weighted more heavily due to the square term, thereby enhancing prediction accuracy.

2. ```
   model_linear_res %>% yardstick::rmse(.pred,Credit_Limit)
   ```

   ```
   # A tibble: 1 x 3
     .metric .estimator .estimate
     <chr>   <chr>          <dbl>
   1 rmse    standard       5628.
   ```

   An RMSE of 5628 indicates that, on average, a customer's predicted credit limit deviates 5628 USD.

3. As for the MSE, the error units are expressed as squared terms. Therefore, the scales can not be interpreted directly. MSE is usually deployed in practice since it has some nice properties like differentiability at 0, which the MAE lacks. Moreover, MSE is easier to compute, thanks to the absence of a square root, which reduces computational time.

4. ```
   model_linear_rmse <- model_linear_res %>% rmse(.pred,Credit_Limit) %>%
     pluck(".estimate")
   model_linear_rmse^2
   ```

   ```
   [1] 31676421
   ```

2 Linear Models

2.1 Overview

In this exercise session, we will review linear regression and polynomial regression. We will also learn how to efficiently split our data into training and test data, how to perform cross-validation, and why that is important. Before we dive into the details, we will discuss developing statistical models in R.

2.2 Good practices for applied Machine Learning

In previous courses, we mainly focused on fitting a certain model to a given dataset. However, this process could be described as model specification rather than model development. So, what is the difference between specifying and building a model?

2.2.1 Developing a model (What we have done so far!):

- The given dataset has been cleaned, transformed, and manipulated using various packages and libraries.
- Resampling methods (like the ones we consider today) have been applied, but training the model on each subset or newly generated dataset is usually performed using a *for-loop* or similar methods. Loops should mostly be avoided in programming languages like R since they can be slow compared to optimized methods specifically written for higher performance.
- Similar to applying resampling methods, hyperparameter tuning is usually performed in the same fashion.

In summary, we have only specified the model we want to train and used a somewhat arbitrary and inconsistent approach for everything else.

One of the most significant issues we face, however, is when switching the model. The approach we have been using so far emphasizes working with one selected model that we wish to keep using after data preprocessing.

2.2.2 Developing a model (What we want to do moving forward!):

The main difference between the old and new approaches is leveraging the advantages of the `{tidyverse}` and `{tidymodels}` frameworks. These frameworks allow for consistently preprocessing the data, setting model specifications, and performing steps like resampling and hyperparameter tuning simultaneously.

Another huge advantage is that we can swiftly switch between different ML models by following this procedure. For example, applying a random forest algorithm and switching to a neural network approach for the same data is only a matter of changing a few lines of code, as we will see in later exercises.

So, where is the catch? At first, the process might seem complicated or even "overkill" for the models we use. However, as the lecture progresses, our models will also (at least sometimes) become increasingly sophisticated. We want to get used to this new process as early as possible, as it will be useful once we consider more sophisticated models.

The biggest takeaways are:

- **Consistency:** Independent of what the dataset or desired model looks like, we can (almost) always use the same procedure when building a model.
- **Effectiveness:** Once we get used to this new approach, we can develop our models more effectively.
- **Safety:** Developing an ML model has many pitfalls and potholes on the way, and by design, `{tidymodels}` helps us to avoid those.

We will introduce and explore some of the concepts above in this session's exercises and dive deeper in later sessions.

2.3 Introduction to model development with `{tidymodels}`

This section briefly introduces the most important concepts when working with the `{tidymodels}` framework.

The data set for this introduction is called "Wine Quality White," and it contains roughly 5,000 different white wines tested for their physicochemical properties, such as citric acid, pH value, and density. After assessing these properties, experts rated the wine quality and assigned a score between 0 and 10, where 0 is the lowest, and 10 is the highest score a wine can achieve.

The data set can be downloaded directly from the UC Irvine Machine Learning Repository or by clicking the button below.

Download Wine Data

Note, that the button only works in the web based version

2.3.1 Data Exploration

Since the data set is relatively nice in that we do not have to do much cleaning, we will keep this section relatively short.

```
library("tidyverse")
library("tidymodels")
```

```
data_wine <- read.csv("data/winequality-white.csv")
data_wine %>% glimpse()
```

```
Rows: 4,898
Columns: 12
$ fixed.acidity        <dbl> 7.0, 6.3, 8.1, 7.2, 7.2, 8.1, 6.2, 7.0, 6.3, 8.1,~
$ volatile.acidity     <dbl> 0.27, 0.30, 0.28, 0.23, 0.23, 0.28, 0.32, 0.27, 0~
$ citric.acid          <dbl> 0.36, 0.34, 0.40, 0.32, 0.32, 0.40, 0.16, 0.36, 0~
$ residual.sugar       <dbl> 20.70, 1.60, 6.90, 8.50, 8.50, 6.90, 7.00, 20.70,~
$ chlorides            <dbl> 0.045, 0.049, 0.050, 0.058, 0.058, 0.050, 0.045, ~
$ free.sulfur.dioxide  <dbl> 45, 14, 30, 47, 47, 30, 30, 45, 14, 28, 11, 17, 1~
$ total.sulfur.dioxide <dbl> 170, 132, 97, 186, 186, 97, 136, 170, 132, 129, 6~
$ density              <dbl> 1.0010, 0.9940, 0.9951, 0.9956, 0.9956, 0.9951, 0~
$ pH                   <dbl> 3.00, 3.30, 3.26, 3.19, 3.19, 3.26, 3.18, 3.00, 3~
$ sulphates            <dbl> 0.45, 0.49, 0.44, 0.40, 0.40, 0.44, 0.47, 0.45, 0~
$ alcohol              <dbl> 8.8, 9.5, 10.1, 9.9, 9.9, 10.1, 9.6, 8.8, 9.5, 11~
$ quality              <int> 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 5, 5, 7, 5, 7, 6~
```

Except for the `quality` variable, every other variable is of type double.

2.3.2 Training a simple linear model

For this example we build a linear model to predict the alcohol content of the wines. Recall the model equation

$$y = \beta_0 + \beta_1 x_1 + ... + \beta_n x_n + \varepsilon, \tag{2.1}$$

where $x_1, ..., x_n \in \mathbb{R}^k$ denotes n different features with k samples, $\varepsilon \sim \mathcal{N}(0, 1)$ a k-dimensional error term and $\beta_0, ..., \beta_n$ the $n + 1$ model parameters. In our example, y denotes the variable `alcohol`, $k = 4898$, and $n = 11$.

The {parsnip} packages which is part of {tidymodels} contains the function linear_reg which creates a linear model when called.

```
lm_mod <- linear_reg()
lm_mod
```

```
Linear Regression Model Specification (regression)

Computational engine: lm
```

Only calling the function name does not help much when trying to model the alcohol contents of the wine, since we haven't specified the input variables, output variable, and data used for the regression.

To add these, we can use the %>% pipe operator and fit function. As discussed in the previous exercise, the pipe operator passes the output of one operation in the other. Therefore, passing the lm_mod object into the fit function, specifies that the fit function fits a linear model. Fitting in that context refers to estimating the parameters $\beta_0, ..., \beta_n$. Besides the model specification, arguments for the fit function contain the formula which specifies the independent and dependent variables and the data argument which specifies the data that is used for training the model.

The formula in the code cell below specifies that we want to regress alcohol on every other variable indicated by the . after ~. ~ (tilde) is used to separate the left- and right-hand sides in the model formula.

As data we simply pass the whole data_wine data set.

```
lm_mod <- linear_reg() %>% fit(
 formula = alcohol ~.,
 data = data_wine
)
lm_mod
```

```
parsnip model object


Call:
stats::lm(formula = alcohol ~ ., data = data)

Coefficients:
         (Intercept)         fixed.acidity      volatile.acidity
```

```
          6.719e+02             5.099e-01             9.636e-01
        citric.acid       residual.sugar             chlorides
          3.658e-01             2.341e-01            -1.832e-01
 free.sulfur.dioxide total.sulfur.dioxide               density
         -3.665e-03             6.579e-04            -6.793e+02
                 pH             sulphates               quality
          2.383e+00             9.669e-01             6.663e-02
```

The return value is a `parsnip model object` that prints the model coefficients $\beta_0, ..., \beta_n$ when called.

Instead of fitting the linear model on all parameters using `.`, we can also specify the relationship between the independent variables using arithmetic notation:

```
lm_mod <- linear_reg() %>% fit(
 formula = alcohol ~ fixed.acidity+volatile.acidity+citric.acid+
   residual.sugar+chlorides+free.sulfur.dioxide+
   total.sulfur.dioxide+density+pH+
   sulphates+quality,
 data = data_wine
)
```

Note, that this notation does not fall under *best practices of model development* as it is more advisable to create a separate data set for training and fit the model on every variable contained using the `.` notation.

2.3.3 Evaluating a model

2.3.3.1 Creating a summary of the model parameters

Using the `tidy` function on the fitted model returns an overview of the model parameters including p-values and the t-statistic.

```
lm_mod %>% tidy()
```

```
# A tibble: 12 x 5
   term                  estimate std.error statistic  p.value
   <chr>                    <dbl>     <dbl>     <dbl>    <dbl>
 1 (Intercept)           672.        5.56       121.    0
 2 fixed.acidity           0.510     0.00986     51.7   0
 3 volatile.acidity        0.964     0.0672      14.3   1.00e-45
```

```
 4 citric.acid             0.366     0.0560       6.54  6.88e-11
 5 residual.sugar          0.234     0.00296     79.1   0
 6 chlorides              -0.183     0.321       -0.571 5.68e- 1
 7 free.sulfur.dioxide    -0.00366   0.000494    -7.42  1.33e-13
 8 total.sulfur.dioxide    0.000658  0.000222     2.97  3.01e- 3
 9 density              -679.        5.70      -119.    0
10 pH                     2.38      0.0519      45.9   0
11 sulphates              0.967     0.0575      16.8   1.02e-61
12 quality                0.0666    0.00834      7.99  1.70e-15
```

Recall that the `statistic` column refers to the t-statistic which corresponds to the following hypotheses for any $\hat{\beta}_i$, $i = 1, ..., 12$:

$$H_0 : \hat{\beta}_i = 0 \qquad \text{vs.} \qquad H_1 : \hat{\beta}_i \neq 0.$$

If the p–value regarding this test is low, we can confidently reject the null hypothesis.

Similar to the `tidy` function, the `summary` function can be called on the `fit` attribute of the model, which also returns a summary which contains a few more details, such as the F-statistic, R^2, and residual standard error.

```
lm_mod$fit %>% summary()
```

```
Call:
stats::lm(formula = alcohol ~ fixed.acidity + volatile.acidity +
    citric.acid + residual.sugar + chlorides + free.sulfur.dioxide +
    total.sulfur.dioxide + density + pH + sulphates + quality,
    data = data)

Residuals:
    Min      1Q  Median      3Q     Max
-3.3343 -0.2553 -0.0255  0.2214 15.7789

Coefficients:
                       Estimate Std. Error  t value Pr(>|t|)
(Intercept)           6.719e+02  5.563e+00  120.790  < 2e-16 ***
fixed.acidity         5.099e-01  9.855e-03   51.745  < 2e-16 ***
volatile.acidity      9.636e-01  6.718e-02   14.342  < 2e-16 ***
citric.acid           3.658e-01  5.596e-02    6.538 6.88e-11 ***
residual.sugar        2.341e-01  2.960e-03   79.112  < 2e-16 ***
chlorides            -1.832e-01  3.207e-01   -0.571  0.56785
free.sulfur.dioxide  -3.665e-03  4.936e-04   -7.425 1.33e-13 ***
```

```
total.sulfur.dioxide  6.579e-04  2.217e-04    2.968  0.00301 **
density              -6.793e+02  5.696e+00 -119.259  < 2e-16 ***
pH                    2.383e+00  5.191e-02   45.916  < 2e-16 ***
sulphates             9.669e-01  5.751e-02   16.814  < 2e-16 ***
quality               6.663e-02  8.341e-03    7.988 1.70e-15 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4409 on 4886 degrees of freedom
Multiple R-squared:  0.8719,    Adjusted R-squared:  0.8716
F-statistic:  3024 on 11 and 4886 DF,  p-value: < 2.2e-16
```

To extract these statistics in *tidy* fashion, the `{yardsticks}` library (see Exercise Session 01) can help.

2.3.3.2 Using metric sets and glance

Instead of using the `summary` function on the `fit` attribute to extract certain metrics, we can also use the `metric_set` function. First, define a metric set by passing different metrics such as `rsq`, `rmse`, and `mae` into the `metric_set` function. Then, pass the predictions of the model into this newly defined metric set which returns the specified metrics.

```
multi_metric <- metric_set(rsq,rmse,mae)

lm_mod %>%
  augment(data_wine) %>%
  multi_metric(.pred,alcohol)
```

```
# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rsq     standard       0.872
2 rmse    standard       0.440
3 mae     standard       0.294
```

A general metric set can be created using the `glance` function which returns a comprehensive list of metrics for the underlying model.

```
glance(lm_mod)
```

```
# A tibble: 1 x 12
  r.squared adj.r.squared sigma statistic p.value    df logLik   AIC   BIC
      <dbl>         <dbl> <dbl>     <dbl>   <dbl> <dbl>  <dbl> <dbl> <dbl>
1     0.872         0.872 0.441     3024.       0    11 -2933. 5892. 5976.
# i 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

2.3.4 Training and test split

Splitting the data into three different subsets, called *training*, *validation*, and *testing* data, is a crucial aspect in Machine Learning.

Figure 2.1

The basic idea is to train the model on the training data, validate the results on the validation data, and finally test the performance on the testing data that has previously not been observed.

Without this separation the model might become prone to overfitting meaning that the model does not perform well on previously unseen data.

The general procedure for training and testing a model is depicted in the following figure.

Here, the training and validation data set are used in the training procedure and the testing data set in the testing procedure.

> **Note**
>
> The reevaluation during training only makes sense when changing parameters can lead to an improvement. For a simple linear regression this is not the case, since once the model parameters $\beta_0, ..., \beta_n$ are estimated, they are optimal (cf. Gauss-Markov theorem).

A simple training and test split with 80% training data and 20% test data can be generated with the `initial_split`, `training`, and `testing` function. Before splitting the data (which is done randomly) we can set a *seed*. Setting a seed allows us to reproduce the outcome of our code, even when randomness is involved.

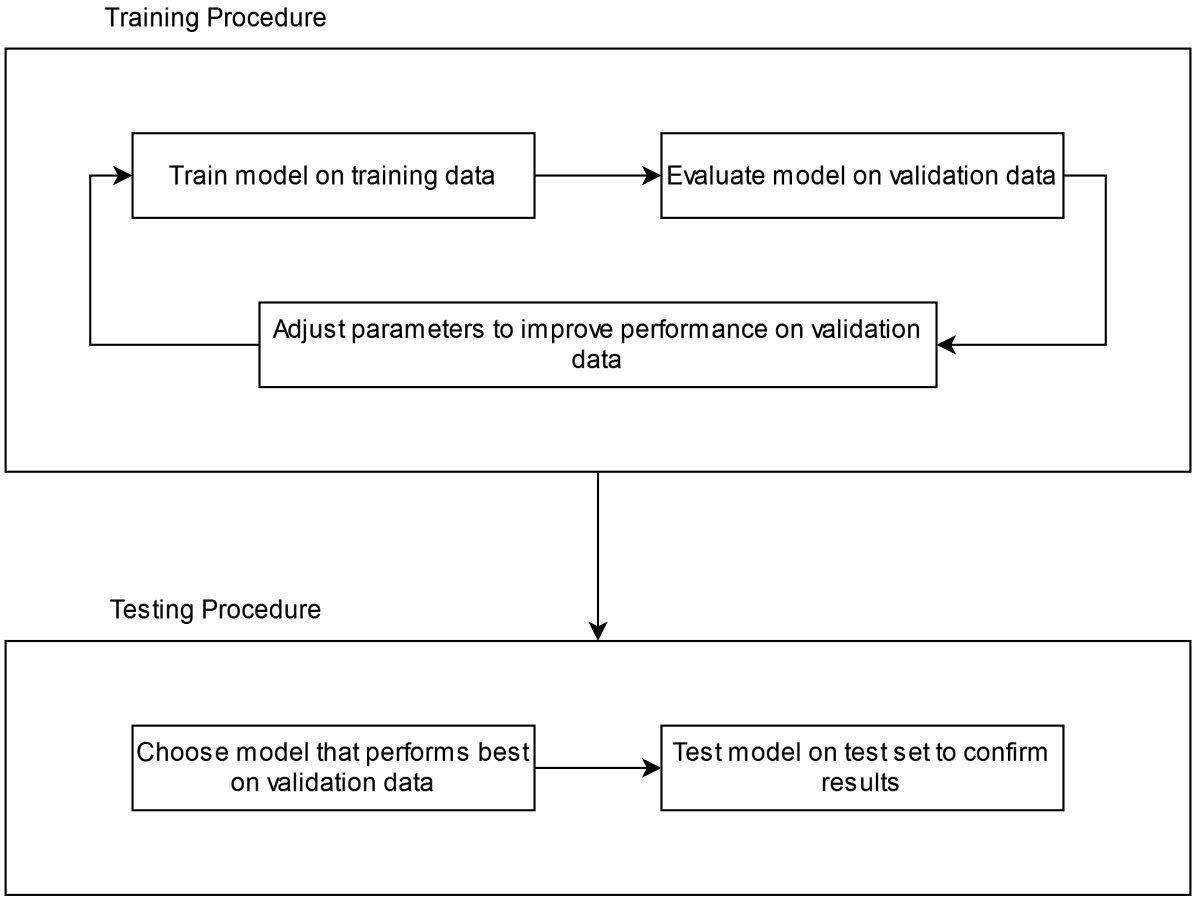


Figure 2.2: Based on this Google Developers Figure

```r
set.seed(123)
tt_split<-initial_split(data_wine, 0.8)
data_train <- tt_split %>% training()
data_test <- tt_split %>% testing()
```

Training the linear model on the training data and evaluating it on the test data then yields

```r
lm_mod <- linear_reg() %>%
  fit(
   formula = alcohol ~.,
   data = data_train
  )

lm_mod %>%
  augment(data_test) %>%
  multi_metric(.pred,alcohol)
```

```
# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rsq     standard       0.887
2 rmse    standard       0.408
3 mae     standard       0.307
```

2.3.5 Cross validation

A popular approach we will use in most exercises extends the procedure described in Figure 2.1 and Figure 2.2. Instead of using a single training and validation data set, we create **multiple instances** of training and validation data by randomly assigning a data point to the training or validation set. More formally, v-fold cross validation (CV) randomly splits the training and validation data into v equally sized subsets. In each iteration, one of these subsets is set aside to be the validation data, while the other $v-1$ subsets are used for training. The figure below depicts how that process works for `v = 5`.

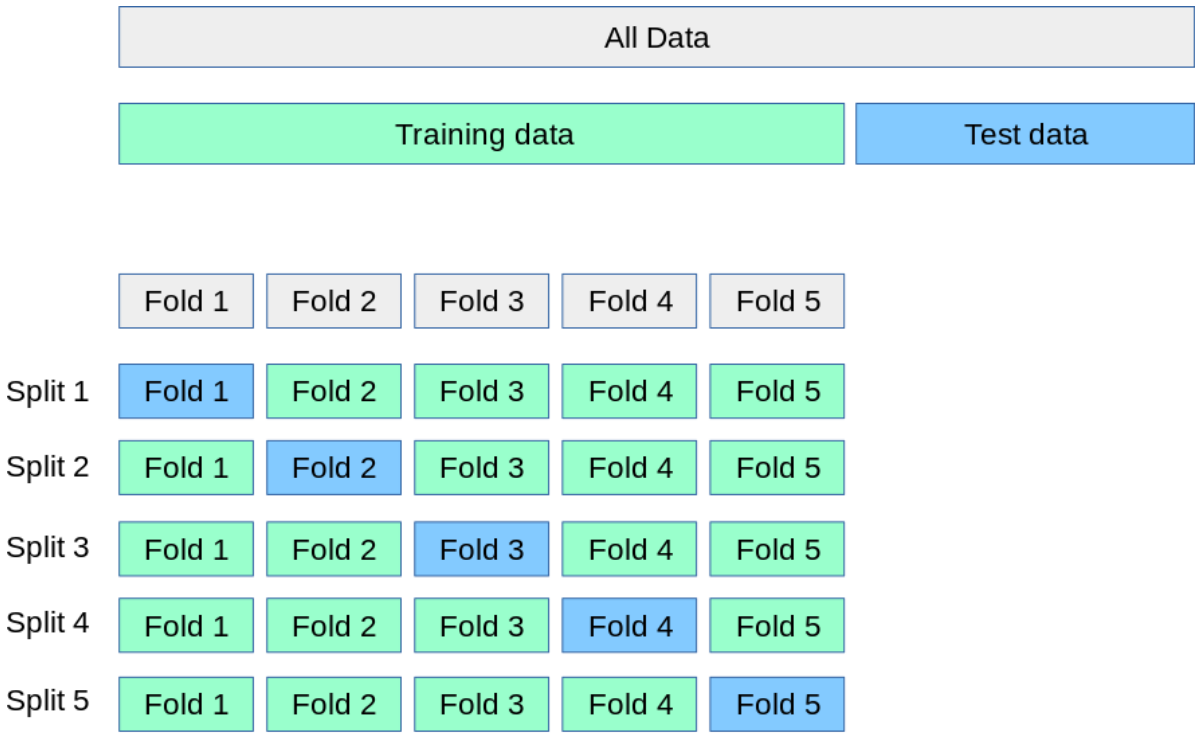


Figure 2.3: 5-fold Cross Validation

A part of the whole data set is left as testing data. The blue boxes in the lower part of the figure contain the validation (sub)sets for each split while the training data is contained in the remaining (sub)sets.

2.3.5.1 Creating a cross validation set in R

Cross validation in R can be performed using the `{resample}` library that is also part of the `{tidymodels}` framework.

The `vfold_cv` function creates a nested data frame, meaning that each entry in the data frame contains another data frame. `v` denotes the number of created folds and `data_train` specifies that the folds are created from the `data_train` data set.

```
set.seed(123)
folds <- vfold_cv(data_train, v = 5)
```

Note, that since the folds are also created stochastically, setting a seed ensures that the results are reproducible.

The `folds` tibble currently contains two columns `"splits"` and `"id"`. The column `"splits"` contains lists with the respective splits, while the column `"id"` contains a unique identifier for the respective splits. We can access split i in a fold by applying the `pluck()`-function iteratively. By using `pluck(1)`, we can extract the list of splits:

```
folds %>%
  pluck(1)
```

```
[[1]]
<Analysis/Assess/Total>
<3134/784/3918>

[[2]]
<Analysis/Assess/Total>
<3134/784/3918>

[[3]]
<Analysis/Assess/Total>
<3134/784/3918>

[[4]]
<Analysis/Assess/Total>
<3135/783/3918>

[[5]]
<Analysis/Assess/Total>
<3135/783/3918>
```

Say, we want to take a closer look at the second split, then calling `pluck(2)` allows accessing it.

```
folds %>%
  pluck(1)%>%
  pluck(2)
```

```
<Analysis/Assess/Total>
<3134/784/3918>
```

In our example, each split contains 5 folds, of which are comprised of four parts training data and one part validation data. To access the respective parts, we can use the `analysis()`- and `assessment()`-functions:

```
folds %>%
  pluck(1)%>%
  pluck(2) %>%
  analysis() %>%
  glimpse()
```

```
Rows: 3,134
Columns: 12
$ fixed.acidity        <dbl> 8.3, 6.2, 5.9, 5.7, 6.7, 6.0, 6.6, 6.6, 9.0, 8.6,~
$ volatile.acidity     <dbl> 0.25, 0.28, 0.32, 0.26, 0.18, 0.29, 0.15, 0.17, 0~
$ citric.acid          <dbl> 0.33, 0.45, 0.39, 0.24, 0.28, 0.25, 0.32, 0.26, 0~
$ residual.sugar       <dbl> 2.5, 7.5, 3.3, 17.8, 10.2, 1.4, 6.0, 7.4, 10.4, 1~
$ chlorides            <dbl> 0.053, 0.045, 0.114, 0.059, 0.039, 0.033, 0.033, ~
$ free.sulfur.dioxide  <dbl> 12, 46, 24, 23, 29, 30, 59, 45, 52, 42, 14, 14, 3~
$ total.sulfur.dioxide <dbl> 72, 203, 140, 124, 115, 114, 128, 128, 195, 240, ~
$ density              <dbl> 0.99404, 0.99573, 0.99340, 0.99773, 0.99469, 0.98~
$ pH                   <dbl> 2.89, 3.26, 3.09, 3.30, 3.11, 3.08, 3.19, 3.16, 3~
$ sulphates            <dbl> 0.48, 0.46, 0.45, 0.50, 0.45, 0.43, 0.71, 0.37, 0~
$ alcohol              <dbl> 9.5, 9.2, 9.2, 10.1, 10.9, 13.2, 12.1, 10.0, 10.2~
$ quality              <int> 5, 6, 6, 5, 7, 6, 8, 6, 6, 6, 6, 5, 5, 7, 6, 7, 6~
```

```
folds %>%
  pluck(1) %>%
  pluck(2) %>%
  assessment() %>%
  glimpse()
```

```
Rows: 784
Columns: 12
$ fixed.acidity        <dbl> 7.7, 7.2, 9.3, 6.8, 6.4, 6.7, 5.9, 7.0, 6.1, 6.3,~
$ volatile.acidity     <dbl> 0.28, 0.24, 0.34, 0.37, 0.20, 0.36, 0.28, 0.24, 0~
$ citric.acid          <dbl> 0.58, 0.29, 0.49, 0.47, 0.28, 0.26, 0.14, 0.24, 0~
$ residual.sugar       <dbl> 12.10, 2.20, 7.30, 11.20, 2.50, 7.90, 8.60, 9.00,~
$ chlorides            <dbl> 0.046, 0.037, 0.052, 0.071, 0.032, 0.034, 0.032, ~
$ free.sulfur.dioxide  <dbl> 60, 37, 30, 44, 24, 39, 30, 42, 37, 24, 26, 42, 3~
$ total.sulfur.dioxide <dbl> 177, 102, 146, 136, 84, 123, 142, 219, 136, 108, ~
$ density              <dbl> 0.99830, 0.99200, 0.99800, 0.99680, 0.99168, 0.99~
```

```
$ pH                  <dbl> 3.08, 3.27, 3.17, 2.98, 3.31, 2.99, 3.28, 3.47, 3~
$ sulphates           <dbl> 0.46, 0.64, 0.61, 0.88, 0.55, 0.30, 0.44, 0.46, 0~
$ alcohol             <dbl> 8.9, 11.0, 10.2, 9.2, 11.5, 12.2, 9.5, 10.2, 10.8~
$ quality             <int> 5, 7, 5, 5, 5, 7, 6, 6, 7, 6, 7, 5, 5, 5, 6, 6, 7~
```

Since we will be using cross validation in R throughout the manuscript many different times, it is essential to understand its components and construction.

2.3.5.2 Training a model using cross validation

To train the linear model on each split, we can use the `fit_resamples` function. Training the model on each fold is the as simple as training he model without resamples: We simply pass the model specification, formula, and additionally the cross validation object into the `fit_resamples` formula. Additionally, we can also pass the metric set `multi_metric` to specify which metrics we want to use for model evaluation.

```
lm_mod_resampling <- linear_reg()
lm_mod_resampling_res <- fit_resamples(lm_mod_resampling,
                            alcohol ~.,
                            folds,
                            metrics = multi_metric)
```

The return value of the `fit_resamples` function is a data frame containing 5 linear models (since we specified `v=5` when creating the folds object).

```
lm_mod_resampling_res
```

```
# Resampling results
# 5-fold cross-validation
# A tibble: 5 x 4
  splits             id    .metrics          .notes
  <list>             <chr> <list>            <list>
1 <split [3134/784]> Fold1 <tibble [3 x 4]> <tibble [0 x 3]>
2 <split [3134/784]> Fold2 <tibble [3 x 4]> <tibble [0 x 3]>
3 <split [3134/784]> Fold3 <tibble [3 x 4]> <tibble [0 x 3]>
4 <split [3135/783]> Fold4 <tibble [3 x 4]> <tibble [0 x 3]>
5 <split [3135/783]> Fold5 <tibble [3 x 4]> <tibble [0 x 3]>
```

We are mainly interested in the cross validation error (CV-RMSE) defined as

$$\text{CV} - \text{RMSE} = \frac{1}{v} \sum_{i=1}^{v} \text{RMSE}_i,$$

where RMSE_i stands for the RMSE of the i-th. hold-out sample.

We can collect this metric by applying the `collect_metrics` function:

```
lm_mod_resampling_res %>% collect_metrics()
```

```
# A tibble: 3 x 6
  .metric .estimator  mean     n std_err .config
  <chr>   <chr>      <dbl> <int>   <dbl> <chr>
1 mae     standard   0.300     5 0.00460 Preprocessor1_Model1
2 rmse    standard   0.451     5 0.0710  Preprocessor1_Model1
3 rsq     standard   0.866     5 0.0414  Preprocessor1_Model1
```

The third column `mean` depicts the mean `rmse` and `rsq` across all the splits. Comparing the CV-RMSE (0.451) to the true out of sample (OOS) RMSE of the test set (0.408) reveals that the performance of the linear model seems stable, meaning that it is not prone to overfitting. Furthermore, the CV-RMSE seems to overestimate the true OOS RMSE, since $0.451 > 0.408$. Using this information we can make prediction about the alcohol contents of a wine with the estimated model parameters on the training data set.

2.3.6 Polynomial regression and the bias variance trade off

Polynomial regression is a special case of multiple linear regression where the model is still linear in its coefficients but the dependent variable y is modeled as polynomial in x. For an n-th degree polynomial model the model equation is therefore given by

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + ... + \beta_n x^n.$$

> **Note**
>
> For this specific model we only use one independent variable x instead of n different independent variables x_1,...,x_n.

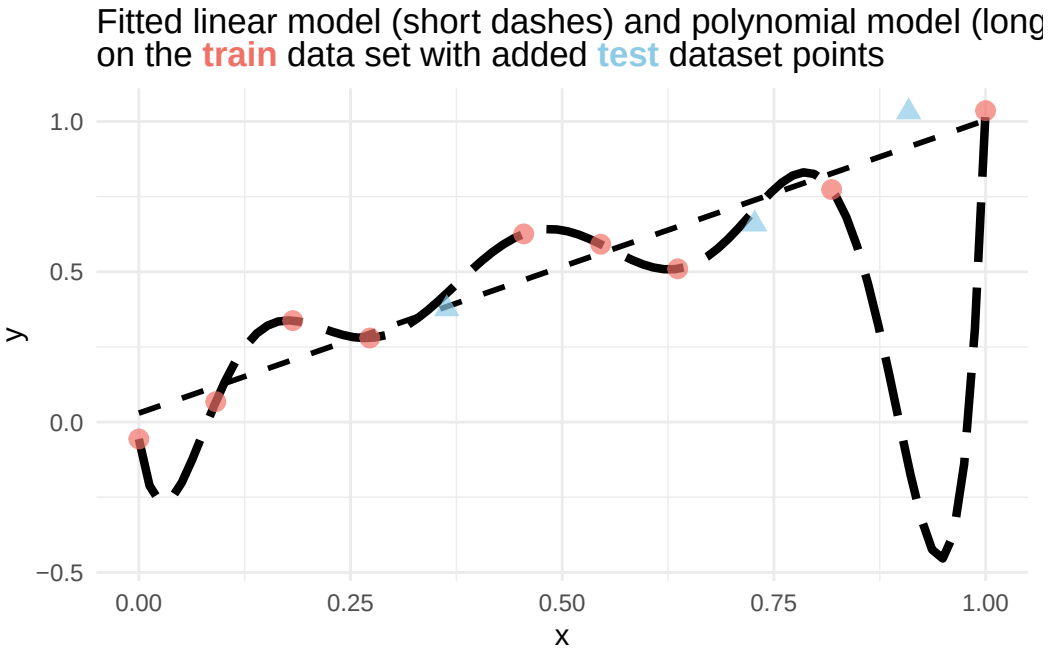
Say, we want to model the alcohol contents of wine with a MLR model using the wine density as the only predictor.

Consider the following synthetic dataset consisting of 12 observations.

```r
set.seed(123)
data_synth <- tibble(
  x = seq(0,1,1/11),
  y = x+rnorm(12,0,0.1)
)

split_synth <- initial_split(data_synth,0.8)
data_train_synth <- split_synth %>% training()
data_test_synth <- split_synth %>% testing()

data_synth
```

```
# A tibble: 12 x 2
        x       y
    <dbl>   <dbl>
 1 0      -0.0560
 2 0.0909  0.0679
 3 0.182   0.338
 4 0.273   0.280
 5 0.364   0.377
 6 0.455   0.626
 7 0.545   0.592
 8 0.636   0.510
 9 0.727   0.659
10 0.818   0.774
11 0.909   1.03
12 1       1.04
```

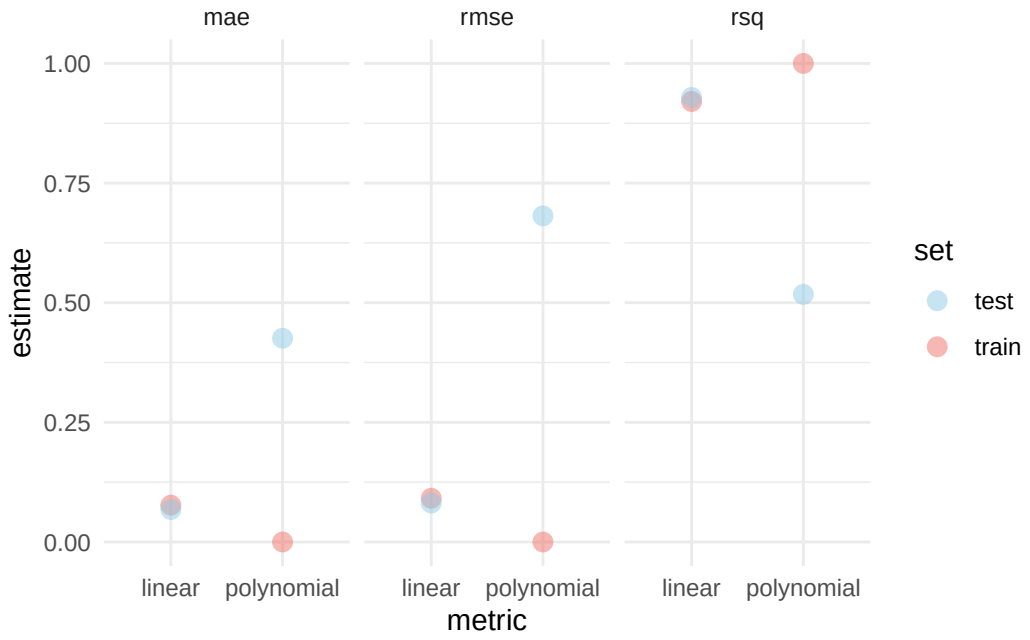
From the figure below, it is immediately evident, that a polynomial model perfectly fits the training data, but severely fails to estimate the rightmost point of the testing data. While the linear model does not fit the training data



To verify this, we can also compare the training and test error for different metrics of each model.

```
lm_poly_mod <- linear_reg() %>%
  fit(formula = y ~ poly(x, 8),
      data = data_train_synth
      )

lm_lin_mod <- linear_reg() %>%
  fit(formula = y ~ x,
      data = data_train_synth
      )
```

Visualizing the difference in train and test error then yields



The example above demonstrates the phenomenon of bias-variance tradeoff. A low variance and high bias can be observed in the linear model, since there are only few (*two*) model parameters and a small discrepancy between training and test error. The polynomial model exhibits a large discrepancy between training and test error and since there are many (*nine*) parameters, indicating that it has a high variance but low bias.

2.4 Exercises

Throughout the exercises, we will work with a subset of the Apartment rental offers in Germany data set that can be downloaded using the button below. It contains 239 unique rental listings for flats in Augsburg which were sourced at three different dates in 2018 and 2019 and contains 28 different variables.

Download AugsburgRental

Note, that the button only works in the web based version

Exercise 2.1. Instead of focusing on a comprehensive data cleaning and manipulation process, we will simply use the two variables `livingSpace` measuring the area of living in m^2 of a listing and `baseRent` in EUR, representing the monthly base rent.

Import the data and visualize the relationship between the two variables `livingSpace` and `baseRent` with a scatter plot. Rename the axis of the plot such that they display the words `base rent` and `living space` (seperated by a whitespace).

Exercise 2.2. Without conducting a thorough outlier analysis, remove every listing that either costs more than 2500 EUR or is bigger than 200 m^2.

Exercise 2.3. Create a training (80%) and test data set using the filtered data. Use `set.seed(2)` to generate reproducible results.

Exercise 2.4. Train a simple linear model on the training data.

Exercise 2.5. Generate a model summary and interpret the coefficients. Is the independent variable statistically significant?

Exercise 2.6. Evaluate the model on the test data by considering the adjusted R^2 and MAE. On average, how far off is the estimated base rent?

Exercise 2.7. Create a 10-fold cross validation split of the training data using the same seed as before and retrain the simple linear model. Compare the cross validation MAE to the OOS MAE and interpret the result.

Exercise 2.8. Repeat Exercises Exercise 2.6 and Exercise 2.8 for a polynomial model of degree 20. Compare the test and cross-validation MAE of the linear model with the polynomial model.

Exercise 2.9. Create a scatter plot of the training data and insert both fitted curves using the `geom_smooth` function.

Exercise 2.10. Assume we have fitted a simple linear model

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x.$$

For $\hat{\beta}_1$ the t-statistic has a p-value of $1e-16$ ($= 1 \cdot 10^{-16}$). Describe the null hypothesis of the underlying test and explain what concnulision can be drawn based on this p-value.

Exercise 2.11. Assume we have a data set with $k = 400$ containing a single independent variable x and a real valued dependent variable y. We fit a simple regression model and a polynomial regression with degree 5 on 75% of the data and leave the remaining 25% for testing .

1. Assume that the true relationship between x and y is linear. Consider the training RMSE for the linear regression, and also the training RMSE for the polynomial regression. Choose the most appropriate answer and justify your choice.

   1.1 The polynomial model has more flexibility due to the additional terms, so it will fit the data better than the linear model, resulting in a lower training RMSE.

1.2 The linear model is simpler and less prone to overfitting, so it will produce a lower training RMSE compared to the polynomial model.

1.3 Since both models are trying to explain the same data, their training RMSE should be approximately the same.

1.4 Without knowing the true underlying relationship between the predictor and response, we cannot definitively predict the behavior of the training RMSE for either model.

2. Repeat the previous exercise but instead of the training RMSE, consider the test RMSE.

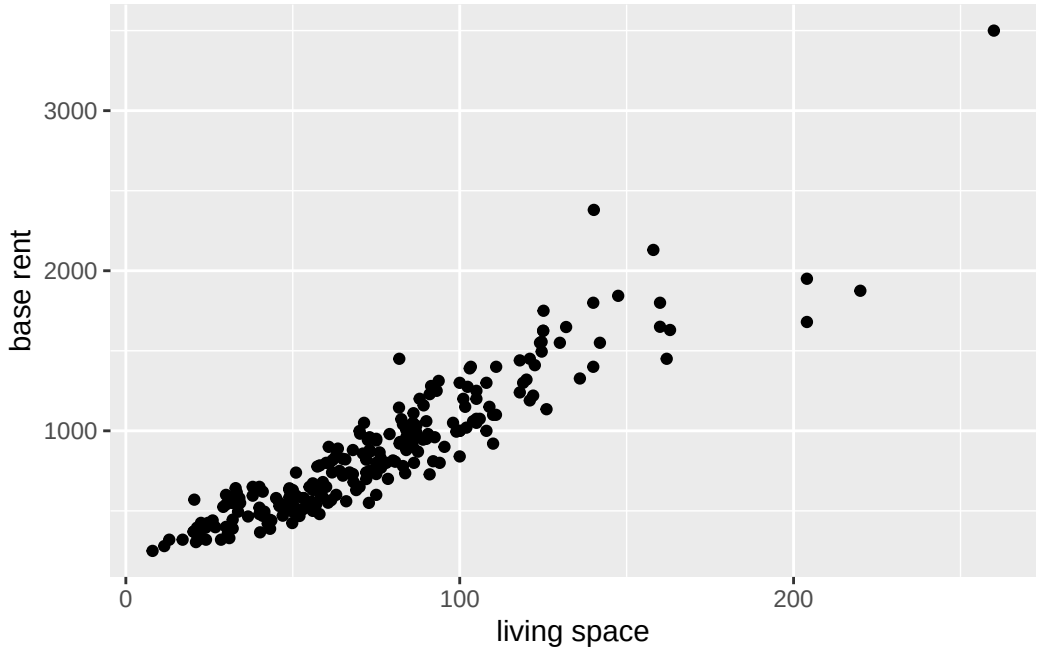
2.5 Solutions

Solution 2.1 (Exercise 2.1). The data set can be imported using the `read_csv` function.

```
data_aux <- read_csv("data/rent_aux.csv")
```

Using the `ggplot` function, we can create a simple scatter plot. The `labs` function allows to define new axis labels by specifying the axis and assigning the names to the respective axis. x corresponds to the horizontal axis and y to the vertical axis.

```
data_aux %>%
  ggplot(aes(x = livingSpace, y = baseRent)) +
  geom_point() +
  labs(
    x = "living space",
    y = "base rent"
  )
```



Solution 2.2 (Exercise 2.2). To select all listings with base rent lower than 2500 EUR and living space less than 200 sqm, we can use the `filter` function and overwrite the old data set.

```
data_aux <- data_aux %>%
  filter(baseRent <= 2500, livingSpace <= 200)
```

Solution 2.3 (Exercise 2.3). By setting `set.seed(2)` we can make sure that the following results are reproducible. Similar to Section 2.3.4, we can define a split object using the `initial_split` function and select the training and test portion using the `training` and `testing` functions respectively.

```
set.seed(2)
split <-initial_split(data_aux,0.8)
data_train <- split %>% training(split)
data_test <- split %>% testing(split)
```

Solution 2.4 (Exercise 2.4).

```
lm_mod <- linear_reg() %>%
  fit(baseRent ~ livingSpace, data_train)
```

Solution 2.5 (Exercise 2.5). A simple model summary can be created by passing the trained model into the `tidy` function.

```
lm_mod %>% tidy()
```

```
# A tibble: 2 x 5
  term        estimate std.error statistic  p.value
  <chr>          <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept)     87.3    25.2        3.46 6.58e- 4
2 livingSpace     10.4     0.322     32.2  5.49e-79
```

The estimated parameter $\hat{\beta}_1 = 10.4$ indicates that according to the linear model, we expect the base rent to rise 10.4 EUR for every additional square meter of living space.

Solution 2.6 (Exercise 2.6). We can calculate the adjusted R^2 and MAE using the `metric_set` function.

```
multi_metric<- metric_set(rsq,mae)
lm_mod %>%
  augment(data_test) %>%
  multi_metric(.pred,baseRent)
```

```
# A tibble: 2 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rsq     standard       0.883
2 mae     standard     108.
```

MAE= 108 indicates that on average our estimated base rent is off by 108 EUR.

Solution 2.7 (Exercise 2.8).

```
set.seed(2)
folds <- vfold_cv(data_train)

lm_mod_resampling_res <- linear_reg() %>%
  fit_resamples(baseRent ~ livingSpace,
                folds,
                metrics = multi_metric)

lm_mod_resampling_res %>% collect_metrics()
```

```
# A tibble: 2 x 6
  .metric .estimator    mean     n std_err .config
  <chr>   <chr>        <dbl> <int>   <dbl> <chr>
1 mae     standard   116.       10  5.12   Preprocessor1_Model1
2 rsq     standard     0.869    10  0.0153 Preprocessor1_Model1
```

The cross-validation MAE of 116 EUR compared to the test MAE of 108 EUR indicates that the cross-validation error is overestimating the true test error. This is in fact favorable, since it is preferable to have a pessimistic bias towards an estimated error. In other words, overestimating an error is always better than underestimating an error.

Solution 2.8 (Exercise 2.8).

```
lm_poly_mod <- linear_reg() %>% fit(
  formula = baseRent ~ poly(livingSpace, 20),
  data = data_train
)

lm_poly_mod %>%
  augment(data_test) %>%
  multi_metric(.pred,baseRent)
```

```
# A tibble: 2 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rsq     standard       0.349
2 mae     standard     153.
```

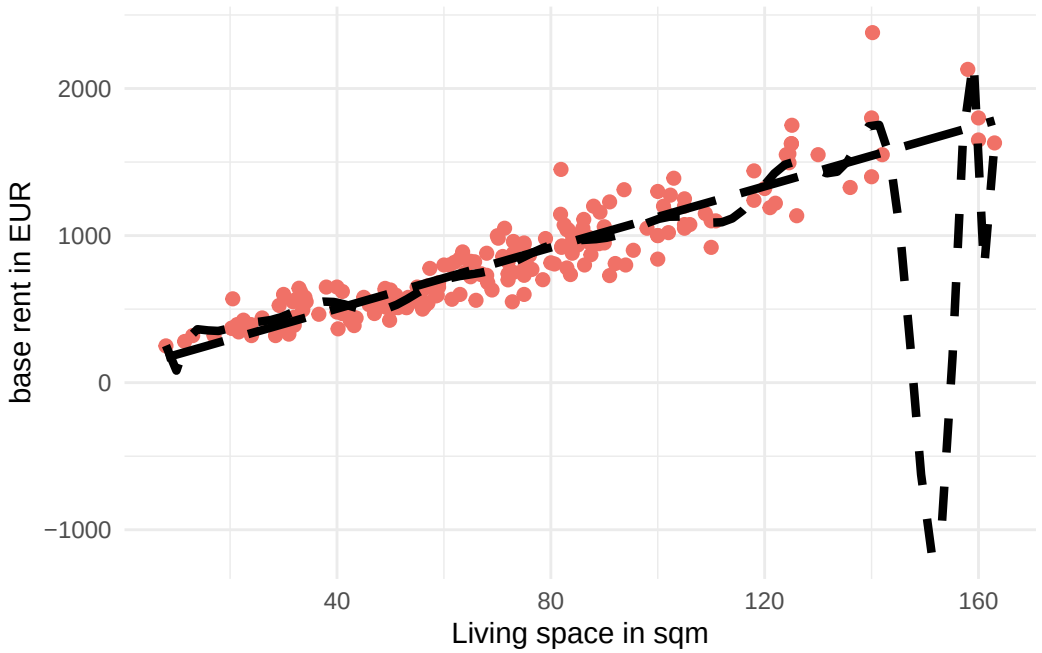
```
lm_poly_mod_resampling_res <- linear_reg() %>%
  fit_resamples(baseRent ~ poly(livingSpace,20),
                folds,
                metrics = multi_metric)

lm_poly_mod_resampling_res %>% collect_metrics()
```

```
# A tibble: 2 x 6
  .metric .estimator     mean     n   std_err .config
  <chr>   <chr>         <dbl> <int>     <dbl> <chr>
1 mae     standard   1541.       10 1205.     Preprocessor1_Model1
2 rsq     standard      0.624    10    0.0937 Preprocessor1_Model1
```

Compared to the simple linear model, the polynomial model performs a lot worse on the test data as it severely ovefits on the training data. The CV-MAE of 1541 compared to the test MAE of 153 is another redflag we should consider when evaluating model performance.

Solution 2.9 (Exercise 2.9). We can directly pipe the training data `data_train` into the `ggplot` function to create the plot. The `color` argument in the `geom_point` function changes the color of the points and the `size` argument toggles the point sizes in the figure. When using `geom_smooth` have to express the formula in terms of `x` and `y` instead of using `livingSpace` and `baseRent`. Setting `se=FALSE` removes the confidence band of the estimated lines. `color = "black"`, `linetype=2`, and `linewidth=1.5` specifies the visual characteristics of the line. As with the previous plot, we can also change the axis labels using `labs`.

```
data_train %>% ggplot(aes(x=livingSpace,y=baseRent))+
  geom_point(color="#f07167",
             size = 2)+
  geom_smooth(formula = y ~ poly(x, 20),
              method = "lm",
              color = "black",
              se = FALSE,
              linetype= 2,
              linewidth = 1.5)+
  geom_smooth(formula = y ~ x,
              method = "lm",
              color = "black",
              se = FALSE,
              linetype= 5,
              linewidth = 1.5)+
  labs(x="Living space in sqm",
       y="base rent in EUR")+
  theme_minimal()
```



Solution 2.10 (Exercise 2.10). A p-value of $1e-16$ $(= 1 \cdot 10^{-16})$ indicates that we can reject the null hypotheses $H_0 : \beta_1 = 0$ to the significance level $1 - (10^{-16})$.

Solution 2.11 (Exercise 2.11).

1. Since the polynomial model has more flexibility due to the additional terms, it will likely fit the training data better than the linear model, resulting in a lower training RMSE.
2. The linear model is simpler and less prone to overfitting, so it will likely produce a lower test RMSE compared to the polynomial model.

3 Regularization

3.1 Introduction

This cahpter aims to review multiple linear regression (MLR) and different regularization techniques in theory and practice. Regularization techniques in linear regression involve *hyperparameters*, namely the `penalty` and `mixture`. The models we consider in this session are, ridge, lasso and elastic net regression models.

As for new R concepts, we will consider `recipes` and `workflows`, which are helpful for keeping track of the preprocessing and model training process.

A `recipe` aggregates steps applied to a data set before specifying a model. Therefore, recipes include the formula, i.e., the specification for the target and features, and most of the preprocessing steps we have applied manually before.

Workflows serve as a container for recipes and model specifications. They streamline the training process and are especially helpful whenever many different models are involved.

We use the same white wine data set as in Session 02 for the introduction.

3.1.1 Recipes and Workflows

We start out by importing the wine data set again. For detailed description of each parameter see Cortez et al.

```
library("tidyverse")
library("tidymodels")
library("glmnet")
```

```
data_wine <- read.csv("data/winequality-white.csv")
data_wine %>% glimpse()
```

```
Rows: 4,898
Columns: 12
$ fixed.acidity        <dbl> 7.0, 6.3, 8.1, 7.2, 7.2, 8.1, 6.2, 7.0, 6.3, 8.1,~
```

```
$ volatile.acidity     <dbl> 0.27, 0.30, 0.28, 0.23, 0.23, 0.28, 0.32, 0.27, 0~
$ citric.acid          <dbl> 0.36, 0.34, 0.40, 0.32, 0.32, 0.40, 0.16, 0.36, 0~
$ residual.sugar       <dbl> 20.70, 1.60, 6.90, 8.50, 8.50, 6.90, 7.00, 20.70,~
$ chlorides            <dbl> 0.045, 0.049, 0.050, 0.058, 0.058, 0.050, 0.045, ~
$ free.sulfur.dioxide  <dbl> 45, 14, 30, 47, 47, 30, 30, 45, 14, 28, 11, 17, 1~
$ total.sulfur.dioxide <dbl> 170, 132, 97, 186, 186, 97, 136, 170, 132, 129, 6~
$ density              <dbl> 1.0010, 0.9940, 0.9951, 0.9956, 0.9956, 0.9951, 0~
$ pH                   <dbl> 3.00, 3.30, 3.26, 3.19, 3.19, 3.26, 3.18, 3.00, 3~
$ sulphates            <dbl> 0.45, 0.49, 0.44, 0.40, 0.40, 0.44, 0.47, 0.45, 0~
$ alcohol              <dbl> 8.8, 9.5, 10.1, 9.9, 9.9, 10.1, 9.6, 8.8, 9.5, 11~
$ quality              <int> 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 5, 5, 7, 5, 7, 6~
```

Again, the goal is to estimate the alcohol contents (measured in alc/vol), but instead of using multiple linear regression, we use a penalized linear regression. First, we create a data split and cross-validation object:

```
set.seed(123)

split_wine <- initial_split(data_wine)
data_train_wine <- training(split_wine)
data_test_wine <- testing(split_wine)
```

3.1.1.1 Recipes

Instead of directly specifying a linear model, we describe how we want to fit any model to the data and which preprocessing steps we want to add. Compared to the formula method inside the `fit` function, the recipe can perform more preprocessing steps via `step_*()` functions without executing them directly. Independent of the type of model we fit (or train, for that matter), we can reuse the following recipe:

```
1 rec_wine <- recipe(
2   alcohol ~.,
3   data_train_wine
4 ) %>%
5   step_normalize(all_predictors())
```

In the first four lines, the `recipe` function specifies the formula (`alcohol ~.`) and template `data_train_wine`. The template initializes the recipe but can be changed later (e.g., when we apply these preprocessing steps to the test data). The `step_normalize` function in the fifth line normalizes the data passed to the recipe. Normalizing coefficients allows a comparison

between features invariant of the original scale. In other words, a large absolute value of a standardized coefficient strongly influences the model.

Calling the recipe then creates a summary of the input and operations that will be performed when training a model based on this recipe:

```
rec_wine
```

```
-- Recipe ----------------------------------------------------------------------



-- Inputs

Number of variables by role

outcome:    1
predictor: 11



-- Operations

* Centering and scaling for: all_predictors()
```

We can also use the recipe to preprocess directly. By passing the `rec_wine` recipe into the `prep()` and `bake` functions, the specified steps are applied to the provided data set.[1]

```
rec_wine %>%
  prep() %>%
  bake(data_train_wine) %>%
  glimpse()
```

[1]The function names `prep` and `bake` are references to literal baking recipes: You start out by specifying the ingredients for something you want to bake. Then you prepare the ingredients before finally baking the prepped ingredients.

```
Rows: 3,673
Columns: 12
$ fixed.acidity        <dbl> 1.74676532, -0.77276698, 1.02689895, -1.13270017,~
$ volatile.acidity     <dbl> -0.28262884, 0.01405471, 0.01405471, 0.40963277, ~
$ citric.acid          <dbl> -0.02709580, 0.96236127, 2.03427309, 0.46763273, ~
$ residual.sugar       <dbl> -0.75671284, 0.23276240, 1.14307962, -0.59839680,~
$ chlorides            <dbl> 0.326331917, -0.035460154, 0.009763855, 3.0849964~
$ free.sulfur.dioxide  <dbl> -1.35630369, 0.61777490, 1.43063080, -0.65957007,~
$ total.sulfur.dioxide <dbl> -1.56230134, 1.52596077, 0.91302325, 0.04076601, ~
$ density              <dbl> 0.01579382, 0.57999326, 1.43797700, -0.19786750, ~
$ pH                   <dbl> -1.96826221, 0.46863813, -0.71688096, -0.65101879~
$ sulphates            <dbl> -0.093372770, -0.267595877, -0.267595877, -0.3547~
$ quality              <dbl> -0.9943613, 0.1353291, -0.9943613, 0.1353291, -0.~
$ alcohol              <dbl> 9.5, 9.2, 8.9, 9.2, 10.1, 10.9, 11.0, 10.2, 13.2,~
```

Note that by applying the `prep` and `bake` functions, the new data set now contains the variable `quality` as an ordinal feature.

3.1.1.2 Workflows

Workflows combine model specifications and recipes. A workflow object can be defined using the `workflow` function. Adding a recipe is as simple as calling the function `add_recipe` and specifying the desired recipe. Calling the workflow object shows that a recipe is present, but a model is still missing.

```
wf_wine <- workflow() %>%
  add_recipe(rec_wine)

wf_wine
```

```
== Workflow ====================================================================
Preprocessor: Recipe
Model: None

-- Preprocessor ----------------------------------------------------------------
1 Recipe Step

* step_normalize()
```

Specifying an MLR model and adding it to the workflow is achieved by calling the `add_model` function:

```
lm_spec<- linear_reg()

wf_wine <- wf_wine %>%
  add_model(lm_spec)
```

We can then finally train the model using the `fit` function again:

```
lm_fit_res <- wf_wine %>%
  fit(data_train_wine)
```

The whole process can be visualized as follows:

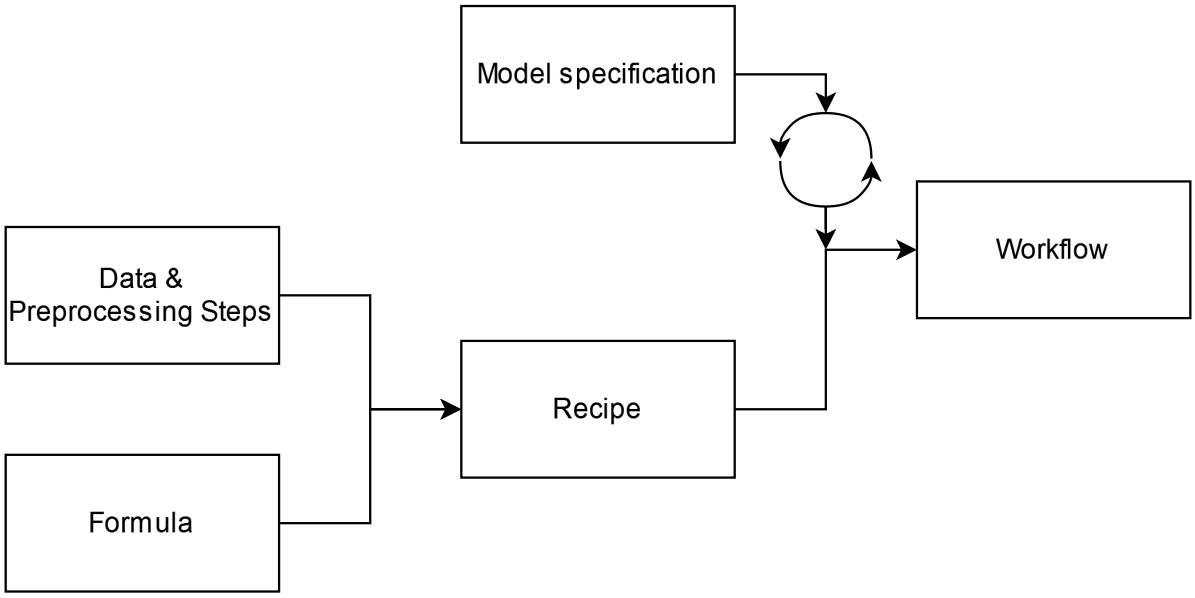


Figure 3.1: Components of a workflow object

The circling arrows symbolize that the model specification can be swapped, meaning that we can simply specify a different model and replace it in the workflow.

3.1.2 Tuning an elastic net regression model

3.1.2.1 Training an elastic net regression model

Similar to specifying a unpenalized linear regression model, we can specify an elastic net model with the `linear_reg()`-function as well:

```
lm_elnet_spec <- linear_reg(
  penalty = 0.05,
  mixture = 0.6
  ) %>%
  set_engine("glmnet")
```

Here, the `mixture` argument $\alpha \in [0, 1]$ specifies the ratio of ridge and lasso penalty in the elastic net. The hyperparameter λ specifies the `penalty` in both terms of the elastic net loss function:

$$\mathcal{L}_{\text{el\_net}}(\beta, \lambda, \alpha) = \sum_{n=1}^{N}(y_n - \hat{y}_n)^2 + \lambda \left(\alpha \sum_{i=1}^{k} \beta_i^2 + (1 - \alpha) \sum_{i=1}^{k} \|\beta_i\| \right) \tag{3.1}$$

Note, that we do not penalize the coefficient β_0!

Setting `mixture = 0` indicates the model is a ridge regression, while `mixture=1` corresponds to lasso regression.

In the specification above, $\alpha = 0.6$ and $\lambda = 0.05$.

Using the `set_engine("glmnet")` command, we specify that the elastic net regression is performed using the `{glmnet}` package. Besides the `{glmnet}` library, there are around 13 other packages for fitting linear models. However, it is far from the scope of this manuscript to discuss every other library in detail.

To fit the elastic net regression model, swap the model specification `lm_spec` with the model specification of the elastic net regression `lm_elnet_spec` in the workflow using `update_model` and pass the output to the `fit` function.

```
lm_elnet_fit_res <- wf_wine %>%
  update_model(lm_elnet_spec) %>%
  fit(data_train_wine)
```

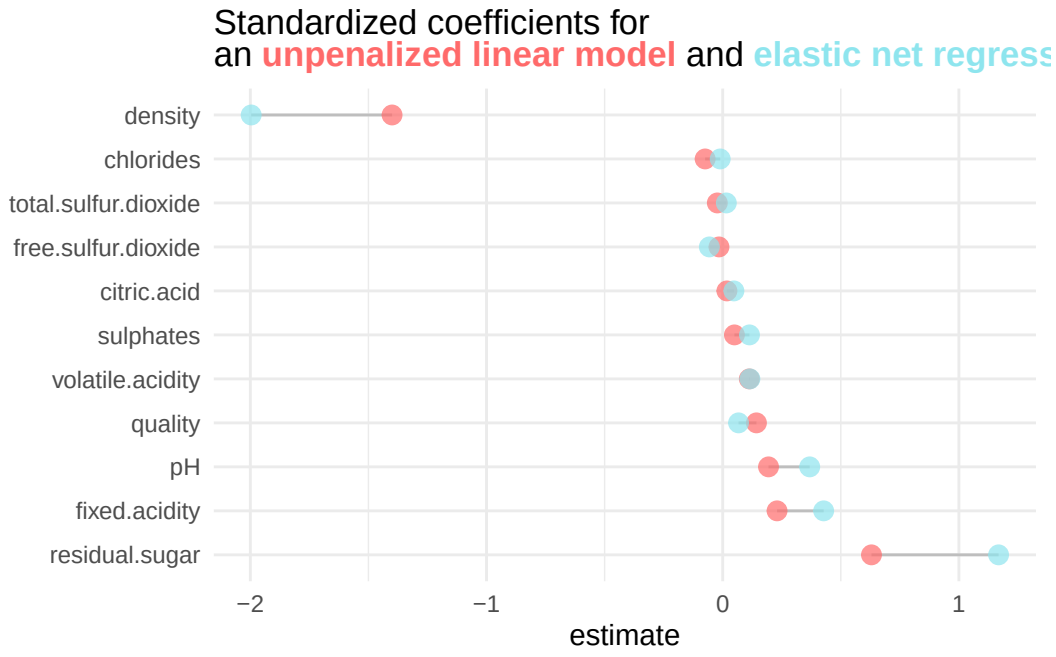
The estimated coefficients can then be viewed using the `tidy()` function:

```
lm_elnet_fit_res %>%
  tidy()
```

```
# A tibble: 12 x 3
   term                estimate penalty
   <chr>                  <dbl>   <dbl>
 1 (Intercept)            10.5     0.05
```

```
 2 fixed.acidity          0.230     0.05
 3 volatile.acidity       0.113     0.05
 4 citric.acid            0.0178    0.05
 5 residual.sugar         0.630     0.05
 6 chlorides             -0.0745    0.05
 7 free.sulfur.dioxide   -0.0158    0.05
 8 total.sulfur.dioxide  -0.0229    0.05
 9 density               -1.40      0.05
10 pH                     0.194     0.05
11 sulphates              0.0496    0.05
12 quality                0.143     0.05
```

Comparing the model coefficients of the penalized model with the unpenalized model yields:



Judging by the figure above, the variables `density` and `residual_sugar` are affected most by the penalisation.

3.1.2.2 Determining an optimal value for λ

The penalty value `0.05` and mixing parameter `0.6` were chosen arbitrarily. Depending on each of the parameters, the performance of the penalized model might increase or decrease. It is, therefore, essential to find optimal hyperparameters!

Hyperparameter tuning aims to streamline this approach. Instead of *randomly* choosing a penalty and mixture value, we can search for an optimal one by trying out a range of different penalties and choosing the optimal one within this range. Usually, one deploys an equidistant grid of candidate values. For example, if the penalty can take any value in the interval $[0, 1]$, then try $\lambda \in \{0, \frac{1}{n}, \frac{2}{n}, ..., \frac{n}{n}\}$, where $n \in \mathbb{N}$. However, this approach comes with challenges as well. Consider the following hypothetical example where an equidistant grid with $n = 8$ is chosen.

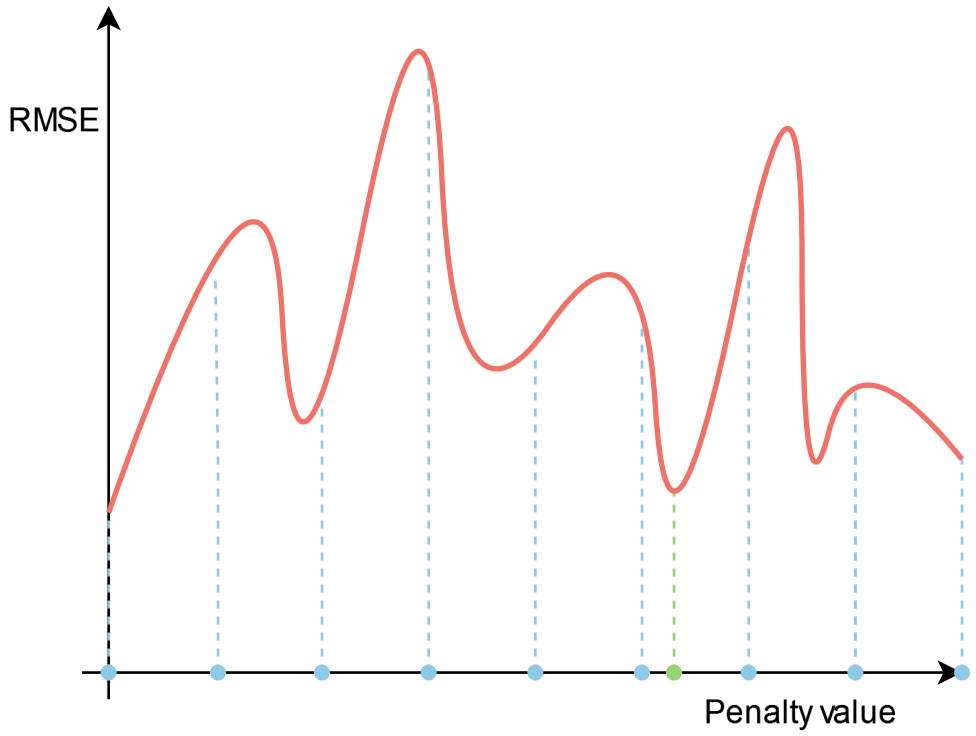


Figure 3.2: Hypothetical example of the model performance depending on the penalty value

The chosen grid (blue dots) does not cover the optimal penalty value (green dot). A solution could be increasing the grid size. However, more complex models require more computing time, so testing many different grid values eventually becomes unfeasible

It is, therefore, important to balance grid size and compute time to obtain satisfactory results.

3.1.2.3 Tuning for an optimal penalty value

When computational resources are available, it is encouraged to perform hyperparameter tuning via cross-validation instead of a single train/validation/test split.

```
folds_wine <- vfold_cv(data_train_wine,10)
```

Using the `{tidymodels}` framework makes hyperparameter tuning fairly straightforward, even when using cross-validation. By setting a hyperparameters in the model specification to `tune()`, we indicate that a parameter should later be tuned.

For the specific example of elastic net regression, we can achieve that using the following code snippet:

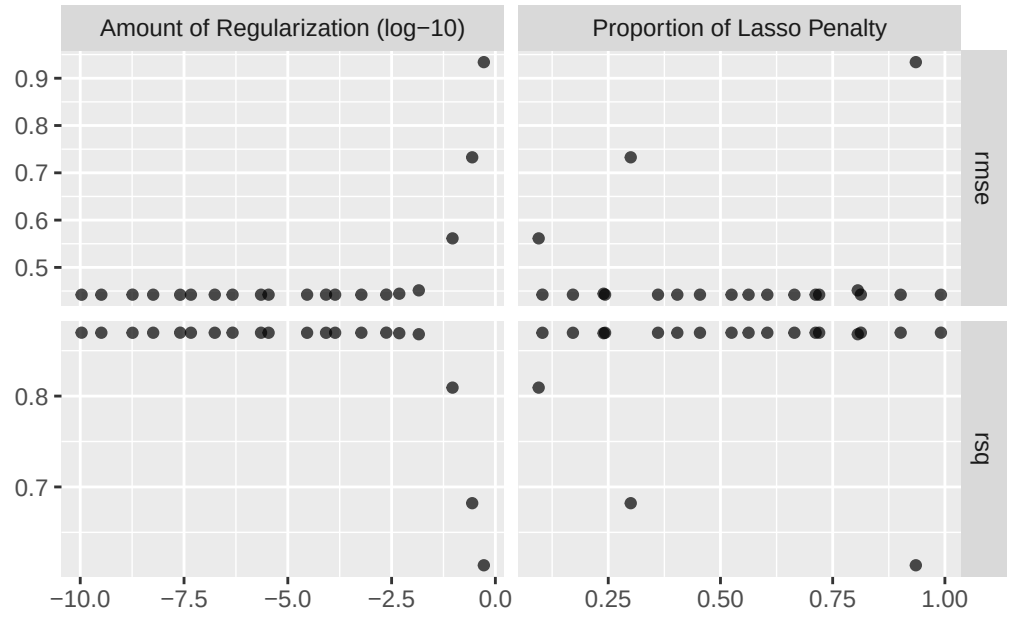
```
lm_elnet_spec_tune <- linear_reg(
  penalty = tune(),
  mixture = tune()
  ) %>%
  set_engine("glmnet")
```

By updating the workflow and subsequently training the model for different hyperparameters on each split using the `tune_grid` function, we obtain the results of hyperparameter tuning.

```
lm_elnet_tune_fit_res <- wf_wine %>%
  update_model(lm_elnet_spec_tune) %>%
  tune_grid(
    resamples = folds_wine,
    grid = 20
  )
```

Setting `grid = 20` automatically generates 20 different candidate hyperparameters. We can analyze the results using the `autoplot()`-function:

```
lm_elnet_tune_fit_res %>% autoplot()
```



It seems like an increasing regularisation generally leads to worse performance on the validation sets. On the other hand, there does not seem to be a clear influence of the `mixture` parameter on the model performance based on this limited sample. We should, therefore, check if increasing the hyperparameter combinations yields some more insights.

By setting a grid manually, we can specify the parameters and values to be tuned explicitly:

```
hyper_grid <- expand.grid(
  penalty = seq(0,1,length.out=15),
  mixture = seq(0,1,length.out = 15)
  ) %>%
  as_tibble()
```

Note, that we used the `expand.grid()`-function prior to converting the data into a tibble, to ensure that we generate evey possible combination of penatly and mixture values. In the example above, we created a parameter grid of size $15^2 = 225$. While this approach is fairly ineffecitve it sufficses for the purpose of this simple experiment.

```
lm_elnet_tune_fit_res <- wf_wine %>%
  update_model(lm_elnet_spec_tune) %>%
  tune_grid(
    resamples = folds_wine,
    grid = hyper_grid,
```

```
    metrics = metric_set(mae)
  )
```

By setting `metrics = metric_set(mae)`, we specify that the metric for evaluating the model is the mean absolute error.

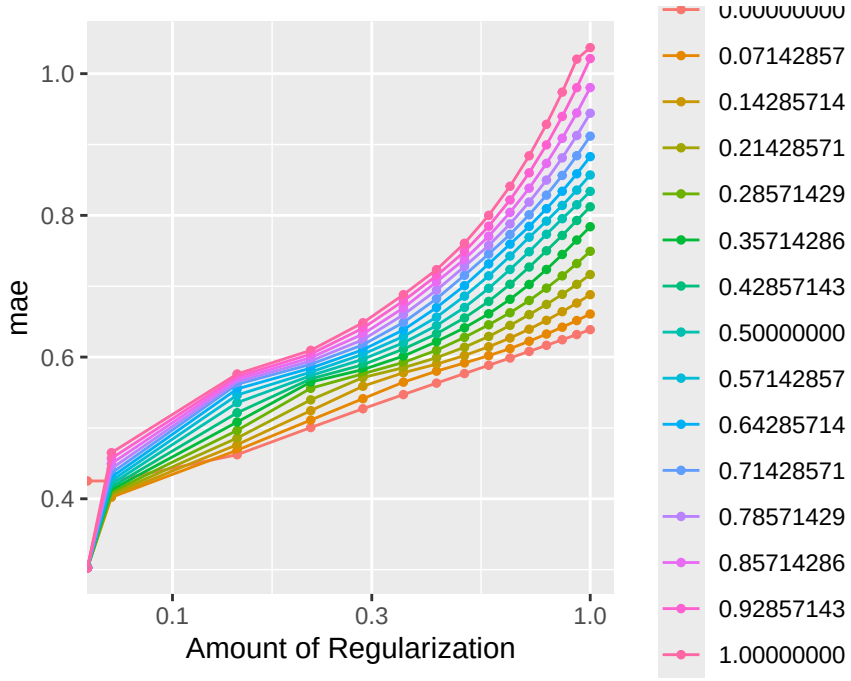
Calling the results of model tuning returns a nested data frame consisting of 10 rows and 4 columns. Each row contains information information like the metric about the underlying split for each tested hyperparameter.

```
lm_elnet_tune_fit_res
```

```
# Tuning results
# 10-fold cross-validation
# A tibble: 10 x 4
   splits             id     .metrics           .notes
   <list>             <chr>  <list>             <list>
 1 <split [3305/368]> Fold01 <tibble [225 x 6]> <tibble [0 x 3]>
 2 <split [3305/368]> Fold02 <tibble [225 x 6]> <tibble [0 x 3]>
 3 <split [3305/368]> Fold03 <tibble [225 x 6]> <tibble [0 x 3]>
 4 <split [3306/367]> Fold04 <tibble [225 x 6]> <tibble [0 x 3]>
 5 <split [3306/367]> Fold05 <tibble [225 x 6]> <tibble [0 x 3]>
 6 <split [3306/367]> Fold06 <tibble [225 x 6]> <tibble [0 x 3]>
 7 <split [3306/367]> Fold07 <tibble [225 x 6]> <tibble [0 x 3]>
 8 <split [3306/367]> Fold08 <tibble [225 x 6]> <tibble [0 x 3]>
 9 <split [3306/367]> Fold09 <tibble [225 x 6]> <tibble [0 x 3]>
10 <split [3306/367]> Fold10 <tibble [225 x 6]> <tibble [0 x 3]>
```

Again, using the `autoplot()`-function we can visualize some finding right away:

```
lm_elnet_tune_fit_res %>%
  autoplot()
```



Considering the figure above, the cross-validation MAE (CV-MAE) increases for each parameter combination different from $(0, 0)$. Furthermore, the increase in CV-MAE seems to be more profound as the lasso proportion increases.

This is an example where penalizing model parameters solely for model performance does not seem to be beneficial. However, if we wish to extract the optimal hyperparameter values, we can proceed as follows:

```
(hyper_opt <- select_best(lm_elnet_tune_fit_res,
                          metric = "mae")
 )
```

```
# A tibble: 1 x 3
  penalty mixture .config
    <dbl>   <dbl> <chr>
1       0       1 Preprocessor1_Model211
```

Unsurprisingly, the optimal penalty is $\lambda = 0$. Since the CV-MAE is indifferent of the mixture parameter when $\lambda = 0$, the mixture parameter is set to 1.

We can also choose λ according to the "one-standard error" rule:

```
(hyper_opt_osr <- select_by_one_std_err(lm_elnet_tune_fit_res,
                            metric = "mae",
                            desc(penalty))
 )
```

```
# A tibble: 1 x 3
  penalty mixture .config
    <dbl>   <dbl> <chr>
1       0  0.0714 Preprocessor1_Model016
```

The optimal penalty according to the "one-standard error" rule, which selects the most simple model within one standard error of the numerically optimal results, is different from 0.

> Pro Tip
>
> You can use the `pull()`-function in a similar fashion as `pluck` to extract the raw values of a column. While the `pluck()`-function requires passing a string or the column index, the `pull()`-function works fine with passing the column name verbatim.
>
> ```
> hyper_opt %>%
>   pull(penalty)
> ```
>
> ```
> [1] 0
> ```
>
> ```
> hyper_opt_osr %>%
>   pull(penalty)
> ```
>
> ```
> [1] 0
> ```

3.1.2.4 Training a final model with the chosen hyperparameter values

After determining the best hyperparameter, a final model can be trained on the whole training data and evaluated on the testing data.

We can use the `finalize_workflow()`-function to set the hyperparameters specified in `hyper_opt_osr` for a final model that is trained on the entire training data. By passing this final workflow to the `last_fit`-function, we fit this final model.

```
lm_elnet_spec_final_fit <- finalize_workflow(
  wf_wine,
  hyper_opt_osr
  ) %>%
  last_fit(
    split = split_wine,
    metrics = metric_set(mae)
    )
```

The metrics can then be extracted using the `collect_metrics` function.

```
lm_elnet_spec_final_fit %>%
  collect_metrics()
```

```
# A tibble: 1 x 4
  .metric .estimator .estimate .config
  <chr>   <chr>          <dbl> <chr>
1 mae     standard       0.302 Preprocessor1_Model1
```

A visualization of the steps for effectively tuning the penalty value in a ridge regression setting can be found below.

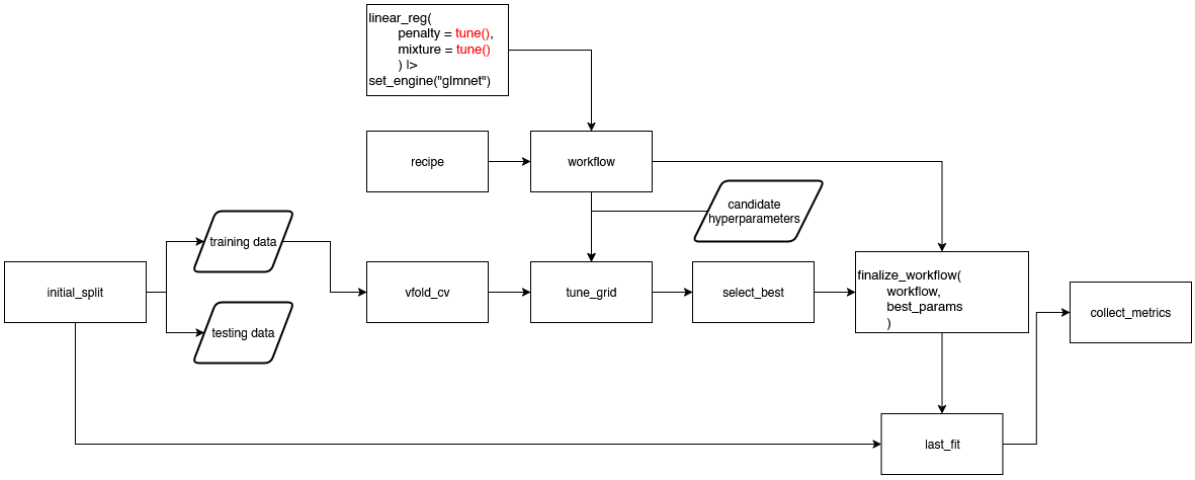


Figure 3.3: Tuning procedure for the penalty in ridge regression

3.2 Exercises

3.2.1 Theoretical Exercises

Exercise 3.1. The goal of this initial exercise is to review some theoretical aspects of OLS, ridge, and lasso regression.

In Statistics I/II, we learned that OLS is the cornerstone of linear regression analysis. It allows us to explore and quantify the relationship between the response variable and the regressors in a relatively simple but meaningful way. We can extend the idea of a simple linear regression by adding a penalty term to the loss function we want to minimize. This process is called regularization and has been introduced in the lecture regarding ridge and lasso regression.

Consider a simple linear model with a quantitative response variable Y and a single predictor X. The simple linear model then assumes (among other things) that there is approximately a linear relationship between Y and X, i.e.,

$$Y \approx \beta_0 + \beta_1 X.$$

with unknown coefficients β_0, β_1. To obtain the best estimate β_0 and β_1 for a given sample we can minimize the MSE

$$\min_{\beta_0,\beta_1} MSE = \frac{1}{N}\sum_{n=1}^{N}(y_n - (\beta_0 + \beta_1 x_n))^2 \tag{3.2}$$

where $N = \text{length(Y)}$, $y_1, ..., y_N$ is a realized sample of Y, and $x_1, ..., x_N$ is a realized sample of X.

Show, that

$$\begin{aligned}\hat{\beta}_1 &= \frac{\sum_{n=1}^{N}(x_n - \bar{x})(y_n - \bar{y})}{\sum_{n=1}^{N}(x_n - \bar{x})^2} \\ \hat{\beta}_0 &= \bar{y} - \hat{\beta}_1 \bar{x}.\end{aligned}$$

with $\bar{x} = \frac{1}{N}\sum_{n=1}^{N} x_n$ and $\bar{y} = \frac{1}{N}\sum_{n=1}^{N} y_n$ minimizes the minimization problem above. You can assume that the critical points calculated using the partial derivatives are in fact minima and that $\sum_{n=1}^{N}(x_n - \bar{x})^2 \neq 0$.

Exercise 3.2. Explain the meaning of the following meme in relation to ridge regression and OLS regression:

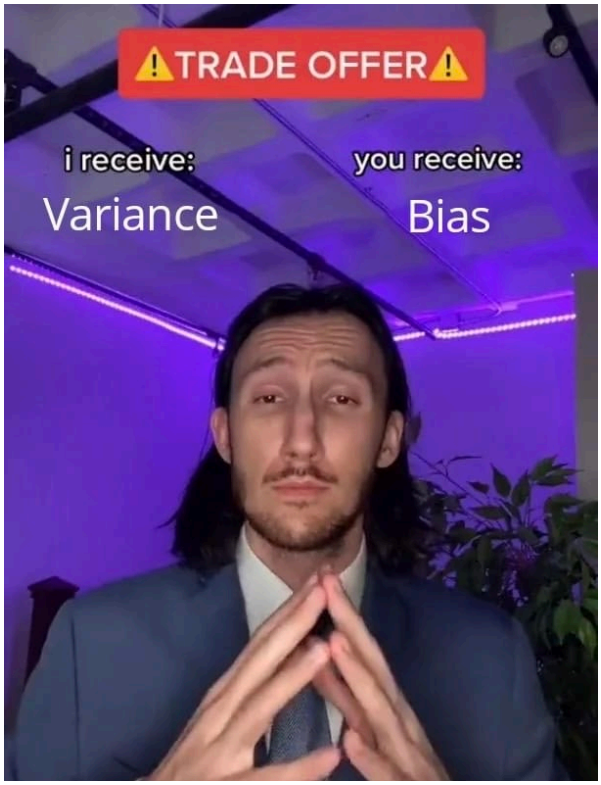


Figure 3.4: Source

Exercise 3.3. Consider the following statements and decide whether ridge or lasso regression should be applied.

1. You are building a predictive model for stock price prediction, and you have a large number of potential predictors. Some of these predictors might be highly correlated with each other.
2. You are modeling housing prices, and you want to prevent the model from overfitting to the training data.
3. You are working on a marketing project where you have a dataset with a mix of numerical and categorical features. You need to build a regression model to predict customer lifetime value.

Exercise 3.4. Come up with a scenario where a mixed model, i.e. an elastic net might be a good choice.

3.2.2 Programming Exercises

In the following exercises, we will revisit the ImmoScout data set Apartment rental offers in Germany, but instead of considering rent prices in Augsburg, we will now consider rent prices in Munich. The data set can be downloaded using the button below.

Download MunichRental

It contains 4383 unique rental listings for flats in Munich, sourced on three dates in 2018 and 2019, and 28 different variables. We will briefly prepare the data set, create a recipe, and create a workflow before training and tuning different models.

Exercise 3.5. Import the data set and apply the following steps to the data:

1. Remove the following columns from the data set

```
c("serviceCharge","heatingType","picturecount","totalRent",
        "firingTypes","typeOfFlat","noRoomsRange", "petsAllowed",
        "livingSpaceRange","regio3","heatingCosts", "floor",
        "date", "pricetrend")
```

2. Remove all the `NA` values.
3. Next, mutate the data as follows:
    1. Convert the feature `interiorQual` to an ordered factor variable using the following levels:

```
c("simple", "normal", "sophisticated", "luxury")
```

    2. Convert the feature `condition` to an ordered factor variable using the following levels:

```
c("need_of_renovation", "negotiable","well_kept","refurbished",
         "first_time_use_after_refurbishment",
         "modernized", "fully_renovated", "mint_condition",
         "first_time_use")
```

    3. Convert the feature `geo_plz` to an unordered factor.
    4. Convert every logical feature into a numerical feature such that `TRUE` corresponds to `1` and `FALSE` to `0`.
    5. Remove any flat that costs more than 4000 EUR or is bigger than 200 m^2 from the data set.

Exercise 3.6. Use the seed `24` and create a training and test split with 80% training data based on the newly mutated data set. Then, with the same seed, create a 5-fold cross-validation data set from the training data.

Exercise 3.7. Explain the following recipe by describing each function.

```
rec_rent <- recipe(
    formula = baseRent ~.,
    data = data_train
  ) %>%
  update_role(scoutId, new_role = "ID") %>%
  step_ordinalscore(interiorQual, condition)%>%
  step_dummy(geo_plz)%>%
  step_zv(all_predictors())
```

Exercise 3.8. In the previous exercises, we initially mutated the data set (cf. Exercise 3.5) before adding mutations using the `recipe` function. This procedure can add unnecessary complexity or confusion since the preprocessing steps are spread across the markdown document.

Therefore, modify the code of Exercises Exercise 3.6 and Exercise 3.7 so that the preprocessing steps of Exercise 3.5 are included in the recipe.

Hint: You can use the `step_select`*,* `step_mutate`*,* `step_naomit`*,* `step_novel`*, and* `step_dummy` *functions for incorporating the preprocessing steps.*

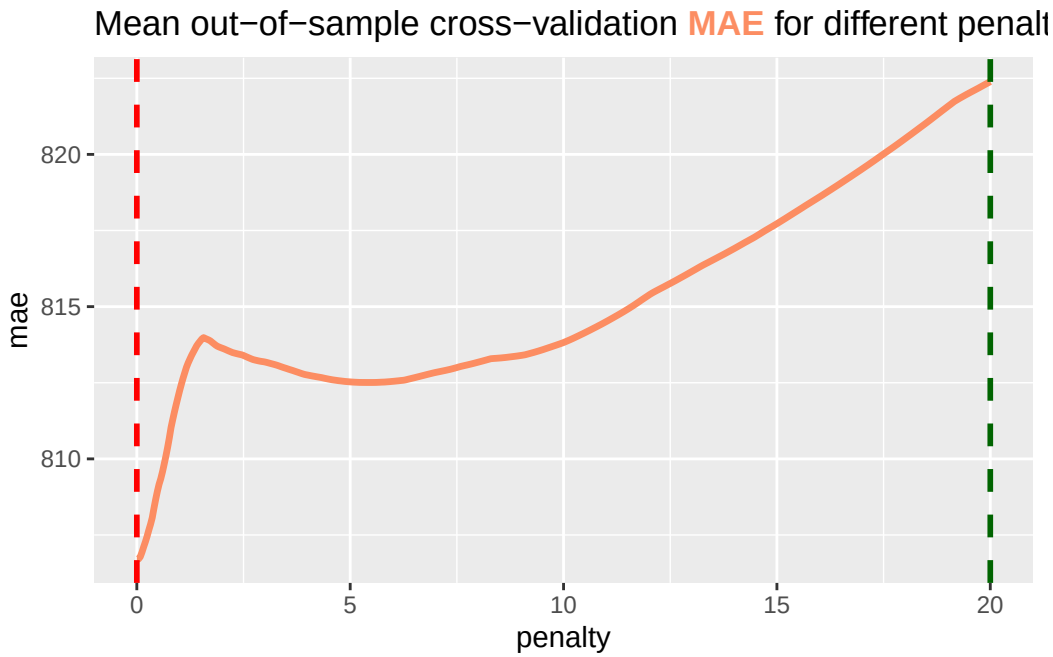
Exercise 3.9. Create a lasso model with penalty set to `tune`. Then, use the following penalty values for tuning the lasso model. Finally, create a new workflow object to add the model specification and recipe.

```
penalty_vals <- tibble(penalty = seq(0, 20, length.out = 500))
```

Exercise 3.10. Train the lasso model using cross-validation across all penalty values. Then, evaluate the results by finding the *optimal penalty value* and the *optimal penalty value according to the one standard error rule* with respect to the metric MAE.

Exercise 3.11. Consider the following plot, which depicts the mean out-of-sample RMSE for different penalty values. The dashed lines represent the optimal penalty and the largest penalty value, such that the mean MSE is within one standard error of the optimum.

Decide and present an argument for which line is the optimal penalty. Furthermore, explain why we would choose the non-optimal penalty in lasso regression.



Exercise 3.12. Given the optimal penalty, train the Train a lasso model with the optimal penalty value on the whole training data and find out which parameters are set to 0. Then, evaluate the model on the test data by calculating the out of sample MAE and R^2.

3.3 Solutions

Solution 3.1 (Exercise 3.1). First, calculate the partial derivatives of

$$L(\beta_0, \beta_1) = \frac{1}{N} \sum_{n=1}^{N} (y_n - (\beta_0 + \beta_1 x_n))^2 \tag{3.3}$$

with respect to β_1 and β_0.

$$
\begin{aligned}
\frac{\partial}{\partial \beta_0} L(\beta_0, \beta_1) &= \frac{1}{N} \sum_{n=1}^{N} -2(y_n - (\beta_0 + \beta_1 x_n)) \\
&= -2\bar{y} + 2\beta_0 + 2\beta_1 \bar{x} \\
&\stackrel{!}{=} 0 \\
\frac{\partial}{\partial \beta_1} L(\beta_0, \beta_1) &= \frac{1}{N} \sum_{n=1}^{N} -x_n(2(y_n - (\beta_0 + \beta_1 x_n))) \\
&= -2\overline{xy} + 2\frac{1}{N} \sum_{n=1}^{N} x_n \beta_0 + \beta_1 x_n^2 \\
&= -2\overline{xy} + 2\beta_0 \bar{x} + 2\beta_1 \overline{x^2} \\
&\stackrel{!}{=} 0
\end{aligned}
$$

Solving the first term for β_0 yields

$$\beta_0 = \bar{y} - \beta_1 \bar{x}.$$

In order to solve the second term for β_1 we can utilize this newly acquired representation of β_0:

$$-2\overline{xy} + 2\beta_0 \bar{x} + 2\beta_1 \overline{x^2} = -2\overline{xy} + 2(\bar{y} - \beta_1 \bar{x})\bar{x} + 2\beta_1 \overline{x^2} \stackrel{!}{=} 0.$$

Then,

$$
\begin{aligned}
-2\overline{xy} + 2(\bar{y} - \beta_1 \bar{x})\bar{x} + 2\beta_1 \overline{x^2} &= -2\overline{xy} + 2\bar{y}\bar{x} - 2\beta_1 \bar{x}^2 + 2\beta_1 \overline{x^2} \\
&\stackrel{!}{=} 0.
\end{aligned}
$$

This can easily be solved for β_1, which yields

$$\beta_1 = \frac{\sum_{n=1}^{N} (x_n - \bar{x})(y_n - \bar{y})}{\sum_{n=1}^{N} (x_n - \bar{x})^2}.$$

Note, that

$$\begin{aligned}\sum_{n=1}^{N}(x_n-\bar{x})(y_n-\bar{y}) &= \sum_{n=1}^{N} x_n y_n - \bar{x}y_n - x_n\bar{y} + \bar{x}\bar{y} \\ &= N(\overline{xy} - \bar{x}\bar{y} - \bar{x}\bar{y} + \bar{x}\bar{y}) \\ &= N(\overline{xy} - \bar{x}\bar{y})\end{aligned}$$

and

$$\begin{aligned}\sum_{n=1}^{N}(x_n-\bar{x})^2 &= \sum_{n=1}^{N} x_n^2 - 2x_n\bar{x} + \bar{x}^2 \\ &= N(\overline{x^2} - 2\bar{x}^2 + \bar{x}^2) \\ &= N(\overline{x^2} - \bar{x}^2)\end{aligned}$$

Solution 3.2 (Exercise 3.2). The key point addresses the bias-variance trade-off.

From the lecture, we know that

$$\hat{\beta}_{\text{ridge}}(\lambda) = \frac{\hat{\beta}_{\text{OLS}}}{1+\lambda}$$

if the features are standardized and orthogonal.

- Bias:
  By growing the parameter λ, the parameter $\hat{\beta}_{\text{OLS}}$ shrinks. In other words, the regularization term encourages the model to have smaller coefficient values, which means it may not fit the training data as closely as an unregularized model. This means that systematic errors are introduced to the model's predictions.
- Variance:
  By growing the parameter λ, we reduce variance by shrinking the coefficients' values, which discourages them from taking very high values. This effectively constrains the model's complexity and makes it less prone to overfitting.

Solution 3.3 (Exercise 3.3).

1. Lasso regression should be used in this scenario because it can perform feature selection by driving some coefficients to zero. This is especially helpful if there are many features as it helps in dealing with correlated predictors.
2. Ridge regression is more suitable because it provides a smoother and more continuous shrinkage of coefficients, which reduces the risk of overfitting.

3. Lasso regression might be a more suitable choice as it can perform feature selection and even drive the coefficient for some categorical values to 0.

Solution 3.4 (Exercise 3.4). A healthcare dataset is given to predict a patient's readmission probability with numerous correlated features. The aim is to build a model that predicts accurately, selects the most relevant features, and mitigates multicollinearity. Here, an elastic net is the preferred choice because it combines Ridge and Lasso regression, effectively handling multicollinearity while performing feature selection, making it ideal for this complex healthcare dataset.

Solution 3.5 (Exercise 3.5).

```
data_muc_filtered <- data_muc %>%
  select(!c("serviceCharge","heatingType","picturecount","totalRent",
            "firingTypes","typeOfFlat","noRoomsRange", "petsAllowed",
            "livingSpaceRange","regio3","heatingCosts", "floor",
            "date", "pricetrend")) %>%
  na.omit %>%
  mutate(
    interiorQual = factor(
      interiorQual,
      levels = c("simple", "normal", "sophisticated", "luxury"),
      ordered = TRUE
    ),
    condition = factor(
      condition,
      levels = c("need_of_renovation", "negotiable","well_kept","refurbished",
                 "first_time_use_after_refurbishment",
                 "modernized", "fully_renovated", "mint_condition",
                 "first_time_use"),
      ordered = TRUE
    ),
    geo_plz = factor(geo_plz)
  ) %>%
  filter(baseRent <= 4000, livingSpace <= 200)
```

Solution 3.6 (Exercise 3.6).

```
set.seed(24)

split <- initial_split(data_muc_filtered, 0.8)
data_train <- training(split)
```

```
data_test <- testing(split)

folds <- vfold_cv(data_train, v = 5)
```

Solution 3.7 (Exercise 3.7).

1. We first set up the recipe by specifying the formula and data used in each step. Note, that by using the expression `baseRent ~.` we indicate that we want to fit every variable in the `data_train` dataset on the dependent variable `baseRent`.
2. The `update_role` function assigns the feature `scoutId` to a new role called `ID`. By doing so, the feature `scoutId` is no longer used as a predictor and will no longer influence our model. We will still keep it in the loop, however in case we want to access a specific listing that is only accessible using the unique `scoutId`.
3. We then convert the factors `interiorQual` and `condition` to ordinal scores by using the `step_ordinalscore` function. The translation uses a linear scale, i.e. for the feature `interiorQual` the level `simple` corresponds to the value `0` and `luxury` corresponds to the value `4`.
4. Afterward, we create dummy variables for each zip code. Here, every zip code in the `data_train` is treated as a new variable. Thus, we are basically replacing the feature `geo_plz` with 82 new features, each representing one of the 82 zip codes available in the training data.
5. The `step_zv` (zero variance filter) function removes all variables that contain only a single value. If, for example, a zip code does not occur in any of the entries of `data_train`, the whole column will be set to zero and effectively not affect our model. Thus it is in our best interest to remove those columns.

Solution 3.8 (Exercise 3.8). Creating a new data split based on the unmutated data.

```
set.seed(24)

split <- initial_split(data_muc, 0.8)
data_train <- training(split)
data_test <- testing(split)

folds <- vfold_cv(data_train, v = 5)
```

Inserting the previous preprocessing steps into the recipe framework. Note, that this only requires changing:

- `select` to `step_select`,

- `na.omit` to `step_naomit`,
- `mutate` to `step_mutate`, and
- `filter` to `step_filter`.

```
rec_rent <- recipe(
    formula = baseRent ~.,
    data = data_train
  ) %>%
  update_role(scoutId, new_role = "ID") %>%
  step_select(!c("serviceCharge","heatingType","picturecount","totalRent",
            "firingTypes","typeOfFlat","noRoomsRange", "petsAllowed",
            "livingSpaceRange","regio3","heatingCosts", "floor",
            "date", "pricetrend")) %>%
  step_naomit(all_predictors()) %>%
  step_mutate(
    interiorQual = factor(
      interiorQual,
      levels = c("simple", "normal", "sophisticated", "luxury"),
      ordered = TRUE
    ),
    condition = factor(
      condition,
      levels = c("need_of_renovation", "negotiable","well_kept","refurbished",
                 "first_time_use_after_refurbishment",
                 "modernized", "fully_renovated", "mint_condition",
                 "first_time_use"),
      ordered = TRUE
    ),
    geo_plz = factor(geo_plz),
    across(where(is.logical),as.numeric)
  ) %>%
  step_filter(baseRent <= 4000, livingSpace <= 200) %>%
  step_ordinalscore(interiorQual, condition)%>%
  step_novel(geo_plz)%>%
  step_dummy(geo_plz)%>%
  step_zv(all_predictors())
```

Solution 3.9 (Exercise 3.9).

```
model_lasso <- linear_reg(penalty = tune(), mixture = 1.0) %>%
  set_engine(engine = "glmnet", path_values = penalty_vals$penalty )
```

```
wf_rent <- workflow() %>%
  add_recipe(rec_rent) %>%
  add_model(model_lasso)
```

Solution 3.10 (Exercise 3.10).

```
lasso_tune_res <-
  wf_rent %>%
  tune_grid(
    grid = penalty_vals,
    metrics = multi_metric,
    resamples = folds
  )

tib <- lasso_tune_res %>% collect_metrics %>%
  pivot_wider(
    names_from = .metric,
    values_from = c(mean, std_err)
)

penalty_one_std <- select_by_one_std_err(
  lasso_tune_res,
  metric = "mae",
  desc(penalty)
  ) %>%
  pull(penalty)

penalty_opt <- select_best(lasso_tune_res, metric = "mae") %>% pull(penalty)

glue::glue("The optimal penalty is {round(penalty_opt,2)} and \n the optimal penalty accordi
```

```
The optimal penalty is 0 and
the optimal penalty according to the one standard error rule is 20.
```

Solution 3.11 (Exercise 3.11). The red line depicts the optimal penalty, since it intersects the minimum of the RMSE. Especially in Lasso regression an optimal penalty parameter is often smaller than we desire. The effect of a smaller penalty parameter is, that we do not eliminate as many features as we anticipated. By increasing the penalty we can effectively overcome this problem as more features are eliminated. A disadvantage however is, that we sacrefice out-of-sample performance, as the newly chosen penalty is not optimal anymore.

Solution 3.12 (Exercise 3.12).

```
glm_res_best<- lasso_tune_res %>%
  select_by_one_std_err(metric = "mae", desc(penalty))

best_penalty <- glm_res_best$penalty


last_glm_model <- linear_reg(penalty = best_penalty, mixture = 1.0) %>%
  set_engine("glmnet")

wf_rent_final <- wf_rent %>%
  update_model(last_glm_model)

last_glm_fit <-
  wf_rent_final %>%
  last_fit(split,
           metrics = multi_metric)

last_glm_fit %>%
  collect_metrics()
```

```
# A tibble: 2 x 4
  .metric .estimator .estimate .config
  <chr>   <chr>          <dbl> <chr>
1 rsq     standard       0.824 Preprocessor1_Model1
2 mae     standard     254.    Preprocessor1_Model1
```

4 Regression Trees

In this fourth exercise session, we want to introduce regression trees. Regression trees, up to this day, are still one of the most important statistical models as they address important aspects like interpretability but also versatility in terms of the prediction approach since they are non-linear. Another important aspect is their integration with other, more powerful statistical models like random forests, which we will cover in a later session.

Besides regression, tree-based models can also be used for classification tasks, however, in this exercise in particular, we will focus on the regression case and consider classification tasks later on.

Some of the packages we will use throughout the session.

```
library("tidyverse")
library("tidymodels")
```

4.1 Intermezzo: Imputation

When working with data sets, missing values can disrupt analysis and modeling processes. In previous exercises when working with the rental listing data set, we resorted to either remove missing values completely or (in case of numerical or ordinal features) assign the worst possible value. Both of these approaches have significant drawbacks:

- By removing every `NA` value we potentially remove a large chunk of the underlying data which weakens the training capabilities.
- Assigning the worst possible value to each missing observation introduces a bias which can distort the true underlying distribution.

To handle these drawbacks, several imputation techniques are commonly used:

1. Mean imputation replaces missing values with the average of the observed data for a given feature, offering simplicity but also risking distortion if the data contains outliers.
2. Median imputation substitutes missing entries with the median value, making it more robust against extreme values.

3. Linear imputation estimates the missing values using a linear model making it particularly robust if there are highly correlated features.

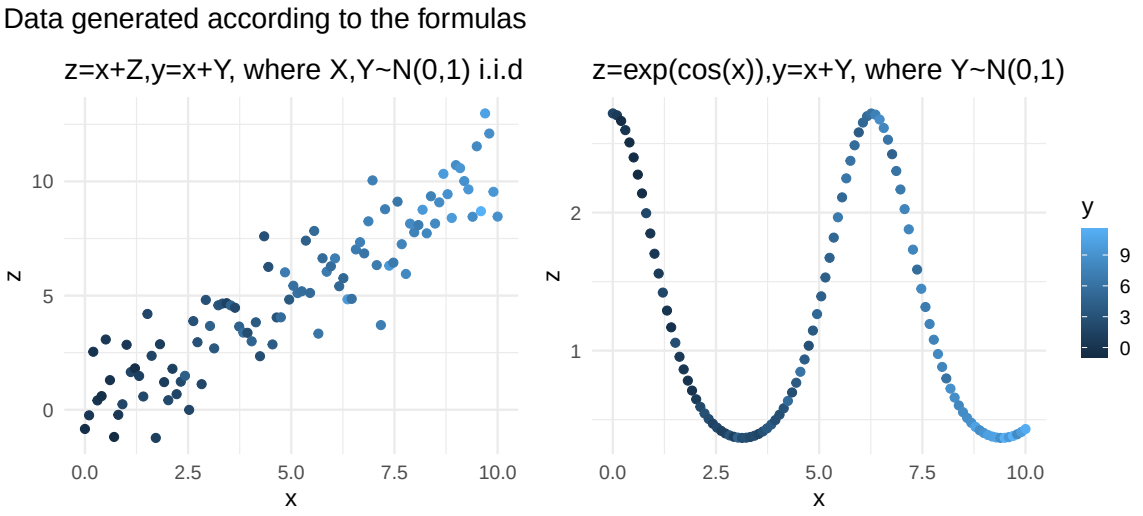
The approaches above require the missing data to be numerical, else we can't really calculate a mean or median.

4. k-nearest neighbors imputation identifies the k most similar records and fills missing values using their observed data. This approach can also handle ordinal or even nominal data.

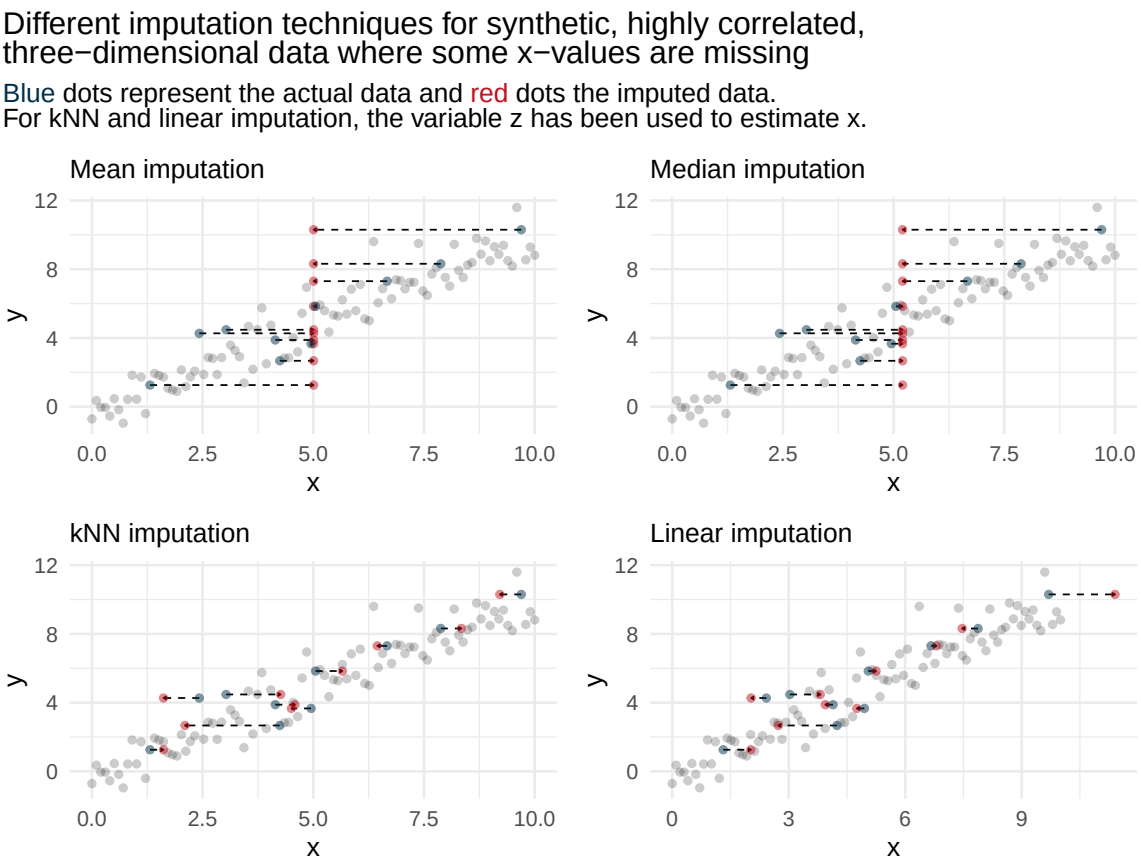
Consider the following two synthetic data set examples:

```
set.seed(123)

n<- 100

data_synth_high_corr <- tibble(
  x = seq(0,10,length.out = n),
  z = x+rnorm(n,0,1.5),
  y = x+rnorm(n),
)

data_synth_low_corr <- tibble(
  x = seq(0,10,length.out = n),
  z = exp(cos(x)),
  y = x+rnorm(n),
)
```

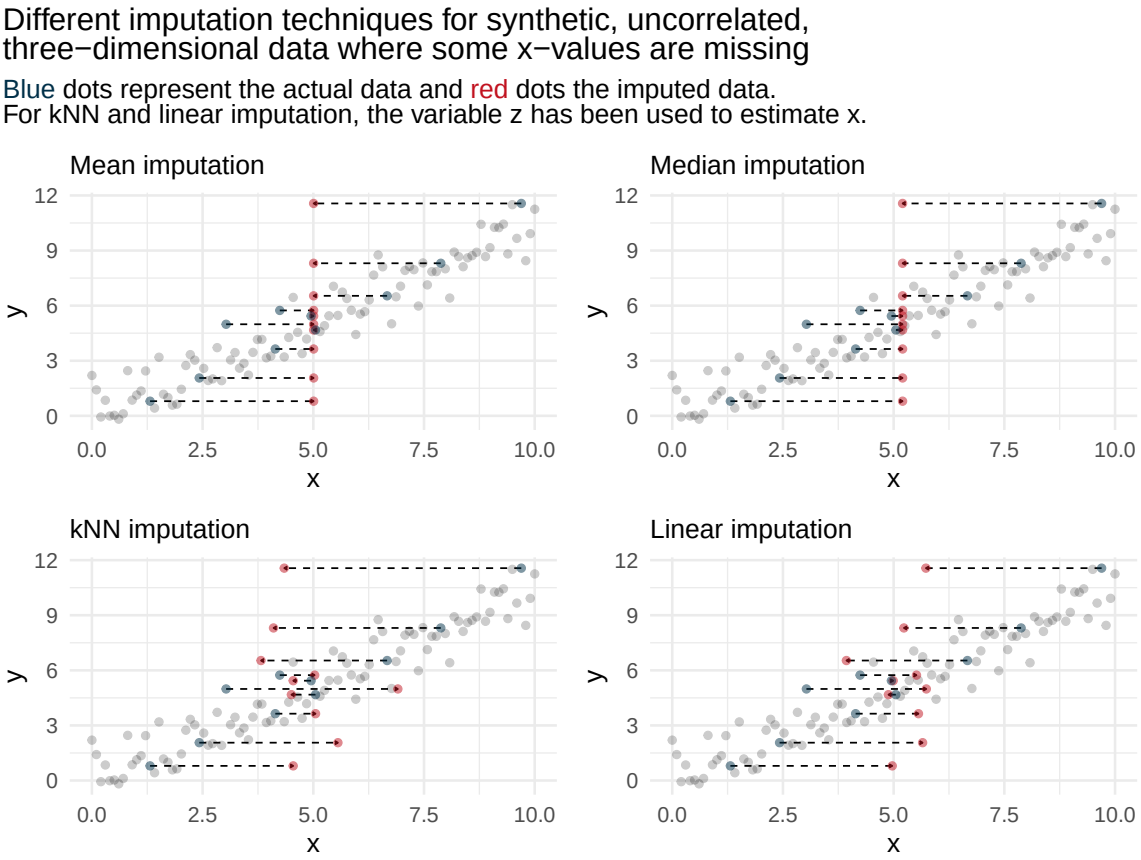
Both data sets consist of three variables x, y, z, where both, y and z are generated using x. In the first data set all three variables are highly correlated, whereas in the second data set only x and y are highly correlated. Depending on the complexity of the imputation technique, correlation between variables can determine how well they perform.



Consider the case where 10% of the x-values are removed. We can apply all four imputation techniques to impute those missing values:



In the figure above we can observe that the highly correlated data is estimated better using the more sophisticated imputation models compared to the simple mean and median imputation.



While sophisticated imputation techniques seem to outperform simple techniques on highly correlated data, simple imputation techniques seem to be a good choice if the different features are less correlated.

However, as a rule of thumb, if computational resources are available, I recommend trying different techniques and choose the one that yields best validation results.

4.2 Regression trees in R

We use the same white wine data set as in Session 02 for the introduction.

4.2.1 Data preprocessing, recipe and workflow creation

For detailed description of each parameter see Cortez et al.

The goal is to predict the alcohol contents of each wine given different attributes like density and residual sugar.

The following code cell contains the usual preprocessing steps required for fitting a model. For details, see tutorial on recipes and workflows.

```
set.seed(123)

data_wine <- read.csv("data/winequality-white.csv")

split_wine <- initial_split(data_wine)

data_train_wine<- training(split_wine)
data_test_wine<- testing(split_wine)

rec_wine <- recipe(
  alcohol ~.,
  data = data_train_wine
)

wf_wine <- workflow() %>%
  add_recipe(rec_wine)
```

4.2.2 Fitting a basic tree

Fitting a basic tree model is as simple as fitting any other model using the tidymodels framework. The `decision_tree` function creates an `{rpart}` tree object. Since decision entail regression and classification trees, we have to specify the mode as `"regression"`. Using the `fit` function, we can directly fit the model and evaluate it on the test set using the `augment` function and a preferred metric:

```
tree_spec_basic<- decision_tree(
  mode = "regression"
)

wf_wine <- wf_wine %>% add_model(tree_spec_basic)

tree_res_basic <- wf_wine %>%
  fit(data_train_wine)

tree_res_basic %>%
```

```
  augment(data_test_wine) %>%
  rmse(.pred,alcohol)
```

```
# A tibble: 1 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rmse    standard       0.632
```

Since we have not passed any hyperparameters, the following `rpart` specific default parameters are used:

- `min_n = 20`,
- `tree_depth = round(minsplit/3)`, and
- `cost_complexity = 0.01`,

To potentially obtain better results, the parameters should be tuned. We, therefore, have to define a new model specification and set the tuneable parameters to `tune()`.

```
tree_spec_tune <- decision_tree(
  mode = "regression",
  min_n = tune(),
  tree_depth = tune(),
  cost_complexity = tune()
)
```

In Exercise 03, Section 3.1.2.3, we specified the candidate hyperparameters by defining a data frame with the respective hyperparameter names and candidate values, or by setting the `grid` attribute in the `tune_grid` function to a positive number indicating the number of automatically created candidate parameters.

Instead of manually creating a data frame, we can use the `extract_parameter_set_dials()` function to extract all tuneable parameters and pass the output into the `grid_regular()` function. The `grid_regular()` function takes a positiv number `levels` $= n$ as an input and returns a data frame where each column contains n^m candidate values, where m is the number of tunable hyperparameters.

```
tree_grid <- tree_spec_tune %>%
  extract_parameter_set_dials() %>%
  grid_regular(levels = 5)
```

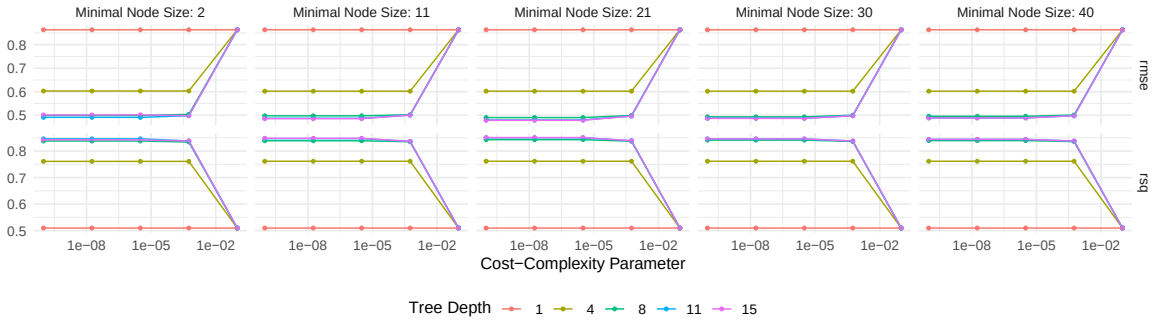
Since there are $m = 3$ hyperparameters and `levels` is set to 5, the data frame `tree_grid` contains 125 combinations of candidate hyperparameters.

After specifying the candidate hyperparameters, we can tune the models following the standard procedure:

```
set.seed(123)

folds_wine <-vfold_cv(data_train_wine,5)

multi_metric <- metric_set(rmse,rsq)

wf_wine <- wf_wine %>%
  update_model(tree_spec_tune)

tree_res_tune <- wf_wine %>%
  tune_grid(
    grid = tree_grid,
    resamples = folds_wine,
    metrics = multi_metric
  )
```

Given the tuning results, we can visualize our finding with the `autoplot()` function.

```
tree_res_tune %>% autoplot()+
  theme_minimal(base_size = 12)+
  theme(legend.position = "bottom")
```



Using the `select_best()` function, we can select the best set of hyperparameters. After extracting these, passing them to the `finalize_workflow()` function together with the workflow object, updates the finalizes the workflow object by replacing the hyperparameters set to `tune()` with the selected hyperparameters. Passing the finalized workflow into the `last_fit()`

function together with the whole data split fits the final model on the whole training data and evaluates it on the test data.

```
tree_res_final <- tree_res_tune %>%
  select_best(metric = "rmse") %>%
  finalize_workflow(wf_wine,.) %>%
  last_fit(split_wine)

tree_res_final %>%
  collect_metrics()
```

```
# A tibble: 2 x 4
  .metric .estimator .estimate .config
  <chr>   <chr>          <dbl> <chr>
1 rmse    standard       0.496 Preprocessor1_Model1
2 rsq     standard       0.837 Preprocessor1_Model1
```

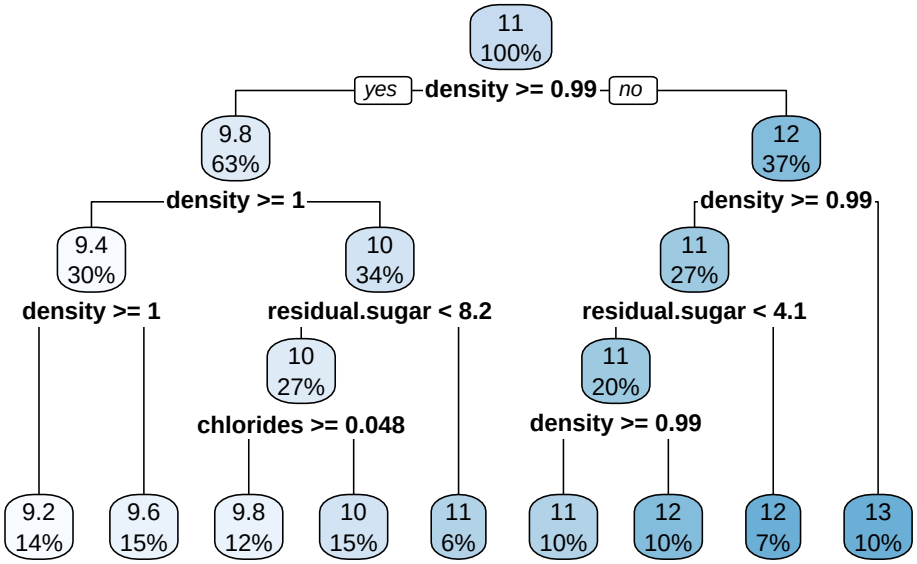
Comparing the RMSE of the basic model (0.632) with the tuned model's RMSE (0.496) shows that the tuning improved the performance of the tree!

4.2.3 Visualizing results

Since the basic tree only has a depth parameter with value 5 compared to the tuned tree with depth 15, we will consider the basic tree for a visualization. The `{rpart.plot}` library contains the eponymous function `rpart.plot()` that visualizes trees in a top-down fashion. Note: Before applying the `rpart.plot()` function, the `fit_engine` has to be extracted from the model. Strictly speaking, the object `tree_res_basic` contains a workflow object which in turns contains the actual model.

Each node contains the percentage of samples at the specific level in the bottom entry and an estimate in the top entry. For example, after the first split, the left node contains 63% of all samples and the estimate for the alcohol contents is 9.8 percent. The node on the right contains the remaining 37% percent with an estimated alcohol content of 12 percent. The branches of the tree contain splitting conditions.

```
library(rpart.plot)

tree_res_basic %>%
  extract_fit_engine() %>%
  rpart.plot(roundint=FALSE)
```



4.2.4 cp-table

The complexity parameter (cp) table is a useful tool for asessing model performance as well. It contains the complecity parameter itself, and for each value the respective split, relative error, cross-validation error and cross-validation standard deviation.

```
tree_res_basic %>%
  extract_fit_engine() %>%
  printcp()
```

```
Regression tree:
rpart::rpart(formula = ..y ~ ., data = data)

Variables actually used in tree construction:
[1] chlorides      density        residual.sugar

Root node error: 5587.9/3673 = 1.5213

n= 3673

        CP nsplit rel error  xerror      xstd
```

```
1 0.524872      0   1.00000 1.00029 0.0186933
2 0.065803      1   0.47513 0.48862 0.0114048
3 0.063815      2   0.40932 0.41249 0.0100885
4 0.030752      3   0.34551 0.36373 0.0091955
5 0.024820      4   0.31476 0.33361 0.0087150
6 0.022225      5   0.28994 0.31958 0.0081634
7 0.010587      6   0.26771 0.29882 0.0076719
8 0.010347      7   0.25713 0.27522 0.0074280
9 0.010000      8   0.24678 0.27426 0.0074155
```

The `cptable` provides, in addition to the error reduction from a split, further information which we will analyze in the following.

1. The column `CP` (Complexity Parameter) indicates an improvement value for each split. In the first row, this means the improvement value from splitting the root node is calculated.
   The subsequent improvement values also refer to the improvement relative to the error of the root node, which is why the following formula is used:

$$\frac{|K_{\text{parent}}| \cdot \text{MSE}_{K_{\text{parent}}} - (|K_{\text{child}_1}| \cdot \text{MSE}_{K_{\text{child}_1}} + |K_{\text{child}_2}| \cdot \text{MSE}_{K_{\text{child}_2}})}{|K_{\text{root}}| \cdot \text{MSE}_{K_{\text{root}}}} \tag{4.1}$$

   In equation (4.1), the denominator is no longer $|K_{\text{parent}}| \cdot \text{MSE}_{K_{\text{parent}}}$, but instead $|K_{\text{root}}| \cdot \text{MSE}_{K_{\text{root}}}$.
   The value `CP` thus describes the proportion of error improvement a split creates, relative to the total error of the model without any splits (i.e., only the root node).
   So, for example, to calculate the `CP` value in the second row, we can use the tabular-hierarchical representation:

```
n= 3673

node), split, n, deviance, yval
      * denotes terminal node

 1) root 3673 5587.8860 10.522440
   2) density>=0.992695 2320 1499.3140  9.840027
     4) density>=0.995555 1088  359.0238  9.416391
       8) density>=0.99759 531  104.7181  9.180289 *
       9) density< 0.99759 557  196.4870  9.641472 *
     5) density< 0.995555 1232  772.5910 10.214150
      10) residual.sugar< 8.15 1006  490.8750 10.055120
```

```
      20) chlorides>=0.0475 440  177.2503  9.780076 *
      21) chlorides< 0.0475 566  254.4635 10.268930 *
    11) residual.sugar>=8.15 226  143.0244 10.922040 *
  3) density< 0.992695 1353 1155.6450 11.692570
    6) density>=0.990315 985  665.7154 11.378780
     12) residual.sugar< 4.05 730  384.3869 11.131920
       24) density>=0.99147 374  141.8090 10.729500 *
       25) density< 0.99147 356  118.3850 11.554680 *
     13) residual.sugar>=4.05 255  109.4894 12.085480 *
    7) density< 0.990315 368  133.3411 12.532470 *
```

The CP value is then calculated using the deviance terms that result from splitting node 3) into nodes 6) and 7):

$$\frac{1155.6450 - (665.7154 + 133.3411)}{5587.89} = 0.0638$$

The last entry in the column CP (0.01) is the default parameter of the decision_tree function for the argument cp.

The column nsplit describes the number of splits. Since the last entry is 9, this means the feature space was split nine times.

The column rel error describes the relative error of the decision tree at the respective split in relation to the root node. This can be calculated using the formula:

$$\text{rel error}i = \begin{cases} 1, & \text{if } i = 1 \\ \text{rel error}i - 1 - \text{CP}_{i-1}, & \text{if } i > 1 \end{cases}$$

So, for example, to calculate the `rel err` for the last entry ($i = 9$), the formula with numbers inserted is:

$$0.25713 - 0.0103 = 0.246783$$

The columns `xerror` and `xstd` describe the `rel err` and its standard deviation, which result from a cross-validation.

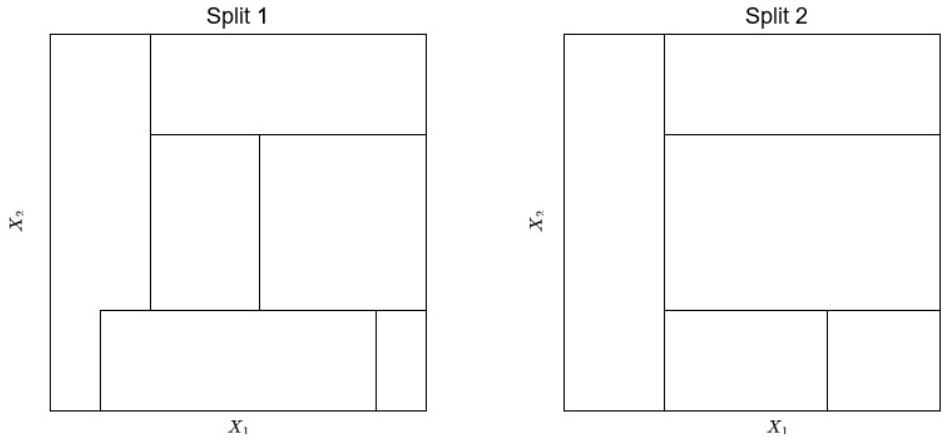
4.3 Exercises

4.3.1 Theoretical exercises

In this first exercise, we want to gain intuition for the theoretical aspects of regression trees.

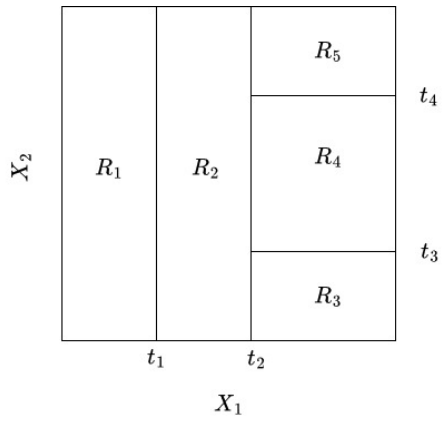
Before diving into the process of building and evaluating a tree rigorously, we first consider different representations of binary trees, check their validity, and decide for simple datasets, whether they are suitable for regression trees.

Exercise 4.1. Consider the following two splits of the feature space generated by two features X_1 and X_2. Argue, which one of the splits was generated by a binary splitting tree!

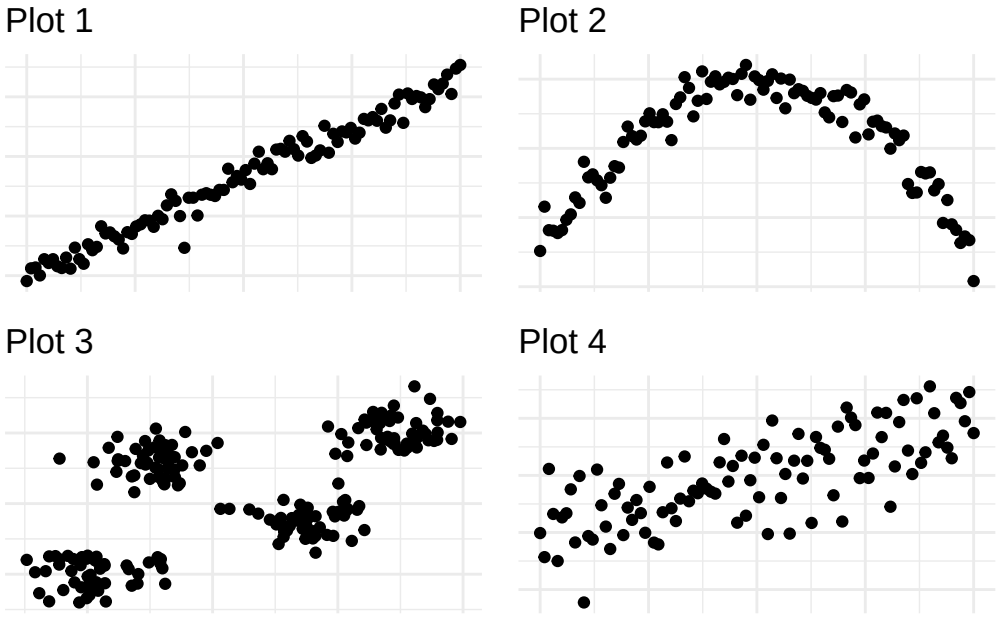


Exercise 4.2. Consider the following split generated by a binary tree. $t_1, ..., t_4$ denote the splitting conditions, $R_1, ..., R_4$ the final regions, and X_1, X_2 the variables used for evaluating the splitting conditions.

Draw a binary tree that corresponds to the split given below.



Exercise 4.3. For the following scatterplots, decide whether a simple linear model ($y = \widehat{\beta}_1 x + \widehat{\beta}_0$) or a regression tree should be chosen for modeling the data.



Exercise 4.4. Now, that we have considered some visual examples of trees and gained an intuition of situations where trees might be a suitable model, we now want to focus on the process of building a tree.

Consider the following dataset. Calculate the first optimal splitting point with respect to x.

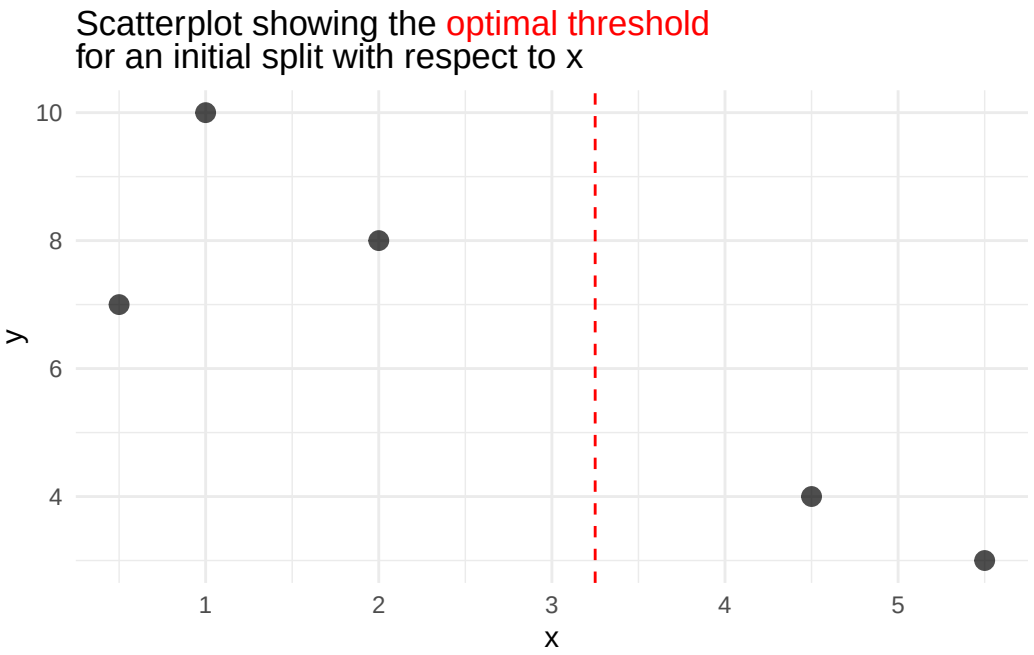
```
data_tmp <- tibble(
  x = c(1,0.5,2.0,5.5,4.5),
  y = c(10,7,8,3,4)
)
data_tmp
```

```
# A tibble: 5 x 2
      x     y
  <dbl> <dbl>
1   1      10
2   0.5     7
3   2       8
4   5.5     3
5   4.5     4
```

In order to do so, you have to proceed as follows:

1. Derive the order statistics of $\{x_1, ..., x_n\}$
2. Derive the set $S := \left\{\frac{1}{2}(x_{(r)} + x_{(r+1)}) : r = 1, ..., n-1\right\}$ of all potential splitting points.
3. For each potential splitting point, derive the regions R_1 and R_2 and calculate the estimate $\hat{y}_1$ and $\hat{y}_2$ for the respective regions.
4. Calculate the loss $\mathcal{L}(y, \hat{y}) := \sum_{i:x_i \in R_1}(y_i - \hat{y})^2 + \sum_{i:x_i \in R_2}(y_i - \hat{y})^2$.
5. Derive the optimal splitting point by settling for the splitting point leading to the smallest loss $\mathcal{L}$.

Exercise 4.5. Given the tibble `data`, create a simple scatter plot and add a dashed line indicating the initial splitting point. An example of what such a plot could look like can be found below.



Exercise 4.6. Calculate the improvement of the given split. Recall, that the improvement of a split is given by

$$\frac{\text{MSE}_1 \cdot n_1 - (\text{MSE}_2 \cdot n_2 + \text{MSE}_3 \cdot n_3)}{\text{MSE}_1 \cdot n_1},$$

where MSE_1 denotes the mean squared error of the region before the split and MSE_2 and MSE_3 are the mean square errors of the respective regions after the split. n_i, $i = 1, 2, 3$ denotes the number of samples in the respective region.

4.3.2 Programming Exercises

In this exercise, we want to apply our theoretical knowledge to training a tree-based model on the Apartment rental offers in Germany dataset. As in Session 03 we will be using the rental offers in Munich to build a predictive model for the base rent.

Exercise 4.7. Import the data set, create a training/testing split and a 5-fold CV object on the training data using `set.seed(24)`.

Exercise 4.8. Explain the syntax and semantics of the following code snippet.

```r
data_muc %>%
  select_if(where(~sum(is.na(.))>0)) %>%
  is.na() %>%
  colSums() %>%
  tibble( names = names(.),
          n = .) %>%
  arrange(desc(n))
```

```
# A tibble: 12 x 2
   names                n
   <chr>            <dbl>
 1 heatingCosts      3565
 2 petsAllowed       2013
 3 interiorQual      1584
 4 firingTypes       1243
 5 condition         1206
 6 heatingType       1108
 7 typeOfFlat         847
 8 yearConstructed    751
 9 floor              631
10 totalRent          440
11 serviceCharge      150
12 pricetrend          22
```

Exercise 4.9. Create a recipe based on the following description:

1. Create a recipe for a regression model with baseRent as the target variable and all other columns as predictors.
2. Update the role of the scoutId column to "ID".
3. Remove the following specified columns from the dataset:

```r
c("serviceCharge","heatingType","picturecount","totalRent",
      "firingTypes","typeOfFlat","noRoomsRange", "petsAllowed",
      "livingSpaceRange","regio3","heatingCosts", "floor",
      "date", "pricetrend")
```

```
 [1] "serviceCharge"    "heatingType"      "picturecount"     "totalRent"
 [5] "firingTypes"      "typeOfFlat"       "noRoomsRange"     "petsAllowed"
 [9] "livingSpaceRange" "regio3"           "heatingCosts"     "floor"
[13] "date"             "pricetrend"
```

4. Convert `interiorQual` and `condition` into ordered factors with specified levels and `geo_plz` into an unordered factor.
5. Create a specification that assigns a previously unseen factor level to `new` using the `step_novel()` function.
6. Convert `geo_plz` into dummy variables.
7. Create ordinal scores for every ordered predictor.
8. Impute missing values for all ordered predictors using k-nearest neighbors.
9. Filter rows in the dataset to retain only observations where `baseRent` is at most 4000 EUR and `livingSpace` is at most 200 sqm.

Exercise 4.10. Create an instance of the `decision_tree` class where the parameters `min_n`,`tree_depth`, and `cost_complexity` are set to tune.

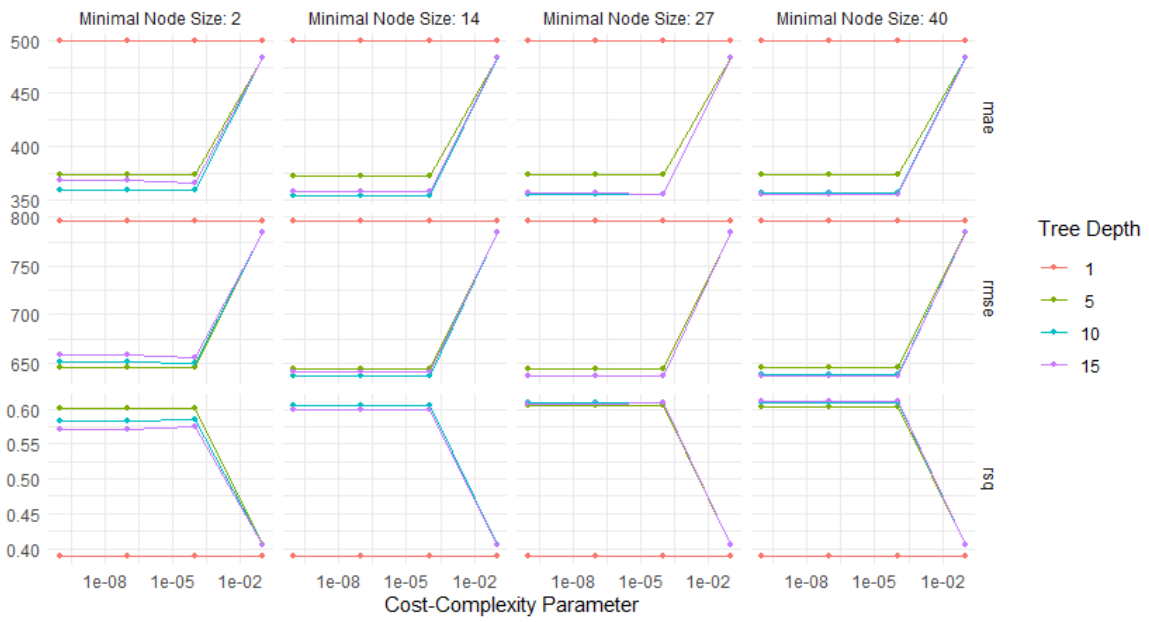
Exercise 4.11. Create a workflow and add the previously specified model and recipe.

Exercise 4.12. Instead of specifying the grid manually, use the `extract_parameter_set_dials` function to create a regular grid with 4 levels.

Exercise 4.13. Tune the model on the cross-validation set created in Exercise 4.7. As for the Use the following metric set for evaluating the tuning results.

```
multi_metric <- metric_set(rmse,rsq)
```

Exercise 4.14. Given the following plot. What can you say about the relationship between the tree parameters `Tree Depth`, `Minimal Node Size`, and `Cost-Complexity Parameter` with respect to the RMSE?



Exercise 4.15. Select the best model with respect to the metric MAE and fit a final model using these parameters. Then, fit the best model on the whole training data and evaluate it on the test data using the previously defined metrics set.

Exercise 4.16. It is usually easier to get a feeling for model performance by visualizing the results. One way to do that would be to plot the predicted values of our model against the true values. By adding a simple line through the origin with slope one, we can then evaluate the estimates as follows:

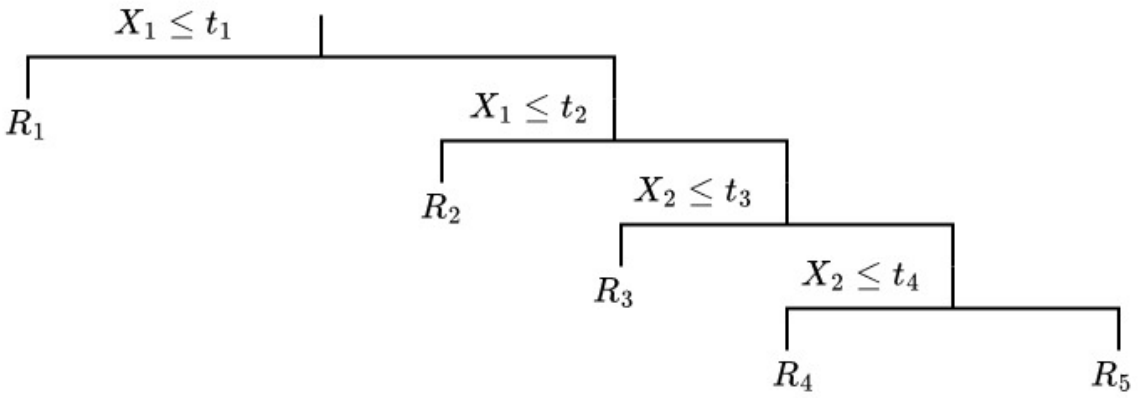
Points that are closely scattered around this line are well predicted, whereas points further away from this line indicate that the model performed badly.

The goal of this exercise is for you to rebuild the plot that is depicted below.

4.4 Solutions

Solution 4.1 (Exercise 4.1). Split 1 can't be produced by a binary tree, because the bottom-center rectangle is overlapping the left-most rectangle.

Solution 4.2 (Exercise 4.2).



Solution 4.3 (Exercise 4.3).

1. For the data in the first plot, we should use a simple linear model, as the data seems to follow a simple linear trend.
2. A linear model is most likely not suitable for modelling this data, as the shape of the cloud of points looks more like a parabola instead of a line.
3. As the third plot consists of points that can be assigned to four (almost) distinct regions, a regression tree seems to be more suitable than a linear model.
4. At first, the data in the fourth plot seems to be too messy to make a decision. However, upon closer inspection, there are several indicators that a linear model might perform better:

    1. The points in the center seem to follow a positive linear trend.
    2. The deviations of points around this linear trend seem to be distributed in a way, that there are more points towards the line than further away. So the residuals could be assumed to be normally distributed.

Solution 4.4 (Exercise 4.4).

```
loss_x<- function(data,r) {
  xr = sort(data$x)[r]
  y1 <- mean(data$y[data$x<=xr])
  y2 <- mean(data$y[data$x>xr])

  loss<-sum((data$y[data$x<=xr]-y1)^2) + sum((data$y[data$x>xr]-y2)^2)

  return(loss)
}
```

Since we are interested in finding the optimal split with respect to $x_1 = x$, consider the sets of all possible splits

$$S := \left\{ \frac{1}{2}(x_{(r)} + x_{(r+1)}) : r = 1, ..., n-1 \right\} = \{0.75, 1.5, 3.25, 5\}.$$

Here, $\{x_{(r)}, r = 1, ..., n\} = \{0.5, 1, 2, 4.5, 5.5\}$ denotes the order statistic of x.

For $r = 1$ we have $s = 0.75$ and

$$\begin{aligned} R_1(1, 0.75) &= \{x : x \leq 0.75\} = \{0.5\}, \\ R_2(1, 0.75) &= \{x : x > 0.75\} = \{1.0, 2.0, 5.5, 4.5\}. \end{aligned}$$

Then,

$$\begin{aligned} \hat{y}_1 &= \frac{1}{|R_1|} \sum_{i: x_i \in R_1} y_i = \frac{1}{1} \cdot 7 = 7, \\ \hat{y}_2 &= \frac{1}{|R_2|} \sum_{i: x_i \in R_2} y_i = \frac{1}{4}(10 + 8 + 3 + 4) = 6.25. \end{aligned}$$

Given the above, we can calculate the Loss with respect to $s = 0.75$, which is given by

$$\begin{aligned} \mathcal{L}(y, \hat{y}) &= \sum_{i: x_i \in R_1} (y_i - \hat{y}_{R_1})^2 + \sum_{i: x_i \in R_2} (y_i - \hat{y}_{R_2})^2 \\ &= (7-7)^2 + (10-6.25)^2 + (8-6.25)^2 + (3-6.25)^2 + (4-6.25)^2 \\ &= 32.75 \end{aligned}$$

```
loss_x(data_tmp,1)
```

```
[1] 32.75
```

For $r = 2$ we have $s = 1.5$ and

$$\begin{aligned} R_1(1, 1.5) &= \{x : x \leq 1.5\} = \{0.5, 1.0\}, \\ R_2(1, 1.5) &= \{x : x > 1.5\} = \{2.5, 5.5, 4.5\}. \end{aligned}$$

Then,

$$\hat{y}_1 = \frac{1}{|R_1|}\sum_{i:x_i\in R_1} y_i = \frac{1}{2}\cdot(7+10) = 8.5,$$
$$\hat{y}_2 = \frac{1}{|R_2|}\sum_{i:x_i\in R_2} y_i = \frac{1}{3}(8+3+4) = 5.$$

Given the above, we can calculate the Loss with respect to $s = 1.5$, which is given by

$$\begin{aligned}\mathcal{L}(y,\hat{y}) &= \sum_{i:x_i\in R_1}(y_i-\hat{y}_{R_1})^2 + \sum_{i:x_i\in R_2}(y_i-\hat{y}_{R_2})^2\\ &= (7-8.5)^2+(10-8.5)^2+(8-5)^2+(3-5)^2+(4-5)^2\\ &= 18.5\end{aligned}$$

```
loss_x(data_tmp,2)
```

```
[1] 18.5
```

For $r = 3$ we have $s = 3.25$ and

$$R_1(1, 3.25) = \{x : x \leq 3.25\} = \{0.5, 1.0, 2.5\},$$
$$R_2(1, 3.25) = \{x : x > 3.25\} = \{5.5, 4.5\}.$$

Then,

$$\hat{y}_1 = \frac{1}{|R_1|}\sum_{i:x_i\in R_1} y_i = \frac{1}{3}\cdot(7+10+8) = 8.333,$$
$$\hat{y}_2 = \frac{1}{|R_2|}\sum_{i:x_i\in R_2} y_i = \frac{1}{2}(3+4) = 3.5.$$

Given the above, we can calculate the Loss with respect to $s = 4$, which is given by

$$\begin{aligned}\mathcal{L}(y,\hat{y}) &= \sum_{i:x_i\in R_1}(y_i-\hat{y}_{R_1})^2 + \sum_{i:x_i\in R_2}(y_i-\hat{y}_{R_2})^2\\ &= (7-8.333)^2+(10-8.333)^2+(8-8.333)^2+(3-3.5)^2+(4-3.5)^2\\ &= 5.167\end{aligned}$$

```
loss_x(data_tmp,3)
```

```
[1] 5.166667
```

For $r = 4$ we have $s = 5$ and

$$R_1(1,5) = \{x : x \leq 5\} = \{0.5, 1.0, 2.5, 4.5\},$$
$$R_2(1,5) = \{x : x > 5\} = \{5.5\}.$$

Then,

$$\hat{y}_1 = \frac{1}{|R_1|} \sum_{i:x_i \in R_1} y_i = \frac{1}{4} \cdot (7 + 10 + 8 + 4) = 7.25,$$
$$\hat{y}_2 = \frac{1}{|R_2|} \sum_{i:x_i \in R_2} y_i = \frac{1}{1} \cdot 3 = 3.$$

Given the above, we can calculate the Loss with respect to $s = 5$, which is given by

$$\begin{aligned}\mathcal{L}(y, \hat{y}) &= \sum_{i:x_i \in R_1} (y_i - \hat{y}_{R_1})^2 + \sum_{i:x_i \in R_2} (y_i - \hat{y}_{R_2})^2 \\ &= (7 - 7.25)^2 + (10 - 7.25)^2 + (8 - 7.25)^2 + (4 - 7.25)^2 + (3 - 3)^2 \\ &= 18.75\end{aligned}$$

```
loss_x(data_tmp,4)
```

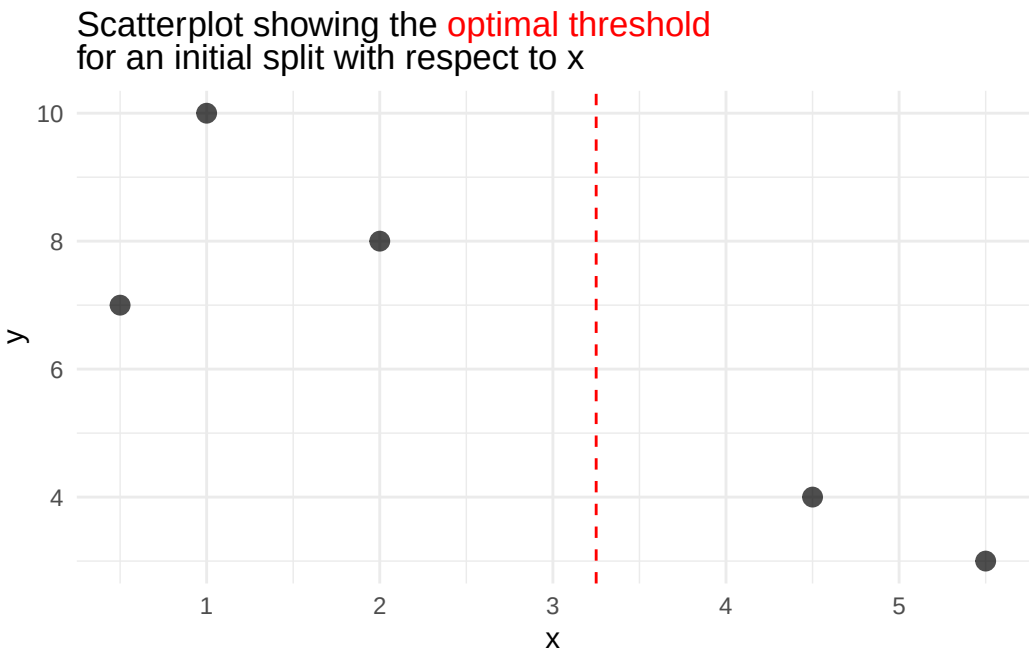
```
[1] 18.75
```

Since $\mathcal{L}(y, \hat{y})$ is the lowest for $r = 3$, i.e., $\mathcal{L}(y, \hat{y}) = 5.167$, $s = 3.25$ is the optimal splitting point with respect to x.

Solution 4.5 (Exercise 4.5).

```
title_text = "Scatterplot showing the
              <span style='color:red'>optimal threshold</span> <br/>
              for an initial split with respect to x"

data_tmp %>% ggplot(aes(x,y))+
  geom_point(size = 3, alpha = 0.7) +
  geom_vline(xintercept = 3.25, linetype = "dashed", color = "red") +
```

```
  theme_minimal(base_size=11)+
  theme(
    plot.title = element_markdown()
  )+
  labs( x = "x",
        title = title_text)
```



Solution 4.6 (Exercise 4.6). The improvement is given by the following term.

$$\frac{\text{MSE}_1 \cdot n_1 - (\text{MSE}_2 \cdot n_2 + \text{MSE}_3 \cdot n_3)}{\text{MSE}_1 \cdot n_1}$$

Calculating MSE_i for $i = 1, 2, 3$ yields

$$\begin{aligned}
n_1 \cdot \text{MSE}_1 &= (10 - 6.4)^2 + (7 - 6.4)^2 + (8 - 6.4)^2 + (3 - 6.4)^2 + (4 - 6.4)^2 = 33.2, \\
n_2 \cdot \text{MSE}_2 &= (7 - 8.333)^2 + (10 - 8.333)^2 + (8 - 8.333)^2 = 4.667, \\
n_3 \cdot \text{MSE}_3 &= (3 - 3.5)^2 + (4 - 3.5)^2 = 0.5 \ .
\end{aligned}$$

The improvement for this split is therefore

$$\frac{\text{MSE}_1 \cdot n_1 - (\text{MSE}_2 \cdot n_2 + \text{MSE}_3 \cdot n_3)}{\text{MSE}_1 \cdot n_1} = \frac{33.2 - (4.667 + 0.5)}{33.2} = 0.8444$$

Solution 4.7 (Exercise 4.7).

```
data_muc <- read.csv("data/rent_muc.csv")
```

```
set.seed(24)
split_rent <- initial_split(data_muc)
data_train <- training(split_rent)
data_test <- testing(split_rent)
folds <- vfold_cv(data_train, v = 5)
```

Solution 4.8 (Exercise 4.8).

```
data_muc %>%
  select_if(where(~sum(is.na(.))>0)) %>%
  is.na() %>%
  colSums() %>%
  tibble( names = names(.),
          n = .) %>%
  arrange(desc(n))
```

1. The `%>%` operator passes the data set `data_muc` to the next function.
2. `select_if` selects columns in a data frame based on the condition that `where(~ sum(is.na(.)) > 0)` which checks if the sum of `NA` values in a column is greater than 0.
3. The `is.na()` function checks whether an entry in the data set is `NA` or not and returns `True` or `FALSE` respectively. Therefore, by applying the `is.na()` function, a data set containing only boolean values is returned.
4. The `colSums` function adds up all values in each column, where `TRUE = 1` and `FALSE = 0`, returning a named vector containing the sum of all `TRUE` values and the respective variable names.
5. Then, the data set is transformed using the `tibble()` function such that the returned tibble only consists of two columns containing the variable name and column sum.
6. In the last step, the data set is ordered with respect to the number of missing values in descending order using the `arrange()` and `desc()` function.

Solution 4.9 (Exercise 5.6).

```
rec_rent <- recipe(
    formula = baseRent ~.,
    data = data_train
  ) %>%
  update_role(scoutId, new_role = "ID") %>%
  step_select(!c("serviceCharge","heatingType","picturecount","totalRent",
            "firingTypes","typeOfFlat","noRoomsRange", "petsAllowed",
            "livingSpaceRange","regio3","heatingCosts", "floor",
            "date", "pricetrend")) %>%
  step_mutate(
    interiorQual = factor(
      interiorQual,
      levels = c("simple", "normal", "sophisticated", "luxury"),
      ordered = TRUE
    ),
    condition = factor(
      condition,
      levels = c("need_of_renovation", "negotiable","well_kept",
                 "refurbished","first_time_use_after_refurbishment",
                 "modernized","fully_renovated", "mint_condition",
                 "first_time_use"),
      ordered = TRUE
    ),
    geo_plz = factor(geo_plz)
  )%>%
  step_novel(geo_plz)%>%
  step_dummy(geo_plz)%>%
  step_ordinalscore(all_ordered_predictors())%>%
  step_impute_knn(all_predictors()) %>%
  step_filter(baseRent <= 4000,
              livingSpace <= 200)
```

Solution 4.10 (Exercise 4.10).

```
model_rt_tune <-
  decision_tree(
    min_n = tune(),
    tree_depth = tune(),
    cost_complexity = tune()
    ) %>%
```

```
  set_mode("regression") %>%
  set_engine("rpart")
```

Solution 4.11 (Exercise 4.11).

```
wf_rent <-
  workflow() %>%
  add_recipe(rec_rent) %>%
  add_model(model_rt_tune)
```

Solution 4.12 (Exercise 4.12).

```
tree_grid <- wf_rent %>%
  extract_parameter_set_dials %>%
  grid_regular(levels = 4)
```

Solution 4.13 (Exercise 4.13).

```
multi_metric <- metric_set(rmse,rsq,mae)
```

```
rt_res <-
  wf_rent %>%
  tune_grid(
    grid = tree_grid,
    metrics = multi_metric,
    resamples = folds
  )
```

```
> A | warning: skipping variable with zero or non-finite range.

There were issues with some computations   A: x1

There were issues with some computations   A: x2

There were issues with some computations   A: x4

There were issues with some computations   A: x6
```

```
There were issues with some computations   A: x8

There were issues with some computations   A: x10

There were issues with some computations   A: x12

There were issues with some computations   A: x14

There were issues with some computations   A: x16

There were issues with some computations   A: x18

There were issues with some computations   A: x19

There were issues with some computations   A: x22

There were issues with some computations   A: x23

There were issues with some computations   A: x25

There were issues with some computations   A: x27

There were issues with some computations   A: x29

There were issues with some computations   A: x30

There were issues with some computations   A: x32

There were issues with some computations   A: x34

There were issues with some computations   A: x36

There were issues with some computations   A: x38

There were issues with some computations   A: x40

There were issues with some computations   A: x42
```

```
There were issues with some computations   A: x43

There were issues with some computations   A: x45

There were issues with some computations   A: x47

There were issues with some computations   A: x49

There were issues with some computations   A: x51

There were issues with some computations   A: x53

There were issues with some computations   A: x55

There were issues with some computations   A: x56

There were issues with some computations   A: x58

There were issues with some computations   A: x60

There were issues with some computations   A: x62

There were issues with some computations   A: x64

There were issues with some computations   A: x66

There were issues with some computations   A: x68

There were issues with some computations   A: x71

There were issues with some computations   A: x73

There were issues with some computations   A: x75

There were issues with some computations   A: x77

There were issues with some computations   A: x79
```

```
There were issues with some computations   A: x81

There were issues with some computations   A: x84

There were issues with some computations   A: x86

There were issues with some computations   A: x88

There were issues with some computations   A: x90

There were issues with some computations   A: x92

There were issues with some computations   A: x94

There were issues with some computations   A: x96

There were issues with some computations   A: x98

There were issues with some computations   A: x100

There were issues with some computations   A: x102

There were issues with some computations   A: x104

There were issues with some computations   A: x106

There were issues with some computations   A: x108

There were issues with some computations   A: x110

There were issues with some computations   A: x112

There were issues with some computations   A: x114

There were issues with some computations   A: x116

There were issues with some computations   A: x118
```

```
There were issues with some computations   A: x120

There were issues with some computations   A: x122

There were issues with some computations   A: x124

There were issues with some computations   A: x126

There were issues with some computations   A: x128

There were issues with some computations   A: x130

There were issues with some computations   A: x131

There were issues with some computations   A: x132

There were issues with some computations   A: x134

There were issues with some computations   A: x136

There were issues with some computations   A: x137

There were issues with some computations   A: x139

There were issues with some computations   A: x141

There were issues with some computations   A: x142

There were issues with some computations   A: x144

There were issues with some computations   A: x145

There were issues with some computations   A: x147

There were issues with some computations   A: x149

There were issues with some computations   A: x150
```

```
There were issues with some computations   A: x152

There were issues with some computations   A: x154

There were issues with some computations   A: x155

There were issues with some computations   A: x157

There were issues with some computations   A: x159

There were issues with some computations   A: x160

There were issues with some computations   A: x162

There were issues with some computations   A: x163

There were issues with some computations   A: x165

There were issues with some computations   A: x166

There were issues with some computations   A: x168

There were issues with some computations   A: x169

There were issues with some computations   A: x170

There were issues with some computations   A: x172

There were issues with some computations   A: x174

There were issues with some computations   A: x176

There were issues with some computations   A: x177

There were issues with some computations   A: x178

There were issues with some computations   A: x179
```

```
There were issues with some computations   A: x180

There were issues with some computations   A: x181

There were issues with some computations   A: x182

There were issues with some computations   A: x183

There were issues with some computations   A: x184

There were issues with some computations   A: x185

There were issues with some computations   A: x186

There were issues with some computations   A: x187

There were issues with some computations   A: x188

There were issues with some computations   A: x189

There were issues with some computations   A: x190

There were issues with some computations   A: x191

There were issues with some computations   A: x192

There were issues with some computations   A: x193

There were issues with some computations   A: x194

There were issues with some computations   A: x195

There were issues with some computations   A: x196

There were issues with some computations   A: x198

There were issues with some computations   A: x199
```

```
There were issues with some computations   A: x201

There were issues with some computations   A: x202

There were issues with some computations   A: x204

There were issues with some computations   A: x206

There were issues with some computations   A: x207

There were issues with some computations   A: x209

There were issues with some computations   A: x211

There were issues with some computations   A: x212

There were issues with some computations   A: x214

There were issues with some computations   A: x216

There were issues with some computations   A: x218

There were issues with some computations   A: x220

There were issues with some computations   A: x222

There were issues with some computations   A: x223

There were issues with some computations   A: x225

There were issues with some computations   A: x227

There were issues with some computations   A: x229

There were issues with some computations   A: x231

There were issues with some computations   A: x233
```

```
There were issues with some computations   A: x235

There were issues with some computations   A: x237

There were issues with some computations   A: x239

There were issues with some computations   A: x241

There were issues with some computations   A: x243

There were issues with some computations   A: x245

There were issues with some computations   A: x247

There were issues with some computations   A: x249

There were issues with some computations   A: x251

There were issues with some computations   A: x253

There were issues with some computations   A: x255

There were issues with some computations   A: x257

There were issues with some computations   A: x259

There were issues with some computations   A: x261

There were issues with some computations   A: x263

There were issues with some computations   A: x265

There were issues with some computations   A: x267

There were issues with some computations   A: x269

There were issues with some computations   A: x271
```

```
There were issues with some computations   A: x273

There were issues with some computations   A: x274

There were issues with some computations   A: x276

There were issues with some computations   A: x278

There were issues with some computations   A: x280

There were issues with some computations   A: x282

There were issues with some computations   A: x284

There were issues with some computations   A: x286

There were issues with some computations   A: x287

There were issues with some computations   A: x289

There were issues with some computations   A: x291

There were issues with some computations   A: x293

There were issues with some computations   A: x295

There were issues with some computations   A: x297

There were issues with some computations   A: x299

There were issues with some computations   A: x301

There were issues with some computations   A: x302

There were issues with some computations   A: x304

There were issues with some computations   A: x306
```

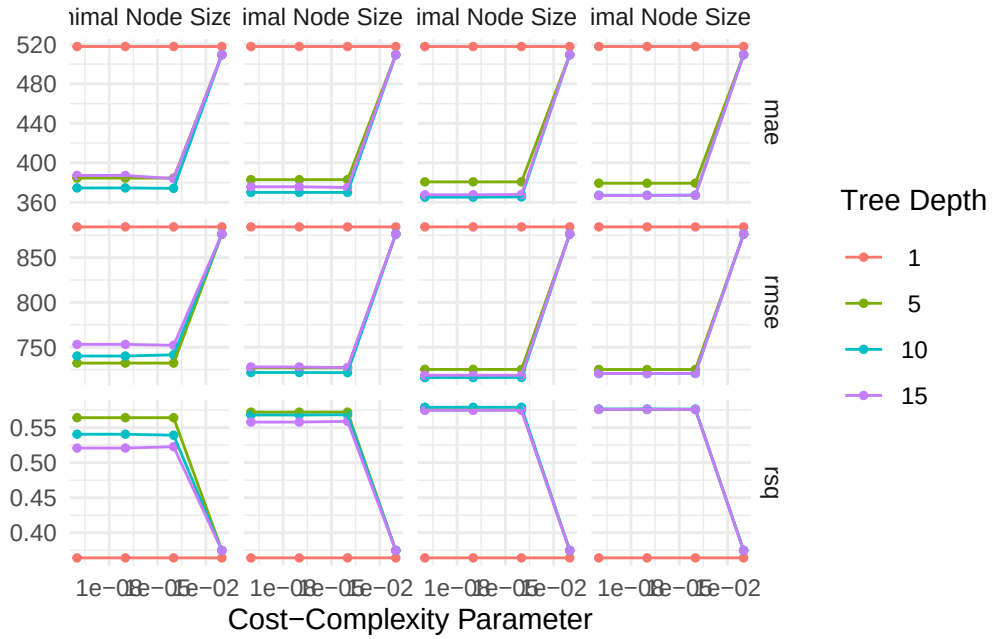
```
There were issues with some computations   A: x307

There were issues with some computations   A: x308

There were issues with some computations   A: x310

There were issues with some computations   A: x311

There were issues with some computations   A: x313

There were issues with some computations   A: x315

There were issues with some computations   A: x317

There were issues with some computations   A: x319

There were issues with some computations   A: x321

There were issues with some computations   A: x323

There were issues with some computations   A: x325
There were issues with some computations   A: x325
```

To speed things up, the following tibble already contains the optimal parameters according to the tuning procedure:

```
rt_res_tune_best <- tibble(
  min_n = 14,
  tree_depth = 10,
  cost_complexity = 0.0001
)
```

Solution 4.14 (Exercise 4.14).

```
autoplot(rt_res) + theme_minimal()
```



1. **Tree Depth**:
   - Increasing tree depth (lines for depths 1, 5, 10, 15) generally reduces the RMSE across all plots, indicating better model performance as depth grows.
   - However, for deeper trees, RMSE increases when the **Cost-Complexity Parameter** is high, due to over-regularization.
2. **Minimal Node Size**:
   - Smaller minimal node sizes (e.g., 2) allow the tree to split more finely, resulting in lower RMSE when the **Cost-Complexity Parameter** is low.
   - Larger minimal node sizes (e.g., 40) limit splitting, which leads to higher RMSE across all complexity values.
3. **Cost-Complexity Parameter**:
   - A small **Cost-Complexity Parameter** (e.g., (10^{-8})) corresponds to minimal pruning, leading to low RMSE (better performance).
   - As the **Cost-Complexity Parameter** increases, the tree gets pruned more aggressively, and RMSE rises, especially for shallow trees or larger node sizes.

Solution 4.15 (Exercise 4.15).

```
rt_res_best<- rt_res %>%
  select_best(metric = "mae")

last_rt_fit <- wf_rent %>%
  finalize_workflow(rt_res_best) %>%
  last_fit(split_rent)

last_rt_fit %>% collect_metrics()
```

```
# A tibble: 2 x 4
  .metric .estimator .estimate .config
  <chr>   <chr>          <dbl> <chr>
1 rmse    standard     618.    Preprocessor1_Model1
2 rsq     standard       0.626 Preprocessor1_Model1
```

Using the parameters from the pre-specified tibble to avoid tuning time:

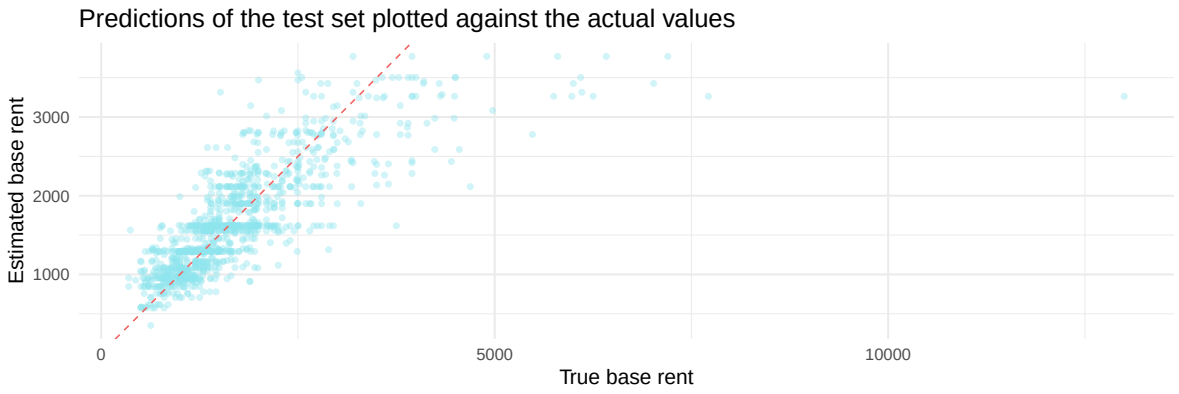
```
last_rt_fit <- wf_rent %>%
  finalize_workflow(rt_res_tune_best) %>%
  last_fit(split_rent)

last_rt_fit %>% collect_metrics()
```

```
# A tibble: 2 x 4
  .metric .estimator .estimate .config
  <chr>   <chr>          <dbl> <chr>
1 rmse    standard     622.    Preprocessor1_Model1
2 rsq     standard       0.616 Preprocessor1_Model1
```

Solution 4.16 (Exercise 4.16).

```
title_text <- "Predictions of the test set plotted against the actual values"
last_rt_fit %>%
  collect_predictions() %>%
  ggplot(aes(baseRent, .pred)) +
  geom_point(alpha = 0.4, color = "cadetblue2")+
  geom_abline(slope = 1, lty = 2, color = "indianred2", alpha = 1) +
  labs(
    x = "True base rent",
    y = "Estimated base rent",
```

```
  title = title_text
)+
theme_minimal(base_size=14)+
coord_fixed()
```



5 Random Forests

5.1 Introduction

In this exercise session we will briefly talk about some theoretical considerations when applying bootstrap sampling, bagging and training a random forest before performing a more in-depth case study of binary classification. The case-study assumes that you are familiar with *penalized logistic regressions* (similar to lasso regression, see Session 03) and *classification trees* (similar to regression trees introduced in Session 4). We will however, revisit different evaluation metrics for a binary classifier before training any of the models which should help you to develop a feeling for model performance.

Note, that some of the models we train might take a few (up to many) minutes depending on your hardware. One way to circumvent long training processes is to use a simple training/validation/test split instead of CV.

5.1.1 Evaluation of binary classifiers

Before we start with the actual exercise, let us quickly review some important metrics and concepts for evaluating binary classifiers.

5.1.1.1 Confusion Matrix

		Predicted condition	
	Total population = P + N	Positive (PP)	Negative (PN)
Actual condition	Positive (P)	True positive (TP), hit	False negative (FN), type II error, miss, underestimation
	Negative (N)	False positive (FP), type I error, false alarm, overestimation	True negative (TN), correct rejection

Figure 5.1: Confusion Matrix, Source: Wikipedia

5.1.1.2 ROC Curve and Precision-Recall Curve

When passing a sample into the classification model, the return-value is usually a probability $p \in [0, 1]$ that denotes the probability of the sample belonging to the Positives (in this hypothetical setting we assume that there are two classes "Positives" and "Negatives"). Intuitively it makes sense to say, that a given sample x belongs to the Positives if $p \geq q = 0.5$. However, this threshold $q = 0.5$ can be adjusted. Depending on this threshold q , the values in our confusion matrix change.

Example:

Set $q = 0$, then p is always larger or equal to q, which means that we assign every value to the positives. Then, our True Positive Rate ($\text{TPR} = \frac{\text{TP}}{\text{P}}$) will be equal to 1 since all samples are assigned to the Positives. However, the True Negative Rate ($\text{TNR} = \frac{\text{TN}}{\text{N}}$) will be equal to 0, since not one sample has been assigned to the Negatives, meaning that $\text{TN} = 0$.

A way to visualize the change in our confusion matrix depending on the threshold q is given by the so-called ROC (Receiver-Operator Curve) curve and Precision-Recall Curve.

ROC Curve:

The ROC curve shows the TPR (also known as *recall* or *sensitivity*) plotted against the TNR (also known as *1-specificity*). By plotting these two values against each other, we can identify a good model by checking whether the curve generated by all the thresholds is approaching

the left top corner of a plot, indicating that both TPR and TNR are equal to 1, i.e. the model perfectly classifies all True Positives and all True Negatives. An exemplary plot can be found below.

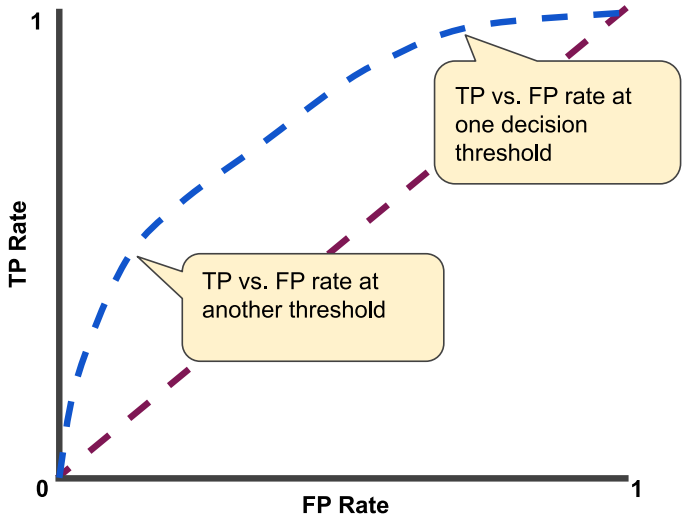


Figure 5.2: ROC Curve, Source: developers.google.com

PR Curve:

The Precision-Recall Curve on the other hand shows the Precision (Precision $= \frac{\text{TP}}{\text{TP+FP}}$) plotted against the Recall (Recall $= \frac{\text{TP}}{\text{TP+FN}}$). Precision basically measures the proportion of correctly identified Positives, while Recall measures the proportion of correctly identified actual Positives. If the line generated by a model is close to the top right corner, the model is relevant, has a high precision, and sensitivity.

An exemplary plot can be found below.

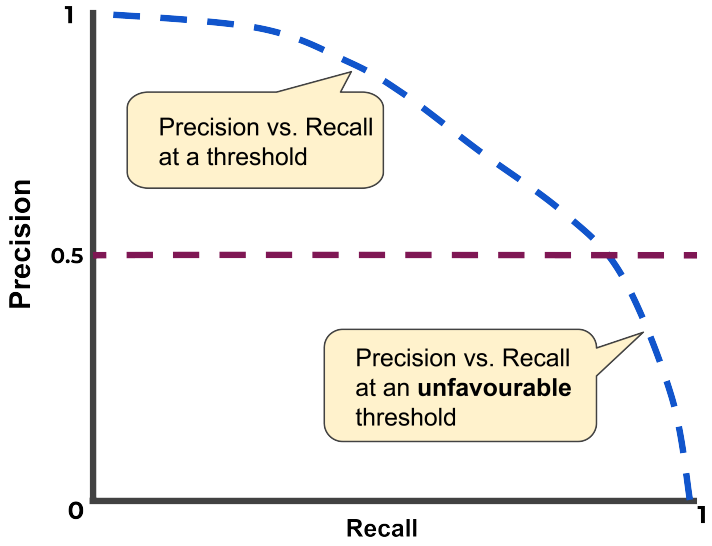


Figure 5.3: PR Curve

What does a high precision and recall actually mean (I find the definition above kind of hard to grasp)? Let us consider a simple example:

Example:

Imagine you are developing a classification model for detecting a rare disease (Positves corresponds to detecting the decease). There are 10000 people in the observed data and only 10 of them are infected with said disease. One way to set up a model (regardless of the features) can be to simply label everyone as not having the disease which would effectively result in an accuracy of 99.9%, *Hurray* . However, this model is obviously not the best since we failed to identify any of the sick people correctly. So, after tweaking the model we now have a model that might be less accurate but identifies sick patients better. What does *better* in this context mean? By looking at the definition of Precision and Recall above, you may notice that they only differ by the second summand in the denominator of the fraction, namely *False Positives* and *False Negatives*. Before we continue, think for yourself, which of those are worse in the scenario of detecting a disease? The right answer would be *False Negatives*, since we fail to identify a sick patient! That is why a sensitive model is crucial here as well

In summary: especially for imbalanced data sets, we do not only want to achieve a high accuracy since that is fairly easy to achieve. We Also want a high precision or recall, focusing on one or the other depending on the model objective.

5.2 Random forests

Throughout this exercise we will use the following libraries:

```
library(tidyverse)
library(tidymodels)

#variable importance plots

library("vip")

#Stitching together plots and adding markdown text

library("patchwork")
library("ggtext")
```

For this sessions example, we will also use the white wine data set, but extend it by adding the red wine data set. The red wine data set can be downloaded directly from the UC Irvine Machine Learning Repository or by clicking the button below.

Download Red Wine Data

A detailed description of each parameter see Cortez et al.

Note, that importing the red wine data set with the `read.csv` function requires the argument `sep = ";"` which indicates that
the columns in the csv are separated by a semicolon.

After importing the data, we add a new column names `wine_color` that indicates the color of the wine. The wine color will be the target variable, meaning that we try to determine the color of a wine given all the other attributes.

To combine both data sets, we can use the `rbind()` function which binds together the rows of a data set. Before binding together the rows of the data set, we need to make sure that the names of the columns coincide. Otherwise, the columns can't be matched.

```
data_wine_red<-read.csv("data/winequality-red.csv", sep = ";")
data_wine_white<- read.csv("data/winequality-white.csv")

data_wine_red<- data_wine_red %>% mutate(wine_color = "red")
data_wine_white<- data_wine_white %>% mutate(wine_color = "white")

data_wine <- rbind(data_wine_red,data_wine_white)
```

The newly created data set contains approximately 5000 wine samples with around 25% being red wine and the remaining 75% being white wine.

```r
data_wine %>% group_by(wine_color) %>%
  summarise(n = n()) %>%
  mutate(ratio = n/sum(n))
```

```
# A tibble: 2 x 3
  wine_color     n ratio
  <chr>      <int> <dbl>
1 red         1599 0.246
2 white       4898 0.754
```

Since our data set is imbalanced we should apply stratification in our data split. Stratification ensures that the same ration of red and white wine samples is in the training and testing data. After splitting our data, we can create a 5-fold CV object on the training data.

```r
set.seed(123)
split_wine <- initial_split(data_wine, strata = wine_color)

data_wine_train <- training(split_wine)
data_wine_test <- testing(split_wine)

folds_wine <- vfold_cv(data_wine_train,5)
```

Then, we can set up a receipe containing a simple formula and step to convert the target feature `wine_color` to type `factor`.

```r
rec_wine <- recipe(
  wine_color ~.,
  data = data_wine_train
  ) %>%
  step_string2factor(wine_color)
```

A random forest model can be specified using the `rand_forest()` function. Additional arguments include, but are not limited to:

- `mode`: indicates whether a classifier or a regression model is specified. (required)
- `trees`: indicates the number of trees fitted in the forest. (default = `500`)
- `min_n`: indicates the minimum number of data points in a node that is required for the node to be split further. (default = `20`)
- `mtry`: indicates the number of variables to possibly split at in each node. (default = `sqrt(ncol(data)-1)`))

Note, that the `mtry` parameter depends on the number of independent variables. If `mtry = ncol(data)-1`, meaning that we select every single independent variable for a potential split, we are creating a bag, rather than a random forest.

By setting every hyper parameter to `tune()`, we specify that the respective hyper parameters are to be tuned.

```
rf_mod_tune_spec <- rand_forest(
  mode = "classification",
  trees = tune(),
  min_n = tune(),
  mtry = tune()
) %>%
  set_engine("ranger", importance = "permutation")
```

As with any other model, we can create a workflow, add the recipe and model specification, and create a metrics set. The metric set below contains the following metrics:

- `roc_auc`: measures the area under the receiver operator characteristic (values $\in [0, 1]$, with 1 being the best possible value).
- `pr_auc`: measures the area under the precision-recall curve (values $\in [0, 1]$, with 1 being the best possible value).
- `precision`: measures the positive predictive value (values $\in [0, 1]$, with 1 being the best possible value).
- `recall`: measures the true positive rate (values $\in [0, 1]$, with 1 being the best possible value).

```
wf_wine <- workflow() %>%
  add_recipe(rec_wine) %>%
  add_model(rf_mod_tune_spec)

multi_metrics <- metric_set(roc_auc,pr_auc,precision,recall)
```

The random forest model can be tuned on the 5-fold CV object in the same fashion as every other model. By specifying `grid=10`, we circumvent specifying the range for the `mtry()` parameter.

> **Warning**
>
> Tuning a random forest can take a while. Instead of using 5-fold CV, a simple training/validation/test split can decrease training time.

```
rf_tune_res <- wf_wine %>%
  tune_grid(
    resamples = folds_wine,
    metrics = multi_metrics,
    grid = 10
  )
```

```
i Creating pre-processing data to finalize unknown parameter: mtry
```

After tuning the model, we can select the best set of hyper parameters with respect to different metrics. If we aim for a model that emphasizes correctly classifying the minority class, the metric `pr_auc` metric can be more useful (why?). We, therefore, select the best parameters according to the metric `pr_auc` and train a final model using these parameters.

```
best_parm_rf_wine <- rf_tune_res %>%
  select_best(metric = "pr_auc")

last_rf_fit <- wf_wine %>%
  finalize_workflow(best_parm_rf_wine) %>%
  last_fit(split_wine,
           metrics= multi_metrics)
```

To evaluate our model, we can either collect the specified metrics using the `collect_metrics()` function, or generate PR- and ROC-curves.

The latter can be achieved with the following Code snippet. We first collect the predictions of the test data using the `collect_predictions()` function. Then, we generate a ROC- and PR-Curve using the functions `roc_curve()` and `pr_curve()`. The `roc_curve()` function returns a data frame containing three columns:

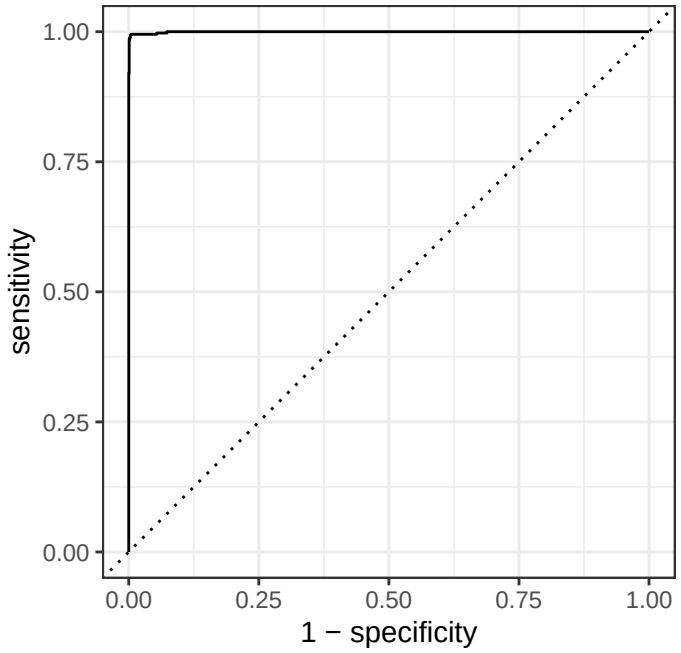
1. `.threshold`: containing the threshold probability for which a sample is assigned to the positive class (in that case `red`).
2. `specificity`: containing the specificity of the model for the given thresholds.
3. `sensitivity`: containing the sensitivity of the model for the given thresholds.

The `pr_curve()` function returns a similar data frame containing the `recall` and `precision` instead of `specificity` and `sensitivity`.

```
rf_auc<- last_rf_fit %>%
  collect_predictions() %>%
  roc_curve(wine_color,.pred_red) %>%
  mutate(model = "Random Forest")

rf_pr<- last_rf_fit %>%
  collect_predictions() %>%
  pr_curve(wine_color,.pred_red) %>%
  mutate(model = "Random Forest")
```

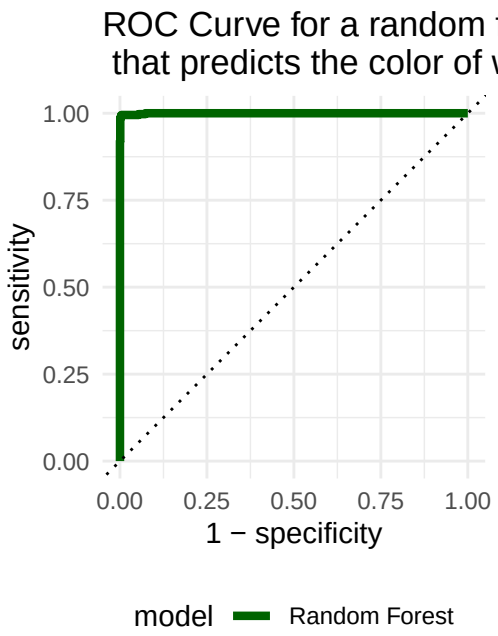
We can generate the curve plots using the `autoplot()` function or `ggplot`. An example for both can be found below.

```
rf_auc %>% autoplot()
```



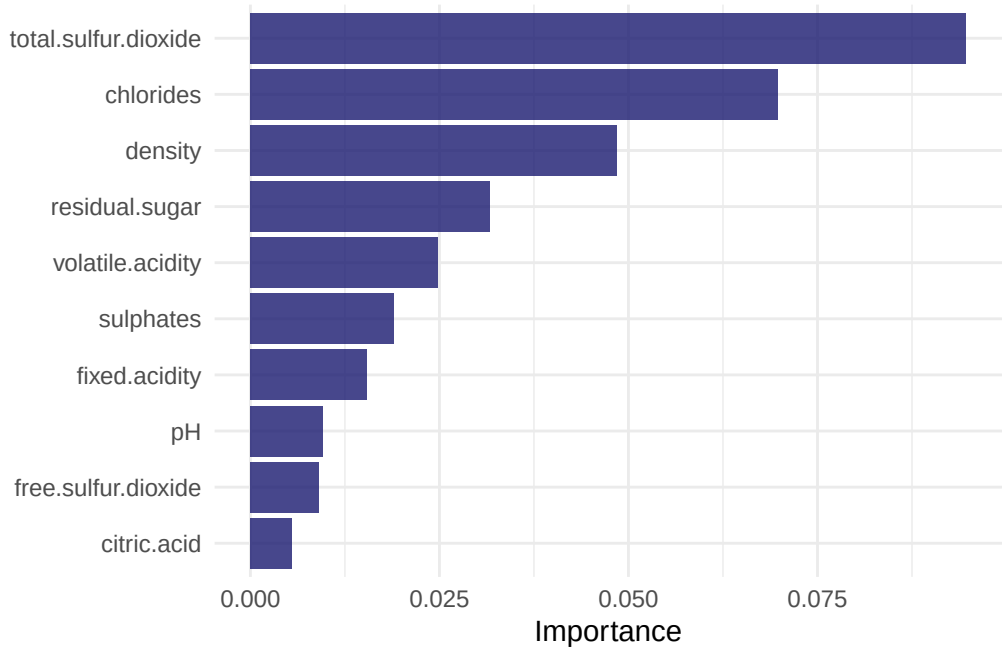

```
rf_auc %>%
  ggplot(aes(x = 1 - specificity, y = sensitivity, col = model)) +
  geom_path(lwd = 1.5) +
  geom_abline(lty = 3) +
  coord_equal() +
  scale_color_manual(values = "darkgreen")+
  labs(
```

```
    title = "ROC Curve for a random forest \n that predicts the color of wine"
  )+
  theme_minimal(base_size = 11)+
  theme(legend.position = "bottom")
```



To generate a feature importance plot with respect to the measure `"permutation"` feature importance, we first have to extract the fit engine from final model fit and then apply the `vip()` function of the `vip` library. The `vip` function creates a (ggplot) plot, showing the importance scores for the predictors in a model. The `geom` argument specifies what kind of plot is generated. Setting `geom = "col"` therefore creates a column plot. Other options include `"boxplot"`, `"point"`, and `"violin"`.

```
library(vip)

last_rf_fit %>%
  extract_fit_engine()%>%
  vip(geom = "col",
           aesthetics = list(fill = "midnightblue",
                             alpha = 0.8)
           )+
  theme_minimal(base_size = 11)
```



Considering the figure above, the variable `total.sulfur.dioxide` has the highest score which indicates that this variable helps determining the color of the wines most.

5.3 Exercises

5.3.1 Theoretical Exercises

In this exercise we will discuss some aspects of bootstrap sampling, bagging and random forest.

Exercise 5.1. Assume, we have a data set with n sample and a bootstrap sample of size n. Furthermore, assume that the probability of an observation not being in the bootstrap sample is $\left(1-\frac{1}{n}\right)^n$. Show that the probability for any sample j to be in the data set is approximately 0.6321206.

Exercise 5.2. In terms of bagging, explain the following sentence from the lecture:

> Having similar trees leads to correlated estimates.

Exercise 5.3. Random forests can solve the problem mentioned in Exercise 5.2 of having trees that are too similar. Describe how this is achieved!

5.3.2 Programming Exercises

In this exercise we want to utilize our newly gained knowledge about Bagging and compare a random forest model to a single classification tree and penalized logistic regression.

The dataset we will consider in this exercise will be the Credit Card Customers data set that can either be downloaded using the provided link or the button below.

Download BankChurners

Recall that the data set consists of 10,127 entries that represent individual customers of a bank including but not limited to their age, salary, credit card limit, and credit card category.

The main idea for such classification tasks is the following:

1. Start out by building a simple base model, which allows for an easy interpretation of parameters. A penalized logistic regression will be this base model in our case.
2. Move to a slightly more complex model where the interpretation of model parameters is less straight forward, but the model performance increases. The model we will consider for this scenario is a decision tree.
3. As a last step, a highly complex model is trained where the focus is no longer on explainability rather than getting the best possible out of sample performance. An example of such a model is a random forest, which will also be our model of choice for this step.

Consider the following glimpse into the dataset:

```
Rows: 10,127
Columns: 21
$ CLIENTNUM                <int> 768805383, 818770008, 713982108, 769911858, 7~
$ Attrition_Flag           <chr> "Existing Customer", "Existing Customer", "Ex~
$ Customer_Age             <int> 45, 49, 51, 40, 40, 44, 51, 32, 37, 48, 42, 6~
$ Gender                   <chr> "M", "F", "M", "F", "M", "M", "M", "M", "M", ~
$ Dependent_count          <int> 3, 5, 3, 4, 3, 2, 4, 0, 3, 2, 5, 1, 1, 3, 2, ~
$ Education_Level          <chr> "High School", "Graduate", "Graduate", "High ~
$ Marital_Status           <chr> "Married", "Single", "Married", "Unknown", "M~
$ Income_Category          <chr> "$60K - $80K", "Less than $40K", "$80K - $120~
$ Card_Category            <chr> "Blue", "Blue", "Blue", "Blue", "Blue", "Blue~
$ Months_on_book           <int> 39, 44, 36, 34, 21, 36, 46, 27, 36, 36, 31, 5~
$ Total_Relationship_Count <int> 5, 6, 4, 3, 5, 3, 6, 2, 5, 6, 5, 6, 3, 5, 5, ~
$ Months_Inactive_12_mon   <int> 1, 1, 1, 4, 1, 1, 1, 2, 2, 3, 3, 2, 6, 1, 2, ~
$ Contacts_Count_12_mon    <int> 3, 2, 0, 1, 0, 2, 3, 2, 0, 3, 2, 3, 0, 3, 2, ~
$ Credit_Limit             <dbl> 12691.0, 8256.0, 3418.0, 3313.0, 4716.0, 4010~
$ Total_Revolving_Bal      <int> 777, 864, 0, 2517, 0, 1247, 2264, 1396, 2517,~
$ Avg_Open_To_Buy          <dbl> 11914.0, 7392.0, 3418.0, 796.0, 4716.0, 2763.~
```

```
$ Total_Amt_Chng_Q4_Q1     <dbl> 1.335, 1.541, 2.594, 1.405, 2.175, 1.376, 1.9~
$ Total_Trans_Amt          <int> 1144, 1291, 1887, 1171, 816, 1088, 1330, 1538~
$ Total_Trans_Ct           <int> 42, 33, 20, 20, 28, 24, 31, 36, 24, 32, 42, 2~
$ Total_Ct_Chng_Q4_Q1      <dbl> 1.625, 3.714, 2.333, 2.333, 2.500, 0.846, 0.7~
$ Avg_Utilization_Ratio    <dbl> 0.061, 0.105, 0.000, 0.760, 0.000, 0.311, 0.0~
```

Since some of the features are kind of ambiguous, let us briefly talk about what they mean.

Feature	Description
CLIENTNUM	Client number. Unique identifier for the customer holding the account
Attrition_Flag	Internal event (customer activity) variable - if the account is closed then 1 else 0
Months_on_book	Period of relationship with bank
Months_Inactive_12_mon	No. of months inactive in the last 12 months
Credit_Limit	Credit Limit on the Credit Card
Total_Revolving_Bal	Portion of credit card spending that goes unpaid at the end of a billing cycle
Avg_Open_To_Buy	Open to Buy Credit Line (Average of last 12 months)
Total_Amt_Chng_Q4_Q1	Change in Transaction Amount (Q4 over Q1)
Total_Trans_Amt	Total Transaction Amount (Last 12 months)
Total_Trans_Ct	Total Transaction Count (Last 12 months)
Total_Ct_Chng_Q4_Q1	Change in Transaction Count (Q4 over Q1)
Avg_Utilization_Ratio	Average Card Utilization Ratio (Divide the total balance by the total credit limit)

Exercise 5.4. In the first exercise session, we already performed some exploratory data analysis, focusing on the demographics of the customers. Since we are mainly interested in predicting the attrition flag, find out the no-information rate (NIR) defined by

$$\max\left(\frac{\mathrm{P}}{\mathrm{N}+\mathrm{P}}, \frac{\mathrm{N}}{\mathrm{N}+\mathrm{P}}\right)$$

Exercise 5.5. Create a training and test split using `set.seed(121)` and a 5-fold CV object based on the training data. Use stratification for the target variable `Attrition_Flag` to ensure that the ratio of positive and negative sample remains the same in the training and testing data.

Exercise 5.6. Create a recipe by following the steps described below.

1. As a formula, fit the variable `Attrition_Flag` on every other feature and set the `data` parameter to `data_train`.

2. Update the role of the variable `CLIENTNUM` by setting it to `"ID"`.
3. Convert all `"Unknown"` and `"unknown"` values contained in character or factor columns into NA values using the code snippet

```
across(
  where(~is.character(.)|is.factor(.)),
  ~if_else(.%in% c("Unknown","unknown"),NA,.)
  )
```

4. Convert the features `Income_Category` and `Education_Level` into ordered factors.
5. Convert the features `Marital_Status`, `Card_Category`, `Gender`, `CLIENTNUM`, and the outcome variable `Attrition_Flag` into factors.
6. For the factor Attrition_Flag, change the labels to `"Inactive"` and `"Active"`.
7. Create ordinal scores for all ordered predictors.
8. Impute all `NA` values using kNN imputation.
9. Create dummy variables for all factor predictors.
10. Apply a zero variance filter on all predictors.
11. Familiarize yourself with the `step_corr` function, add it to the recipe, and apply it to all predictors.

Exercise 5.7. Create a workflow object and add the newly created recipe `rec_ci`. Afterwards, create a `metric_set` that contains the metrics `roc_auc`,`pr_auc`,`accuracy`,`precision`, and `recall`.

Exercise 5.8 (Tuning a lasso model)**.**

1. Utilize the `logistic_reg` function to create a lasso model.
2. Create a regular grid for the logistic model penalty with 30 levels.
3. Tune the linear model using the 5-fold CV object created in Exercise 5.5, the grid specified in 2., and the metric set specified in Exercise 5.7.

Exercise 5.9.

1. Given the results of the previous exercise, select the best model according to the "one-standard" rule based on the `"pr_auc"` metric.
2. Train a final model on the whole training data.
3. Create two data frames containing the points of the models' PR- and ROC-curve and visualize them.

Exercise 5.10 (Bonus Exercise)**.** The following exercise is not mandatory but still helps for gaining a deeper understanding of the penalization behavior. Since we have used a lasso logistic regression, some of the parameters might have been driven to 0. Find if there were any!

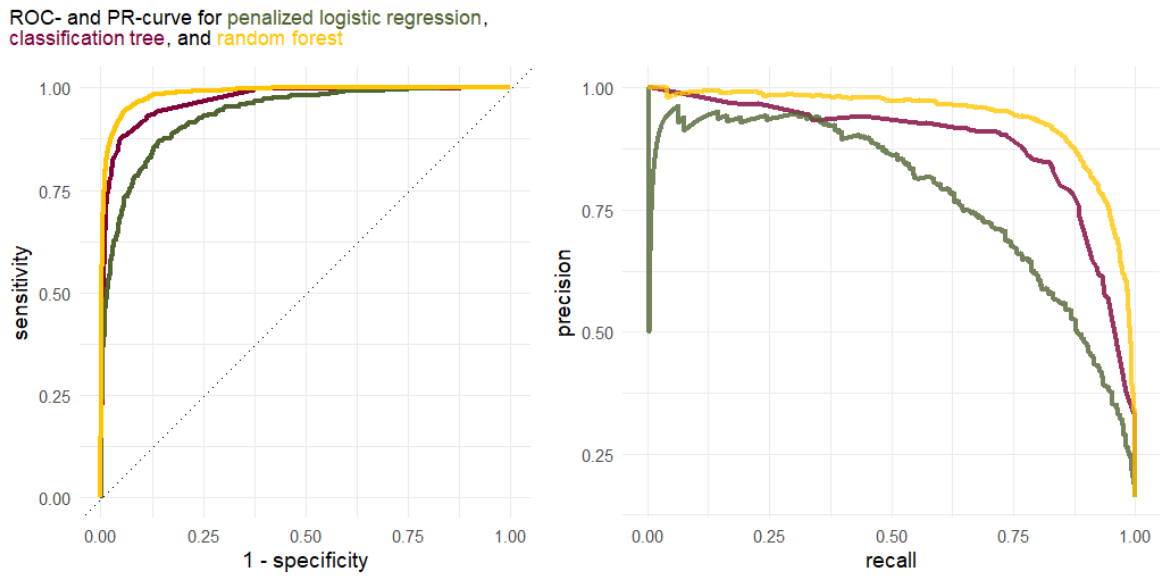
Exercise 5.11. Repeat Exercise 5.8 - Exercise 5.9 by tuning a classification tree. Tune the parameters `min_n`, `tree_depth`, and `cost_complexity` using 5-fold CV and a regular grid with four levels. Instead of using the one standard error rule, use the `select_best` function instead.

Exercise 5.12. Use the `vip::vip` function to find the most important features of the final classification tree.

Exercise 5.13. Repeat Exercise 5.11 and Exercise 5.12 for a random forest model with 1000 trees. Tune the parameters `mtry` and `min_n` with a grid of size 10 using 5-fold CV.

Exercise 5.14. Given the following plots, answer the following questions:

1. What can be said about the discriminatory power of the classes?
2. Which curve should be considered for assessing the accurracy of the models?
3. Which model performs the best?



Exercise 5.15 (Bonus exercise)**.** Use the previously saved data frames containing the ROC- and PR-curve data for each model to recreate the plot in exercise Exercise 5.14

5.4 Solutions

Solution 5.1 (Exercise 5.1). For n sufficiently big, the estimate $\left(1-\frac{1}{n}\right)^n \approx \exp(-1)$ holds. Since the probability of any sample not being in the data set is therefore approximately $\exp(-1)$, we can simply calculate the complementary probability. The complementary probability is given by $1-\exp(-1) \approx 0.63212$.

Solution 5.2 (Exercise 5.2). Bagging trees leads to fitting many trees with similar structure as the same features tend to be selected in the same step in different trees. Given that the estimates are depending on the splits of a tree, the estimates can be highly correlated if the tree structures are similar.

Solution 5.3 (Exercise 5.3). When bagging trees, the number of features for building a tree stays the same. A random forest on the other hand only selects a subset of all the features. This ensures that there is enough variability in the different trees and thus directly tackles the problem of the trees being too similar.

Solution 5.4 (Exercise 5.4).

```
NIR<- credit_info %>%
  group_by(Attrition_Flag)%>%
  summarise(n=n()) %>%
  mutate(NIR = n/sum(n)) %>%
  pluck(3) %>%
  max()

glue::glue(
  "The NIR of the underlying dataset is {round(NIR,3)},
   meaning that a classification model should have
   an accuracy of at least {round(NIR,3)}."
  )
```

```
The NIR of the underlying dataset is 0.839,
meaning that a classification model should have
an accuracy of at least 0.839.
```

Solution 5.5 (Exercise 5.5). Create a training and test split using `set.seed(121)` and a 5-fold CV object based on the training data.

```
set.seed(121)
split <- initial_split(credit_info, strata = Attrition_Flag)
data_train_ci <- training(split)
data_test_ci <- testing(split)
folds_ci <- vfold_cv(data_train_ci, v = 5)
```

Solution 5.6 (Exercise 5.6).

```
levels_income <- c("Less than $40K","$40K - $60K",
                   "$60K - $80K","$80K - $120K","$120K +")

levels_education <- c("Uneducated", "High School","College",
                      "Graduate",  "Post-Graduate", "Doctorate")

rec_ci <- recipe(Attrition_Flag ~., data = data_train_ci) %>%
  update_role(CLIENTNUM, new_role = "ID") %>%
  step_mutate_at(all_nominal_predictors(),
                 fn = ~if_else(.%in% c("Unknown","unknown"),NA,.)
                 ) %>%
  step_string2factor(Income_Category,
                     levels = levels_income,
                     ordered = TRUE) %>%
  step_string2factor(Education_Level,
                     levels = levels_education,
                     ordered = TRUE) %>%
  step_string2factor(Attrition_Flag) %>%
  step_ordinalscore(all_ordered_predictors()) %>%
  step_unknown(all_factor_predictors()) %>%
  step_impute_knn(all_predictors()) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_corr(all_predictors())
```

Solution 5.7 (Exercise 5.7).

```
ci_wf <- workflow() %>%
  add_recipe(rec_ci)

multi_metrics <- metric_set(roc_auc,pr_auc,accuracy,recall)
```

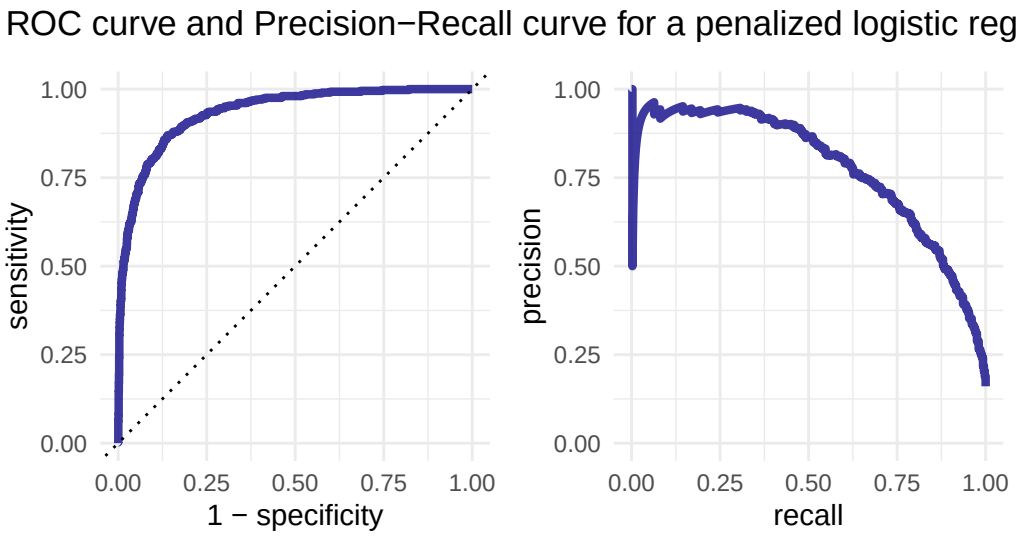
Solution 5.8 (Exercise 5.8).

```
log_mod_tune_spec <-logistic_reg(penalty = tune(), mixture = 1) %>%
  set_engine("glmnet")

ci_wf <- ci_wf %>% add_model(log_mod_tune_spec)

lr_grid <- ci_wf %>%
  extract_parameter_set_dials %>%
  grid_regular(levels = 30)

lr_tune_res <- ci_wf %>%
  tune_grid(
    grid = lr_grid,
    metrics = multi_metrics,
    resamples = folds_ci
  )
```

Solution 5.9 (Exercise 5.9).

```
lr_res_best <- lr_tune_res %>%
  select_by_one_std_err(metric = "pr_auc", desc(penalty))

last_lr_fit <- ci_wf %>%
  finalize_workflow(lr_res_best) %>%
  last_fit(split,
           metrics = multi_metrics)

lr_auc<- last_lr_fit %>%
  collect_predictions() %>%
  roc_curve(Attrition_Flag,`.pred_Attrited Customer`) %>%
  mutate(model = "Logistic Regression")

lr_pr<- last_lr_fit %>%
  collect_predictions() %>%
  pr_curve(Attrition_Flag,`.pred_Attrited Customer`) %>%
  mutate(model = "Logistic Regression")


p1 <- lr_auc %>%
  ggplot(aes(x = 1 - specificity, y = sensitivity, col = model)) +
  geom_path(lwd = 1.5, alpha = 0.8) +
```

```
  geom_abline(lty = 3) +
  coord_equal() +
  scale_color_viridis_d(option = "plasma", end = .6)+
  ylim(c(0,1))+
  theme_minimal(base_size = 11)+
  theme(legend.position = "none")

p2 <- lr_pr %>%
  ggplot(aes(x = recall, y = precision, col = model)) +
  geom_path(lwd = 1.5, alpha = 0.8) +
  coord_equal() +
  scale_color_viridis_d(option = "plasma", end = .6)+
  ylim(c(0,1))+
  theme_minimal(base_size = 11)+
  theme(legend.position = "none")

p<-p1 | p2

p + plot_annotation(
  title = "ROC curve and Precision-Recall curve for a penalized logistic regression"
)
```



Exercise 5.16 (Exercise 5.16).

```
last_lr_fit %>%
  extract_fit_parsnip() %>%
  tidy() %>%
  filter(estimate == 0) %>%
  select(term)
```

```
# A tibble: 10 x 1
   term
   <chr>
 1 Avg_Utilization_Ratio
 2 Education_Level_Graduate
 3 Education_Level_High.School
 4 Education_Level_Uneducated
 5 Education_Level_unknown
 6 Marital_Status_unknown
 7 Income_Category_X.40K....60K
 8 Income_Category_X.80K....120K
 9 Income_Category_Less.than..40K
10 Income_Category_unknown
```

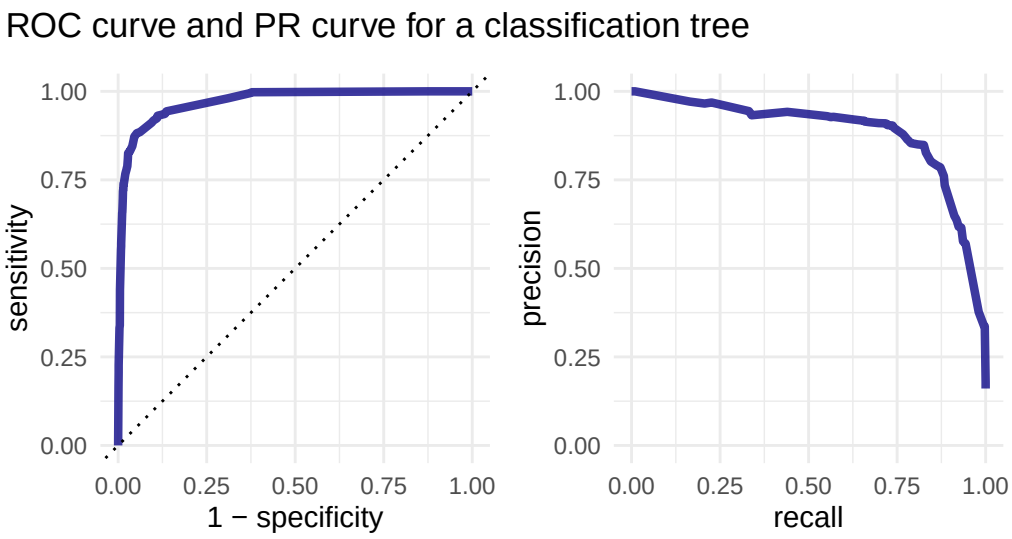
```
last_lr_fit %>%
  extract_fit_parsnip() %>%
  tidy() %>%
  filter(estimate > 0) %>%
  arrange(desc(estimate)) %>%
  select(term)
```

```
# A tibble: 11 x 1
   term
   <chr>
 1 Total_Ct_Chng_Q4_Q1
 2 Gender_M
 3 Total_Relationship_Count
 4 Marital_Status_Married
 5 Total_Amt_Chng_Q4_Q1
 6 Total_Trans_Ct
 7 Income_Category_X.60K....80K
 8 Months_on_book
 9 Customer_Age
10 Total_Revolving_Bal
11 Credit_Limit
```

Solution 5.10 (Exercise 5.11).

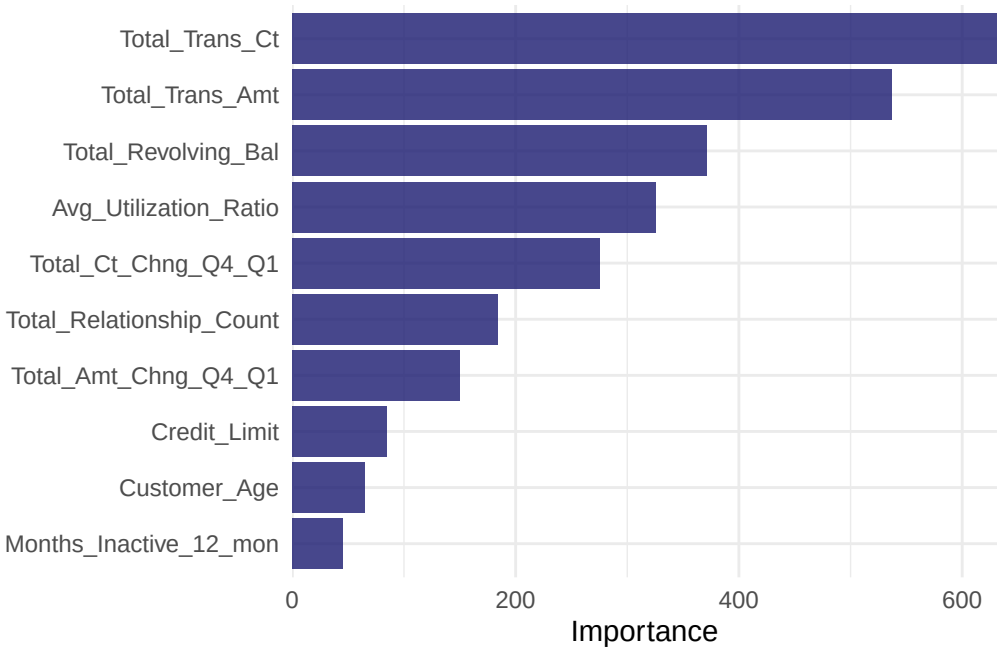
```
ct_model_spec <- decision_tree(
  min_n = tune(),
  tree_depth = tune(),
  cost_complexity = tune()
) %>%
  set_mode("classification")

ci_wf <- ci_wf %>% update_model(ct_model_spec)
ct_grid <- ci_wf %>%
  extract_parameter_set_dials() %>%
  grid_regular(levels = 4)

ct_tune_res <- ci_wf %>%
 tune_grid(
   grid = ct_grid,
   metrics = multi_metrics,
   resamples = folds_ci
)

ct_res_best <- ct_tune_res %>%
  select_best(metric = "pr_auc")

last_ct_fit <- ci_wf %>%
  finalize_workflow(ct_res_best) %>%
  last_fit(split,
           metrics = multi_metrics)


ct_auc<- last_ct_fit %>%
  collect_predictions() %>%
  roc_curve(Attrition_Flag,`.pred_Attrited Customer`) %>%
  mutate(model = "Classification Tree")

ct_pr<- last_ct_fit %>%
  collect_predictions() %>%
  pr_curve(Attrition_Flag,`.pred_Attrited Customer`) %>%
  mutate(model = "Classification Tree")

p1 <- ct_auc %>%
  ggplot(aes(x = 1 - specificity, y = sensitivity, col = model)) +
  geom_path(lwd = 1.5, alpha = 0.8) +
```

```
  geom_abline(lty = 3) +
  coord_equal() +
  scale_color_viridis_d(option = "plasma", end = .6)+
  ylim(c(0,1))+
  theme_minimal(base_size = 11)+
  theme(legend.position = "none")

p2 <- ct_pr %>%
  ggplot(aes(x = recall, y = precision, col = model)) +
  geom_path(lwd = 1.5, alpha = 0.8) +
  coord_equal() +
  scale_color_viridis_d(option = "plasma", end = .6)+
  ylim(c(0,1))+
  theme_minimal(base_size = 11)+
  theme(legend.position = "none")

p<-p1 | p2

p + plot_annotation(
  title = "ROC curve and PR curve for a classification tree"
)
```



Solution 5.11 (Exercise 5.12).

```
last_ct_fit %>%
  extract_fit_engine()%>%
  vip(geom = "col", aesthetics = list(fill = "midnightblue", alpha = 0.8)) +
  scale_y_continuous(expand = c(0, 0))+
  theme_minimal(base_size = 11)
```



Solution 5.12 (Exercise 5.13).

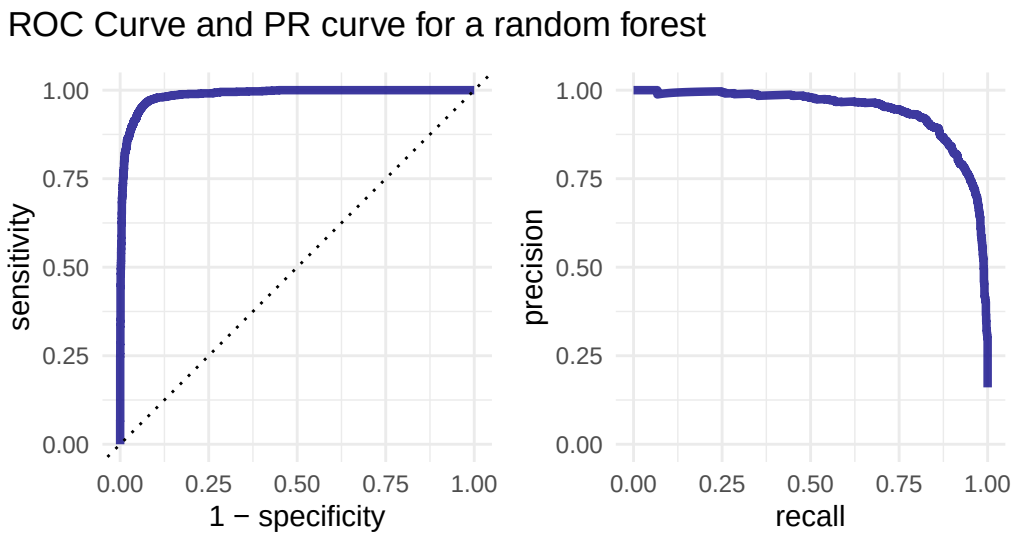
```
cores <- parallel::detectCores()

rf_model_spec <- rand_forest(
  mode = "classification",
  mtry = tune(),
  min_n = tune(),
  trees = 1000
) %>%
  set_engine("ranger",
             num.threads = cores,
             importance = "permutation")

ci_wf <- ci_wf %>% update_model(rf_model_spec)
```

```
rf_res <- ci_wf %>%
  tune_grid(grid = 10,
            metrics = multi_metrics,
            resamples = folds_ci,
            control = control_grid(save_pred = TRUE)
  )
```

```
i Creating pre-processing data to finalize unknown parameter: mtry
```

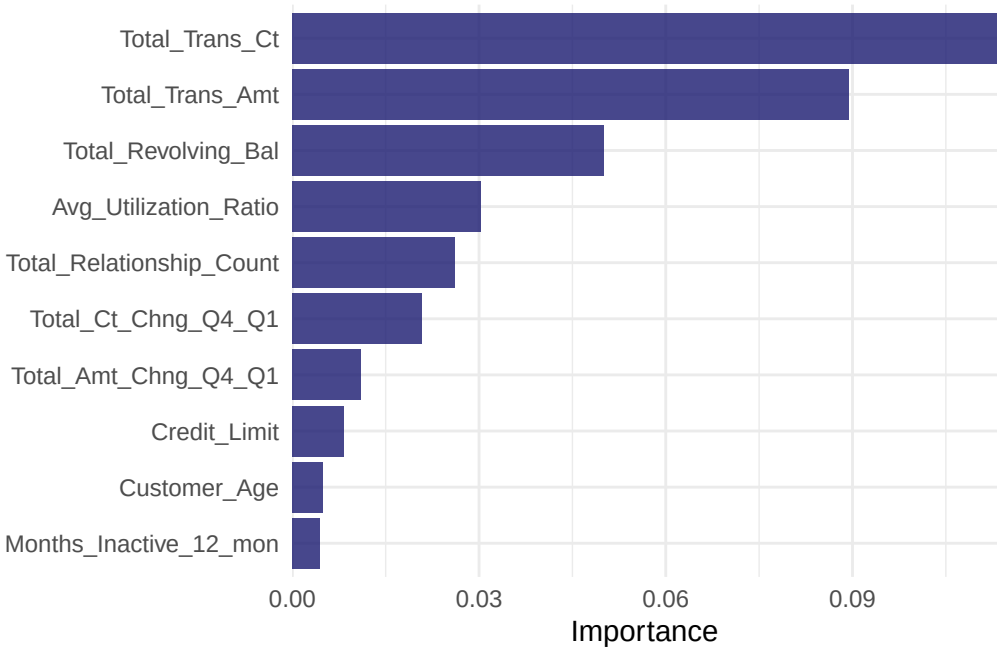
```
rf_res_best <- rf_res %>% select_best(metric = "roc_auc")

rf_auc <-
  rf_res %>%
  collect_predictions(parameters = rf_res_best) %>%
  roc_curve(Attrition_Flag, `.pred_Attrited Customer`) %>%
  mutate(model = "Random Forest")

rf_pr <-
  rf_res %>%
  collect_predictions(parameters = rf_res_best) %>%
  pr_curve(Attrition_Flag, `.pred_Attrited Customer`) %>%
  mutate(model = "Random Forest")

p1 <- rf_auc %>%
  ggplot(aes(x = 1 - specificity, y = sensitivity, col = model)) +
  geom_path(lwd = 1.5, alpha = 0.8) +
  geom_abline(lty = 3) +
  coord_equal() +
  scale_color_viridis_d(option = "plasma", end = .6)+
  ylim(c(0,1))+
  theme_minimal(base_size = 11)+
  theme(legend.position = "none")

p2 <- rf_pr %>%
  ggplot(aes(x = recall, y = precision, col = model)) +
  geom_path(lwd = 1.5, alpha = 0.8) +
  coord_equal() +
  scale_color_viridis_d(option = "plasma", end = .6)+
  ylim(c(0,1))+
  theme_minimal(base_size = 11)+
  theme(legend.position = "none")
```

```
p<-p1 | p2

p + plot_annotation(
  title = "ROC Curve and PR curve for a random forest"
)
```



Feature importance plot:

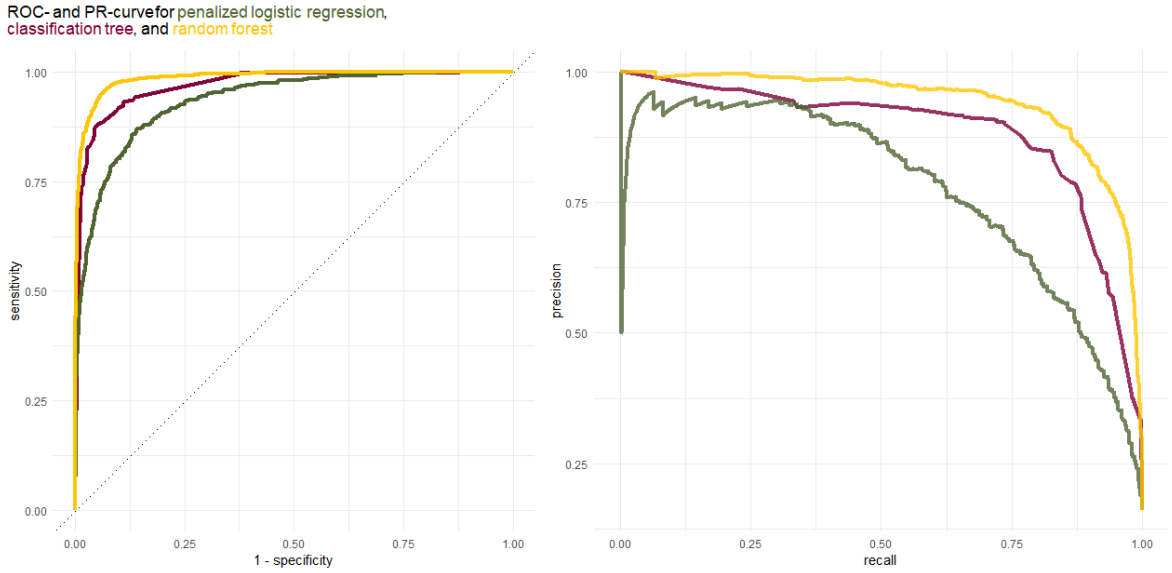
```
last_rf_fit <- ci_wf %>%
  finalize_workflow(rf_res_best) %>%
  last_fit(split)

last_rf_fit %>%
  extract_fit_parsnip()%>%
  vip(geom = "col", aesthetics = list(fill = "midnightblue", alpha = 0.8)) +
  scale_y_continuous(expand = c(0, 0))+
  theme_minimal(base_size = 11)
```



Solution 5.13 (Exercise 5.15).

```
cols <- c("#80003A","#506432","#FFC500")
names(cols) <- c("cl", "lr", "rf")
plot_title <- glue::glue(
    "ROC- and PR-curve for <span style='color:{cols['lr']};'>
    penalized logistic regression</span>,<br>
    <span style='color:{cols['cl']};'>classification tree</span>,
    and <span style='color:{cols['rf']};'>random forest</span>"
    )
p1 <- bind_rows(ct_auc, lr_auc, rf_auc) %>%
  ggplot(aes(x = 1 - specificity, y = sensitivity, col = model)) +
  geom_path(lwd = 1.5) +
  geom_abline(lty = 3) +
  coord_equal() +
  scale_color_manual(values = unname(cols))+
  theme_minimal(base_size = 11)+
  theme(legend.position = "none")


p2 <- bind_rows(ct_pr, lr_pr, rf_pr) %>%
  ggplot(aes(x = recall, y = precision, col = model)) +
  geom_path(lwd = 1.5, alpha = 0.8) +
```

```
  coord_equal() +
  scale_color_manual(values = unname(cols))+
  theme_minimal(base_size = 11)+
  theme(legend.position = "none")


(p1|p2) +
  plot_annotation(
  title = plot_title,
  theme = theme(plot.title = element_markdown()))
```



6 Boosting

6.1 Introduction

In this exercise session, we will consider multiple advanced machine learning models. Our base model will not be a penalized logistic regression as in Session 05 rather than a random forest. The models we are considering subsequently are also widely used in application as their performance on classification tasks is superb! However, similar to random forests, their explainability is still subpar compared to a simple logistic regression or classification tree. Before we learn how to train and finetune these models, we will discuss some theoretical aspects.

6.1.1 Confusion matrices in R

Since we will be working with on a classification task in the exercises, being able to construct a confusion matrix is crucial.

Consider the following example data set which is part of the `{yardstick}` library:

```
library(tidyverse)
library(yardstick)

two_class_example %>% glimpse()
```

```
Rows: 500
Columns: 4
$ truth     <fct> Class2, Class1, Class2, Class1, Class2, Class1, Class1, Clas~
$ Class1    <dbl> 0.0035892426, 0.6786210540, 0.1108935221, 0.7351617031, 0.01~
$ Class2    <dbl> 9.964108e-01, 3.213789e-01, 8.891065e-01, 2.648383e-01, 9.83~
$ predicted <fct> Class2, Class1, Class2, Class1, Class2, Class1, Class1, Clas~
```

Say, we want to label change the label `Class1` to `Positive` and `Class2` to `Negative`. Then, we can simply apply the `mutate()` function:

```
two_class_example <- two_class_example %>%
  mutate(
    truth = fct_relabel(truth,
                        ~if_else(.=="Class1","Positive","Negative")
                        ),
    predicted = fct_relabel(predicted,
                          ~if_else(.=="Class1","Positive","Negative")
                          )
  )
```

To create a simple confusion matrix, we can use the `conf_mat` function that is also part of the `{yardstick}` library:

```
(cm_example <- two_class_example %>%
   conf_mat(truth =truth,estimate = predicted))
```

```
          Truth
Prediction Positive Negative
  Positive      227       50
  Negative       31      192
```

We can also use the ggplot function to create a visually more appealing version of this matrix. To do so, we first have to convert the confusion matrix into a proper data frame and set the levels of the Predictions and Truth.

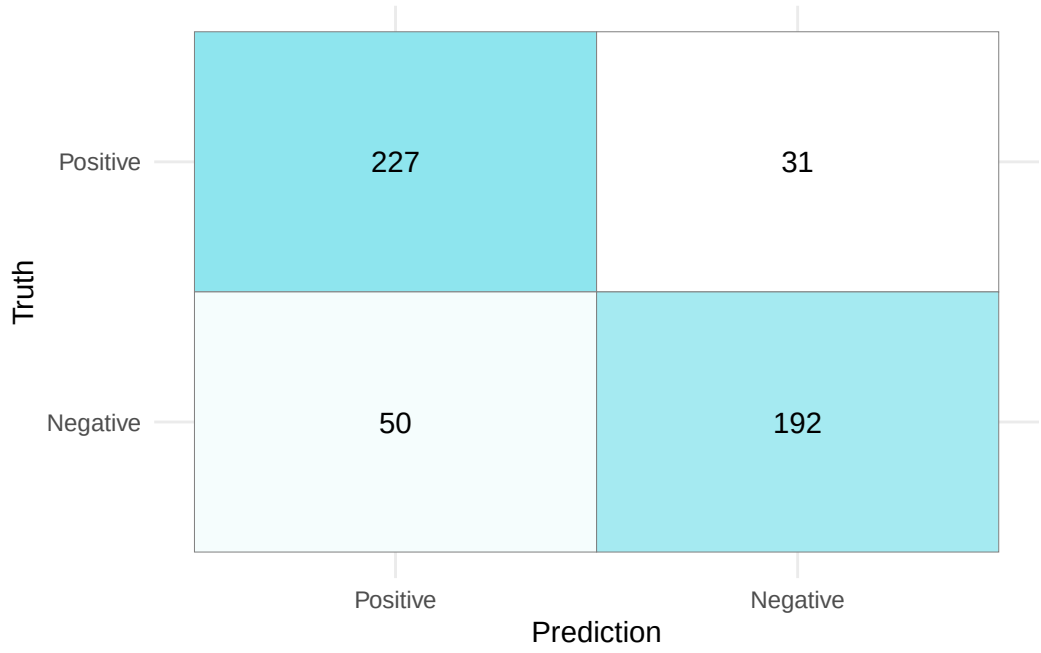
```
cm_tib<- as_tibble(cm_example$table)%>%
  mutate(
    Prediction = factor(Prediction,
                        levels = rev(levels(factor(Prediction)))),
    Truth = factor(Truth)
)
```

In the snippet above we had to reverse the levels of the variable `Prediction` so that we can place the `TP` values in the top left, and `TN` values in the bottom right of the confusion matrix.

Once the confusion matrix has been converted to a data frame, we can pass it into the ggplot function with the argument `fill` set to `n`. The `geom_tile()` function places a tile at each coordinate provided by the data frame (Note: Coordinates are discrete and given by `Negative` and `Positive`). The argument `colour = "gray50"` adds a gray border to each tile. By adding the `geom_text()` function where the aesthetics are provided by the `label` argument, we can add the number of samples falling into each class (TP, TN, FP, and FN) to each tile. The

`scale_fill_gradient()` function, allows to change the colors of the tiles with respect to the value of `n`. Here, a low value of `n` indicates the the tile will be `"white"` and a high value of `n` indicates that the color of the tile will be light green (with HEX Code `"#9AEBA3"`). Setting the theme to minimal, and removing the legend yields a cleaner representation of the confusion matrix.

```
cm_tib %>% ggplot(aes(x = Prediction, y = Truth,fill = n)) +
    geom_tile( colour = "gray50")+
    geom_text(aes(label = n))+
    scale_fill_gradient(low = "white", high = "cadetblue2")+
    theme_minimal()+
    theme(legend.position = "none")
```



6.1.2 Tuning an XGBoost model

Since we have intensively covered random forests in previous exercises, we only consider an XGBoost model in this introduction. AdaBoost is rarely used in practice anymore, which is why we will directly move towards training an XGBoost model. The approach is similar to training and tuning every other model but compared to previous exercises we will not perform cross validation, rather than a simple training/validation/test split to save some time.

We can create an XGBoost model by using the `boost_tree` function. Looking at the documentation, you will notice that there are quite a few parameters for us to consider:

Parameter	Description
`trees`	Number of trees contained in the ensamble
`tree_depth`	Integer for the maximum depth of the trees
`min_n`	Minimum number of data points in a node required for a split
`mtry`	Number of randomly selected at each split

The parameters above are not new to us. In fact, they are the exact same parameters we use for training a random forest model. That is why we will not go into detail with respect to the ones above.

There are, however, a few new parameters that are worth an explanation:

Parameter	Description
`loss_reduction`	Number for the reduction in loss that is required to split further $\in [0, \infty]$
`sample_size`	Subsample ratio of the training instances $\in (0, 1]$.
`learn_rate`	Rate at which the algorithm adapts from iteration to iteration $\in [0, 1]$.

The three parameters above have only been referenced in the lectures so far, so let's quickly describe them in a bit more detail.

6.1.2.0.1 `loss_reduction` :

In Exercise 6.2, we will derive the optimal expansion coefficient α (similar to the coefficients in linear regression) which solves the minimization problem

$$(\alpha_b, h_b) = \arg \min_{\alpha>0, h \in \mathcal{H}} \sum_{n=1}^{N} L(y_n, \hat{f}^{(b-1)}(x_n) + \alpha h(x_n))$$

Here, L denotes a loss function that we aim to minimize with respect to α and an additional (potentially weak) learner h that we add to the previous estimator.

If the term

$$\left| \sum_{n=1}^{N} L(y_n, \hat{f}^{(b-1)}(x_n) + \alpha h(x_n)) - \sum_{n=1}^{N} L(y_n, \hat{f}^{(b)}(x_n) + \alpha h(x_n)) \right|,$$

i.e., the loss reduction between step b and $b+1$ is smaller than the parameter `loss_reduction`, the algorithm stops.

6.1.2.0.2 `sample_size :`

Let $q \in (0, 1]$ denote the `sample_size` parameter and N the number of samples in our training data. Then, XGBoost selects $q \cdot N$ samples prior to growing trees. This subsampling occurs once in every boosting iteration.

6.1.2.0.3 `learn_rate :`

In simple terms, the learning rate specifies how quickly the model adapts to the training data. An analogy can be drawn to gradient based models that use gradient descent on a loss function. Here, the goal is to minimize the loss function by stepping towards its minimum. To illustrate the learning rate in a gradient descent context, consider the following examples where we can imagine the polynomial of degree four to be a loss function that we try to minimize.

Choosing a learning rate that is too high, might result in missing an optimal model because it is being stepped over, while a learning rate chosen too small might result in the objective never being reached at all.

Similar to choosing a learning rate that is too high, we could also choose a learning rate that is too low, resulting in the global minimum never being reached at all.

The learning rate in the XGBoost algorithm describes a factor γ that scales the output of the most recently fit tree that is added to the model. In simple terms, the learning rate in the XGBoost algorithm describes a shrinkage parameter.

In the following example, we will try to predict the base rent prices in Munich using an XGBoost model. The data Apartment rental offers in Germany is the same as in Exercise 04.

```
library(tidymodels)

data_muc <- read.csv("data/rent_muc.csv")
```

Instead of using a cross validation approach, we will use a simple training/validation/test split to reduce computing time.

By using the `validation` split function on the training data, we split the training data into a training and validation subset. The `data_val` object can then be passed into the `tune_grid` function in the same fashion as we did with a cross validation object.

```
set.seed(24)
split_rent <- initial_split(data_muc)
data_train <- training(split_rent)
data_val <- validation_split(data_train)
data_test <- testing(split_rent)
```

Preprocessing of the data is handled by the following recipe.

```
rec_rent <- recipe(
    formula = baseRent ~.,
    data = data_train
  ) %>%
  update_role(scoutId, new_role = "ID") %>%
  step_select(!c("serviceCharge","heatingType","picturecount",
                 "totalRent",   "firingTypes","typeOfFlat",
                 "noRoomsRange", "petsAllowed",
                 "livingSpaceRange","regio3","heatingCosts",
                 "floor","date", "pricetrend")) %>%
  step_mutate(
    interiorQual = factor(
      interiorQual,
      levels = c("simple", "normal", "sophisticated", "luxury"),
      ordered = TRUE
      ),
    condition = factor(condition,
      levels = c("need_of_renovation", "negotiable","well_kept",
                 "refurbished","first_time_use_after_refurbishment",
                 "modernized","fully_renovated", "mint_condition",
                 "first_time_use"),
      ordered = TRUE),
    geo_plz = factor(geo_plz),
              across(where(is.logical),~as.numeric(.))) %>%
  step_string2factor(all_nominal_predictors(),
                     all_logical_predictors()) %>%
  step_ordinalscore(all_ordered_predictors())%>%
  step_novel(all_factor_predictors())%>%
  step_unknown(all_factor_predictors()) %>%
  step_dummy(geo_plz)%>%
  step_impute_knn(all_predictors()) %>%
  step_filter(baseRent <= 4000, livingSpace <= 200)
```

After preprocessing the data, we can create a workflow object and specify our XGBoost model.

```
wf_rent <- workflow() %>%
  add_recipe(rec_rent)
```

We want to tune every parameter except for `trees`. Since we are using a regression model, we need to use the mode `"regression"`.

```
set.seed(121)

xgb_model <- boost_tree(
  trees = 1000,
  tree_depth = tune(),
  min_n = tune(),
  mtry = tune(),
  loss_reduction = tune(),
  learn_rate = tune()
) %>%
  set_mode("regression")

wf_rent <- wf_rent %>% add_model(xgb_model)
```

To tune the model and select the best model based on the performance on the validation data, we use the `tune_grid` function.

```
multi_metrics <- metric_set(rmse,rsq,mae)

xgb_tune_res <- wf_rent %>%
  tune_grid(
    resamples = data_val,
    metrics = multi_metrics,
    grid = 20,
  )
```

After tuning the model parameters, we use the optimal candidate hyperparameters to train a final model on all the training data and evaluate it on the test data.

```
xgb_best_parm <- xgb_tune_res %>% select_best(metric = "rmse")

last_xgb_fit <- wf_rent %>%
  finalize_workflow(xgb_best_parm) %>%
  last_fit(split_rent)

last_xgb_fit %>% collect_metrics()
```

```
# A tibble: 2 x 4
  .metric .estimator .estimate .config
  <chr>   <chr>          <dbl> <chr>
1 rmse    standard     543.    Preprocessor1_Model1
2 rsq     standard       0.723 Preprocessor1_Model1
```

In Exercise 04, where we performed the same regression task with a decision tree, the OOS performance was substantially worse:

Metric	Estimate
RMSE	622
R^2	0.616

We can, therefore, conclude that the XGBoost model is more suitable for estimating the base rent for rental apartments in Munich.

6.2 Exercises

6.2.1 Theoretical exercises

Exercise 6.1. Explain in your own words, the difference between Boosting (Trees), Bagging (Trees), and Random Forests.

Exercise 6.2. On slide 89 in the lecture notes, the AdaBoost algorithm stated as follows.

Algorithmus 1: AdaBoost for binary classification

Input: Let $(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_n, y_n)$ with $y_i \in \{-1; 1\}$ and $w_1(i) = 1/n$.

1 For $b \leftarrow 1$ **to** B draw a bootstrap sample of size n, so that $(\boldsymbol{x}_i, y_i)$ is picked up with prob. $w_b(i) \rightsquigarrow$ weighted sample

2 Train a classifier f_b using the bootstrapped sample

3 Set $err_b = \sum_{i=1}^{n} w_b(i) I(y_i \neq f_b(\boldsymbol{x}_i))$ and $\alpha_b = \frac{1}{2} \ln\left(\frac{1-err_b}{err_b}\right)$.

4 Scale $\hat{w}_{b+1}(i) = \hat{w}_b(i) e^{-\alpha_b y_i f_b(\boldsymbol{x}_i)}$ and set $w_{b+1}(i) = \frac{\hat{w}_{b+1}(i)}{\sum_{j=1}^{n} \hat{w}_{b+1}(j)}$

Output:

$$f_{boost}(\boldsymbol{x}) = sign\left(\sum_{b=1}^{B} \alpha_b f_b(\boldsymbol{x})\right)$$

Figure 6.1: AdaBoost Algorithm

In the third line of Figure 6.1 , the scaling coefficients α_b at step $b \in \{1, .., B\}$ are set to

$$\alpha_b = \frac{1}{2} \log \left(\frac{1 - \text{err}_b}{\text{err}_b} \right). \tag{6.1}$$

The goal of this exercise is to figure out, why the scaling coefficients are defined that way. The essence of this derivation lies in the more general idea of boosting, where the minimization problem at step $b \in \{1, ..., B\}$ is given by (cf. Slide 91)

$$(\alpha_b, h_b) = \underset{\alpha>0, h \in \mathcal{H}}{\text{argmin}} \sum_{i=1}^{n} L(y_i, \hat{f}^{(b-1)}(x_i) + \alpha h(x_i)) \tag{6.2}$$

For AdaBoost, the loss function L is defined by

$$L(y, \hat{f}(x)) = \exp(-y\hat{f}(x)) \tag{6.3}$$

By minimizing (6.4) with respect to α, we obtain the desired coefficient.

1. Show that

$$\underset{\alpha>0, h \in \mathcal{H}}{\text{argmin}} \sum_{i=1}^{n} L(y_i, \hat{f}^{(b-1)}(x_i) + \alpha h(x_i)) = \underset{\alpha>0, h \in \mathcal{H}}{\text{argmin}} \sum_{i=1}^{n} w_b(i) \exp(-\alpha y_i h(x_i)) \tag{6.4}$$

2. Show that the objective function of the right hand side of (6.4) can be expressed as

$$e^{-\alpha} \sum_{y_i = h(x_i)} w_b(i) + e^{\alpha} \sum_{y_i \neq h(x_i)} w_b(i) \tag{6.5}$$

3. Show that (6.5) is equal to

$$(e^{\alpha} - e^{-\alpha}) \sum_{i=1}^{n} w_b(i) I(y_i \neq h(x_i)) + e^{-\alpha} \sum_{i=1}^{n} w_b(i) \tag{6.6}$$

4. Argue by using (6.6) that for any $\alpha > 0$ the solution to (6.2) for h is given by

$$h_b = \underset{h}{\text{argmin}} \sum_{i=1}^{n} w_b(i) I(y_i \neq h(x_i)). \tag{6.7}$$

5. Finally, plug the objective function (6.6) into (6.4) and show that minimizing the loss function for α yields

$$\alpha_b = \frac{1}{2} \log \left(\frac{1 - \text{err}_b}{\text{err}_b} \right), \tag{6.8}$$

where

$$\text{err}_b = \frac{\sum_{i=1}^{n} w_b(i) I(y_i \neq h_b(x_i))}{\sum_{i=1}^{n} w_b(i)}. \tag{6.9}$$

Hint: You can assume that the candidate for α is indeed a minimizer.

6.2.2 Programming Exercises

The following exercise is similar to Exercise 5.3.2. However, instead of fitting penalized logistic regression and classification tree, we fit a XGBoost and LightGBM model on the credit card data.

```
library("finetune")
library("bonsai")
library("patchwork")
library("ggtext")
```

The dataset we will consider in this exercise will be the Credit Card Customers data set that we already used in previous exercises. You can either download it again using the provided link or the button below.

Download BankChurners

Recall that the data set consists of 10,127 entries that represent individual customers of a bank including but not limited to their age, salary, credit card limit, and credit card category.

The goal is to find out whether a customer will stay or leave the bank given the above features.

The following training, validation and test split should be used for training the models of the subsequent exercises.

```
credit_info <- read.csv("data/BankChurners.csv")

set.seed(121)
split <- initial_split(credit_info, strata = Attrition_Flag)
data_train_ci <- training(split)
```

```
data_val_ci <- validation_split(data_train_ci)
data_test_ci <- testing(split)
```

Preprocessing of the data is handled by the following recipe.

```
levels_income <- c("Less than $40K","$40K - $60K",
                   "$60K - $80K","$80K - $120K","$120K +")

levels_education <- c("Uneducated", "High School","College",
                      "Graduate",  "Post-Graduate", "Doctorate")

rec_ci <- recipe(Attrition_Flag ~., data = data_train_ci) %>%
  update_role(CLIENTNUM, new_role = "ID") %>%
  step_mutate_at(all_nominal_predictors(),
                 fn = ~if_else(.%in% c("Unknown","unknown"),NA,.)
  ) %>%
  step_mutate(Attrition_Flag = factor(
                  Attrition_Flag,
                  labels = c("Positive", "Negative")
                )
                ) %>%
  step_string2factor(Income_Category,
                     levels = levels_income,
                     ordered = TRUE) %>%
  step_string2factor(Education_Level,
                     levels = levels_education,
                     ordered = TRUE) %>%
  step_ordinalscore(all_ordered_predictors()) %>%
  step_unknown(all_factor_predictors()) %>%
  step_impute_knn(all_predictors()) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_corr(all_predictors())

ci_wf <- workflow() %>%
  add_recipe(rec_ci)

multi_metrics <- metric_set(roc_auc,pr_auc,accuracy,recall)
```

Note, that we encoded the target variable `Attrition_Flag` with new labels, namely `Positive` and `Negative`. `Positive` corresponds to a customer leaving the bank, while `Negative` corresponds to a customer staying with the bank.

Exercise 6.3. Create and train a random forest model using the provided recipe with 1000 trees and tune the parameters `mtry` and `min_n`.

Tune the model on a grid of size `20` using the `tune_grid` function on the validation split generated with the training data.

Find the best model by evaluating the tuning results with respect to the models' accuracy.

Based on these parameters train a model on the whole training data.

Exercise 6.4. Create two tibbles containing the data necessary to plot a ROC- and PR curve. When creating the tibbles, add a column containing the model name `"Random forest"`, so that we can correctly identify the models later during model evaluation.

Exercise 6.5. Tune a XGBoost model in the same fashion as the random forest. Set the number of trees to `1000`, and every other parameter, except for `sample_size`, to `tune()`.

After tuning and refitting the best model on the whole training data, repeat Exercise 6.4 for this XGBoost model on the test data.

> **Note**
>
> The following model is not relevant for the exam. **However**, it is extremely relevant in today's ML landscape, so I encourage you to solve the following exercises as well.

Exercise 6.6 (Bonus Exercise)**.** The last model we want to train is called LightGBM. It was developed by Microsoft and is, as well as XGBoost, a gradient-based ensemble learner. An advantage compared to XGBoost is the focus on performance and scalability, meaning that it is designed to work well on CPUs while trying to at least match the performance of XGBoost.

The steps for training a LightGBM model are exactly the same as for training an XGBoost model, except for the model specification. Here we set the engine to `"lightgbm"` instead of `"xgboost"`. Every other parameter stays the same, thanks to the `{tidymodels}` framework.

Repeat Exercise 6.5 for a LightGBM model.

> **Tip**
>
> If you get stuck recreating the following plots, revisit the solutions to Exercise Sheet 05, where we created the same plot for a penalized logistic regression, a classification tree, and a random forest.

Exercise 6.7. Create a plot showing the ROC- and PR-curve for each of the models we trained in the previous exercises (Random Forest, XGBoost, LightGBM). Compare the performances visually and decide which model performed the best. For reference, you can find what such a plot could look like below.

Exercise 6.8. For each of the previously trained models (Random Forest, XGBoost, LightGBM), create a confusion matrix based on the test sets to evaluate which model performed best on unseen data.

Exercise 6.9. For the confusion matrices above, find out which model has the overall best out-of-sample performance. For this best model, calculate the following metrics:

1. Sensitivity
2. Precision
3. Accuracy

6.3 Solutions

Solution 6.1 (Exercise 6.1).

1. Bagging (bootstrap aggregation) is a special case of random forests. Here, we also create a predetermined number of trees. However, the main difference is that in Bagging the full set of features is considered when creating a split for a node. In a random forest, only a subset of all features is *randomly* considered when creating a split for a new node.
2. Boosting (Trees) combines many weak learners, e.g., tree stumps, to make a prediction. Compared to Bagging and Random forests, those weak learners are weighted, e.g., one tree stump has more say than another when making a final decision. Furthermore, weak learners are not created independently because each weak learner is built by considering the previous learners' mistakes.

Solution 6.2 (Exercise 6.2).

1. Plugging (6.3) into (6.2) yields

$$\begin{aligned}
\underset{\alpha>0, h\in\mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^{n} L(y_i, \hat{f}^{(b-1)}(x_i) + \alpha h(x_i)) &= \underset{\alpha>0, h\in\mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^{n} \exp(-y_i(\hat{f}^{(b-1)}(x_i) + \alpha h(x_i))) \\
&= \underset{\alpha>0, h\in\mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^{n} \underbrace{\exp(-y_i \hat{f}^{(b-1)}(x_i))}_{:=w_b(i)} \exp(-\alpha y_i h(x_i)) \\
&= \underset{\alpha>0, h\in\mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^{n} w_b(i) \exp(-\alpha y_i h(x_i)).
\end{aligned}$$

2. Since $y_i \in \{1, -1\}$ and $h(x_i) \in \{-1, 1\}$ as well, either $y_i \cdot h(x_i) = 1$ if $y_i = h(x_i)$, or $y_i \cdot h(x_i) = -1$ if $y_i \neq h(x_i)$ (since one of the two terms is equal to -1 and the other equal to 1). The condition above can be formalized as

$$y_i \cdot h(x_i) = \begin{cases} 1 & \text{if } y_i = h(x_i) \\ -1 & \text{if } y_i \neq h(x_i) \end{cases}, \tag{6.10}$$

and rewriting the right hand side of (6.4) using these conditions yields

$$\begin{aligned}
\sum_{i=1}^{n} w_b(i) \exp(-\alpha y_i h(x_i)) &= \sum_{i=1}^{n} (I(y_i = h(x_i)) + I(y_i \neq h(x_i)) w_b(i) \exp(-\alpha y_i h(x_i)) \\
&= \sum_{i=1}^{n} I(y_i = h(x_i)) w_b(i) \exp(-\alpha y_i h(x_i)) \\
&\quad + \sum_{i=1}^{n} I(y_i \neq h(x_i)) w_b(i) \exp(-\alpha y_i h(x_i)) \\
&= \sum_{y_i = h(x_i)} w_b(i) \exp(-\alpha \cdot 1) + \sum_{y_i \neq h(x_i)} w_b(i) \exp(-\alpha \cdot -1) \\
&= e^{-\alpha} \sum_{y_i = h(x_i)} w_b(i) + e^{\alpha} \sum_{y_i \neq h(x_i)} w_b(i)
\end{aligned}$$

3. By expanding and rearranging (6.6), we obtain

$$e^{\alpha} \sum_{i=1}^{n} w_b(i) I(y_i \neq h(x_i)) + e^{-\alpha} \sum_{i=1}^{n} w_b(x_i) - w_b(i) I(y_i \neq h(x_i)) \tag{6.11}$$

$$= e^{\alpha} \sum_{y_i \neq h(x_i)}^{n} w_b(i) + e^{-\alpha} \sum_{I(y_i = h(x_i))}^{n} w_b(x_i). \tag{6.12}$$

4. Using the results of sub tasks 1)-3), we can rewrite the minimization problem of (6.2) as follows:

$$\underset{\alpha>0, h\in\mathcal{H}}{\operatorname{argmin}} \left\{ (e^{\alpha} - e^{-\alpha}) \sum_{i=1}^{n} w_b(i) I(y_i \neq h(x_i)) + e^{-\alpha} \sum_{i=1}^{n} w_b(i) \right\} \tag{6.13}$$

(6.13) only contains one term that depends h, i.e.

$$(e^{\alpha} - e^{-\alpha}) \sum_{i=1}^{n} w_b(i) I(y_i \neq h(x_i)). \tag{6.14}$$

Therefore, any function h that minimizes (6.7) also minimizes (6.13).

5. To minimize (6.6) with respect to α, we have to set the derivative of (6.6) with respect to α to 0. Define $t := \sum_{i=1}^{n} w_b(i) I(y_i \neq h(x_i))$ and $s := \sum_{i=1}^{n} w_b(i)$. Then,

$$\frac{\partial}{\partial \alpha}(e^{\alpha} - e^{-\alpha})t + e^{-\alpha}s) = te^{\alpha} - (s-t)e^{-\alpha}.$$

Now,

$$\begin{aligned} \frac{\partial}{\partial \alpha}((e^{\alpha} - e^{-\alpha})t + e^{-\alpha}s) = te^{\alpha} - (s-t)e^{-\alpha} \stackrel{!}{=} 0 & \\ \iff te^{\alpha} &= (s-t)e^{-\alpha} \\ \iff e^{2\alpha} &= \frac{(s-t)}{t} \\ \iff 2\alpha &= \log\left(\frac{(s-t)}{t}\right) \\ \iff \alpha &= \frac{1}{2}\log\left(\frac{(s-t)}{t}\right). \end{aligned}$$

Defining

$$\text{err}_b := \frac{\sum_{i=1}^{n} w_b(i) I(y_i \neq h(x_i))}{\sum_{i=1}^{n} w_b(i)} \tag{6.15}$$

yields

$$\frac{1-\text{err}_b}{\text{err}_b} = \frac{1}{\text{err}_b} - 1 = \frac{\sum_{i=1}^{n} w_b(i)}{\sum_{i=1}^{n} w_b(i) I(y_i \neq h(x_i))} - 1 = \frac{s}{t} - 1 = \frac{s-t}{t}. \tag{6.16}$$

Finally, re-substituting s and t in $\frac{1}{2}\log\left(\frac{(s-t)}{t}\right)$ yields

$$\alpha = \frac{1}{2}\log\left(\frac{1-\text{err}_b}{\text{err}_b}\right). \tag{6.17}$$

Solution 6.3 (Exercise 6.3).

```
set.seed(121)

cores <- parallel::detectCores()

rf_model <- rand_forest(
  mode = "classification",
  mtry = tune(),
  min_n = tune(),
  trees = 1000
) %>%
  set_engine("ranger",
             num.threads = cores
  )

ci_wf <- ci_wf %>% add_model(rf_model)

rf_tune_res <- ci_wf %>%
    tune_grid(grid = 20,
              resamples = data_val_ci,
              metrics = multi_metrics
    )
```

```
i Creating pre-processing data to finalize unknown parameter: mtry
```

```
rf_best_parm <- rf_tune_res %>%
  select_best(metric = "accuracy")

last_rf_fit <- ci_wf %>%
  finalize_workflow(rf_best_parm) %>%
  last_fit(split)
```

Solution 6.4 (Exercise 6.4).

```
rf_roc <- last_rf_fit %>%
  collect_predictions() %>%
  roc_curve(Attrition_Flag, .pred_Positive) %>%
  mutate(model = "Random Forest")

rf_pr <- last_rf_fit %>%
  collect_predictions() %>%
  pr_curve(Attrition_Flag, .pred_Positive) %>%
  mutate(model = "Random Forest")
```

Solution 6.5 (Exercise 6.5).

```
set.seed(121)

xgb_model <- boost_tree(
  trees = 1000,
  tree_depth = tune(),
  min_n = tune(),
  mtry = tune(),
  loss_reduction = tune(),
  learn_rate = tune()
) %>%
  set_engine("xgboost") %>%
  set_mode("classification")

ci_wf <- ci_wf %>% update_model(xgb_model)

doParallel::registerDoParallel()

xgb_tune_res <- tune_grid(
  ci_wf,
  resamples = data_val_ci,
  grid = 20,
  metrics = multi_metrics
)
```

```
i Creating pre-processing data to finalize unknown parameter: mtry
```

```
xgb_best_parm <- xgb_tune_res %>% select_best(metric = "accuracy")

last_xgb_fit <- ci_wf %>%
  finalize_workflow(xgb_best_parm) %>%
  last_fit(split)

xgb_roc <-
  last_xgb_fit %>%
  collect_predictions() %>%
  roc_curve(Attrition_Flag, .pred_Positive) %>%
  mutate(model = "XGBoost")

xgb_pr <-
  last_xgb_fit %>%
```

```
  collect_predictions() %>%
  pr_curve(Attrition_Flag, .pred_Positive) %>%
  mutate(model = "XGBoost")
```

Solution 6.6 (Exercise 6.6).

```
set.seed(121)

lightgbm_model <- boost_tree(
  trees = 1000,
  tree_depth = tune(),
  min_n = tune(),
  loss_reduction = tune(),
  mtry = tune(),
  learn_rate = tune()
) %>%
  set_engine("lightgbm") %>%
  set_mode("classification")

ci_wf <- ci_wf %>% update_model(lightgbm_model)

lightgbm_res <- tune_grid(
  ci_wf,
  resamples = data_val_ci,
  grid = 20,
  metrics = multi_metrics
)
```

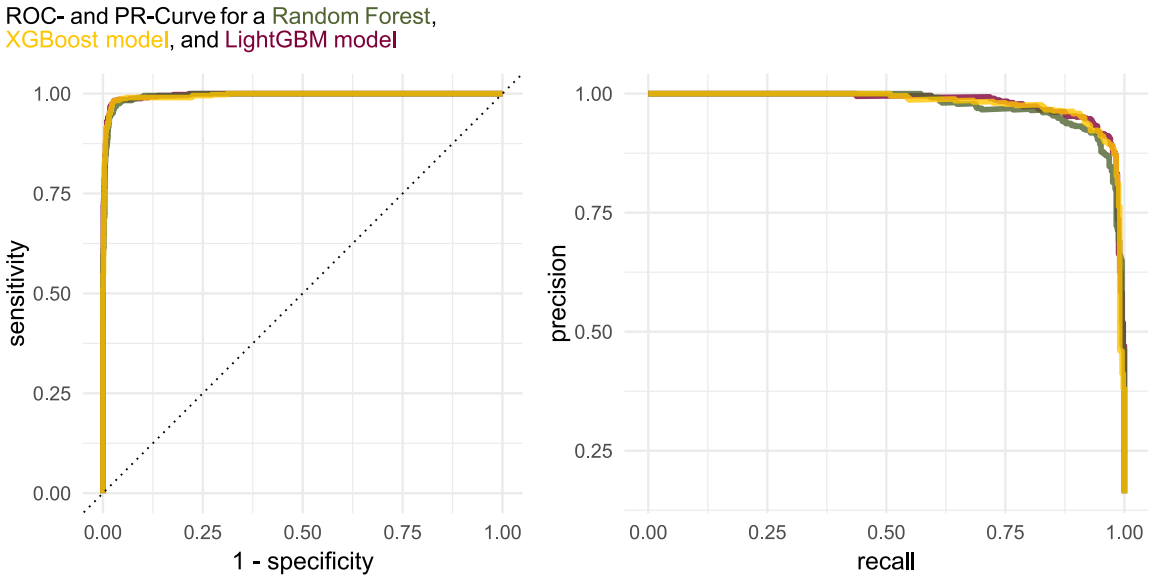
```
i Creating pre-processing data to finalize unknown parameter: mtry
```

```
lightgbm_res_best <- lightgbm_res %>% select_best(metric = "accuracy")

last_lightgbm_fit <- ci_wf %>%
  finalize_workflow(lightgbm_res_best) %>%
  last_fit(split)

lightgbm_roc <- last_lightgbm_fit %>%
  collect_predictions() %>%
  roc_curve(Attrition_Flag, .pred_Positive) %>%
  mutate(model = "lightGBM")
```

```
lightgbm_pr <- last_lightgbm_fit %>%
  collect_predictions() %>%
  pr_curve(Attrition_Flag, .pred_Positive) %>%
  mutate(model = "lightGBM")
```

Solution 6.7 (Exercise 6.7).

```
cols <- c("#80003A","#506432","#FFC500")
names(cols) <- c("lgbm", "rf", "xgb")
plot_title <- glue::glue(
  "ROC- and PR-Curve for a <span style='color:{cols['rf']};'>Random Forest</span>,<br>
   <span style='color:{cols['xgb']};'>XGBoost model</span>,
   and <span style='color:{cols['lgbm']};'>LightGBM model</span>"
  )
p1 <- bind_rows(rf_roc, xgb_roc, lightgbm_roc) %>%
  ggplot(aes(x = 1 - specificity, y = sensitivity, col = model)) +
  geom_path(lwd = 1.5, alpha = 0.8) +
  geom_abline(lty = 3) +
  coord_equal() +
  scale_color_manual(values = unname(cols))+
  theme_minimal(base_size = 14)+
  theme(legend.position = "none")

p2 <- bind_rows(rf_pr, xgb_pr, lightgbm_pr) %>%
  ggplot(aes(x = recall, y = precision, col = model)) +
  geom_path(lwd = 1.5, alpha = 0.8) +
  coord_equal() +
  scale_color_manual(values = unname(cols))+
  theme_minimal(base_size = 14)+
  theme(legend.position = "none")

(p1|p2) +
  plot_annotation(
  title = plot_title,
  theme = theme(plot.title = element_markdown()))
```

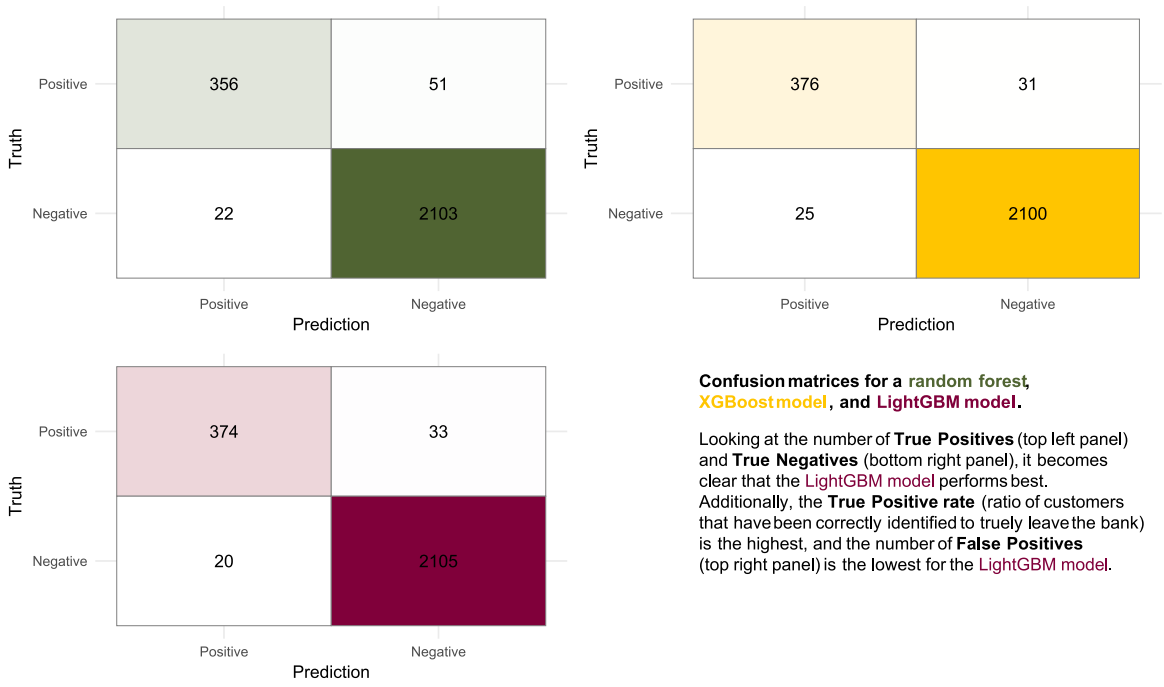


Solution 6.8 (Exercise 6.8).

```
cols <- c("#80003A","#506432","#FFC500")
names(cols) <- c("lgbm", "rf", "xgb")

title_tib <- tibble(
  x=0,
  y=1,
  label = glue::glue(
  "<p><b>Confusion matrices for a
  <span style='color:{cols['rf']};'>random forest</span>, <br/>
  <span style='color:{cols['xgb']};'>XGBoost model</span>,
  and <span style='color:{cols['lgbm']};'>LightGBM model</span>.</b></p>
  <p> Looking at the number of <b>True Positives </b>(top left panel) <br/>
  and <b>True Negatives</b> (bottom right panel), it becomes <br />
  clear that the
  <span style='color:{cols['lgbm']};'>LightGBM model</span> performs best.<br />
  Additionally, the <b>True Positive rate</b> (ratio of customers <br />
  that have been correctly identified to truely leave the bank)<br />
  is the highest, and the number of <b>False Positives</b> <br />
  (top right panel) is the lowest for the
  <span style='color:{cols['lgbm']};'>LightGBM model</span>.</p>"
    )
```

```
)

cm_plot <- function(last_fit_model,high){
  cm <- last_fit_model %>%
    collect_predictions() %>%
    conf_mat(Attrition_Flag, .pred_class)

  cm_tib <- as_tibble(cm$table)%>% mutate(
    Prediction = factor(Prediction),
    Truth = factor(Truth),
    Prediction = factor(Prediction,
                        levels = rev(levels(Prediction)))
  )

  cm_tib %>% ggplot(aes(x = Prediction, y = Truth,fill = n)) +
    geom_tile( colour = "gray50")+
    geom_text(aes(label = n))+
    scale_fill_gradient(low = "white", high = high)+
    theme_minimal()+
    theme(legend.position = "none")
}

# Random Forest
cm1<- cm_plot(last_rf_fit,"#506432")

# XGBoost
cm2<- cm_plot(last_xgb_fit,"#FFC500")

# LightGBM
cm3 <- cm_plot(last_lightgbm_fit,"#80003A")

title_pane <- ggplot()+
  geom_richtext(
    data = title_tib,
    aes(x, y, label = label),
    hjust = 0, vjust = 1,
    label.color = NA
  ) +
  xlim(0, 1) + ylim(0, 1)+
  theme_void()

cm1+cm2+cm3+title_pane+
```

```
plot_layout(ncol =2, widths = c(1,1.04))
```



Solution 6.9 (Exercise 6.9). According to the confusion matrices, ROC-, and PR-Curve the LightGBM model performs best.

1. Sensitivity:
$$\frac{\text{TP}}{\text{P}} = \frac{374}{374+33} = 0.9189189$$
2. Precision:
$$\frac{\text{TP}}{\text{PP}} = \frac{374}{374+20} = 0.9492386$$
3. Accuracy:
$$\frac{\text{TP}+\text{TN}}{\text{P}+\text{N}} = \frac{374+2105}{374+2105+20+33} = 0.9790679$$

7 Maximum Margin Classifier

7.1 Introduction

Linear maximum margin classifiers which are also known as linear support vector machines allow us to classify binary data using a geometric approach. In two dimensions, we can use a simple line that separates the classes (under the assumption that they are indeed separable) and additionally maximizes the margins between those classes. In higher dimensions, the line is replaced by a hyperplane.

The goal of this chapter is to gain an intuition about how such a line can be derived visually and analytically.

Throughout the exercises, the following libraries are used for creating figures:

```
library("tidyverse")
library("ggtext")
```

7.2 Exercises

Exercise 7.1 (Visual derivation)**.** To solve the first exercise, you can either draw everything in a hand sketched figure, or create your own figures with the `{ggplot}` library. Consider the following data set comprised of ten data points and two classes with labels `-1` and `1`.

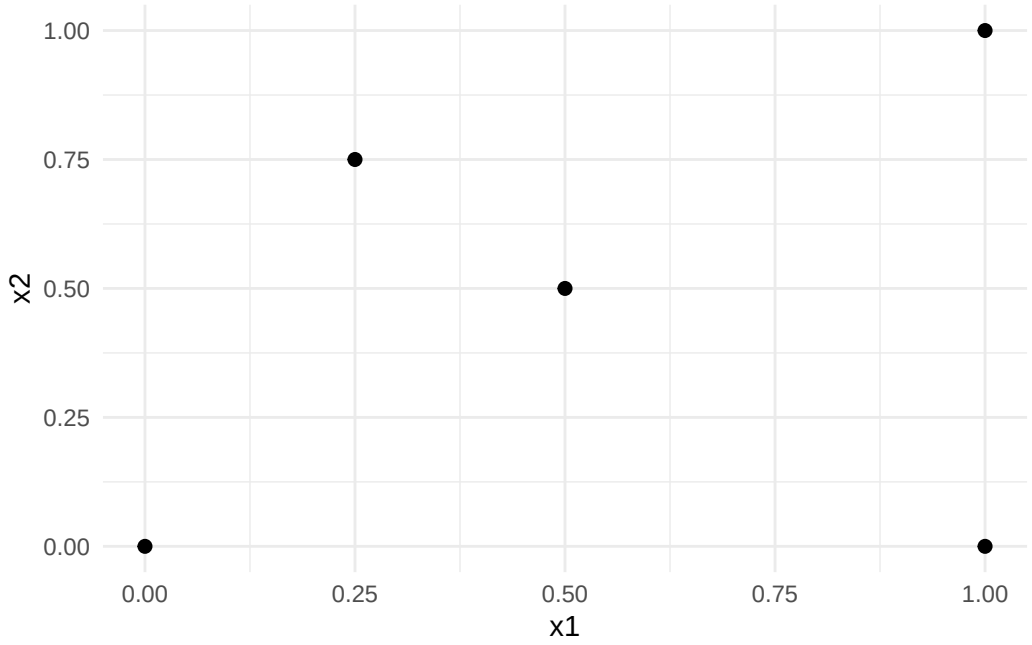
```
data <- tibble(x = c(-1,-1,0,0,1,2,2,2.5,3,4),
               y = c(-1,1,1,0,0,2.5,2,3,2.5,4),
               label = factor(c(rep(1,5),rep(-1,5)))
               )
```

1. Generate a scatter plot visualizing the data points and their respective classes. *Hint: You can visualize the classes using colors or shapes.*
2. To find the optimal separation line geometrically, it is often useful to consider the convex hull of the dataset. Start out by drawing the convex hull in the figure generated in Exercise 1.

A note on convex hulls

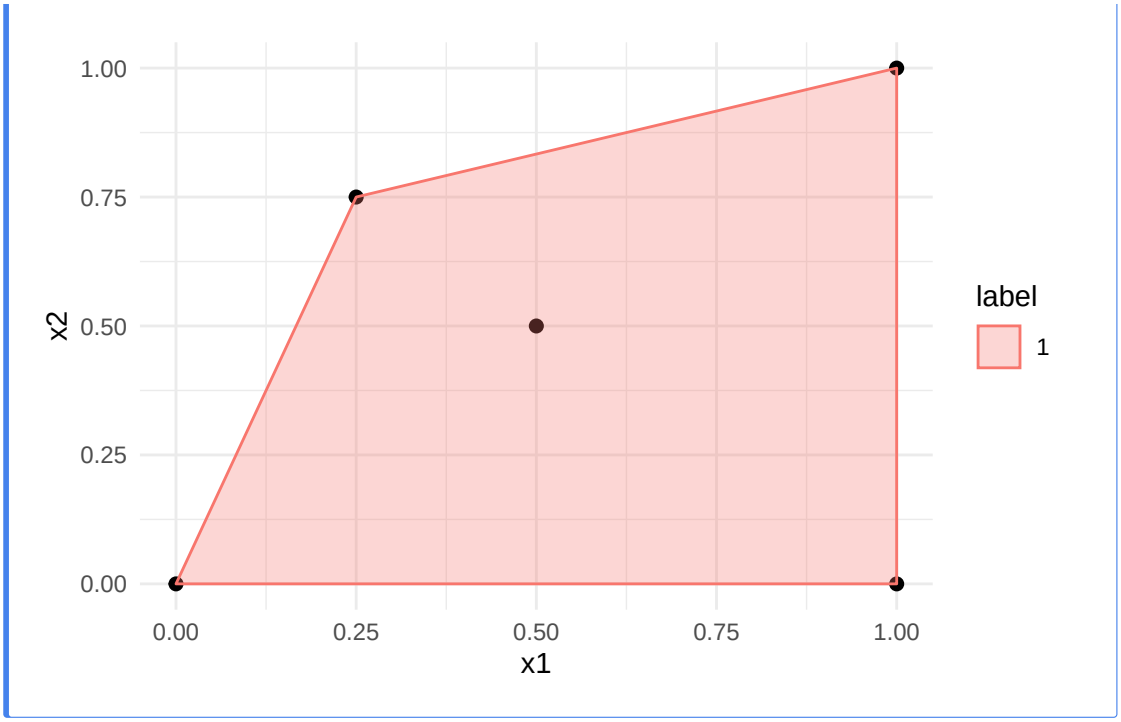
Recall, that the convex hull of a set of points X is defined as the minimal convex set containing X. To create a convex hull with ggplot, consider the following example:
Example: Let $X = \{(0,0)^\top, (0.25, 0.75)^\top, (0.5, 0.5)^\top, (1,1)^\top, (1,0)^\top\}$.

```
data_example <- tibble(x1 = c(0,0.25,0.5,1,1),
                       x2 = c(0,0.75,0.5,0,1),
                       label = factor(rep(1,5))
                       )
p_example <- data_example %>% ggplot(aes(x=x1,y=x2)) +
  geom_point(size = 2)+
  theme_minimal()
p_example
```



Then, the convex hull can be generated as follows:

```
hull_example <- data_example %>%
  group_by(label) %>%
  slice(chull(x1,x2))

p_example +
  geom_polygon(data = hull_example,
               aes(x=x1, y=x2,color = label, fill = label),
               alpha = 0.3)
```



3. The line with the maximal margin is defined by the line with minimal distance between the points of the different classes. Find these two points on the convex hulls you have just drawn/plotted and label them with c_1 and c_{-1} (for the classes with label 1 and -1, respectively). Note that **every** point on a convex hull is a possible candidate, and they **do not** necessarily need to correspond with the data points.

4. Connect the points c_1 and c_{-1} with a line perpendicular to the points.

5. The separation line passes through the center of the line you have just drawn and is orthogonal to it, i.e., the two lines enclose a 90° angle. Draw/plot the separation line s, the line l_1 that passes through the support vectors of the class with label 1, and the line l_{-1} that passes through the support vector of the class with label -1.

6. Add two arbitrary points from each class to the feature space, so that the separation line s found in the previous exercise does not change.

7. Start fresh with the same data and add a new data point x_5 that belongs to a class of your choice so that **the new margin between the two classes is equal to** 1. As before, sketch/plot the convex hull, the two points points $\tilde{c}_1$ and $\tilde{c}_{-1}$ on the convex hull, and the three lines $\tilde{l}_1, \tilde{l}_{-1}$, and $\tilde{s}$ as in Exercise 1.-4.

Exercise 7.2 (Analytical derivation of a maximum margin classifier)**.** From the previous exercise, we know what kind of separation line we can expect when using support vector machines and how to find it graphically. However, determining the separation line analytically

is difficult, even for the simple problem of Exercise 7.1. The number of variables and conditions make this task impractical for a "Pen and Paper" exercise. However, to still get a basic idea of the linear SVM algorithm, we consider an even simpler problem with only two points $x_1, x_2 \in \mathbb{R}^2$ that each belongs to their own class. The two points are given by

$$x_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \in \omega_1 = \{x_i : T_i = 1\},$$
$$x_2 = \begin{pmatrix} 2 \\ 3 \end{pmatrix} \in \omega_{-1} = \{x_i : T_i = -1\}.$$

Since both points are the single representative of their respective classes, they are also the support vectors. Furthermore, they are located on the margin. The goal of this exercise is to find the parameters of a separation line with maximal margin.

Recall from the lecture, that the dual problem is given by

$$L_D = \sum_{i=1}^{2} \alpha_i - \frac{1}{2} \sum_{i=1}^{2} \sum_{j=1}^{2} \alpha_i \alpha_j T_i T_j x_i^\top x_j$$

subject to the constraint

$$\sum_{i=1}^{2} \alpha_i T_i = 0.$$

Technically, we also need the constraint $\alpha_i \geq 0$, $\forall i$. However, to keep things simple, we assume that this is satisfied here.

1. Set up the Lagrange function by plugging all the values into L_D and the constraint above. Subsequently, simplify the terms.

2. To maximize the term L_D under the constraint

$$\sum_{i=1}^{2} \alpha_i T_i = 0,$$

we need an additional Lagrange function with Lagrange multiplier λ

$$\Lambda(\alpha_1, \alpha_2, \lambda) = L_D + \lambda \left(\sum_{i=1}^{2} \alpha_i T_i \right).$$

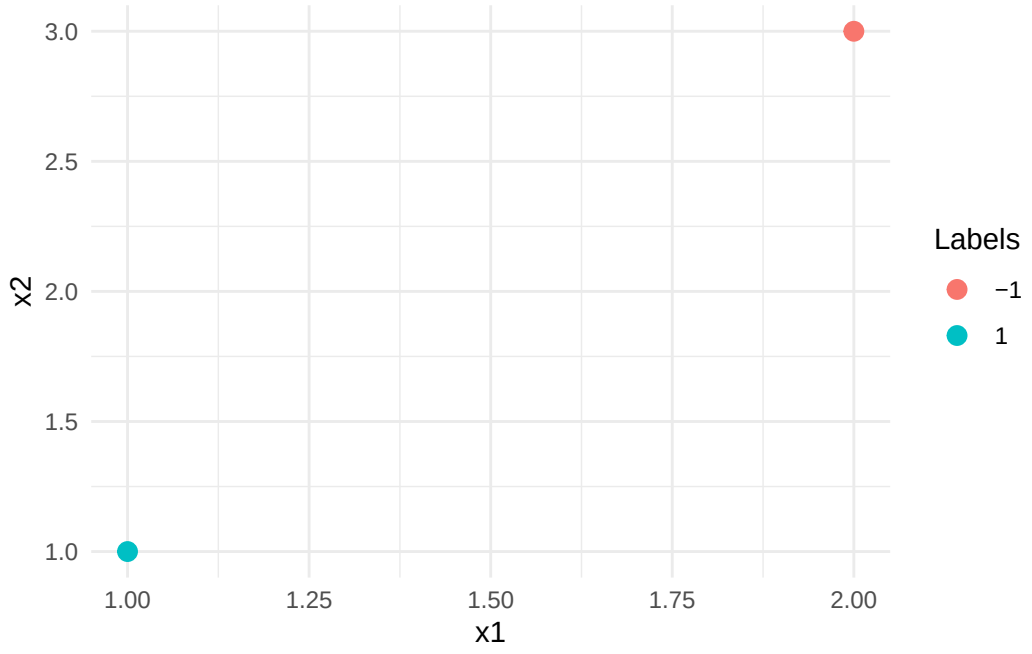
Maximize this function and show that the optimal values are given by

$$\alpha_1^* = \frac{2}{5} \quad \text{and} \quad \alpha_2^* = \frac{2}{5}.$$

It is sufficient to only calculate the potential extrema since we will also assume for them to be actual extrema.

3. Based on the previous results, calculate the line parameters w_1, w_2, and \$ b\$.
4. Add the line to the figure below.

```
data_ex02 <- tibble(x1= c(1,2), x2 = c(1,3), "T" = factor(c(1,-1)))

data_ex02 %>% ggplot(aes(x=x1,y=x2, color = T)) +
  geom_point(size = 3) +
  xlim(1,2) +
  ylim(1,3) +
  labs(
  color = "Labels"
  )+
  theme_minimal()
```



5. Suppose we want to classify a new point $x_3 = (2, 2)^\top$. Assign this point to the correct

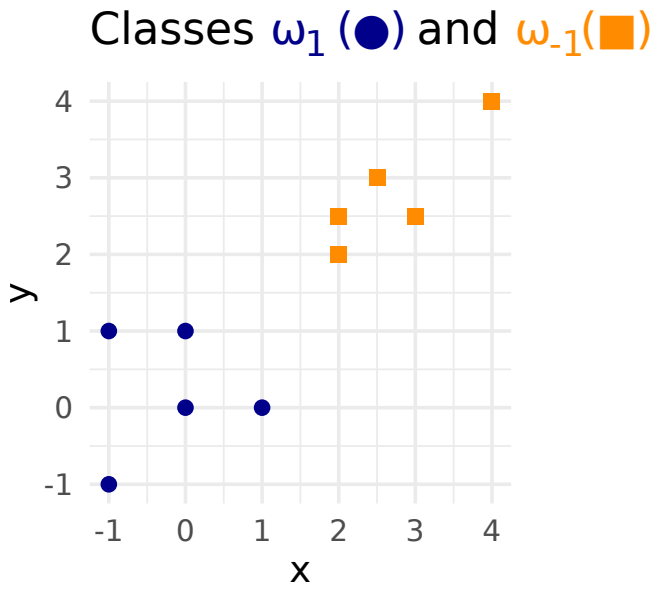
class, both visually and using the decision function

$$f(\tilde{x}) = \text{sign}\left(\sum_{i=1}^{2} \alpha_i^* T_i(x_i^\top \tilde{x} + b)\right).$$

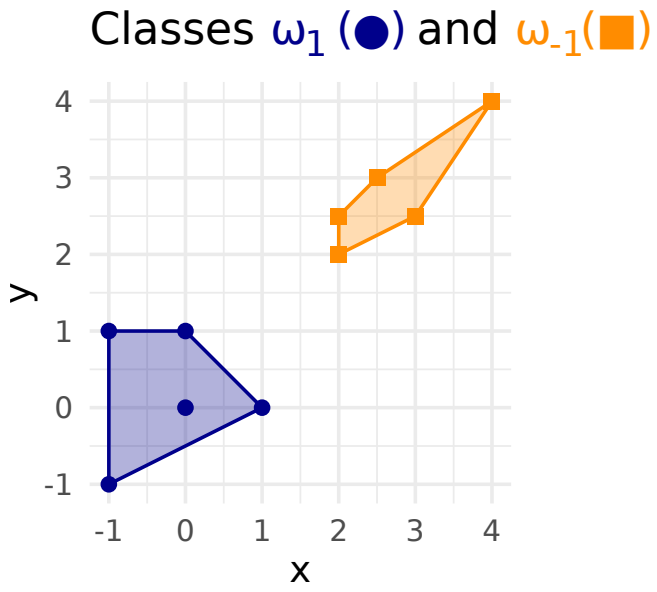
7.3 Solutions

Solution 7.1 (Exercise 7.1).

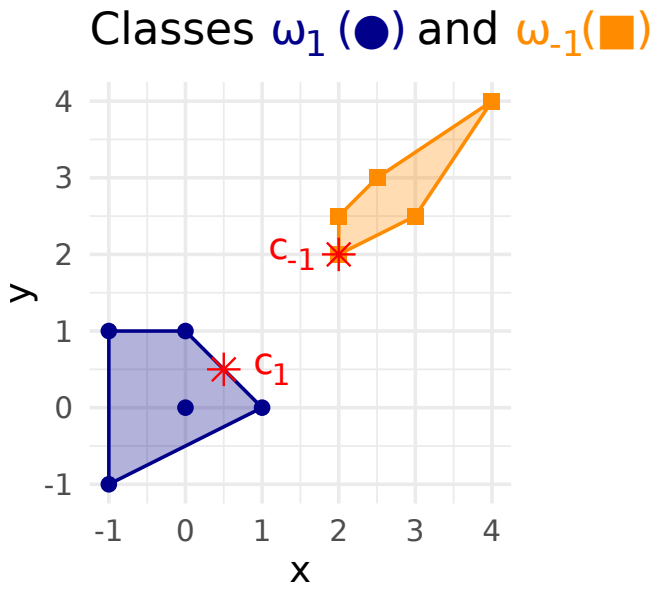
1.
```
cols <- c("1" = "darkblue", "-1" = "darkorange")

title_text <- glue::glue(
"Classes
 <span style='color:{cols['1']};'>
  &omega;<sub>1</sub> (\u25CF)
 </span>
 and
 <span style='color:{cols['-1']};'>
   &omega;<sub>-1</sub>(\u25A0)
 </span>
")

p <- ggplot() +
  geom_point(data = data,
             aes(x=x,y=y,color = label, shape = label),
             size = 2)+
  scale_color_manual(values = cols) +
  scale_shape_manual(values = c(15, 16)) +
  labs(
    title = title_text
  )+
  theme_minimal()+
  theme(
    plot.title = element_markdown(),
    legend.position = "None"
  )+
  coord_fixed()
p
```



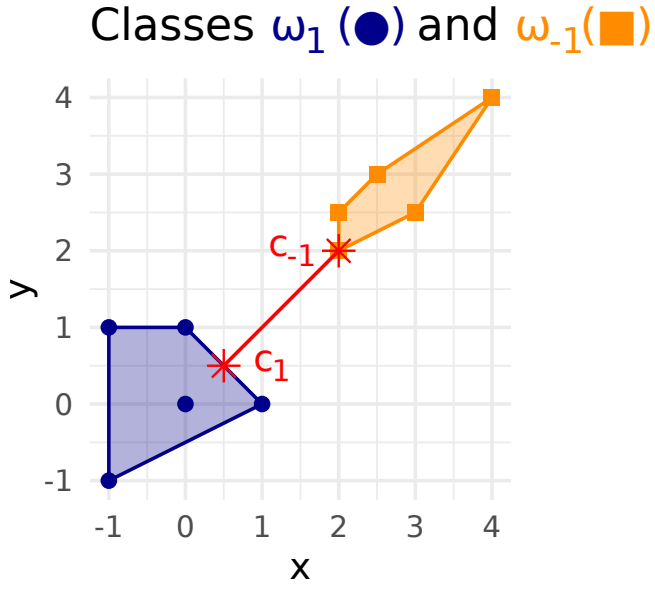
2.
```
hull <- data %>% group_by(label) %>% slice(chull(x,y))
p1 <- p +
    geom_polygon(data = hull,
                 aes(x=x, y=y,color = label, fill = label),
                 alpha = 0.3)+
    scale_fill_manual(values = cols)
p1
```



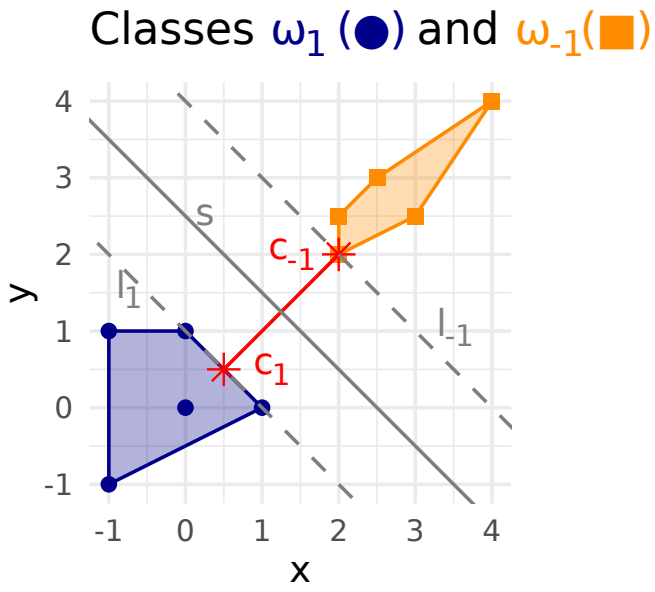
3.
```
df_annotate <- tibble(
  label = c(
    "<span style='color:red;'> c<sub>1</sub></span>",
    "<span style='color:red;'> c<sub>-1</sub></span>"
  ),
  x = c(0.5,2),
  y = c(0.5,2),
  hjust = c(-0.3, 1.2)
)
df_c <- tibble(x = c(0.5,2),
               y = c(0.5,2))

p2 <- p1 + geom_point(data = df_c,
                      aes(x=x,y=y),
                      shape = c(8,8),
                      size = 3,
                      color = c("red","red"))+
  geom_richtext(data = df_annotate,
                      aes(x=x, y=y, label=label, hjust = hjust),
                      fill = NA,
                      label.color = NA)
p2
```



4.
```
df_line <- tibble(x1 = 0.5, x2 = 2, y1 = 0.5, y2 = 2)
p3 <- p2 + geom_segment(data = df_line,
                        aes(x = x1, y = y1, xend = x2, yend = y2),
                        color = "red")
p3
```

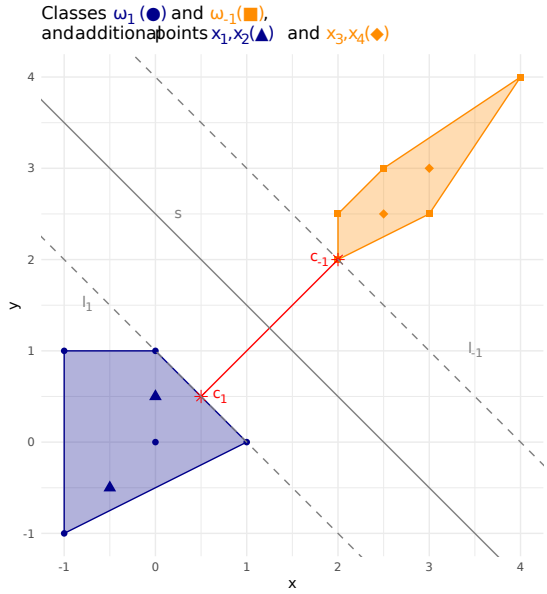


5.
```
df_annotate_l <- tibble(
  label = c(
    "<span style='color:grey50;'> l<sub>1</sub></span>",
    "<span style='color:grey50;'> l<sub>-1</sub></span>",
    "<span style='color:grey50;'> s</span>"
  ),
  x = c(-0.75,3.5,0.25),
  y = c(1.5,1,2.5)
)

p4 <- p3 + geom_abline(slope = -1,
                       intercept = 2.5,
                       color = "grey50")+
           geom_abline(slope = -1,
                       intercept = 1,
                       color = "grey50",
                       linetype = 2)+
           geom_abline(slope = -1,
                       intercept = 4,
                       color = "grey50",
                       linetype = 2)+
           geom_richtext(data = df_annotate_l,
                aes(x=x, y=y, label=label),
                fill = NA,
                label.color = NA)
p4
```



6.
```
title_text <- glue::glue(
"Classes
  <span style='color:{cols['1']};'>
    &omega;<sub>1</sub> (\u25CF)
  </span>
and
  <span style='color:{cols['-1']};'>
    &omega;<sub>-1</sub>(\u25A0)
  </span>,
</br> and additional points
  <span style='color:{cols['1']};'>
    x<sub>1</sub>,x<sub>2</sub>(\u25B2)
  </span>
and
  <span style='color:{cols['-1']};'>
    x<sub>3</sub>,x<sub>4</sub>(\u25C6)
  </span>"
)
df_x <- tibble(x = c(-0.5,0,2.5,3),
               y = c(-0.5,0.5,2.5,3),
               label =factor(c(1,1,-1,-1)))
p5 <- p4 + geom_point(data = df_x,
                      aes(x=x,y=y, color = label),
                      shape = c(17,17,18,18),
```

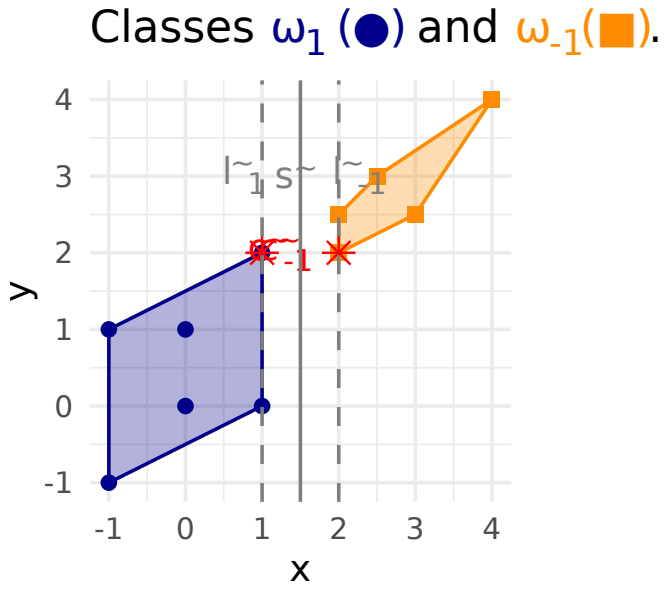
```
                      size = 3) +
  labs(
    title = title_text
  )+
  theme_minimal(base_size = 11)+
  theme(
    plot.title = element_markdown(),
    legend.position = "None"
  )
p5
```



7.
```
title_text <- glue::glue(
"Classes
<span style='color:{cols['1']};'>
  &omega;<sub>1</sub> (\u25CF)
</span>
and
<span style='color:{cols['-1']};'>
  &omega;<sub>-1</sub>(\u25A0)
</span>."
)

#Define new Data and Hull
```

```r
data_new <- rbind(data,c(1,2,1))
hull_new <- data_new %>% group_by(label) %>% slice(chull(x,y))

#New Annotations
## Annotate c
df_annotate_new <- tibble(
  label = c(
    "<span style='color:red;'> c<sup>~</sup><sub>1</sub></span>",
    "<span style='color:red;'> c<sup>~</sup><sub>-1</sub></span>"
  ),
  x = c(0.5,2),
  y = c(2,2),
  hjust = c(-0.3, 1.2)
)
df_c_new <- tibble(x = c(1,2),
             y = c(2,2))
## Annotate l and s
df_annotate_l_new <- tibble(
  label = c(
    "<span style='color:grey50;'> l<sup>~</sup><sub>-1</sub></span>",
    "<span style='color:grey50;'> l<sup>~</sup><sub>1</sub></span>",
    "<span style='color:grey50;'> s<sup>~</sup></span>"
  ),
  x = c(2.25,0.75,1.4),
  y = c(3,3,3)
)

# Generate new plot
p_new <- ggplot() +
  geom_point(data = data_new,
             aes(x=x,y=y,color = label, shape = label),
             size = 2)+
  geom_polygon(data = hull_new,
               aes(x=x, y=y,color = label, fill = label),
               alpha = 0.3)+
  geom_point(data = df_c_new,
               aes(x=x,y=y),
               shape = c(8,8),
               size = 3,
               color = c("red","red"))+
  geom_richtext(data = df_annotate_new,
               aes(x=x, y=y, label=label, hjust = hjust),
```

```
                fill = NA,
                label.color = NA)+
  geom_vline(xintercept = 1.5, color = "grey50")+
  geom_vline(xintercept = 1, color = "grey50", linetype = 2)+
  geom_vline(xintercept = 2, color = "grey50", linetype = 2)+
  geom_richtext(data = df_annotate_l_new,
       aes(x=x, y=y, label=label),
       fill = NA,
       label.color = NA)+
  scale_fill_manual(values = cols)+
  scale_color_manual(values = cols) +
  scale_shape_manual(values = c(15, 16)) +
  labs(
    title = title_text
  )+
  theme_minimal()+
  theme(
    plot.title = element_markdown(),
    legend.position = "None"
  )+
  coord_fixed()

p_new
```



Solution 7.2 (Exercise 7.2).

1. Setting up the Lagrange function:

$$\begin{aligned} L_D &= (\alpha_1 + \alpha_2) - \frac{1}{2}(\alpha_1^2 \cdot 1^2 \cdot 2 + \alpha_1 \cdot \alpha_2 \cdot 1 \cdot -1 \cdot 5 + \alpha_2 \cdot \alpha_1 \cdot -1 \cdot 1 \cdot 5 + \alpha_2^2 \cdot (-1)^2 \cdot 13) \\ &= (\alpha_1 + \alpha_2) - \frac{1}{2}(2 \cdot \alpha_1^2 - 2 \cdot 5 \cdot \alpha_1 \cdot \alpha_2 + 13 \cdot \alpha_2^2) \\ &= \alpha_1 + \alpha_2 - \alpha_1^2 + 5 \cdot \alpha_1 \cdot \alpha_2 - \frac{13}{2}\alpha_2^2 \end{aligned}$$

   Setting up the constraint

$$\alpha_1 - \alpha_2 = 0$$

2. In order to optimize

$$\Lambda(\alpha_1, \alpha_2, \lambda) = \alpha_1 + \alpha_2 - \alpha_1^2 + 5 \cdot \alpha_1 \cdot \alpha_2 - \frac{13}{2} \cdot \alpha_2^2 + \lambda \cdot (\alpha_1 - \alpha_2),$$

   first calculate the partial derivatives with respect to $\alpha_1, \alpha_2, \lambda$:

$$\frac{\partial \Lambda}{\partial \alpha_1} = 1 - 2 \cdot \alpha_1 + 5 \cdot \alpha_2 + \lambda \tag{1}$$
$$\frac{\partial \Lambda}{\partial \alpha_2} = 1 + 5 \cdot \alpha_1 - 13 \cdot \alpha_2 - \lambda \tag{2}$$
$$\frac{\partial \Lambda}{\partial \lambda} = \alpha_1 - \alpha_2 \tag{3}$$

   In order to obtain an extrema, we have to set each of the equations above to 0 and solve them for α_1 and α_2.

   Adding the terms (1) and (2), we obtain

$$\begin{aligned} & 1 - 2 \cdot \alpha_1 + 5 \cdot \alpha_2 + \lambda + 1 + 5 \cdot \alpha_1 - 13 \cdot \alpha_2 - \lambda && = 0 \\ \iff & 2 + 3 \cdot \alpha_1 - 8\alpha_2 && = 0 \\ \iff & \alpha_1 && = \frac{8 \cdot \alpha_2 - 2}{3} \end{aligned}$$

   Plugging α_1 into (3) then yields

$$\begin{aligned} & \frac{8 \cdot \alpha_2 - 2}{3} - \alpha_2 && = 0 \\ \iff & \frac{5 \cdot \alpha_2 - 2}{3} && = 0 \\ \iff & \alpha_2 && = \frac{2}{5} \end{aligned}$$

Since we also need to satisfy $\alpha_1 - \alpha_2 = 0$ and thus $\alpha_1 = \alpha_2$, we can deduce $\alpha_1 = \frac{2}{5}$. Therefore $\alpha_1^* = \alpha_2^* = \frac{2}{5}$.

3. The first order conditions require

$$w = \sum_{i=1}^{2} \alpha_i \cdot T_i x_i,$$

i.e.,

$$\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \alpha_1^* \cdot T_1 \cdot x_1 + \alpha_2^* \cdot T_2 \cdot x_2 = \begin{pmatrix} \frac{2}{5} \\ \frac{2}{5} \end{pmatrix} - \begin{pmatrix} \frac{4}{5} \\ \frac{6}{5} \end{pmatrix} = - \begin{pmatrix} \frac{2}{5} \\ \frac{4}{5} \end{pmatrix}$$

Calculating b using the Karush–Kuhn–Tucker conditions yields

$$\alpha_i \left(T_i \left(x_i^\top w + b \right) - 1 \right) = 0 \iff b = \frac{1}{T_i} - x_i^\top w$$

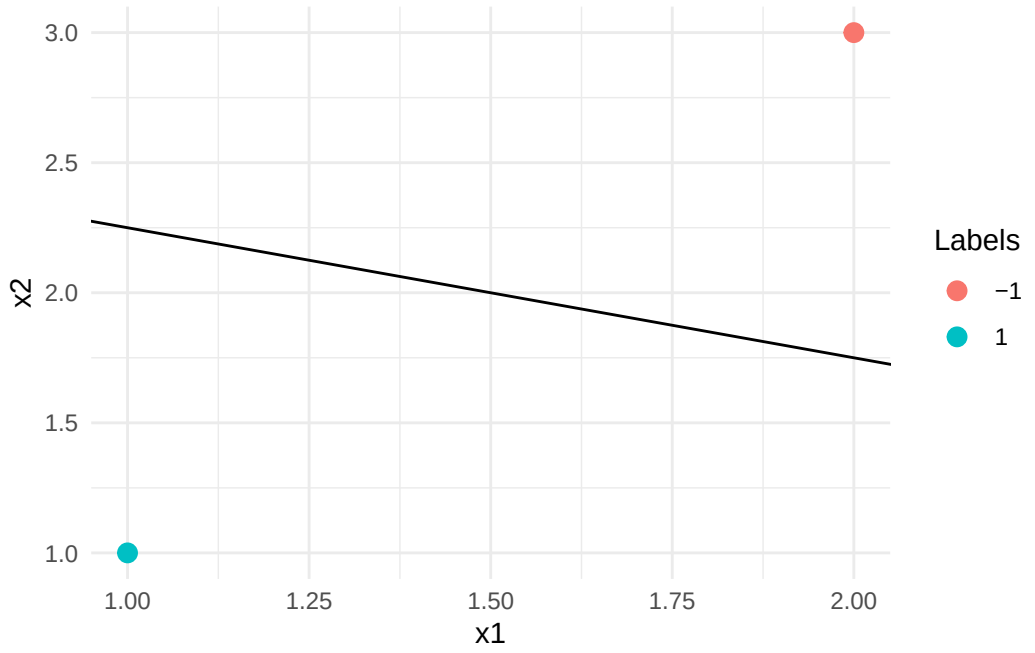
Plugging in the values we obtained for w:

$$b = \frac{1}{1} - \begin{pmatrix} 1 \\ 1 \end{pmatrix}^\top \begin{pmatrix} -\frac{2}{5} \\ -\frac{4}{5} \end{pmatrix} = \frac{1}{-1} - \begin{pmatrix} 2 \\ 3 \end{pmatrix}^\top \begin{pmatrix} -\frac{2}{5} \\ -\frac{4}{5} \end{pmatrix} = \frac{11}{5}$$

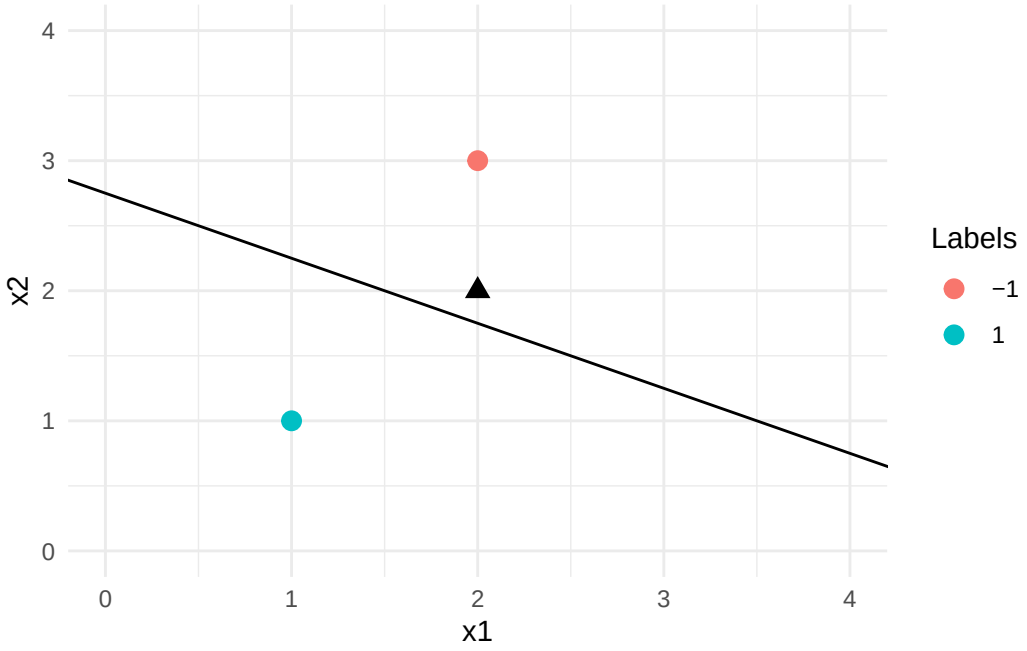
The separation line is therefore give by

$$w_1 \cdot x_1 + w_2 \cdot x_2 + b = 0 \iff x_2 = \frac{-b - w_1 \cdot x_1}{w_2} = \frac{-\frac{11}{5} + \frac{2}{5} \cdot x_1}{-\frac{4}{5}} = \frac{11}{4} - \frac{1}{2} x_1$$

4.
```r
data_ex02 %>% ggplot(aes(x=x1,y=x2, color = T)) +
  geom_point(size = 3) +
  geom_abline( slope = -0.5, intercept = 11/4) +
  xlim(1,2) +
  ylim(1,3) +
  labs(color = "Labels")+
  theme_minimal()
```



5.
```
data_ex02 %>% ggplot() +
  geom_point(aes(x=x1,y=x2, color = T), size = 3) +
  geom_abline( slope = -0.5, intercept = 11/4) +
  geom_point(data = tibble(x=2,y=2), aes(x=x,y=y), size = 3, shape = 17)+
  xlim(0,4) +
  ylim(0,4) +
  labs(color = "Labels")+
  theme_minimal()
```



Since the point $(2, 2)$ is above the decision line, it belongs to class ω_{-1}.

Plugging all the values into the decision function yields:

$$\begin{aligned} f\begin{pmatrix}2\\2\end{pmatrix} &= \text{sign}\left(\frac{2}{5}\cdot 1\cdot\begin{pmatrix}1\\1\end{pmatrix}^{\top}\cdot\begin{pmatrix}2\\2\end{pmatrix}+\frac{11}{5}+\frac{2}{5}\cdot -1\left(\begin{pmatrix}2\\3\end{pmatrix}^{\top}\cdot\begin{pmatrix}2\\2\end{pmatrix}+\frac{11}{5}\right)\right)\\ &= \text{sign}\left(-\frac{12}{5}\right)\\ &= -1. \end{aligned}$$

Therefore, the new point belongs to class ω_{-1}.

8 Support Vector Machines and Stacking

In this exercise session, we discuss how to train different support vector machines (SVMs) and create a stack in R.

SVMs and stacks are available in the `{tidymodels}` framework which speeds setting up a training and testing routine significantly.

8.1 Introduction

We will shortly discuss how to train and tune SVMs on a toy data set in R, before training a stack on the same data.

Libraries used throughout the session:

```
library(tidyverse)
library(tidymodels)
library(finetune)
library(patchwork)
library(ggtext)
library(stacks)
library(kernlab)
```

8.1.1 Support Vector Machines

Recall, solving the the dual problem

$$L_D = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j x_i^\top x_j, \quad \text{s.t.} \quad \alpha_i \geq 0 \text{ for all } i \tag{8.1}$$

yields a linear SVM.

Instead of using a linear SVM enrich our feature space by replacing the term $x_i^\top x_j$ with a function $K : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}$. For polynomials of degree $d \geq 2$, we apply the kernel function

$K(x_i, x_j) = (1 + x_i^\top x_j)^d$. Another popular choice we will consider is the so-called radial basis kernel (RBF) given by

$$K(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{c}\right)^d, \quad c \geq 0. \tag{8.2}$$

The `{tidymodels}` frameworks contains three different SVM model specifications, namely

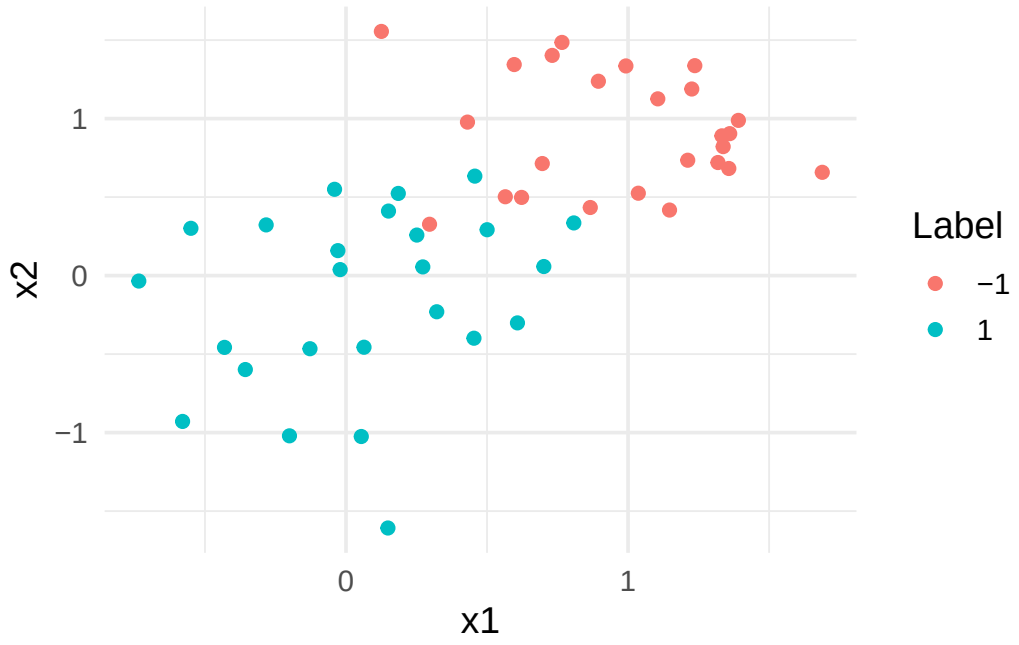
1. `svm_linear`, which specifies a simple linear SVM,
2. `svm_poly`, which specifies a polynomial SVM (including polynomials of degree 1), and
3. `svm_rbf`, which specifies an SVM with RBF Kernel.

Consider the following toy data set:

```
set.seed(121)
data_toy <- tibble(
  x1 = c(
    rnorm(25,0,0.5),
    rnorm(25,1,0.5)
  ),
  x2 = c(
    rnorm(25,0,0.5),
    rnorm(25,1,0.5)
  ),
  y = factor(rep(c(1,-1),each = 25))
)
```

The two class data contains a two dimensional dataset where both axes are generated using a normal distribution. The samples with label `1` are drawn from a $\mathcal{N}((0,0)^\top, 0.5 \cdot ID_2)$ distribution, while the samples with label `-1` are drawn from a $\mathcal{N}((1,1)^\top, 0.5 \cdot ID_2)$ distribution.

```
data_toy %>% ggplot(aes(x=x1,y=x2, color = y)) +
  geom_point(size = 2)+
  labs(color = "Label")+
  theme_minimal(base_size = 14)
```



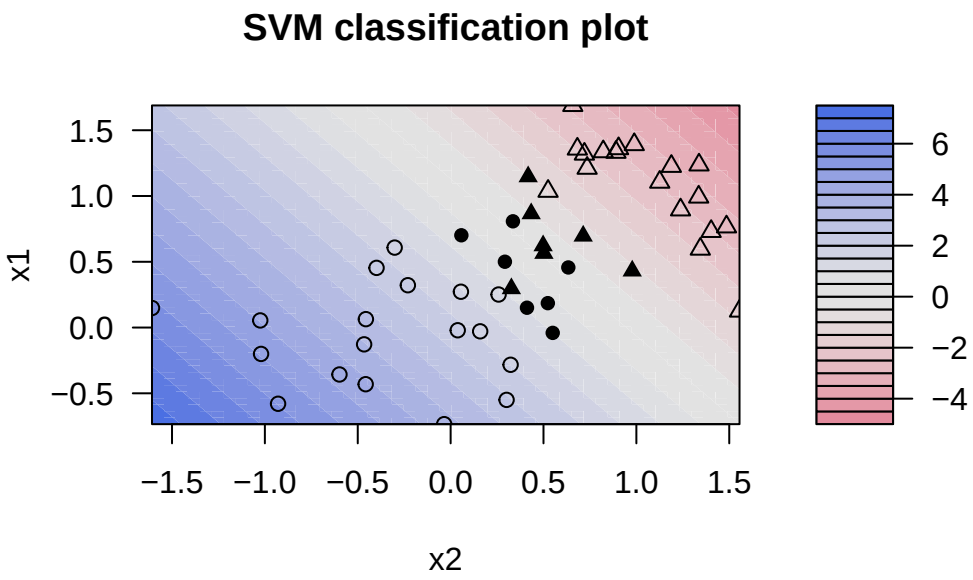
Without creating a workflow or recipe, we now want to train three different SVMs.

Using the `svm_poly` function, we can specify a linear SVM for the toy data by setting the `degree` to `1` and `mode` to `"classification"`. Setting the engine to `"kernlab"` and applying the `fit` function trains the linear SVM.

```
svm_lin_res <- svm_poly(
  mode = "classification",
  degree = 1
) %>%
  set_engine("kernlab") %>%
  fit(y ~ ., data_toy)
```

To plot the decision surfaces, we need to extract the fit engine and pass it to the `kernlabs::plot` function:

```
svm_lin_res %>%
  extract_fit_engine() %>%
  plot(data = data_toy)
```



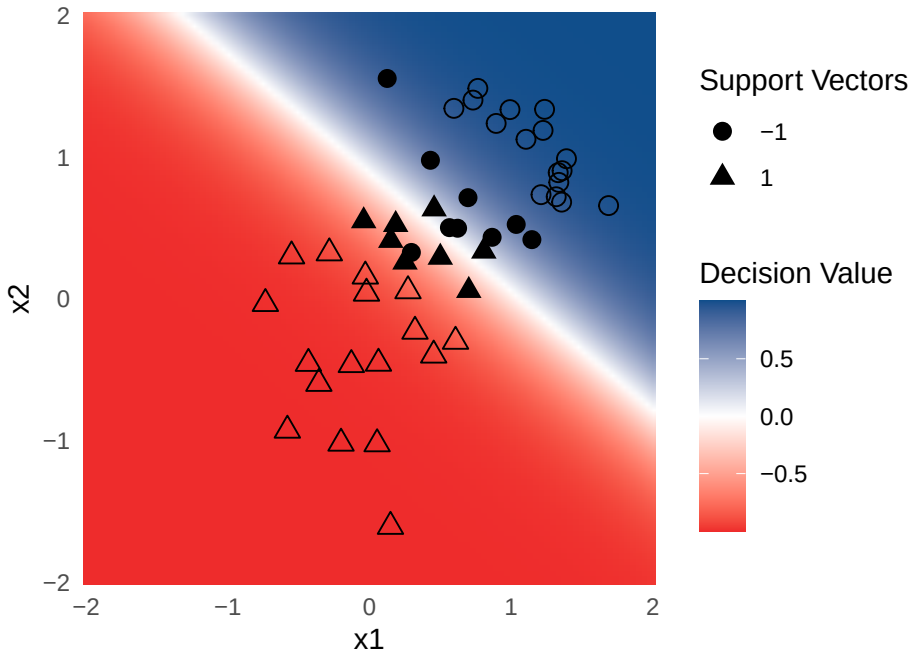
The plot depicts the contour plot of the decision surface of the linear SVM. The boundary between the two classes (where the decision function equals zero) is represented by the transition area in the middle of the plot, where the colors shift from blue to red. The greyed-out area indicates a lower confidence for the decision, meaning that the algorithm is not sure whether the values belong to class -1 or 1. Any point on the blue shape is classified as a member of the class with label `1`, whereas any point located on the red shade is classified as a member of the class with label `-1`. The filled shapes denote support vectors, i.e., vectors that influenced the decision surfaces, while shapes that are only outlined do not represent support vectors. indicate samples with class label `-1` and indicate samples with class label `1`.

Instead of using the `kernlab::plot()` function, you can also visualize the decision surface using ggplot.

However, as it turns out, this is relatively complicated. In case you are still interested, the following snippet generates the decision surface using `ggplot`.

```
library(ggnewscale)

supp_vec <- svm_poly(
  mode = "classification",
  degree = 1
) %>%
  set_engine("kernlab", scaled = c(F,F,T,T)) %>%
  fit(y ~ ., data_toy) %>%
```

```
  extract_fit_engine() %>%
  xmatrix() %>%
  pluck(1) %>%
  as_tibble()


data_toy <- data_toy %>%
  mutate(supp_vec = data_toy$x1 %in% supp_vec$x1)

x_grid <- expand_grid(x1 = seq(-2,2,length.out = 100),
                      x2 = seq(-2,2,length.out = 100)) %>%
  bind_cols(predict(svm_lin_res,.,type="prob")) %>%
  mutate(val = (`.pred_-1`*2)-1)

#transform probabilities to take values between -1,1


x_grid %>% ggplot(aes(x=x1,y=x2))+
  geom_raster(aes(fill= val), interpolate = TRUE)+
  geom_point(data = data_toy %>% filter(supp_vec==F),
             aes(shape = y),
             cex = 3,
             show.legend=FALSE) +
  scale_shape_manual(values = c("-1" = 1,"1" = 2))+
  new_scale("shape")+
  geom_point(data = data_toy %>% filter(supp_vec==T),
             aes(shape = y),
             cex = 3) +
  labs(
    shape = "Support Vectors"
  )+
  coord_fixed(expand = FALSE)+
  scale_fill_gradient2(high = "dodgerblue4",
                       midpoint = 0,
                       low = "firebrick2")+
  labs(fill = "Decision Value")+
  theme_minimal()
```



8.1.2 Stacking

The idea behind linear stacking is relatively simple: Assume there are M different models $\hat{f}_1, ..., \hat{f}_M$, all fitted on the same training data. Then, a target can be predicted by taking a weighted average of all M models, i.e.,

$$\hat{y} = \sum_{m=1}^{M} \omega_m \hat{f}_m(x), \tag{8.3}$$

where $\omega_1, ..., \omega_M \in [0, 1]$ such that $\sum_{m=1}^{M} \omega_m = 1$.

To find out which $\{\omega_m\}_{m=1}^{M}$ yield the best results, we can solve the optimization problem minimizing the loss function

$$\min_{\omega_1,...,\omega_M} L(y, \hat{y}) = \min_{\omega_1,...,\omega_M} \left\{ \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \sum_{m=1}^{M} \omega_m \hat{f}_m(x_i) \right)^2 \right\}. \tag{8.4}$$

The modeling process for a stack with `{tidymodels}` can be described as follows:

1. Define candidates for the stack by fitting (tuned) models to the training data by using the `control_stack_grid` function in the tune or model specification.

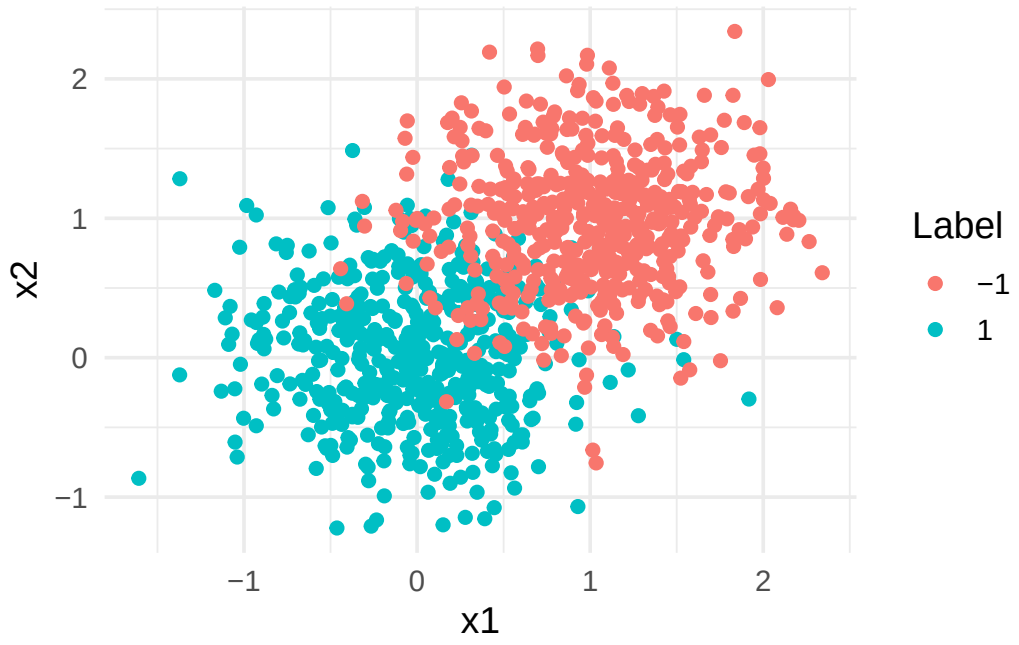
2. Initialize a stack object using the `stacks` function.
3. Add candidate models to the stack using the `add_candidates` function.
4. Pass the `stack` object to the `blend_predictions` function, which specifies how the predictions of each candidate are evaluated.
5. Fit the candidate ensemble with non-zero stacking coefficients using the `fit_members` function.
6. Predict on test data using the `predict` function to evaluate out-of-sample performance.

Consider the following toy example data which is created in a similiar fashion as the data set from the previous section.

```
set.seed(121)

data_toy_big <- tibble(
  x1 = c(
    rnorm(500,0,0.5),
    rnorm(500,1,0.5)
  ),
  x2 = c(
    rnorm(500,0,0.5),
    rnorm(500,1,0.5)
  ),
  y = factor(rep(c(1,-1),each = 500))
)
```

Consider the following figure, displaying the extended data set:

```
data_toy_big %>% ggplot(aes(x=x1,y=x2, color = y)) +
  geom_point(size = 2)+
  labs(color = "Label")+
  theme_minimal(base_size = 14)
```



Considering the figure above, it is quickly becomes evident that the classes are not linearly seperable.

We, therefore, now want to tune and fit different classification models and combine them to a stack predict the class labels as good as possible.

1. First, set up the control grid:

```
ctrl_grid <- control_stack_grid()
```

2. Then, create a data split and tune the candidate models. For our simple example, we will tune a random forest and XGBoost classifier using a simple training, validation and test split. The candidate models for the stack are then given by the models fitted with different hyperparameters. First, we create the data split:

```
set.seed(121)
split_toy <- initial_split(data_toy_big,prop = 4/5)
data_train <- training(split_toy)
data_val <- validation_split(data_train, prop = 4/5)
```

```
Warning: `validation_split()` was deprecated in rsample 1.2.0.
i Please use `initial_validation_split()` instead.
```

```
data_test <- testing(split_toy)
```

Then, we train several random forests by tuning the hyper parameters `min_n` and `trees` :

```
set.seed(121)
rf_model_spec <- rand_forest(
  mode = "classification",
  mtry = 1,
  min_n = tune(),
  trees = tune()
) %>%
  set_engine("ranger")

rec_toy<- recipe(
  y~., data = data_train
)

wf_toy <- workflow() %>%
  add_model(rf_model_spec) %>%
  add_recipe(rec_toy)

rf_tune_res <- wf_toy %>%
  tune_grid(grid = 10,
            resamples = data_val,
            control = ctrl_grid
  )
```

Then, we train a XGBoost model in the same fashion:

```
set.seed(121)
xgb_model_spec <- boost_tree(
  trees = 1000,
  tree_depth = tune(),
  min_n = tune(),
  mtry = 1,
  loss_reduction = tune(),
  learn_rate = tune()
) %>%
  set_engine("xgboost") %>%
  set_mode("classification")

wf_toy <- wf_toy %>% update_model(xgb_model_spec)

xgb_tune_res <- wf_toy %>% tune_grid(
  resamples = data_val,
```

```
  grid = 20,
  control = ctrl_grid
)
```

3. The stack candidates can be added with the `add_candidates()` function:

```
stack_toy <- stacks() %>%
  # add candidate members
  add_candidates(rf_tune_res) %>%
  add_candidates(xgb_tune_res)
```

4. By using the `blend_predictions` function, we can specify, how the predictions should be blended together. The options we are interested in are the following:

   - `mixture`: A number between zero and one (inclusive) specifying the proportion of L1 regularization (i.e. lasso) in the model. `mixture = 1` indicates a pure lasso model, `mixture = 0` indicates ridge regression, and values in the open interval (0,1) specify an elastic net.
   - `penalty`: A vector containing penalty values for the amount of regularization used in member weighting. If more than one value is contained in the vector, the library will tune over the candidte values to select the best penalty.
   - `metric`: The metric(s) to use in tuning the regularization penalty on the stacking coefficients.
   - `times`: Number of bootstrap samples tuned over by the model that determines stacking coefficients.

```
set.seed(121)
stack_toy <- stack_toy %>% blend_predictions(
   mixture = 0.9,
   penalty = 10^(-6:-1),
   metric = metric_set(pr_auc),
   times = 20
 )
```

5. Finally, we can fit the stack by passing the previously specified stack object that contains the candidates and blend specifications to the `fit_members()` function.

```
set.seed(121)
stack_toy <- stack_toy %>%
 fit_members()
```

6. We want to evaluate our stack and in order to do so, we first need to create predictions from our test dataset. The following code snippet creates predictions based on the test set and binds them to the test data columns.

```
stack_pred <-
  data_test %>%
  bind_cols(predict(stack_toy, .))
```

To evaluate the stack results, we can use the typical metrics provided in the {yardstick} library. For example, the accuracy of our stacked model is around 92%.

```
stack_pred %>% accuracy(truth = y, .pred_class)
```

```
# A tibble: 1 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary          0.93
```

Note, to obtain probabilities for the predictions the argument `type = "prob"` has to be passed in the `predict` function.

8.2 Exercises

The goal of this exercise is to create an SVM model with an RBF kernel that serves as a classifier for an underlying dataset.

The dataset we will consider in this exercise will be the Credit Card Customers data set that we already used in previous exercises. You can either download it again using the provided link or the button below.

Download BankChurners

Recall that the data set consists of 10,127 entries that represent individual customers of a bank including but not limited to their age, salary, credit card limit, and credit card category.

The goal is to find out whether a customer will stay or leave the bank given the above features.

The following training, validation and test split should be used for training the models of the subsequent exercises.

```
credit_info <- read.csv("data/BankChurners.csv")

set.seed(121)
split <- initial_split(credit_info, strata = Attrition_Flag)
data_train_ci <- training(split)
```

```
data_val_ci <- validation_split(data_train_ci)
data_test_ci <- testing(split)
```

Preprocessing of the data is handled by the following recipe.

```
levels_income <- c("Less than $40K","$40K - $60K",
                   "$60K - $80K","$80K - $120K","$120K +")

levels_education <- c("Uneducated", "High School","College",
                      "Graduate",  "Post-Graduate", "Doctorate")

rec_ci <- recipe(Attrition_Flag ~., data = data_train_ci) %>%
  update_role(CLIENTNUM, new_role = "ID") %>%
  step_mutate_at(all_nominal_predictors(),
                 fn = ~if_else(.%in% c("Unknown","unknown"),NA,.)
  )%>%
  step_string2factor(Income_Category,
                     levels = levels_income,
                     ordered = TRUE) %>%
  step_string2factor(Education_Level,
                     levels = levels_education,
                     ordered = TRUE) %>%
  step_ordinalscore(all_ordered_predictors()) %>%
  step_unknown(all_factor_predictors()) %>%
  step_impute_knn(all_predictors()) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_corr(all_predictors())

ci_wf <- workflow() %>%
  add_recipe(rec_ci)
```

Note, that compared to the previous session, we did not change the values of the target variable `Attrition_Flag`. By changing the targets name in the recipe, we potentially run into problems when fitting the stack. Therefore, we change the labels once all the predictions have been generated to obtain an easier to understand confusion matrix.

Exercise 8.1. In this exercise, we want to tune and evaluate an SVM. The hyperparameter `cost`, i.e. the parameter D in the maximum margin formulation

$$\max_{\omega,b,\|\omega\|\leq 1} D \quad \text{s.t.} \quad y_i(x_i^\top \omega + b) \geq D, \quad \forall i = 1, ..., n.$$

is the only parameter we wish to tune.

Recall, that the cost parameter penalizes samples that are predicted to be in the wrong class. A larger cost will thus lead to a more flexible model with fewer misclassifications (Bias-Variance tradeoff).

1. Create and tune a SVM model with RBF kernel, where the `cost` parameter is set to tune. Candidates for the optimal `cost` parameter can be created by passing `grid = 20` in the `tune_grid()` function.
2. Evaluate the performance on the test set by plotting a ROC and PR Curve.
3. Create a confusion matrix for the test data and calculate the accuracy, precision and recall.

Exercise 8.2. In this last exercise, we want to create a stack model to predict the target variable `Attrition_Flag`.

The potential stack candidates are the XGBoost and random forest tuning results of Exercise 06.

1. Create a control stack grid that can be added to the tuning specifications.
2. Retune the XGBoost and random forest model and add the candidates to a stack. As grid size for the hyper parameters, choose `30`.
3. Add the candidate models to a stack and blend the predictions. To blend the predictions, use a pure lasso approach and tune the penalty with a grid given by the vector `10^(-6:-2)`. Use the metric `accuracy` to select the best model coefficients.
4. Finally, fit the stack on the whole training data and evaluate it on the test data by generating a confusion matrix and calculating the sensitivity, precision, and accuracy. Before generating the confusion matrix, change the labels of the target feature and prediction to `"Negative"` if the customer is (predicted to be) an existing customer and to `"Positive"` if the customer is (predicted to be) an attrited customer. Is the stacked model better than the LightGBM model that was presented in Session 06?

8.3 Solutions

Solution 8.1 (Exercise 8.1).

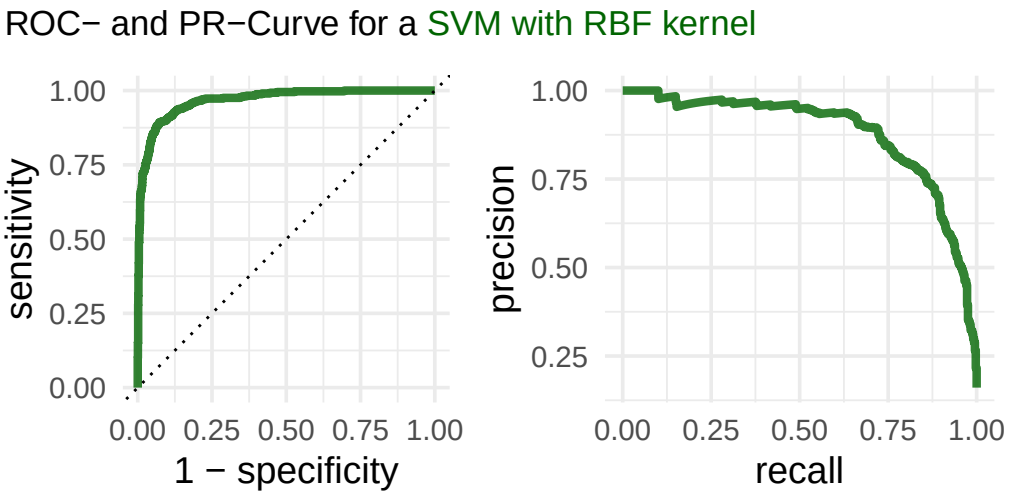
```
svm_model_rbf_tune <- svm_rbf(
  mode = "classification",
  cost = tune()
  ) %>%
  set_engine("kernlab")

ci_wf <- ci_wf %>% add_model(svm_model_rbf_tune)
```

```
svm_rbf_tune_res <- ci_wf %>%
  tune_grid(
    resamples = data_val_ci,
    grid = 20
  )
```

```
svm_tune_best <- svm_rbf_tune_res %>% select_best(metric = "roc_auc")

last_rbf_fit <- ci_wf %>%
  finalize_workflow(svm_tune_best) %>%
  last_fit(split)


svm_rbf_roc_tuned <- last_rbf_fit %>%
  collect_predictions() %>%
  roc_curve(Attrition_Flag, `.pred_Attrited Customer`) %>%
  mutate(model = 'SVM')

svm_rbf_pr_tuned <- last_rbf_fit %>%
  collect_predictions() %>%
  pr_curve(Attrition_Flag,  `.pred_Attrited Customer`) %>%
  mutate(model = 'SVM')
```

```
cols <- c("darkgreen")
names(cols) <- c("svm")
plot_title <- glue::glue(
  "ROC- and PR-Curve for a
  <span style='color:{cols['svm']};'>SVM with RBF kernel</span>"
  )
p1 <- svm_rbf_roc_tuned %>%
  ggplot(aes(x = 1 - specificity, y = sensitivity, col = model)) +
  geom_path(lwd = 1.5, alpha = 0.8) +
  geom_abline(lty = 3) +
  coord_equal() +
```

```
  scale_color_manual(values = unname(cols))+
  theme_minimal(base_size = 14)+
  theme(legend.position = "none")

p2 <- svm_rbf_pr_tuned %>%
  ggplot(aes(x = recall, y = precision, col = model)) +
  geom_path(lwd = 1.5, alpha = 0.8) +
  coord_equal() +
  scale_color_manual(values = unname(cols))+
  theme_minimal(base_size = 14)+
  theme(legend.position = "none")


(p1|p2) +
  plot_annotation(
  title = plot_title,
  theme = theme(plot.title = element_markdown()))
```



Solution 8.2 (Exercise 8.2).

1. `ctrl_grid_ci <- control_stack_grid()`

2. First, we specify the random forest model and add it to the recipe.

```
rf_model <- rand_forest(
  mode = "classification",
  mtry = tune(),
  min_n = tune(),
  trees = 1000
) %>%
  set_engine("ranger")

ci_wf <- ci_wf %>% update_model(rf_model)
```

Then, we train it on the training data and validate the candidate hyper parameters on the validation data. By setting the argument `control` to `crtl_grid_ci`, we specify that the candidate models can be added to the stack.

```
set.seed(121)
rf_tune_res <- ci_wf %>%
    tune_grid(grid = 30,
              resamples = data_val_ci,
              control = ctrl_grid_ci
    )
```

```
i Creating pre-processing data to finalize unknown parameter: mtry
```

After tuning the random forest, we specify and tune a XGBoost model.

```
set.seed(121)
xgb_model <- boost_tree(
  trees = 1000,
  tree_depth = tune(),
  min_n = tune(),
  mtry = tune(),
  loss_reduction = tune(),
  learn_rate = tune()
) %>%
  set_engine("xgboost") %>%
  set_mode("classification")

ci_wf <- ci_wf %>% update_model(xgb_model)

doParallel::registerDoParallel()

xgb_tune_res <- tune_grid(
  ci_wf,
  resamples = data_val_ci,
```

```
  grid = 30,
  control = ctrl_grid_ci
)
```

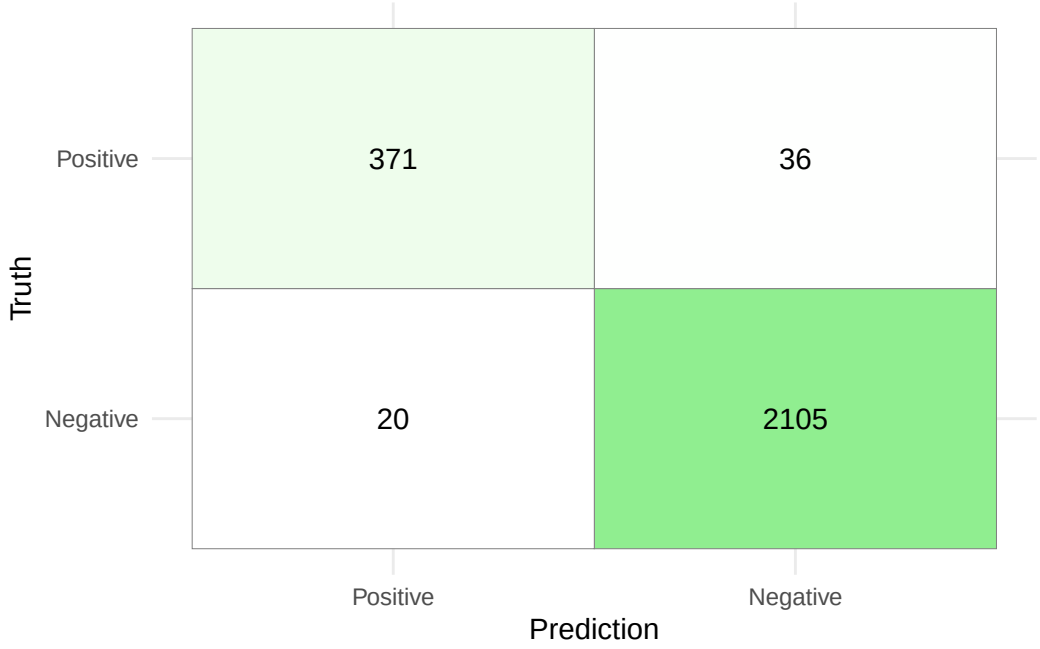
```
i Creating pre-processing data to finalize unknown parameter: mtry
```

3. ```
   stack_ci <- stacks() %>%
     add_candidates(rf_tune_res) %>%
     add_candidates(xgb_tune_res) %>%
     blend_predictions(
      mixture = 1.0,
      penalty = 10^(-6:-2),
      metric = metric_set(accuracy),
      times = 20
    )
   ```

4. ```
   set.seed(121)
   stack_ci <- stack_ci %>%
    fit_members()

   stack_pred <- data_test_ci %>%
     bind_cols(predict(stack_ci, .)) %>%
     mutate(
       .pred_class = factor(
         if_else(.pred_class == 'Existing Customer',
                 "Negative",
                 "Positive" )
         ),
       Attrition_Flag = factor(
         if_else(
           Attrition_Flag == 'Existing Customer',
           "Negative",
           "Positive")
         )
     )
   ```

```
stack_pred %>%
  conf_mat(Attrition_Flag,.pred_class) %>%
  pluck(1) %>%
  as_tibble() %>%
  mutate(
      Prediction = factor(Prediction,
```

```
                          levels = rev(levels(factor(Prediction)))),
    Truth = factor(Truth)
) %>%
ggplot(aes(x = Prediction, y = Truth,fill = n)) +
  geom_tile( colour = "gray50")+
  geom_text(aes(label = n))+
  scale_fill_gradient(low = "white", high = "lightgreen")+
  theme_minimal()+
  theme(legend.position = "none")
```



$$\text{Sensitivity} = \frac{370}{370+37} = 0.9090909$$

$$\text{Prescision} = \frac{370}{370+18} = 0.9536082$$

$$\text{Accuracy} = \frac{370+2107}{2532} = 0.978278$$

9 Neural Networks

9.1 Introduction

In this exercise session, we will consider some theoretical and practical aspects of neural networks. Neural networks are by now, probably the most commonly used models in Machine Learning Research and application, as they offer a highly versatile and well-performing approach for both, classification and regression tasks. However, as we have seen when considering models like XGBoost *with great performance comes great computational cost*. Training neural networks not only requires a lot of data to yield satisfactory results but can also take a long time to train, depending on the number of features and samples.

Training neural networks using the `{tidymodels}` approach is certainly possible with the `{brulee}` library, but not ideal. Most machine learning libraries and neural network architectures are far more accessible in scripting languages like Python, which encompass a much wider variety of development frameworks.

9.1.1 Installing TensorFlow for R

The `{tensorflow}` library provides a high-level API like Keras for model development, that is easy to use and highly productive.

To to get everything working, you will need to follow the steps outlined below.

1. Install the `{tensorflow}` package using the `{remotes}` package:

```
# install.packages("remotes")
remotes::install_github("rstudio/tensorflow")
```

2. Once the `{tensorflow}` package is installed and updated, we need to make sure that `Python` is installed on our machine. The reason is, that `TensoFflow` is written in `C++` and `CUDA` with `Python` being an interface to those languages. Since this interface has not yet been translated directly to R the most efficient approach was to create another interface that allows R code to be translated into Python without having to write a single line of Python code. To check if `Python` is already installed, you can execute the following snippet. If the snipped returns an empty character then `Python` is likely not yet installed.

```
Sys.which("python")
```

`Python` can then be installed by utilizing the `{reticulate}` library:

```
# install.packages("reticulate")
reticulate::install_python()
```

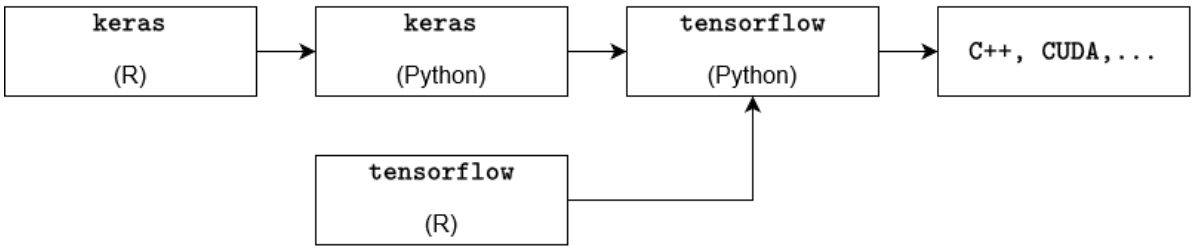
3. After installing `Python` and the `{tensorflow}` R library, we need to create a `Python` virtual environment and install the `TensorFlow` package in this virtual environment. This can be done with the following snippet:

```
library(tensorflow)
install_tensorflow(envname = "r-tensorflow")
```

4. An additional layer on top of `TensorFlow` is provided by `Keras`. `Keras` is a `Python` interface that communicates with `TensorFlow` with the benefit of being simple, flexible, and powerful.

```
install.packages("keras")
library(keras)
install_keras()
```

In summary, the process can be illustrated by the following diagram:



If everything was installed successfully, the following snippet should run without any issues:

```
library(tensorflow)
library(keras)
tf$constant("Hello TensorFlow!")
```

```
tf.Tensor(b'Hello TensorFlow!', shape=(), dtype=string)
```

Troubleshooting

There are several common problems you can run into while trying to install `Python` and TensorFlow which will be listed below with a potential solution.

- Installation of `{tensorflow}`:

```
remotes::install_github("rstudio/tensorflow")
```

If the return value is

> Skipping install of 'tensorflow' from a github remote, the SHA1 (eeb1e66b) has not changed since last install. Use `force = TRUE` to force installation

then `{tensorflow}` has already been installed.

- Installation of the `Python` package `tensorflow`:

```
library(tensorflow)
install_tensorflow(envname = "r-tensorflow")
```

If the return value is

> Error in install_tensorflow(envname = "r-tensorflow") : You should call install_tensorflow()/install_keras() only in a fresh R session that has not yet initialized Keras and TensorFlow (this is to avoid DLL in use errors during installation)

then restart the R under `Session -> Restart R`. It is likely that the `Python` package has already been installed, so instead of calling `install_tensorflow(envname = "r-tensorflow")` again, try the following snippet instead:

```
library(tensorflow)
tf$constant("Hello TensorFlow!")
```

```
tf.Tensor(b'Hello TensorFlow!', shape=(), dtype=string)
```

- Installation of the `Python` package `keras`:

```
library(keras)
install_keras()
```

If the return value is

> Error in tensorflow::install_tensorflow(method = method, conda = conda, : You should call install_tensorflow()/install_keras() only in a fresh R session that has not yet initialized Keras and TensorFlow (this is to avoid DLL in use errors during installation)

then restart the R under `Session -> Restart R`. It is likely that the `Python` package has already been installed, so instead of calling `install_keras()` again, try the following snippet instead:

```
library(tensorflow)
library(keras)
tf$constant("Hello TensorFlow!")
```

```
tf.Tensor(b'Hello TensorFlow!', shape=(), dtype=string)
```

9.2 Introduction

Besides the `keras` and `tensorflow` library we will need the following as well:

```
library(tidyverse)
library(tidymodels)
```

9.2.1 Estimating rent prices with feed forward neural networks

In this example, we once again, try to predict the base rent prices in Munich using the data Apartment rental offers in Germany, which is the same as in Exercise 04.

Recall that it contains roughly 850 rental listings for flats in Augsburg sourced from the online platform ImmoScout24 on three different dates in 2018 and 2019.

```
data_muc <- read.csv("data/rent_muc.csv")

set.seed(24)
split_rent <- initial_split(data_muc)
data_train <- training(split_rent)
data_test <- testing(split_rent)
```

Preprocessing of the data is handled by the following recipe:

```
rec_rent <- recipe(
    formula = baseRent ~.,
    data = data_train
  ) %>%
  update_role(scoutId, new_role = "ID") %>%
  step_select(!c("serviceCharge","heatingType","picturecount",
                 "totalRent",   "firingTypes","typeOfFlat",
                 "noRoomsRange", "petsAllowed",
                 "livingSpaceRange","regio3","heatingCosts",
                 "floor","date", "pricetrend")) %>%
  step_mutate(
    interiorQual = factor(
      interiorQual,
      levels = c("simple", "normal", "sophisticated", "luxury"),
      ordered = TRUE
      ),
    condition = factor(condition,
      levels = c("need_of_renovation", "negotiable","well_kept",
                 "refurbished","first_time_use_after_refurbishment",
                 "modernized","fully_renovated", "mint_condition",
                 "first_time_use"),
      ordered = TRUE),
    geo_plz = factor(geo_plz),
              across(where(is.logical),~as.numeric(.))) %>%
  step_string2factor(all_nominal_predictors(),
                     all_logical_predictors()) %>%
  step_ordinalscore(all_ordered_predictors())%>%
  step_novel(all_factor_predictors())%>%
  step_unknown(all_factor_predictors()) %>%
  step_dummy(geo_plz)%>%
  step_impute_knn(all_predictors()) %>%
  step_filter(baseRent <= 4000,
              livingSpace <= 200) %>%
  step_normalize(all_predictors())
```

Compared to the previously used recipes, we used an additional step called `step_normalize()`, that normalizes the specified variables. In other words, by applying the `step_normalize()` function, we transform the specified data such that each passed feature has a mean of zero and standard deviation of one.

After setting up the recipe, the approach of building a neural network model starts to differ from the usual `{tidymodels}` procedure. Since Keras and TensorFlow models are not part

of the {tidymodels} framework, we have to apply the recipe to the training and test data manually and add it to the training procedure. This can be done by using the previously introduced prep and bake functions. The preprocessed datasets then need to be split into training features and training labels. When creating the training features, we need to make sure to remove the feature scoutId as it should not be used as a predictor.

```
data_train_nn <- rec_rent %>%
  prep() %>%
  bake(data_train)

train_features <- data_train_nn %>%
  select(-c("scoutId", "baseRent"))

train_labels <- data_train_nn %>%
  select("baseRent")

data_test_nn <- rec_rent %>%
  prep() %>%
  bake(data_test)

test_features <- data_test_nn %>%
  select(-c("scoutId", "baseRent"))

test_labels <- data_test_nn %>%
  select(c("baseRent"))
```

Once we have successfully set up our data, we can move on to specifying our neural network architecture.

To specify a neural network architecture with two hidden layers and two dropout layers, we write a function build_model that takes the following inputs:

- n_input: input dimension (number of training labels).
- h1: dimension of the first hidden layer.
- h2: dimension of the second hidden layer.
- n_output: output dimension.
- d1: dropout rate of the first hidden layer.
- d2: dropout rate of the second hidden layer.

Given those inputs, the function should define a sequential model with 3 dense layers (where the first layer is equipped with the tanh activation function, and the second and third layer with the "relu" activation function) and 2 dropout layers in between.

```
build_model <- function(n_input, h1,h2,n_output,d1,d2) {
  model <- keras_model_sequential() %>%
    layer_dense(h1, activation = 'tanh', input_shape = n_input) %>%
    layer_dropout(rate = d1) %>%
    layer_dense(h2, activation = 'relu') %>%
    layer_dropout(rate = d2) %>%
    layer_dense(n_output, activation = 'relu')
  model
}
```

The idea behind writing such a function is to create a simple wrapper function that allows to specify the network parameters.

We can now define a model `dnn_model` with the newly written function from the previous code cell by passing the following inputs:

- `n_input`= `ncol(training_features)` (note, that the variable name might differ in your implementation)
- `h1= 100`
- `h2= 50`
- `n_output = 1`
- `d1 = 0.2`
- `d2 = 0.1`

Once the model is specified, compile the model using the `compile` function with loss specified as `"mae"` and optimizer `"optimizer_adam"`, where the learning rate is equal to `0.001` and the weight decay equal to `0`. By setting the additional argument `metrics = 'RootMeanSquaredError'`, we can directly evaluate the model using the same metric as for the models we used in previous exercises.

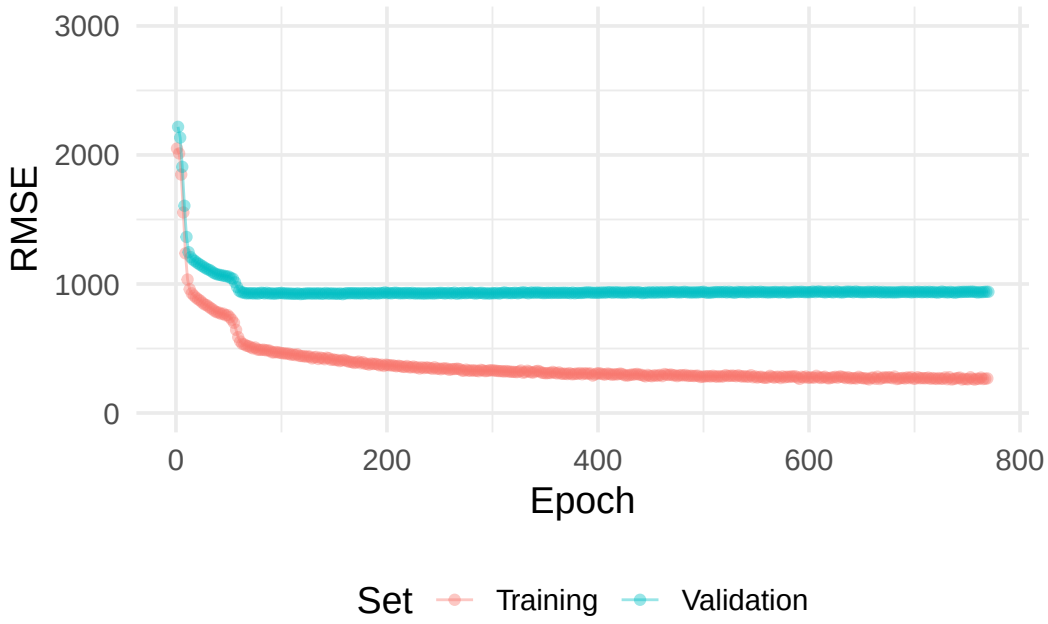
After compiling the model, we need to train it by using the `fit` function. As input parameters for the `fit` function, use the previously defined training features and training labels. Furthermore, set the validation split to `0.175` and the number of epochs to `400`. By setting the argument `verbose=1`, we can track the training process in the Viewer and Console pane.

```
set.seed(24)
tensorflow::set_random_seed(20)

dnn_model <- build_model(ncol(train_features), 256, 128, 1, 0.1, 0.1)
dnn_model <- dnn_model %>%
    compile(
    loss = 'mse',
    optimizer = optimizer_adam(0.0005, weight_decay = 0.001),
```

```
    metrics = 'RootMeanSquaredError'
  )

history <- dnn_model %>% fit(
  as.matrix(train_features),
  as.matrix(train_labels),
  validation_split = 0.175,
  verbose = 1,
  epochs = 385
)
```

We can also visualize the training procedure using the `{ggplot2}` library:

```
history %>%
  pluck(2) %>%
  as_tibble() %>%
  select(-c("loss","val_loss")) %>%
  pivot_longer(cols=c("root_mean_squared_error",
                      "val_root_mean_squared_error"),
               names_to = "set_name") %>%
  ggplot(aes(x=seq(1,nrow(.)),y=value, color = set_name))+
    geom_point(alpha=0.4)+
    geom_line(alpha=0.4)+
    scale_color_discrete(labels = c("Training","Validation"))+
    labs(
      color = "Set"
    )+
    xlab("Epoch")+
    ylab("RMSE")+
    ylim(0,3000)+
    theme_minimal(base_size = 14)+
    theme(
      legend.position = "bottom"
    )
```



Note, that in the example above. No hyperparametertuning was performed. Hyperparametertuning with Keras and TensorFlow is certainly possible, (see e.g., {kerastuneR}). However, this would be beyond the scope of this exercises and will be left for the reader to explore on their own.

To evaluate the model on the test set, we can use the `keras::evaluate` function:

```
test_results <- dnn_model %>% keras::evaluate(
  as.matrix(test_features),
  as.matrix(test_labels)
)
```

```
35/35 - 0s - loss: 236652.1875 - root_mean_squared_error: 486.4691 - 171ms/epoch - 5ms/step
```

```
test_results
```

```
                loss root_mean_squared_error
      236652.1875                 486.4691
```

The test error of 486.46 is better than the XGBoost test error (543) we observed in Session 06.

We can now use the neural network object `dnn_model_reg` to explore different interpretability methods. The package that provides these methods is `{iml}` which stands for "interpretable machine learning".

```
library(iml)
```

Before creating a new `Predictor` object called `mod` by piping the model `dnn_model_reg` into the `Predictor$new` function with additional arguments `data = train_features, y = train_labels`.

```
data_train_nn_filtered <- data_train_nn %>%
  filter(livingSpace<=quantile(livingSpace,0.95))

train_features_filtered <- data_train_nn_filtered %>%
  dplyr::select(-c("scoutId", "baseRent"))

train_labels_filtered <- data_train_nn_filtered %>%
  dplyr::select("baseRent")


mod <- Predictor$new(dnn_model,
                     data = train_features_filtered,
                     y = train_labels_filtered)
```

This predictor `mod` will be used throughout the remaining introduction.

We can use the `LocalModel$new()` function to create a (linear) local surrogate model for the second observation of the training features. By using the `"gower"` distance, the data points passed are weighted by their proximity to the instance to be explained. The argument `k` sets the number of features used for the approximation.

```
(loc.mod <- LocalModel$new(mod,
                           x.interest = train_features[2, ],
                           dist.fun = "gower",
                           k = 15))
```

```
Lade nötiges Paket: glmnet
```

```
Lade nötiges Paket: Matrix
```

```

Attache Paket: 'Matrix'
```

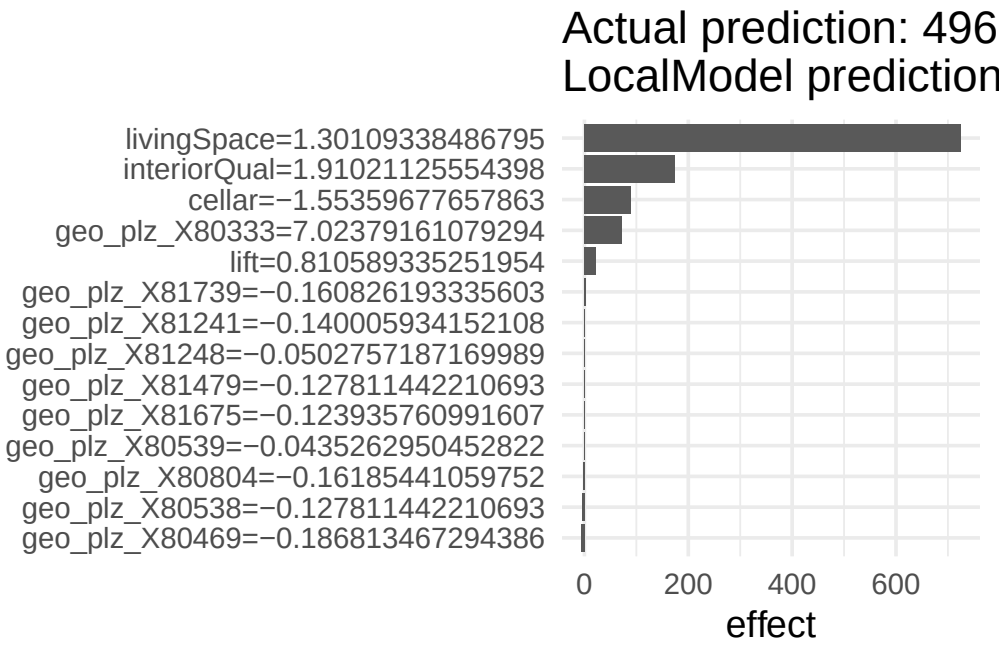
```
Die folgenden Objekte sind maskiert von 'package:tidyr':

    expand, pack, unpack

Loaded glmnet 4.1-8

Lade nötiges Paket: gower

98/98 - 0s - 218ms/epoch - 2ms/step

Warning in gower_work(x = x, y = y, pair_x = pair_x, pair_y = pair_y, n = NULL,
: skipping variable with zero or non-finite range.

Warning in gower_work(x = x, y = y, pair_x = pair_x, pair_y = pair_y, n = NULL,
: skipping variable with zero or non-finite range.

Warning in private$aggregate(): Had to choose a smaller k

Interpretation method:  LocalModel


Analysed predictor:
Prediction task: unknown


Analysed data:
Sampling from data.frame with 3123 rows and 100 columns.


Head of results:
                    beta  x.recoded     effect         x.original
cellar         -57.99440 -1.5535968  90.099908  -1.55359677657863
livingSpace    557.33055  1.3010934 725.139089   1.30109338486795
interiorQual    90.73355  1.9102113 173.320245   1.91021125554398
lift            27.32996  0.8105893  22.153371  0.810589335251954
geo_plz_X80333  10.16760  7.0237916  71.415139   7.02379161079294
geo_plz_X80469  35.07396 -0.1868135  -6.552289 -0.186813467294386
                     feature                      feature.value
cellar                cellar          cellar=-1.55359677657863
livingSpace       livingSpace      livingSpace=1.30109338486795
interiorQual     interiorQual     interiorQual=1.91021125554398
```

```
lift                      lift            lift=0.810589335251954
geo_plz_X80333 geo_plz_X80333   geo_plz_X80333=7.02379161079294
geo_plz_X80469 geo_plz_X80469 geo_plz_X80469=-0.186813467294386
```

After creating the model, we can use the `plot` function to visualize the model and identify the feature that has the largest effect on the outcome variable `baseRent`.

```
plot(loc.mod)+theme_minimal(base_size = 14)
```

```
1/1 - 0s - 24ms/epoch - 24ms/step
```



According to the generated plot, the most influential feature for the second sample is `livingSpace`.

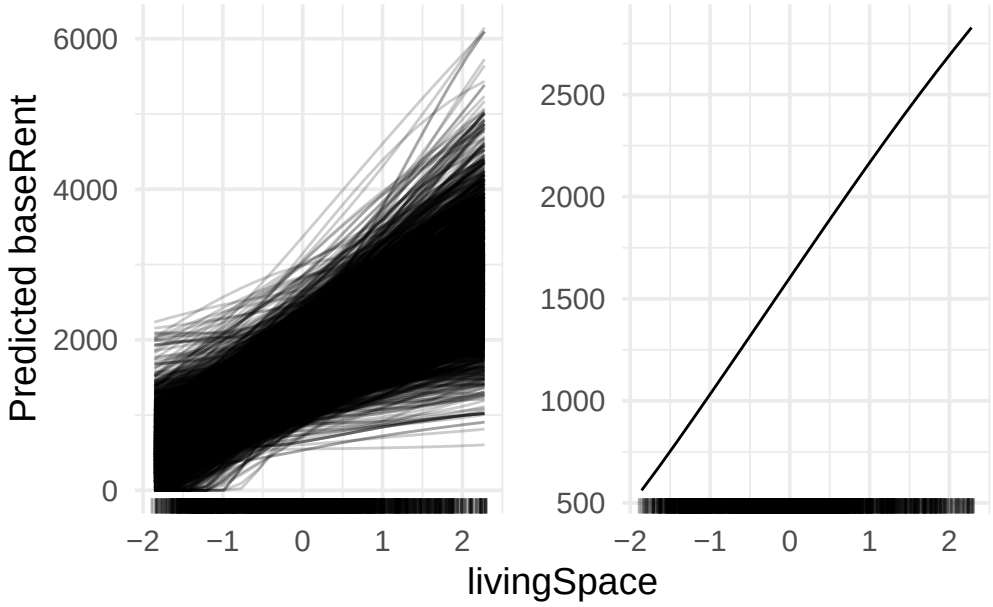
For the feature `livingspace`, we can create and plot feature effect plots where one plot is created using the method `"ice"` and the second plot created by using the method `"pdp"`.

Recall, that for ICE (Individual Conditional Expectation) plots, we choose one *individual* feature X_j, vary its value between $\min(x_j)$ and $\max(X_j)$, while keeping every other feature constant. By evaluating the model for each value, we obtain new estimates that (hopefully) show some trend with the varied feature. By repeating this procedure for every observation in the dataset, we obtain a curve for every instance that describes how the estimate behaves for varying X_j between the smallest and largest observed value.

A partial dependence plot can be obtained by considering the average of all curves for one feature.

```
library(patchwork)

# ICE
p1_fe <-  FeatureEffect$new(mod,
                            feature = "livingSpace",
                            method = "ice")$plot()

#Partial dependence plot
p2_fe <- FeatureEffect$new(mod,
                           feature = "livingSpace",
                           method = "pdp")$plot()
```

```
(p1_fe|p2_fe) +
    plot_layout(
      axes = "collect"
    )&
    theme_minimal(base_size = 14)
```



The ICE plot shows that for almost all observations, an increase in `livingSpace` corresponds to an increase in estimated base rent. The PDP plot shows, that on average, this is true as well.

To check if there are any observations that defeat this trend, we can analyse the points generated by the `FeatureEffect` function.

```
1  df_ice_filtered <- p1_fe$data %>%
2    group_by(.id) %>%
3    filter(livingSpace == min(livingSpace) |
4           livingSpace == max(livingSpace)) %>%
5    pivot_wider(names_from = livingSpace, values_from = .value) %>%
6    rename(min_rent = 3,
7           max_rent = 4)
8
9
10 df_ice_filtered %>% glimpse()
```

```
Rows: 3,123
Columns: 4
Groups: .id [3,123]
$ .type    <chr> "ice", "ice", "ice", "ice", "ice", "ice", "ice", "ice", "ice"~
$ .id      <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18~
$ min_rent <dbl> 442.1158, 1329.9663, 439.5775, 522.1421, 297.3833, 458.4910, ~
$ max_rent <dbl> 3068.9495, 6094.7588, 3161.6467, 3252.5591, 2675.5125, 2207.1~
```

First, we group the points by the `.id` column which represents the index of the samples. Since not only the smallest and largest living space is used to predict a new base rent, we filter every value between. By using the `pivot_wider` function in line 5, we create a wider data set with two new columns displaying the base rent estimated for the smallest and largest living space respectively. Since the newly created columns have messy names (these are the normalized values for the smallest and largest living space), we change them to `min_rent` and `max_rent`.

This newly created dataset can then be used to check for any abnormalities:

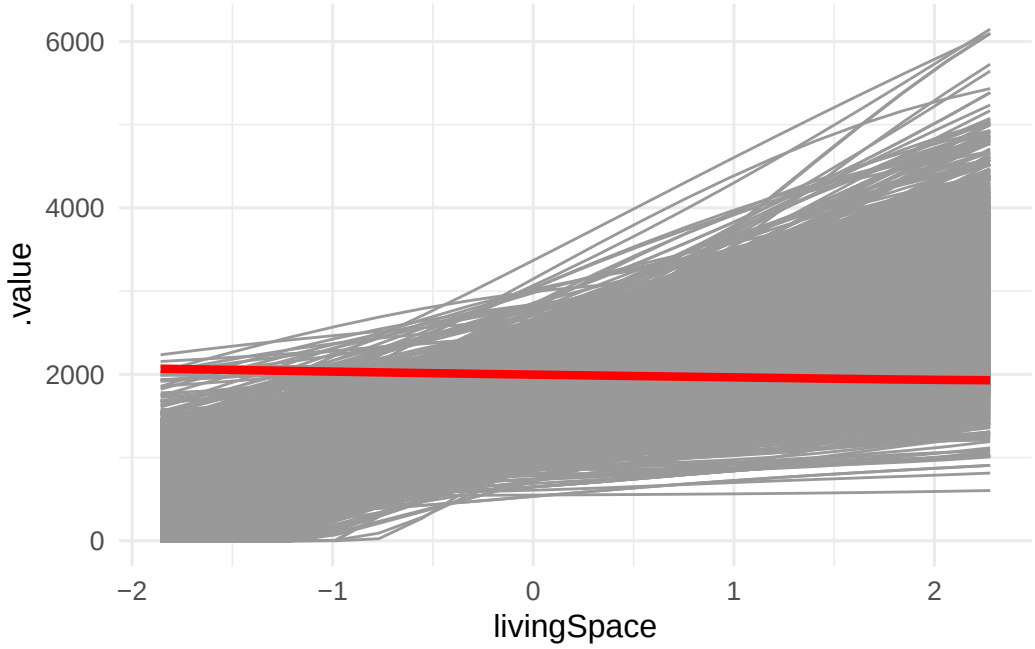
To check whether there are any listings where the base rent decreases with increasing living space, the following snippet provides an answer:

```
df_ice_filtered %>%
  filter(min_rent-max_rent >=0)
```

```
# A tibble: 1 x 4
# Groups:   .id [1]
  .type   .id min_rent max_rent
  <chr> <int>    <dbl>    <dbl>
1 ice    2618    2067.    1929.
```

Surprisingly there is one listing where an increase in living space yields a decreasing base Rent! Using the following snippet, we can visualize our finding:

```
p1_fe$data %>%
  filter(.id!=2618) %>%
  ggplot(aes(x=livingSpace,y=.value, group = .id))+
  geom_line(color="gray60")+
  geom_line(data=(filter(p1_fe$data,.id==2618)),
            aes(x=livingSpace,y=.value),
            color = "red",
            linewidth=1.5)+
  theme_minimal(base_size = 12)
```



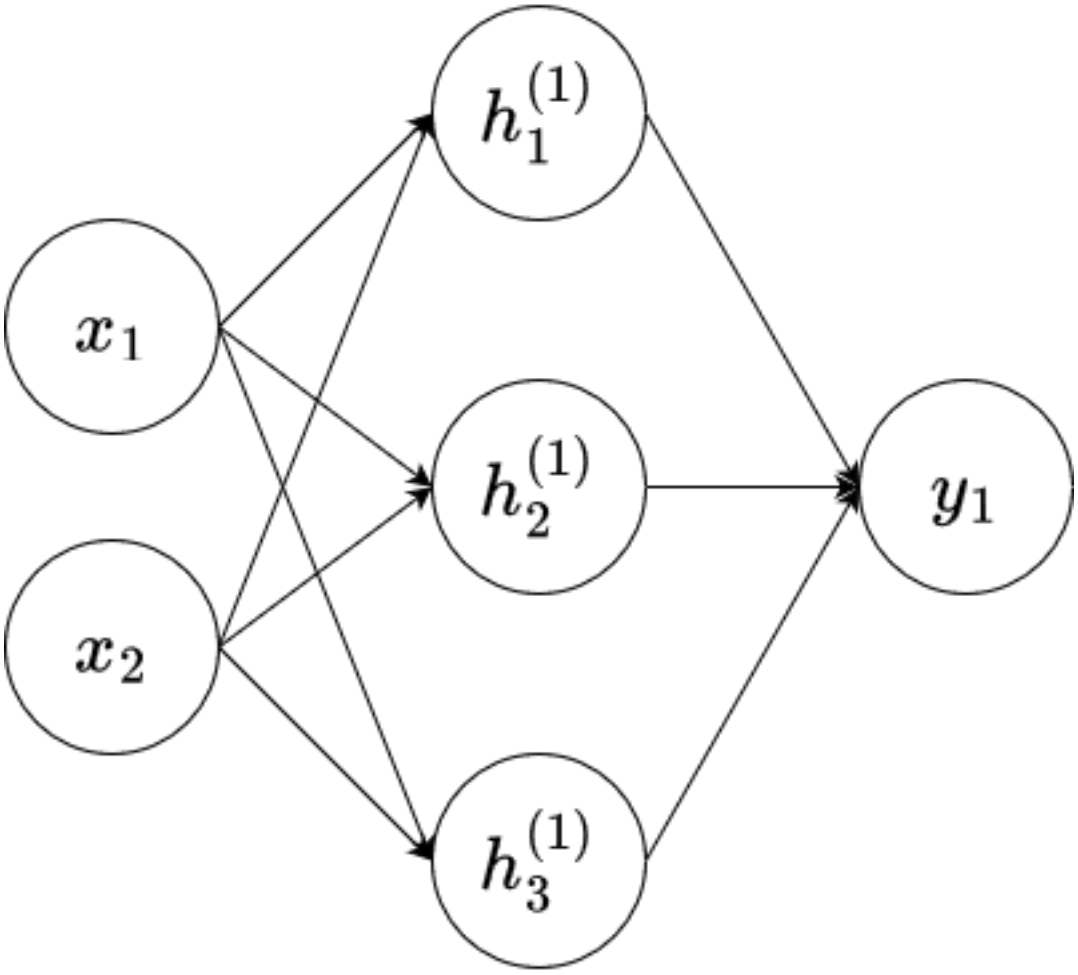
At this point, further analysis could be conducted to find out why that is the case. However, this is beyond the scope of this exercise.

9.3 Exercises

9.3.1 Theoretical Exercises

The notation used for the following exercise will be somewhat different from the notation in the lecture. However, I will try to explain every notational aspect as clearly as possible.

Consider the neural network depicted below.



Here,

- $(x_1, x_2)^\top \in \mathbb{R}^2$ denotes an input vector
- $\{h_i^{(1)}\}_{i=1}^3$ denote a hidden layer with each neuron defined by

$$h_i^{(1)} := \sigma_1 \left(\sum_{j=1}^{2} w_{ij}^{(1)} x_j + b_1 \right)$$

  where $\sigma_1 : \mathbb{R} \to \mathbb{R}$ denotes an activation function, $w_{i1}^{(1)}, w_{i2}^{(1)}, b_1 \in \mathbb{R}, i = 1, ..., 3$ denote weights and a bias for the first layer.

- y_1 is the output of the neural network that is defined as

$$y_1 = \sigma_2 \left(\sum_{i=1}^{3} w_{i1}^{(2)} h_i^{(1)} + b_2 \right), \tag{9.1}$$

  where $\sigma_2 : \mathbb{R} \to \mathbb{R}$ denotes another activation function, and $w_{i1}^{(2)}, b_2 \in \mathbb{R}, i = 1, ..., 3$ denote weights and a bias for the second/final layer.

Exercise 9.1. Assume that $\sigma_1 = \sigma_2 = \text{ReLU}$, i.e., $x \mapsto \max\{0, x\}$. Set up the model equation for this neural network by expanding the term y_1.

Exercise 9.2. In this exercise, we want to implement and visualize the neural network of Exercise 9.1.

1. Write a function `nn` that represents the neural network of the previous exercise. It should take a vector $x \in \mathbb{R}^2$, a weight matrix $w_1 \in \mathbb{R}^{3\times 2}$ used to calculate $h_i^{(1)}$, a weight vector $w_2 \in \mathbb{R}^3$ used to calculate the term $\sum_{i=1}^{3} w_{i1}^{(2)} h_i^{(1)}$ and two bias terms $b_1, b_2 \in \mathbb{R}$ as an input. The return value should be the output of the neural network we used in the previous exercises.
2. Given the following weights, biases, and input matrix `x`, calculate the return value and plot the results using the `geom_tile()` function where the return value of the `nn` function is set to be the fill parameter.

> **Tip**
>
> You will have to use the `lapply` function to be able to apply the input matrix `x` to the function `nn`.

Exercise 9.3. We can generalize the neural network above in a way that the hidden layer has $n \in \mathbb{N}$ instead of 3 neurons. Modify Equation 9.1, such that the hidden layer now has $n \in \mathbb{N}$ neurons! What are the benefits and drawbacks of adding additional neurons? Name one each.

Exercise 9.4. Explain in your own words the difference between global interpretability models and local interpretability models.

Exercise 9.5. Explain in your own words the meaning of "post-hoc" interpretation mehtods.

Post-hoc intepretation refers to the process of interpreting the model's behavior subsequent to model training. The goal is to make use of the predifined structure of the model without modifying the training process itself.

Exercise 9.6. Explain in your own words the difference between individual conditional expectation (ICE) for a feature and partial dependence on a feature.

9.3.2 Neural Networks for Classification

The dataset for this exercise will be the Credit Card Customers data set that we already used in previous exercises. You can either download it again using the provided link or the button below.

Download BankChurners

Recall that the data set consists of 10,127 entries that represent individual customers of a bank including but not limited to their age, salary, credit card limit, and credit card category.

The goal is to find out whether a customer will stay or leave the bank given the above features.

The following training, validation and test split should be used for training the models of the subsequent exercises.

```
credit_info <- read.csv("data/BankChurners.csv")

credit_info <- credit_info %>%
  mutate(
    Attrition_Flag = if_else(Attrition_Flag=="Attrited Customer",
                             1,
                             0)
)

set.seed(121)
split <- initial_split(credit_info, strata = Attrition_Flag)
data_train_ci <- training(split)
data_test_ci <- testing(split)
```

Preprocessing of the data is handled by the following recipe.

```
levels_income <- c("Less than $40K","$40K - $60K",
                   "$60K - $80K","$80K - $120K","$120K +")

levels_education <- c("Uneducated", "High School","College",
                      "Graduate",  "Post-Graduate", "Doctorate")

rec_ci <- recipe(Attrition_Flag ~., data = data_train_ci) %>%
  update_role(CLIENTNUM, new_role = "ID") %>%
  step_mutate_at(all_nominal_predictors(),
                 fn = ~if_else(.%in% c("Unknown","unknown"),NA,.)
  )%>%
```

```
  step_string2factor(Income_Category,
                     levels = levels_income,
                     ordered = TRUE) %>%
  step_string2factor(Education_Level,
                     levels = levels_education,
                     ordered = TRUE) %>%
  step_ordinalscore(all_ordered_predictors()) %>%
  step_unknown(all_factor_predictors()) %>%
  step_impute_knn(all_predictors()) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_corr(all_predictors()) %>%
  step_normalize(all_predictors())
```

Exercise 9.7. Given the recipe and data split, create the training and testing features and labels that will later be used for training a neural network.

Exercise 9.8. For this exercise set the internal R seed to `24` and `tensorflow::set_random_seed(2)` to obtain reproducible results.

1. Write a function `build_model` with arguments `n_input, h1,h2,n_output,d1,d2`, where

   - `n_input` denotes the dimension of the input,
   - `h_i` $i = 1, 2$ denotes the dimensions of two hidden layers,
   - `n_output` denotes the dimension of the output, and
   - `d_i` $i = 1, 2$ denotes the dropout rate of the two hidden layers.

   Given these inputs, the function should create a Keras sequential model consisting of three dense layers with two dropout layers in between having a dropout rate of `d1` and `d2` respectively. The first and second dense layer should be equipped with the `'relu'` activation function and the last layer with the sigmoid activation function. The function `build_model` should then return the specified model.

2. Create an instance of a keras sequential model using the function described with arguments

   - `n_input = ncol(train_features)`,
   - `h_1 = 30, h_2 =  4`,
   - `n_output = 1`, and
   - `d_1 = 0.2, d_2 = 0`.

3. Compile the model with `loss` set to `binary_crossentropy`, `optimizer` set to `optimizer_adam(5e-4, weight_decay = 1e-6)` and `metrics= 'Accuracy'`. Then, train the model using a validation split size of 10% and 100 epochs using the `fit()` function.

4. Finally, evaluate the model on the test data.

Exercise 9.9. Use the following Code snippet to generate predictions for the test data and construct a confusion matrix based on the predictions. Then, calculate the sensitivity, precision and accuracy for the model.

```
test_pred <- dnn_model %>%
  predict(
    as.matrix(test_features)
  ) %>%
  as_tibble() %>%
  mutate(
    pred_class = if_else(V1>0.5,"Positive","Negative") %>% factor(),
    truth = if_else(test_labels == 1,"Positive","Negative") %>% factor()
  )
```

9.4 Solutions

Solution 9.1 (Exercise 9.1).

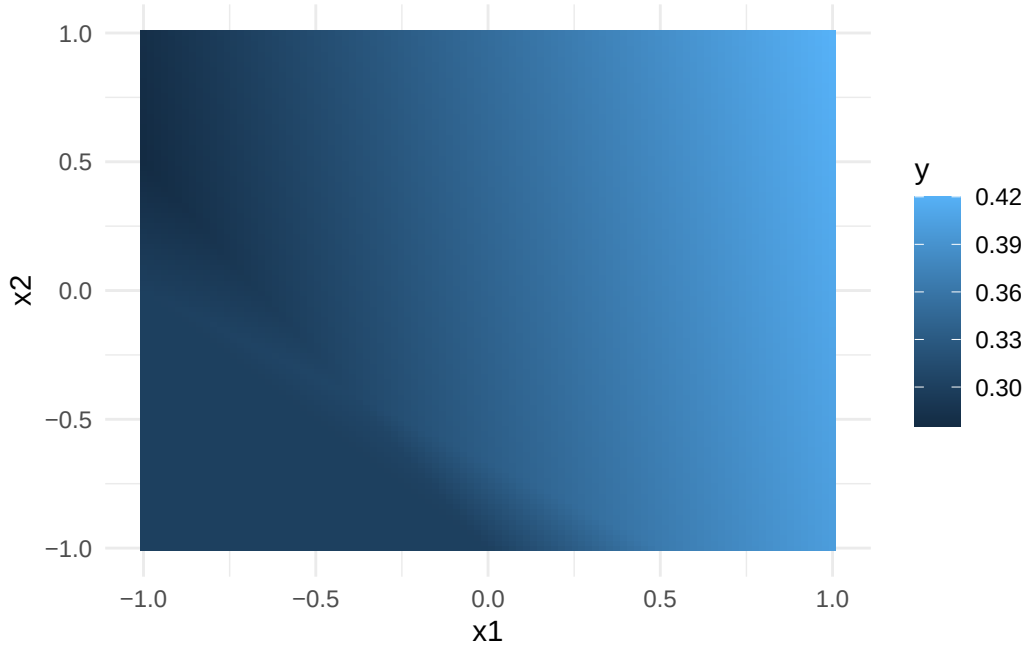
$$
\begin{aligned}
y_1 = & \quad \sigma_2\left(\sum_{i=1}^{3} w_{i1}^{(2)} h_i^{(1)} + b_2\right) \\
= & \quad \max\left\{\sum_{i=1}^{3} w_{i1}^{(2)} h_i^{(1)} + b_2, 0\right\} \\
= & \quad \max\left\{\sum_{i=1}^{3} w_{i1}^{(2)} \sigma_1\left(\sum_{j=1}^{2} w_{ij}^{(1)} x_j + b_1\right) + b_2, 0\right\} \\
= & \quad \max\left\{\sum_{i=1}^{3} w_{i1}^{(2)} \max\left\{\sum_{j=1}^{2} w_{ij}^{(1)} x_j + b_1, 0\right\} + b_2, 0\right\} \\
= & \quad \max\left\{\sum_{i=1}^{3} w_{i1}^{(2)} \max\left\{w_{i1}^{(1)} x_1 + w_{i2}^{(1)} x_2 + b_1, 0\right\} + b_2, 0\right\} \\
= & \quad \max\{ \\
& w_{11}^{(2)} \max\left\{w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1, 0\right\} + \\
& w_{21}^{(2)} \max\left\{w_{21}^{(1)} x_1 + w_{22}^{(1)} x_2 + b_1, 0\right\} + \\
& w_{31}^{(2)} \max\left\{w_{31}^{(1)} x_1 + w_{32}^{(1)} x_2 + b_1, 0\right\} + \\
& b_2, 0\}
\end{aligned}
$$

Solution 9.2 (Exercise 9.2).

1.
```r
nn <- function(x, w1, w2, b1, b2){
  h = pmax(w1%*%x+b1,0)
  y = pmax(w2%*%h+b2,0)
  return(y[1])
}
```

2.
```r
w1 <- matrix(c(0.2,0.2,0.3,0.3,0.2,0.2), ncol = 2)
w2 <- matrix(c(-0.3,0.2,0.3), nrow = 1)
b1 <- 0.2
b2 <- 0.3

x1 <- seq(-1,1,length.out = 100)
x2 <- seq(-1,1,length.out = 100)

x <- expand.grid(x1,x2) %>% as.matrix()
```

```r
y <- apply(x, 1, function(row) nn(row, w1, w2, b1, b2))

cbind(x, y) %>%
  as_tibble() %>% set_names(c("x1","x2","y")) %>%
  ggplot(aes(x=x1,y=x2,fill = y)) +
  geom_tile() +
  theme_minimal()
```



Solution 9.3 (Exercise 9.3).

$$y_1 = \sigma_2\left(\sum_{i=1}^{n} w_{i1}^{(2)} h_i^{(1)} + b_2\right)$$

Advantage The model is more flexible and can therefore better estimate the training data.

Disadvantage: The model is prone to over fitting.

Solution 9.4 (Exercise 9.4).

- The goal of local interpretability is to explain the model's behavior for a specific input, or a single observation in general.
- The goal of global interpretability models is to explain the model's behavior on the whole input data.

Solution 9.5 (Exercise 9.5). Post-hoc intepretation refers to the process of interpreting the model's behavior subsequent to model training. The goal is to make use of the predifined structure of the model without modifying the training process itself.

Solution 9.6 (Exercise 9.6). ICE is used to model the relationship between a single input feature and the model's predictions. Partial dependence is a modification of ICE, as it simply takes the average of ICE over the whole sample.

Solution 9.7 (Exercise 9.7).

```
data_train_nn <- rec_ci %>%
  prep() %>%
  bake(data_train_ci)

train_features <- data_train_nn %>%
  select(-c(CLIENTNUM,Attrition_Flag))

train_labels <- data_train_nn %>%
  select(Attrition_Flag)

data_test_nn <- rec_ci %>%
  prep() %>%
  bake(data_test_ci)

test_features <- data_test_nn %>%
  select(-c(CLIENTNUM,Attrition_Flag))

test_labels <- data_test_nn %>%
  select(Attrition_Flag)
```

Solution 9.8 (Exercise 9.8).

```
build_model <- function(n_input, h1,h2,n_output,d1,d2) {
  model <- keras_model_sequential() %>%
    layer_dense(h1, activation = 'relu', input_shape = n_input) %>%
    layer_dropout(rate = d1) %>%
    layer_dense(h2, activation = 'relu') %>%
    layer_dropout(rate = d2) %>%
    layer_dense(n_output, activation = 'sigmoid')
  model
}
```

```
set.seed(24)
tensorflow::set_random_seed(2)

dnn_model <- build_model(ncol(train_features), 30, 4, 1, 0.2, 0)
dnn_model <- dnn_model %>%
    compile(
    loss = 'binary_crossentropy',
    optimizer = optimizer_adam(5e-4, weight_decay = 1e-6),
    metrics='Accuracy'
  )

history <- dnn_model %>% fit(
  as.matrix(train_features),
  as.matrix(train_labels),
  validation_split = 0.1,
  verbose = 1,
  epochs = 100
)
```

```
test_results <- dnn_model %>% keras::evaluate(
  as.matrix(test_features),
  as.matrix(test_labels)
)
```

```
80/80 - 0s - loss: 0.7566 - Accuracy: 0.9021 - 158ms/epoch - 2ms/step
```

```
test_results
```

```
     loss  Accuracy
0.7566464 0.9020537
```

Solution 9.9 (Exercise 9.9).

```
test_pred <- dnn_model %>%
  predict(
    as.matrix(test_features)
  ) %>%
  as_tibble() %>%
  mutate(
    pred_class = if_else(V1>0.5,"Positive","Negative") %>% factor(),
```

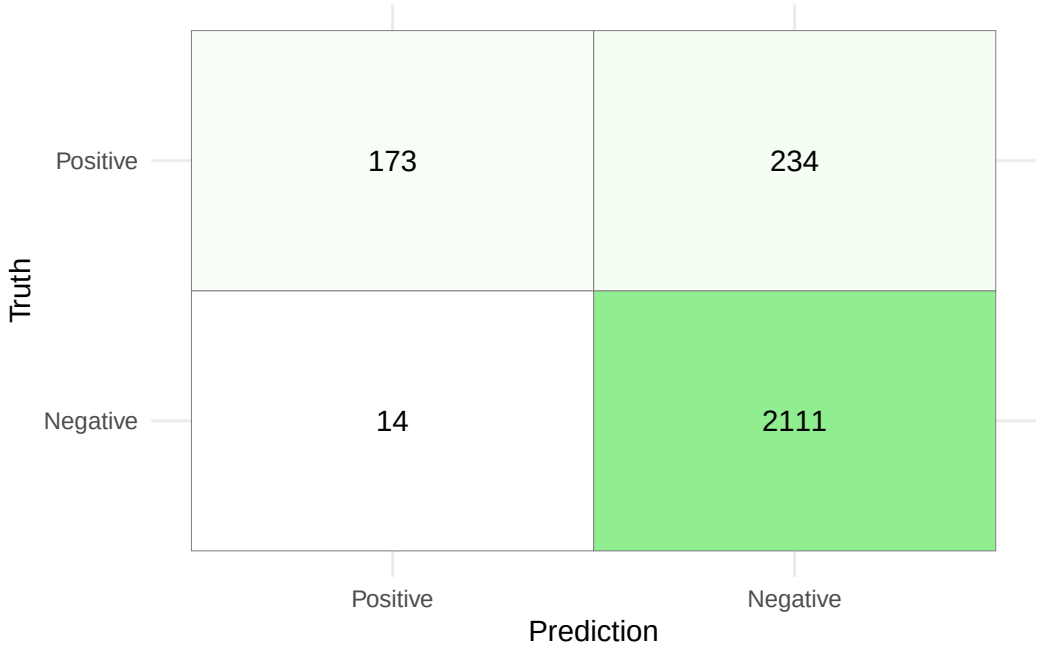
```
    truth = if_else(test_labels == 1,"Positive","Negative") %>% factor()
  )
```

```
80/80 - 0s - 228ms/epoch - 3ms/step
```

```
cm_nn <- test_pred %>% conf_mat(estimate = pred_class,truth=truth)

table(test_pred$pred_class,test_pred$truth)
```

```
          Negative Positive
  Negative     2111      234
  Positive       14      173
```

```
cm_tib <- as_tibble(cm_nn$table)%>%
  mutate(
    Prediction = factor(Prediction,
                        levels = rev(levels(factor(Prediction)))),
    Truth = factor(Truth)
)

cm_tib %>% ggplot(aes(x = Prediction, y = Truth,fill = n)) +
    geom_tile( colour = "gray50")+
    geom_text(aes(label = n))+
    scale_fill_gradient(low = "white", high = "lightgreen")+
    theme_minimal()+
    theme(legend.position = "none")
```



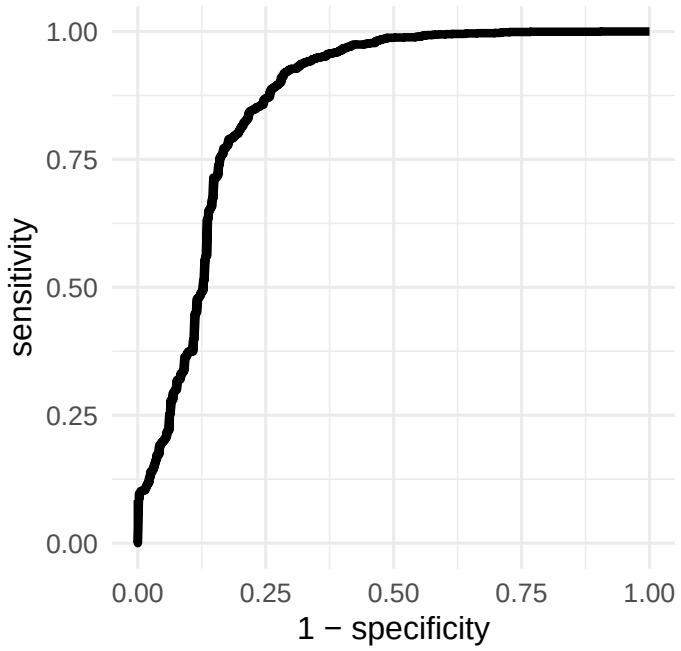
The metrics are then given by

$$\begin{aligned}\text{Accuracy} &= \frac{173+2111}{173+2111+14+234} = 0.902\\ \text{Recall} &= \frac{173}{173+234} = 0.4250\\ \text{Precision} &= \frac{173}{173+14} = 0.925\end{aligned}$$

While the accuracy and precision of the model are relatively high, these metrics should be chosen carefully when the data set is unbalanced. Especially the precision is worrisome, since the model only correctly identifies roughly 42% of the customers that might want to leave the bank.

Another aspect we have not considered: The classification threshold is set to 0.5, which might not be optimal. Consider the following ROC curve was produced by the DNN model.

```
test_pred %>%
  roc_curve(truth = truth,V1) %>%
  ggplot(aes(x=1-specificity,y=sensitivity))+
  geom_line(linewidth=1.5)+
  coord_equal()+
  theme_minimal(base_size=12)
```



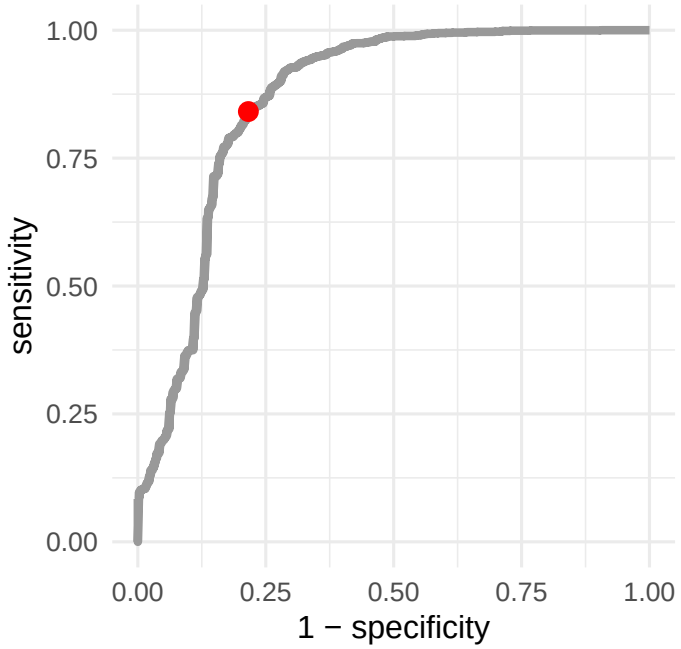
The optimal threshold value with respect to the ROC curve is the value that maximizes both, the sensitivity and specificity of the model. In simpler terms, the optimal threshold value is the one closest to the top left corner. To find this value, we can simply calculate the euclidean distance of all the points on the ROC curve to the top left corner and select the threshold of the one with minimal distance to the top left corner.

```
test_pred %>%
  roc_curve(truth = truth,V1) %>%
  mutate(
    dist = sqrt((1-specificity)^2+(sensitivity-1)^2)
  ) %>%
  filter(dist == min(dist))
```

```
# A tibble: 1 x 4
  .threshold specificity sensitivity  dist
       <dbl>       <dbl>       <dbl> <dbl>
1      0.997       0.784       0.841 0.268
```

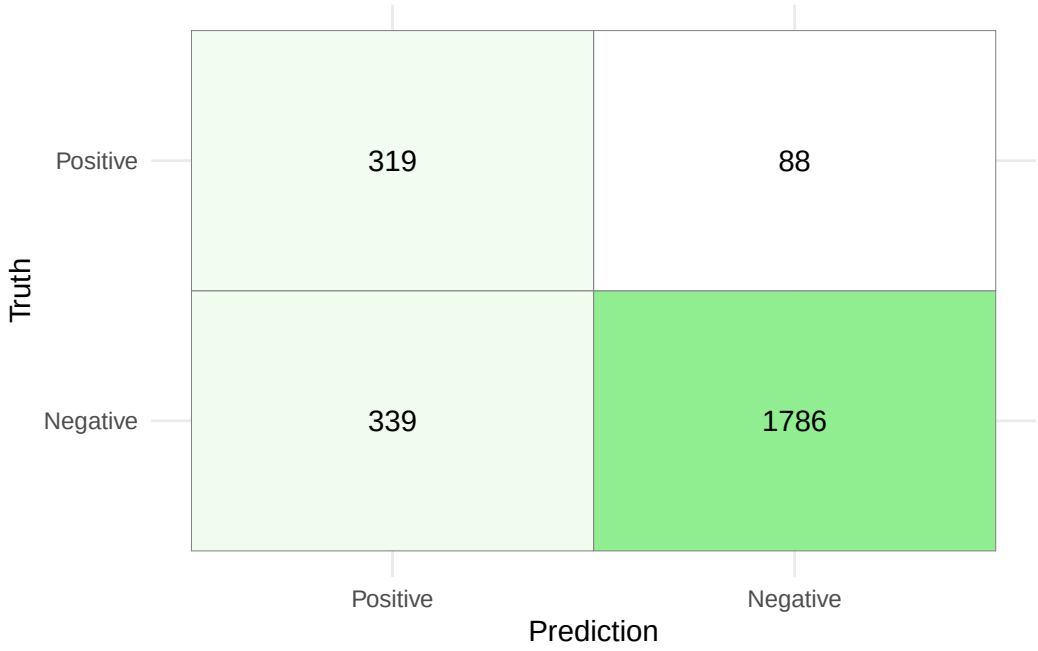
To visualize the point on the ROC curve that corresponds to the optimal threshold, consider the following code snippet:

```r
test_pred %>%
  roc_curve(truth = truth,V1) %>%
  ggplot(aes(x=1-specificity,y=sensitivity))+
  geom_line(linewidth=1.5, color = "gray60")+
  geom_point(data = tibble(x=1-0.784,y=0.841), aes(x=x,y=y), color = "red", size = 3)+
  coord_equal()+
  theme_minimal(base_size=12)
```



Using this threshold produces the following confusion matrix:

```
80/80 - 0s - 175ms/epoch - 2ms/step
```



While the accuracy of the model with the new threshold is lower (90.2% vs. 83.1%), the new model identifies people who are at risk of leaving the bank a lot better.

10 Unsupervised Learning

10.1 Introduction

In this last exercise session, we will consider a major subfield of machine learning: unsupervised learning. In particular, we will consider clustering with hierarchical methods, the k-means algorithm, and dbscan. Besides unsupervised learning, we will also consider dimension reduction algorithms such as principal component analysis and autoencoders.

10.1.1 Dimension reduction

Many of the data sets we encountered so far have a dimension $\gg 2$. We can therefore not directly visualize all features and the response variable in one figure. One solution is to build a facet plot (similar to Exercise 1.15) that displays every possible combination of variables. However, this is increasingly difficult: Consider the Credit Info data set we frequently use. There are $n = 10$ variables we usually use and to display every variable combination, $\binom{10}{2} = 45$ subplots would be required. The idea of dimension reduction aims to solve this problem. Instead of considering every feature individually, dimension reduction usually merges feature values based on some rule set. Principal component analysis combines features linearly in a way that the resulting linear combinations explain most of the variability in the data set. Autoencoders on the other hand aim to map the features into a lower dimensional subspace (latent space) and rebuild the true features based on the features in the latent space.

10.1.1.1 Principal Component Analysis (PCA)

Heuristically speaking, PCA assumes that in a n dimensional (feature) space, not every variable is equally important in terms of variability. Consider a feature matrix $X \in \mathbb{R}^{n \times p}$ (p features and n observations) with normalized features. To find the features that explain the most variance, the following steps are performed:

1. Center the features such that $\mathbb{E}(X_j) = \frac{1}{n} \sum_{i=1}^{n} x_{ij} = 0$ for $j = 1, ..., p$.
2. Calculate the empirical covariance matrix $X'X =: \Sigma \in \mathbb{R}^{p \times p}$ that measures the joint variability of each pair of features.
3. Calculate the singular value decomposition of Σ:

i) Solve the system $\det(\Sigma - \lambda I_p) = 0$ where

$$I_p = \begin{pmatrix} 1 & 0 & ... & 0 \\ 0 & 1 & ... & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & ... & 1 \end{pmatrix} \in \mathbb{R}^{p \times p} \text{ and} \tag{10.1}$$

$$\lambda \in \mathbb{R}. \tag{10.2}$$

for λ. Here, we assume that there are p distinct solutions.

ii) Reorder the solutions λ_j, $j = 1, ..., p$ descending such that $\lambda_{(1)} \geq ... \geq \lambda_{(q)}$.

iii) For each ordered solution $\lambda_{(j)}$, $j = 1, ..., p$ that solves $\det(\Sigma - \lambda I_p) = 0$, solve the system

$$(\Sigma - \lambda_{(j)} I_p)\gamma_{(j)} = 0, \tag{10.3}$$

where $\gamma_{(j)} \in \mathbb{R}^p$ for $j = 1, ..., p$.

iv) Normalize the resulting vectors to obtain the principal components of the data. Note, that the normalizing constant of each vector $\gamma_{(j)}$ is given by $\frac{1}{\|\gamma_{(j)}\|}$.

By choosing the first $k \leq p$ of the p ordered principal components, we can map the feature matrix X to the k-dimensional subspace by multiplying X with the respective matrix $(\gamma_{(1)}, ..., \gamma_{(k)}) \in \mathbb{R}^{p \times k}$. The relative amount of variance explained by the first k principal components is then given by the fraction

$$\frac{1}{\sum_{j=1}^{p} \lambda_{(j)}} \sum_{j=1}^{k} \lambda_{(j)}. \tag{10.4}$$

10.1.1.2 PCA in R

We will need the following libraries throughout the session:

```
library(tidyverse)
library(tidymodels)
library(tensorflow)
library(keras)
library(ggtext)
library(patchwork)
```

Let us consider the Bank Churners or Credit Info data set that we already used in previous exercises. You can either download it again using the provided link or the button below.

Download BankChurners

Recall that the data set consists of 10,127 entries that represent individual customers of a bank including but not limited to their age, salary, credit card limit, and credit card category.

The goal is to reduce the dimensionality of the data set while leaving out the variable `Attrition_Flag`. By doing so, we might be able to get a better intuition on the distribution of attrited customers within the reduced data set.

The following code snippet takes care of the pre processing:

```
credit_info <- read.csv("data/BankChurners.csv")

credit_info <- credit_info %>% select(-c(CLIENTNUM))

levels_income <- c("Less than $40K","$40K - $60K",
                   "$60K - $80K","$80K - $120K","$120K +")

levels_education <- c("Uneducated", "High School","College",
                      "Graduate",  "Post-Graduate", "Doctorate")

rec_ci <- recipe(Attrition_Flag~., data =credit_info ) %>%
  step_mutate_at(all_nominal_predictors(),
                 fn = ~if_else(.%in% c("Unknown","unknown"),NA,.)
  )%>%
  step_string2factor(Income_Category,
                     levels = levels_income,
                     ordered = TRUE) %>%
  step_string2factor(Education_Level,
                     levels = levels_education,
                     ordered = TRUE) %>%
  step_ordinalscore(all_ordered_predictors()) %>%
  step_unknown(all_factor_predictors()) %>%
  step_impute_knn(all_predictors()) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_corr(all_predictors()) %>%
  step_normalize(all_predictors())

credit_info_prepped <- rec_ci %>%
  prep() %>%
  bake(credit_info)

credit_info_prepped %>% glimpse()
```

```
Rows: 10,127
```

```
Columns: 32
$ Customer_Age                   <dbl> -0.16539741, 0.33355391, 0.58302958, -0~
$ Dependent_count                <dbl> 0.5033433, 2.0430978, 0.5033433, 1.2732~
$ Months_on_book                 <dbl> 0.38460189, 1.01066492, 0.00896407, -0.~
$ Total_Relationship_Count       <dbl> 0.7639049, 1.4072367, 0.1205731, -0.522~
$ Months_Inactive_12_mon         <dbl> -1.3270705, -1.3270705, -1.3270705, 1.6~
$ Contacts_Count_12_mon          <dbl> 0.4923795, -0.4115957, -2.2195459, -1.3~
$ Credit_Limit                   <dbl> 0.446599851, -0.041364610, -0.573669472~
$ Total_Revolving_Bal            <dbl> -0.473398843, -0.366648718, -1.42678789~
$ Total_Amt_Chng_Q4_Q1           <dbl> 2.6233649, 3.5631169, 8.3668007, 2.9426~
$ Total_Trans_Amt                <dbl> -0.9596592, -0.9163874, -0.7409451, -0.~
$ Total_Trans_Ct                 <dbl> -0.9738471, -1.3572734, -1.9111113, -1.~
$ Total_Ct_Chng_Q4_Q1            <dbl> 3.833813303, 12.607950374, 6.807527542,~
$ Avg_Utilization_Ratio          <dbl> -0.77584393, -0.61624523, -0.99710576, ~
$ Attrition_Flag                 <fct> Existing Customer, Existing Customer, E~
$ Gender_M                       <dbl> 1.0599033, -0.9433891, 1.0599033, -0.94~
$ Education_Level_Doctorate      <dbl> -0.2158832, -0.2158832, -0.2158832, -0.~
$ Education_Level_Graduate       <dbl> -0.6684885, 1.4957644, 1.4957644, -0.66~
$ Education_Level_High.School    <dbl> 2.0075861, -0.4980615, -0.4980615, 2.00~
$ Education_Level_Post.Graduate  <dbl> -0.2316963, -0.2316963, -0.2316963, -0.~
$ Education_Level_Uneducated     <dbl> -0.4148367, -0.4148367, -0.4148367, -0.~
$ Education_Level_unknown        <dbl> -0.4200552, -0.4200552, -0.4200552, -0.~
$ Marital_Status_Married         <dbl> 1.077285, -0.928168, 1.077285, -0.92816~
$ Marital_Status_Single          <dbl> -0.7984674, 1.2522756, -0.7984674, -0.7~
$ Marital_Status_unknown         <dbl> -0.2825949, -0.2825949, -0.2825949, 3.5~
$ Income_Category_X.40K....60K   <dbl> -0.4633404, -0.4633404, -0.4633404, -0.~
$ Income_Category_X.60K....80K   <dbl> 2.4945216, -0.4008389, -0.4008389, -0.4~
$ Income_Category_X.80K....120K  <dbl> -0.4226546, -0.4226546, 2.3657644, -0.4~
$ Income_Category_Less.than..40K <dbl> -0.7364005, 1.3578225, -0.7364005, 1.35~
$ Income_Category_unknown        <dbl> -0.3511948, -0.3511948, -0.3511948, -0.~
$ Card_Category_Gold             <dbl> -0.1076388, -0.1076388, -0.1076388, -0.~
$ Card_Category_Platinum         <dbl> -0.04448181, -0.04448181, -0.04448181, ~
$ Card_Category_Silver           <dbl> -0.2407818, -0.2407818, -0.2407818, -0.~
```

We now have a preprocessed data set `credit_info_prepped` that can be applied to different dimension reduction algorithms.

To fit a PCA model, we have to pass the preprocessed data into the `prcomp` function. The return value is a list containing the standard deviations of the principal components (the square roots of the singular values of the covariance matrix.) and the matrix of variable loadings.

```
pca_fit <- credit_info_prepped %>%
  select(-Attrition_Flag) %>%
  prcomp()
```

We can then use the `augment` function to create the reduced dataset. To access the first two principal components, we can use the `select` function.

```
data_reduced <- pca_fit %>%
  augment(credit_info_prepped) %>%
  select(c(.fittedPC1,.fittedPC2))
```

```
title_text_p1 <- "Visualisation of the first two principal components"
subtitle_text_p1 <- "The <span style='color:#0000FF;'> blue </span>
                     dots represent customers with positive attrition flag"

title_text_p2 <- "Visualisation of the explained variance </br>
                  per principal component"


pca_fit_pos <- data_reduced %>%
  mutate(Attrition_Flag = credit_info_prepped$Attrition_Flag) %>%
  filter(Attrition_Flag == "Attrited Customer")

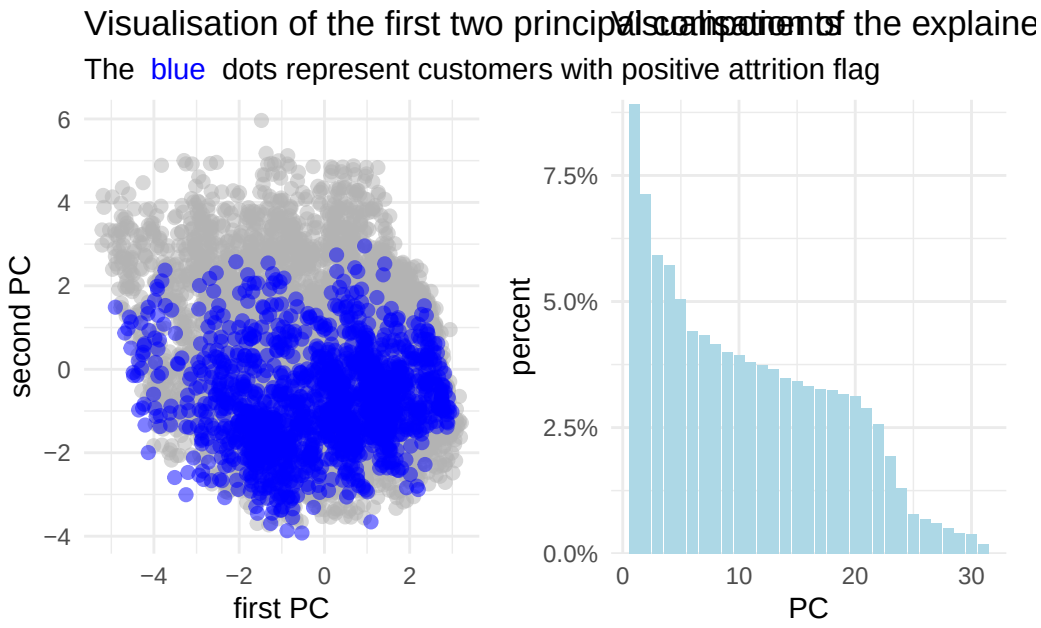
pca_fit_neg <- data_reduced %>%
  mutate(Attrition_Flag = credit_info_prepped$Attrition_Flag) %>%
  filter(Attrition_Flag != "Attrited Customer")

p1 <- pca_fit_neg %>%
  ggplot(aes(.fittedPC1,.fittedPC2)) +
  geom_point(size = 2, color = "gray70", alpha = 0.5)+
  geom_point(data=pca_fit_pos,
             aes(x=.fittedPC1,y=.fittedPC2),
             color="blue",
             size = 2,
             alpha = 0.5)+
  labs(
    title = title_text_p1,
    subtitle = subtitle_text_p1,
    x = "first PC",
    y = "second PC"
    )+
  theme_minimal()+
```

```
  theme(plot.subtitle = element_markdown())


p2 <- pca_fit %>% tidy(matrix = "eigenvalues") %>%
  ggplot(aes(PC, percent)) +
  geom_col(fill = "lightblue") +
  scale_y_continuous(
    labels = scales::percent_format(),
    expand = expansion(mult = c(0, 0.01))
  ) +
  labs(title = title_text_p2)+
  theme_minimal()+
  theme(plot.title = element_markdown())

p1|p2
```



It is evident from the figures above that PCA is not the best choice for this data set. Considering the figure on the left, the general point cloud looks quite "random" and there does not seem to be any sort of trend.

Furthermore, the figure on the right implicates, that less than 20% of the variance in the data set is explained by the first two principal components.

10.1.1.3 Autoencoders

An autoencoder consists of two feed forward neural networks, namely an encoder and decoder. As the names suggest, the encoder applies a mapping to some data which in this case reduces the dimension. On the other hand, the decoder aims to rebuild the encoded data to obtain the data that has originially been passed into the encoder. Figure 10.1 displays a general autoencoder architecture.

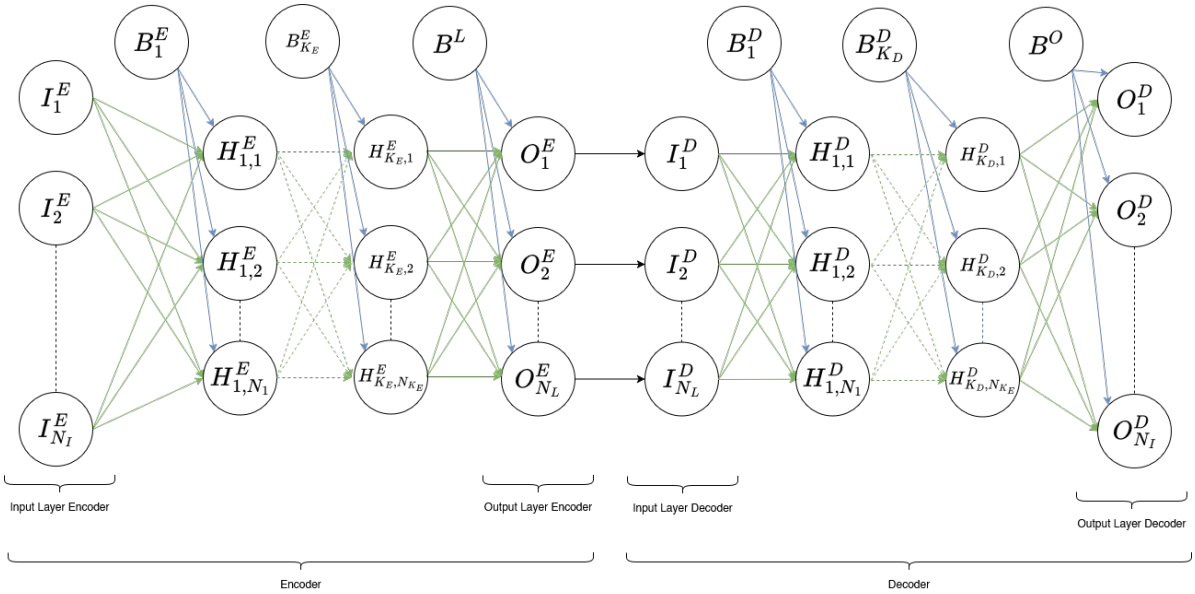


Figure 10.1

On the left hand side, the encoder network is displayed. The input layer I^E consists of N_I neurons that is passed into K_E hidden layers. Each hidden layer $H_1^E, ..., H_{K_E}^E$ consists of $N_1, ..., N_{K_E}$ neurons. The output layer of the encoder consists of N_L neurons, where L denotes the dimension of the (feature) latent space. It is important that $L < N_I$ to ensure that the data is indeed compressed. The output O^E of the encoder is directly passed into the input layer I^D. Note that the dimension of the encoder output is equal to the dimension of the decoder input and that there are no cross connection between the neurons, meaning that O_i^E only maps to I_i^D. The architecture of the decoder is the same as the architecture as the encoder but mirrored.

In simple terms, an autoencoder are two neural networks stitchted together, where the output approximates the input and the section where the networks are stitched together has fewer neurons than the input/output layers.

10.1.1.4 Autoencoders in R

We can use the `{keras}` and `{tensorflow}` framework to specify an autoencoder.

Similar to the neural networks defined in Session 09, we can create functions that aid us in building the autoencoder.

The encoder network in this example consists of an input layer with dimension `n_imp`, two hidden layers with dimension `h1` and `h2`, activation function layers (ReLU) for the hidden layers, and two dropout layers with dropout probabilities `d1` and `d2` respectively that are applied to the output of the hidden layers.

Instead of returning a `keras_sequential` model, we can also specify a model by passing the input and output into the `keras_model` function.

```
encoder_spec <- function(n_inp,h1,h2,d1,d2,n_out){

  enc_input <- layer_input(shape = n_inp)

  enc_output <- enc_input %>%
    layer_dense(units=h1) %>%
    layer_activation_relu() %>%
    layer_dropout(d1) %>%
    layer_dense(units=h2) %>%
    layer_activation_relu() %>%
    layer_dropout(d2)%>%
    layer_dense(units=n_out)

  keras_model(enc_input, enc_output)

}
```

The decoder specification function has the same input as the encoder specification function. Why we choose to do that can be seen in the auto encoder specification function.

```
decoder_spec <- function(n_input,h1,h2,d1,d2,n_out){

  dec_input <- layer_input(shape = n_input)

  dec_output <- dec_input %>%
    layer_dense(units=h1) %>%
    layer_activation_relu() %>%
    layer_dropout(d1) %>%
    layer_dense(units=h2) %>%
```

```
    layer_activation_relu() %>%
    layer_dropout(d2) %>%
    layer_dense(units=n_out)

  keras_model(dec_input, dec_output)

}
```

The function `aen_spec` specifies an autoencoder with input dimension `n_inp`, **four** hidden layers `(h1,h2,h2,h1)`, and a dense layer specifying the dimension of the latent space `n_latent`. Inside the function, we first specify the input `aen_input` of the autoencoder as an input layer with dimension `n_inp`. Then, we define an encoder using the previously defined `encoder_spec` function. Note, that we defined the encoder using the `<<-` symbol. This is not a typo! The `<<-` operator allows us to access the specified encoder outside the `aen_spec` function once the autoencoder has been trained.

We then specify the decoder architecture where the output dimension (latent dimension) of the encoder is passed as the input dimension of the decoder. Additionally the dimension of the hidden layer are mirrored, meaning that the first hidden layer of the decoder has dimension `h2` and the second hidden layer has the dimension `d1`. When mirroring the dimensions of the hidden layers, we need to adapt the drop out rates accordingly. The output dimension of the decoder is given by the dimension of the input.

The return value of the autoencoder specification is a keras model with input specified by an input layer and output specified as the sequence `aen_input %>% encoder() %>% decoder()`.

```
aen_spec<- function(n_inp,h1,h2,d1,d2,n_latent){

 aen_input <- layer_input(shape = n_inp)

 encoder <<- encoder_spec(n_inp,h1,h2,d1,d2,n_latent)
 decoder <- decoder_spec(n_latent,h2,h1,d2,d1,n_inp)

 aen_output <- aen_input %>%
  encoder() %>%
  decoder()

 keras_model(aen_input, aen_output)

}
```

The training procedure of the autoencoder is the same as for any other neural network we previously considered:

1. Create training data.
2. Use the autoencoder specification function to create an untrained autoencoder.
3. Compile the model to prepare it for the training process.
4. Pass the compiled model into the `fit` function to train the model.

```
set.seed(6)
tensorflow::set_random_seed(6)

train_features <- credit_info_prepped %>%
  select(-Attrition_Flag)


aen <- aen_spec(n_inp = ncol(train_features),
                h1 = 256,
                h2 = 128,
                d1 = 0.2,
                d2 = 0,
                n_latent = 2)

ed_nn <- aen %>%
        compile(
        loss = 'mse',
        optimizer = optimizer_adam(1e-4)
      )


history_aen <- ed_nn %>% fit(
      as.matrix(train_features),
      as.matrix(train_features),
      validation_split = 0,
      epochs = 100
    )
```

We can access the trained encoder directly using the obejct name (`encoder`). To create the lower dimensional representation of the data, we have to pass the data set as a matrix into the `predict` function.

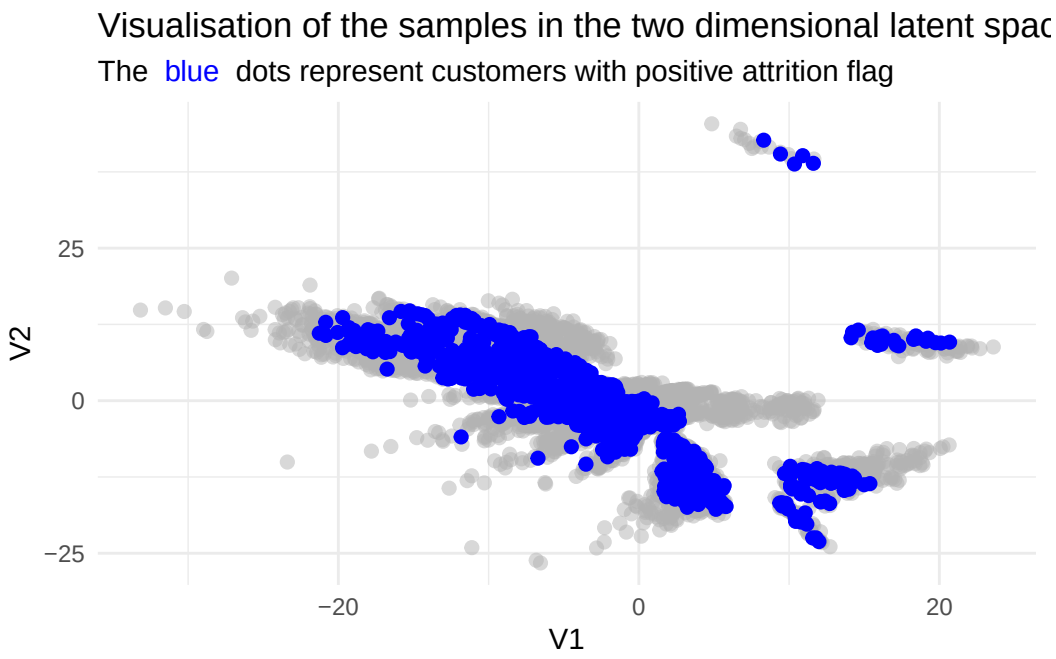
```
data_enc <- encoder %>% predict(
  as.matrix(
    train_features
  )
) %>%
  as_tibble()
```

We can then plot the results in a similar fashion as with PCA:

```
data_enc <- data_enc %>%
  mutate(groups = credit_info$Attrition_Flag)

data_enc_pos <- data_enc %>%
  filter(groups=="Attrited Customer")

data_enc %>%
  filter(groups=="Existing Customer") %>%
  ggplot()+
  geom_point(aes(x=V1,y=V2),
             alpha = 0.5,
             size=2,
             color = "gray70")+
  geom_point(data = data_enc_pos,
             aes(x=V1,y=V2, color = groups),
             size=2,
             color = "blue")+
  labs(title = "Visualisation of the samples in the two dimensional latent space",
       subtitle = subtitle_text_p1)+
  theme_minimal()+
  theme(
    plot.subtitle = element_markdown()
  )
```



While we do not have a direct comparisson in explained variance, we can still see that the resulting lower dimensional data set seems more refined.

10.1.2 Clustering

Using the lower dimensional representations of the data from the previous sections, we can apply some clustering algorithms to find out more about the data.

In simple terms, clustering aims to find subgroups in the data where members are similar to each other. How these subgroups are found is a bit more complex. There are a multitude of algorithms and metrices that can be used to describe how "far" different smaples are apart from each other and how they should be grouped.

In this session, we will consider the *hard* clustering algorithm k-means and density-based clustering algorithm DBSCAN. Both algorithms have their advantages and disadvanteges and as with any other machine learning model we considered so far, it is advisable to apply both algorithms and compare their performance to make a decision on which model to choose.

Evaluating clustering methods is also not trivial. Since there is no target feature like in regression and classification, metrics like MSE or Accuracy do not make sense in this context. To measure how well a clustering algorithm performs, we can describe how homogeneous the clusters are.

The *Silhouette Score* measures the separation distance between clusters by taking values between $[-1, 1]$. If an object is close to its own cluster and far from other clusters, the Silhouette

Score is closer to 1. On the other hand, if an observation has a low score, it is an indicator that the sample does not fit with the assigned cluster. Reasons for poor scores could be too many or too few clusters.

To calculate the silhoutte score for each sample, we first have to calculate the mean distance between the considered sample i and every other data point in the same cluster:

$$a(i) := \frac{1}{|C_I| - 1} \sum_{j \in C_I, i \neq j} d(i, j)$$

Then, we have to calculate the smallest mean distance of the considered sample i to all points in any other cluster:

$$b(i) := \min_{J \neq I} \frac{1}{|C_J|} \sum_{j \in C_J} d(i, j)$$

The silhouette score for a sample i is then defined as:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}},$$

under the assumption that $|C_I| > 1$.

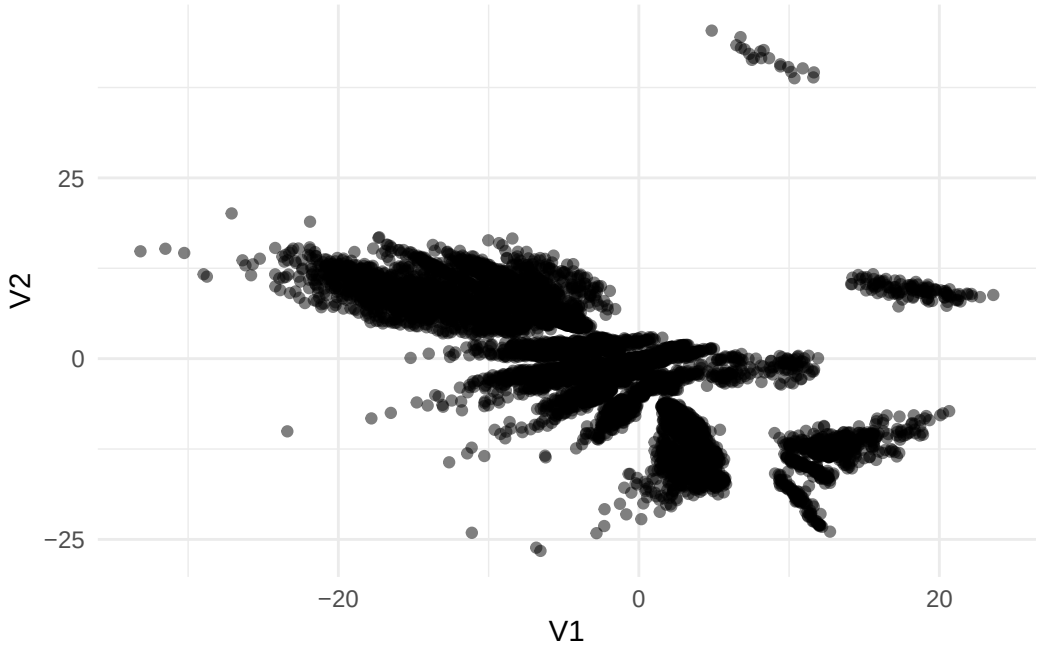
Finding the optimal number of clusters has to be done manually with algorithms like k-means, whereas density based clustering algorithms determine the number of clusters on their own.

The `{tidyclust}` library includes algorithms like k-means and k-medoids. To apply density-based clustering algorithms like DBSCAN, we can use the `{dbscan}` library.

For this introduction, we will only use the low dimensional dataset produced by the autoencoder.

```
data_clust <- data_enc %>%
  select(-groups)

data_clust %>% ggplot(aes(x=V1,y=V2))+
  geom_point(alpha = 0.5, size = 1.5)+
  theme_minimal()
```



10.1.2.1 k-means in R

A k-means model can be trained in `{tidymodels}` fashion using the `{tidyclust}` library.

```
library(tidyclust)
```

To specify and tune a k-means model, the `k_means` function has to be called and the number of clusters has to be set to `tune()`. The engine `"stats"` is the default engine. The `nstart` argument specifies how many times the algorithm should choose random starting values.

```
kmeans_spec <- k_means(num_clusters = tune()) %>%
  set_engine("stats", nstart = 5)

rec_kmeans <- recipe(~., data = data_clust)

folds <- vfold_cv(data_clust)

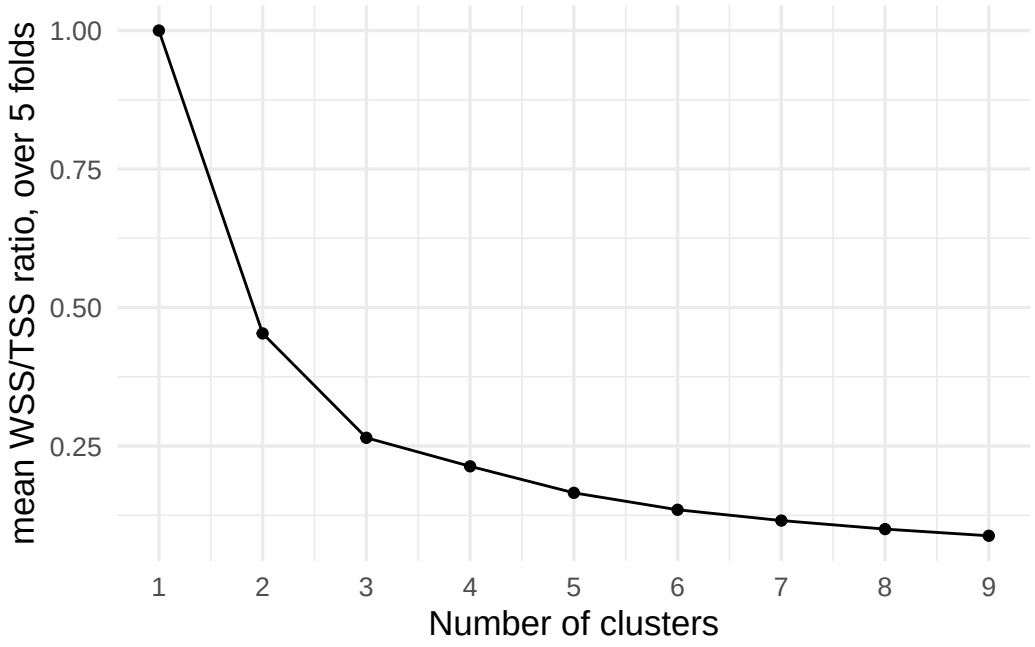
kmeans_wf <- workflow() %>%
  add_recipe(rec_kmeans) %>%
  add_model(kmeans_spec)
```

Instead of using the `tune_grid` function, the `tune_cluster function can be used to tune the number of clusters. As for the metrics, the`sse_ratio‘ function determins the ratio between the within-cluster-sum (wss) and sum of squared errors (sse).

```
tune_res <- kmeans_wf %>%
  tune_cluster(
    grid = 10,
    resamples = folds,
    metrics =cluster_metric_set(sse_ratio)
  )
```

We can plot the `sse_ratio` and determine the optimal number of clusters by the considering each angle enclosed by three points. The point that is enclosed in the smalles angle between two other points can be chosen as a candidate.

```
tune_res %>% collect_metrics() %>%
  filter(.metric == "sse_ratio") %>%
  ggplot(aes(x = num_clusters, y = mean)) +
  geom_point() +
  geom_line() +
  theme_minimal(base_size = 13) +
  ylab("mean WSS/TSS ratio, over 5 folds") +
  xlab("Number of clusters") +
  scale_x_continuous(breaks = 1:10)
```



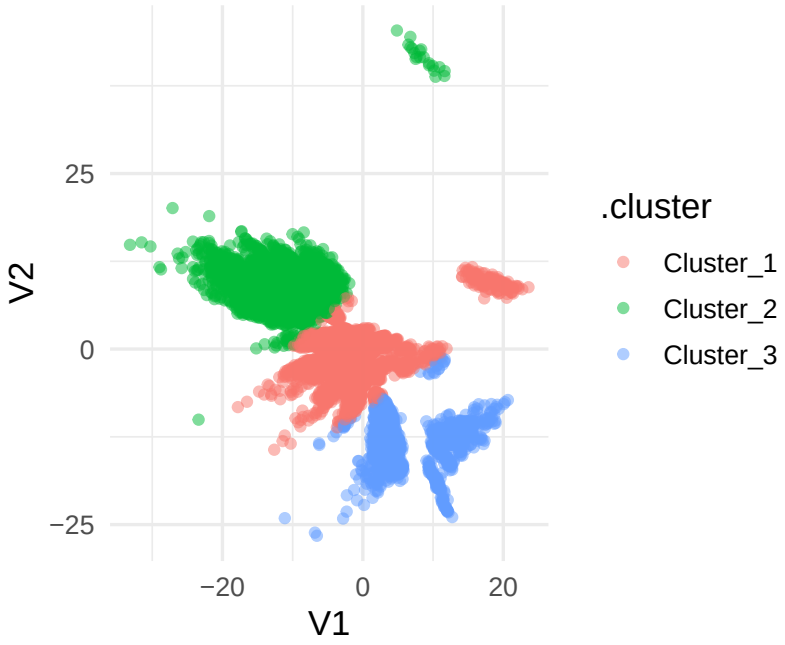
Given the plot above, it is not entirely clear which point is enclosed in the smallest angle. However, based on the information given, we can try `num_clusters = 3`. This is also referred to as the *Elbow Method*

To visualize the clusters, we can use the `extract_cluster_assignment()` function on the final fit and bind the clusters to the data as a new column.

```
kmeans_spec_final <- k_means(num_clusters = 3) %>%
  set_engine("stats", nstart = 5)

res_clust <- kmeans_wf %>%
  update_model(kmeans_spec_final) %>%
  fit( data = data_clust)

res_clust %>%
  extract_cluster_assignment() %>%
  cbind(data_clust) %>%
  ggplot(aes(x=V1,y=V2, color = .cluster)) +
  geom_point(alpha = 0.5,size=1.5)+
  theme_minimal(base_size = 13)+
  coord_equal()
```

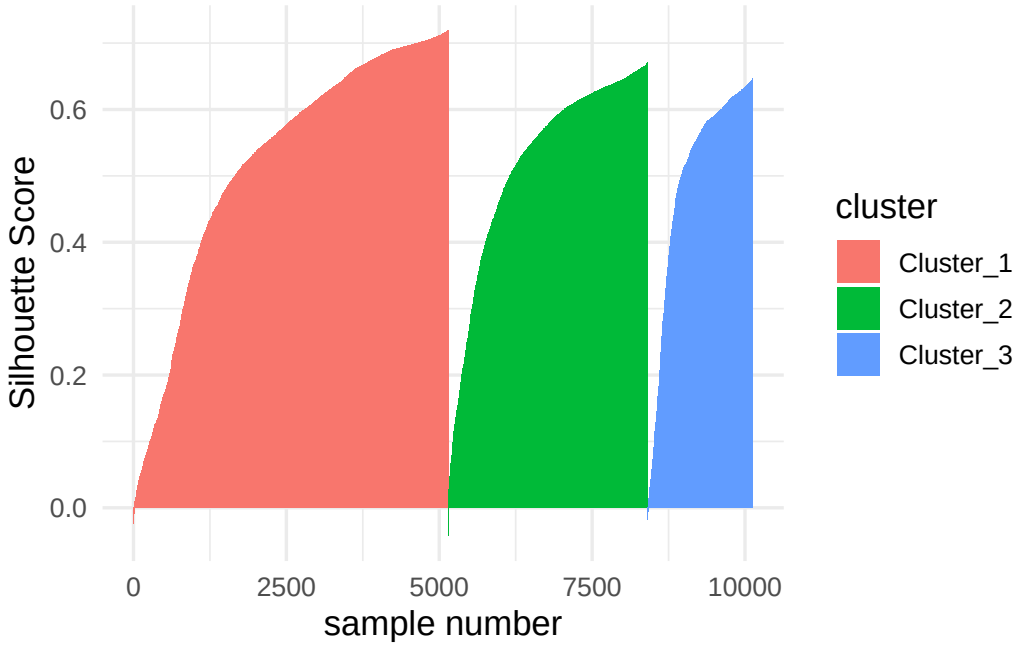


Given the clusters, we could now analyze each cluster on their own, however this is also kept as an exercise for the reader :-).

To build a Silhouette plot, consider the following code snippet:

```
dists <- data_clust %>%
  as.matrix() %>%
  dist()

silhouette(res_clust, dists = dists) %>%
  arrange(cluster,sil_width) %>%
  ggplot(aes(x=1:10127,y=sil_width, fill = cluster))+
  geom_col()+
  labs(x = "sample number",
       y = "Silhouette Score")+
  theme_minimal(base_size = 13)
```



10.1.2.2 DBSCAN in R

Fitting a DBSCAN model in R is also straightforward. The `{dbscan}` library contains functions that can make fitting a DBSCAN model easier. The `{dbscan}` algorithm has unfortunately not yet been incorporated in the `{tidyclust}` framework, so tuning the hyperparameters can be challenging.

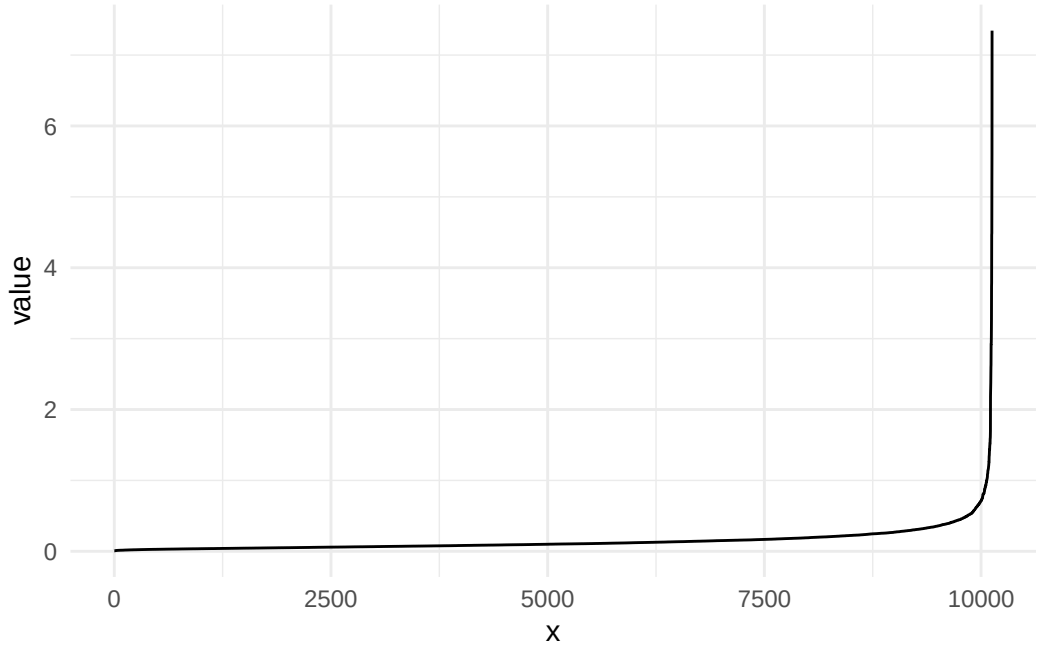
```
library(dbscan)
```

Since the `dbscan()` function requires the parameters `eps` and `minPts` as arguments, choosing their values without relying on tuning requires additional analyses.

The `kNNdist()` function calculates the k-nearest neighbor distances for a given dataset. Choosing $k = 2$ indicates, that the function returns the distance of every point to its k nearest neighbor. A larger distance between points and their respective neighbors indicates that a data point might be an outlier. We can visualize the distances with the following code snippet:

```
kNNdist(data_clust,2) %>%
  as_tibble() %>%
  arrange(value) %>%
  mutate(x=1:nrow(.)) %>%
  ggplot(aes(x=x,y=value))+
```

```
    geom_line()+
    theme_minimal()
```

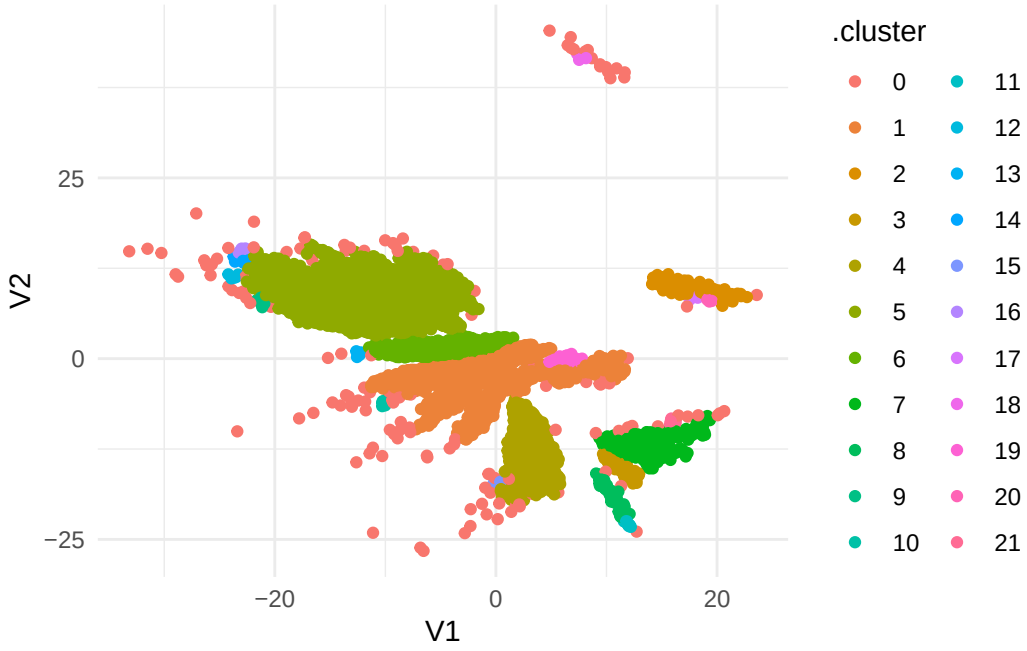


The point of the biggest increase in incline is usually referred to as the *knee-point*. This point can be used for the parameter `eps`. The parameter `eps` describes the radius of the neighborhood of every point.

The second parameter `minPts` describes how many points need to be in the ε neighborhood in order for it to be core point.

```
dbscan_res <- dbscan(data_clust, eps = 0.65, minPts = 4)

dbscan_res %>%
  pluck("cluster") %>%
  as_factor() %>%
  cbind(data_clust) %>%
  as_tibble() %>%
  rename(.cluster=1) %>%
  ggplot(aes(x=V1,y=V2, color = .cluster)) +
  geom_point()+
  theme_minimal()
```



10.2 Exercises

```
library(tidyclust)
```

10.2.1 Dimension Reduction

Exercise 10.1. Given the covariance matrix Σ of a sample $x_1, x_2, x_3 \in \mathbb{R}^3$

$$\Sigma = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix},$$

calculate the principal components of this sample.

Consider the following data set:

```
pen <- palmerpenguins::penguins %>%
  select(-year) %>%
  na.omit()
```

The data set comprises various specific characteristics of a total of $K = 333$ penguins. Calling the `glimpse()` function on the data set yields the following overview:

```
pen %>% glimpse()
```

```
Rows: 333
Columns: 7
$ species           <fct> Adelie, Adelie, Adelie, Adelie, Adelie, Adelie, Adel~
$ island            <fct> Torgersen, Torgersen, Torgersen, Torgersen, Torgerse~
$ bill_length_mm    <dbl> 39.1, 39.5, 40.3, 36.7, 39.3, 38.9, 39.2, 41.1, 38.6~
$ bill_depth_mm     <dbl> 18.7, 17.4, 18.0, 19.3, 20.6, 17.8, 19.6, 17.6, 21.2~
$ flipper_length_mm <int> 181, 186, 195, 193, 190, 181, 195, 182, 191, 198, 18~
$ body_mass_g       <int> 3750, 3800, 3250, 3450, 3650, 3625, 4675, 3200, 3800~
$ sex               <fct> male, female, female, female, male, female, male, fe~
```

Exercise 10.2. Write a recipe that for a clustering pipeline that converts all nominal features into dummy variables and normalizes the underlying data.

Use this recipe to create a preprocessed data set.

Exercise 10.3.

1. Using the functions described in Section 10.1.1.4, train an autoencoder with an architecture of your choice to map the transformed dataset on a two dimensional subspace.
2. Plot the resulting two-dimensional dataset using a scatter plot.

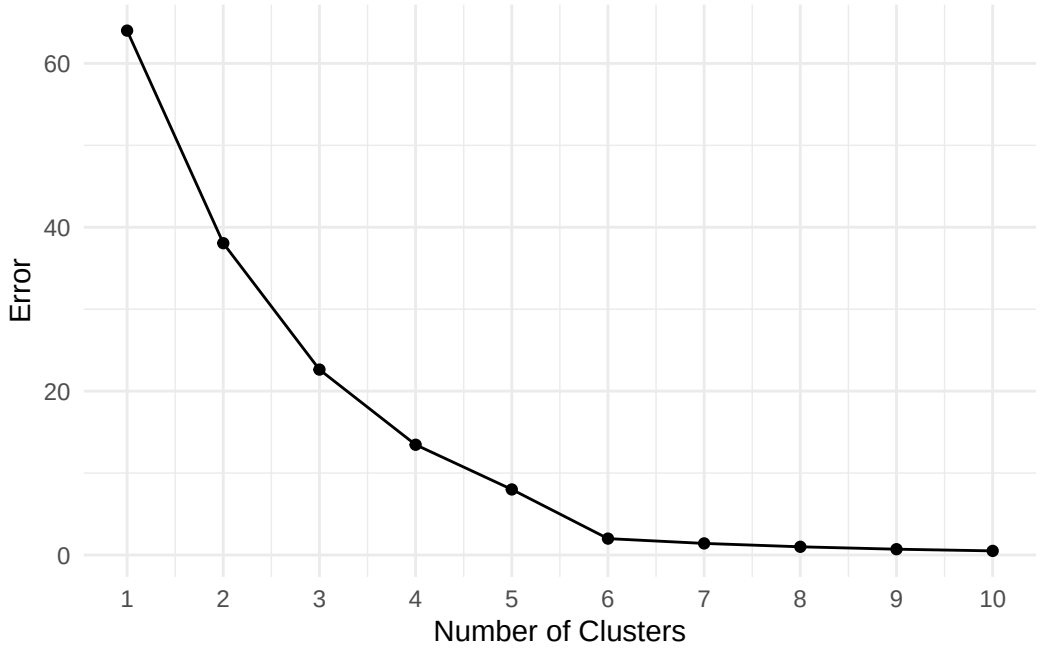
Exercise 10.4.

1. Repeat Exercise 10.3 but instead of training an autoencoder, reduce the dimension using PCA.
2. How many principal components are needed to explain at least 70% of the variance in the dataset?

10.2.2 Clustering

Exercise 10.5. Heuristically, describe the iterative nature of the k-means algorithm.

Exercise 10.6. Suppose you're given the following Scree-Plot. Decide for an optimal number of clusters and justify your choice.



10.3 Solutions

Solution 10.1 (Exercise 10.1). To calculate principal components, we first need to find the eigenvalues of Σ. The eigenvalues λ are the solutions to the characteristic equation $\det(A - \lambda I) = 0$, where I is the identity matrix, and det denotes the determinant.

For the given matrix Σ, the characteristic equation is given by:

$$\det(\Sigma - \lambda I) = \begin{vmatrix} 2-\lambda & 0 & 1 \\ 0 & 2-\lambda & 0 \\ 0 & 0 & 3-\lambda \end{vmatrix} = (2-\lambda)(2-\lambda)(3-\lambda) \tag{10.5}$$

Setting this equation to zero and solving for λ, we find three eigenvalues:

$$\begin{aligned} \lambda_1 &= 2 \\ \lambda_2 &= 2 \\ \lambda_3 &= 3 \end{aligned}$$

Next, we need to find the eigenvectors for each eigenvalue by solving the system of linear equations $(\Sigma - \lambda I)\nu = 0$, where ν is the eigenvector corresponding to eigenvalue λ.

For $\lambda = 2$:

Note, that for $\lambda = 2$ we need two eigenvectors, since the characteristic polynomial has a root of degree two at $\lambda = 2$. By substituting $\lambda = 2$ into $(\Sigma - \lambda I)\nu = 0$ and solving for $\nu_{1,2}$ we obtain:

$$(\Sigma - 2I)\nu = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \nu = 0.$$

The general solution is:

$$\nu_1 = \begin{pmatrix} x_1 \\ x_2 \\ 0 \end{pmatrix} \quad \nu_2 = \begin{pmatrix} x_3 \\ x_4 \\ 0 \end{pmatrix},$$

where x_1, x_2, x_3, x_4 are arbitrary real constants.

For $\lambda = 3$:

Substitute $\lambda = 3$ into $(\Sigma - \lambda I)\nu = 0$ and solve for ν_3:

$$(\Sigma - 3I)\nu_3 = \begin{pmatrix} -1 & 0 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \nu_3 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

The general solution is:

$$\nu_3 = \begin{pmatrix} x_5 \\ 0 \\ x_6 \end{pmatrix}$$

where x_5 and x_6 are arbitrary real constants with $x_5 = x_6$.

We have to ensure, that the vectors are orthonormal, i.e. $\|\nu_i\| = 1$ for $i = 1, 2, 3$ and $\langle \nu_i, \nu_j \rangle = 0$ if $i \neq j$.

Setting $x_1 = ... = x_5 = 1$ and $x_6 = -1$ yields:

$$\nu_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad \nu_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad \nu_3 = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}$$

The given vectors ν_1, ν_2 are already normalized but not orthogonal to vector ν_3, i.e. $\langle \nu_1, \nu_3 \rangle = -1 \neq 0$. To orthogonalize the three vectors, we have to employ an algorithm like Gram-Schmidt. This yields the principal components

$$\tilde{\nu}_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad \tilde{\nu}_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad \tilde{\nu}_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

which are normal and orthogonal to each other.

Solution 10.2 (Exercise 10.2).

```
rec_pen <- recipe(~.,data=pen) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_normalize(all_predictors())

pen_prepped <- rec_pen %>% prep() %>% bake(pen)
```

Solution 10.3 (Exercise 10.3).

1. ```
   set.seed(6)
   tensorflow::set_random_seed(6)

   aen <- aen_spec(n_inp = ncol(pen_prepped),
                   h1 = 256,
                   h2 = 128,
                   d1 = 0.2,
                   d2 = 0,
                   n_latent = 2)

   ed_nn <- aen %>%
           compile(
           loss = 'mse',
           optimizer = optimizer_adam(1e-4)
         )

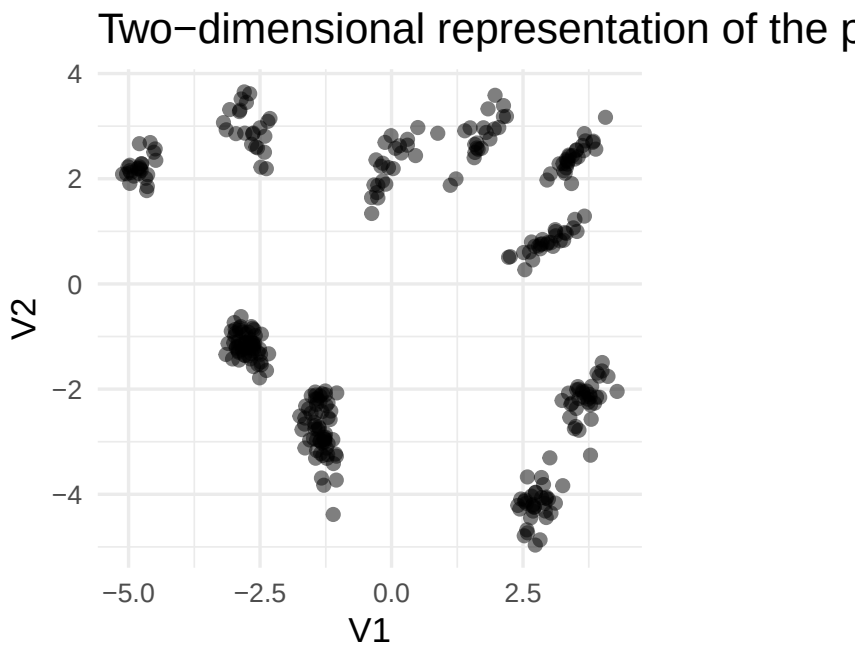

   history_aen <- ed_nn %>% fit(
         as.matrix(pen_prepped),
         as.matrix(pen_prepped),
         validation_split = 0,
   ```
   ```

```
      epochs = 100
    )
```

2.
```
data_enc <- encoder %>% predict(
  as.matrix(
    pen_prepped
  )
) %>%
  as_tibble()
```

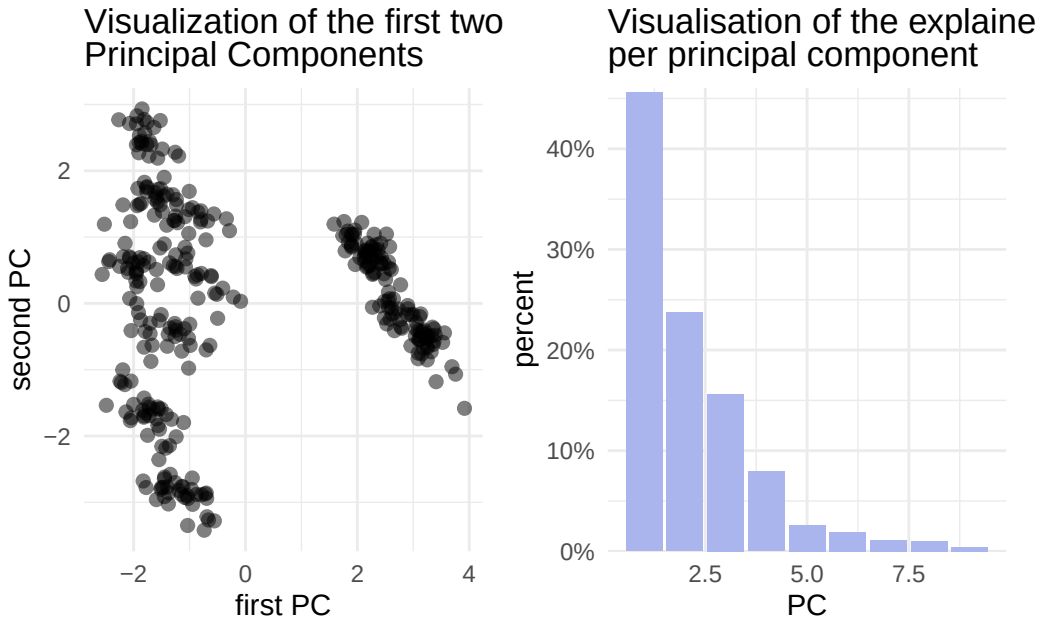
```
11/11 - 0s - 103ms/epoch - 9ms/step
```

```
data_enc %>%
  ggplot(aes(x=V1,y=V2))+
  geom_point(alpha = 0.5, size = 2)+
  labs(
    title = "Two-dimensional representation of the penguins data set"
  )+
  theme_minimal(base_size = 13)+
  coord_fixed()
```



Solution 10.4 (Exercise 10.4).

```r
pca_fit <- pen_prepped %>%
  prcomp()

data_reduced <- pca_fit %>%
  augment(pen_prepped) %>%
  select(c(.fittedPC1,.fittedPC2))


p1 <- data_reduced %>%
  ggplot(aes(.fittedPC1,.fittedPC2)) +
  geom_point(size = 2, alpha = 0.5)+
  labs(
    title = "Visualization of the first two <br> Principal Components",
    x = "first PC",
    y = "second PC"
    )+
  theme_minimal(base_size = 11)+
  theme(plot.title = element_markdown())

p2 <- pca_fit %>% tidy(matrix = "eigenvalues") %>%
  ggplot(aes(PC, percent)) +
  geom_col(fill = "#aab5ee") +
  scale_y_continuous(
    labels = scales::percent_format(),
    expand = expansion(mult = c(0, 0.01))
  ) +
  labs(
    title = "Visualisation of the explained variance <br/>
             per principal component")+
  theme_minimal()+
  theme(plot.title = element_markdown())

p1|p2
```




```
pca_fit %>% tidy(matrix = "eigenvalues")
```

```
# A tibble: 9 x 4
     PC std.dev percent cumulative
  <dbl>   <dbl>   <dbl>      <dbl>
1     1   2.03  0.456        0.456
2     2   1.46  0.237        0.693
3     3   1.19  0.157        0.85
4     4   0.845 0.0794       0.929
5     5   0.488 0.0264       0.956
6     6   0.414 0.0190       0.975
7     7   0.315 0.011        0.986
8     8   0.301 0.0101       0.996
9     9   0.192 0.00409      1
```

To explain at least 70% of the variance, we need to consider the first three principal components.

Solution 10.5 (Exercise 10.5). The iterative nature of the k-means algorithm finds its origin in the calculation of the means in each step: Initially, we set a predetermined number of centers randomly in the dataset and assign points to each center (cluster) for which the distance is minimal. Once every point has been assigned we calculate the mean of each cluster and set it as

the new center. This iterative process is repeated until the algorithm has either converged (no new cluster assignments after a step), or a predetermined number of steps has been performed.

Solution 10.6 (Exercise 10.6). Using the elbow method, we should choose 6 clusters, as the angle enclosing the 5th and 7th cluster at this point is the smallest.