Machine Learning with Tidymodels

R for exploratory data analysis and statistical modeling

Peter Gedeck

2026-01-26

Table of contents

Introduction

In the *DS-6030 Statistical Learning* course, we will use

- the *tidyverse* packages for data loading and processing
- the *tidymodels* packages for model building and validation

Compared to the classical base-R packages covered in *An Introduction to Statistical Learning* (James et al. 2021), these packages offer many advantages that will make working with data easier and more streamlined.

Tidyverse

The *tidyverse* is a collection of packages that share a common design philosophy and are designed to work together. Hadley Wickham outlined the principles of the tidyverse in 2014 in the Tidy Data paper published in the Journal of Statistical Software 59(10), 1–23.

To load the tidyverse, use the following command:

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.2     v tibble    3.3.0
v lubridate 1.9.4     v tidyr     1.3.1
v purrr     1.0.4
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becor
```

You will see that this loads a number of packages. The most important ones are:

- `ggplot2` for plotting
- `dplyr` for data manipulation

- `readr` for data import
- `tibble` for improved data frames
- `tidyr` for getting data into tidy form
- `purrr` for functional programming
- `stringr` for string manipulation
- `forcats` for categorical/factor data

Tidymodels

The *tidymodels* package was first released in 2018 and is still under active development and maintained by the company Posit as an open source project. It is an ecosystem of packages that share a common design philosophy and are designed to work together. The packages include

- `parsnip` for model specification
- `recipes` for data preprocessing
- `rsample` for resampling
- `yardstick` for model evaluation
- `tune` for hyperparameter tuning
- `workflows` for modeling workflows
- `tidyposterior` for Bayesian modeling

The *tidymodels* packages are designed to work with the *tidyverse* and *tidydata* principles. The packages are designed to be modular and extensible.

Getting Help

- A good source of basic data analysis using R is found in the free book R for Data Science (2e) by Wickham et al. (Wickham, Çetinkaya-Rundel, and Grolemund 2023).
- Web search, especially stackoverflow.com and stats.stackexchange.com
- Troubleshooting/Debugging.
  - Check one line of code at a time.
  - Google your error message
  - Use scripts

RStudio

- Install R and RStudio
- Make use of Projects in RStudio

Part I

Exploratory data analysis

1 Data loading and cleaning

The objective of exploratory data analysis is to understand your data, find patterns, identify outliers, and possibly form hypotheses for further analysis. In this chapter, we will use the `penguins` dataset to learn how to load data, inspect it, and process it by removing missing values and outliers. The following Chapter 2 will cover more details on data transformation. Chapter 3 will cover data visualization.

Let's start with loading the *tidyverse* packages:

```
library(tidyverse)
library(DT)
```

1.1 Load data

We use the function `readr::read_csv` to load the data. This function and its variations like `readr::read_delim` are able to read a wide variety of formats (e.g. CSV, TSV, Excel). Here, we read a compressed file in comma separated (*csv*) format.

```
penguins <- readr::read_csv("data/penguins_modified.csv.gz")
```

```
Rows: 347 Columns: 7
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (3): species, island, sex
dbl (4): bill_length_mm, bill_depth_mm, flipper_length_mm, body_mass_g

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

We learn that the data set contains 347 observations with 7 columns.[1] Three of the columns are string (*chr*: `species`, `island`, `sex`), the rest are numeric (*dbl*: `bill_length_mm`, `bill_depth_mm`, `flipper_length_mm`, `body_mass_g`). It is useful to know the data types of

[1] Note: we use a modified version of the original dataset here

the imported data, because they determine what operations can be performed on the data. However, the extensive output will clutter the document. You can suppress the output by adding the argument `show_col_types = FALSE` to the `read_csv` function:

```
penguins <- readr::read_csv("data/penguins_modified.csv.gz",
  show_col_types = FALSE)
```

Note that the command `read_csv` is spelt similarly to the base-R function `read.csv`. To avoid confusion, we use the notation `readr::read_csv` to indicate that we use the function `read_csv` from the package `readr`.

> **Todo**
>
> Read the help page for `readr::read_csv` and find out what the arguments `file`, `delim`, `comment`, and `skip` do.

1.2 Inspect data to identify missing values

The data is now stored in the variable `penguins` as a *tibble*. A *tibble* is the tidyverse form of a data frame. It is a bit more strict than a data frame, but also more consistent. Let's have a look at the first few rows of the data:

```
penguins
```

```
# A tibble: 347 x 7
   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
   <chr>   <chr>              <dbl>         <dbl>             <dbl>       <dbl>
 1 Adelie  Torgersen           39.1          18.7               181        3750
 2 Adelie  Torgersen           39.5          17.4               186        3800
 3 Adelie  Torgersen           40.3          18                 195        3250
 4 Adelie  Torgersen           NA            NA                  NA          NA
 5 Adelie  Torgersen           36.7          19.3               193        3450
 6 Adelie  Torgersen           39.3          20.6               190        3650
 7 Adelie  Torgersen           38.9          17.8               181        3625
 8 Adelie  Torgersen           39.2          19.6               195        4675
 9 Adelie  Torgersen           34.1          18.1               193        3475
10 Adelie  Torgersen           42            20.2               190        4250
# i 337 more rows
# i 1 more variable: sex <chr>
```

Compared to standard data frames, this is a more informative representation of your data. It shows the first few rows, the column names, and the data types of the columns. The data types are important, because they determine what you can do with the data.

A useful first step is to check for missing data. We can already see in the output above, that the dataset contains missing values indicated by `NA`. There are various ways of doing this. One way is to use the function `is.na` to check for missing values in each column:

```
colSums(is.na(penguins))
```

```
          species            island    bill_length_mm     bill_depth_mm
                0                 0                 2                 2
flipper_length_mm       body_mass_g               sex
                2                 2                11
```

Several columns contain missing values. Here are the rows with missing values:

```
penguins[rowSums(is.na(penguins)) > 0, ]
```

```
# A tibble: 11 x 7
   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
   <chr>   <chr>              <dbl>         <dbl>             <dbl>       <dbl>
 1 Adelie  Torgersen           NA            NA                  NA          NA
 2 Adelie  Torgersen           34.1          18.1               193        3475
 3 Adelie  Torgersen           42            20.2               190        4250
 4 Adelie  Torgersen           37.8          17.1               186        3300
 5 Adelie  Torgersen           37.8          17.3               180        3700
 6 Adelie  Dream               37.5          18.9               179        2975
 7 Gentoo  Biscoe              44.5          14.3               216        4100
 8 Gentoo  Biscoe              46.2          14.4               214        4650
 9 Gentoo  Biscoe              47.3          13.8               216        4725
10 Gentoo  Biscoe              44.5          15.7               217        4875
11 Gentoo  Biscoe              NA            NA                  NA          NA
# i 1 more variable: sex <chr>
```

With just a few missing values in the dataset, we can simply remove them. This is done with the function `drop_na`.

```
df <- penguins %>%
  drop_na()
```

The pipe operator `%>%` (pronounced as then) is used to chain commands together. The command `penguins %>% drop_na()` takes the data frame `penguins` and passes it to the function `drop_na`. The statement is equivalent to:

```
df <- drop_na(penguins)
```

The real power of the pipe operator becomes apparent when we combined multiple commands in a processing pipeline. The function `drop_na` removes all rows that contain at least one missing value. You can also specify which columns to check. This restricts the check to specific columns. For example, the following command removes all rows that contain missing values in the columns `bill_length_mm` and `bill_depth_mm`:

```
df <- penguins %>%
  drop_na(bill_length_mm, bill_depth_mm)
```

1.3 Identifying outliers

The tidyverse comes with a powerful plotting package called `ggplot2`. We will use the `ggplot2` package throughout this class. To get started, we look at the values of the numerical columns to get a feel for the data. Here is a graph (Figure 1.1) of the values of the `bill_length_mm` column:

```
df <- penguins %>%
  drop_na() %>%
  mutate(id = row_number())
ggplot(df, aes(x = id, y = bill_length_mm)) + geom_line()
```

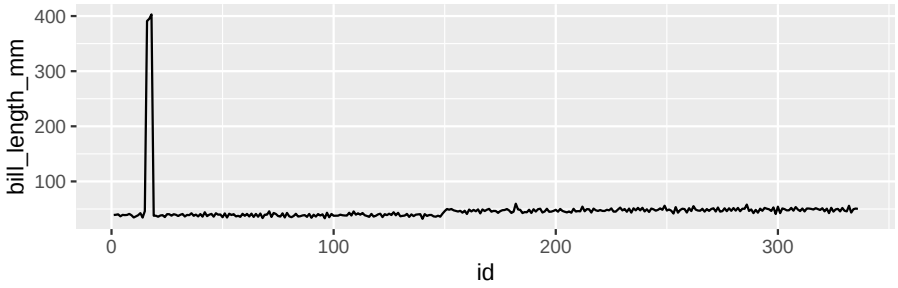


Figure 1.1: Values of the `bill_length_mm` column.

We first create a temporary *tibble* `df` that contains the `id` column and the `bill_length_mm` column. The `id` column is just a sequence of numbers from 1 to 347. We use this column to

plot the values of `bill_length_mm` against the row number. The `geom_line` function adds a line plot to the plot. The result is a line plot of the values of `bill_length_mm` against the row number. We can see that the values are mostly between 30 and 60, but there are some outliers. The values are:

```
penguins %>%
  mutate(id = row_number(), .before = species) %>%
  slice(19:25) %>%
  datatable(rownames = FALSE)
```

file:////private/var/folders/_8/ms0ft4913k3290v7f0g_yfpc0000gn/T/Rtmpq5jbxi/file15c8e7558f57

Show 10 entries Search:

id	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
19	Adelie	Torgersen	34.4	18.4	184	3325	female
20	Adelie	Torgersen	46	21.5	194	4200	male
21	Adelie	Torgersen	391	18.7	181	3750	male
22	Adelie	Torgersen	395	17.4	186	3800	female
23	Adelie	Torgersen	403	18	195	3250	female
24	Adelie	Biscoe	37.8	18.3	174	3400	female
25	Adelie	Biscoe	37.7	18.7	180	3600	male

Showing 1 to 7 of 7 entries Previous 1 Next

We can see that rows 21, 22, and 23 have values that are around 10 times higher than all other values. It's likely that these values are due to an error in the data collection process. We can remove these values with the following command:

```
df <- penguins %>%
  drop_na() %>%
  filter(bill_length_mm < 100)
```

Todo

Check if any of the other numerical columns contain outliers.

1.4 Summary

In this chapter, we have learned how to load data, inspect it, and process it. We have also learned how to remove missing values and outliers.

The analysis of this chapter results in the following processing pipeline:

```
filename <- "data/penguins_modified.csv.gz"
penguins <- readr::read_csv(filename) %>%
  drop_na() %>%
  filter(bill_length_mm < 100)
```

```
Rows: 347 Columns: 7
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (3): species, island, sex
dbl (4): bill_length_mm, bill_depth_mm, flipper_length_mm, body_mass_g

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

> **i** Further information
>
> The data import cheatsheet is a short summary of all the main features of `readr` and `readxl`. For more details see https://readr.tidyverse.org/. The `readxl` package allows to import Excel files, see https://readxl.tidyverse.org/.

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE, fig.align = "center")
library(tidyverse)
library(DT)
penguins <- readr::read_csv("data/penguins_modified.csv.gz")
penguins <- readr::read_csv("data/penguins_modified.csv.gz",
  show_col_types = FALSE)
penguins
colSums(is.na(penguins))
penguins[rowSums(is.na(penguins)) > 0, ]
```

```r
df <- penguins %>%
  drop_na()
df <- penguins %>%
  drop_na(bill_length_mm, bill_depth_mm)
df <- penguins %>%
  drop_na() %>%
  mutate(id = row_number())
ggplot(df, aes(x = id, y = bill_length_mm)) + geom_line()
penguins %>%
  mutate(id = row_number(), .before = species) %>%
  slice(19:25) %>%
  datatable(rownames = FALSE)
df <- penguins %>%
  drop_na() %>%
  filter(bill_length_mm < 100)
filename <- "data/penguins_modified.csv.gz"
penguins <- readr::read_csv(filename) %>%
  drop_na() %>%
  filter(bill_length_mm < 100)
```

2 Manipulating data

The `dplyr` package is the main component of the tidyverse for data manipulation. You can load it either by loading the tidyverse package or by loading dplyr directly.

```
library(tidyverse)    # or library(dplyr)
```

2.1 Example: manipulating flights data

We've already seen some of the basic functions in Chapter 1. Let's look at a more elaborate example using a dataset from the `nycflights13` package. This package contains information about all flights that departed from NYC (e.g., EWR, JFK and LGA) in 2013. The dataset is quite large and includes information on delays, airlines, airports, weather, and planes. It contains 336,776 rows and 19 columns.

We first load the `nycflights13` package to get the `flights` dataset.

```
library(nycflights13) # load flight data
dim(flights)
```

```
[1] 336776     19
```

For our analysis, we want to focus on flights that were less than 1000 miles (distance) and restrict the dataset to keep only the columns: `dep_delay`, `arr_delay`, `origin`, `dest`, `air_time`, and `distance`. In addition, we convert the departure and arrival delays into hours, and calculate the average flight speed (in mph). We also want to add a new column with the Z-score for departure delays. Finally, we want to order by average flight speed (fastest to slowest) and return the first four rows.

This is the full data processing pipeline:

```r
df <- flights %>%
  # step 1
  filter(distance < 1000) %>%
  # step 2
  select(dep_delay, arr_delay, origin, dest, air_time, distance) %>%
  # step 3
  mutate(
    Z_dep_delay = (dep_delay - mean(dep_delay, na.rm = TRUE)) /
      sd(dep_delay, na.rm = TRUE),
    dep_delay = dep_delay / 60,
    arr_delay = arr_delay / 60,
    speed = distance / (air_time / 60)
  ) %>%
  # step 4
  arrange(-speed) %>%
  # step 5
  print(n = 4)
```

```
# A tibble: 189,671 x 8
  dep_delay arr_delay origin dest  air_time distance Z_dep_delay speed
      <dbl>     <dbl> <chr>  <chr>    <dbl>    <dbl>       <dbl> <dbl>
1    0.15     -0.233  LGA    ATL         65      762     -0.108   703.
2    0.25     -0.0167 EWR    GSP         55      594      0.0355  648
3    0.0667    0.0333 EWR    BNA         70      748     -0.228   641.
4    0.267    -0.367  EWR    CVG         62      569      0.0595  551.
# i 189,667 more rows
```

Step 1: The `filter` function affects the rows of the dataframe. Here, it keeps only flights with a `distance` less than 1000 miles. The `dplyr` package allows us to use the column names without quotes and without the table name.

Step 2: `select` affects the columns of the dataframe. Here, it keeps the listed six columns. The select statement accepts a number of helper functions to select columns based on their names. For example, `starts_with("delay")` selects all columns that start with the word "delay". The `:` operator is used to select a range of columns. Using this functionality, we could have written the `select` statement also as:

```r
    select(ends_with("delay"), origin:distance) %>%
```

Step 3: This steps modifies columns and adds new ones. The first assignment, creates the new column `Z_dep_delay` which is the Z-score of the departure delay. The second and third

assignments convert the departure and arrival delays from minutes to hours. The fourth assignment calculates the average flight speed in miles per hour.

Step 4: The `arrange` function is used to order the rows by speed. The `-` sign is used to indicate descending order.

Step 5: Finally, we print the first 4 rows. This serves only for information and doesn't change the `tibble` which is assigned to the variable `df`.

2.2 Overview of `dplyr` functionality

We've seen some of the basic functions in the example above. In most cases, these will be sufficient for your data manipulation needs. It is however useful to have an understanding of the whole package.

> Todo
>
> Visit the `dplyr` website (https://dplyr.tidyverse.org/) and read the *Get started* section. This will give you a good overview of the package.

The functionality can be organized into several categories.

2.2.1 Sorting data

The function `arrange()` is used to sort the data. Here are a few examples:

```
df <- tibble(a = c(1, 3, 2, 1, 3, 2), b = c(4, 4, 5, 5, 6, 6))
df %>% arrange(a)
```

```
# A tibble: 6 x 2
      a     b
  <dbl> <dbl>
1     1     4
2     1     5
3     2     5
4     2     6
5     3     4
6     3     6
```

```
df %>% arrange(b, a)
```

```
# A tibble: 6 x 2
      a     b
  <dbl> <dbl>
1     1     4
2     3     4
3     1     5
4     2     5
5     2     6
6     3     6
```

```
df %>% arrange(desc(b), a)
```

```
# A tibble: 6 x 2
      a     b
  <dbl> <dbl>
1     2     6
2     3     6
3     1     5
4     2     5
5     1     4
6     3     4
```

```
df %>% arrange(-a, b)
```

```
# A tibble: 6 x 2
      a     b
  <dbl> <dbl>
1     3     4
2     3     6
3     2     5
4     2     6
5     1     4
6     1     5
```

The examples show that we can sort by one or more columns. The default is ascending order. To sort in descending order, we can use the `desc()` function or the `-` sign. Multiple columns are sorted in the order they are listed. For example, `-a, b` first sorts by column `a` in descending order, then for rows with the same value in column `a`, it sorts by column `b` in ascending order.

2.2.2 Selecting columns

The function `select()` is used to select columns. It can be used to keep only certain columns or to drop columns. The `select()` function accepts a number of helper functions to select columns based on their names. Here are a few examples:

```
flights %>%
  colnames()
```

```
 [1] "year"           "month"          "day"            "dep_time"
 [5] "sched_dep_time" "dep_delay"      "arr_time"       "sched_arr_time"
 [9] "arr_delay"      "carrier"        "flight"         "tailnum"
[13] "origin"         "dest"           "air_time"       "distance"
[17] "hour"           "minute"         "time_hour"
```

```
# select by part of name:
flights %>%
  select(starts_with("arr")) %>%
  colnames()
```

```
[1] "arr_time"  "arr_delay"
```

```
flights %>%
  select(contains("time")) %>%
  colnames()
```

```
[1] "dep_time"       "sched_dep_time" "arr_time"       "sched_arr_time"
[5] "air_time"       "time_hour"
```

```
flights %>%
  select(ends_with("delay")) %>%
  colnames()
```

```
[1] "dep_delay" "arr_delay"
```

```
# using regular expressions
flights %>%
  select(matches("(time|hour|minute)")) %>%
  colnames()
```

```
[1] "dep_time"       "sched_dep_time" "arr_time"       "sched_arr_time"
[5] "air_time"       "hour"           "minute"         "time_hour"
```

```
# select columns using a vector of column names (all_of or any_of)
variables <- c("dep_delay", "arr_delay", "origin", "dest",
  "air_time", "distance")
flights %>%
  select(all_of(variables)) %>%
  colnames()
```

```
[1] "dep_delay" "arr_delay" "origin"    "dest"      "air_time"  "distance"
```

```
# remove columns by name:
flights %>%
  select(-c(carrier, flight, tailnum)) %>%
  colnames()
```

```
 [1] "year"           "month"          "day"            "dep_time"
 [5] "sched_dep_time" "dep_delay"      "arr_time"       "sched_arr_time"
 [9] "arr_delay"      "origin"         "dest"           "air_time"
[13] "distance"       "hour"           "minute"         "time_hour"
```

```
flights %>%
  select(! matches("(time|hour|minute)")) %>%
  colnames()
```

```
 [1] "year"      "month"     "day"       "dep_delay" "arr_delay" "carrier"
 [7] "flight"    "tailnum"   "origin"    "dest"      "distance"
```

2.2.3 Filtering rows

The function `filter()` is used to filter rows. It keeps only rows that satisfy the condition. Here are a few examples:

```
df <- tibble(
  row = c(1:7),
  a = c(1, 2, 3, 1, 3, 1, NA),
  b = c(4, 5, 6, 4, 5, 6, 8)
)
```

```r
# filter by column values
df %>%
  filter(a == 1)
```

```
# A tibble: 3 x 3
    row     a     b
  <int> <dbl> <dbl>
1     1     1     4
2     4     1     4
3     6     1     6
```

```r
df %>%
  filter(a == 1 & b == 4)
```

```
# A tibble: 2 x 3
    row     a     b
  <int> <dbl> <dbl>
1     1     1     4
2     4     1     4
```

```r
df %>%
  filter(a > 2 | b == 4)
```

```
# A tibble: 4 x 3
    row     a     b
  <int> <dbl> <dbl>
1     1     1     4
2     3     3     6
3     4     1     4
4     5     3     5
```

```r
df %>%
  filter(a %in% c(1, 3))
```

```
# A tibble: 5 x 3
    row     a     b
  <int> <dbl> <dbl>
1     1     1     4
2     3     3     6
```

```
3     4     1     4
4     5     3     5
5     6     1     6
```

```
# remove by column values - use the ! operator
df %>%
  filter(! a %in% c(1, 3))
```

```
# A tibble: 2 x 3
    row     a     b
  <int> <dbl> <dbl>
1     2     2     5
2     7    NA     8
```

```
# filter to missing values in column
df %>%
  filter(is.na(a))
```

```
# A tibble: 1 x 3
    row     a     b
  <int> <dbl> <dbl>
1     7    NA     8
```

`distinct` removes duplicate rows:

```
df %>%
  distinct(a, b, .keep_all = TRUE)
```

```
# A tibble: 6 x 3
    row     a     b
  <int> <dbl> <dbl>
1     1     1     4
2     2     2     5
3     3     3     6
4     5     3     5
5     6     1     6
6     7    NA     8
```

It's also possible to select rows using the row number. The function `slice()` is used for this purpose.

```r
df %>%
  slice(1:3)
```

```
# A tibble: 3 x 3
    row     a     b
  <int> <dbl> <dbl>
1     1     1     4
2     2     2     5
3     3     3     6
```

```r
df %>%
  slice(1, 3, 5)
```

```
# A tibble: 3 x 3
    row     a     b
  <int> <dbl> <dbl>
1     1     1     4
2     3     3     6
3     5     3     5
```

```r
idx <- c(2, 4, 6)
df %>%
  slice(idx)
```

```
# A tibble: 3 x 3
    row     a     b
  <int> <dbl> <dbl>
1     2     2     5
2     4     1     4
3     6     1     6
```

`slice_min` and `slice_max` are useful if you want to identify the rows with the smallest or largest values in a column.

```r
df %>%
  slice_min(a)
```

```
# A tibble: 3 x 3
    row     a     b
```

```
  <int> <dbl> <dbl>
1     1     1     4
2     4     1     4
3     6     1     6
```

```
df %>%
  slice_max(a, n = 2)
```

```
# A tibble: 2 x 3
    row     a     b
  <int> <dbl> <dbl>
1     3     3     6
2     5     3     5
```

`slice_sample` selects a random sample of rows.

```
df %>%
  slice_sample(n = 5)  # sample without replacement
```

```
# A tibble: 5 x 3
    row     a     b
  <int> <dbl> <dbl>
1     3     3     6
2     6     1     6
3     2     2     5
4     4     1     4
5     1     1     4
```

```
df %>%
  slice_sample(n = 5, replace = TRUE)  # sample with replacement
```

```
# A tibble: 5 x 3
    row     a     b
  <int> <dbl> <dbl>
1     6     1     6
2     6     1     6
3     5     3     5
4     3     3     6
5     6     1     6
```

2.2.4 Modifying tables

We've already seen the `mutate` function that is used to add new columns or modify existing ones.

```
df <- tibble(
  a = c(1, 2, 3),
  b = c(4, 5, 6)
)
# Add one or more new columns
df %>%
  mutate(c = a + b)
```

```
# A tibble: 3 x 3
      a     b     c
  <dbl> <dbl> <dbl>
1     1     4     5
2     2     5     7
3     3     6     9
```

```
df %>%
  mutate(c = a + b, d = a - b, e = c * d)
```

```
# A tibble: 3 x 5
      a     b     c     d     e
  <dbl> <dbl> <dbl> <dbl> <dbl>
1     1     4     5    -3   -15
2     2     5     7    -3   -21
3     3     6     9    -3   -27
```

The last example shows that the new columns (`e`) can be based on previously created columns (`c` and `d`).

```
# Modify existing columns
df %>%
  mutate(a = a + 1)
```

```
# A tibble: 3 x 2
      a     b
  <dbl> <dbl>
```

```
1     2     4
2     3     5
3     4     6
```

If a column name already exists, the original column will be replaced. To avoid this, use the `add_column` function. It will throw an error if the column name exists. The `add_column` function can also be used to add a column in a specific position.

```
df %>%
  add_column(idx = seq_len(nrow(df)))
```

```
# A tibble: 3 x 3
      a     b   idx
  <dbl> <dbl> <int>
1     1     4     1
2     2     5     2
3     3     6     3
```

```
df %>%
  add_column(idx = seq_len(nrow(df)), .before = "a")
```

```
# A tibble: 3 x 3
    idx     a     b
  <int> <dbl> <dbl>
1     1     1     4
2     2     2     5
3     3     3     6
```

`add_row` adds new rows to the table. We can again use `.before` and `.after` to specify the position.

```
df %>%
  add_row(a = 4, b = 7)
```

```
# A tibble: 4 x 2
      a     b
  <dbl> <dbl>
1     1     4
2     2     5
3     3     6
4     4     7
```

```r
df %>%
  add_row(a = 4, b = 7, .after = 1)
```

```
# A tibble: 4 x 2
      a     b
  <dbl> <dbl>
1     1     4
2     4     7
3     2     5
4     3     6
```

2.3 Handling missing values

Real life data are often messy and contain missing values. The tidyverse packages `dplyr` and `tidyr` have functions that help dealing with this. In datasets, missing values are not always identified using NA. Sometimes, they are identified using a placeholder value, e.g. -9999.

2.3.1 Convert placeholders to missing values

Here is an example of a dataset with placeholder values that get converted to missing values.

```r
df <- tibble(
  a = c(1, 2, 3, -9999),
  b = c(4, -9999, 6, 8)
)
df <- df %>%
  mutate_all(na_if, -9999) %>%
  print()
```

```
# A tibble: 4 x 2
      a     b
  <dbl> <dbl>
1     1     4
2     2    NA
3     3     6
4    NA     8
```

2.3.2 Remove or replace missing values

The `tidyr` package has quite a few functions for handling missing values. We already encountered the `drop_na` function to remove rows with missing values.

```
df <- tibble(
  a = c(1, 2, 3, NA),
  b = c(4, NA, 6, 8)
)
df %>%
  drop_na()  # drop rows with missing values
```

```
# A tibble: 2 x 2
      a     b
  <dbl> <dbl>
1     1     4
2     3     6
```

```
df %>%
  drop_na(a)  # drop rows in which column "a" has missing values
```

```
# A tibble: 3 x 2
      a     b
  <dbl> <dbl>
1     1     4
2     2    NA
3     3     6
```

The `replace_na` function replaces missing values with a specified value.

```
df %>%
  replace_na(list(a = 0, b = 99))
```

```
# A tibble: 4 x 2
      a     b
  <dbl> <dbl>
1     1     4
2     2    99
3     3     6
4     0     8
```

To replace the missing values with the mean of each column, we can use:

```
df %>%
  replace_na(as.list(colMeans(df, na.rm = TRUE)))
```

```
# A tibble: 4 x 2
      a     b
  <dbl> <dbl>
1     1     4
2     2     6
3     3     6
4     2     8
```

2.4 Split - Apply - Combine

The `dplyr` operations are even more powerful when they can be used with grouping variables. This approach to data processing is also known as *Split - Apply - Combine* (Wickham 2011).

- `Split`: The data is split into groups based on one or more grouping variables.
- `Apply`: A function is applied to each group independently.
- `Combine`: The results are combined into a new data structure.

2.4.1 Two examples

Let's see this approach in action using the `flights` dataset. We want to calculate the average departure delay for each airline and sort the resulting table by the average delay.

```
flights %>%
  group_by(carrier) %>%
  summarize(avg_dep_delay = mean(dep_delay, na.rm = TRUE)) %>%
  arrange(avg_dep_delay)
```

```
# A tibble: 16 x 2
   carrier avg_dep_delay
   <chr>           <dbl>
 1 US               3.78
 2 HA               4.90
 3 AS               5.80
 4 AA               8.59
 5 DL               9.26
```

```
 6 MQ              10.6
 7 UA              12.1
 8 OO              12.6
 9 VX              12.9
10 B6              13.0
11 9E              16.7
12 WN              17.7
13 FL              18.7
14 YV              19.0
15 EV              20.0
16 F9              20.2
```

Step 1: The `group_by` function is used to specify the grouping variable(s). In this case, we group by the `carrier` variable. Note: grouping should to be applied on discrete variables (categorical, factor, or maybe integer valued columns).

Step 2: This step is applied to the individual groups. The `summarize` function is used to calculate the average departure delay for each group. The `na.rm = TRUE` argument is used to ignore missing values. Following the summarization, the data is combined into a single tibble. It contains only the grouping variable(s) and the new summary variable(s).

Step 3: We can now continue with normal processing. The `arrange` function is used to sort the resulting table by the average delay.

We can group by multiple variables. For example,

```
flights %>%
  group_by(origin, dest) %>%   # group by both origin and dest
  summarise(
    max.delay = max(arr_delay, na.rm = TRUE),
    avg.delay = mean(arr_delay, na.rm = TRUE),
    min.delay = min(arr_delay, na.rm = TRUE),
    count = n(),       # n() gives the group count
    .groups = "keep",  # keep information about the grouping
  )
```

```
# A tibble: 224 x 6
# Groups:   origin, dest [224]
   origin dest  max.delay avg.delay min.delay count
   <chr>  <chr>     <dbl>     <dbl>     <dbl> <int>
 1 EWR    ALB         328    14.4         -34   439
 2 EWR    ANC          39    -2.5         -47     8
 3 EWR    ATL         796    13.2         -39  5022
```

```
 4 EWR    AUS          349    -0.474       -59   968
 5 EWR    AVL          228     8.80        -26   265
 6 EWR    BDL          266     7.05        -43   443
 7 EWR    BNA          364    12.7         -41  2336
 8 EWR    BOS          422     4.78        -47  5327
 9 EWR    BQN          208    10.9         -43   297
10 EWR    BTV          306    12.2         -41   931
# i 214 more rows
```

Useful to know

Instead of first grouping the data and then summarizing, we can use the `.by` argument int the summarize function. This is more efficient and can be used for simple summaries.

```
flights %>%
  summarize(
    max.delay = max(arr_delay, na.rm = TRUE),
    avg.delay = mean(arr_delay, na.rm = TRUE),
    min.delay = min(arr_delay, na.rm = TRUE),
    count = n(),
    .by = c(origin, dest),
  )
```

```
# A tibble: 224 x 6
   origin dest  max.delay avg.delay min.delay count
   <chr>  <chr>     <dbl>     <dbl>     <dbl> <int>
 1 EWR    IAH         374      5.41       -63  3973
 2 LGA    IAH         435      1.45       -59  2951
 3 JFK    MIA         614     -1.99       -64  3314
 4 JFK    BQN         183      6.94       -32   599
 5 LGA    ATL         895     11.3        -49 10263
 6 EWR    ORD        1109      9.00       -59  6100
 7 EWR    FLL         375      7.76       -56  3793
 8 LGA    IAD         398     13.7        -47  1803
 9 JFK    MCO         329      5.30       -63  5464
10 LGA    ORD         448      1.83       -62  8857
# i 214 more rows
```

2.4.2 Applying functions to groups adding to the original table

The `mutate` function can be used to add new columns to the original table but calculated for each group individually. For example,

```r
df %>%
  group_by(b) %>%
  mutate(mean_a = mean(a)) %>%
  ungroup()
```

```
# A tibble: 4 x 3
      a     b mean_a
  <dbl> <dbl>  <dbl>
1     1     4      1
2     2    NA      2
3     3     6      3
4    NA     8     NA
```

Step 1: Group the data using the column `b`

Step 2: Calculate the mean of column `a` for each group. The `mutate` function applies in this case only to the group and not the full table.

Step 3: To restore the full table, use `ungroup`. This removes the grouping.

2.4.3 Shortcuts

Some *Split - Apply - Combine* steps occur so frequently that `dplyr` provides shortcuts for them. The function `count` is one of them. It combines `group_by` and `summarize` to count the number of occurrences of each value in a column. For example,

```r
flights %>%
  count(origin)
```

```
# A tibble: 3 x 2
  origin      n
  <chr>   <int>
1 EWR    120835
2 JFK    111279
3 LGA    104662
```

is equivalent to

```r
flights %>%
  group_by(origin) %>%
  summarize(n = n())
```

```
# A tibble: 3 x 2
  origin      n
  <chr>   <int>
1 EWR    120835
2 JFK    111279
3 LGA    104662
```

2.5 Concatenate and join data

2.5.1 Concatenating data

Concatenating data is the process of combining two or more datasets into a single one. The `dplyr` package provides the `bind_rows` and `bind_cols` functions for this purpose. The `bind_rows` function is used to combine rows from two datasets. The `bind_cols` function is used to combine columns from two datasets.

The `dplyr::bind_rows` function combines rows from two or more datasets. This is useful if your dataset is split into parts and you need to combine them.

```
df1 <- tibble(a = c(1, 2, 3), b = c(4, 5, 6))
df2 <- tibble(a = c(4, 5, 6), b = c(7, 8, 9), c = c(10, 11, 12))

# bind rows
bind_rows(df1, df2)
```

```
# A tibble: 6 x 3
      a     b     c
  <dbl> <dbl> <dbl>
1     1     4    NA
2     2     5    NA
3     3     6    NA
4     4     7    10
5     5     8    11
6     6     9    12
```

Columns are matched by name. If a data frame has missing columns, the missing columns are filled with `NA`.

The `dplyr::bind_cols` function combines columns from two or more datasets. This is useful if you want to add columns to an existing dataset.

```r
df3 <- tibble(c = c(4, 5, 6), d = c(7, 8, 9))
bind_cols(df1, df3)
```

```
# A tibble: 3 x 4
      a     b     c     d
  <dbl> <dbl> <dbl> <dbl>
1     1     4     4     7
2     2     5     5     8
3     3     6     6     9
```

The rows are combined in the order they appear, so be careful that the rows match. If you have a common identifier in the data frames, using and of the join functions is a safer option.

If columns occur in both dataframes, the column names will be adjusted to be unique. In the following example, the `a` columns are renamed to `a...1` and `a...3`.

```r
df4 <- tibble(a = c(4, 5, 6), c = c(7, 8, 9))
bind_cols(df1, df4)
```

```
New names:
* `a` -> `a...1`
* `a` -> `a...3`
```

```
# A tibble: 3 x 4
  a...1     b a...3     c
  <dbl> <dbl> <dbl> <dbl>
1     1     4     4     7
2     2     5     5     8
3     3     6     6     9
```

2.5.2 Joins

Joins are used to combine or merge two datasets, based on common identfiiers. The `dplyr` package provides a number of functions for this purpose. The main functions are `inner_join`, `left_join`, `right_join`, and `full_join`. The `inner_join` function is the most commonly used one. It keeps only rows that have matching values in both datasets. The `left_join` function keeps all rows from the first dataset and adds columns from the second dataset if there is a match. The `right_join` function is the reverse of `left_join`. The `full_join` function keeps all rows from both datasets. We will not cover this here but you can read more about joins in the R4DS book.

2.6 Additional tidyr functions

The `tidyr` package has a few more functions that are useful for data manipulation and cleanup. We list these here for completeness.

- `pivot_wider()/spread()`: Spreads a pair of key:value columns into a set of tidy columns
- `pivot_longer()/gather()`: Gather takes multiple columns and collapses into key-value pairs, duplicating all other columns as needed. You use `pivot_longer()/gather()` when you notice that you have columns that are not variables
- `separate()` turns a single character column into multiple columns
- `unite()` paste together multiple columns into one (reverse of `separate()`)

> **i** Further information
>
> The dplyr cheatsheet is a two-page summary of all the main features of dplyr. For more details about dplyr, see the main website at https://dplyr.tidyverse.org/.

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
library(tidyverse)    # or library(dplyr)
library(nycflights13) # load flight data
dim(flights)
df <- flights %>%
  # step 1
  filter(distance < 1000) %>%
  # step 2
  select(dep_delay, arr_delay, origin, dest, air_time, distance) %>%
  # step 3
  mutate(
    Z_dep_delay = (dep_delay - mean(dep_delay, na.rm = TRUE)) /
      sd(dep_delay, na.rm = TRUE),
    dep_delay = dep_delay / 60,
    arr_delay = arr_delay / 60,
    speed = distance / (air_time / 60)
  ) %>%
  # step 4
  arrange(-speed) %>%
```

```
  # step 5
  print(n = 4)
df <- tibble(a = c(1, 3, 2, 1, 3, 2), b = c(4, 4, 5, 5, 6, 6))
df %>% arrange(a)
df %>% arrange(b, a)
df %>% arrange(desc(b), a)
df %>% arrange(-a, b)
flights %>%
  colnames()

# select by part of name:
flights %>%
  select(starts_with("arr")) %>%
  colnames()
flights %>%
  select(contains("time")) %>%
  colnames()
flights %>%
  select(ends_with("delay")) %>%
  colnames()

# using regular expressions
flights %>%
  select(matches("(time|hour|minute)")) %>%
  colnames()

# select columns using a vector of column names (all_of or any_of)
variables <- c("dep_delay", "arr_delay", "origin", "dest",
  "air_time", "distance")
flights %>%
  select(all_of(variables)) %>%
  colnames()

# remove columns by name:
flights %>%
  select(-c(carrier, flight, tailnum)) %>%
  colnames()
flights %>%
  select(! matches("(time|hour|minute)")) %>%
  colnames()
df <- tibble(
  row = c(1:7),
```

```r
  a = c(1, 2, 3, 1, 3, 1, NA),
  b = c(4, 5, 6, 4, 5, 6, 8)
)

# filter by column values
df %>%
  filter(a == 1)
df %>%
  filter(a == 1 & b == 4)
df %>%
  filter(a > 2 | b == 4)
df %>%
  filter(a %in% c(1, 3))

# remove by column values - use the ! operator
df %>%
  filter(! a %in% c(1, 3))

# filter to missing values in column
df %>%
  filter(is.na(a))
df %>%
  distinct(a, b, .keep_all = TRUE)
df %>%
  slice(1:3)
df %>%
  slice(1, 3, 5)
idx <- c(2, 4, 6)
df %>%
  slice(idx)
df %>%
  slice_min(a)
df %>%
  slice_max(a, n = 2)
df %>%
  slice_sample(n = 5)  # sample without replacement
df %>%
  slice_sample(n = 5, replace = TRUE)  # sample with replacement
df <- tibble(
  a = c(1, 2, 3),
  b = c(4, 5, 6)
)
```

```r
# Add one or more new columns
df %>%
  mutate(c = a + b)
df %>%
  mutate(c = a + b, d = a - b, e = c * d)
# Modify existing columns
df %>%
  mutate(a = a + 1)
df %>%
  add_column(idx = seq_len(nrow(df)))
df %>%
  add_column(idx = seq_len(nrow(df)), .before = "a")
df %>%
  add_row(a = 4, b = 7)
df %>%
  add_row(a = 4, b = 7, .after = 1)
df <- tibble(
  a = c(1, 2, 3, -9999),
  b = c(4, -9999, 6, 8)
)
df <- df %>%
  mutate_all(na_if, -9999) %>%
  print()
df <- tibble(
  a = c(1, 2, 3, NA),
  b = c(4, NA, 6, 8)
)
df %>%
  drop_na()  # drop rows with missing values
df %>%
  drop_na(a)  # drop rows in which column "a" has missing values
df %>%
  replace_na(list(a = 0, b = 99))
df %>%
  replace_na(as.list(colMeans(df, na.rm = TRUE)))
flights %>%
  group_by(carrier) %>%
  summarize(avg_dep_delay = mean(dep_delay, na.rm = TRUE)) %>%
  arrange(avg_dep_delay)
flights %>%
  group_by(origin, dest) %>%   # group by both origin and dest
  summarise(
```

```
    max.delay = max(arr_delay, na.rm = TRUE),
    avg.delay = mean(arr_delay, na.rm = TRUE),
    min.delay = min(arr_delay, na.rm = TRUE),
    count = n(),       # n() gives the group count
    .groups = "keep",  # keep information about the grouping
  )
flights %>%
  summarize(
    max.delay = max(arr_delay, na.rm = TRUE),
    avg.delay = mean(arr_delay, na.rm = TRUE),
    min.delay = min(arr_delay, na.rm = TRUE),
    count = n(),
    .by = c(origin, dest),
  )
df %>%
  group_by(b) %>%
  mutate(mean_a = mean(a)) %>%
  ungroup()
flights %>%
  count(origin)
flights %>%
  group_by(origin) %>%
  summarize(n = n())
df1 <- tibble(a = c(1, 2, 3), b = c(4, 5, 6))
df2 <- tibble(a = c(4, 5, 6), b = c(7, 8, 9), c = c(10, 11, 12))

# bind rows
bind_rows(df1, df2)
df3 <- tibble(c = c(4, 5, 6), d = c(7, 8, 9))
bind_cols(df1, df3)
df4 <- tibble(a = c(4, 5, 6), c = c(7, 8, 9))
bind_cols(df1, df4)
```

3 Data visualization

The tidyverse package `ggplot2` is a great way to create complex data visualization. It is based on *The Grammar of Graphics* by Wilkinson (L. Wilkinson 2005). The basic idea is that you can describe a graph in several layers:

- **Data**: the data that contain the variables to be visualized
- **Aesthetics**: mapping of data variables to visual properties (aesthetics) such as position, color, size, shape, etc.
- **Geometries**: the geometric objects that represent the data points such as points, lines, bars, etc.
- **Facets**: splitting the data into subsets and creating a separate plot for each subset
- **Statistics**: statistical transformations of the data such as smoothing, binning, etc.
- **Coordinates**: the coordinate system used for the plot such as Cartesian, polar, etc.
- **Theme**: control the visual appearance of the plot such as background color, grid lines, font size, etc.

While the package is conceptually founded on this *grammar*, you will see that there is no one-to-one correspondence. However, the ideas shine through when building plots.

3.1 `ggplot2` — the basics

`ggplot2` is loaded either with `library(ggplot2)` or `library(tidyverse)`.

```
library(tidyverse)
library(patchwork)
library(GGally)
library(hexbin)
```

We also load the package `patchwork` which allows us to combine multiple graphs into a single figure.

Here is an example of a `ggplot2` graph:

```
ggplot(data = mtcars, mapping = aes(x = wt, y = mpg)) +                    ①
  geom_point(color = "darkgreen") +                                        ②
  geom_smooth(formula = y ~ x, method = "lm")                              ③
```

① The `ggplot` command creates a new plot. It combines the *data* and *aesthetics* layers. The first argument is the data frame, and the second argument is the mapping of data onto aesthetics. It maps the variables from the dataframe to the visual properties of the plot. In this case, we are mapping the variable `wt` to the x-axis and the variable `mpg` to the y-axis.

② The `geom_point` command adds a scatter plot using the `+` operator. This corresponds to the *geometries* layer. In addition, we specify the `color` of the points. This overrides the default color *aesthetic* of the points from black to `darkgreen`. In the same way we changed the aesthetics of the points, we can change other aesthetics such as size, shape, and transparency, We could also change the mapping easily.

③ The `geom_smooth` command adds a fit curve. This is an example of the *statistics* layer. The mapped `x` and `y` aesthetics (`wt` and `mpg`) are used in a linear regression and the resulting fit line added to the graph. The `formula` argument specifies the formula for the curve. The default is the linear regression `y~x`. The `method` argument specifies the method for fitting the curve. In this case, we are using the linear model. The default is to fit a loess curve or a spline fit dependent on the dataset size.

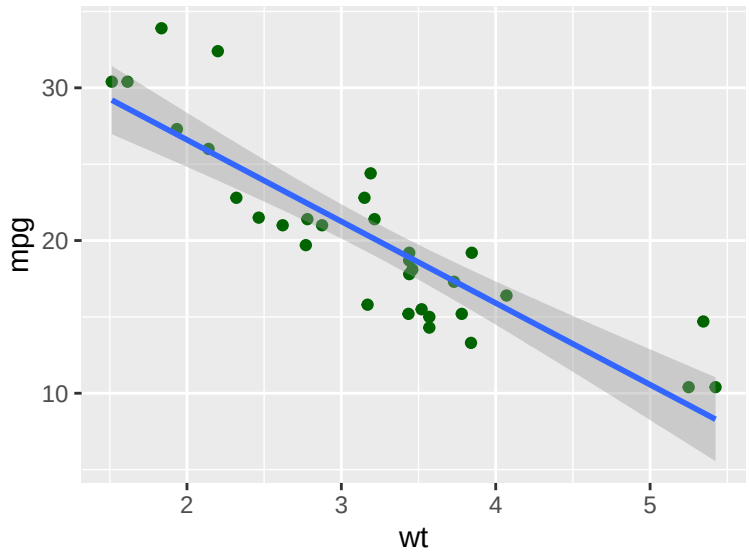


Figure 3.1: Example of a ggplot2 graph

Figure 3.1 gives the resulting plot. There are many ways in which we can extend the plot. For example, we can color the points by another data property. In the following example, we color by the number of cylinders (`cyl`).

```
ggplot(data = mtcars, mapping = aes(x = wt, y = mpg,
    color = factor(cyl), shape = factor(cyl))) +                    ①
  geom_point() +                                                    ②
```

```
  geom_smooth(formula = y ~ x, method = "lm")                    ③
```

① The new aesthetic mapping is added as an argument the `aes` function. The variable `cyl`, the number of cylinders, is mapped to the `color` aesthetic.[1]

② Without any change to this command, the scatterplot now uses different colors based on the `cyl` value of the datapoint.

③ The *color* aesthetic also influences the linear regression fit. The data are grouped by `cyl` and individual regression lines are determined and drawn for each value. The same colors are used as for the data points.

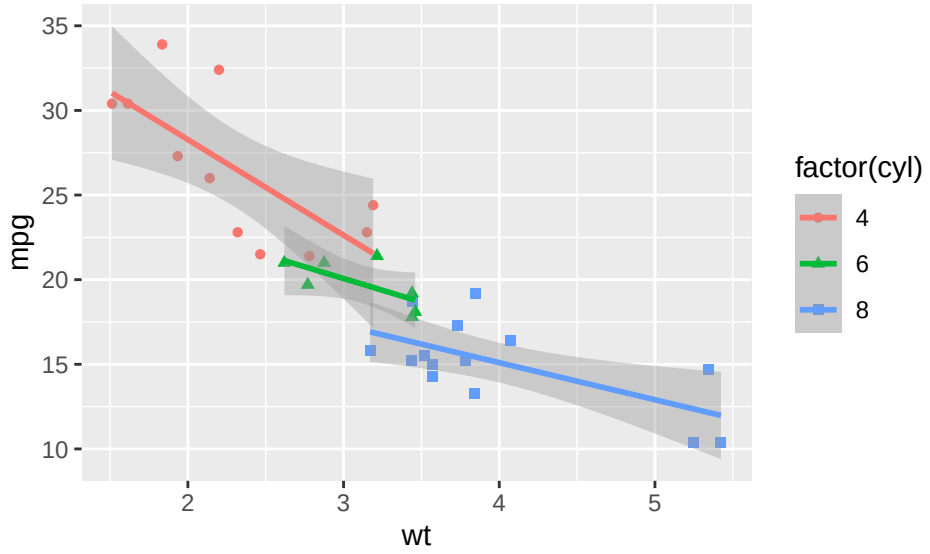


Figure 3.2: Example of a ggplot2 graph with color representing a property

From just looking at the graph in Figure 3.2 alone, it is not clear what the different colors represent. Best practice is to add a legend; `ggplot2` does this automatically for you. With the new *aesthetic*, a legend is added to the plot that explains what the colors represent. This is done automatically to ensure that the resulting visualization follows best practices.

The graph in Figure 3.2 uses the column names `wt` and `mpg` as labels for the axis and `factor(cyl)` for the color information in the legend. We can provide better labels these using the `labs` command getting the final plot in Figure 3.3.

```
ggplot(data = mtcars, mapping = aes(x = wt, y = mpg,
    color = factor(cyl), shape = factor(cyl))) +
  geom_point() +
  geom_smooth(formula = y ~ x, method = "lm") +
```

[1]Note: we convert the `cyl` variable to a factor. It would be better to do this at the data preprocessing stage.

```
labs(title = "Plot of MPG vs Weight",                    ①
  x = "Weight",                                          ②
  y = "MPG",
  color = "Number of Cylinders",
  shape = "Number of Cylinders")                         ③
```

① You can add a `title` to graph using the `labs` command.
② The values of the `x` and `y` arguments are used to label the axis.
③ The `color` value is used in the legend.

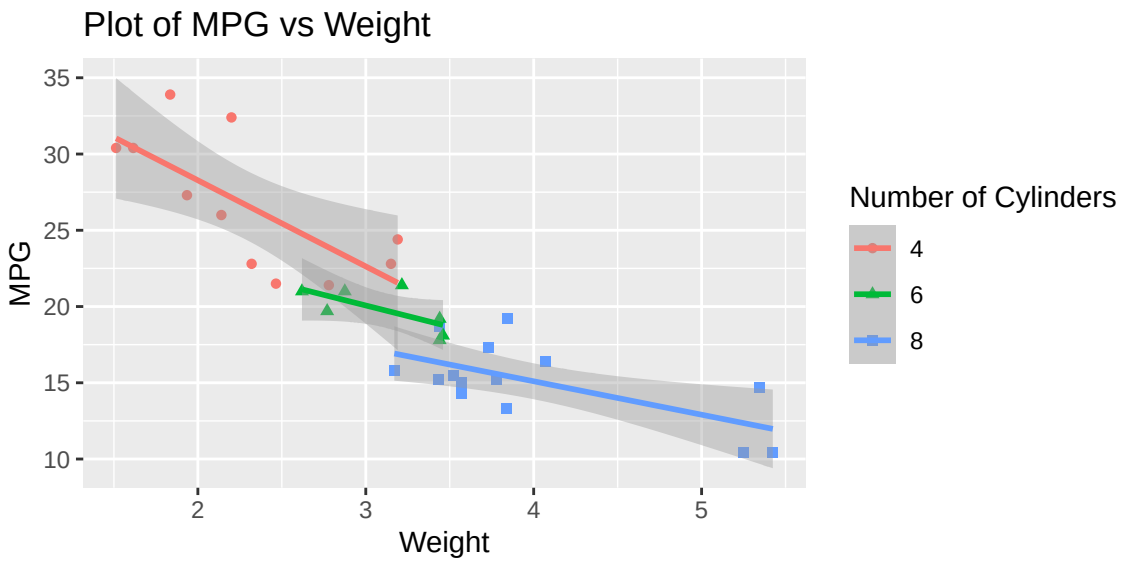


Figure 3.3: Adding labels to the plot

This short example should demonstrate the power and flexibility of `ggplot2`. It is useful to get an understanding of the full potential of `ggplot2`.

Todo

- Go to the ggplot2 website at https://ggplot2.tidyverse.org/ and look at the Reference section.
- Visit the R graph gallery at https://r-graph-gallery.com/ggplot2-package.html to get an overview of the different types of plots that can be created with `ggplot2`.

In the following we will look at more examples of graphs that are useful for exploratory data analysis.

3.2 Visualizing a single variable

In exploratory data analysis, we are often interested in the distribution of single variables in a dataset. Commonly used graphs are boxplots, histograms, and density plots.

```
library(patchwork)
g1 <- ggplot(data = mtcars, mapping = aes(y = mpg)) +
  geom_boxplot() +
  labs(y = "MPG", title = "Boxplot")
g2 <- ggplot(data = mtcars, mapping = aes(x = mpg)) +
  geom_histogram(bins = 20) +
  labs(x = "MPG", title = "Histogram")
g3 <- ggplot(data = mtcars, mapping = aes(x = mpg)) +
  geom_density() +
  labs(x = "MPG", title = "Density plot")

g1 + g2 + g3 + plot_layout(widths = c(1, 2, 2))
```

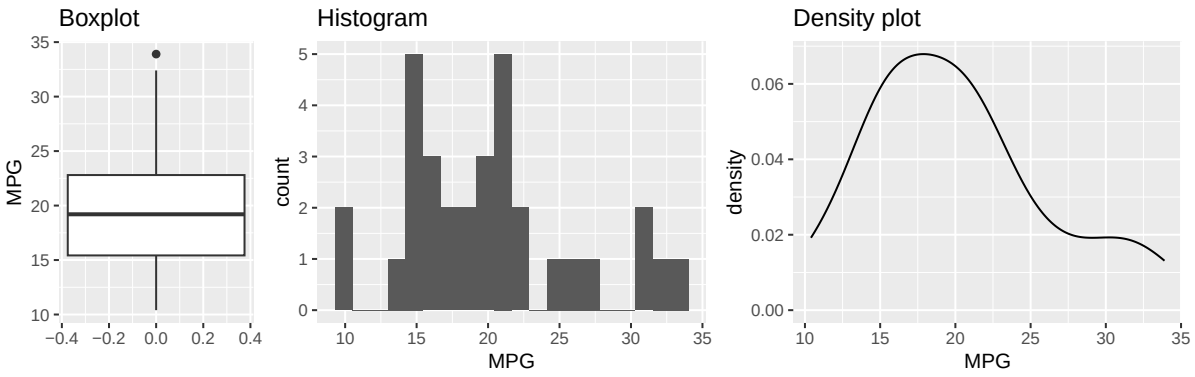


Figure 3.4: Visualizing a single variable with a boxplot, histogram, and density plot

Figure 3.4 shows the three plots. The first plot is a boxplot (`geom_boxplot`). It shows the median, the first and third quartile, and the minimum and maximum values. The variable of interest is mapped onto the `y` axis. This is different from the histogram and densityplot where the variable is mapped onto the `x` axis.[2]

The second plot is a histogram (`geom_histogram`). It shows the distribution of the data. By default, the graph uses 30 bins, which may be fine for your data. However, it is often useful to experiment with different bin sizes (`binwidth`) or counts (`bins`) and see how the graph changes. It can be helpful to also change the position of the bins using `center` or `boundary`. Figure 3.5 shows how these arguments change the creation of bins.

[2]You could map the variable also onto the `x` aesthetic. This would give you a horizontal boxplot.

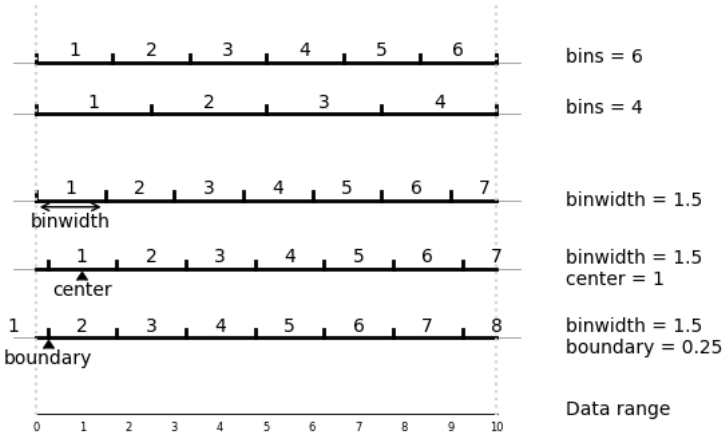


Figure 3.5: Definition of `bin`, `binwidth` and `boundary` in `geom_histogram`

The third plot in Figure 3.4 is a density plot. It is similar to a histogram but uses a smooth curve instead of bars. Similar to histograms, the shape of density plots can be controlled using arguments. The `bw` argument controls the smoothness of the density plot. By default, a bandwidth is chosen automatically from the data using one of several approaches. `nrd0` (Silverman 1986) or `nrd` (Scott 1992) are good choices. The `adjust` argument (default 1) can be used to adjust this automatically determined bandwidth.

Figure 3.6 shows how the histogram and density plot from Figure 3.4 changes when varying `binwidth`, `center`, `boundary`, and `bw`.

```
library(patchwork)
g <- ggplot(data = mtcars, mapping = aes(x = mpg)) +
  labs(x = "MPG")
g1 <- g + geom_histogram(binwidth = 3) +
  labs(title = "binwidth = 3")
g2 <- g +
  geom_histogram(binwidth = 3, center = 10) +
  labs(title = "binwidth = 3, center = 10")
g3 <- g +
  geom_histogram(binwidth = 3, boundary = 10) +
  labs(title = "binwidth = 3, boundary = 10")

g <- ggplot(data = mtcars, mapping = aes(x = mpg)) +
  labs(x = "MPG")
g4 <- g + geom_density() + labs(title = "bw = \"nrd0\"")
```

```
g5 <- g + geom_density(bw = 1.5) + labs(title = "bw = 1.5")
g6 <- g + geom_density(bw = 0.5) + labs(title = "bw = 0.5")
(g1 + g2 + g3) / (g4 + g5 + g6)
```

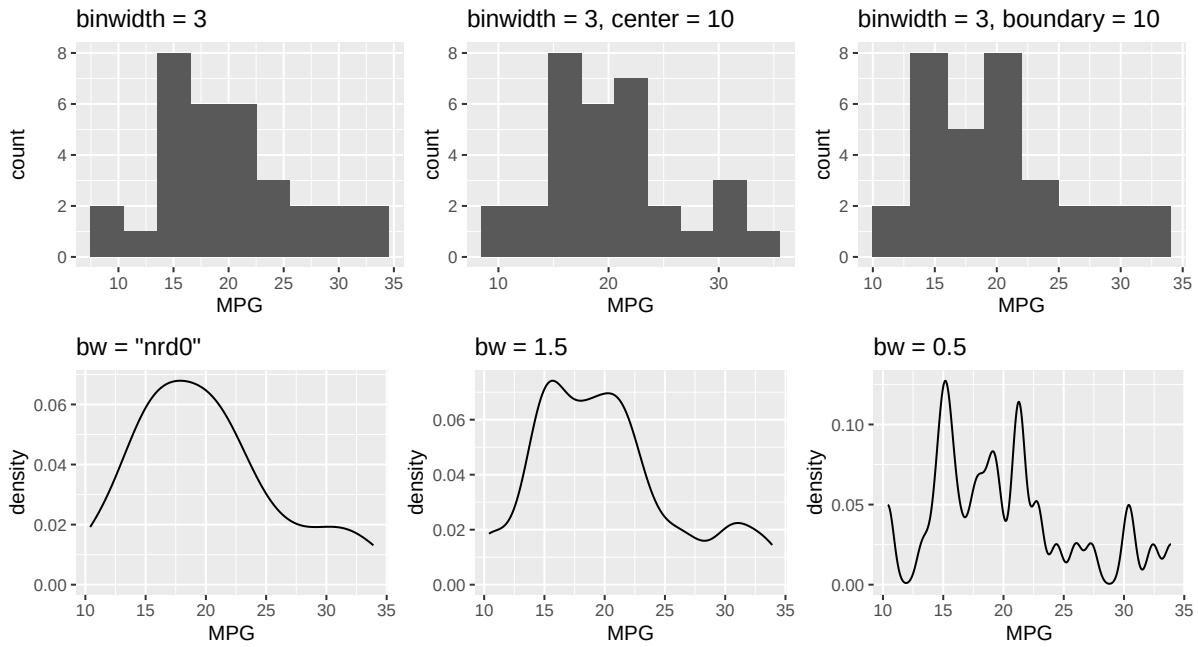


Figure 3.6: Effect of changing `binwidth`, `center`, `boundary`, and `bw` on histogram and density plots

> **Useful to know**
>
> In Figure 3.4, we combined multiple plots into a single figure using the `patchwork` package. We create three plots `g1`, `g2`, and `g3` and then combine them using the `+` operator. The `plot_layout` function is used to control the relative sizes here. You will find more examples of this throughout the book.
> In Figure 3.6, six plots were combined using
>
> ```
> (g1 + g2 + g3) / (g4 + g5 + g6)
> ```

Sometimes you will be interested in separating the data by a factor. For example, you may want to compare the distribution of the `mpg` variable for different numbers of cylinders. Figure 3.7 shows the same three plots as before but now grouped by the number of cylinders.

```
mtcars <- datasets::mtcars %>% mutate(cyl = as.factor(cyl))
g1 <- ggplot(data = mtcars, mapping = aes(y = mpg, x = cyl,
    color = cyl)) +
  geom_boxplot() +
  labs(x = "Cylinders", y = "MPG", title = "Boxplot") +
  theme(legend.position = "none")
g2 <- ggplot(data = mtcars, mapping = aes(x = mpg, fill = cyl)) +
  geom_histogram(bins = 20) +
  labs(x = "MPG", title = "Stacked histogram") +
  theme(legend.position = "none")
g3 <- ggplot(data = mtcars, mapping = aes(x = mpg, fill = cyl)) +
  geom_histogram(bins = 20, alpha = 0.5, position = "identity") +
  labs(x = "MPG", title = "Histogram") +
  theme(legend.position = "none")
g4 <- ggplot(data = mtcars, mapping = aes(x = mpg, fill = cyl)) +
  geom_density(alpha = 0.5) +
  labs(x = "MPG", title = "Density plot") +
  theme(legend.position = "none")

g1 + g2 + g3 + g4 + plot_layout(widths = c(1, 1, 1, 1))
```

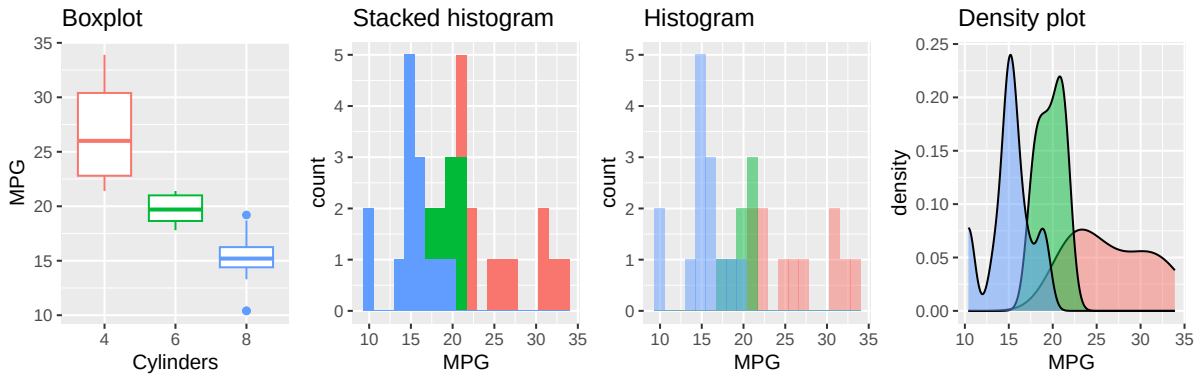


Figure 3.7: Visualizing a single variable with a boxplot, histogram, and density plot grouped by a factor

Boxplot: We map the `cyl` factor both to the `y` and the `color` aesthetic. This creates a separate boxplot for each level of the factor.

Stacked histogram: For the histogram, we map the factor to the `fill` aesthetic. This creates a stacked histogram.

Histogram: To create a histogram for each level of the factor, we need to set the `position`

argument to `"identity"`. This creates a separate histogram for each level of the factor. To avoid the histograms being plotted on top of each other, we set the `alpha` argument to 0.5. This makes the histograms transparent.

Densityplot: The densityplot is similar to the histogram. We map the factor to the `fill` aesthetic and set the `alpha` argument to 0.5 to create overlayed densityplots for each level of the factor. The `position` argument has the same effect as for `geom_histogram`. The difference is that default are overlayed densityplots. Using `position="stack"` creates stacked densityplots.

By changing the `x` and `y` mapping, the boxplot can be arranged horizontally. See Figure 3.8.

```
g <- ggplot(data = mtcars, mapping = aes(x = mpg, y = cyl,
    color = cyl)) +
  geom_boxplot() +
  labs(x = "MPG", y = "Cylinders", title = "Boxplot") +
  theme(legend.position = "none")
g
```

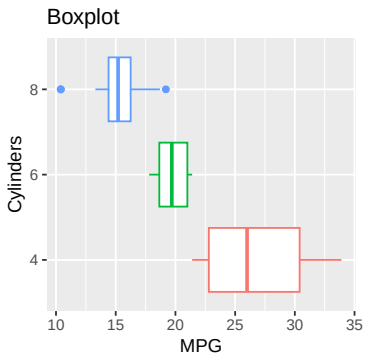


Figure 3.8: Horizontal boxplot grouped by a factor

3.3 Visualizing two variables

The introductory example showed the relationship between two variables using a scatterplot. Scatterplots are a good choice if the number of data points isn't too large. If the number of points gets larger, data points will be shown on top of each other. In this case, using transparent points, will reveal the density of the data. The argument `alpha` changes the transparency. `alpha=1` is the default no transparency. Reducing it increases the transparency; 0 makes the point invisible. A good starting point is 0.5. Always try a variety of alpha values to see which one works best for your data. See Figure 3.9 that demonstrates the effect of adding transparency.

```r
auto <- ISLR2::Auto %>%
  mutate(cylinders = as.factor(cylinders))

g1_1 <- ggplot(data = auto, mapping = aes(x = weight, y = mpg)) +
  geom_point() +
  labs(title = "Scatterplot", x = "Weight", y = "MPG")
g1_2 <- ggplot(data = auto, mapping = aes(x = weight, y = mpg)) +
  geom_point(alpha = 0.5) +
  labs(title = "Scatterplot with transparency",
    x = "Weight", y = "MPG")

g1_1 + g1_2
```

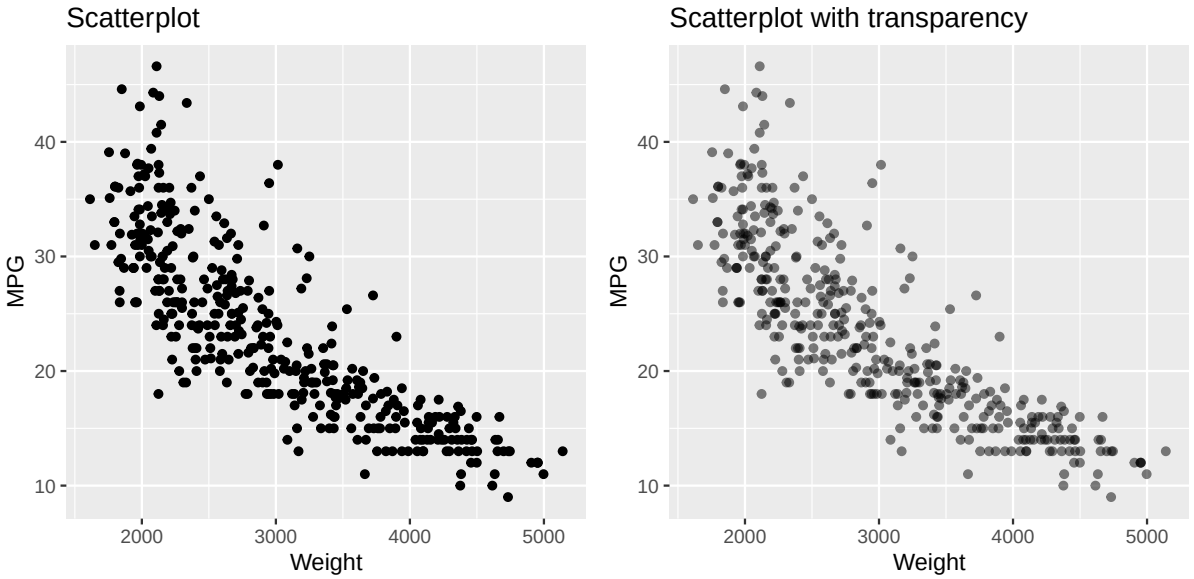


Figure 3.9: Using transparency if overplotting occurs

For very large datasets, it is better to use a heatmap or a two-dimensional density plot. Figure 3.10 shows the two versions of the heatmap for the `ISLR2::Auto` dataset.

```r
g1 <- ggplot(data = auto, mapping = aes(x = weight, y = mpg)) +
  geom_bin_2d(bins = 15) +
  labs(x = "Weight", y = "MPG", title = "Rectangular heatmap")
g2 <- ggplot(data = auto, mapping = aes(x = weight, y = mpg)) +
  geom_hex(bins = 15) +
  scale_fill_viridis_c(direction = -1) +
```

```
  labs(x = "Weight", y = "MPG", title = "Hexagonal heatmap")

g1 + g2
```

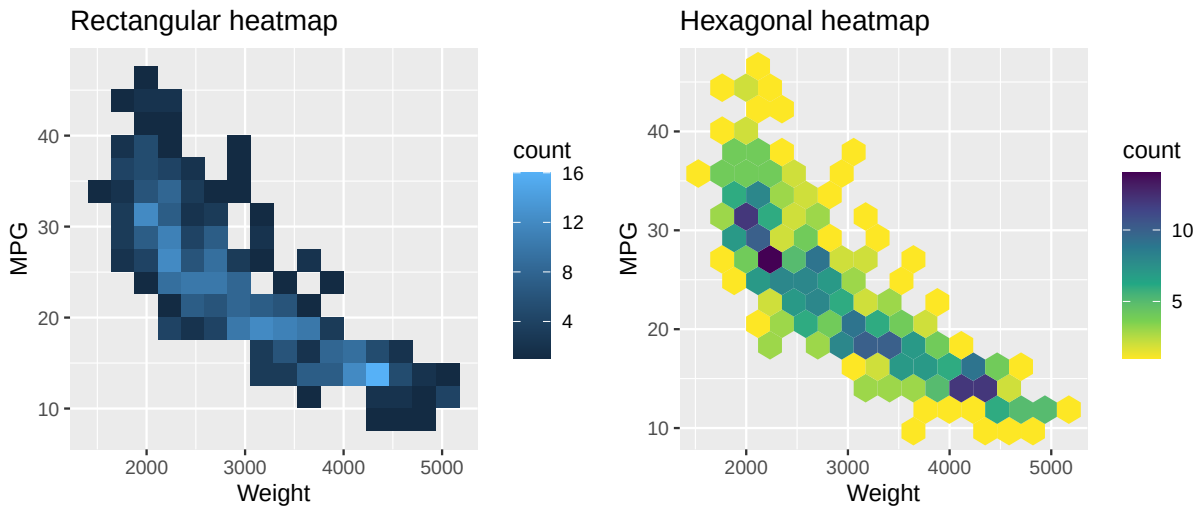


Figure 3.10: Visualizing two variables with a heatmap

The functions `geom_bin_2d` and `geom_hex` create heatmap representations of the distribution. The first uses rectangular, the second hexagonal patches. Use the `bins` argument to change the number of bins in a direction.Similar to histograms, try different values for bins for your data. There are other arguments to control binning. In the second example, we use a different colormap. Check the documentation for details.

By default, the color represents the count of data points in a bin. If you want to use a value, e.g. the average of a variable, you can use the `stat_summary_2d` function. See Figure 3.11 for an example.

```
ggplot(data = auto, mapping = aes(x = weight, y = displacement)) +
  stat_summary_hex(aes(z = mpg), bins = 10, fun = mean) +
  scale_fill_viridis_c(direction = -1) +
  geom_point() +
  labs(title = "Color heatmap by average of `mpg`",
    x = "Weight", y = "Displacement")
```

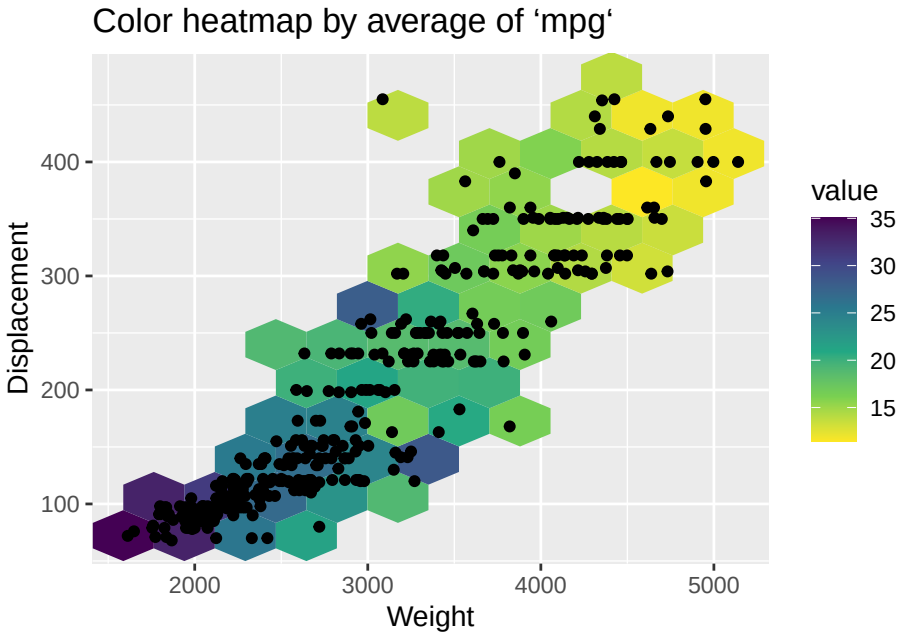


Figure 3.11: Color heatmap by average of `mpg`

Examples for two dimensional density plots are shown in Figure 3.12. The `geom_density_2d` function adds the density plot layer.

```
g1 <- ggplot(data = auto, mapping = aes(x = weight, y = mpg)) +
  geom_density_2d() +
  geom_point(size = 0.5, color = "darkblue") +
  labs(x = "Weight", y = "MPG", title = "Two-dimensional density")
g2 <- ggplot(data = auto, mapping = aes(x = weight, y = mpg,
    color = cylinders, shape = cylinders)) +
  geom_point(size = 0.5) +
  geom_density_2d() +
  scale_colour_brewer(palette = "Set1") +
  labs(title = "Two-dimensional density by categorical variable",
    x = "Weight", y = "MPG", color = "Number of Cylinders",
    shape = "Number of Cylinders")
g1 + g2
```

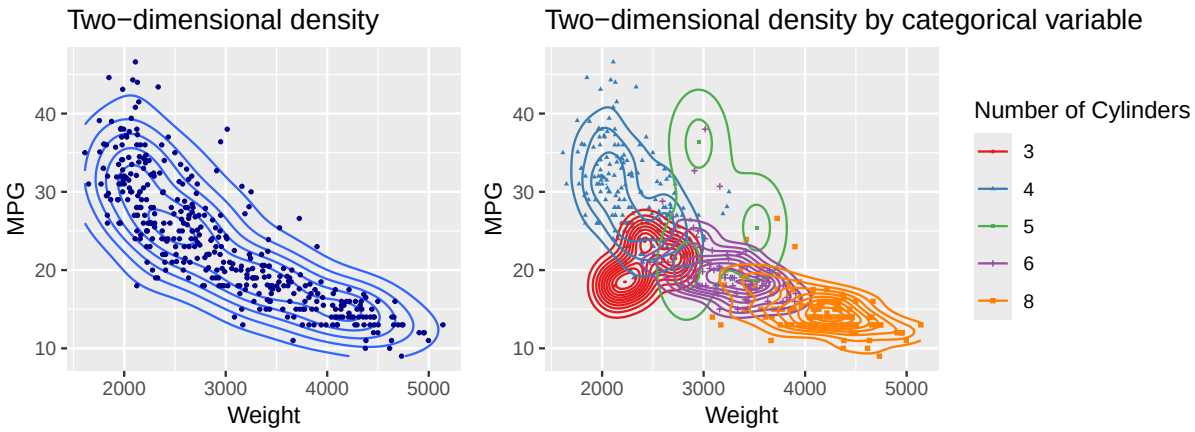


Figure 3.12: Visualizing two variables with a density plot

The left graph shows the density using contour lines. The right graph overlays individual density contours for each of the subsets formed by the categorical variable `cylinders`. The `scale_colour_brewer` function selects the colors. The `palette` argument specifies the color palette. `Set1` is a good choice for categorical variables.

> **Useful to know**
>
> The "Brewer" color scales are based on the work of Cynthia Brewer who designed color palettes for different use cases. While it was initially developed for coloring maps, the various palettes have become popular options for coloring graphs. You can go to https://colorbrewer2.org/ to explore this more.

We can add filled contour lines using the function `geom_density_2d_filled`. See Figure 3.13.

```
g1 <- ggplot(data = auto, mapping = aes(x = weight, y = mpg)) +
  geom_density_2d_filled(alpha = 0.5) +
  geom_point(size = 0.5) +
  labs(title = "Two-dimensional density (filled)",
    x = "Weight", y = "MPG") +
  theme(legend.position = "none")
g2 <- ggplot(data = auto, mapping = aes(x = weight, y = mpg)) +
  geom_density_2d_filled() + #contour_var = "ndensity", bins = 5) +
  geom_point(size = 0.5) +
  scale_fill_brewer() +
  labs(title = "Alternative color scheme",
    x = "Weight", y = "MPG") +
  theme(legend.position = "none")
```

```
g1 + g2
```

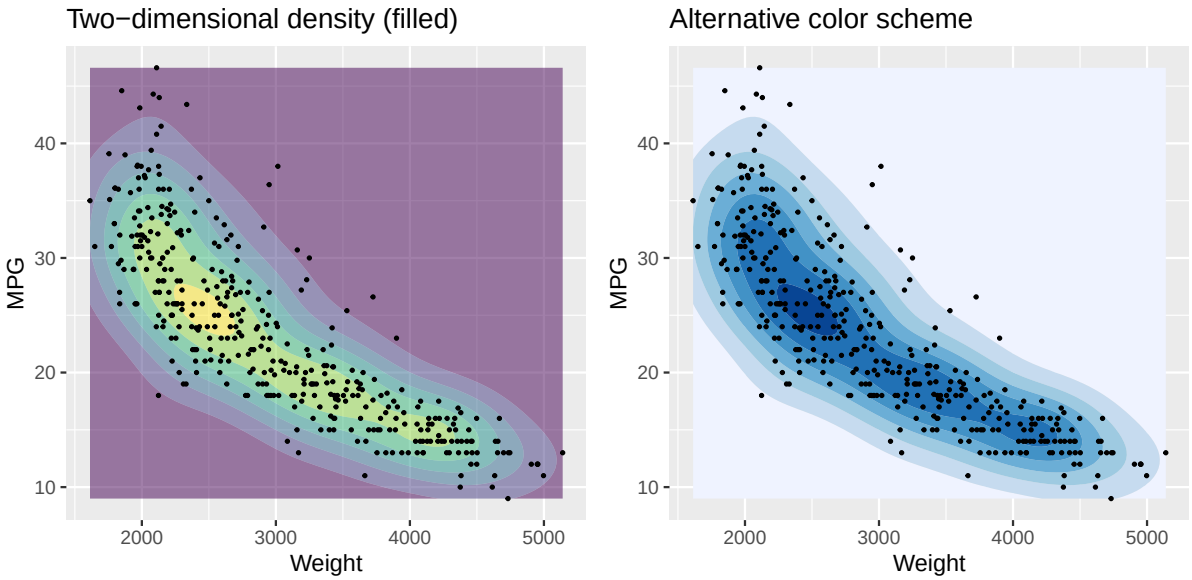


Figure 3.13: Filled two-dimensional density plot

3.4 Visualizing multiple variables

One option to visualize multiple variables in a graph is a pairplot. Figure 3.14 uses the `ggpairs` function from the `GGally` package.

```
pair_auto <- ISLR2::Auto %>%
  mutate(
    cylinders = as.factor(cylinders),
    origin = as.factor(origin),
  ) %>%
  select(-name)
ggpairs(pair_auto,
  lower = list(combo = wrap("facethist", binwidth = 0.5)))
```

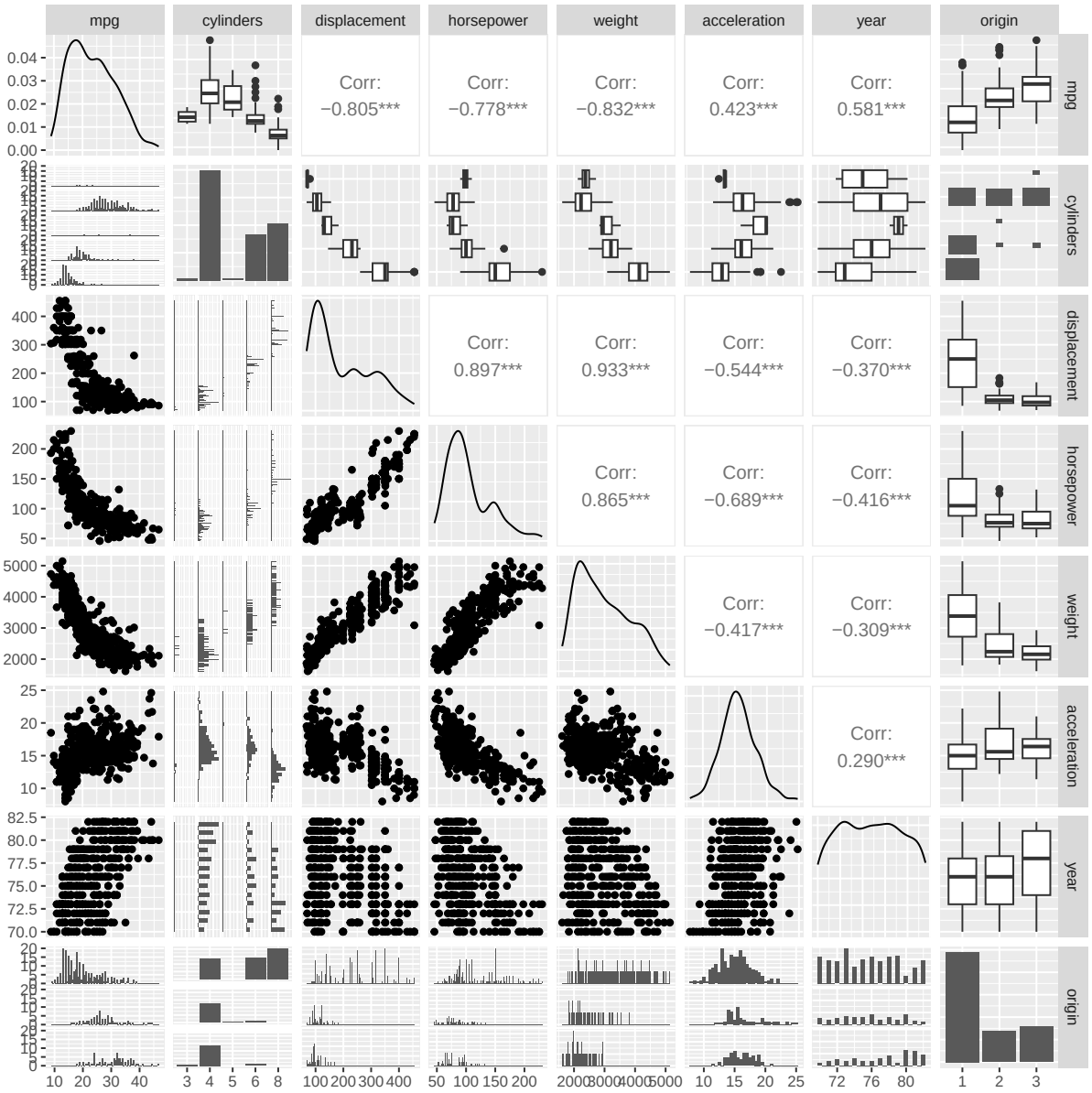


Figure 3.14: Pair plot

A pairplot shows visualizations of pairs of variables in a compact presentation. By default, `ggpairs` uses densityplots and bar charts along the diagonal to show the distribution of continuous and categorical variables. The upper and lower triangle visualizations depend on the type of the two variables. If both variables are are continuous, the upper triangle shows their correlation as a value and the lower triangle a scatterplot. If one variable is continuous and the other is categorical, the upper triangle uses boxplots and the lower triangle a bar chart. If both variables are categorical, the lower triangle shows a bar chart and the upper triangle

a type of two dimensional bar chart.

Figure 3.15 shows an alternative visualization of multiple variables. The `ggparcoord` function from the `GGally` package creates a parallel coordinate plot. It shows the values of each data point as a line.

```
g1 <- pair_auto %>%
  ggparcoord(columns = 1:7, groupColumn = 8)
g2 <- pair_auto %>%
  ggparcoord(columns = c(2:5, 7, 6, 1), groupColumn = 8, alpha = 0.5,
    splineFactor = 10)
g1 + g2
```

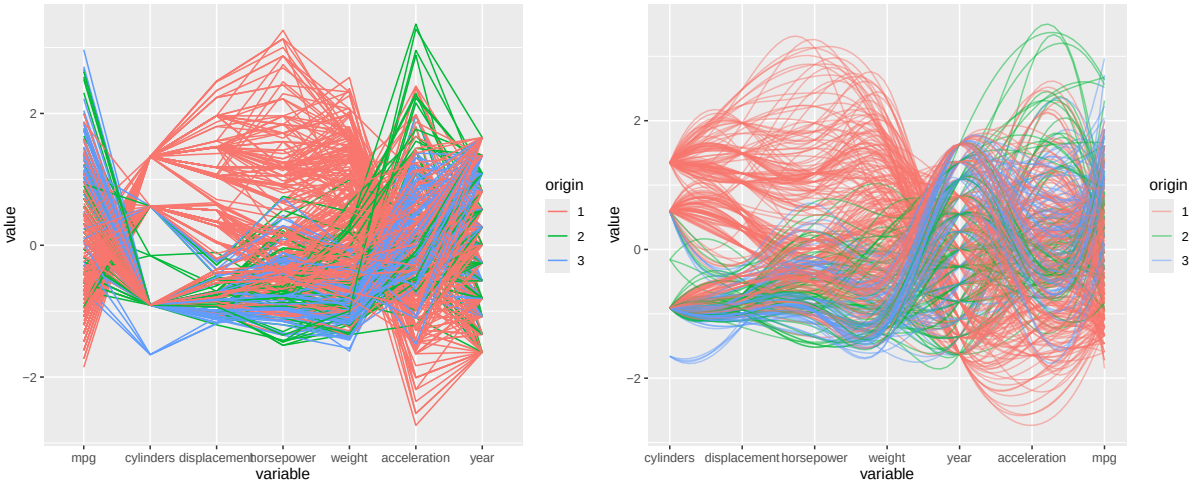


Figure 3.15: Parallel coordinate plot. Lines connect the values of each data point and are colored by the origin of the car.

Parallel coordinate plots can be hard to read and it is worth exerimenting with different orderings of the variables and other settings. Here, we used `alpha` to make the lines transparent which helps for larger datasets. The `splineFactor` argument controls the smoothness of the lines. Without smoothing, the variable values would be connected by straight lines. This separates the lines and makes it easier to see the distribution of the data. It also adds information on how the coordinates to the left (`mpg`) and the right (`displacement`) are connected. The `groupColumn` argument is used to color the lines by the origin of the car.

Parallel coordinate plots also benefit greatly from interactivity. You can use the `plotly` package to create interactive parallel coordinate plots (see https://plotly.com/r/parallel-coordinates-plot/ for examples).

3.5 Saving plots to file

You can save plots to file using the `ggsave` function. The following example saves the scatterplot from Figure 3.13 to a png file.

```
ggsave(filename = "example.png", plot = g1 + g2,
  width = 12.6, height = 6.3, units = "in", dpi = 300)
```

Here is the saved figure:

```
knitr::include_graphics("example.png")
```

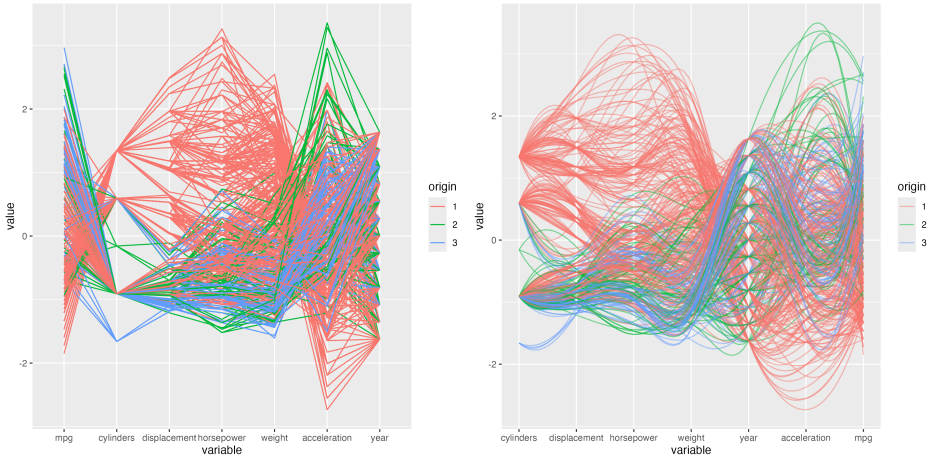


Figure 3.16: Graphs stored in example.png

3.6 `autoplot` and `autolayer` functions

Some *R* packages provide `autoplot` functions that create `ggplot2` graphs. If available, these functions are useful for quickly visualizing special data or the result of calculations. The functions return a `ggplot2` graph that can be further customized using the methods shown in this chapter. Packages that implement the `autoplot` function often also provide an `autolayer` function. This function adds a layer with a specialized visualization to an existing `ggplot2` graph.

In this book, we use `autoplot` functions to visualize the results of model tuning (see Chapter 14) and ROC curves (see Section 10.3).

For example, the `forecast` package provides `autoplot` and `autolayer` functions for time series objects. Figure 3.17 shows how the `autoplot` function selects an appropriate axis scale for time series data.

```
library(forecast)
```

```
Registered S3 method overwritten by 'quantmod':
  method            from
  as.zoo.data.frame zoo
```

```
autoplot(AirPassengers) +
  autolayer(seasadj(decompose(AirPassengers, "multiplicative"))) +
  theme(legend.position = "none")
```

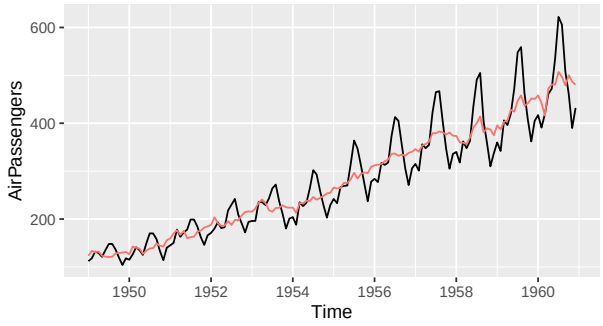


Figure 3.17: Example of an autoplot: air passenger counts (black) with seasonal adjustment (red)

Further information

- The ggplot2 cheatsheet is a two-page summary of all the main features of ggplot2.
- For more details about ggplot2, see the main ggplot2 website at https://ggplot2.tidyverse.org/.
- The R graph gallery provides an overview of the different types of plots that can be created with ggplot2.
- ggplot2: Elegant Graphics for Data Analysis by Hadley Wickham et al. is the definitive guide to ggplot2.
- The R Graphics Cookbook by Winston Chang is a great resource for learning how to create different types of plots in R.
- https://ggobi.github.io/ggally/ is the website of the `GGally` package. It provides a number of useful functions for creating more complex plots with `ggplot2`.

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
library(tidyverse)
library(patchwork)
library(GGally)
library(hexbin)
ggplot(data = mtcars, mapping = aes(x = wt, y = mpg)) +                    ①
  geom_point(color = "darkgreen") +                                        ②
  geom_smooth(formula = y ~ x, method = "lm")                              ③
ggplot(data = mtcars, mapping = aes(x = wt, y = mpg,
    color = factor(cyl), shape = factor(cyl))) +                           ①
  geom_point() +                                                           ②
  geom_smooth(formula = y ~ x, method = "lm")                              ③
ggplot(data = mtcars, mapping = aes(x = wt, y = mpg,
    color = factor(cyl), shape = factor(cyl))) +
  geom_point() +
  geom_smooth(formula = y ~ x, method = "lm") +
  labs(title = "Plot of MPG vs Weight",                                    ①
    x = "Weight",                                                          ②
    y = "MPG",
    color = "Number of Cylinders",
    shape = "Number of Cylinders")                                         ③
library(patchwork)
g1 <- ggplot(data = mtcars, mapping = aes(y = mpg)) +
  geom_boxplot() +
  labs(y = "MPG", title = "Boxplot")
g2 <- ggplot(data = mtcars, mapping = aes(x = mpg)) +
  geom_histogram(bins = 20) +
  labs(x = "MPG", title = "Histogram")
g3 <- ggplot(data = mtcars, mapping = aes(x = mpg)) +
  geom_density() +
  labs(x = "MPG", title = "Density plot")

g1 + g2 + g3 + plot_layout(widths = c(1, 2, 2))
knitr::include_graphics("images/bin_binwidth.png")
library(patchwork)
g <- ggplot(data = mtcars, mapping = aes(x = mpg)) +
  labs(x = "MPG")
```

```r
g1 <- g + geom_histogram(binwidth = 3) +
  labs(title = "binwidth = 3")
g2 <- g +
  geom_histogram(binwidth = 3, center = 10) +
  labs(title = "binwidth = 3, center = 10")
g3 <- g +
  geom_histogram(binwidth = 3, boundary = 10) +
  labs(title = "binwidth = 3, boundary = 10")

g <- ggplot(data = mtcars, mapping = aes(x = mpg)) +
  labs(x = "MPG")
g4 <- g + geom_density() + labs(title = "bw = \"nrd0\"")
g5 <- g + geom_density(bw = 1.5) + labs(title = "bw = 1.5")
g6 <- g + geom_density(bw = 0.5) + labs(title = "bw = 0.5")
(g1 + g2 + g3) / (g4 + g5 + g6)
(g1 + g2 + g3) / (g4 + g5 + g6)
mtcars <- datasets::mtcars %>% mutate(cyl = as.factor(cyl))
g1 <- ggplot(data = mtcars, mapping = aes(y = mpg, x = cyl,
    color = cyl)) +
  geom_boxplot() +
  labs(x = "Cylinders", y = "MPG", title = "Boxplot") +
  theme(legend.position = "none")
g2 <- ggplot(data = mtcars, mapping = aes(x = mpg, fill = cyl)) +
  geom_histogram(bins = 20) +
  labs(x = "MPG", title = "Stacked histogram") +
  theme(legend.position = "none")
g3 <- ggplot(data = mtcars, mapping = aes(x = mpg, fill = cyl)) +
  geom_histogram(bins = 20, alpha = 0.5, position = "identity") +
  labs(x = "MPG", title = "Histogram") +
  theme(legend.position = "none")
g4 <- ggplot(data = mtcars, mapping = aes(x = mpg, fill = cyl)) +
  geom_density(alpha = 0.5) +
  labs(x = "MPG", title = "Density plot") +
  theme(legend.position = "none")

g1 + g2 + g3 + g4 + plot_layout(widths = c(1, 1, 1, 1))
g <- ggplot(data = mtcars, mapping = aes(x = mpg, y = cyl,
    color = cyl)) +
  geom_boxplot() +
  labs(x = "MPG", y = "Cylinders", title = "Boxplot") +
  theme(legend.position = "none")
g
```

```r
auto <- ISLR2::Auto %>%
  mutate(cylinders = as.factor(cylinders))

g1_1 <- ggplot(data = auto, mapping = aes(x = weight, y = mpg)) +
  geom_point() +
  labs(title = "Scatterplot", x = "Weight", y = "MPG")
g1_2 <- ggplot(data = auto, mapping = aes(x = weight, y = mpg)) +
  geom_point(alpha = 0.5) +
  labs(title = "Scatterplot with transparency",
    x = "Weight", y = "MPG")

g1_1 + g1_2
g1 <- ggplot(data = auto, mapping = aes(x = weight, y = mpg)) +
  geom_bin_2d(bins = 15) +
  labs(x = "Weight", y = "MPG", title = "Rectangular heatmap")
g2 <- ggplot(data = auto, mapping = aes(x = weight, y = mpg)) +
  geom_hex(bins = 15) +
  scale_fill_viridis_c(direction = -1) +
  labs(x = "Weight", y = "MPG", title = "Hexagonal heatmap")

g1 + g2
ggplot(data = auto, mapping = aes(x = weight, y = displacement)) +
  stat_summary_hex(aes(z = mpg), bins = 10, fun = mean) +
  scale_fill_viridis_c(direction = -1) +
  geom_point() +
  labs(title = "Color heatmap by average of `mpg`",
    x = "Weight", y = "Displacement")
g1 <- ggplot(data = auto, mapping = aes(x = weight, y = mpg)) +
  geom_density_2d() +
  geom_point(size = 0.5, color = "darkblue") +
  labs(x = "Weight", y = "MPG", title = "Two-dimensional density")
g2 <- ggplot(data = auto, mapping = aes(x = weight, y = mpg,
    color = cylinders, shape = cylinders)) +
  geom_point(size = 0.5) +
  geom_density_2d() +
  scale_colour_brewer(palette = "Set1") +
  labs(title = "Two-dimensional density by categorical variable",
    x = "Weight", y = "MPG", color = "Number of Cylinders",
    shape = "Number of Cylinders")
g1 + g2
g1 <- ggplot(data = auto, mapping = aes(x = weight, y = mpg)) +
  geom_density_2d_filled(alpha = 0.5) +
```

```
  geom_point(size = 0.5) +
  labs(title = "Two-dimensional density (filled)",
    x = "Weight", y = "MPG") +
  theme(legend.position = "none")
g2 <- ggplot(data = auto, mapping = aes(x = weight, y = mpg)) +
  geom_density_2d_filled() + #contour_var = "ndensity", bins = 5) +
  geom_point(size = 0.5) +
  scale_fill_brewer() +
  labs(title = "Alternative color scheme",
    x = "Weight", y = "MPG") +
  theme(legend.position = "none")

g1 + g2
pair_auto <- ISLR2::Auto %>%
  mutate(
    cylinders = as.factor(cylinders),
    origin = as.factor(origin),
  ) %>%
  select(-name)
ggpairs(pair_auto,
  lower = list(combo = wrap("facethist", binwidth = 0.5)))
g1 <- pair_auto %>%
  ggparcoord(columns = 1:7, groupColumn = 8)
g2 <- pair_auto %>%
  ggparcoord(columns = c(2:5, 7, 6, 1), groupColumn = 8, alpha = 0.5,
    splineFactor = 10)
g1 + g2
ggsave(filename = "example.png", plot = g1 + g2,
  width = 12.6, height = 6.3, units = "in", dpi = 300)
knitr::include_graphics("example.png")
library(forecast)
autoplot(AirPassengers) +
  autolayer(seasadj(decompose(AirPassengers, "multiplicative"))) +
  theme(legend.position = "none")
```

4 Interactive visualization

In Chapter 3, we used `ggplot` to create data visualizations. These are great for reports and when you already know what you want to demonstrate with your data. At an early stage of exploratory data analysis, it can however be useful to explore data interactively. A popular open-source graphing library is `plotly`. It can be used with Python, R, and Javascript and allows the creation of interactive dashboards. Here, we will only briefly discuss how to create interactive visualizations in R using the `plotly` package and embed them into R Markdown docuemnts.

4.1 `plotly` in R.

The `plotly` package is available on CRAN and can be installed as follows:

```
install.packages("plotly", repos = "http://cran.rstudio.com")
```

and loaded using:

```
library(plotly)
```

Interactivity depends of course on the output format; interactivity works well in HTML pages or in RStudio, but is not supported when the Rmarkdown is converted to PDF.

4.2 Two dimensional scatter plot using `plot_ly`

Here is the code to create a simple two dimensional scatterplot:

```
auto <- ISLR2::Auto %>%
  mutate(cylinders = as.factor(cylinders))

plot_ly(                                                    ①
  data = auto,
  x = ~horsepower,                                          ②
```

```
  y = ~mpg,
  text = ~name,
  type = "scatter",                                                    ③
  mode = "markers",
  hoverinfo = "text+x+y"                                               ④
)
```

① The `plot_ly` command takes a variety of arguments to specify the data, how they are mapped onto aesthetics (`x`, `y`, `text`) and what type of graph should be created. We map `horsepower` to `x`, `mpg` to `y`, and the `name` to 'text.
② You can also specify transformatios of data here, e.g. `y = ~log(mpg)`
③ We specify a scatterplot where the data are represented using markers.
④ The `hoverinfo` argument defines that we want to display the `x`, `y`, and `text` values, so `horsepower`, `mpg`, and `name` when the mouse is hovered over a point in the scatterplot.

`file:////private/var/folders/_8/ms0ft4913k3290v7f0g_yfpc0000gn/T/Rtmp6POWQ9/file33673534faa5/`

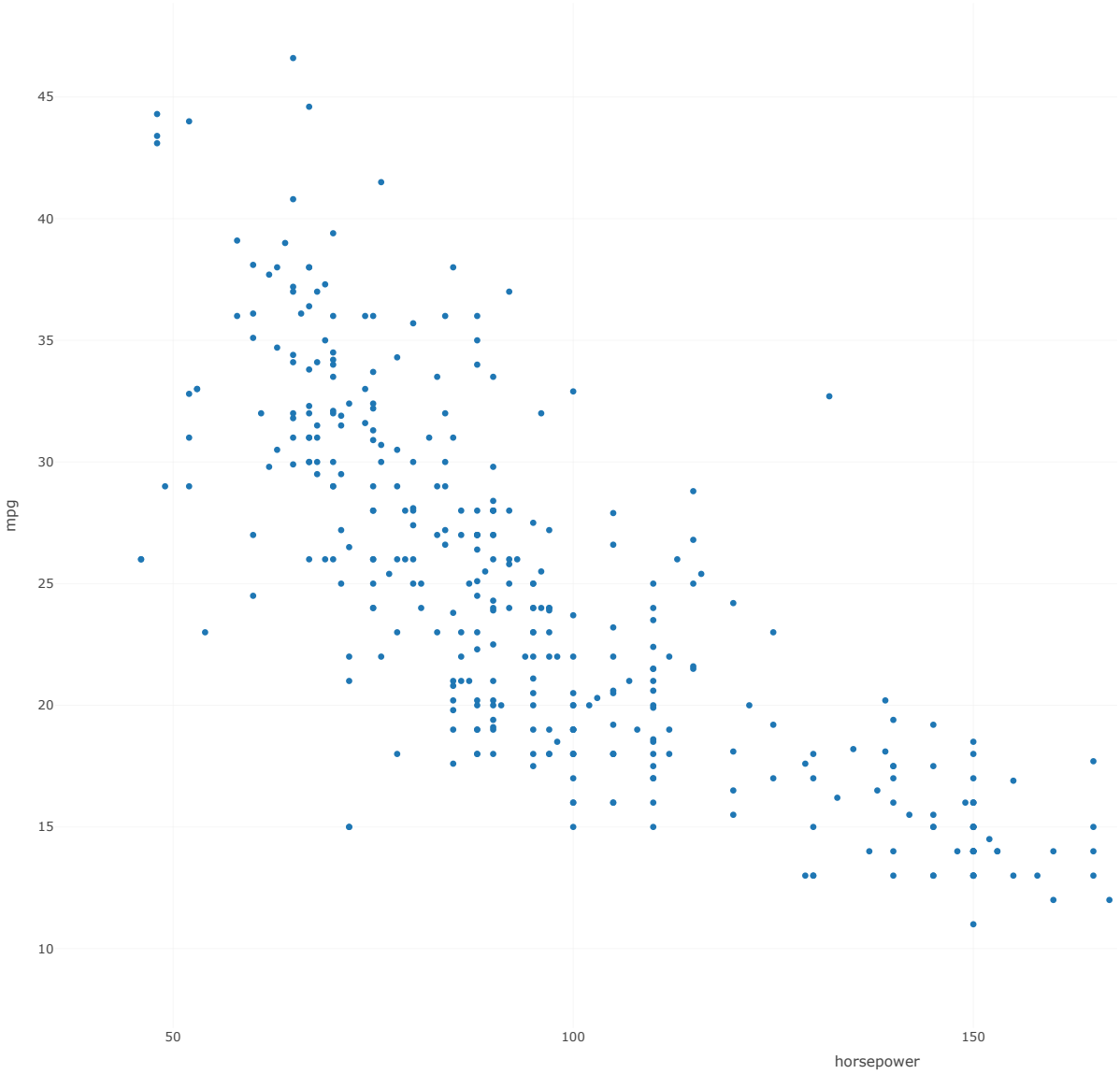


Figure 4.1: Interactive scatterplot of mpg versus horsepower (`plot_ly` version)

The resulting plot (Figure 4.1) can be zoomed and dragged. Hovering over a point displays information about the car.

4.3 Add interactivity to ggplot figure using `ggplotly`

If you already have a `ggplot2` visualization, you can conveniently convert it into an interactive *plotly* graph using the `ggplotly()` function. For example, the following code creates a scatterplot of `mpg` versus `horsepower` from the `auto` data set and converts it into an interactive plot:

```
g <- ggplot(auto, aes(x = horsepower, y = mpg, label = name)) +
  geom_point()

ggplotly(g)
```

`file:////private/var/folders/_8/ms0ft4913k3290v7f0g_yfpc0000gn/T/RtmpxebGSn/file63d44ecd0555/`

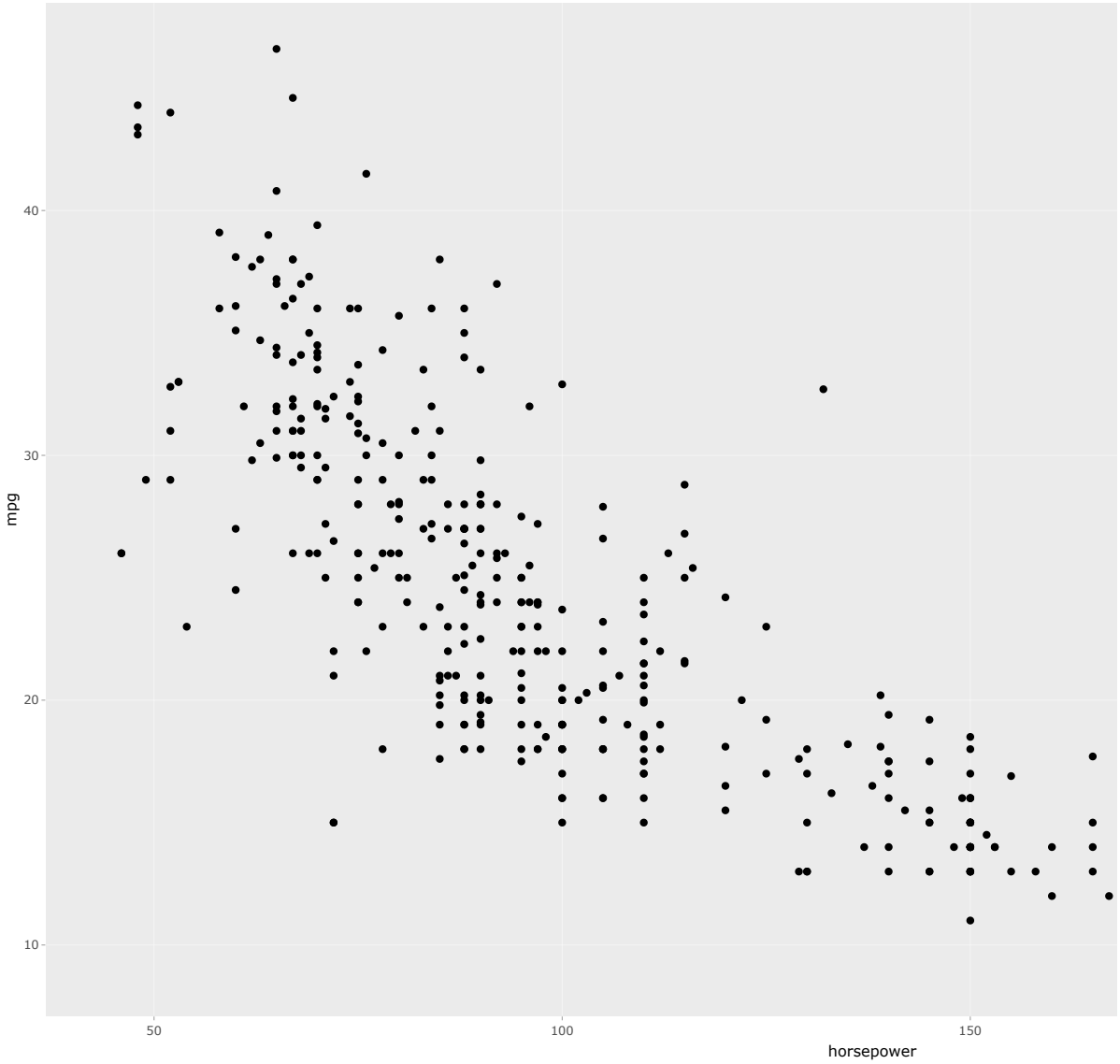


Figure 4.2: Interactive scatterplot of mpg versus horsepower (`ggplotly` version)

This is all that needs to be done.

4.4 Three dimensional plots using `plot_ly`

You can use `plot_ly` also to create three dimensional plots as shown in the following example.

```
plot_ly(
  x = auto$mpg,
  y = auto$weight,
  z = auto$horsepower,                                    ①
  color = auto$cylinders,
  type = "scatter3d",                                     ②
  mode = "markers",
  marker = list(size = 4)                                 ③
)
```

① 3D scatterplots require mapping a feature to `z`
② The `marker` argument allows configuring the symbol, e.g. . Here, we control the size of the markers.

`file:////private/var/folders/_8/ms0ft4913k3290v7f0g_yfpc0000gn/T/RtmpxebGSn/file63d45d6c07b0/`

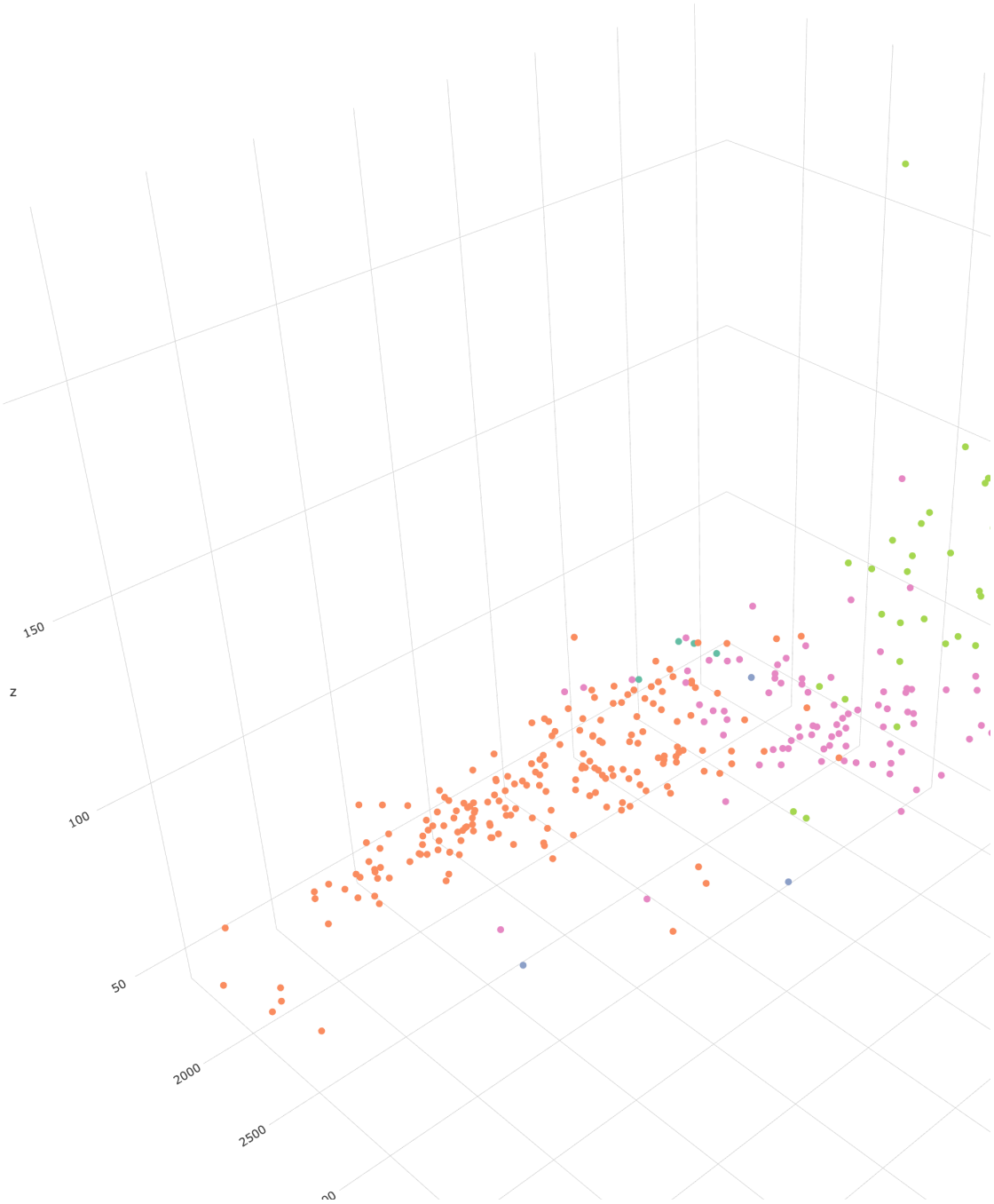


Figure 4.3: Interactive 3D scatterplot of mpg versus horsepower and weight

> **Further information**
>
> *Plotly* is more than just scatterplots. There are a much wider variety of interactive plots possible. Check out the documentation at https://plotly.com/r/ to learn more about plotly's interface to R.

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
install.packages("plotly", repos = "http://cran.rstudio.com")
library(plotly)
auto <- ISLR2::Auto %>%
  mutate(cylinders = as.factor(cylinders))

plot_ly(                                                              (1)
  data = auto,
  x = ~horsepower,                                                    (2)
  y = ~mpg,
  text = ~name,
  type = "scatter",                                                   (3)
  mode = "markers",
  hoverinfo = "text+x+y"                                              (4)
)
g <- ggplot(auto, aes(x = horsepower, y = mpg, label = name)) +
  geom_point()

ggplotly(g)
plot_ly(
  x = auto$mpg,
  y = auto$weight,
  z = auto$horsepower,                                                (1)
  color = auto$cylinders,
  type = "scatter3d",                                                 (2)
  mode = "markers",
  marker = list(size = 4)                                             (3)
)
```

Part II

Training models

5 Training predictive models

R provides a very large number of packages with functions for fitting predictive models. While there is some consistency in how models are trained, there are many differences. For example, some models can only be trained using the matrix interface so cannot be used easily with a formula. All of this can be overwhelming for new users. It would be nice if there was a consistent interface to all models.

There are various packages that address this. Here is a small selection.

- *tidymodels* (a collection of packages that share a common design philosophy with *tidyverse* and are designed to work together)
- `caret` (**C**lassification **A**nd **RE**gression **T**raining)
- `modelr` (part of the tidyverse, but less powerful than *tidymodels*
- `mlr3`
- `h2o.ai` (in contrast to the other packages, this is a commercial product that supports the full machine learning workflow from development to deployment)

In this course, we will focus on *tidymodels*.

5.1 What is *tidymodels*?

The *tidymodels* package is developed by Max Kuhn who now works at RStudio / posit. It was first released in 2018 and is still under active development. It is an ecosystem of packages that share a common design philosophy and are designed to work together. The packages include

- `parsnip` for model specification
- `recipes` for data preprocessing
- `rsample` for resampling
- `yardstick` for model evaluation
- `tune` for hyperparameter tuning
- `workflows` for modeling workflows
- `tidyposterior` for Bayesian modeling

The *tidymodels* packages are designed to work with the `tidyverse` and `tidydata` principles. The packages are designed to be modular and extensible. They are loaded using the command.

```
library(tidymodels)
```

```
-- Attaching packages ------------------------------------ tidymodels 1.3.0 --

v broom        1.0.8     v recipes      1.3.1
v dials        1.4.0     v rsample      1.3.0
v dplyr        1.1.4     v tibble       3.3.0
v ggplot2      3.5.2     v tidyr        1.3.1
v infer        1.0.8     v tune         1.3.0
v modeldata    1.4.0     v workflows    1.2.0
v parsnip      1.3.1     v workflowsets 1.1.0
v purrr        1.0.4     v yardstick    1.3.2

-- Conflicts --------------------------------------- tidymodels_conflicts() --
x purrr::discard() masks scales::discard()
x dplyr::filter()  masks stats::filter()
x dplyr::lag()     masks stats::lag()
x recipes::step()  masks stats::step()
```

The packages were developed with the aim to making it easy to follow best practices.

> **i** Further information
>
> - Go to https://www.tidymodels.org/ to learn more about *tidymodels*
> - Links to all *tidymodels* packages https://www.tidymodels.org/packages/
> - Example code for all supported models https://parsnip.tidymodels.org/articles/Examples.html
> - Tidy modeling with R by Max Kuhn and Julia Silge
> - An overview of all packages in the *tidymodels* ecosystem can be found at https://www.tidymodels.org/packages/

Code

The code of this chapter is summarized here.

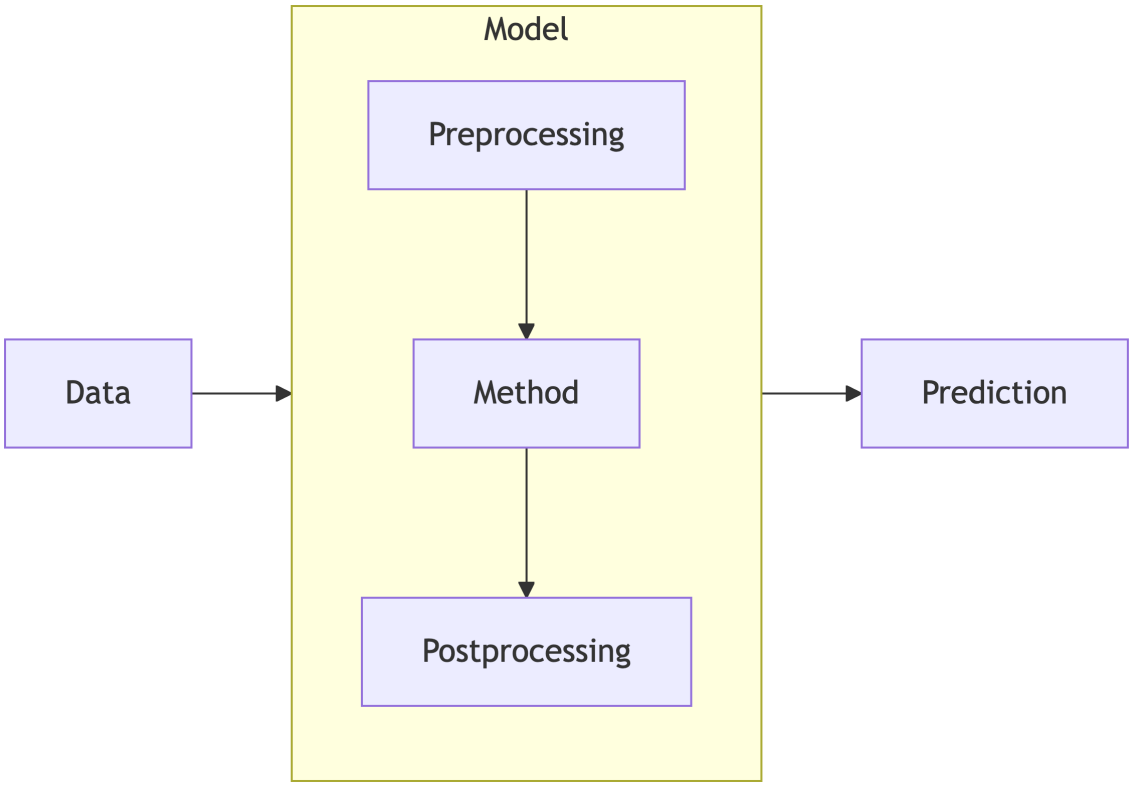
```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
library(tidymodels)
```

6 Workflows: Connecting the parts

Initially, one might think of a model as just the specific method or algorithm, for example a linear regression or a random forest; it is however more than that. Before we train the algorithm, we may need to preprocess the data. To name a few examples, we may want to

- normalize the data,
- transform into principal components,
- select a subset of features based on their properties,
- and/or handle missing values.

In many cases, these steps depend on the training data as much as the trained method itself. If we want to predict new data, they also need to pass the same preprocessing steps before the trained algorithm can be applied. It is therefore better to consider a model as being a combination of preprocessing and method. We may take this a step further and consider postprocessing the outcome/prediction of the trained method. The following graph summarizes the modeling workflow. The data is first preprocessed, then the method is applied, and finally the predictions are postprocessed. Preprocessing, method, and postprocessing together make up the model and can all depend on the training data.



Modeling workflow

6.1 Workflows in *tidymodels*

As we've seen, training a model is a multi-step process. We need to consider:

- defining the model
- training and validating the model
- deploying the model

Figure 7.1 summarizes this modeling workflow and shows how the individual steps are implemented in the *tidymodels* framework.

The left column covers the model definition part. A complete model definition requires:

- Preprocessing (`recipe` package - see Chapter 7)
- Model specification (`parsnip` package - see Chapter 8, Chapter 10)
- Postprocessing (`probably` - see Section 11.1.2)

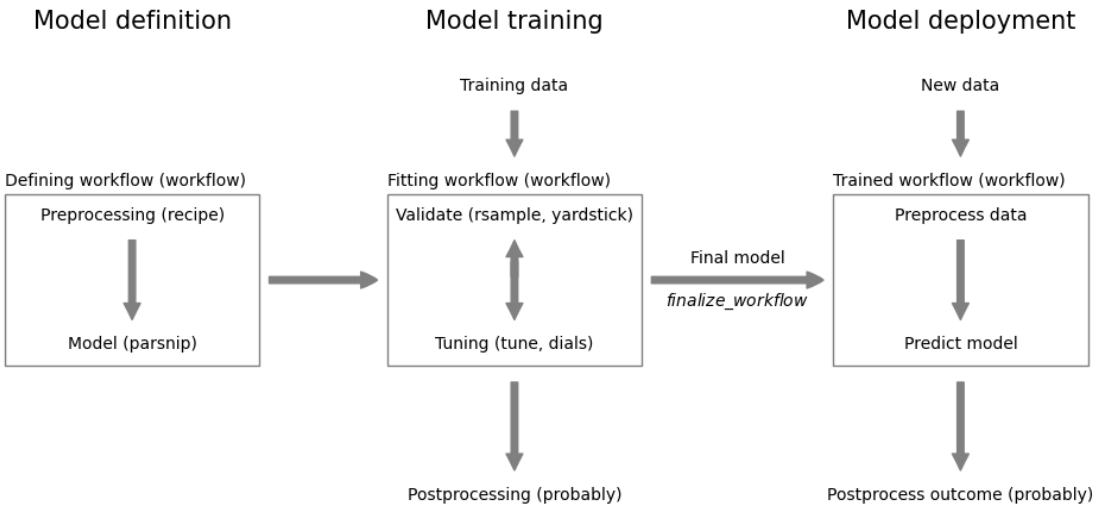


Figure 6.1: Modeling workflow

The `workflow` package from *tidymodels*, allows to combine the first two steps into a single `workflow` object. The `workflow` object can then be used to train and validate the model. Only the preprocessing and model specification can be included in the `workflow` at the moment. While the `postprocessing` step should be part of the full model process, the `workflow` package doesn't support it. For now, the `postprocessing` step has to be done separately. For example, we will see in Section 11.1.2 how the `probably` package can be used to define a threshold for binary classification models. In this class, we will only use postprocessing for classification models.

The `workflow` package is also able to orchestrate the model tuning and validation. It involves:

- Model tuning (`tune` package - see Chapter 14)
- Model validation (`rsample`, `yardstick` packages - see Chapter 12, Chapter 13)
- Tune postprocessing (`probably` package - see Chapter 14)

The objective of model tuning is to find the best model parameters. This can include the model hyperparameters (e.g. the number of trees in a random forest) and the preprocessing parameters (e.g. the number of principal components in a PCA). The `tune` package allows to define potential values and combinations of these parameters. This combined with the validation strategy defined using the `rsample` package, allows `tune` to examine the performance of different models and select the "best" one. The performance is measured using the various metrics provided by the `yardstick` package.

At the end of the model training step, we end up with a final trained workflow for deployment. For now, this means

- predict new data using the final model by:
  - preprocessing the new data using the (tuned) preprocessing steps
  - predicting with the (tuned) model
- if applicable, postprocessing the predictions (e.g. applying a threshold for the predicted class probabilities)

6.2 Workflow example

The following chapters covers the components of workflows in more detail. This can make it difficult to see the big picture. You can find complete workflows in the examples part.

- Chapter 25 covers preprocessing, model definition, tuning using cross-validation, finalizing the tuned model, and validating with a holdout set.
- Chapter 26 covers preprocessing, model definition, tuning using cross-validation, threshold selection using cross-validation results of the best mdoel, and prediction with the tuned model and threshold.

6.3 Models vs. workflows

It may initially be confusing to have a second way to build models. However, there is consistency between using both. As can be seen from the following table, the two approaches are similar and only differ in the way the models and the formula are specified.

Task

Model

Workflow

Specification

Validation

Model fit

Prediction

Augmenting a dataset

As we will see in Chapter 7 and Chapter 14, workflows are required to incorporate preprocessing into the model building process and to tune model parameters. It is therefore best, to use workflows and use simple models only when absolutely necessary.

Further information

- Take the short datacamp course at https://app.datacamp.com/learn/courses/modeling-with-tidymodels-in-r
- Go to https://workflows.tidymodels.org/ to learn more about the `workflows` package
- The workflowsets package allows to combine multiple workflows into a single object. This is useful when you want to compare multiple preprocessing steps and/or multiple models at the same time. We will not cover this package in this class.

7 Data preprocessing

We learned in the previous chapters, how to use `dplyr` to preprocess data. While this is useful for data exploration and cleanup, it is not enough for building models. For example, you may need to normalize predictors prior to the actual training step. The normalization transformation depends on the distribution of the training data and the exact same transformation needs to be applied to new data. Because of this, it is important to include preprocessing steps in the modeling pipeline.

The preprocessing steps can be used to:

- create new features,
- transform the data to make it more suitable for the model,
- introduce non-linearity into the model
- reduce the number of features, and
- impute missing data

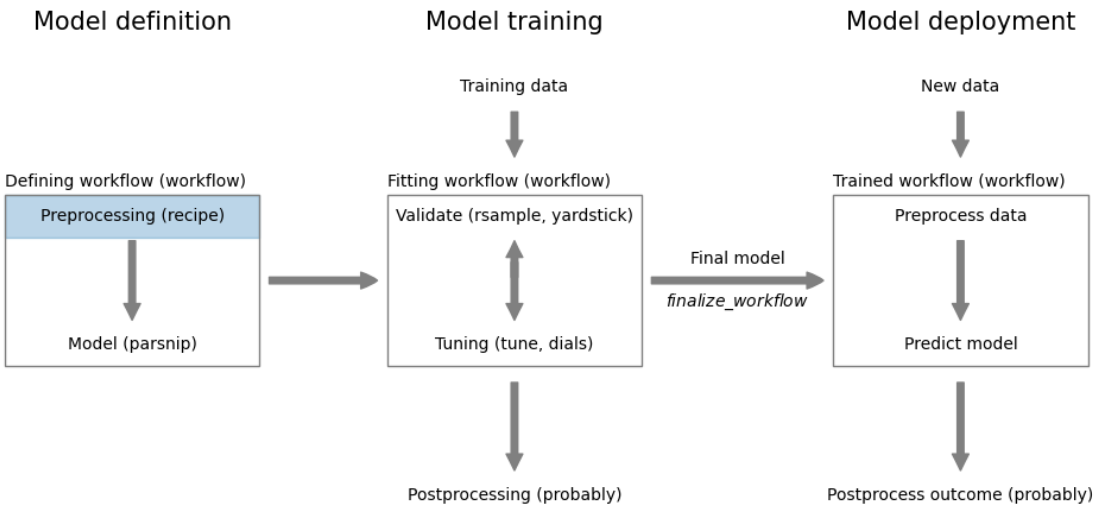


Figure 7.1: Preprocessing using `recipe`

The *tidymodels* framework makes this easy. The preprocessing steps are defined using the `recipe` package and combined with the model using a pipeline that is created using the

`workflows` package. In this chapter, we will learn how to use the `recipe` package to preprocess data and build models.

Load required packages:

```
library(tidyverse)
library(tidymodels)
library(patchwork)
library(kableExtra)
library(DT)
```

7.1 Preprocessing data with recipes

Let's use the `mtcars` dataset as an example. The `mtcars` dataset contains 32 observations (rows) and 11 variables (columns); check `?mtcars` for details on the dataset. The goal is to predict the fuel consumption (mpg) of a car based on the other variables. Here is the dataset:

```
data <- datasets::mtcars %>% as_tibble(rownames = "car")
data %>%
  head() %>%
  knitr::kable()
```

car	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

Two of the variables are categorical; transmission type (am) and engine shape (vs). We can also see that the continuous variables have different scales. For example, the displacement (wt) is in the hundreds while the number of cylinders (cyl) is in the single digits. Our plan for the preprocessing steps in the modeling pipeline is to:

- Convert the categorical variables to factors
- Normalize the continuous variables

We will use the `recipe` package to define the preprocessing steps. The first step is to create a recipe object using the `recipe()` function. The first argument is a formula that specifies the outcome variable and the predictors. The second argument is the data frame that contains the data. The `recipe()` function returns a recipe object that contains the preprocessing steps. The `summary()` function can be used to display the recipe object:

```
formula <- mpg ~ cyl + disp + hp + drat + wt + qsec + vs +
  am + gear + carb
rec_obj <- recipe(formula, data = data)
summary(rec_obj)
```

```
# A tibble: 11 x 4
   variable type      role      source
   <chr>    <list>    <chr>     <chr>
 1 cyl      <chr [2]> predictor original
 2 disp     <chr [2]> predictor original
 3 hp       <chr [2]> predictor original
 4 drat     <chr [2]> predictor original
 5 wt       <chr [2]> predictor original
 6 qsec     <chr [2]> predictor original
 7 vs       <chr [2]> predictor original
 8 am       <chr [2]> predictor original
 9 gear     <chr [2]> predictor original
10 carb     <chr [2]> predictor original
11 mpg      <chr [2]> outcome   original
```

The output tells us which variables are included in the model and what their respective role is. The role is just a label that is used to identify the variables. For our use, the automatically assigned roles of `predictor` and `outcome` are fine.

Now that we have a *recipe*, we can add preprocessing steps. The functions have the general format `step_{X}`,

```
rec_obj <- step_{X}(rec_obj, ..., arguments)    ## or
rec_obj <- rec_obj %>% step_{X}(..., arguments)
```

The `...` stands for a selection of variables. This could either be a list of variable names or a selector like `all_predictors`, `all_numeric`, or similar ones. More about this later. The remaining arguments are keyword arguments and require specifying the name of the argument.

The function `step_num2factor` converts a numerical column to a factor column. The first argument is the recipe object. The second argument is the name of the variable to be converted.

The `levels` argument is used to specify the levels of the factor. The `transform` argument is used to specify a function that is applied to the variable before it is converted to a factor.

```
rec_obj <- rec_obj %>%
  step_num2factor(vs, transform = function(x) x + 1,
    levels = c("V-shaped", "straight")) %>%
  step_num2factor(am, transform = function(x) x + 1,
    levels = c("automatic", "manual"))
```

The `levels` array is used to map the number in the column `vs` to a string. The values of `vs` are 0 and 1, so a simple lookup won't work. We need to first transform the value before we can use it as an index into the `levels` array. This is done using the `transform` function. For a value of 0 is changed to 1 by the `transform` function and then used as a index to look up the string "V-shaped" in the `levels` array. Similarly, a value of 1 is changed to 2 and then through lookup converted to "straight". If your values are already mapping to the correct indices, you can omit the `transform` argument. Finally, the whole column is changed to a factor. The second step does a similar transformation of the `am` column.

We can look at the result of the recipe so far using the `prep()` and `bake()` functions. The `prep()` function trains the steps using, in this case, the data. You can use the `training` argument to specify a different dataset. The `bake()` function applies the recipe to the data. The `new_data` argument is used to specify a different dataset. If the `new_data` argument is omitted, the recipe is applied to the data that was used to train the recipe.

```
rec_obj %>%
  prep() %>%
  bake(new_data = NULL) %>%
  top_n(4)
```

```
Selecting by mpg
```

```
# A tibble: 4 x 11
    cyl  disp    hp  drat    wt  qsec vs       am      gear  carb   mpg
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fct>    <fct>  <dbl> <dbl> <dbl>
1     4  78.7    66  4.08  2.2   19.5 straight manual     4     1  32.4
2     4  75.7    52  4.93  1.62  18.5 straight manual     4     2  30.4
3     4  71.1    65  4.22  1.84  19.9 straight manual     4     1  33.9
4     4  95.1   113  3.77  1.51  16.9 straight manual     5     2  30.4
```

Applying the recipe to the dataset results in a tibble where the columns `vs` and `am` are now factors.

The next step is to normalize the continuous variables. The `step_normalize()` function is used to normalize the variables.

```
rec_obj <- rec_obj %>%
  step_normalize(all_numeric_predictors())
rec_obj %>%
  prep() %>%
  bake(new_data = NULL) %>%
  top_n(4)
```

```
Selecting by mpg
```

```
# A tibble: 4 x 11
    cyl  disp     hp  drat    wt   qsec vs       am      gear   carb   mpg
  <dbl> <dbl>  <dbl> <dbl> <dbl>  <dbl> <fct>    <fct>  <dbl>  <dbl> <dbl>
1 -1.22 -1.23 -1.18  0.904 -1.04  0.907 straight manual 0.424 -1.12   32.4
2 -1.22 -1.25 -1.38  2.49  -1.64  0.376 straight manual 0.424 -0.503  30.4
3 -1.22 -1.29 -1.19  1.17  -1.41  1.15  straight manual 0.424 -1.12   33.9
4 -1.22 -1.09 -0.491 0.324 -1.74 -0.531 straight manual 1.78  -0.503  30.4
```

Here, we use the `all_numeric_predictors()` selector to specify all the numeric variables that are labeled as `predictor`. During the `prep` step, the mean and standard deviation of the variables are computed and stored with the recipe. The values are used in the `bake` step to transform the data. As we can see, the continuous variables are now normalized. Had we used `all_numeric` instead of `all_numeric_predictors`, the outcome variable `mpg` would have been normalized as well.

To summarize, the recipe for preprocessing the data is:

```
formula <- mpg ~ cyl + disp + hp + drat + wt + qsec + vs +
  am + gear + carb
rec_obj <- recipe(formula, data = data)  %>%
  step_num2factor(vs, transform = function(x) x + 1,
    levels = c("V-shaped", "straight")) %>%
  step_num2factor(am, transform = function(x) x + 1,
    levels = c("automatic", "manual")) %>%
  step_normalize(all_numeric_predictors())
```

Todo

Now is a good time to look through the reference of the `recipe` package to get an overview of what is available.

In the following, we highlight some of the more commonly used functions.

7.2 Transformations of individual features

The following steps apply numerical transformations

- `step_inverse`: $f(x) = 1/x$
- `step_invlogit`: $f(x) = 1/(1 + exp(-x))$
- `step_log`: $f(x) = log(x)$
- `step_logit`: $f(x) = log(x/(1 - x))$
- `step_sqrt`: $f(x) = \sqrt{x}$

The `step_mutate()` can be used like the `dplyr::mutate` function.

The Box-Cox transformation and the Yeo-Johnson transformation can be used to transform skewed data to have a more normal distribution (see wikipedia).

- `step_BoxCox`: Box-Cox transformation for non-negative data
- `step_YeoJohnson`: Yeo-Johnson transformation

All these steps transform a single column and replace the column with the transformed value. This is different for the `step_poly` function.

```
xy <- tibble(
  x = seq(-1, 1, length.out = 100),
  y = seq(-1, 1, length.out = 100)
)

transformed <- recipe(y ~ x, data = xy)  %>%
  step_poly(x, degree = 3) %>%
  prep() %>%
  bake(new_data = NULL)

transformed %>%
  head() %>%
  knitr::kable(digits = 4) %>%
  kableExtra::kable_styling(full_width = FALSE)
```

y	x_poly_1	x_poly_2	x_poly_3
-1.0000	-0.1715	0.2170	-0.2492
-0.9798	-0.1680	0.2038	-0.2190
-0.9596	-0.1646	0.1910	-0.1903
-0.9394	-0.1611	0.1783	-0.1632
-0.9192	-0.1576	0.1660	-0.1375
-0.8990	-0.1542	0.1539	-0.1132

The result of applying the `step_poly` function is three new columns `x_poly_1`, `x_poly_2`, and `x_poly_3`. The columns contain the first three orthogonal polynomials of the `x` column. The `degree` argument specifies the degree of the polynomials, here 3. The `role` argument is used to specify the role of the new columns. The default is `predictor`.

Figure 7.2 shows the effect of applying the `step_poly` function to the `x` column. The first polynomial `x_poly_1` is linear, `x_poly_2` is a transformation with a quadratic function, and `x_poly_3` is a cubic function. The polynomials are orthogonal, which means that they are uncorrelated. This is useful when using the polynomials as predictors in a regression model.

```
transformed_long <- transformed %>%
  pivot_longer(c(x_poly_1, x_poly_2, x_poly_3))
ggplot(transformed_long, aes(x = y, y = value,
    color = name, linetype = name)) +
  geom_line() +
  labs(x = "x", y = "x_poly_i", color = "Name", linetype = "Name")
```

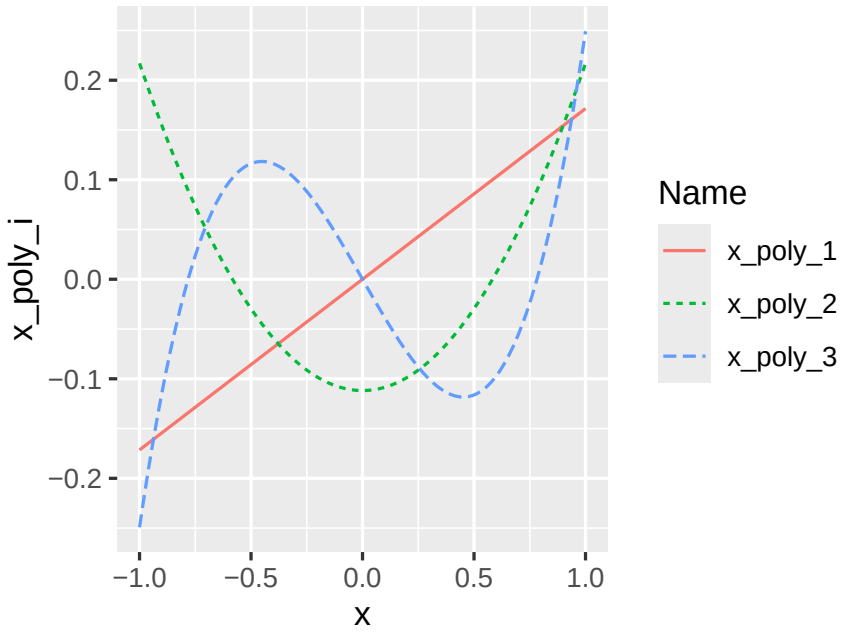


Figure 7.2: Orthogonal polynomials created using `step_poly`

Finally, there are several functions to convert a column to a variety of splines.

- `step_ns`: Natural spline basis functions
- `step_bs`: B-spline basis functions
- `step_spline_b`: Basis splines
- `step_spline_convex`: Convex splines
- `step_spline_monotone`: Monotone splines
- `step_spline_natural`: Natural splines
- `step_spline_nonnegative`: Non-negative splines

7.3 Discretizing numeric variables

Sometimes, it can be useful to discretize a numeric variable, this means, convert the numeric values into a set of factors. This can be used for stepwise linear regression. The `step_discretize` function will convert a numeric variable into a set of factors using the quantiles of the variable.

```
transformed <- recipe(y ~ x, data = xy)  %>%
  step_discretize(x, num_breaks = 5) %>%
  prep() %>%
  bake(new_data = NULL)
```

```
transformed %>%
  head() %>%
  knitr::kable(digits = 4) %>%
  kableExtra::kable_styling(full_width = FALSE)
```

x	y
bin1	-1.0000
bin1	-0.9798
bin1	-0.9596
bin1	-0.9394
bin1	-0.9192
bin1	-0.8990

By default, `step_discretize` will create four factors. Here, we specify `num_breaks = 5` to create five factors. Figure 7.3 shows the effect of applying the `step_discretize` function.

```
ggplot(transformed, aes(x = y, y = x)) +
  geom_point()
```

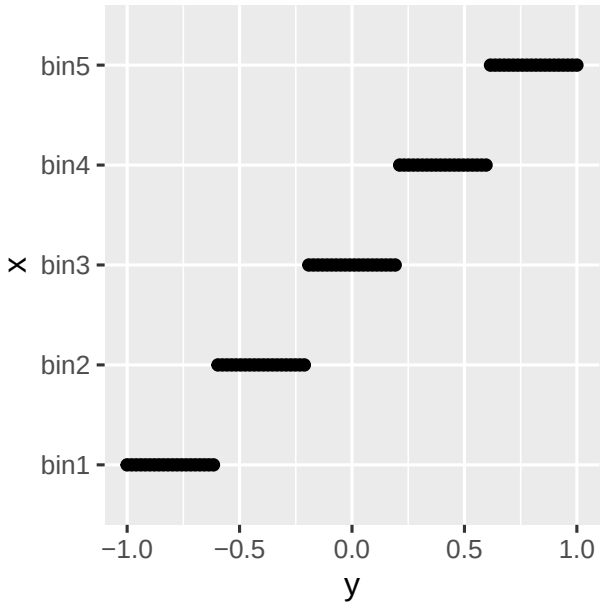


Figure 7.3: Factor levels created using `step_discretize`

An alternative to using quantiles is to specify the breaks explicitly using the `step_cut` function.

```
breaks <- c(-1.1, -0.8, 0.6, 0.7, 0.75)
transformed <- recipe(y ~ x, data = xy)  %>%
  step_cut(x, breaks = breaks) %>%
  prep() %>%
  bake(new_data = NULL)

transformed %>%
  head() %>%
  knitr::kable(digits = 4) %>%
  kableExtra::kable_styling(full_width = FALSE)
```

x	y
[-1.1,-0.8]	-1.0000
[-1.1,-0.8]	-0.9798
[-1.1,-0.8]	-0.9596
[-1.1,-0.8]	-0.9394
[-1.1,-0.8]	-0.9192
[-1.1,-0.8]	-0.8990

Figure 7.4 shows the effect of applying the `step_cut` function.

```
ggplot(transformed, aes(x = y, y = x)) +
  geom_vline(xintercept = breaks, color = "grey") +
  geom_point()
```

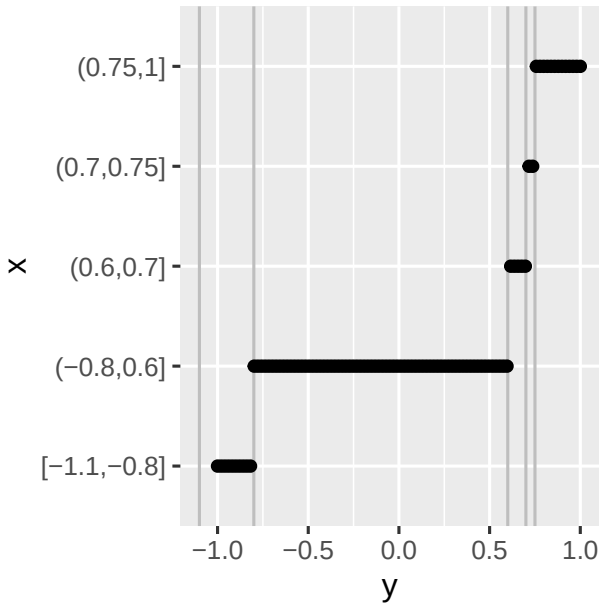


Figure 7.4: Factor levels created using `step_cut`

During training, the range of the data will be used to determine the left and right boundaries of the bins. If a new data point falls outside this range, the value will be mapped to `NA`. will cause problems when predicting new data. To avoid this, we can use the `include_outside_range` argument to specify that values outside the range will be assigned to the first or last bin.

```
transformed <- recipe(y ~ x, data = xy)  %>%
  step_cut(x, breaks = breaks, include_outside_range = TRUE) %>%
  prep() %>%
  bake(new_data = NULL)

transformed %>%
  head() %>%
  knitr::kable(digits = 4) %>%
  kableExtra::kable_styling(full_width = FALSE)
```

x	y
[min,-0.8]	-1.0000
[min,-0.8]	-0.9798
[min,-0.8]	-0.9596
[min,-0.8]	-0.9394
[min,-0.8]	-0.9192

[min,-0.8]	-0.8990

You can see that the lowest range is now labeled as `[min, -0.8]`.

7.4 Data normalization

Several model methods require data to be on the same scale. For example, assume a case where one property has a values in the 1000s, while another property has values between 0 and 10. In a k-nearest neighbor model the first property will dominate any distance measure while the second property will have little influence. To avoid this, we can normalize the data. The `step_normalize` function is used to normalize the data.

```
rec_obj <- recipe(formula, data = data)  %>%
  step_normalize(all_numeric_predictors())
transformed <- rec_obj %>%
  prep() %>%
  bake(new_data = NULL)

transformed %>%
  head() %>%
  knitr::kable(digits = 3)
```

cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	mpg
-0.105	-0.571	-0.535	0.568	-0.610	-0.777	-0.868	1.190	0.424	0.735	21.0
-0.105	-0.571	-0.535	0.568	-0.350	-0.464	-0.868	1.190	0.424	0.735	21.0
-1.225	-0.990	-0.783	0.474	-0.917	0.426	1.116	1.190	0.424	-1.122	22.8
-0.105	0.220	-0.535	-0.966	-0.002	0.890	1.116	-0.814	-0.932	-1.122	21.4
1.015	1.043	0.413	-0.835	0.228	-0.464	-0.868	-0.814	-0.932	-0.503	18.7
-0.105	-0.046	-0.608	-1.565	0.248	1.327	1.116	-0.814	-0.932	-1.122	18.1

Normalization will shift and scale each numerical column, so that its mean is 0 and the standard deviation is 1.

An alternative to normalization is `set_range`. In this case, the data will be transformed to fall into a given range.

```
rec_obj <- recipe(formula, data = data)  %>%
  step_range(all_numeric_predictors())
transformed <- rec_obj %>%
  prep() %>%
  bake(new_data = NULL)

transformed %>%
  head() %>%
  knitr::kable(digits = 3)
```

cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	mpg
0.5	0.222	0.205	0.525	0.283	0.233	0	1	0.5	0.429	21.0
0.5	0.222	0.205	0.525	0.348	0.300	0	1	0.5	0.429	21.0
0.0	0.092	0.145	0.502	0.206	0.489	1	1	0.5	0.000	22.8
0.5	0.466	0.205	0.147	0.435	0.588	1	0	0.0	0.000	21.4
1.0	0.721	0.435	0.180	0.493	0.300	0	0	0.0	0.143	18.7
0.5	0.384	0.187	0.000	0.498	0.681	1	0	0.0	0.000	18.1

The default range is `[0, 1]`. You can specify a different range using the `min` and `max` argument.

> Useful to know
>
> While methods like nearest neighbor‘ require normalization to work properly, other methods are not affected by the scale of the data. For example, decision trees handle each variable independently. However, it can still be beneficial to bring data to the same scale for numerical efficiency and stability.

7.5 Imputing missing data

If you expect your future data to have missing data, it will be useful to derive a strategy to deal with missing data not only for your training data but also for new data. The family of `step_impute_*` functions provide a variety of imputation strategies that are trained on the training data and applied to new data. To demonstrate this functionality, we will create a new dataset that contains missing values.

```
set.seed(123)
data <- datasets::mtcars %>%
  as_tibble(rownames = "car") %>%
```

```
  mutate_at(vars(cyl, wt, am),
    function(x) ifelse(runif(length(x)) < 0.1, NA, x)) %>%
  mutate(
    vs = factor(vs, labels = c("V-shaped", "straight")),
    am = factor(am, labels = c("automatic", "manual")),
  )

missing_cyl <- is.na(data["cyl"])
missing_wt <- is.na(data["wt"])
missing_am <- is.na(data["am"])
missing_rows <- missing_cyl | missing_wt | missing_am

datatable(data[missing_rows, ], rownames = FALSE)
```

file:////private/var/folders/_8/ms0ft4913k3290v7f0g_yfpc0000gn/T/Rtmpo95Vee/file16ad8487e74b8

Show 10 entries Search:

car	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Datsun 710	22.8	4	108	93	3.85		18.61	straight	manual	4	1
Valiant	18.1		225	105	2.76	3.46	20.22	straight	automatic	3	1
Merc 280	19.2	6	167.6	123	3.92	3.44	18.3	straight		4	4
Fiat 128	32.4		78.7	66	4.08	2.2	19.47	straight	manual	4	1
Honda Civic	30.4	4	75.7	52	4.93		18.52	straight	manual	4	2
Ferrari Dino	19.7	6	145	175	3.62		15.5	V-shaped	manual	5	6

Showing 1 to 6 of 6 entries Previous 1 Next

The `mutate_at` function adds about 10% missing data to the columns `cyl`, `wt`, and `am`.

For continuous numeric data, the mean and median are the most common imputation strategies (`step_impute_mean` or `step_impute_median`). For nominal data, the most common value is used (`step_impute_mode`).

```r
transformed <- recipe(mpg ~ ., data = data)  %>%
  step_impute_mean(wt) %>%
  step_impute_median(cyl) %>%
  step_impute_mode(am) %>%
  prep() %>%
  bake(new_data = NULL)

datatable(transformed[missing_rows, ] %>% head(),
  rownames = FALSE) %>%
  formatRound(columns = c("mpg", "cyl", "disp", "hp", "drat",
      "wt", "qsec", "gear", "carb"), digits = 3)
```

file:////private/var/folders/_8/ms0ft4913k3290v7f0g_yfpc0000gn/T/Rtmpo95Vee/file16ad8597e705

Show 10 entries Search:

car	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	mpg
Datsun 710	4.000	108.000	93.000	3.850	3.319	18.610	straight	manual	4.000	1.000	22.800
Valiant	6.000	225.000	105.000	2.760	3.460	20.220	straight	automatic	3.000	1.000	18.100
Merc 280	6.000	167.600	123.000	3.920	3.440	18.300	straight	automatic	4.000	4.000	19.200
Fiat 128	6.000	78.700	66.000	4.080	2.200	19.470	straight	manual	4.000	1.000	32.400
Honda Civic	4.000	75.700	52.000	4.930	3.319	18.520	straight	manual	4.000	2.000	30.400
Ferrari Dino	6.000	145.000	175.000	3.620	3.319	15.500	V-shaped	manual	5.000	6.000	19.700

Showing 1 to 6 of 6 entries Previous 1 Next

In our example, this added the values 3.3188621 to the missing values in the `wt` column, 6 to the missing values in the `cyl` column, and numeric to the missing values in the `am` column.

In some cases, a better approach is to use a model to impute the missing values.

- `step_impute_linear`: Impute numeric variables via a linear model
- `step_impute_bag`: Impute via bagged trees
- `step_impute_knn`: Impute via k-nearest neighbors

We known from exploratory data that the `wt` columns is correlated with the `disp` and `hp` columns. We can use this information to impute the missing values in the `wt` column.

```
transformed <- recipe(mpg ~ ., data = data)  %>%
  step_impute_linear(wt, impute_with = imp_vars(disp, hp)) %>%
  prep() %>%
  bake(new_data = NULL)

datatable(transformed[missing_wt, ], rownames = FALSE) %>%
  formatRound(columns = c("wt"), digits = 3)
```

```
file:////private/var/folders/_8/ms0ft4913k3290v7f0g_yfpc0000gn/T/Rtmpo95Vee/file16ad8437c26f
```

Show 10 entries Search:

car	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	mpg
Datsun 710	4	108	93	3.85	2.390	18.61	straight	manual	4	1	22.8
Honda Civic	4	75.7	52	4.93	2.231	18.52	straight	manual	4	2	30.4
Ferrari Dino	6	145	175	3.62	2.492	15.5	V-shaped	manual	5	6	19.7

Showing 1 to 3 of 3 entries Previous 1 Next

Here we used the `step_impute_linear` function to impute the missing values in the `wt` column. The `impute_with` argument is used to specify the variables that are used to impute the missing values. The `imp_vars` function is required here to specify the variables. We can see that in this case, the imputed values are different.

7.6 Dummy variables

Most methods cannot handle categorical variables without preprocessing. A common approach is to convert a categorical variable with C levels into several new columns that contain only 1 and 0 values. If *reference cell* parametrization is used, $C - 1$ new columns are created for all but the first factor level. In `recipes`, we use the function `step_dummy` to create these new *dummy* variables. Here is an example:

```
penguins <- readr::read_csv("data/penguins_modified.csv.gz") %>%
  sample_frac()
recipe(~ species, data = penguins) %>%
  step_dummy(species, keep_original_cols = TRUE) %>%
  prep() %>%
  bake(new_data = NULL) %>%
  head()
```

```
# A tibble: 6 x 3
  species   species_Chinstrap species_Gentoo
  <fct>                 <dbl>          <dbl>
1 Chinstrap                 1              0
2 Chinstrap                 1              0
3 Gentoo                    0              1
4 Gentoo                    0              1
5 Chinstrap                 1              0
6 Chinstrap                 1              0
```

We set the argument `keep_original_cols` to `TRUE` to include the original variable. By default, it would be removed. We can see that the step creates two new variables `species_Chinstrap` and `species_Gentoo`. They have values of 0 or 1, depending the value of `species`. If the species is *Gentoo*, `species_Gentoo` is set to 1 and the other variable to 0. For *Chinstrap* it is the other way round. For *Adelie*, both values are set to 0. *Adelie* is the reference value. This type of encoding is used for models that cannot deal with correlated data.

An alternative is *one hot encoding*. In this case, we create new columns for each factor level.

```
recipe(~ species, data = penguins) %>%
  step_dummy(species, one_hot = TRUE, keep_original_cols = TRUE) %>%
  prep() %>%
  bake(new_data = NULL) %>%
  head()
```

```
# A tibble: 6 x 4
  species   species_Adelie species_Chinstrap species_Gentoo
  <fct>              <dbl>             <dbl>          <dbl>
1 Chinstrap              0                 1              0
2 Chinstrap              0                 1              0
3 Gentoo                 0                 0              1
4 Gentoo                 0                 0              1
5 Chinstrap              0                 1              0
6 Chinstrap              0                 1              0
```

Setting `one_hot = TRUE` creates the additional column `species_Adelie` which is set to 1 for species *Adelie*. One hot encoding is usually used for k-NN and neural networks to treat each factor equivalently. As can be seen in Figure 7.5, with one hot encoding Euclidean distances between the different factor levels are identical. With *reference cell* encoding, the reference level has the same distance to all other levels and this distance is shorter than the distances between the other levels.

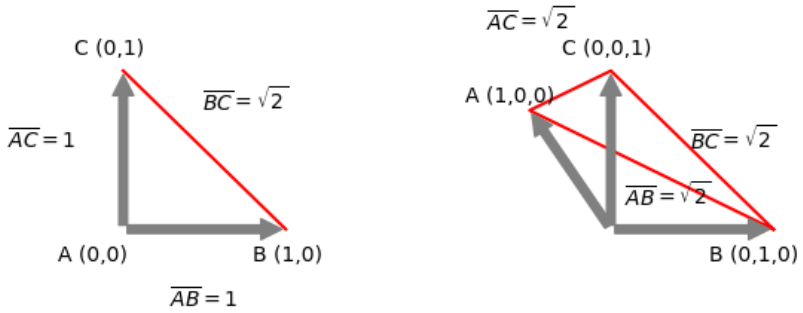


Figure 7.5: Effect of approach to generate dummy variables on distances. Left: reference cell encoding, right: one hot encoding.

To convert all nominal or categorical predictors into dummy variables use:

```
step_dummy(all_nominal_predictors())
```

See Handling categorical predictors for more details on handling categorical variables in *tidymodels* using `recipes`.

7.7 Interactions

The `recipe` package has also a way of defining interaction terms. While this could be done using a formula, the `step_interact` function is particularly useful to define interactions with variables that were created in a previous step.

Here is an example, where we first convert the `vs` predictor into a dummy variable using one-hot-encoding. this replaces the `vs` predictor with `vs_V.shaped` and `vs_straight`. In the next step, we want to create interaction terms of these two predictors with `hp`. We define this using

the formula `~ (vs_V.shaped + vs_straight):hp`. If the factor has several levels, it will be more concise to select the predictors using 'starts_with("vs")'.

```
transformed <- recipe(mpg ~ vs + hp, data = data)  %>%
  step_dummy(vs, one_hot = TRUE) %>%
  step_interact(~ starts_with("vs"):hp) %>%
  prep() %>%
  bake(new_data = NULL)
transformed %>% head(3)
```

```
# A tibble: 3 x 6
     hp   mpg vs_V.shaped vs_straight vs_V.shaped_x_hp vs_straight_x_hp
  <dbl> <dbl>       <dbl>       <dbl>            <dbl>             <dbl>
1   110  21             1           0              110                 0
2   110  21             1           0              110                 0
3    93  22.8           0           1                0                93
```

We can see that `step_interact` creates two new columns, `vs_V.shaped_x_hp` and `vs_straight_x_hp`. The interacting terms are separated using `_x_` (`sep` argument).

The following example adds interaction terms between two factors that were one-hot-encoded.

```
transformed <- recipe(mpg ~ vs + am, data = data)  %>%
  step_dummy(vs, am, one_hot = TRUE) %>%
  step_interact(~ starts_with("vs"):starts_with("am")) %>%
  prep() %>%
  bake(new_data = NULL)
```

```
Warning: ! There are new levels in `am`: NA.
i Consider using step_unknown() (`?recipes::step_unknown()`) before
  `step_dummy()` to handle missing values.
```

```
datatable(transformed %>% head(), rownames = FALSE)
```

```
file:////private/var/folders/_8/ms0ft4913k3290v7f0g_yfpc0000gn/T/Rtmpo95Vee/file16ad83adfd804
```

Show 10 entries Search:

mpg	vs_V.shaped	vs_straight	am_automatic	am_manual	vs_V.shaped_x_am_automatic	vs_V.shaped_x_am_manual	vs
21	1	0	0	1	0	1	
21	1	0	0	1	0	1	
22.8	0	1	0	1	0	0	
21.4	0	1	1	0	0	0	
18.7	1	0	1	0	1	0	
18.1	0	1	1	0	0	0	

Showing 1 to 6 of 6 entries Previous 1 Next

This adds four new columns.

You can also create all possible interactions by using `mpg ~ .:.`. Here, the `.` represents all remaining columns after removing `mpg`.

```
transformed <- recipe(mpg ~ hp + wt + vs, data = data)  %>%
  step_interact(mpg ~ .^2) %>%
  prep() %>%
  bake(new_data = NULL)
```

```
Warning: Categorical variables used in `step_interact()` should probably be avoided;
This can lead to differences in dummy variable values that are produced by
?step_dummy (`?recipes::step_dummy()`). Please convert all involved variables
to dummy variables first.
```

```
datatable(transformed %>% head(), rownames = FALSE)
```

```
file:////private/var/folders/_8/ms0ft4913k3290v7f0g_yfpc0000gn/T/Rtmpo95Vee/file16ad8713856e1
```

Show 10 entries Search:

hp	wt	vs	mpg	hp_x_wt	hp_x_vsstraight	wt_x_vsstraight
110	2.62	V-shaped	21	288.2	0	0
110	2.875	V-shaped	21	316.25	0	0
93		straight	22.8		93	
110	3.215	straight	21.4	353.65	110	3.215
175	3.44	V-shaped	18.7	602	0	0
105	3.46	straight	18.1	363.3	105	3.46

Showing 1 to 6 of 6 entries Previous 1 Next

Alternatively, you can use `mpg ~ .**2` or `mpg ~ .^2` to achieve the same result.

If you only want to include interactions between numerical predictors, you can use the `all_numeric_predictors()` selector like in this formula: `~ all_numeric_predictors() ** 2`

```
transformed <- recipe(mpg ~ hp + disp + vs, data = data)  %>%
  step_interact(~ all_numeric_predictors() ** 2) %>%
  prep() %>%
  bake(new_data = NULL)

datatable(transformed %>% head(), rownames = FALSE)
```

`file:////private/var/folders/_8/ms0ft4913k3290v7f0g_yfpc0000gn/T/Rtmpo95Vee/file16ad82a4b94c`

Note, this will not add quadratic terms.

> **Useful to know**
>
> Table 7.8 shows the results of adding an interaction between `hp` and `disp`. The original predictors have ranges of `hp = [52, 335]` and `disp = [71.1, 472]`. Both ranges are similar. The interaction term however has a much wider and larger range `hp_x_disp =`

Table 7.8: Table created by adding all interaction between numerical predictors

Show 10 entries Search:

hp	disp	vs	mpg	hp_x_disp
110	160	V-shaped	21	17600
110	160	V-shaped	21	17600
93	108	straight	22.8	10044
110	258	straight	21.4	28380
175	360	V-shaped	18.7	63000
105	225	straight	18.1	23625

Showing 1 to 6 of 6 entries Previous 1 Next

`[  3936.4, 101200.0]`.
If you use a model that is based on distances like k-NN, it is important to normalize the data (see Section 7.4). Otherwise, the interaction term will dominate the distances and reduce the influence the main terms can have on the model.

7.8 Principal components

The `recipe` package has several functions that combine multiple columns. Here, we will only discuss the `step_pca` function. It is used to create principal components. For more information on PCA, see Section 18.1 It is recommended to use the `step_normalize` function prior to using `step_pca`.

```
data <- datasets::mtcars %>%
  as_tibble(rownames = "car") %>%
  mutate(
    vs = factor(vs, labels = c("V-shaped", "straight")),
    am = factor(am, labels = c("automatic", "manual")),
  )

transformed <- recipe(mpg ~ ., data = data)  %>%
  step_normalize(all_numeric_predictors()) %>%
```

```
  step_pca(all_numeric_predictors(), num_comp = 3) %>%
  prep() %>%
  bake(new_data = NULL)
transformed %>% head() %>% knitr::kable(digits = 3)
```

car	vs	am	mpg	PC1	PC2	PC3
Mazda RX4	V-shaped	manual	21.0	-0.647	1.183	0.270
Mazda RX4 Wag	V-shaped	manual	21.0	-0.622	0.986	-0.061
Datsun 710	straight	manual	22.8	-2.308	-0.293	0.352
Hornet 4 Drive	straight	automatic	21.4	-0.155	-1.981	0.281
Hornet Sportabout	V-shaped	automatic	18.7	1.628	-0.857	0.933
Valiant	straight	automatic	18.1	-0.107	-2.437	-0.058

The predictors are now reduced to three numerical columns called `PC1`, `PC2`, and `PC3`. The outcome `mpg` and the two categorical predictors `vs` and `am` are left unchanged.

7.9 Filtering variables

So far, we covered preprocessing steps that transform columns into one or more columns. The `recipe` package also contains methods to remove columns from the dataset. The most basic ones are `step_rm` and `step_select`, which remove one or more column from the dataset by name.

Other filters take the information in the column into account. `step_filter_missing` removes columns where the number of missing data surpasses a given threshold. This is useful for columns where imputation is not feasible.

The `step_zv` and `step_nzv` functions remove columns that are constant or almost constant. Such columns contain in general little information and can be removed without limiting the performance of models.

Another source for redundant information are columns that are highly correlated with other columns or columns that are linear combinations of other columns. In some cases, leaving these columns in the dataset can cause numerical problems. The `step_corr` function removes columns that are highly correlated with other columns. The `step_lincomb` function removes columns that are linear combinations of other columns.

Further information

The *tidymodels* package **parsnip** is the package that is responsible to define and fit models. You find detailed information about each of the model types, the specific engines and their options in the documentation.

- https://recipes.tidymodels.org/ is the documentation for the **recipe** package.
- https://recipes.tidymodels.org/reference/index.html lists all the different preprocessing steps that are available in **recipe**
- https://bookdown.org/max/FES/ This book by the authors of *tidymodels* covers many aspects of feature engineering.

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
knitr::include_graphics("images/model_workflow_recipe.png")
library(tidyverse)
library(tidymodels)
library(patchwork)
library(kableExtra)
library(DT)
data <- datasets::mtcars %>% as_tibble(rownames = "car")
data %>%
  head() %>%
  knitr::kable()
formula <- mpg ~ cyl + disp + hp + drat + wt + qsec + vs +
  am + gear + carb
rec_obj <- recipe(formula, data = data)
summary(rec_obj)
rec_obj <- rec_obj %>%
  step_num2factor(vs, transform = function(x) x + 1,
    levels = c("V-shaped", "straight")) %>%
  step_num2factor(am, transform = function(x) x + 1,
    levels = c("automatic", "manual"))
rec_obj %>%
  prep() %>%
  bake(new_data = NULL) %>%
  top_n(4)
```

```
rec_obj <- rec_obj %>%
  step_normalize(all_numeric_predictors())
rec_obj %>%
  prep() %>%
  bake(new_data = NULL) %>%
  top_n(4)
formula <- mpg ~ cyl + disp + hp + drat + wt + qsec + vs +
  am + gear + carb
rec_obj <- recipe(formula, data = data)  %>%
  step_num2factor(vs, transform = function(x) x + 1,
    levels = c("V-shaped", "straight")) %>%
  step_num2factor(am, transform = function(x) x + 1,
    levels = c("automatic", "manual")) %>%
  step_normalize(all_numeric_predictors())
xy <- tibble(
  x = seq(-1, 1, length.out = 100),
  y = seq(-1, 1, length.out = 100)
)

transformed <- recipe(y ~ x, data = xy)  %>%
  step_poly(x, degree = 3) %>%
  prep() %>%
  bake(new_data = NULL)

transformed %>%
  head() %>%
  knitr::kable(digits = 4) %>%
  kableExtra::kable_styling(full_width = FALSE)
transformed_long <- transformed %>%
  pivot_longer(c(x_poly_1, x_poly_2, x_poly_3))
ggplot(transformed_long, aes(x = y, y = value,
    color = name, linetype = name)) +
  geom_line() +
  labs(x = "x", y = "x_poly_i", color = "Name", linetype = "Name")
transformed <- recipe(y ~ x, data = xy)  %>%
  step_discretize(x, num_breaks = 5) %>%
  prep() %>%
  bake(new_data = NULL)

transformed %>%
  head() %>%
  knitr::kable(digits = 4) %>%
```

```
  kableExtra::kable_styling(full_width = FALSE)
ggplot(transformed, aes(x = y, y = x)) +
  geom_point()
breaks <- c(-1.1, -0.8, 0.6, 0.7, 0.75)
transformed <- recipe(y ~ x, data = xy)  %>%
  step_cut(x, breaks = breaks) %>%
  prep() %>%
  bake(new_data = NULL)

transformed %>%
  head() %>%
  knitr::kable(digits = 4) %>%
  kableExtra::kable_styling(full_width = FALSE)
ggplot(transformed, aes(x = y, y = x)) +
  geom_vline(xintercept = breaks, color = "grey") +
  geom_point()
transformed <- recipe(y ~ x, data = xy)  %>%
  step_cut(x, breaks = breaks, include_outside_range = TRUE) %>%
  prep() %>%
  bake(new_data = NULL)

transformed %>%
  head() %>%
  knitr::kable(digits = 4) %>%
  kableExtra::kable_styling(full_width = FALSE)
rec_obj <- recipe(formula, data = data)  %>%
  step_normalize(all_numeric_predictors())
transformed <- rec_obj %>%
  prep() %>%
  bake(new_data = NULL)

transformed %>%
  head() %>%
  knitr::kable(digits = 3)
rec_obj <- recipe(formula, data = data)  %>%
  step_range(all_numeric_predictors())
transformed <- rec_obj %>%
  prep() %>%
  bake(new_data = NULL)

transformed %>%
  head() %>%
```

```
  knitr::kable(digits = 3)
set.seed(123)
data <- datasets::mtcars %>%
  as_tibble(rownames = "car") %>%
  mutate_at(vars(cyl, wt, am),
    function(x) ifelse(runif(length(x)) < 0.1, NA, x)) %>%
  mutate(
    vs = factor(vs, labels = c("V-shaped", "straight")),
    am = factor(am, labels = c("automatic", "manual")),
  )

missing_cyl <- is.na(data["cyl"])
missing_wt <- is.na(data["wt"])
missing_am <- is.na(data["am"])
missing_rows <- missing_cyl | missing_wt | missing_am

datatable(data[missing_rows, ], rownames = FALSE)
transformed <- recipe(mpg ~ ., data = data)  %>%
  step_impute_mean(wt) %>%
  step_impute_median(cyl) %>%
  step_impute_mode(am) %>%
  prep() %>%
  bake(new_data = NULL)

datatable(transformed[missing_rows, ] %>% head(),
  rownames = FALSE) %>%
  formatRound(columns = c("mpg", "cyl", "disp", "hp", "drat",
      "wt", "qsec", "gear", "carb"), digits = 3)
transformed <- recipe(mpg ~ ., data = data)  %>%
  step_impute_linear(wt, impute_with = imp_vars(disp, hp)) %>%
  prep() %>%
  bake(new_data = NULL)

datatable(transformed[missing_wt, ], rownames = FALSE) %>%
  formatRound(columns = c("wt"), digits = 3)
penguins <- readr::read_csv("data/penguins_modified.csv.gz") %>%
  sample_frac()
recipe(~ species, data = penguins) %>%
  step_dummy(species, keep_original_cols = TRUE) %>%
  prep() %>%
  bake(new_data = NULL) %>%
  head()
```

```r
recipe(~ species, data = penguins) %>%
  step_dummy(species, one_hot = TRUE, keep_original_cols = TRUE) %>%
  prep() %>%
  bake(new_data = NULL) %>%
  head()
knitr::include_graphics("images/preprocess_dummy.png")
transformed <- recipe(mpg ~ vs + hp, data = data)  %>%
  step_dummy(vs, one_hot = TRUE) %>%
  step_interact(~ starts_with("vs"):hp) %>%
  prep() %>%
  bake(new_data = NULL)
transformed %>% head(3)
transformed <- recipe(mpg ~ vs + am, data = data)  %>%
  step_dummy(vs, am, one_hot = TRUE) %>%
  step_interact(~ starts_with("vs"):starts_with("am")) %>%
  prep() %>%
  bake(new_data = NULL)
datatable(transformed %>% head(), rownames = FALSE)
transformed <- recipe(mpg ~ hp + wt + vs, data = data)  %>%
  step_interact(mpg ~ .^2) %>%
  prep() %>%
  bake(new_data = NULL)

datatable(transformed %>% head(), rownames = FALSE)
transformed <- recipe(mpg ~ hp + disp + vs, data = data)  %>%
  step_interact(~ all_numeric_predictors() ** 2) %>%
  prep() %>%
  bake(new_data = NULL)

datatable(transformed %>% head(), rownames = FALSE)
data <- datasets::mtcars %>%
  as_tibble(rownames = "car") %>%
  mutate(
    vs = factor(vs, labels = c("V-shaped", "straight")),
    am = factor(am, labels = c("automatic", "manual")),
  )

transformed <- recipe(mpg ~ ., data = data)  %>%
  step_normalize(all_numeric_predictors()) %>%
  step_pca(all_numeric_predictors(), num_comp = 3) %>%
  prep() %>%
  bake(new_data = NULL)
```

```r
transformed %>% head() %>% knitr::kable(digits = 3)
```

Part III

Regression models

8 Training regression models using *tidymodels*

Regression models aim to predict a continuous outcome variable from a set of predictor variables. You already learned about linear regression models in your previous class. In this section, we will learn how to define and train models using the `parsnip` package from *tidymodels*. Figure 8.1 shows how the `parsnip` package is part of the model definition component of the modeling workflow.

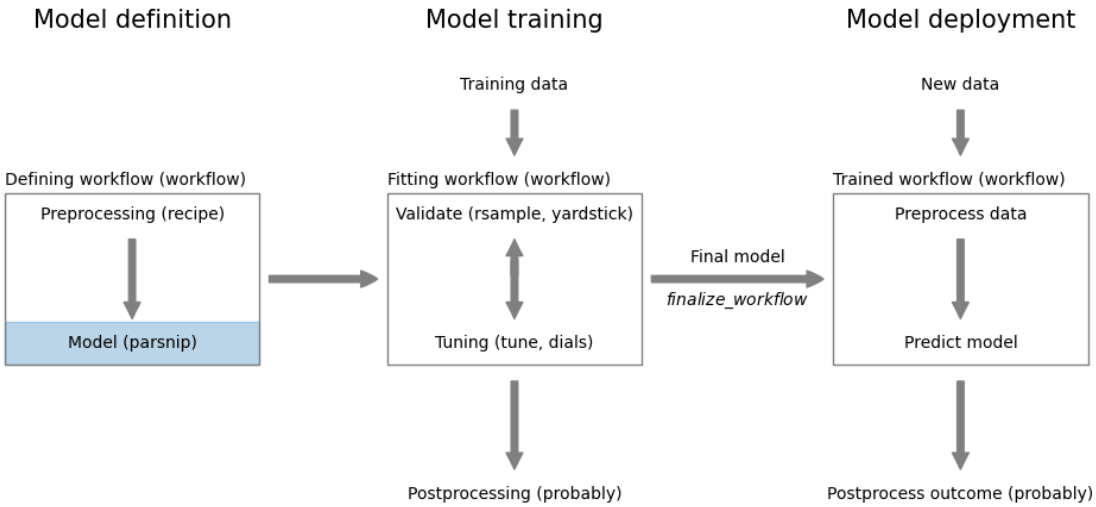


Figure 8.1: Regression model definition using `parsnip`

First, load all the packages we will need.

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

8.1 The `mtcars` dataset

Let's look at the `mtcars` dataset. It is distributed with `R`. We convert it to a tibble and show the first few rows.

```
data <- datasets::mtcars %>%
  as_tibble()
head(data)
```

```
# A tibble: 6 x 11
    mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  21       6   160   110  3.9   2.62  16.5     0     1     4     4
2  21       6   160   110  3.9   2.88  17.0     0     1     4     4
3  22.8     4   108    93  3.85  2.32  18.6     1     1     4     1
4  21.4     6   258   110  3.08  3.22  19.4     1     0     3     1
5  18.7     8   360   175  3.15  3.44  17.0     0     0     3     2
6  18.1     6   225   105  2.76  3.46  20.2     1     0     3     1
```

The `mtcars` dataset contains 32 observations (rows) and 11 variables (columns); check `?mtcars` for details on the dataset. Note that the conversion to a tibble removed the row names. We can preserve the rownames using the `rownames` keyword in the `as_tibble()` function.

Several of the variables are categorical variables. Here, we convert `vs` and `am` to factors and leave the remaining variables as numbers. We first convert the data frame to a tibble and then mutate these variables to factors.

```
data <- datasets::mtcars %>%
  as_tibble(rownames = "car") %>%
  mutate(
    vs = factor(vs, labels = c("V-shaped", "straight")),
    am = factor(am, labels = c("automatic", "manual")),
  )
data
```

```
# A tibble: 32 x 12
   car           mpg   cyl  disp    hp  drat    wt  qsec vs    am     gear  carb
   <chr>       <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fct> <fct> <dbl> <dbl>
 1 Mazda RX4    21       6  160    110  3.9   2.62  16.5 V-sh~ manu~     4     4
 2 Mazda RX4 ~  21       6  160    110  3.9   2.88  17.0 V-sh~ manu~     4     4
 3 Datsun 710   22.8     4  108     93  3.85  2.32  18.6 stra~ manu~     4     1
 4 Hornet 4 D~  21.4     6  258    110  3.08  3.22  19.4 stra~ auto~     3     1
 5 Hornet Spo~  18.7     8  360    175  3.15  3.44  17.0 V-sh~ auto~     3     2
 6 Valiant      18.1     6  225    105  2.76  3.46  20.2 stra~ auto~     3     1
 7 Duster 360   14.3     8  360    245  3.21  3.57  15.8 V-sh~ auto~     3     4
 8 Merc 240D    24.4     4  147.    62  3.69  3.19  20   stra~ auto~     4     2
```

```
 9 Merc 230     22.8     4  141.    95  3.92  3.15  22.9 stra~ auto~     4     2
10 Merc 280     19.2     6  168.   123  3.92  3.44  18.3 stra~ auto~     4     4
# i 22 more rows
```

We now have a preprocessed dataset that we can use to build a model to predict `mpg` using the other variables. To test our model, we create an additional data set with two cars. Note that we apply the same transformations to the new dataset as we did to the training set.

```
new_cars <- tibble(
  car = c("test1", "test2"), cyl = c(4, 6), disp = c(100, 200),
  hp = c(100, 200), drat = c(3, 4), wt = c(2, 3), qsec = c(10, 20),
  vs = c(1, 0), am = c(1, 0), gear = c(3, 4), carb = c(1, 2)
) %>%
  mutate(
    vs = factor(vs, labels = c("V-shaped", "straight")),
    am = factor(am, labels = c("automatic", "manual")),
  )
new_cars
```

```
# A tibble: 2 x 11
  car     cyl  disp    hp  drat    wt  qsec vs       am          gear  carb
  <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fct>    <fct>      <dbl> <dbl>
1 test1     4   100   100     3     2    10 straight manual         3     1
2 test2     6   200   200     4     3    20 V-shaped automatic      4     2
```

8.2 Predicting `mpg` in the `mtcars` dataset using *tidymodels*

Here is how we train a linear regression model using *tidymodels*. The formula specifies the outcome variable and the predictor variables. The formula is defined as `outcome ~ predictor1 + predictor2 + ....`[1] In our case, we want to predict mpg using all the other variables. We could specify this as mpg~.. The . means all the other variables. However, it is better to explicitly list the predictors to avoid mistakes.

```
formula <- mpg ~ cyl + disp + hp + drat + wt + qsec + vs + am +
  gear + carb
model <- linear_reg() %>%
  set_engine("lm") %>%
  fit(formula, data = data)
```

[1]Note that this is a simplified formula. More about formulas later.

The `linear_reg()` function specifies that we want to train a linear regression model. The `set_engine()` function defines the actual model. Here it will be the `lm` model from base-R. We could also use `set_engine("glm")` to use the `glm` function from base-R. The `fit()` function trains the model. The result is an object of class `linear_reg`. Printing the model gives details about the model.

```
model
```

```
parsnip model object


Call:
stats::lm(formula = mpg ~ cyl + disp + hp + drat + wt + qsec +
    vs + am + gear + carb, data = data)

Coefficients:
(Intercept)          cyl         disp           hp         drat           wt
   12.30337     -0.11144      0.01334     -0.02148      0.78711     -3.71530
       qsec   vsstraight     ammanual         gear         carb
    0.82104      0.31776      2.52023      0.65541     -0.19942
```

We can also access the actual model using `model$fit`.

```
model$fit
```

```
Call:
stats::lm(formula = mpg ~ cyl + disp + hp + drat + wt + qsec +
    vs + am + gear + carb, data = data)

Coefficients:
(Intercept)          cyl         disp           hp         drat           wt
   12.30337     -0.11144      0.01334     -0.02148      0.78711     -3.71530
       qsec   vsstraight     ammanual         gear         carb
    0.82104      0.31776      2.52023      0.65541     -0.19942
```

As is usual in *R* for predictive models, we can use the `predict()` function to predict `mpg` for new data.

```
predict(model, new_data = new_cars)
```

```
# A tibble: 2 x 1
  .pred
  <dbl>
1  18.8
2  20.7
```

The `predict()` function returns a tibble with the predicted values. At first glance, it seems that it would be unnecessary to return a tibble. However, `predict` can return additional information, so returning a tibble in this simple case is more consistent.

> Useful to know
>
> The `parsnip::augment` function is shortcut to predict a dataset and return a new tibble that includes the predicted values.
>
> ```
> augment(model, new_data = new_cars)
> ```
>
> ```
> # A tibble: 2 x 12
>   .pred car     cyl  disp    hp  drat    wt  qsec vs       am         gear  carb
>   <dbl> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fct>    <fct>     <dbl> <dbl>
> 1  18.8 test1     4   100   100     3     2    10 straight manual        3     1
> 2  20.7 test2     6   200   200     4     3    20 V-
> shaped automatic     4     2
> ```
>
> The predicted values are added as the new column `.pred`. If the dataset contains a column with the actual values, the predicted values are compared to the actual values and the difference is added as the new column `.resid`.[2] The prefix `.` is used as an indicator for derived columns. It also helps to avoid name clashes with existing columns.

Figure 8.2 shows the actual `mpg` values against the predicted values for the training data.

```
pred_ci <- predict(model, new_data = data, type = "conf_int")
df <- tibble(
  actual = data$mpg,
  predicted = predict(model, new_data = data)$.pred,
  lower = pred_ci$.pred_lower,
  upper = pred_ci$.pred_upper)
```

[2] This only works if you fit a `parsnip` model directly. If the model is created using a workflow, `.resid` is not calculated, even if the outcome is included in the `new_data` column.

```r
ggplot(df,
  aes(x = actual, y = predicted, ymin = lower, ymax = upper)) +
  geom_abline(color = "darkgrey") +
  geom_errorbar(color = "darkgreen") +
  geom_point() +
  labs(x = "Actual mpg", y = "Predicted mpg") +
  coord_fixed(ratio = 1)
```

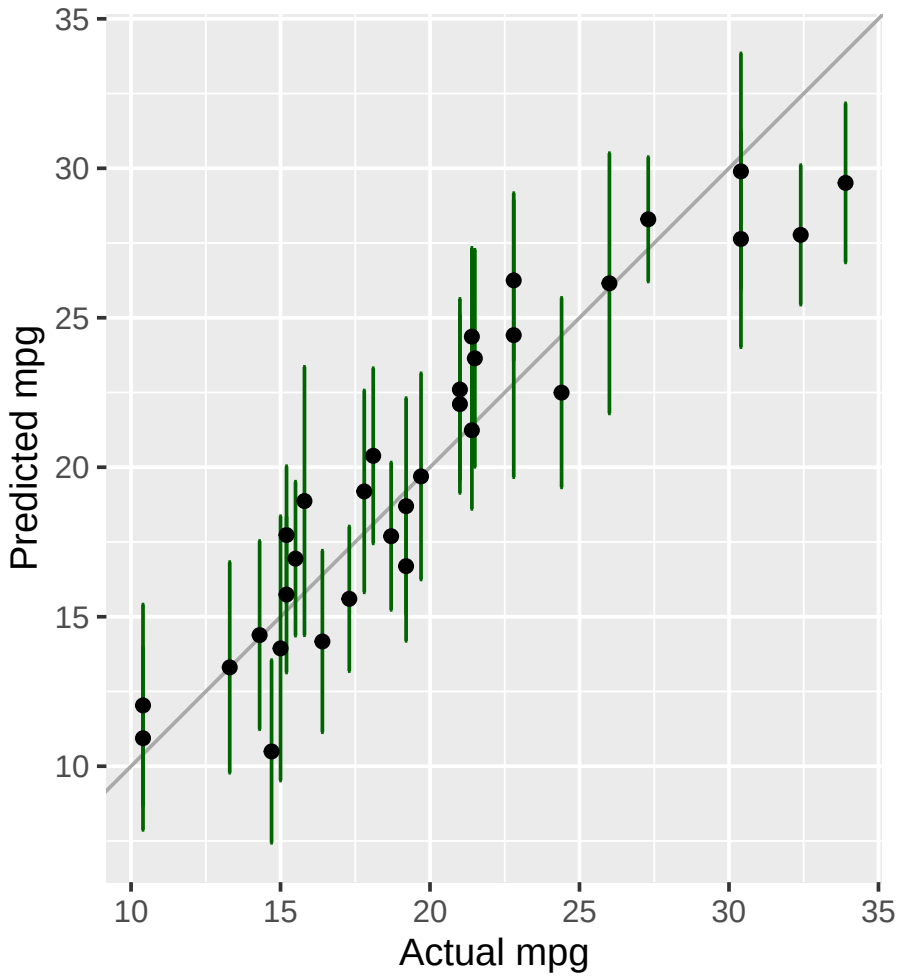


Figure 8.2: Comparing actual against predicted `mpg` values for the *tidymodels* model

In Figure 8.2, we added error bars to show the confidence interval of the predictions. They were calculated using the command `predict(model, new_data=data, type="conf_int")`.

Further information

The *tidymodels* package `parsnip` is the package that is responsible to define and fit models. You find detailed information about each of the model types, the specific engines and their options in the documentation.

- https://parsnip.tidymodels.org/ is the documentation for the `parsnip` package.
- https://parsnip.tidymodels.org/reference/index.html lists all the different model types that are available in `parsnip`
- https://parsnip.tidymodels.org/reference/linear_reg.html is the documentation for the `linear_reg()` function. Here, you find a list of all the *engines* that can be used with `linear_reg()`.
- https://parsnip.tidymodels.org/reference/details_linear_reg_lm.html is the documentation for the `lm` engine. These specific pages will give you more details about the different options that are available for each model.

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
knitr::include_graphics("images/model_workflow_parsnip.png")
library(tidyverse)
library(tidymodels)
data <- datasets::mtcars %>%
  as_tibble()
head(data)
data <- datasets::mtcars %>%
  as_tibble(rownames = "car") %>%
  mutate(
    vs = factor(vs, labels = c("V-shaped", "straight")),
    am = factor(am, labels = c("automatic", "manual")),
  )
data
new_cars <- tibble(
  car = c("test1", "test2"), cyl = c(4, 6), disp = c(100, 200),
  hp = c(100, 200), drat = c(3, 4), wt = c(2, 3), qsec = c(10, 20),
  vs = c(1, 0), am = c(1, 0), gear = c(3, 4), carb = c(1, 2)
) %>%
  mutate(
```

```
    vs = factor(vs, labels = c("V-shaped", "straight")),
    am = factor(am, labels = c("automatic", "manual")),
  )
new_cars
formula <- mpg ~ cyl + disp + hp + drat + wt + qsec + vs + am +
  gear + carb
model <- linear_reg() %>%
  set_engine("lm") %>%
  fit(formula, data = data)
model
model$fit
predict(model, new_data = new_cars)
augment(model, new_data = new_cars)
pred_ci <- predict(model, new_data = data, type = "conf_int")
df <- tibble(
  actual = data$mpg,
  predicted = predict(model, new_data = data)$.pred,
  lower = pred_ci$.pred_lower,
  upper = pred_ci$.pred_upper)

ggplot(df,
  aes(x = actual, y = predicted, ymin = lower, ymax = upper)) +
  geom_abline(color = "darkgrey") +
  geom_errorbar(color = "darkgreen") +
  geom_point() +
  labs(x = "Actual mpg", y = "Predicted mpg") +
  coord_fixed(ratio = 1)
```

9 Measuring performance of regression models

The `yardstick` package from *tidymodels* contains a comprehensive collection of performance metrics for regression and classification models. There are 16 metrics for regression models in the `yardstick` package. You can use these metrics even if you use other packages for modeling.

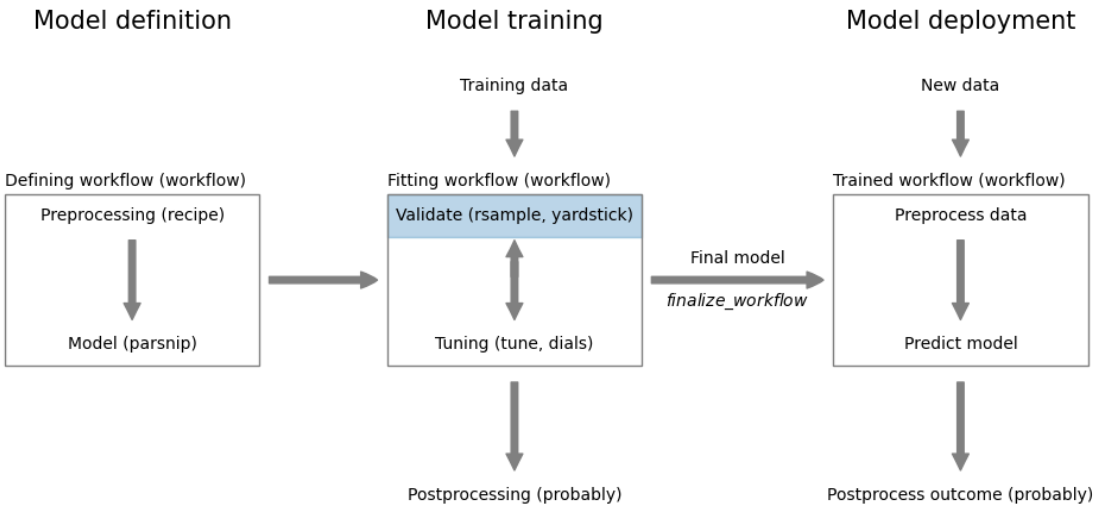


Figure 9.1: Measuring regression performance using `yardstick`

Figure 9.1 shows how the `yardstick` package fits into the modeling workflow. After you have trained a regression model, you can use the `yardstick` package to measure its performance on training data or on new data.

The package is loaded automatically when you load *tidymodels*.

```
library(tidymodels)
```

```
-- Attaching packages ------------------------------------- tidymodels 1.3.0 --
```

```
v broom        1.0.8     v recipes      1.3.1
v dials        1.4.0     v rsample      1.3.0
v dplyr        1.1.4     v tibble       3.3.0
v ggplot2      3.5.2     v tidyr        1.3.1
v infer        1.0.8     v tune         1.3.0
v modeldata    1.4.0     v workflows    1.2.0
v parsnip      1.3.1     v workflowsets 1.1.0
v purrr        1.0.4     v yardstick    1.3.2

-- Conflicts ---------------------------------------- tidymodels_conflicts() --
x purrr::discard() masks scales::discard()
x dplyr::filter()  masks stats::filter()
x dplyr::lag()     masks stats::lag()
x recipes::step()  masks stats::step()
```

9.1 Build a regression model

As described in more detail in Chapter 8, we first build a model for the `mtcars` dataset.

```
# prepare
data <- datasets::mtcars %>%
  as_tibble(rownames = "car") %>%
  mutate(
    vs = factor(vs, labels = c("V-shaped", "straight")),
    am = factor(am, labels = c("automatic", "manual")),
  )

# fit model
formula <- mpg ~ cyl + disp + hp + drat + wt + qsec + vs + am +
  gear + carb
model <- linear_reg() %>%
  set_engine("lm") %>%
  fit(formula, data = data)
```

9.2 Calculate performance metrics

The `yardstick` package contains 16 metrics for regression models. The `metrics()` function calculates the most important metrics for a given model.

```
metrics(augment(model, data), truth = mpg, estimate = .pred)
```

```
# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rmse    standard       2.15
2 rsq     standard       0.869
3 mae     standard       1.72
```

We use the `parsnip::augment` function to add the predictions to the original dataset. For a regression model, this adds prediction (`.pred`) and residuals (`.resid`).[1] The `truth` argument specifies the name of the column with the true values, here `mpg`. The `estimate` argument specifies the name of the column with the predicted values, here `.pred`.

The `metrics()` function calculates the following metrics:

- `rmse`: root mean squared error (RMSE)
- `rsq`: R-squared (R2)
- `mae`: mean absolute error (MAE)

This should be enough for most purposes. If you want to calculate other metrics, you can use the `yardstick::metric_set()` function. This function takes a list of metrics and returns a function that calculates all metrics in the list. For example, if your dataset has a few outlier values, it can be useful to look at robust metrics like MAE. Here, we combine `mae` and `huber_loss` into a custom metric set.

```
robust_metric <- metric_set(mae, huber_loss)
robust_metric(augment(model, data), truth = mpg, estimate = .pred)
```

```
# A tibble: 2 x 3
  .metric    .estimator .estimate
  <chr>      <chr>          <dbl>
1 mae        standard        1.72
2 huber_loss standard        1.30
```

Todo

- Go to the yardstick website at https://yardstick.tidymodels.org/ and get an overview of the various metrics.

[1]

Further information

- https://yardstick.tidymodels.org/ is the documentation for the `yardstick` package.
- https://yardstick.tidymodels.org/reference/index.html lists all the different metrics that are available in `yardstick`.

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
knitr::include_graphics("images/model_workflow_validate.png")
library(tidymodels)
# prepare
data <- datasets::mtcars %>%
  as_tibble(rownames = "car") %>%
  mutate(
    vs = factor(vs, labels = c("V-shaped", "straight")),
    am = factor(am, labels = c("automatic", "manual")),
  )

# fit model
formula <- mpg ~ cyl + disp + hp + drat + wt + qsec + vs + am +
  gear + carb
model <- linear_reg() %>%
  set_engine("lm") %>%
  fit(formula, data = data)
metrics(augment(model, data), truth = mpg, estimate = .pred)
robust_metric <- metric_set(mae, huber_loss)
robust_metric(augment(model, data), truth = mpg, estimate = .pred)
```

Part IV

Classification models

10 Training classification models using *tidymodels*

Regression models predict a quantitative, continuous numerical outcome. In contrast, classification models predict a qualitative categorical outcome variable from a set of predictor variables. Like regression models, classification models are defined with the functions from the `parsnip` package.

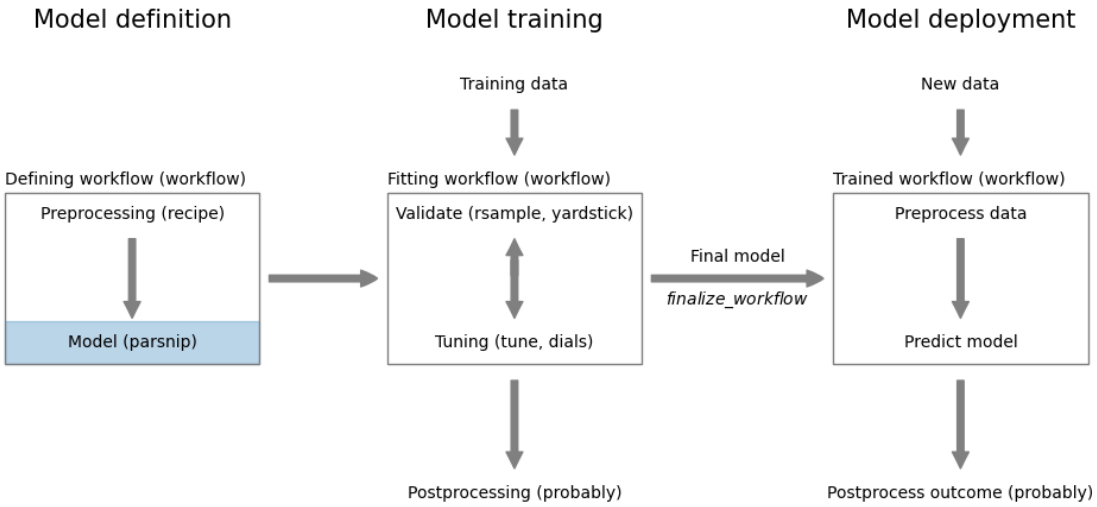


Figure 10.1: Classification model definition using `parsnip`

In this section, we will learn how to train a logistic regression model using the *tidymodels* package. Despite its name, logistic regression is a classification model.

First, load all the packages we will need.

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

10.1 The `UniversalBank` dataset

Let's look at the `UniversalBank` dataset. It is available at https://gedeck.github.io/DS-6030/datasets/UniversalBank.csv.gz. We download it using `readr::read_csv`.

```
file <-
  "https://gedeck.github.io/DS-6030/datasets/UniversalBank.csv.gz"
data <- read_csv(file)
```

```
Rows: 5000 Columns: 14
-- Column specification --------------------------------------------------------
Delimiter: ","
dbl (14): ID, Age, Experience, Income, ZIP Code, Family, CCAvg, Education, M...

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
head(data)
```

```
# A tibble: 6 x 14
     ID   Age Experience Income `ZIP Code` Family CCAvg Education Mortgage
  <dbl> <dbl>      <dbl>  <dbl>      <dbl>  <dbl> <dbl>     <dbl>    <dbl>
1     1    25          1     49      91107      4   1.6         1        0
2     2    45         19     34      90089      3   1.5         1        0
3     3    39         15     11      94720      1   1           1        0
4     4    35          9    100      94112      1   2.7         2        0
5     5    35          8     45      91330      4   1           2        0
6     6    37         13     29      92121      4   0.4         2      155
# i 5 more variables: `Personal Loan` <dbl>, `Securities Account` <dbl>,
#   `CD Account` <dbl>, Online <dbl>, CreditCard <dbl>
```

The *synthetic* dataset contains information about 5000 customers of a bank. The bank wants to know which customers are likely to accept a personal loan. The dataset contains 14 variables. The variable `Personal Loan` is the outcome variable. It is a binary variable that indicates whether the customer accepted the personal loan. The remaining variables are the predictor variables. Details can be found at https://gedeck.github.io/DS-6030/datasets/UniversalBank.html.

Next we need to preprocess the dataset:

```r
data <- data %>%
  select(-c(ID, `ZIP Code`)) %>%
  rename(
    Personal.Loan = `Personal Loan`,
    Securities.Account = `Securities Account`,
    CD.Account = `CD Account`
  ) %>%
  mutate(
    Personal.Loan = factor(Personal.Loan, labels = c("Yes", "No"),
      levels = c(1, 0)),
    Education = factor(Education,
      labels = c("Undergrad", "Graduate", "Advanced")),
  )
str(data)  # compact representation of the data
```

```
tibble [5,000 x 12] (S3: tbl_df/tbl/data.frame)
 $ Age               : num [1:5000] 25 45 39 35 35 37 53 50 35 34 ...
 $ Experience        : num [1:5000] 1 19 15 9 8 13 27 24 10 9 ...
 $ Income            : num [1:5000] 49 34 11 100 45 29 72 22 81 180 ...
 $ Family            : num [1:5000] 4 3 1 1 4 4 2 1 3 1 ...
 $ CCAvg             : num [1:5000] 1.6 1.5 1 2.7 1 0.4 1.5 0.3 0.6 8.9 ...
 $ Education         : Factor w/ 3 levels "Undergrad","Graduate",..: 1 1 1 2 2 2 2 3 2 3 ...
 $ Mortgage          : num [1:5000] 0 0 0 0 0 155 0 0 104 0 ...
 $ Personal.Loan     : Factor w/ 2 levels "Yes","No": 2 2 2 2 2 2 2 2 2 1 ...
 $ Securities.Account: num [1:5000] 1 1 0 0 0 0 0 0 0 0 ...
 $ CD.Account        : num [1:5000] 0 0 0 0 0 0 0 0 0 0 ...
 $ Online            : num [1:5000] 0 0 0 0 0 1 1 0 1 0 ...
 $ CreditCard        : num [1:5000] 0 0 0 0 1 0 0 1 0 0 ...
```

The preprocessing consists of the following steps. First, we remove two columns. `ID` is customer specific and `ZIP Code` is a categorical variable with too many categories. Second, we rename the columns to remove the spaces. This makes it easier to work with the data. Third, we convert the `Personal.Loan` and `Education` variables to factors. In principle, one could convert the variables `Securities.Account`, `CD.Account`, `Online`, and `CreditCard` to factors as well. However, as they have only two levels, we will leave them as numbers.[1]

> **Useful to know**
>
> The outcome variable `Personal.Loan` is converted to a factor despite what we just said. This is **important**! It tells *tidymodels* that we want to train a classification model. If we

[1] You could convert them to make it easier to interpret the model coefficients.

would leave it as a number, the package would assume that we want to train a regression model. Several other packages use the same convention.
An additional advantage is that predictions will be more informative leading to easier to read predictions. In our case, the predictions will be `Yes` or `No` instead of `1` or `0`.

Let's also create a new dataset with new customers. We will use this dataset to predict whether the customer will accept a personal loan.

```
new_customer <- tibble(Age = 40, Experience = 10, Income = 84,
  Family = 2, CCAvg = 2, Education = 2, Mortgage = 0,
  Securities.Account = 0, CD.Account = 0, Online = 1,
  CreditCard = 1) %>%
  mutate(Education = factor(Education,
      labels = c("Undergrad", "Graduate", "Advanced"),
      levels = c(1, 2, 3)))
new_customer
```

```
# A tibble: 1 x 11
    Age Experience Income Family CCAvg Education Mortgage Securities.Account
  <dbl>      <dbl>  <dbl>  <dbl> <dbl> <fct>        <dbl>              <dbl>
1    40         10     84      2     2 Graduate         0                  0
# i 3 more variables: CD.Account <dbl>, Online <dbl>, CreditCard <dbl>
```

Note that we need to convert the `Education` variable to a factor. Otherwise, the prediction will fail.

10.2 *Tidymodels*: predicting `Personal.Loan` in the `UniversalBank` dataset

Defining and training the classification models in *tidymodels* is very similar to training regression models.

```
formula <- Personal.Loan ~ Age + Experience + Income + Family +
  CCAvg + Education + Mortgage + Securities.Account + CD.Account +
  Online + CreditCard

model <- logistic_reg() %>%
  set_engine("glm") %>%
  fit(formula, data = data)
```

We will use the `logistic_reg()` function to define a logistic regression model. The `set_engine()` function defines the actual model. Here it will be the `glm` model from base-R. We could also use `set_engine("glmnet")` to use the model from the `glmnet` package. The `fit()` function finally trains the model.

The result is an object of class `logistic_reg`. Printing the model gives details about the model.

```
model
```

```
parsnip model object


Call:  stats::glm(formula = Personal.Loan ~ Age + Experience + Income +
    Family + CCAvg + Education + Mortgage + Securities.Account +
    CD.Account + Online + CreditCard, family = stats::binomial,
    data = data)

Coefficients:
       (Intercept)                 Age          Experience              Income
        12.3105489           0.0359174          -0.0450379          -0.0601830
            Family               CCAvg   EducationGraduate   EducationAdvanced
        -0.6181693          -0.1633508          -3.9653781          -4.0640537
          Mortgage  Securities.Account          CD.Account              Online
        -0.0007105           0.8701362          -3.8389223           0.7605294
        CreditCard
         1.0382002

Degrees of Freedom: 4999 Total (i.e. Null);  4987 Residual
Null Deviance:      3162
Residual Deviance: 1172     AIC: 1198
```

As we learned for the linear regression model, we can access the actual model using `model$fit`.

```
model$fit
```

```
Call:  stats::glm(formula = Personal.Loan ~ Age + Experience + Income +
    Family + CCAvg + Education + Mortgage + Securities.Account +
    CD.Account + Online + CreditCard, family = stats::binomial,
    data = data)
```

```
Coefficients:
       (Intercept)                 Age          Experience              Income
        12.3105489           0.0359174          -0.0450379          -0.0601830
            Family              CCAvg   EducationGraduate   EducationAdvanced
        -0.6181693          -0.1633508          -3.9653781          -4.0640537
          Mortgage  Securities.Account          CD.Account              Online
        -0.0007105           0.8701362          -3.8389223           0.7605294
        CreditCard
         1.0382002

Degrees of Freedom: 4999 Total (i.e. Null);  4987 Residual
Null Deviance:      3162
Residual Deviance: 1172     AIC: 1198
```

Use the `predict()` function to predict the outcome for the new customer.

```
predict(model, new_data = new_customer)
```

```
# A tibble: 1 x 1
  .pred_class
  <fct>
1 No
```

The model predicts that the customer will not accept the loan offer. The predicted class is in the `.pred_class` column. Classification models can also return a *probability* for the prediction. Use `type="prob"` with the `predict` function.

```
predict(model, new_data = new_customer, type = "prob")
```

```
# A tibble: 1 x 2
  .pred_Yes .pred_No
      <dbl>    <dbl>
1    0.0109    0.989
```

Our new customer has a very high probabily to not accept the offer; `.pred_No = 0.989`. The probability for the other `Yes` class is `.pred_Yes = 0.011`.

Useful to know

The `parsnip::augment` function is shortcut to predict a dataset and return a new tibble that includes the predicted values.

```
augment(model, new_data = new_customer)
```

```
# A tibble: 1 x 14
  .pred_class .pred_Yes .pred_No   Age Experience Income Family CCAvg Education
  <fct>           <dbl>    <dbl> <dbl>      <dbl>  <dbl>  <dbl> <dbl> <fct>
1 No             0.0109    0.989    40         10     84      2     2 Graduate
# i 5 more variables: Mortgage <dbl>, Securities.Account <dbl>,
#   CD.Account <dbl>, Online <dbl>, CreditCard <dbl>
```

The `augment` function returns a tibble that contains all the columns from the original dataset and adds the columns `.pred_class` and `.pred_No` and `.pred_Yes`.

10.3 Visualizing the overall model performance using a ROC curve

There are various ways of analyzing the performance of a classification model. We will discuss this in more detail in Chapter 11. Here, we will create a ROC curve to visualize the performance of our model. A ROC curve can tell you how well the model can distinguish between the two classes.

Figure 10.2 demonstrates how the class separation, the ROC curve and the AUC are related. The density plots in the top row show how the predicted probabilities are distributed for two classes. On the left, we have a model that hardly separates the two classes. On the right, the model separates the two classes very well. The corresponding ROC curves are shown in the second row. The ROC curves are the solid lines. The dashed line is the ROC curve for a random model. For the weakest model, the ROC curve is close to the random model. For the strongest model, the ROC curve gets closer and closer to the ideal model (grey lines). The AUC is the area under the ROC curve. The AUC for the weakest model is close to 0.5, which is the same as for the random model. The AUC for the strongest model is close to 1, which is the best possible value.

Let's see how it looks like for our model. We will use `yardstick::roc_curve` to calculate the ROC curve.

```
augment(model, new_data = data) %>%
  roc_curve(Personal.Loan, .pred_Yes, event_level = "first") %>%
  autoplot()
```

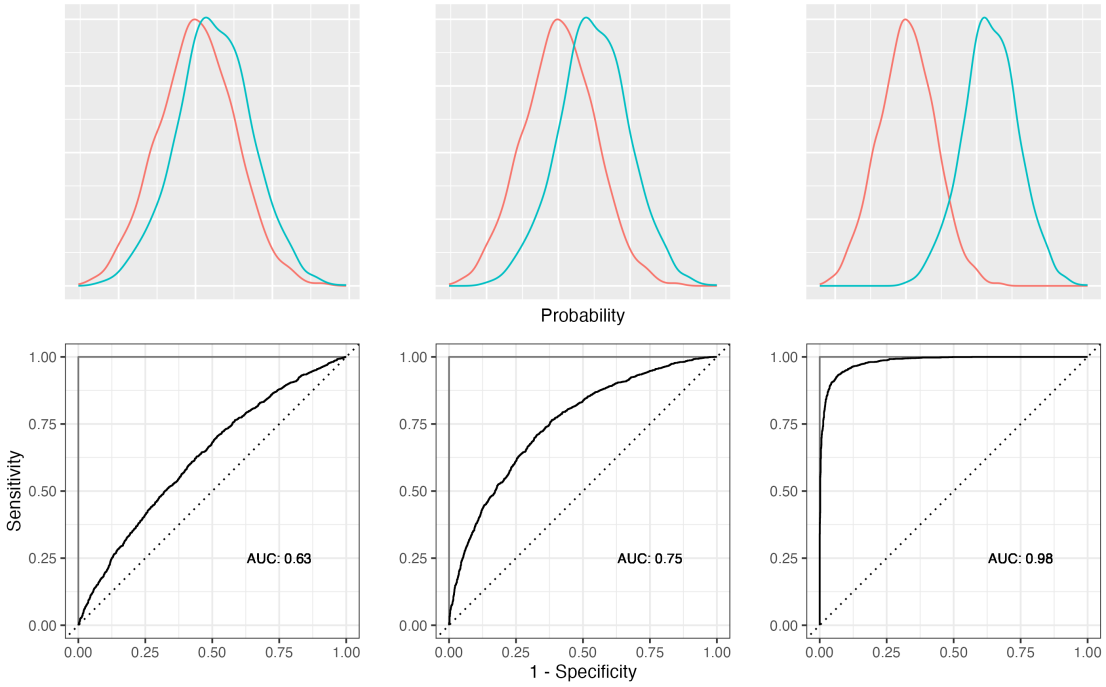


Figure 10.2: Relationship between class separation, ROC curves and AUC

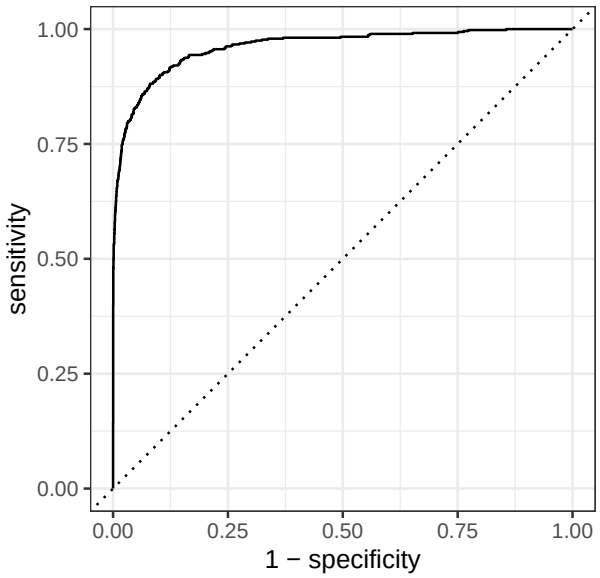


Figure 10.3: ROC curve of the Universal Bank classification model

The `augment()` function adds the predicted class probabilities to the original dataset. This is passed into the `yardstick::roc_curve` function. We need to specify the actual outcome variable (`Personal.Loan`) and the predicted probability for the event that we are interested (`.pred_Yes`). The `event_level` argument specifies which class is the event. In our case, we are interested in the event that the customer accepts the loan. This is the first level, hence `event_level="first"`. The `roc_curve()` function returns a tibble with the false positive rate (FPR) and the true positive rate (TPR). The `autoplot()` function creates the ROC curve as shown in Figure 10.3, but you could also use `ggplot2` to create your own plot.

Further information

The *tidymodels* package `parsnip` is the package that is responsible to define and fit models. You find detailed information about each of the model types, the specific engines and their options in the documentation.

- https://parsnip.tidymodels.org/ is the documentation for the `parsnip` package.
- https://parsnip.tidymodels.org/reference/ lists all the different model types that are available in `parsnip`
- https://parsnip.tidymodels.org/reference/logistic_reg.html is the documentation for the `logistic_reg()` function. Here, you find a list of all the *engines* that can be used with `logistic_reg()`.
- https://parsnip.tidymodels.org/reference/details_logistic_reg_glm.html is the documentation for the `glm` engine. These specific pages will give you more details about the different options that are available for each model.

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
knitr::include_graphics("images/model_workflow_parsnip.png")
library(tidyverse)
library(tidymodels)
file <-
  "https://gedeck.github.io/DS-6030/datasets/UniversalBank.csv.gz"
data <- read_csv(file)
head(data)
data <- data %>%
  select(-c(ID, `ZIP Code`)) %>%
  rename(
    Personal.Loan = `Personal Loan`,
    Securities.Account = `Securities Account`,
    CD.Account = `CD Account`
  ) %>%
  mutate(
    Personal.Loan = factor(Personal.Loan, labels = c("Yes", "No"),
      levels = c(1, 0)),
    Education = factor(Education,
      labels = c("Undergrad", "Graduate", "Advanced")),
  )
str(data)  # compact representation of the data
new_customer <- tibble(Age = 40, Experience = 10, Income = 84,
  Family = 2, CCAvg = 2, Education = 2, Mortgage = 0,
  Securities.Account = 0, CD.Account = 0, Online = 1,
  CreditCard = 1) %>%
  mutate(Education = factor(Education,
      labels = c("Undergrad", "Graduate", "Advanced"),
      levels = c(1, 2, 3)))
new_customer
formula <- Personal.Loan ~ Age + Experience + Income + Family +
  CCAvg + Education + Mortgage + Securities.Account + CD.Account +
  Online + CreditCard

model <- logistic_reg() %>%
  set_engine("glm") %>%
  fit(formula, data = data)
```

```
model
model$fit
predict(model, new_data = new_customer)
predict(model, new_data = new_customer, type = "prob")
augment(model, new_data = new_customer)
knitr::include_graphics("images/roc-auc-class-separation.png")
augment(model, new_data = data) %>%
  roc_curve(Personal.Loan, .pred_Yes, event_level = "first") %>%
  autoplot()
```

11 Measuring performance of classification models

In Chapter 9, we learned how to use the `yardstick` package to measure the performance of a regression model. This package contains also a large collection of performance metrics for classification models.

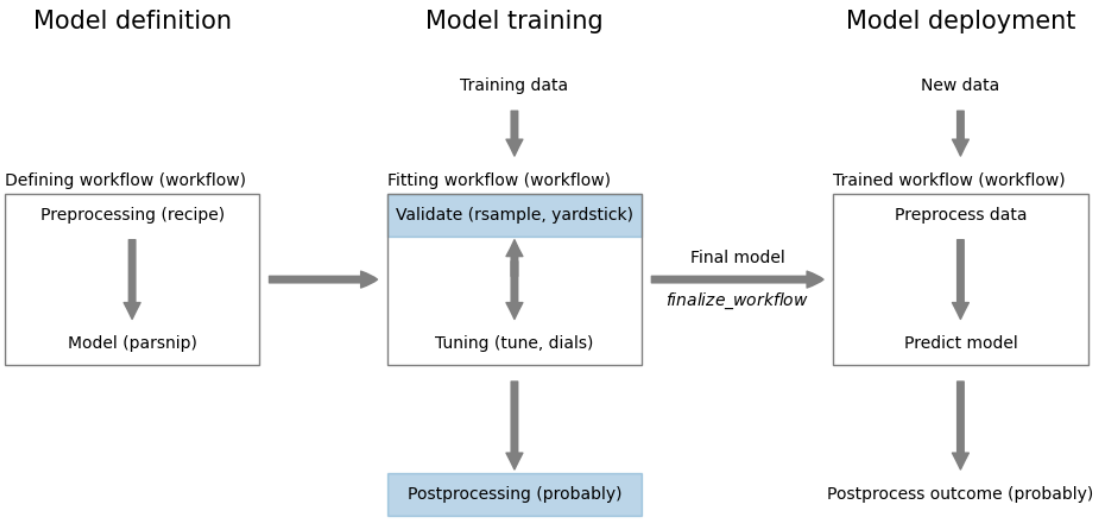


Figure 11.1: Measuring classification performance using `yardstick`

We can divide the classification metrics into two types. The first type requires a hard, class prediction, these are called *classification metrics* in `yardstick`. The second type are metrics that consider the relationship between predicted probabilities and actual class. In `yardstick`, these are referred to as *class probability metrics*. Finally, `yardstick` provides a number of curves (e.g. ROC curves) that can be used to visualize the performance of a classification model.

In this chapter, we will also cover threshold selection using the `probably` package (Section 11.1.2).

Let's demonstrate various measures using the logistic regression classification from the previous Chapter 10.

```
library(tidyverse)
library(tidymodels)
library(yardstick)
library(probably)  # for exploring thresholds
library(patchwork) # for combining plots
```

```
file <-
  "https://gedeck.github.io/DS-6030/datasets/UniversalBank.csv.gz"
data <- read_csv(file)
data <- data %>%
  select(-c(ID, `ZIP Code`)) %>%
  rename(
    Personal.Loan = `Personal Loan`,
    Securities.Account = `Securities Account`,
    CD.Account = `CD Account`
  ) %>%
  mutate(
    Personal.Loan = factor(Personal.Loan, labels = c("Yes", "No"),
      levels = c(1, 0)),
    Education = factor(Education,
      labels =  c("Undergrad", "Graduate", "Advanced")),
  )
formula <- Personal.Loan ~ Income + Family + CCAvg + Education +
  Mortgage + Securities.Account + CD.Account + Online + CreditCard
model <- logistic_reg() %>%
  set_engine("glm") %>%
  fit(formula, data = data)
data <- augment(model, new_data = data)
```

11.1 Classification metrics

The basis of *classification metrics* is the confusion matrix. The confusion matrix is a table that shows the number of correct and incorrect predictions made by a classification model. The confusion matrix is constructed as follows using `yardstick::conf_mat`:

```
cm <- data %>%
  conf_mat(truth = Personal.Loan, estimate = .pred_class)
cm
```

```
          Truth
Prediction  Yes   No
       Yes  323   49
       No   157 4471
```

Here, the confusion matrix lists the predicted class in the rows and the actual class in the column. The diagonal elements are the correct predictions. The off-diagonal elements are the number of incorrect predictions. For example, we can see that the model predicted 49 `Yes` when the actual class was `No`.

The `yardstick` package provides an `autoplot` function for the confusion matrix. The `type` argument specifies the type of plot. The `mosaic` type gives a mosaic plot where areas represent the number of data points. The `heatmap` type is a heatmap. Figure 11.2 shows both types side by side.

```
g1 <- autoplot(cm, type = "mosaic")
g2 <- autoplot(cm, type = "heatmap")
g1 + g2
```

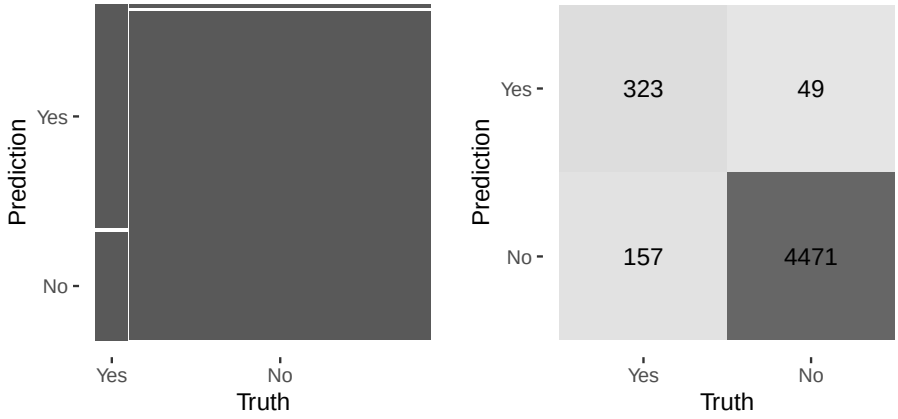


Figure 11.2: Visual representation of the confusion matrix using `autoplot`

> **Useful to know**
>
> The definition of the confusion matrix is not standardized. There are other packages that swap the predicted and actual classes in the matrix. Always check which convention is used for the representation of the confusion matrix.

Using the confusion matrix, we can calculate various classification metrics. For example, the accuracy is the proportion of correct predictions.

```
(cm$table[1, 1] + cm$table[2, 2]) / sum(cm$table)
```

```
[1] 0.9588
```

Accuracy is calculated as the sum of the diagonal divided by the total number of cases. Here, we get an accuracy of 0.9588. `yardstick` provides a large variety of metrics. The following calculates the accuracy using `yardstick::accuracy`:

```
yardstick::accuracy(data, truth = Personal.Loan,
  estimate = .pred_class)
```

```
# A tibble: 1 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.959
```

Other metrics are:

- `sensitivity`: Sensitivity
- `specificity`: Specificity
- `recall`: Recall
- `precision`: Precision
- `mcc`: Matthews correlation coefficient
- `j_index`: J-index
- `f_meas`: F-measure
- `kap`: Kappa
- `ppv`: Positive predictive value
- `npv`: Negative predictive value
- `bal_accuracy`: Balanced accuracy
- `detection_prevalence`: Detection prevalence

The function `yardstick::metrics` calculates two commonly used metrics `accuracy` and `kappa`:

```
yardstick::metrics(data, Personal.Loan, .pred_class)
```

```
# A tibble: 2 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.959
2 kap      binary         0.736
```

You can also define your own combination of metrics and use that to calculate multiple metrics at once.

```
my_metrics <- metric_set(sens, spec, j_index)
my_metrics(data, truth = Personal.Loan, estimate = .pred_class)
```

```
# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 sens    binary         0.673
2 spec    binary         0.989
3 j_index binary         0.662
```

> Useful to know
>
> While you can rely on the default value picked by `yardstick` for the event of interest, it is good practice to specify the event of interest.

11.1.1 Specifying the event of interest

Metrics like `accuracy`, `kap`, or `j_index` treat both outcome classes equally important. However, in many cases, we are interested in the performance of the model for one of the classes. For example, in the case of a medical test, we are interested in the performance of the test for the positive class, i.e. the class that indicates the presence of a disease. Metrics like `sensitivity` or `specificity` have this dependency.

```
my_metrics(data, truth = Personal.Loan, estimate = .pred_class)
```

```
# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 sens    binary         0.673
2 spec    binary         0.989
3 j_index binary         0.662
```

Considering the confusion matrix,

```
          Truth
Prediction  Yes   No
       Yes  323   49
       No   157 4471
```

we can see that `sensitivity`, the true positive rate, was calculated for the `Yes` class; $323/(323+157) = 0.6729167$. `specificity`, the true negative rate, was calculated for the `No` class; $4471/(49+4471) = 0.6729167$. By default, `yardstick` assumes the that first level is the event of interest. In our example, the first level is `Yes`.

However, if we are interested in the `No` class, we can change this by specifying the event of interest using the `event_level` argument.

```
my_metrics(data, truth = Personal.Loan, estimate = .pred_class,
  event_level = "second")
```

```
# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 sens    binary         0.989
2 spec    binary         0.673
3 j_index binary         0.662
```

11.1.2 Thresholds

In order to predict a class, we need to map the proability (or score) p calculated by a classification method to a class by applying a threshold.

$$\text{class} = \begin{cases} \text{Yes} & \text{if } p > \text{threshold} \\ \text{No} & \text{otherwise} \end{cases} \tag{11.1}$$

The classification metrics are calculated using a threshold of 0.5. Using a different threshold, the confusion matrix and therefore all derived metrics change. Figure 11.3 demonstrates this using our example.

The two density plots show the distribution of the predicted probabilities for the two classes; blue for the `Yes` class and red for the `No` class. The vertical lines indicate the thresholds. The confusion matrix shows the number of cases predicted as *Yes* and *No*. Accuracy, sensitivity, and specificity are calculated for the three thresholds.

```
performance <- probably::threshold_perf(data, Personal.Loan,
  .pred_Yes, c(0.1, 0.5, 0.9), event_level = "first",
  metrics = yardstick::metric_set(
    yardstick::accuracy,
    yardstick::specificity,
    yardstick::sensitivity,
```

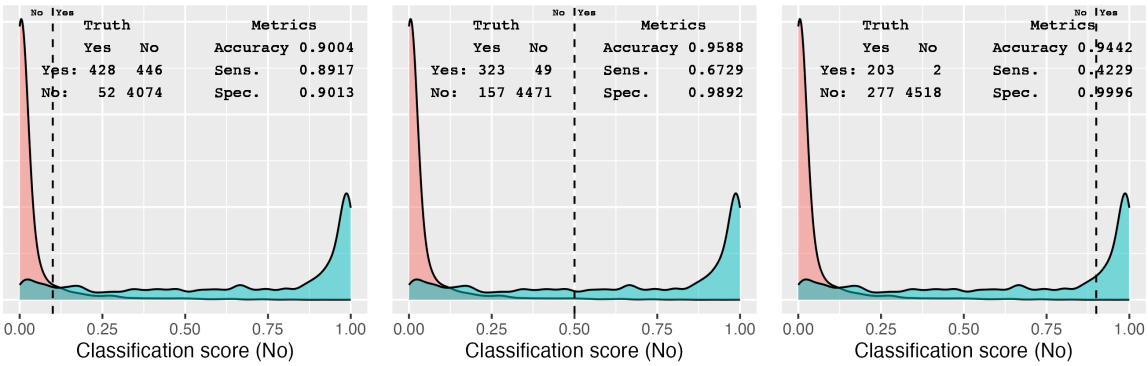


Figure 11.3: Relationship between threshold, confusion matrix, and various performance metrics

```
  ))
perf_1 <- performance %>% filter(.threshold == 0.1)
perf_5 <- performance %>% filter(.threshold == 0.5)
perf_9 <- performance %>% filter(.threshold == 0.9)
```

At the lowest threshold, 0.1, the model predicts most of the *Yes* cases as *Yes*, leading to the highest sensitivity of 0.8917. Increasing the threshold reduces the sensitivity, as more and more of the *Yes* cases are predicted as *No*. At the highest threshold, 0.9, the sensitivity is 0.4229. The specificity behaves in the opposite way. At the lowest threshold, the specificity is 0.9013. Increasing the threshold, more and more of the incorrectly classified *No* cases are now correctly predicted. At the highest threshold, the specificity is 0.9996.

Selecting the best threshold is a trade-off between sensitivity and specificity. The `probably` package provides a function to explore the relationship between thresholds and performance metrics. The function `probably::threshold_perf` calculates the performance metrics for a range of thresholds. The function returns a tibble with the threshold, the performance metric, and the estimate. We can use this tibble to plot the relationship between thresholds and performance metrics and determine a threshold based on a criteria of our choice. Figure 11.4 shows the relationship between thresholds and accuracy, sensitivity, and specificity.

```
performance_1 <- probably::threshold_perf(data, Personal.Loan,
  .pred_Yes, thresholds = seq(0.05, 0.95, 0.01),
  event_level = "first",
  metrics = metric_set(j_index, specificity, sensitivity))
performance_2 <- probably::threshold_perf(data, Personal.Loan,
  .pred_Yes, thresholds = seq(0.05, 0.95, 0.01),
  event_level = "first",
```

```
  metrics = metric_set(accuracy, kap, bal_accuracy, f_meas))
max_j_index <- performance_1 %>%
  filter(.metric == "j_index") %>%
  filter(.estimate == max(.estimate))
max_performance <- performance_2 %>%
  arrange(desc(.threshold)) %>%
  group_by(.metric) %>%
  filter(.estimate == max(.estimate)) %>%
  filter(row_number() == 1)

g1 <- ggplot(performance_1,
  aes(x = .threshold, y = .estimate, color = .metric,
    linetype = .metric)) +
  geom_line() +
  geom_vline(data = max_j_index,
    aes(xintercept = .threshold, color = .metric)) +
  scale_x_continuous(breaks = seq(0, 1, 0.1)) +
  xlab("Threshold") + ylab("Metric value") +
  theme(legend.position = "inside",
    legend.position.inside = c(0.85, 0.75))
g2 <- ggplot(performance_2,
  aes(x = .threshold, y = .estimate, color = .metric,
    linetype = .metric)) +
  geom_line() +
  geom_vline(data = max_performance,
    aes(xintercept = .threshold, color = .metric)) +
  scale_x_continuous(breaks =  seq(0, 1, 0.1)) +
  xlab("Threshold") + ylab("Metric value") +
  theme(legend.position = "inside",
    legend.position.inside = c(0.85, 0.75))
g1 + g2
```

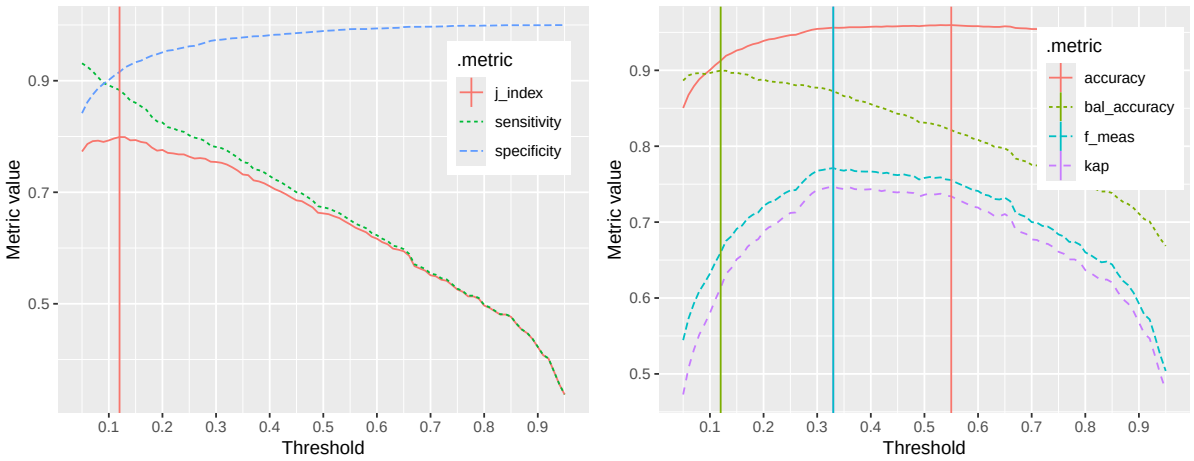


Figure 11.4: Relationship between threshold and sensitivity, specificity, and j-index

The `probably::threshold_perf` function takes a tibble and the names of the columns that contain the truth and the predicted probabilities. The `thresholds` argument specifies the range of thresholds to be explored. The `metrics` argument specifies the metrics to be calculated for each threshold. If you event of interest is *not* the first level, you can specify this using the `event_level` argument. Evaluating the performance metrics is very fast, so you can explore a large number of thresholds. Here, we explored 91 thresholds between 0.05 and 0.95.

The graphs clearly show that depending on the selected metric, the optimal threshold is different. The first graph shows the relationship between thresholds and the J-index, sensitivity, and specificity. The second graph shows the relationship between thresholds and accuracy, kappa, and balanced accuracy. The vertical lines indicate the optimal threshold for each metric. The optimal threshold for the J-index is 0.12. The optimal threshold for accuracy, kappa, and balanced accuracy is 0.55, 0.33, 0.33, 0.12.

> Useful to know
>
> This section has shown you how to calculate classification metrics as a function of the threshold. In any project, you will need to decide which of all possible metrics is the most appropriate for your problem. Sometimes you want to be more risk averse and prefer a higher sensitivity or fewer false positives. Sometimes you can be more risk taking and prefer a higher specificity or fewer false negatives.

11.2 Class probability metrics

In the previous Chapter 10, we encountered AUC, the area under the ROC curve. This is an example of a *class probability metric*. The ROC curve shows the relationship between

sensitivity and specificity for all possible thresholds.

A ROC curve can be constructed by calculating sensitivity and specificity at different thresholds and then plotting the relationship between sensitivity and specificity. Figure 11.5 demonstrates the construction of the ROC curve.

```
performance <- probably::threshold_perf(data, Personal.Loan,
  .pred_Yes, seq(0.00, 1.0, 0.1), event_level = "first")
metrics <- pivot_wider(performance, id_cols = .threshold,
  names_from = .metric, values_from = .estimate)
roc_curve(data, Personal.Loan, .pred_Yes, event_level = "first") %>%
  autoplot() +
  geom_point(data = metrics,
    aes(x = 1 - specificity, y = sensitivity),
    color = "red") +
  geom_text(data = metrics,
    aes(x = 1 - specificity, y = sensitivity, label = .threshold),
    nudge_x = 0.05, check_overlap = TRUE)
```

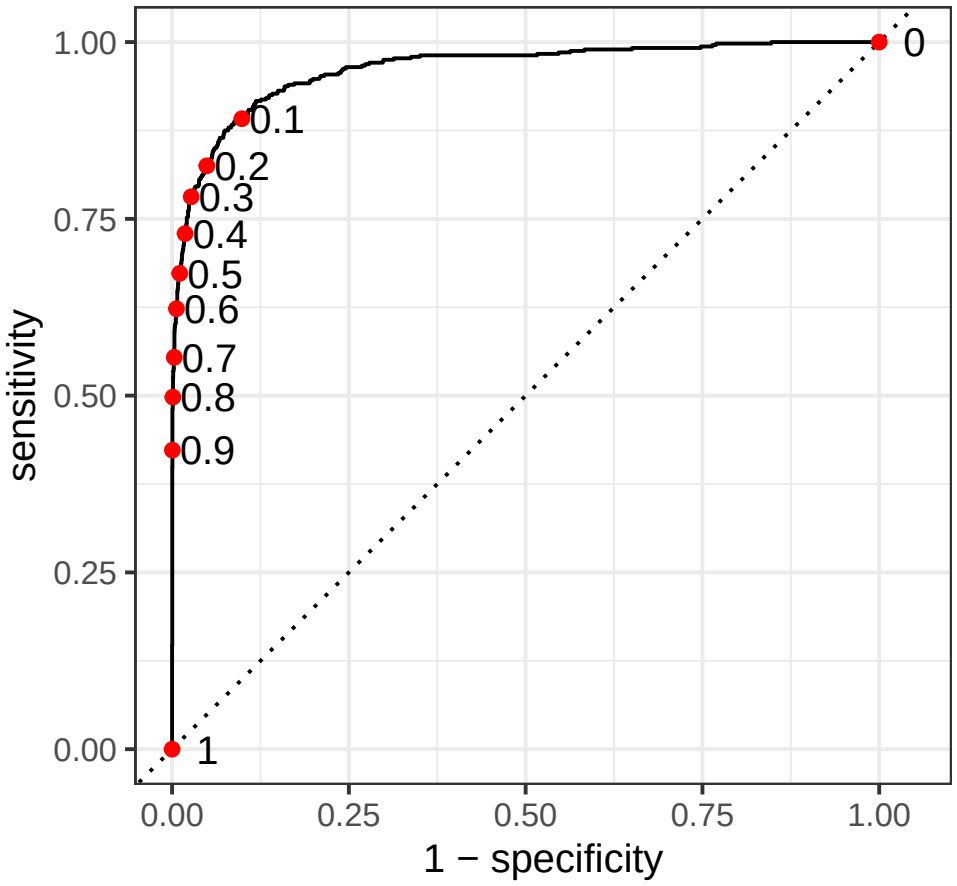


Figure 11.5: Construction of ROC curve

We first use `threshold_perf` to calculate the sensitivity and specificity for a range of thresholds. We then use `pivot_wider` to convert the tibble into a wide format. In addition, we use the `roc_curve` function from `yardstick` to calculate the ROC curve and plot it first. The graph then overlays the results from the `threshold_perf` calculation as red points.

> Useful to know
>
> In reality, calculating the ROC curves is a bit more complicated. In particular, care must be taken on how to resolve ties. The `roc_curve` function from `yardstick` uses the trapezoidal rule to calculate the area under the curve. For more information on how and why ties are important see (Muschelli 2020).

The AUC of the ROC curve is a useful estimate of how well classes are separated by the model. Figure 11.6 demonstrates the relationship between class separation, ROC curves, and AUC. The top graphs show the distribution of the predicted probabilities for the two classes. The

bottom graphs show the corresponding ROC curves. Increasing the class separation, leads to a ROC curve that is closer to the ideal ROC curve and as a consequence, the AUC increases.

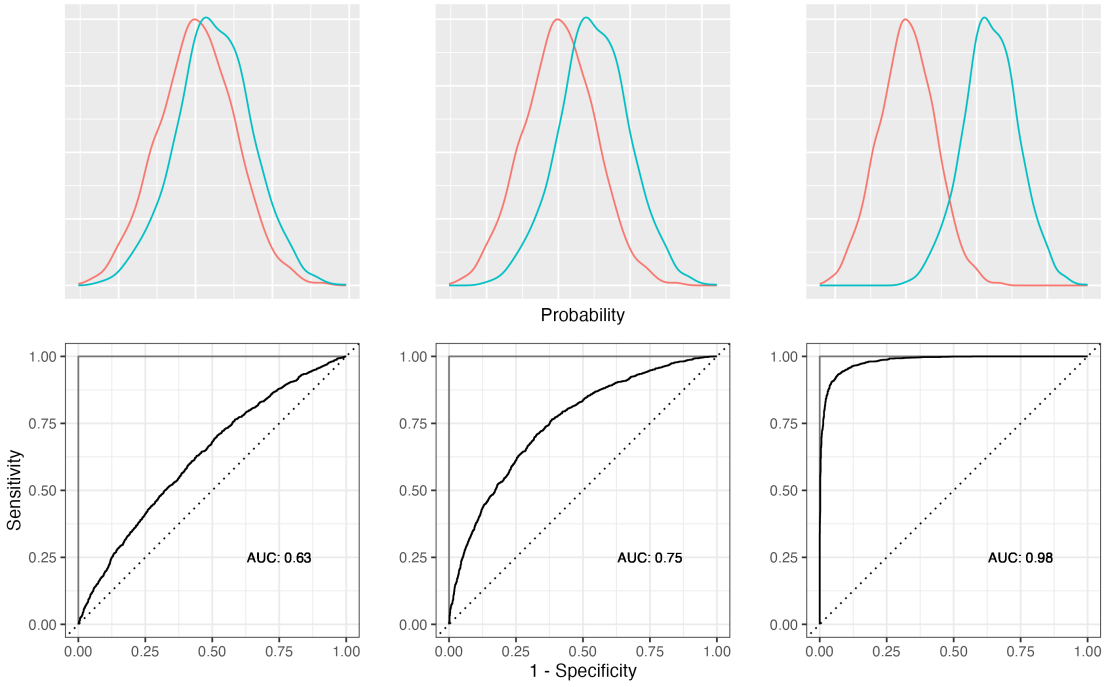


Figure 11.6: Relationship between class separation, ROC curves and AUC

Figure 11.7 gives a guideline on how to interpret the AUC.

Let's calculate the AUC for our example.

```
roc_auc(data, Personal.Loan, .pred_Yes, event_level = "first")
```

```
# A tibble: 1 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 roc_auc binary         0.962
```

The `roc_auc` function requires the truth and the predicted probabilities. The `event_level` argument specifies the event of interest. In our case, the AUC is 0.962. We have an excellent model.

Other class probability metrics are:

- `pr_auc`: Area under the precision recall curve

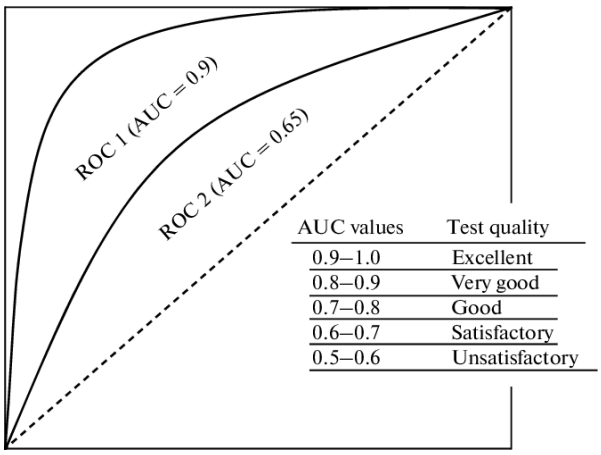


Figure 11.7: Interpretation of AUC (Trifonova, Lokhov, and Archakov 2014)

- `average_precision`: Area under the precision recall curve (variation of `pr_auc`)
- `gain_capture`: Gain capture
- `mn_log_loss`: Mean log loss for multinomial data
- `classification_cost`: Costs function for poor classification
- `brier_class`: Brier score for classification models

```
prob_metrics <- metric_set(
  roc_auc, pr_auc, average_precision, gain_capture,
  mn_log_loss, classification_cost, brier_class
)
prob_metrics(data, Personal.Loan, .pred_Yes)
```

```
# A tibble: 7 x 3
  .metric             .estimator .estimate
  <chr>               <chr>          <dbl>
1 roc_auc             binary         0.962
2 pr_auc              binary         0.851
3 average_precision   binary         0.851
4 gain_capture        binary         0.924
5 mn_log_loss         binary         0.117
6 classification_cost binary         0.0652
7 brier_class         binary         0.0317
```

11.3 Additional curves

In addition to the ROC curves, `yardstick` supports additional curves. Figure 11.8 shows all curves supported by `yardstick`.

```
g1 <- roc_curve(data, Personal.Loan, .pred_Yes,
  event_level = "first") %>%
  autoplot() + labs(title = "ROC curve")
g2 <- gain_curve(data, Personal.Loan, .pred_Yes,
  event_level = "first") %>%
  autoplot() + labs(title = "Gains curve")
g3 <- pr_curve(data, Personal.Loan, .pred_Yes,
  event_level = "first") %>%
  autoplot() + labs(title = "Precision/recall")
g4 <- lift_curve(data, Personal.Loan, .pred_Yes,
  event_level = "first") %>%
  autoplot() + labs(title = "Lift curve")

(g1 + g2) / (g3 + g4)
```

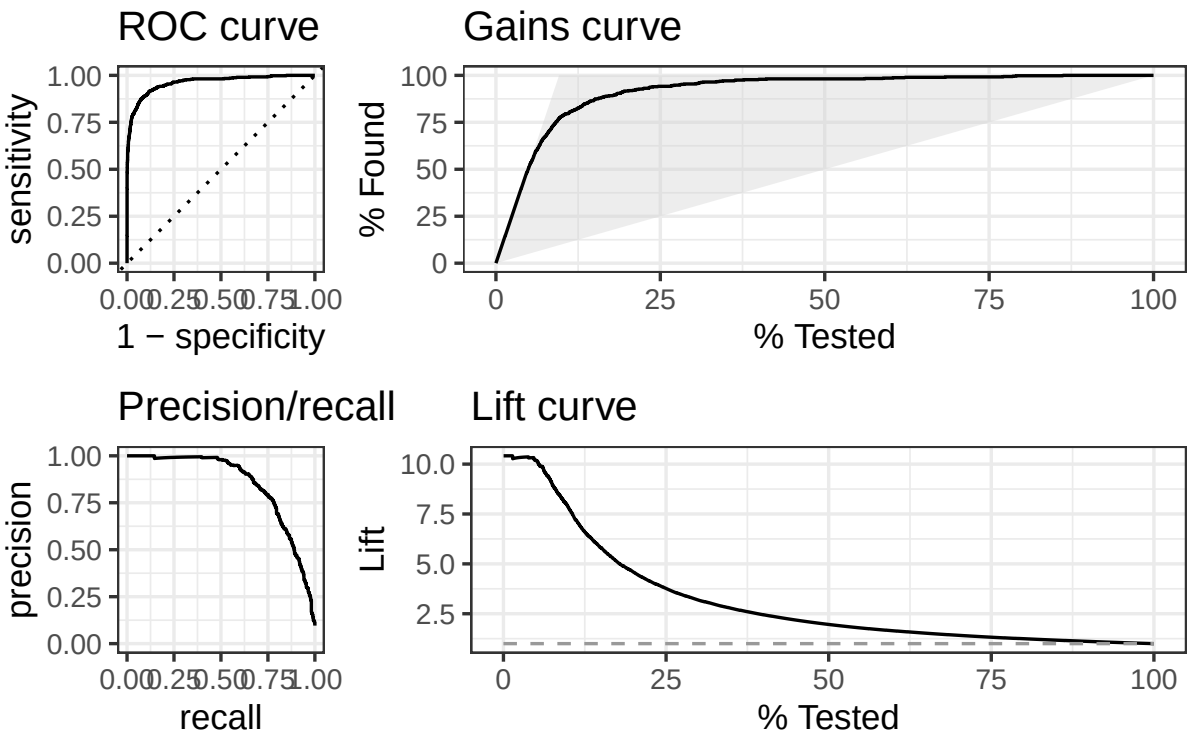


Figure 11.8: All curves supported by `yardstick`

The *precision/recall curve* plots precision against recall.

The *gains curve* is similar to the ROC curve, but instead of plotting sensitivity against specificity, it plots the cumulative number of true positives against the cumulative number of false positives. A *gains curve* focuses on what happens if you use the model to select a subset of the data based on the predicted probability. In our example, approaching 10% of the customers based on the predicted *Yes* score, will give us about 75% of the customers that would get a loan. A variation of this curve type incorporates cost. Figure 11.9 shows an example. The benefit of a correct classification is offset by the cost of missclassifications. Looking a the curve from left to right, we see that initially, the benefit of correct classifications outweighs the cost of missclassifications. However, at some point, the cost of missclassifications outweighs the benefit of correct classifications and ultimately, the cost leads to a negative outcome. The optimal point is where the curve is the highest.

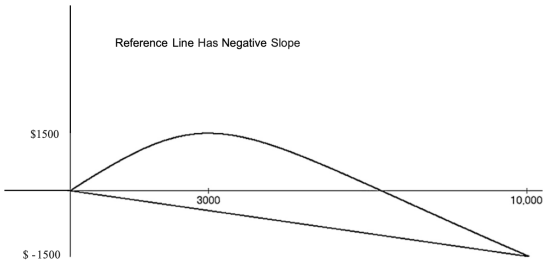


Figure 11.9: Cumulative gains curve incorporating costs (Shmueli et al. 2023)

The *lift curve* is another way of looking at selecting a subset based on the predicted score/probability. The curve tells you how much better (or worse) the model performs compared to random. In our example, we see that the lift for first 10% of the customers is between 7 and 10. This means that the model is 7 to 10 times better than random.

> **Todo**
>
> Look through the manual for `yardstick` to get an overview of all available classification metrics.

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
knitr::include_graphics("images/model_workflow_postprocessing.png")
library(tidyverse)
```

```
library(tidymodels)
library(yardstick)
library(probably)  # for exploring thresholds
library(patchwork) # for combining plots
file <-
  "https://gedeck.github.io/DS-6030/datasets/UniversalBank.csv.gz"
data <- read_csv(file)
data <- data %>%
  select(-c(ID, `ZIP Code`)) %>%
  rename(
    Personal.Loan = `Personal Loan`,
    Securities.Account = `Securities Account`,
    CD.Account = `CD Account`
  ) %>%
  mutate(
    Personal.Loan = factor(Personal.Loan, labels = c("Yes", "No"),
      levels = c(1, 0)),
    Education = factor(Education,
      labels =  c("Undergrad", "Graduate", "Advanced")),
  )
formula <- Personal.Loan ~ Income + Family + CCAvg + Education +
  Mortgage + Securities.Account + CD.Account + Online + CreditCard
model <- logistic_reg() %>%
  set_engine("glm") %>%
  fit(formula, data = data)
data <- augment(model, new_data = data)
cm <- data %>%
  conf_mat(truth = Personal.Loan, estimate = .pred_class)
cm
g1 <- autoplot(cm, type = "mosaic")
g2 <- autoplot(cm, type = "heatmap")
g1 + g2
(cm$table[1, 1] + cm$table[2, 2]) / sum(cm$table)
yardstick::accuracy(data, truth = Personal.Loan,
  estimate = .pred_class)
yardstick::metrics(data, Personal.Loan, .pred_class)
my_metrics <- metric_set(sens, spec, j_index)
my_metrics(data, truth = Personal.Loan, estimate = .pred_class)
my_metrics(data, truth = Personal.Loan, estimate = .pred_class)
conf_mat(data, truth = Personal.Loan, estimate = .pred_class)
my_metrics(data, truth = Personal.Loan, estimate = .pred_class,
  event_level = "second")
```

```
knitr::include_graphics("images/threshold-accuracy.png")
performance <- probably::threshold_perf(data, Personal.Loan,
  .pred_Yes, c(0.1, 0.5, 0.9), event_level = "first",
  metrics = yardstick::metric_set(
    yardstick::accuracy,
    yardstick::specificity,
    yardstick::sensitivity,
  ))
perf_1 <- performance %>% filter(.threshold == 0.1)
perf_5 <- performance %>% filter(.threshold == 0.5)
perf_9 <- performance %>% filter(.threshold == 0.9)
performance_1 <- probably::threshold_perf(data, Personal.Loan,
  .pred_Yes, thresholds = seq(0.05, 0.95, 0.01),
  event_level = "first",
  metrics = metric_set(j_index, specificity, sensitivity))
performance_2 <- probably::threshold_perf(data, Personal.Loan,
  .pred_Yes, thresholds = seq(0.05, 0.95, 0.01),
  event_level = "first",
  metrics = metric_set(accuracy, kap, bal_accuracy, f_meas))
max_j_index <- performance_1 %>%
  filter(.metric == "j_index") %>%
  filter(.estimate == max(.estimate))
max_performance <- performance_2 %>%
  arrange(desc(.threshold)) %>%
  group_by(.metric) %>%
  filter(.estimate == max(.estimate)) %>%
  filter(row_number() == 1)

g1 <- ggplot(performance_1,
  aes(x = .threshold, y = .estimate, color = .metric,
    linetype = .metric)) +
  geom_line() +
  geom_vline(data = max_j_index,
    aes(xintercept = .threshold, color = .metric)) +
  scale_x_continuous(breaks = seq(0, 1, 0.1)) +
  xlab("Threshold") + ylab("Metric value") +
  theme(legend.position = "inside",
    legend.position.inside = c(0.85, 0.75))
g2 <- ggplot(performance_2,
  aes(x = .threshold, y = .estimate, color = .metric,
    linetype = .metric)) +
  geom_line() +
```

```
  geom_vline(data = max_performance,
    aes(xintercept = .threshold, color = .metric)) +
  scale_x_continuous(breaks =  seq(0, 1, 0.1)) +
  xlab("Threshold") + ylab("Metric value") +
  theme(legend.position = "inside",
    legend.position.inside = c(0.85, 0.75))
g1 + g2
performance <- probably::threshold_perf(data, Personal.Loan,
  .pred_Yes, seq(0.00, 1.0, 0.1), event_level = "first")
metrics <- pivot_wider(performance, id_cols = .threshold,
  names_from = .metric, values_from = .estimate)
roc_curve(data, Personal.Loan, .pred_Yes, event_level = "first") %>%
  autoplot() +
  geom_point(data = metrics,
    aes(x = 1 - specificity, y = sensitivity),
    color = "red") +
  geom_text(data = metrics,
    aes(x = 1 - specificity, y = sensitivity, label = .threshold),
    nudge_x = 0.05, check_overlap = TRUE)
knitr::include_graphics("images/roc-auc-class-separation.png")
knitr::include_graphics("images/AUC-ROC.png")
roc_auc(data, Personal.Loan, .pred_Yes, event_level = "first")
prob_metrics <- metric_set(
  roc_auc, pr_auc, average_precision, gain_capture,
  mn_log_loss, classification_cost, brier_class
)
prob_metrics(data, Personal.Loan, .pred_Yes)
g1 <- roc_curve(data, Personal.Loan, .pred_Yes,
  event_level = "first") %>%
  autoplot() + labs(title = "ROC curve")
g2 <- gain_curve(data, Personal.Loan, .pred_Yes,
  event_level = "first") %>%
  autoplot() + labs(title = "Gains curve")
g3 <- pr_curve(data, Personal.Loan, .pred_Yes,
  event_level = "first") %>%
  autoplot() + labs(title = "Precision/recall")
g4 <- lift_curve(data, Personal.Loan, .pred_Yes,
  event_level = "first") %>%
  autoplot() + labs(title = "Lift curve")

(g1 + g2) / (g3 + g4)
knitr::include_graphics("images/c5f012.png")
```

Part V

Validating and tuning models

12 Sampling from a dataset

Sampling from a dataset is at the core of all validation and tuning methods. In most case, the sampling is done using a random process. There are two ways of sampling from a dataset:

- Sampling **without** replacement: each observation can be sampled only once. We use this approach for creating a holdout set and for cross-validation.
- Sampling **with** replacement: the same observation can be sampled more than once. This approach to sampling is at the core of the bootstrap method.

It can also be important to consider the structure of the dataset when sampling. With structure, we mean the distribution of the data with respect to the predictors or the data. This type of sampling is called **stratified** sampling. With stratified sampling, you can make sure that the distribution of the data is preserved in the sample. For example, if you have a binary outcome, you can make sure that the sample contains the same proportion of positive and negative outcomes as the original dataset. For continuous outcomes, it would mean that the sample has the same distribution of the outcome as the original dataset.

You can see the effect of stratified sampling for a continuous variable in Figure 12.1. The two panels shows the distribution of the continuous variable for a random sample (left) and a stratified sample (right). The original distribution is shown in red and the distribution of the samples in grey. We can clearly see that stratified sampling preserves the distribution of the data better.

> Useful to know
>
> Sampling, like all calculations that depend on randomness, uses a random number generator which means that repeated execution of the same code will lead to different results. You can make your calculations reproducible by setting a seed using the `set.seed()` function. The seed can be any number. The same seed will always produce the same random numbers. In general, it is good practice to set a seed to make sure that the results do not change between runs. You will see that we use the `set.seed()` function in the examples below.

Tidymodels provides a wide variety of sampling methods in the `rsample` package. The sampling methods are used to create training, validation, and holdout sets for model building and evaluation. Figure 12.2 shows where sampling fits into the model building workflow.

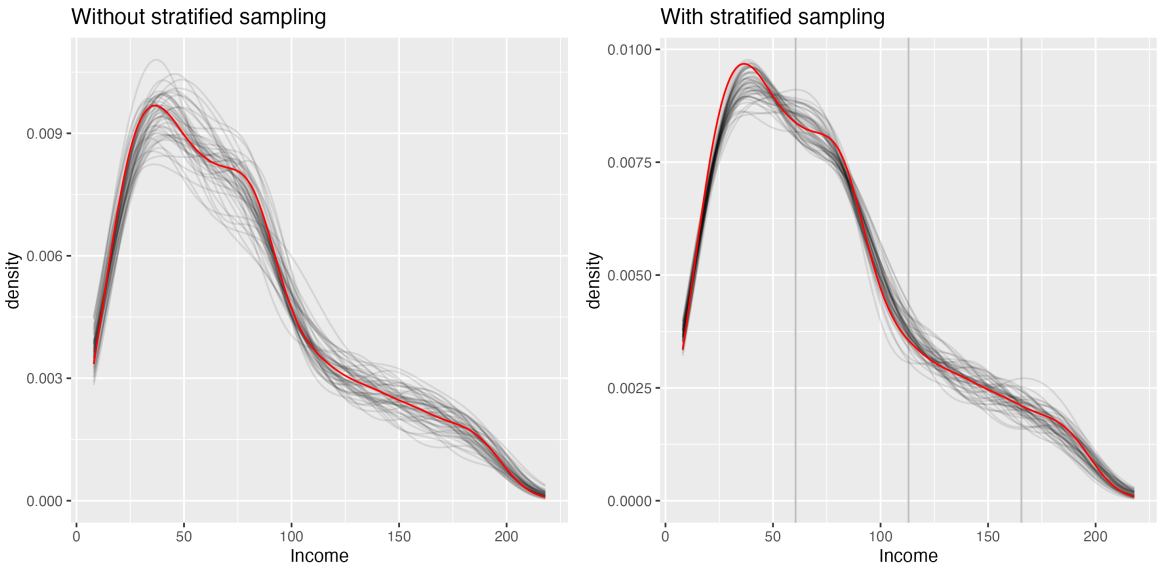


Figure 12.1: Effect of using stratified sampling on a continuous variable

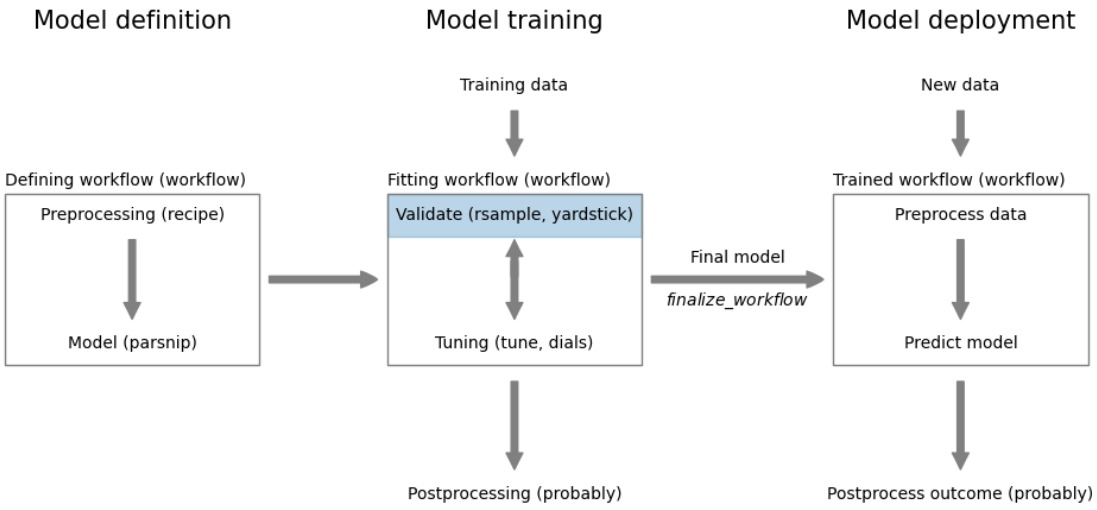


Figure 12.2: Sampling for model validation using `rsample`

12.1 Sampling in statistical modeling

In statistical modeling, we use sampling to create various subsets of the data. Common scenarios are:

- Split the data randomly into training and holdout set; use the training set to fit the model and the holdout set to evaluate the model. This approach can be taken if your model doesn't equire tuning. Using the holdout set to evaluate the model is a way to get an unbiased estimate of the model performance. It is good practice to repeat this process several times to get a more stable estimate of the model performance. This is called **repeated holdout** (see Figure 12.3)

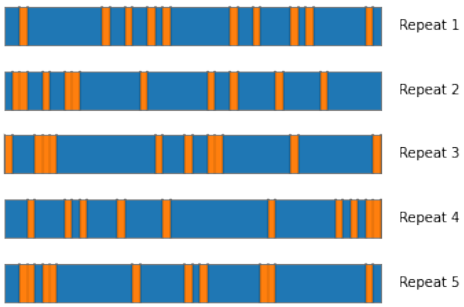


Figure 12.3: Repeated holdout: splitting the dataset randomly into 80% training (blue) and 20% holdout (orange) sets; process is repeated five times

- Split the data randomly into training, validation, and holdout set; use the training set to fit various models, select a specific model using the performance on the validation set, and use the holdout set to evaluate the final model. This approach can be taken if you have sufficient data and training the model is costly. See Figure 12.4 for an illustration of this approach.

- The previous scenario relies on a single training/validation split for comparing the performance of different models. A more robust approach is to use cross-validation. In cross-validation, the data is split systematically into k folds. Each fold is used as a validation set and the model is trained on the remaining $k-1$ folds. In total, we train k models. Each of the folds is used as a validation set once. The performance of the model is then averaged over the k folds. You repeat this k-fold cross validation approach for each of the models you want to compare and pick the best model based on the estimated performance. Finally, holdout set is used to make a decision on deploying the model or not. See Figure 12.5 for an illustration of this approach.

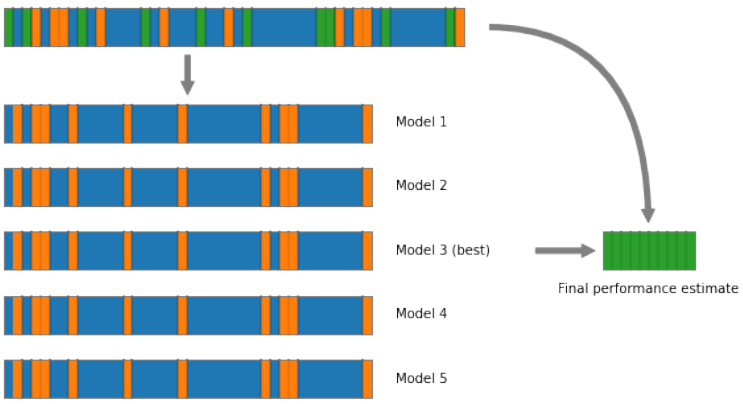


Figure 12.4: Training/validation/holdout split: splitting the dataset randomly into 60% training, 20% validation, and 20% holdout sets

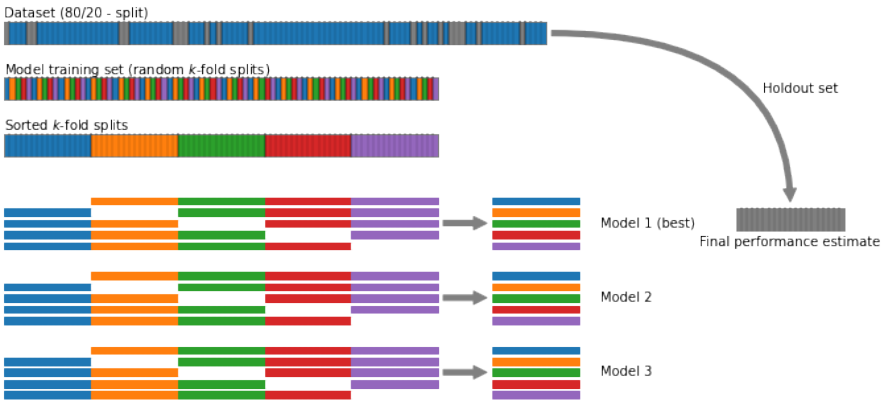


Figure 12.5: Cross-validation: splitting the dataset randomly into 5 folds; each fold is used as a holdout set once and the model is trained on the remaining folds

- A similar approach to cross-validation is using bootstrap. Here the data are split randomly *with replacement* into a training and a validation set. The training set is used to train a model and the validation set to estimate the performance. Because of sampling *with replacement*, the training set will contain duplicates and the validation set will have different size for each bootstrap sample. However, because we repeat this bootstrap splitting several times, each data point will eventually be used in training and in validation. It's up to you how many bootstrap samples you create. Once we evaluated each bootstrap sample, the performance estimates are combined and used to compare the various models and pick the best model based on the estimated performance. It is best to use the same splittings for each of the models. See Figure 12.6 for an illustration of this approach.

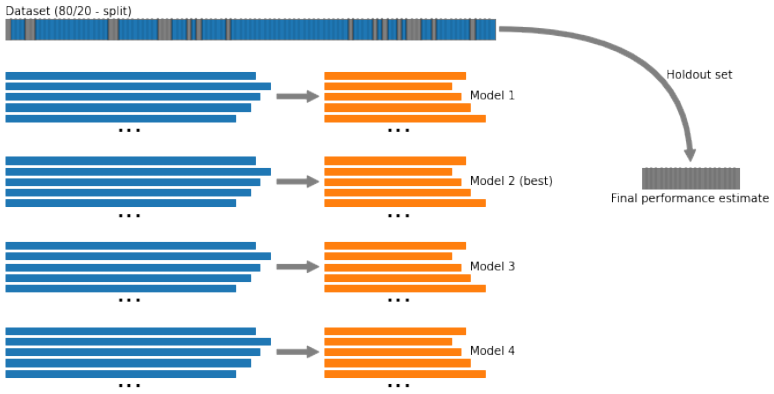


Figure 12.6: Bootstrap: splitting the dataset randomly into training and validation set *with replacement*

There are more variations to these approaches. For example, you can use nested cross-validation to tune the hyperparameters of a model. In this case, you use cross-validation to compare different models and then use cross-validation again to tune the hyperparameters of the best model. There are also specialized approaches for time series data. We will not cover these approaches in this class.

In this section, we will see how to implement these approaches using the `rsample` package which is part of *tidymodels*.

```
library(tidymodels)
# or
library(rsample)
```

Useful to know

If you look at the literature, you will find that the terminology is not always consistent. For example, some authors use the term *validation set* to refer to the holdout set. In this class, we will use the term *validation set* to refer to the set that is used to compare different models and *holdout set* to refer to the set that is used to evaluate the final model. The `rsample` packages uses its own terminology.

12.2 Creating an initial split of the data into training and holdout set

The first step in a model building step is to split the data into a training and a holdout set. The objective of the holdout set is to get an unbiased estimate of the model performance for the selected *best* model. The holdout set is used only once at the end of the modeling process and **not** for any intermediate steps. `rsample` refers to the holdout set as the *testing* set.

We can use the function `rsample::initial_split` to create a single split of the data. Here is an example:

```
set.seed(1353)
car_split <- initial_split(mtcars)
car_split
```

```
<Training/Testing/Total>
<24/8/32>
```

The function `initial_split` by itself doesn't create different subsets of the data, it only creates a blueprint for how the data should be split. Here, we see that the 32 data points are split into 24 data points for *training* and 8 for testing. By default, the function splits the data into 75% for training and 25% for testing.

To get the individual subsets, we need to use the functions `training` and `testing`.

```
train_data <- training(car_split)
test_data <- testing(car_split)
```

The function `rsample::initial_split` is used to create a single split of the data. It takes several arguments. The most commonly used ones are:

- `prop`: the proportion of the data that should be used for training. The default is 0.75.

- `strata`: a variable that is used to stratify the data. The default is `NULL`. It is good practice to use
- `breaks`: this argument is used for creating stratified samples from a continuous variable. It specifies the number of breaks that should be used to create the strata. The default is 4.

12.3 Creating an initial split of the data into training, validation, and holdout set

If for some reason, you cannot afford to use one of the iterative approaches (cross-validation or bootstrap), you can use a single split of the data into training, validation, and holdout set. The training set is used to fit the model, the validation set to compare different models, and the holdout set to evaluate the final model. The function `rsample::initial_validation_split` allows you to create a random split of your dataset.

```
set.seed(9872)

car_split <- initial_validation_split(mtcars)
car_split
```

```
<Training/Validation/Testing/Total>
<19/6/7/32>
```

We see that the 32 data points are split into 19 data points for *training*, 6 for validation, and 7 for testing/holdout. By default, the function splits the data into 60% for training, 20% for validation, and 20% for testing/holdout.

To get the individual subsets, we need to use the functions `training`, `validation`, and `testing`.

```
train_data <- training(car_split)
validation_data <- validation(car_split)
holdout_data <- testing(car_split)
```

> **Further information**
>
> - https://rsample.tidymodels.org/ is the documentation for the `rsample` package.

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
knitr::include_graphics("images/stratified-continuous.png")
knitr::include_graphics("images/model_workflow_validate.png")
knitr::include_graphics("images/validation_repeated_holdout.png")
knitr::include_graphics("images/validation_train_validation_holdout_split.png")
knitr::include_graphics("images/validation_cross_validation.png")
knitr::include_graphics("images/validation_bootstrap.png")
library(tidymodels)
# or
library(rsample)
set.seed(1353)
car_split <- initial_split(mtcars)
car_split
train_data <- training(car_split)
test_data <- testing(car_split)
set.seed(9872)

car_split <- initial_validation_split(mtcars)
car_split
train_data <- training(car_split)
validation_data <- validation(car_split)
holdout_data <- testing(car_split)
```

13 Validating models

In Chapter 12, we covered various ways of splitting the data into subsets. In this chapter, we will use these subsets to assess model performance for model validation using:

- a holdout set,
- cross-validation, and
- bootstrapping.

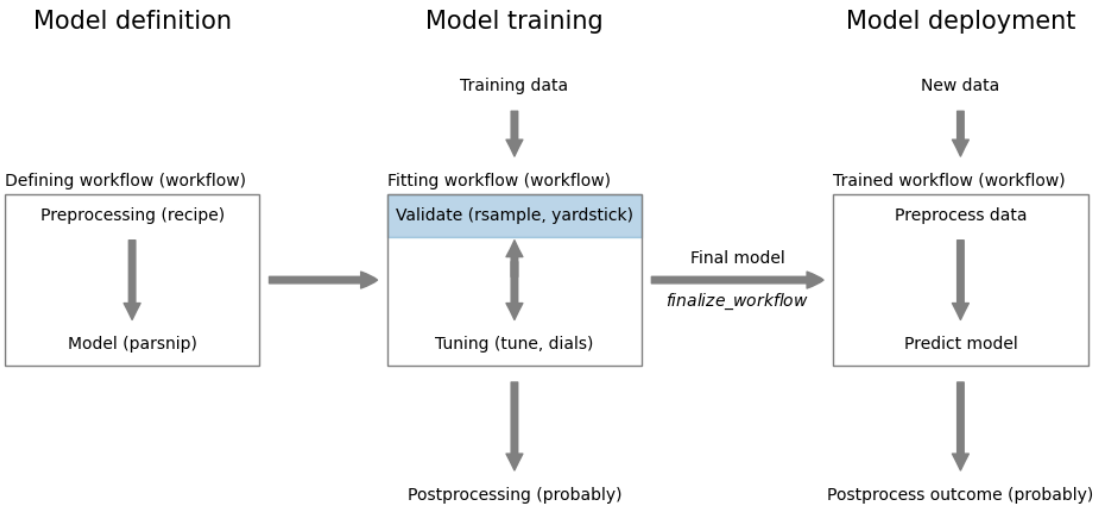


Figure 13.1: Model validation

Load required packages:

```
library(tidyverse)
library(tidymodels)
library(patchwork)
```

and setup the parallel backend for faster processing (see Section D.1 for details):

```
library(future)
plan(multisession, workers = parallel::detectCores(logical = FALSE))
```

13.1 Model validation using holdout set

In the following we demonstrate how to use a holdout set to assess the performance of a regression model to predict mileage for cars.

```
# Load an preprocess the data
data <- datasets::mtcars %>%
  as_tibble(rownames = "car") %>%
  mutate(
    vs = factor(vs, labels = c("V-shaped", "straight")),
    am = factor(am, labels = c("automatic", "manual")),
  )

# Split the data into training and test/holdout sets
set.seed(1353)
car_split <- initial_split(mtcars)
train_data <- training(car_split)
holdout_data <- testing(car_split)

# Train a model using the training set
formula <- mpg ~ cyl + disp + hp + drat + wt + qsec + vs + am +
  gear + carb
model <- linear_reg() %>%
  set_engine("lm") %>%
  fit(formula, data = train_data)
```

We now have a trained model and can use it to assess model performance for the training and holdout set using the default regression metrics from `yardstick`.

```
perf_train <- augment(model, new_data = train_data) %>%
  metrics(truth = mpg, estimate = .pred) %>%
  mutate(data = "Training")
perf_holdout <- augment(model, new_data = holdout_data) %>%
  metrics(truth = mpg, estimate = .pred) %>%
  mutate(data = "Holdout")
```

We can combine the performance metrics for the training and holdout set into a single table for comparison.

```r
# combine the two results
bind_rows(perf_train, perf_holdout) %>%
  # select the columns of interest
  select(data, .estimate, .metric) %>%
  # convert to wide format
  pivot_wider(names_from = .metric, values_from = .estimate) %>%
  knitr::kable(digits = 2) %>%
  kableExtra::kable_styling(full_width = FALSE)
```

data	rmse	rsq	mae
Training	2.06	0.90	1.62
Holdout	3.10	0.74	2.78

The performance metrics on the training set indicate better performance compared to the holdout set. This is expected since the model was trained on the training set.

13.2 Model validation using cross-validation

We will now use cross-validation to assess model performance. This time, we train a logistic regression model for the Universal Bank dataset using the entire dataset and use cross-validation to assess model performance. First download the dataset and preprocess it.

```r
# Load and preprocess the data
file <- "https://gedeck.github.io/DS-6030/datasets/UniversalBank.csv.gz"
data <- read_csv(file)
data <- data %>%
  select(-c(ID, `ZIP Code`)) %>%
  rename(
    Personal.Loan = `Personal Loan`,
    Securities.Account = `Securities Account`,
    CD.Account = `CD Account`
  ) %>%
  mutate(
    Personal.Loan = factor(Personal.Loan, labels = c("Yes", "No"),
      levels = c(1, 0)),
    Education = factor(Education,
      labels = c("Undergrad", "Graduate", "Advanced")),
  )
```

Now we setup and execute the cross-validation.

```
# Use 10-fold cross-validation to assess model performance
set.seed(1353)
folds <- vfold_cv(data, strata = Personal.Loan)

# define the model
formula <- Personal.Loan ~ Age + Experience + Income + Family + CCAvg +
  Education + Mortgage + Securities.Account + CD.Account +
  Online + CreditCard
logreg_model <- logistic_reg() %>%
  set_engine("glm")

# define and execute the cross-validation workflow
logreg_wf <- workflow() %>%
  add_model(logreg_model) %>%
  add_formula(formula)

logreg_fit_cv <- logreg_wf %>%
  fit_resamples(folds)
```

This is all we need to do. We first define our resampling approach using the function `vfold_cv` passing in the dataset and information about the column we want to use for stratified sampling, the outcome variable `Personal.Loan`. The default `v=10` is used to define a 10-fold cross validation. Next we setup our model and define the formulat to use for training. Finaly, we combine the model and formula into a workflow and use the `fit_resamples` function to execute the cross-validation. The results are stored in the `logreg_fit_cv` object. Let's have a look at it:

```
logreg_fit_cv
```

```
# Resampling results
# 10-fold cross-validation using stratification
# A tibble: 10 x 4
   splits             id     .metrics          .notes
   <list>             <chr>  <list>            <list>
 1 <split [4500/500]> Fold01 <tibble [3 x 4]> <tibble [0 x 3]>
 2 <split [4500/500]> Fold02 <tibble [3 x 4]> <tibble [0 x 3]>
 3 <split [4500/500]> Fold03 <tibble [3 x 4]> <tibble [0 x 3]>
 4 <split [4500/500]> Fold04 <tibble [3 x 4]> <tibble [0 x 3]>
 5 <split [4500/500]> Fold05 <tibble [3 x 4]> <tibble [0 x 3]>
 6 <split [4500/500]> Fold06 <tibble [3 x 4]> <tibble [0 x 3]>
```

```
 7 <split [4500/500]> Fold07 <tibble [3 x 4]> <tibble [0 x 3]>
 8 <split [4500/500]> Fold08 <tibble [3 x 4]> <tibble [0 x 3]>
 9 <split [4500/500]> Fold09 <tibble [3 x 4]> <tibble [0 x 3]>
10 <split [4500/500]> Fold10 <tibble [3 x 4]> <tibble [0 x 3]>
```

It's not very informative. `logreg_fit_cv` is a tibble where each row corresponds to models trained for each fold (column `id`).[1] Information about what was used at each iteration is in the `splits` column. The performance on the out-of-fold validation set is in the `.metrics` column.

We can now use the `collect_metrics` function to extract information about the performance metrics for each fold. By default it will summarize the information for each metric into a mean and standard error.

```
cv_metrics <- collect_metrics(logreg_fit_cv)
cv_metrics
```

```
# A tibble: 3 x 6
  .metric     .estimator   mean     n std_err .config
  <chr>       <chr>       <dbl> <int>   <dbl> <chr>
1 accuracy    binary     0.958     10 0.00184 Preprocessor1_Model1
2 brier_class binary     0.0322    10 0.00123 Preprocessor1_Model1
3 roc_auc     binary     0.961     10 0.00502 Preprocessor1_Model1
```

We can see that during cross-validation, the model performance is evaluated using accuracy and the AUC under the ROC curve. The metrics are combined into a mean and an associated standard deviation.

By default, individual predictions on the out-of-fold dataset for the performance metrics are not returned. If we want to keep these for further analysis, we need to add a control statement to the `fit_resamples` call.

```
logreg_fit_cv <- logreg_wf %>%
  fit_resamples(folds, control = control_resamples(save_pred = TRUE))
```

The `control_resamples` function returns the is used to pass additional arguments to the `fit_resamples` function. Here, we override the default behavior by setting `save_pred=TRUE` which instructs the function to preserve the out-of-fold predictions for each fold. The `collect_predictions` function returns a tibble with all predictions for each fold.

[1] This means, the fold was used as the validation set and the remaining folds were used for training.

```
cv_predictions <- collect_predictions(logreg_fit_cv)
cv_predictions
```

```
# A tibble: 5,000 x 7
   .pred_class .pred_Yes .pred_No id      .row Personal.Loan .config
   <fct>           <dbl>    <dbl> <chr>  <int> <fct>         <chr>
 1 No           0.0175      0.982 Fold01     7 No            Preprocessor1_Mode~
 2 No           0.0490      0.951 Fold01    25 No            Preprocessor1_Mode~
 3 No           0.0123      0.988 Fold01    28 No            Preprocessor1_Mode~
 4 Yes          0.785       0.215 Fold01    30 Yes           Preprocessor1_Mode~
 5 No           0.00122     0.999 Fold01    31 No            Preprocessor1_Mode~
 6 No           0.00517     0.995 Fold01    47 No            Preprocessor1_Mode~
 7 No           0.00142     0.999 Fold01    59 No            Preprocessor1_Mode~
 8 Yes          0.654       0.346 Fold01    60 No            Preprocessor1_Mode~
 9 No           0.0113      0.989 Fold01    74 No            Preprocessor1_Mode~
10 No           0.000650    0.999 Fold01    88 No            Preprocessor1_Mode~
# i 4,990 more rows
```

We can use this information to calculate individual ROC curves on the out-of-fold predictions (see Figure 13.2).

```
cv_predictions %>%
  group_by(id) %>%
  roc_curve(truth = Personal.Loan, .pred_Yes, event_level = "first") %>%
  autoplot() +
  geom_abline(lty = 2) +
  theme(legend.position = "none")
```

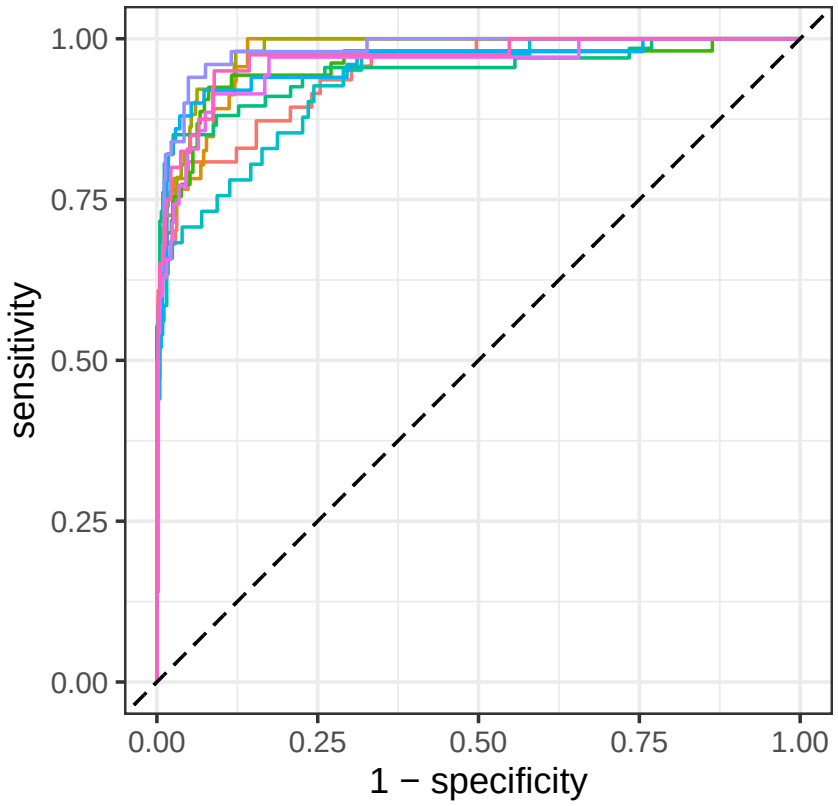


Figure 13.2: Individual ROC curves for cross-validation folds

Instead of showing individual ROC curves for each fold, we can also combine them into a single plot. Figure 13.3 compares the ROC curves for the cross-validation predictions and the predictions on the training set.

```
# Train a model on the full dataset
full_model <- logistic_reg() %>%
  set_engine("glm") %>%
  fit(formula, data = data)

cv_roc <- cv_predictions %>%
  roc_curve(truth = Personal.Loan, .pred_Yes, event_level = "first")

ontrain_roc <- augment(full_model, new_data = data) %>%
  roc_curve(Personal.Loan, .pred_Yes, event_level = "first")

ggplot() +
  geom_path(data = cv_roc,
```

```
    aes(x = 1 - specificity, y = sensitivity)) +
  geom_path(data = ontrain_roc,
    aes(x = 1 - specificity, y = sensitivity),
    color = "red") +
  geom_abline(lty = 2)
```

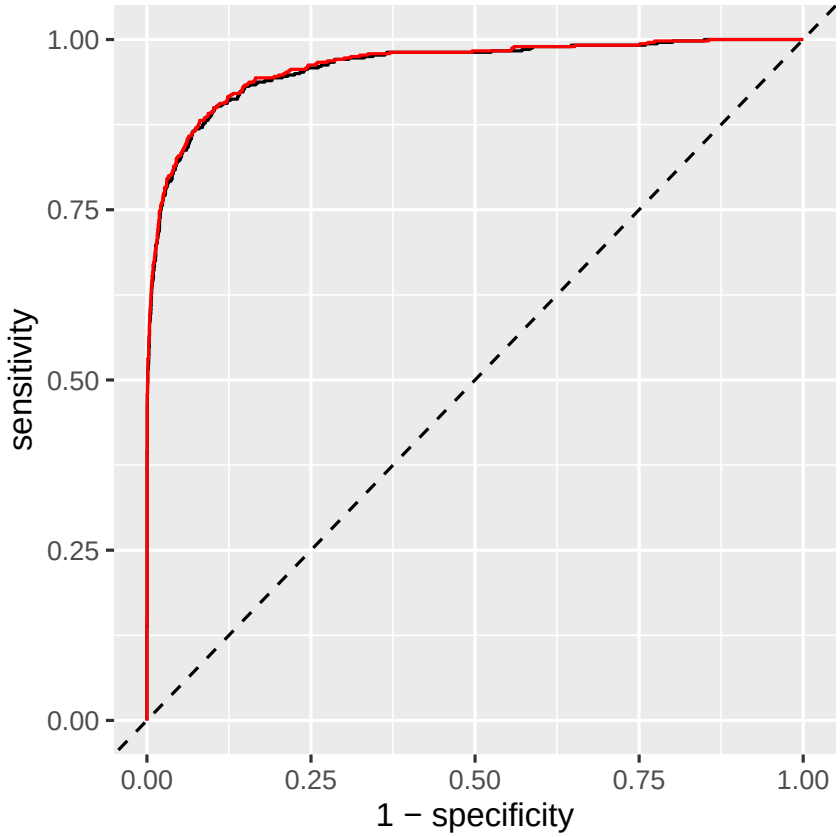


Figure 13.3: Comparison of ROC curves for cross-validation predictions (black) and on-training set predictions (red); the curves are hardly distinguishable

The ROC curves for the cross-validation predictions are very similar to the ROC curves for the predictions on the training set. This indicates that the model is not overfitting the training data.

13.3 Model validation using bootstrapping

The `bootstraps` function from the `rsample` package can be used to generate bootstrap samples.[2] We can use these to assess model performance using bootstrap resampling. This time, we train a nearest neighbor model (see Section A.5) for the Universal Bank dataset. Because the `kknn` package supports both classification and regression, we need to specify the type of model we want to train.

```
# Use bootstrap to assess model performance
set.seed(1353)
resamples <- rsample::bootstraps(data)

# define the model
formula <- Personal.Loan ~ Age + Experience + Income + Family +
  CCAvg + Education + Mortgage + Securities.Account +
  CD.Account + Online + CreditCard

nn_model <- nearest_neighbor(neighbors = 5) %>%
  set_mode("classification") %>%
  set_engine("kknn")

# define and execute the cross-validation workflow
nn_wf <- workflow() %>%
  add_model(nn_model) %>%
  add_formula(formula)

nn_fit_boot <- nn_wf %>%
  fit_resamples(resamples,
    control = control_resamples(save_pred = TRUE))
```

If you compare the code for bootstrap sampling with the code for cross-validation, you will notice that the only difference is the call to `bootstraps` instead of `vfold_cv` and the use of a different model. The rest of the code is identical.

Let's have a look at the results.

```
nn_fit_boot
```

```
# Resampling results
# Bootstrap sampling
```

[2] Be careful for typos here. There is also the `broom::bootstrap` function, which will give you a missing argument warning.

```
# A tibble: 25 x 5
   splits              id          .metrics          .notes   .predictions
   <list>              <chr>       <list>            <list>   <list>
 1 <split [5000/1846]> Bootstrap01 <tibble [3 x 4]> <tibble> <tibble>
 2 <split [5000/1809]> Bootstrap02 <tibble [3 x 4]> <tibble> <tibble>
 3 <split [5000/1874]> Bootstrap03 <tibble [3 x 4]> <tibble> <tibble>
 4 <split [5000/1835]> Bootstrap04 <tibble [3 x 4]> <tibble> <tibble>
 5 <split [5000/1845]> Bootstrap05 <tibble [3 x 4]> <tibble> <tibble>
 6 <split [5000/1856]> Bootstrap06 <tibble [3 x 4]> <tibble> <tibble>
 7 <split [5000/1839]> Bootstrap07 <tibble [3 x 4]> <tibble> <tibble>
 8 <split [5000/1841]> Bootstrap08 <tibble [3 x 4]> <tibble> <tibble>
 9 <split [5000/1859]> Bootstrap09 <tibble [3 x 4]> <tibble> <tibble>
10 <split [5000/1834]> Bootstrap10 <tibble [3 x 4]> <tibble> <tibble>
# i 15 more rows

There were issues with some computations:

  - Warning(s) x25: variable '..y' is absent, its contrast will be ignored

Run `show_notes(.Last.tune.result)` for more information.
```

Again, this is not very informative. We have 25 bootstrap samples and various columns that contain information about each sample. Because we used `save_pred=TRUE` in the `fit_resamples` call, we also have the `.predictions` column with the individual predictions.

We can use `collect_metrics` to extract the performance metrics for each fold and compare the result to the cross-validation results for the logistic regression model.[3]

```r
boot_metrics <- collect_metrics(nn_fit_boot)

bind_rows(
  cv_metrics %>% mutate(model = "Logistic regression"),
  boot_metrics %>% mutate(model = "Nearest neighbor")
) %>%
  select(model, mean, .metric) %>%
  pivot_wider(names_from = .metric, values_from = mean) %>%
  knitr::kable(digits = 3) %>%
  kableExtra::kable_styling(full_width = FALSE)
```

[3] **This is for demonstration only!** In practice, you would use the same validation approach for both models.

model	accuracy	brier_class	roc_auc
Logistic regression	0.958	0.032	0.961
Nearest neighbor	0.961	0.032	0.918

Base on accuracy, we would conclude that the nearest neighbor model is better than the logistic regression model. However, the AUC for the nearest neighbor model is significantly lower.

Let's see if this is reflected in the ROC curves of the two models. Figure 13.4 compares the ROC curves for the bootstrap predictions and the predictions on the training set.

```
# Train a model on the full dataset
full_model <- nn_wf %>% fit(data)

boot_roc <- collect_predictions(nn_fit_boot) %>%
  roc_curve(truth = Personal.Loan, .pred_Yes, event_level = "first")

ontrain_roc <- augment(full_model, new_data = data) %>%
  roc_curve(Personal.Loan, .pred_Yes, event_level = "first")

ggplot() +
  geom_path(data = boot_roc,
    aes(x = 1 - specificity, y = sensitivity)) +
  geom_path(data = ontrain_roc,
    aes(x = 1 - specificity, y = sensitivity),
    color = "red", lty = 2) +
  geom_path(data = cv_roc,
    aes(x = 1 - specificity, y = sensitivity),
    color = "darkgrey", lty = 3) +
  geom_abline(color = "grey")
```

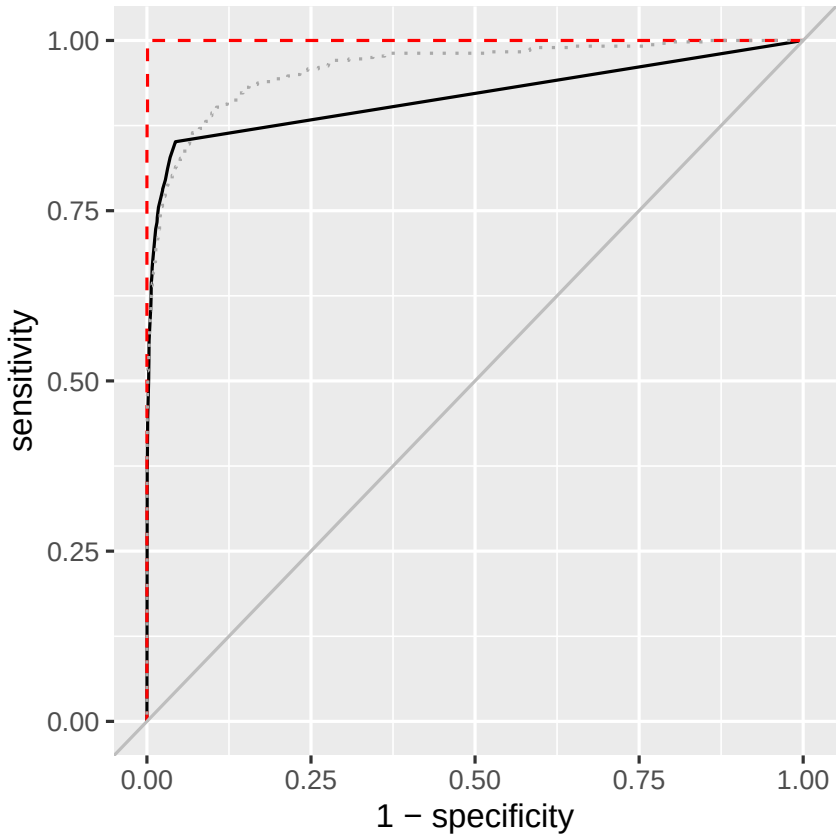


Figure 13.4: Comparison of ROC curves for bootstrap predictions (black, solid) and on-training set predictions (red, dashed) for a nearest-neighbor model. For comparison, the ROC curve for the logistic regression model is overlaid as well in grey (dotted).

Let's first compare the ROC curves for the bootstrap predictions and the predictions on the training set. The ROC curves for prediction of the full model on the training set (red curve) represents an ideal model, i.e. every data point is correctly predicted. This is expected for a nearest neighbor model. This observations emphasizes the importance of using a holdout set or cross-validation to assess model performance.

The ROC curve for the bootstrap predictions (black curve) is more realistic. It is similar to the ROC curve for the cross-validation predictions (grey curve). However, we can also see that at the beginning, the ROC curve for the bootstrap predictions is below the ROC curve for the cross-validation predictions. This confirms what we've seen from the AUC values. However, we have not explored different numbers of neighbors. In fact, as we will see in Section 15.4, increasing the number of neighbors will improve the performance of the nearest neighbor model and ultimately result in a model that has a better ROC curve compared to the logistic regression model.

13.3.1 Distribution of metrics for bootstrap samples

We can also use the bootstrap samples to assess the distribution of the performance metrics. In this case, it is however better to increase the number of resamples. Here, we use 1000 bootstrap samples.

```
set.seed(123)
nn_fit_boot <- nn_wf %>%
  fit_resamples(rsample::bootstraps(data, times = 1000),
    control = control_resamples(save_pred = TRUE))
```

We repeat the bootstrap validation more times. By default, if we call `collect_metrics` we get mean and standard error for each metric.

```
nn_fit_boot %>%
  collect_metrics()
```

```
# A tibble: 3 x 6
  .metric     .estimator   mean     n  std_err .config
  <chr>       <chr>       <dbl> <int>    <dbl> <chr>
1 accuracy    binary     0.961   1000 0.000131 Preprocessor1_Model1
2 brier_class binary     0.0329  1000 0.000102 Preprocessor1_Model1
3 roc_auc     binary     0.913   1000 0.000428 Preprocessor1_Model1
```

However, if we use the `summarize=FALSE` argument, we get the performance metrics calculated for each bootstrap sample.

```
nn_fit_boot %>%
  collect_metrics(summarize = FALSE) %>%
  head()
```

```
# A tibble: 6 x 5
  id            .metric     .estimator .estimate .config
  <chr>         <chr>       <chr>          <dbl> <chr>
1 Bootstrap0001 accuracy    binary        0.956  Preprocessor1_Model1
2 Bootstrap0001 roc_auc     binary        0.917  Preprocessor1_Model1
3 Bootstrap0001 brier_class binary        0.0350 Preprocessor1_Model1
4 Bootstrap0002 accuracy    binary        0.953  Preprocessor1_Model1
5 Bootstrap0002 roc_auc     binary        0.906  Preprocessor1_Model1
6 Bootstrap0002 brier_class binary        0.0364 Preprocessor1_Model1
```

This allows us to calculate the distribution of the performance metrics. Figure 13.5 shows the distribution of the two metrics for the bootstrap samples.

```
quantiles <- nn_fit_boot %>%
  collect_metrics(summarize = FALSE) %>%                    ①
  group_by(.metric) %>%                                     ②
  summarize(
    q0.025 = quantile(.estimate, 0.025),
    median = quantile(.estimate, 0.5),
    q0.975 = quantile(.estimate, 0.975)
  )
nn_fit_boot %>%
  collect_metrics(summarize = FALSE) %>%
  ggplot(aes(x = .estimate)) +
  geom_histogram(bins = 50, fill = "darkgrey") +            ③
  facet_wrap(~.metric, scales = "free") +
  geom_vline(data = quantiles, aes(xintercept = median),    ④
    color = "black") +
  geom_vline(data = quantiles, aes(xintercept = q0.025),
    color = "black", linetype = "dashed") +
  geom_vline(data = quantiles, aes(xintercept = q0.975),
    color = "black", linetype = "dashed")
```

① We use `summarize=FALSE` to get the performance metrics for each bootstrap sample.
② We group the data by metric and calculate the 2.5%, 50%, and 97.5% quantiles for each metric.
③ Together with the `facet_wrap` command, this creates three histograms, one for each metric.
④ This and the following two lines add vertical lines to the plot to indicate the median and the 95% confidence interval for each metric.

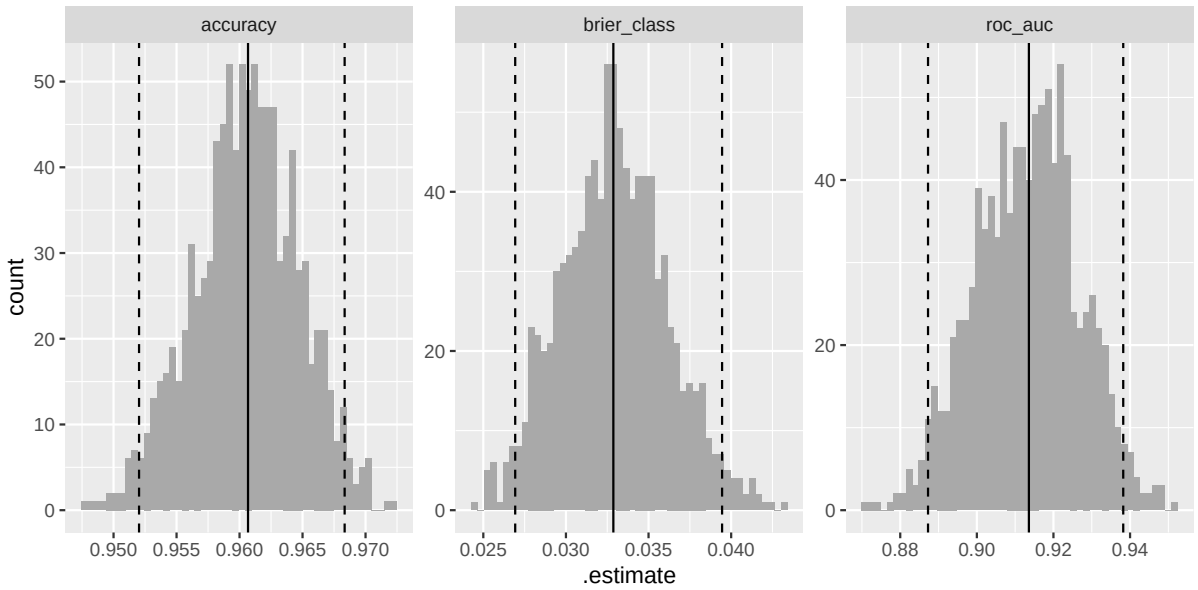


Figure 13.5: Distribution of performance metrics for bootstrap samples; the blue lines show the median and the 95% confidence interval

> **Further information**
>
> - https://tune.tidymodels.org/reference/control_grid.html control the execution of the `fit_resamples` function

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
knitr::include_graphics("images/model_workflow_validate.png")
library(tidyverse)
library(tidymodels)
library(patchwork)
library(future)
plan(multisession, workers = parallel::detectCores(logical = FALSE))
# Load an preprocess the data
data <- datasets::mtcars %>%
  as_tibble(rownames = "car") %>%
```

```
  mutate(
    vs = factor(vs, labels = c("V-shaped", "straight")),
    am = factor(am, labels = c("automatic", "manual")),
  )

# Split the data into training and test/holdout sets
set.seed(1353)
car_split <- initial_split(mtcars)
train_data <- training(car_split)
holdout_data <- testing(car_split)

# Train a model using the training set
formula <- mpg ~ cyl + disp + hp + drat + wt + qsec + vs + am +
  gear + carb
model <- linear_reg() %>%
  set_engine("lm") %>%
  fit(formula, data = train_data)
perf_train <- augment(model, new_data = train_data) %>%
  metrics(truth = mpg, estimate = .pred) %>%
  mutate(data = "Training")
perf_holdout <- augment(model, new_data = holdout_data) %>%
  metrics(truth = mpg, estimate = .pred) %>%
  mutate(data = "Holdout")
# combine the two results
bind_rows(perf_train, perf_holdout) %>%
  # select the columns of interest
  select(data, .estimate, .metric) %>%
  # convert to wide format
  pivot_wider(names_from = .metric, values_from = .estimate) %>%
  knitr::kable(digits = 2) %>%
  kableExtra::kable_styling(full_width = FALSE)
# Load and preprocess the data
file <- "https://gedeck.github.io/DS-6030/datasets/UniversalBank.csv.gz"
data <- read_csv(file)
data <- data %>%
  select(-c(ID, `ZIP Code`)) %>%
  rename(
    Personal.Loan = `Personal Loan`,
    Securities.Account = `Securities Account`,
    CD.Account = `CD Account`
  ) %>%
  mutate(
```

```
    Personal.Loan = factor(Personal.Loan, labels = c("Yes", "No"),
      levels = c(1, 0)),
    Education = factor(Education,
      labels = c("Undergrad", "Graduate", "Advanced")),
  )
# Use 10-fold cross-validation to assess model performance
set.seed(1353)
folds <- vfold_cv(data, strata = Personal.Loan)

# define the model
formula <- Personal.Loan ~ Age + Experience + Income + Family + CCAvg +
  Education + Mortgage + Securities.Account + CD.Account +
  Online + CreditCard
logreg_model <- logistic_reg() %>%
  set_engine("glm")

# define and execute the cross-validation workflow
logreg_wf <- workflow() %>%
  add_model(logreg_model) %>%
  add_formula(formula)

logreg_fit_cv <- logreg_wf %>%
  fit_resamples(folds)
logreg_fit_cv
cv_metrics <- collect_metrics(logreg_fit_cv)
cv_metrics
logreg_fit_cv <- logreg_wf %>%
  fit_resamples(folds, control = control_resamples(save_pred = TRUE))
cv_predictions <- collect_predictions(logreg_fit_cv)
cv_predictions
cv_predictions %>%
  group_by(id) %>%
  roc_curve(truth = Personal.Loan, .pred_Yes, event_level = "first") %>%
  autoplot() +
  geom_abline(lty = 2) +
  theme(legend.position = "none")
# Train a model on the full dataset
full_model <- logistic_reg() %>%
  set_engine("glm") %>%
  fit(formula, data = data)

cv_roc <- cv_predictions %>%
```

```
  roc_curve(truth = Personal.Loan, .pred_Yes, event_level = "first")

ontrain_roc <- augment(full_model, new_data = data) %>%
  roc_curve(Personal.Loan, .pred_Yes, event_level = "first")

ggplot() +
  geom_path(data = cv_roc,
    aes(x = 1 - specificity, y = sensitivity)) +
  geom_path(data = ontrain_roc,
    aes(x = 1 - specificity, y = sensitivity),
    color = "red") +
  geom_abline(lty = 2)
# Use bootstrap to assess model performance
set.seed(1353)
resamples <- rsample::bootstraps(data)

# define the model
formula <- Personal.Loan ~ Age + Experience + Income + Family +
  CCAvg + Education + Mortgage + Securities.Account +
  CD.Account + Online + CreditCard

nn_model <- nearest_neighbor(neighbors = 5) %>%
  set_mode("classification") %>%
  set_engine("kknn")

# define and execute the cross-validation workflow
nn_wf <- workflow() %>%
  add_model(nn_model) %>%
  add_formula(formula)

nn_fit_boot <- nn_wf %>%
  fit_resamples(resamples,
    control = control_resamples(save_pred = TRUE))
nn_fit_boot
boot_metrics <- collect_metrics(nn_fit_boot)

bind_rows(
  cv_metrics %>% mutate(model = "Logistic regression"),
  boot_metrics %>% mutate(model = "Nearest neighbor")
) %>%
  select(model, mean, .metric) %>%
  pivot_wider(names_from = .metric, values_from = mean) %>%
```

```
  knitr::kable(digits = 3) %>%
  kableExtra::kable_styling(full_width = FALSE)
# Train a model on the full dataset
full_model <- nn_wf %>% fit(data)

boot_roc <- collect_predictions(nn_fit_boot) %>%
  roc_curve(truth = Personal.Loan, .pred_Yes, event_level = "first")

ontrain_roc <- augment(full_model, new_data = data) %>%
  roc_curve(Personal.Loan, .pred_Yes, event_level = "first")

ggplot() +
  geom_path(data = boot_roc,
    aes(x = 1 - specificity, y = sensitivity)) +
  geom_path(data = ontrain_roc,
    aes(x = 1 - specificity, y = sensitivity),
    color = "red", lty = 2) +
  geom_path(data = cv_roc,
    aes(x = 1 - specificity, y = sensitivity),
    color = "darkgrey", lty = 3) +
  geom_abline(color = "grey")
set.seed(123)
nn_fit_boot <- nn_wf %>%
  fit_resamples(rsample::bootstraps(data, times = 1000),
    control = control_resamples(save_pred = TRUE))
nn_fit_boot %>%
  collect_metrics()
nn_fit_boot %>%
  collect_metrics(summarize = FALSE) %>%
  head()
quantiles <- nn_fit_boot %>%
  collect_metrics(summarize = FALSE) %>%                    ①
  group_by(.metric) %>%                                     ②
  summarize(
    q0.025 = quantile(.estimate, 0.025),
    median = quantile(.estimate, 0.5),
    q0.975 = quantile(.estimate, 0.975)
  )
nn_fit_boot %>%
  collect_metrics(summarize = FALSE) %>%
  ggplot(aes(x = .estimate)) +
  geom_histogram(bins = 50, fill = "darkgrey") +            ③
```

```
  facet_wrap(~.metric, scales = "free") +
  geom_vline(data = quantiles, aes(xintercept = median),                    ④
    color = "black") +
  geom_vline(data = quantiles, aes(xintercept = q0.025),
    color = "black", linetype = "dashed") +
  geom_vline(data = quantiles, aes(xintercept = q0.975),
    color = "black", linetype = "dashed")
```

14 Model tuning - the basics

In the previous chapters, we learned how to define data preprocessing steps, select a specific model type, train the model and validate its performance. In this chapter, we will learn how to tune our models to get the best performance out of them.

There are many ways a model can be tuned.

- **Feature engineering**: Feature engineering is the process of creating new features from existing features. This can be as simple as replacing a feature with its square root value, or can involve combining several features. We will learn more about feature engineering in Section 15.1.
- **Regularization**: Regularization is a technique used to prevent overfitting in models. It involves adding a penalty term to the loss function used to train the model. We will learn more about regularization in Section 15.2.
- **Feature selection**: Feature selection is the process of selecting the most important features for a given model. We will learn more about feature selection in Section 15.3.
- **Hyperparameter tuning**: Hyperparameters are parameters that are not learned during the training process. Instead, they are set before the training process starts. For example, the number of neighbors in a k-nearest neighbor model is a hyperparameter. Hyperparameter tuning is the process of finding the best hyperparameter values for a given model type. We will learn more about hyperparameter tuning in this Chapter and in Section 15.4.
- **Threshold selection**: Threshold selection is the process of finding the best threshold for a given classification model. This is currently not included in the `workflow` package, but can be done post-training using the `probably` package (see Section 11.1.2).

Figure 14.1 shows how this step fits into the overall model workflow.

Load the packages we need for this chapter.

```
library(tidymodels)
```

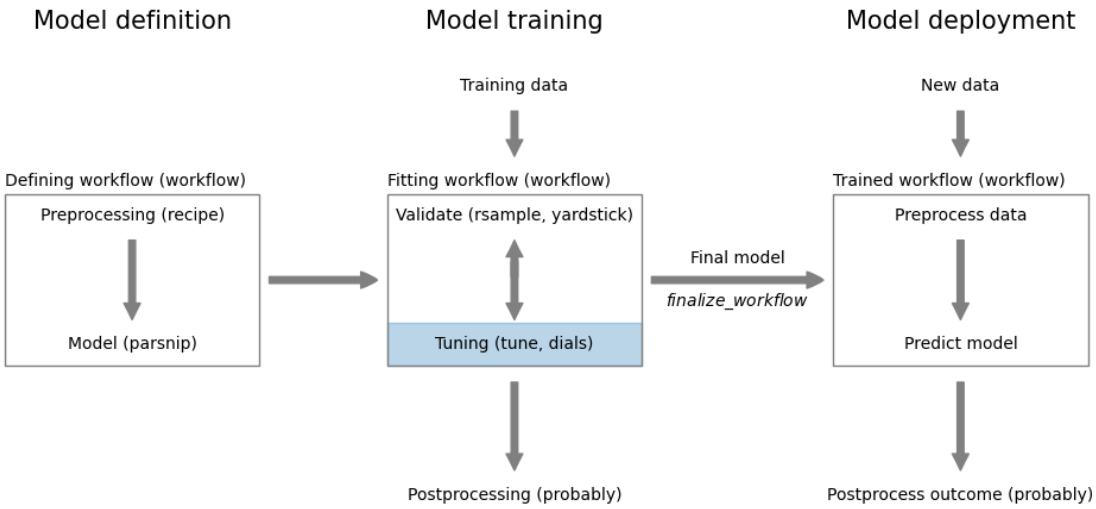


Figure 14.1: Model tuning using `tune`

14.1 Specifying tunable parameters

In the *tidymodels* framework, the `tune` package has the task of optimizing models using tuning. To do this, it needs to know what should be tuned. We can specify tunable parameters in the preprocessing steps defined using `recipe` objects and in the model definition step using `parsnip` objects.

For example, for principal component regression, we want to select the number of principal components to use in the regression model. The `step_pca` function has the `num_comp` argument that specifies the number of principal components to use. If we want to find the optimal value of `num_comp`, we specify this as a tuning parameter using the `tune()` function in the recipe.

```
mtcars_rec <- recipe(mpg ~ ., data = mtcars) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_pca(all_numeric_predictors(), num_comp = tune())
```

Another example, is the number of neighbors in a k-nearest neighbor model. We can specify this number in the `nearest_neighbor()` function using the `neighbors` argument.

```
model <- nearest_neighbor(mode = "regression", neighbors = tune())
```

By combining the recipe and model into a workflow, we define a k-nearest neighbor model using principal components where we optimize both the number of components and the number of neighbors.

```
mtcars_workflow <- workflow() %>%
  add_recipe(mtcars_rec) %>%
  add_model(model)
parameters <- extract_parameter_set_dials(mtcars_workflow)
parameters %>% knitr::kable()
```

name	id	source	component	component_id	object
neighbors	neighbors	model_spec	nearest_neighbor		integer , 1 , 15 , TRUE , TRUE , # Nearest Neighbors
num_comp	num_comp	recipe	step_pca	pca	GfWW1, 4, TRUE, TRUE, # Components, function (object, x, log_vals = FALSE, ...) , {, check_param(object), rngs <- range_get(object, original = FALSE), if (!is_unknown(rngs$upper)), return(object),, $x_dims <- -dim(x), if(is.null($x_dims)), cli :: cli_abort("Cannot determine number of columns. Is arg$x a$ == “integer” & is.null(object$trans)) {, rngs <- as.integer(rngs), }, range_set(object, rngs), }

The `extract_parameter_set_dials` function tells us that the workflow has two tunable parameters: `num_comp` and `neighbors`. The table also informs us where the parameters are used: `neighbors` is used in the `nearest_neighbor` function and `num_comp` is used in the `step_pca` function. Most importantly, it identifies the type of tuning that is used for each parameter. The `name` column tells us the function from the `dials` package that is used to optimize the parameter; see the dials documentation for more information.

The object column contains the actual definition of the search space of the parameter. Let's have a look at the default settings.

```
parameters$object[1]
```

```
[[1]]

# Nearest Neighbors (quantitative)

Range: [1, 15]
```

```
parameters$object[2]
```

```
[[1]]

# Components (quantitative)

Range: [1, 4]
```

The number of components in the PCA can range between 1 and 4, the number of neighbors between 1 and 15. This may not be suitable for our problem and it often isn't. We can change the settings for one or more of the tunable parameters using the `update` function.

```
parameters <- parameters %>%
  update(
    num_comp = num_comp(c(1, 10)),
    neighbors = neighbors(c(1, 5)),
  )
```

This increases the search range for number of components from 1 to 10 and reduces the search space for neighbors from 1 to 5.

> Useful to know
>
> If you have multiple tuning parameters in your workflow of the same type, you identify them using an identifier in the `tune` function, e.g. `tune("pca1")` and `tune("pca2")`. You can then modify the default settings for each of them separately, e.g.
>
> ```
> parameter_set <- parameter_set %>%
>   update(
>     pca1 = num_comp(10, 30),
>     pca2 = num_comp(1, 5),
>   )
> ```

14.2 Data-specific tuning parameters

The previous section we used the number of neighbors as an example. While in principle it could be set to a value larger than the number of data points in the training set, realistically

the value of the tuning parameter will always be in a range that is independent of the data set. [1]

However, there are tuning parameters, where the range can be dependent on the number of features. Some tuning parameters have a data-specific component. For example, in random forests (see Section A.12), we can control the number of features that are evaluated at each split of the decision trees with the `mtry` parameter. This value can range between 1 and the number of features in the dataset after preprocessing. The upper bound of this value is therefore unknown and needs to be explicitly set. *Tidymodel* will let you know when this is the case. Let's look at the random forest case:

```
wf <- workflow() %>%
  add_recipe(mtcars_rec) %>%
  add_model(rand_forest(mtry = tune(), mode = "regression"))
p <- extract_parameter_set_dials(wf)
p
```

```
Collection of 2 parameters for tuning
```

```
 identifier     type    object
       mtry     mtry nparam[?]
   num_comp num_comp nparam[+]
```

```
Model parameters needing finalization:

# Randomly Selected Predictors ('mtry')
```

```
See `?dials::finalize()` or `?dials::update.parameters()` for more information.
```

The output tells us that some

[1] This is not quite true for the number of principal components, however transformation into principal components is usually done only for datasets with a larger number of features.

```
Model parameters needing finalization:
# Randomly Selected Predictors ('mtry')
```

We can find out more about this by inspecting `p$object[[1]]`.

```
p$object[[1]]
```

```
# Randomly Selected Predictors (quantitative)

Range: [1, ?]
```

The range for the `mtry` parameter is defined as `[1, ?]` where `?` indicates that the upper bound is unspecified. If we use the parameters as they are, tuning will throw an error.

```
grid_regular(p)
```

```
Error in `grid_regular()`:
! These arguments contain unknowns: `mtry`.
  See the `finalize()` function.
```

We need to specify the upper bound for the `mtry` parameter. This can be done by setting a range using `dials::update_parameters` in the same way as we did for other parameters in Section 14.1. Alternatively, we can use the `dials::finalize` function as follows:

```
p <- extract_parameter_set_dials(wf) %>%
  finalize(mtcars %>% select(-mpg))
p$object[[1]]
```

```
# Randomly Selected Predictors (quantitative)

Range: [1, 10]
```

The range for the `mtry` parameter is now defined as `[1, 10]`.

14.3 Tuning a workflow

Now that we have specified the tuning parameters, we can use them to search our parameter space to identify the best model.

```
set.seed(123)
tune_results <- tune_grid(mtcars_workflow,
  resamples = vfold_cv(mtcars), grid = grid_regular(parameters))
```

The `tune_grid` function requires the `workflow`, information about the validation strategy (`resamples`) and the search space (`grid`). With the exception of the `grid` argument, this is very similar to the `fit_resamples` function that we encountered in Chapter 13. For each combination of tuning parameters defined in the `grid` argument, the `tune_grid` function trains a model and evaluates its performance.

The `grid` argument specifies the search space for the tuning parameters. In this case, we use a regular grid search. We will learn more about grid search strategies in Section 14.4.

Printing the `tune_results` object is not very informative. The output will only tell us that it's a tibble with tuning results for a 10-fold cross valiation. The tibble contains the tuning parameters and the performance metrics for each model and cross-validation fold.

You can visualize the results using the `autoplot` function:

```
autoplot(tune_results)
```

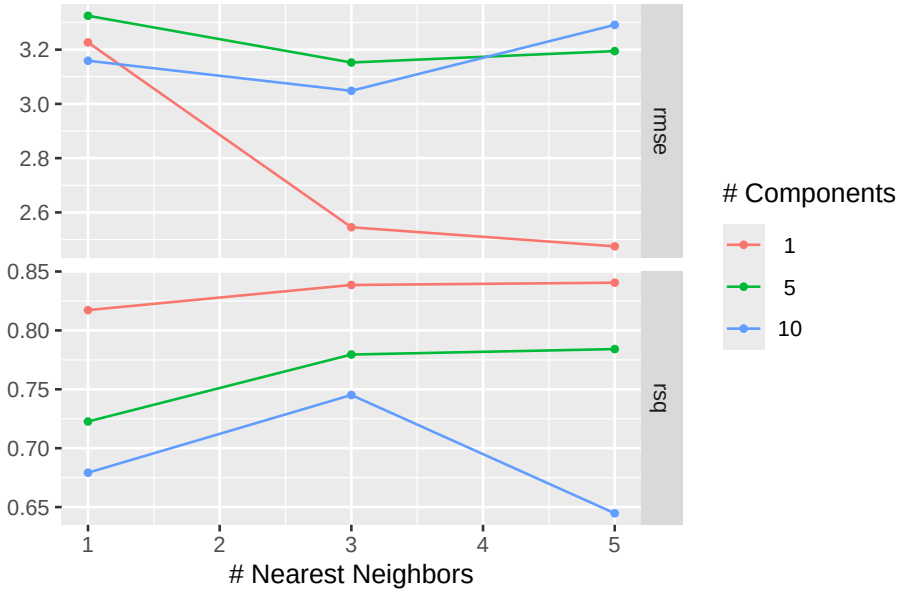


Figure 14.2: Visualization of tuning results for workflow tuning

Figure 14.2 shows the results of the tuning process. The x-axis shows the number of neighbors in the k-nearest neighbor model and the y-axis the values of the two calculated metrics, `rmse`

and `rsq`. The color of the points corresponds to the number of components in the PCA. Points with the same number of components are connected by a line. The best model is the one with the lowest RMSE. We can see that the best model has one component in the PCA and five neighbors in the k-nearest neighbor model.

The `collect_metrics` function can be used to extract the performance metrics from the tuning results.

```
collect_metrics(tune_results) %>% head(7)
```

```
# A tibble: 7 x 8
  neighbors num_comp .metric .estimator  mean     n std_err .config
      <int>    <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
1         1        1 rmse    standard   3.23     10  0.339  Preprocessor1_Model1
2         1        1 rsq     standard   0.817    10  0.0770 Preprocessor1_Model1
3         3        1 rmse    standard   2.55     10  0.233  Preprocessor1_Model2
4         3        1 rsq     standard   0.839    10  0.0910 Preprocessor1_Model2
5         5        1 rmse    standard   2.47     10  0.106  Preprocessor1_Model3
6         5        1 rsq     standard   0.841    10  0.0878 Preprocessor1_Model3
7         1        5 rmse    standard   3.32     10  0.571  Preprocessor2_Model1
```

The function determines the mean and standard deviation for each metric across the 10 folds.

The `show_best` function selects a specific metric and shows the five best model for that metric.

```
show_best(tune_results, metric = "rmse")
```

```
# A tibble: 5 x 8
  neighbors num_comp .metric .estimator  mean     n std_err .config
      <int>    <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
1         5        1 rmse    standard    2.47    10   0.106 Preprocessor1_Model3
2         3        1 rmse    standard    2.55    10   0.233 Preprocessor1_Model2
3         3       10 rmse    standard    3.05    10   0.364 Preprocessor3_Model2
4         3        5 rmse    standard    3.15    10   0.551 Preprocessor2_Model2
5         1       10 rmse    standard    3.16    10   0.549 Preprocessor3_Model1
```

Finally, the `select_best` function selects the best model based on a specific metric.

```
best_parameters <- select_best(tune_results, metric = "rmse") %>%
  select(-.config)
best_parameters
```

```
# A tibble: 1 x 2
  neighbors num_comp
      <int>    <int>
1         5        1
```

The `best_parameters` object contains the best parameters for each step in the workflow. We can use this object to finalize the workflow.

```
best_workflow <- mtcars_workflow %>%
  finalize_workflow(best_parameters) %>%
  fit(mtcars)
best_workflow
```

```
== Workflow [trained] ==========================================================
Preprocessor: Recipe
Model: nearest_neighbor()

-- Preprocessor ----------------------------------------------------------------
2 Recipe Steps

* step_normalize()
* step_pca()

-- Model -----------------------------------------------------------------------

Call:
kknn::train.kknn(formula = ..y ~ ., data = data, ks = min_rows(5L,     data, 5))

Type of response variable: continuous
minimal mean absolute error: 2.104075
Minimal mean squared error: 5.900349
Best kernel: optimal
Best k: 5
```

The `finalize_workflow` function trains the workflow using the best parameters and the full training set. The resulting workflow can be used to make predictions on new data.

```
df <- tibble(
  actual = mtcars$mpg,
  predicted = predict(best_workflow, new_data = mtcars)$.pred
)
```

```
ggplot(df, aes(x = actual, y = predicted)) +
  geom_point() +
  geom_abline()
```

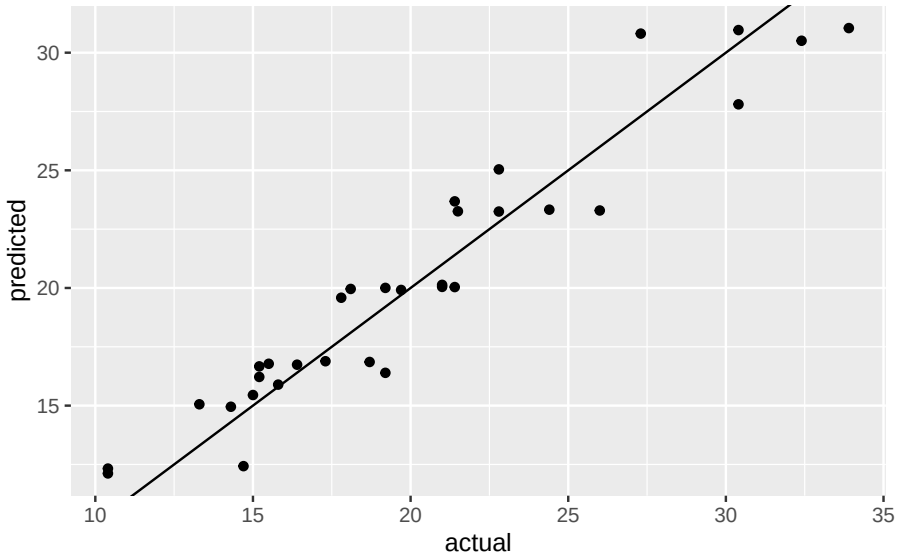


Figure 14.3: Visualization of actual versus predicted values of `mpg`

Figure 14.3 shows the actual versus predicted values for the best model. The model seems to do a good job at predicting the actual values. However, this is the prediction on the training set and we obviously need to assess the model performance on a separate holdout set.

14.4 Grid search strategies

The `tune` package implements a variety of approaches to search the parameter space.

- `grid_regular`: we used this function in the previous section; it searches the parameter space using a systematic grid of parameter values. In design of experiments, this is known as a *full factorial design*.
- `grid_random`: this approach *randomly samples* combinations of parameter values
- `grid_space_filling`: this is a general function that implements different *space filling designs* to cover the parameter space more evenly. The specific design is specified using the `type` argument. Use for example:
  - default: uses one of several pre-defined space filling designs

- `latin_hypercube`: Latin hypercube sampling is an approach that comes from the field of experimental design. It is similar to random sampling, but ensures that the parameter space is sampled more evenly.
- `max_entropy`: maximum entropy sampling also aims to cover the search space evenly.

Figure 14.4 shows the difference between the different methods using an example with two continuous parameters, `penalty` and `mixtures`, the tuning parameters of a penalized logistic regression model.

The regular grid splits each parameter range into a fixed number of points, here 10, and enumerates all possible combinations of the parameters. This results in 10 x 10 = 100 models. The random grid randomly samples 30 combinations of the parameters. We can see that there are areas of the parameters space that are not well covered. The Latin hypercube and maximum entropy sampling approaches also randomly sample 30 combinations of the parameters, but in this case, there seems to be a more uniform coverage of the parameter space.

Let's compare the different search strategies using a real example. We will use the `mtcars` dataset and try to predict the fuel efficiency of cars using a linear regression model. We will use the `glmnet` engine which allows us to tune L1 and L2 regularization using the penalty and mixture parameters in the linear regression model.

```
set.seed(123)

recipe <- recipe(mpg ~ ., data = mtcars) %>%
  step_normalize(all_numeric_predictors())
model <- linear_reg(mode = "regression", engine = "glmnet",
  penalty = tune(), mixture = tune())
wf <- workflow() %>%                                                    ①
  add_recipe(recipe) %>%
  add_model(model)
parameters <- extract_parameter_set_dials(wf) %>%                       ②
  update(
    penalty = penalty(c(-3, 0.75))
  )
```

① Define workflow for a regularized linear regression model
② Define the tuning parameter ranges; defaut values are used for mixture; penalty is adjusted based on preliminary runs (not shown)

We can now apply the different search strategies to tune the model.

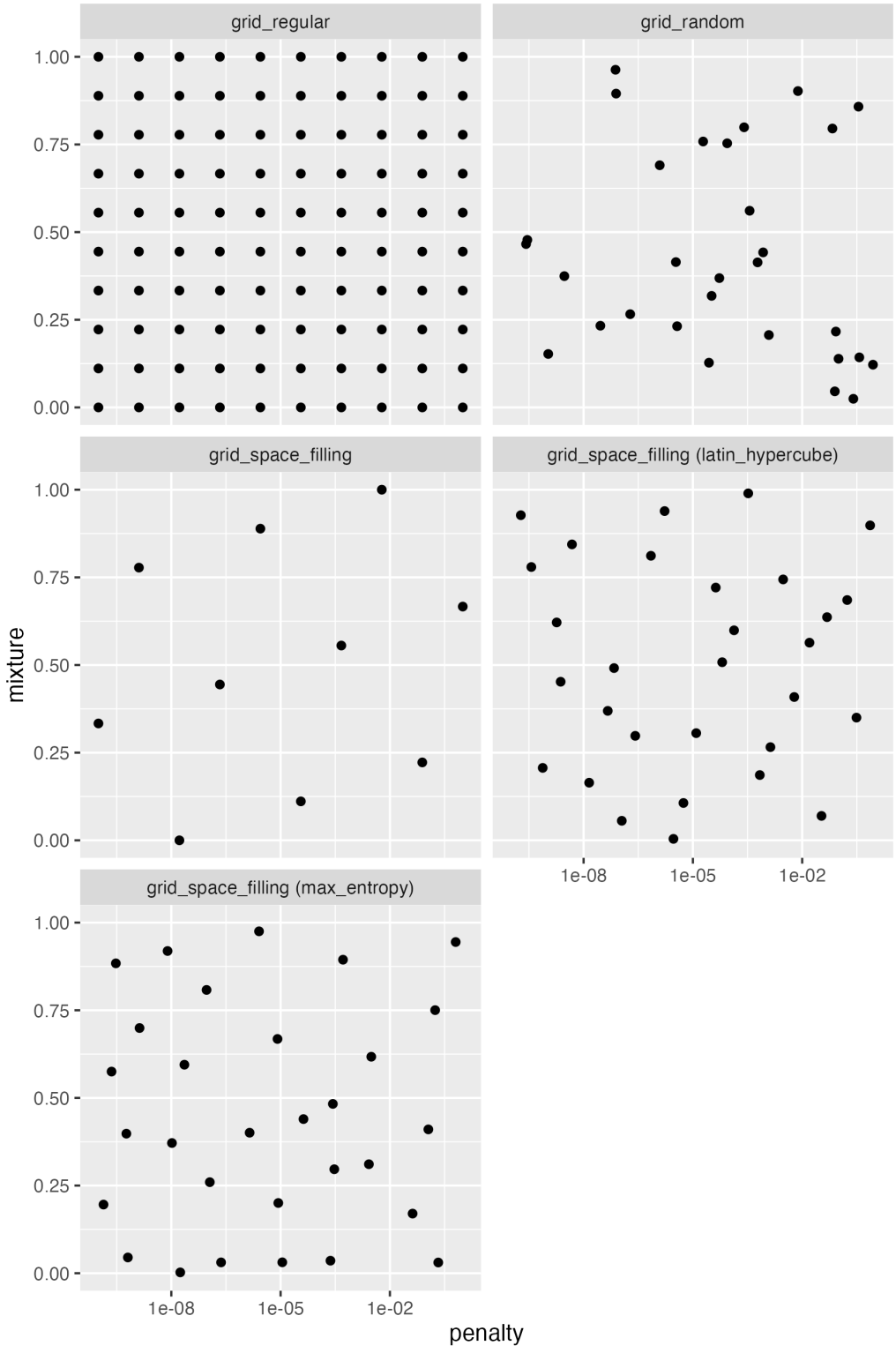


Figure 14.4: Model tuning using `tune`

```
resamples <- vfold_cv(mtcars)                                          ①
model_regular <- tune_grid(wf, resamples = resamples,                  ②
  grid = grid_regular(parameters, levels = 10))
nrandom <- 30
model_random <- tune_grid(wf, resamples = resamples,                   ③
  grid = grid_random(parameters, size = nrandom))
model_latin_hypercube <- tune_grid(wf, resamples = resamples,
  grid = grid_space_filling(parameters, size = nrandom,
    type = "latin_hypercube"))
model_max_entropy <- tune_grid(wf, resamples = resamples,
  grid = grid_space_filling(parameters, size = nrandom,
    type = "max_entropy"))
```

① For better comparison, we use the same resamples for all tuning runs.
② In this tuning run, we use a regular grid with 10 levels for each parameter, resulting in 100 combinations.
③ This and the following two tuning runs, use random sampling from the parameter space. The first uses random sampling, the second uses a latin hypercube, and the third uses max entropy sampling. All three explore 30 random combinations of the parameters.

```
df <- rbind(
  cbind(grid = "regular", tested = 100,
    model_regular %>% show_best(metric = "rmse", n = 1)),
  cbind(grid = "random", tested = nrandom,
    model_random %>% show_best(metric = "rmse", n = 1)),
  cbind(grid = "latin_hypercube", tested = nrandom,
    model_latin_hypercube %>% show_best(metric = "rmse", n = 1)),
  cbind(grid = "max_entropy", tested = nrandom,
    model_max_entropy %>% show_best(metric = "rmse", n = 1))
) %>%
  select(-c(.metric, .estimator, n, .config)) %>%
  mutate(
    grid = factor(grid, levels = c(
      "regular", "random", "latin_hypercube", "max_entropy"))
  )
```

The following table shows the best model for each search strategy.

```
df %>%
  rename(RMSE = "mean") %>%
  mutate_if(is.numeric, format, digits = 3, nsmall = 0) %>%
  knitr::kable()
```

grid	tested	penalty	mixture	RMSE	std_err
regular	100	0.825	1.0000	2.48	0.447
random	30	1.654	0.0705	2.51	0.411
latin_hypercube	30	1.851	0.1087	2.52	0.421
max_entropy	30	1.201	0.7782	2.53	0.466

Superficially, it seems that the regular grid search resulted in the best model. However, the difference to the other models is very small and well within the observed variation of the estimated cross-validation performance. Figure 14.5 shows the mean and standard deviation of the RMSE for each search strategy.

```
ggplot(df, aes(x = grid, y = mean,
    ymin = mean - std_err, ymax = mean + std_err)) +
  geom_point() +
  geom_pointrange() +
  xlab("Method to define grid") + ylab("Mean rmsq error")
```

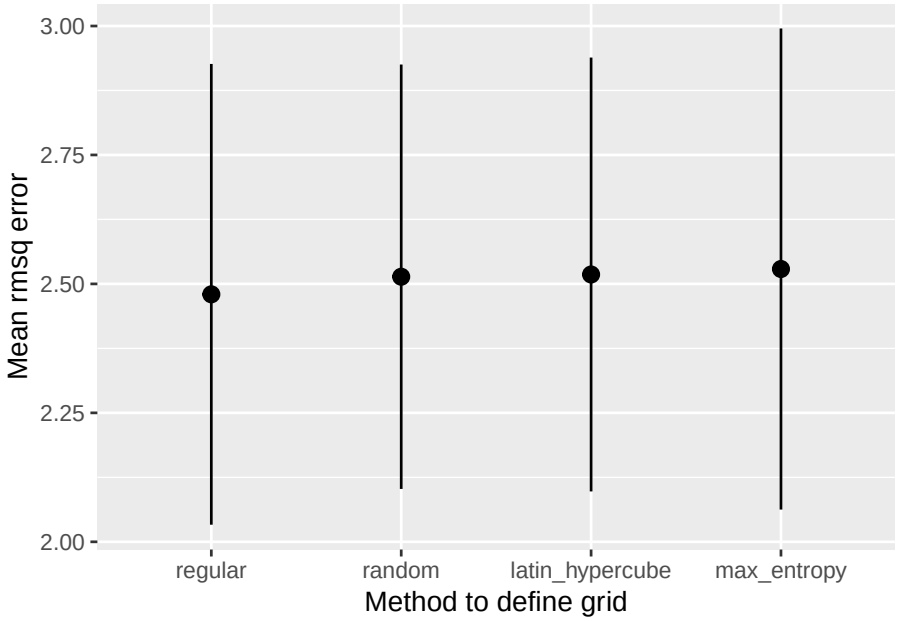


Figure 14.5: Comparison of different search strategies

It is also important to stress, that the regular grid search is much more computationally expensive than the other approaches. The regular grid search tested 100 models, the other methods tested only 30 different models. It also seems as if the best models are obtained for

a mixture value of 1, i.e. a pure lasso model. If the best value in the search space is at the boundary of the search space, it is likely that random methods will not find it. If you think this is the case, remove the parameter from tuning and fix the value in the model definition.

> Useful to know
>
> In the literature, the regular grid search is usually taught as the default approach to tuning. However, it is not necessarily the best approach. The random approaches are often a better choice. They can be more efficient requiring fewer model evaluations.

14.5 Bayesian Hyperparameter optimization

The `tune` package also implements Bayesian hyperparameter optimization. Bayesian optimization is a sequential approach to hyperparameter optimization. To initialize the method, random sampling is used to get a rough exploration of the parameter space. Using the validation results, a Bayesian model, a Gaussian process model to be precise, is trained using the parameter combinations as predictors and the validation performance as the outcome. The Gaussian process model then predicts the validation performance for a large number of parameter combinations. This value combined with the uncertainty of the prediction is used to prioritize the next parameter combination finding a trade-off between global exploration and local exploitation. The most useful combination is then evaluated and the process is repeated with the old and new data until a stopping criteria is met (maximum number of iterations or no improvement).

Bayesian hyperparameter optimization is available with the `tune_bayes` function.

```
model_bayes <- tune_bayes(wf, resamples = resamples,
  param_info = parameters, iter = 25)
```

With these settings, the function will create an initial model with 5 evaluations followed by 25 iterations. The `param_info` argument is used to specify the search space. The `iter` argument specifies the number of iterations. The `tune_bayes` function returns a `tune_results` object that can be used in the same way as the `tune_grid` function.

Figure 14.6 demonstrates the process of the Bayesian hyperparameter optimization.

```
regular_metrics <- model_regular %>%
  collect_metrics() %>%
  filter(.metric == "rmse")
bayes_metrics <- model_bayes %>%
  collect_metrics() %>%
```

```
  filter(.metric == "rmse")

shapes <- c(
  "Grid search" = 16,
  "Bayesian optimization\n(initial phase)" = 17,
  "Bayesian optimization\n(iterations)" = 15,
  "Bayesian optimization\n(best model)" = 18
)

ggplot(regular_metrics, aes(x = penalty, y = mixture, z = mean)) +
  geom_point(data = df, mapping = aes(shape = "Grid search"),
    color = "red", size = 3) +
  geom_contour(color = "black", alpha = 0.75, bins = 20) +
  geom_point(data = bayes_metrics %>% head(5),
    mapping = aes(shape = "Bayesian optimization\n(initial phase)"),
    size = 3, color = "#6CABBC") +
  geom_point(data = bayes_metrics %>% tail(-5),
    mapping = aes(shape = "Bayesian optimization\n(iterations)"),
    color = "#226DCE") +
  geom_point(
    data = model_bayes %>% show_best(n = 1, metric = "rmse"),
    mapping = aes(shape = "Bayesian optimization\n(best model)"),
    color = "#85db66", size = 3) +
  scale_x_log10() +
  scale_shape_manual(name = "Tuning method", values = shapes)
```

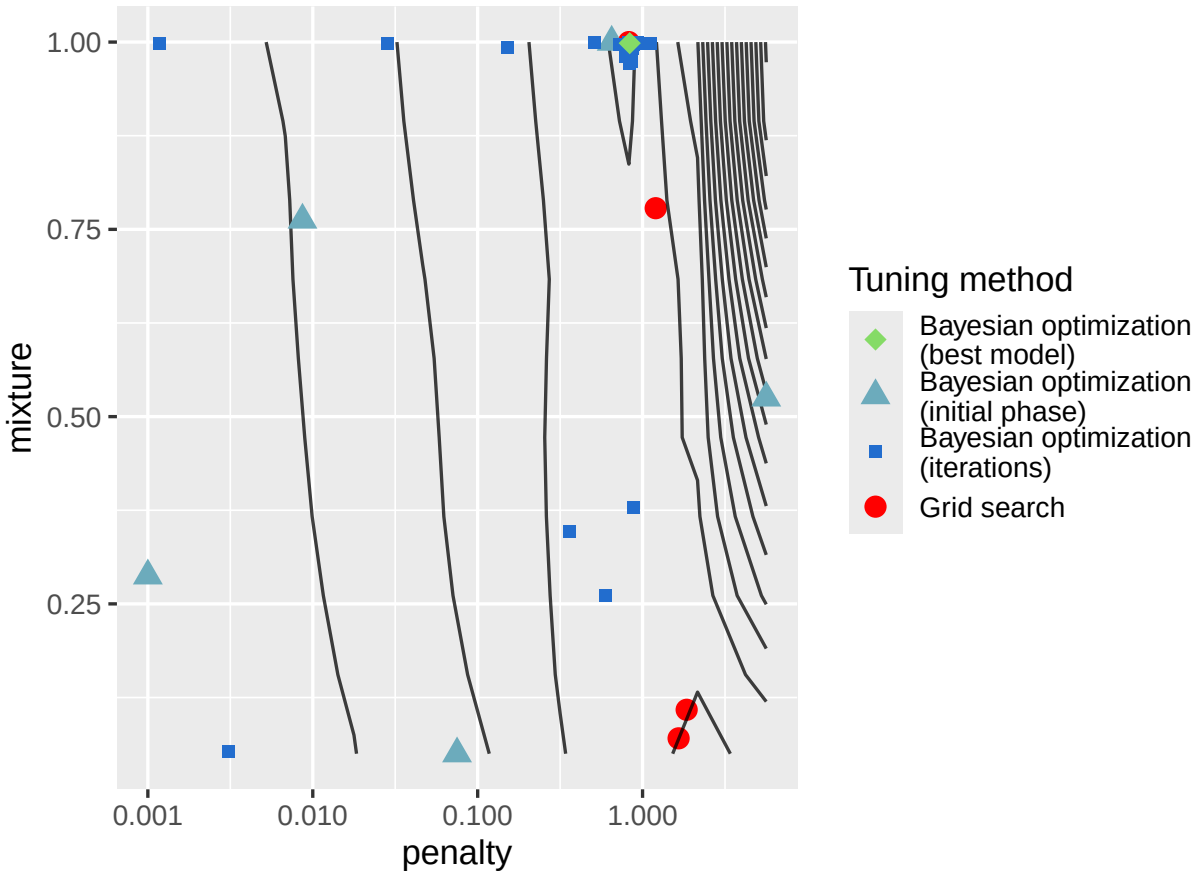


Figure 14.6: Exploration of parameter space during Bayesian hyperparameter tuning for a penalized linear regression model.

The plot shows the parameter combinations that were evaluated during the optimization process. The contour plot was determined from the validation performance calculated for the regular grid search. The blue points show the parameter combinations that were evaluated during the Bayesian hyperparameter optimization. Light blue points were evaluated in the initial phase, the darker blue points were evaluated in the subsequent iterations. The dark blue point highlights the *best* model found during these iterations. The red point shows the best results from the above grid searches for comparison.

Figure 14.6 clearly shows that the Bayesian hyperparameter optimization fullfills two objectives, exploration by evaluating parameter combinations in areas of the parameter space that have not been explored before and exploitation by evaluating parameter combinations that are close to the best parameter combination found so far.

Further information

- https://tune.tidymodels.org/ `tune` package
- https://dials.tidymodels.org/ `dials` package

There are more packages that extend the capabilities of the `tune` package.

- `finetune` package implements iterative methods similar to the Bayesian hyperparameter optimization approach, e.g. simulated annealing for global optimization

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
knitr::include_graphics("images/model_workflow_tune.png")
library(tidymodels)
mtcars_rec <- recipe(mpg ~ ., data = mtcars) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_pca(all_numeric_predictors(), num_comp = tune())
model <- nearest_neighbor(mode = "regression", neighbors = tune())
mtcars_workflow <- workflow() %>%
  add_recipe(mtcars_rec) %>%
  add_model(model)
parameters <- extract_parameter_set_dials(mtcars_workflow)
parameters %>% knitr::kable()
parameters$object[1]
parameters$object[2]
parameters <- parameters %>%
  update(
    num_comp = num_comp(c(1, 10)),
    neighbors = neighbors(c(1, 5)),
  )
wf <- workflow() %>%
  add_recipe(mtcars_rec) %>%
  add_model(rand_forest(mtry = tune(), mode = "regression"))
p <- extract_parameter_set_dials(wf)
p
p$object[[1]]
p <- extract_parameter_set_dials(wf) %>%
```

```
  finalize(mtcars %>% select(-mpg))
p$object[[1]]
set.seed(123)
tune_results <- tune_grid(mtcars_workflow,
  resamples = vfold_cv(mtcars), grid = grid_regular(parameters))
autoplot(tune_results)
collect_metrics(tune_results) %>% head(7)
show_best(tune_results, metric = "rmse")
best_parameters <- select_best(tune_results, metric = "rmse") %>%
  select(-.config)
best_parameters
best_workflow <- mtcars_workflow %>%
  finalize_workflow(best_parameters) %>%
  fit(mtcars)
best_workflow
df <- tibble(
  actual = mtcars$mpg,
  predicted = predict(best_workflow, new_data = mtcars)$.pred
)

ggplot(df, aes(x = actual, y = predicted)) +
  geom_point() +
  geom_abline()
knitr::include_graphics("images/tuning-grids.png")
set.seed(123)

recipe <- recipe(mpg ~ ., data = mtcars) %>%
  step_normalize(all_numeric_predictors())
model <- linear_reg(mode = "regression", engine = "glmnet",
  penalty = tune(), mixture = tune())
wf <- workflow() %>%                                                    ①
  add_recipe(recipe) %>%
  add_model(model)
parameters <- extract_parameter_set_dials(wf) %>%                       ②
  update(
    penalty = penalty(c(-3, 0.75))
  )
resamples <- vfold_cv(mtcars)                                           ①
model_regular <- tune_grid(wf, resamples = resamples,                   ②
  grid = grid_regular(parameters, levels = 10))
nrandom <- 30
model_random <- tune_grid(wf, resamples = resamples,                    ③
```

```
  grid = grid_random(parameters, size = nrandom))
model_latin_hypercube <- tune_grid(wf, resamples = resamples,
  grid = grid_space_filling(parameters, size = nrandom,
    type = "latin_hypercube"))
model_max_entropy <- tune_grid(wf, resamples = resamples,
  grid = grid_space_filling(parameters, size = nrandom,
    type = "max_entropy"))
df <- rbind(
  cbind(grid = "regular", tested = 100,
    model_regular %>% show_best(metric = "rmse", n = 1)),
  cbind(grid = "random", tested = nrandom,
    model_random %>% show_best(metric = "rmse", n = 1)),
  cbind(grid = "latin_hypercube", tested = nrandom,
    model_latin_hypercube %>% show_best(metric = "rmse", n = 1)),
  cbind(grid = "max_entropy", tested = nrandom,
    model_max_entropy %>% show_best(metric = "rmse", n = 1))
) %>%
  select(-c(.metric, .estimator, n, .config)) %>%
  mutate(
    grid = factor(grid, levels = c(
      "regular", "random", "latin_hypercube", "max_entropy"))
  )
df %>%
  rename(RMSE = "mean") %>%
  mutate_if(is.numeric, format, digits = 3, nsmall = 0) %>%
  knitr::kable()
ggplot(df, aes(x = grid, y = mean,
    ymin = mean - std_err, ymax = mean + std_err)) +
  geom_point() +
  geom_pointrange() +
  xlab("Method to define grid") + ylab("Mean rmsq error")
model_bayes <- tune_bayes(wf, resamples = resamples,
  param_info = parameters, iter = 25)
regular_metrics <- model_regular %>%
  collect_metrics() %>%
  filter(.metric == "rmse")
bayes_metrics <- model_bayes %>%
  collect_metrics() %>%
  filter(.metric == "rmse")

shapes <- c(
  "Grid search" = 16,
```

```
  "Bayesian optimization\n(initial phase)" = 17,
  "Bayesian optimization\n(iterations)" = 15,
  "Bayesian optimization\n(best model)" = 18
)

ggplot(regular_metrics, aes(x = penalty, y = mixture, z = mean)) +
  geom_point(data = df, mapping = aes(shape = "Grid search"),
    color = "red", size = 3) +
  geom_contour(color = "black", alpha = 0.75, bins = 20) +
  geom_point(data = bayes_metrics %>% head(5),
    mapping = aes(shape = "Bayesian optimization\n(initial phase)"),
    size = 3, color = "#6CABBC") +
  geom_point(data = bayes_metrics %>% tail(-5),
    mapping = aes(shape = "Bayesian optimization\n(iterations)"),
    color = "#226DCE") +
  geom_point(
    data = model_bayes %>% show_best(n = 1, metric = "rmse"),
    mapping = aes(shape = "Bayesian optimization\n(best model)"),
    color = "#85db66", size = 3) +
  scale_x_log10() +
  scale_shape_manual(name = "Tuning method", values = shapes)
```

15 Model tuning - examples

In the previous chapters, we learned how to define and tune hyperparameters using the *tidymodels* packagesdials`and`tune‘. In this chapter, we will go deeper into specific areas of model tuning.

- Feture engineering: introduce non-linearity using polynomial regression, step functions, and splines
- Regularization: control model complexity using L1, L2, and elasticnet regularization
- Feature selection: select features using random forest variable importance
- Hyperparameter tuning - what to look out for when tuning a model

Figure 15.1 shows how this step fits into the overall model workflow.

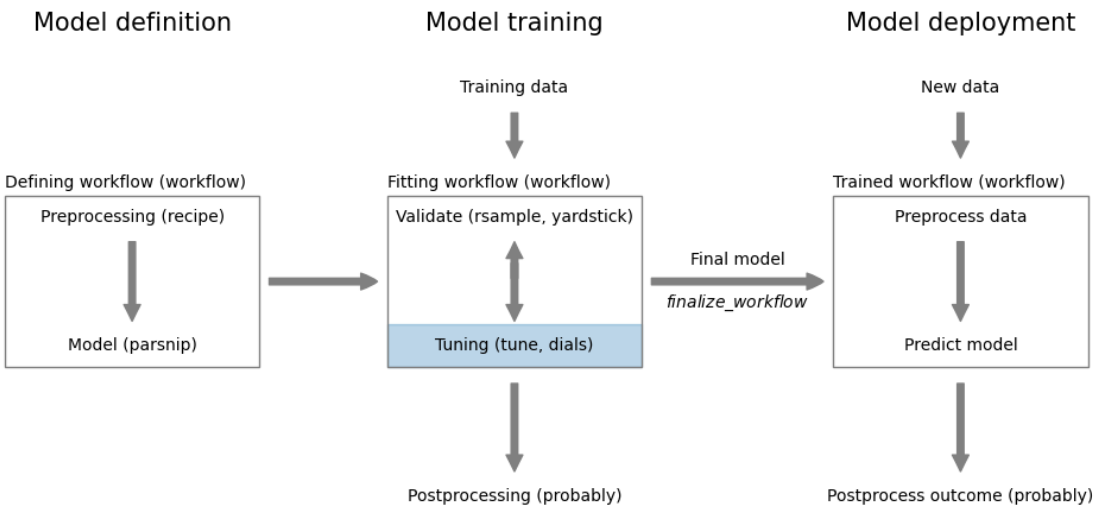


Figure 15.1: Model tuning using `tune`

Load the packages we need for this chapter.

```
library(tidymodels)
library(tidyverse)
library(patchwork)
```

Because tuning requires training many models, we also enable parallel computing.

```
library(future)
plan(multisession, workers = parallel::detectCores(logical = FALSE))
```

15.1 Feature engineering

We already saw in Chapter 14 that preprocessing steps have tunable parameters. The function `tunable()` can be used to identify tunable parameters in a recipe. For example, the following code shows how to identify tunable parameters in a recipe that contains a polynomial step.

```
# tidymodels is very picky about data types and will complain when we
# predict on new data if the age value is not an integer. We therefore
# convert here age to double
data <- ISLR2::Wage %>%
  mutate(age = as.double(age))

recipe(wage ~ age, data = data) %>%
  step_poly(hp) %>%
  step_discretize(hp) %>%
  step_cut(hp, breaks = tune()) %>%
  step_bs(hp) %>%
  tunable()
```

```
# A tibble: 5 x 5
  name       call_info        source component       component_id
  <chr>      <list>           <chr>  <chr>           <chr>
1 degree     <named list [2]> recipe step_poly       poly_eQOaC
2 min_unique <named list [2]> recipe step_discretize discretize_Fsvuk
3 num_breaks <named list [2]> recipe step_discretize discretize_Fsvuk
4 deg_free   <named list [3]> recipe step_bs         bs_V1IPT
5 degree     <named list [3]> recipe step_bs         bs_V1IPT
```

We see that the polynomial step has a tunable parameter `degree`.

15.1.1 Polynomial regression

With this information, we can tune a polynomial regression model.

```r
set.seed(123)
poly_recipe <- recipe(wage ~ age, data = data) %>%
  step_poly(age, degree = tune())
model <- linear_reg(mode = "regression") %>%
  set_engine("glm")
poly_workflow <- workflow() %>%
  add_recipe(poly_recipe) %>%
  add_model(model)
tune_results <- tune_grid(poly_workflow, resamples = vfold_cv(data))
tune_results %>% show_best(metric = "rmse")
```

```
# A tibble: 3 x 7
  degree .metric .estimator  mean     n std_err .config
   <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
1      3 rmse    standard    39.7    10    1.30 Preprocessor2_Model1
2      2 rmse    standard    39.8    10    1.30 Preprocessor1_Model1
3      1 rmse    standard    40.8    10    1.28 Preprocessor3_Model1
```

By default, this will tune the polynomial degree from 1 to 3. The best model was with a quadratic polynomial. We can use the `finalize_workflow()` function to fit the best model to the entire data set.

```r
best_parameters <- select_best(tune_results, metric = "rmse")
final_model <- poly_workflow %>%
  finalize_workflow(best_parameters) %>%
  fit(data)
```

Figure 15.2 shows the best fit of the polynomial regression model. The shaded area shows the 95% confidence interval of the fit. The polynomial regression model is a linear model, but the polynomial step allows us to fit a non-linear relationship between the predictor and the outcome.

```r
df <- tibble(age = seq(min(data$age), max(data$age), length.out = 100))
df %>%
  bind_cols(
    predict(final_model, new_data = df),
    predict(final_model, new_data = df, type = "conf_int")
  ) %>%
  ggplot(aes(x = age, y = .pred)) +
  geom_point(aes(x = age, y = wage), data = data, alpha = 0.1) +
  geom_line() +
```

```
  geom_ribbon(aes(ymin = .pred_lower, ymax = .pred_upper), alpha = 0.2) +
  labs(x = "Age", y = "Wage")
```

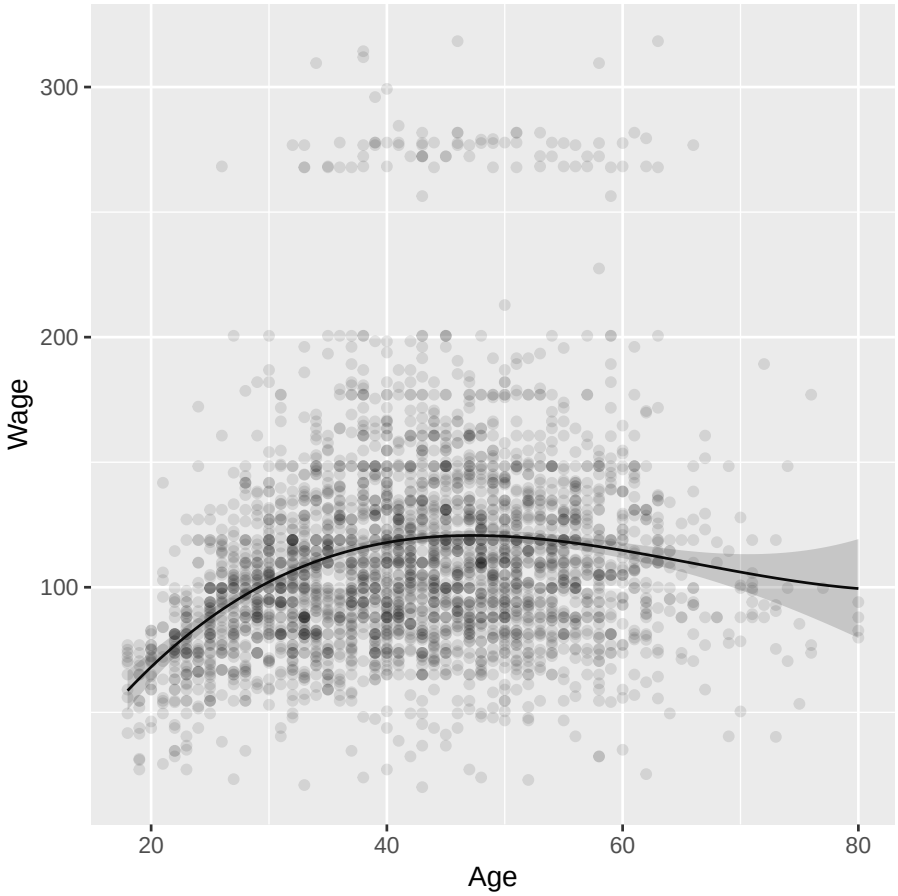


Figure 15.2: Polynomial regression model

15.1.2 Step function regression

In this section, we develop a model using a step function. The `step_discretize()` function is used to convert the `age` features into bins. The tunable parameter is `num_breaks`.

```
set.seed(123)
step_recipe <- recipe(wage ~ age, data = data) %>%
  step_discretize(age, num_breaks = tune())
step_workflow <- workflow() %>%
  add_recipe(step_recipe) %>%
```

```
  add_model(model)
tune_results <- tune_grid(step_workflow, resamples = vfold_cv(data))
tune_results %>% show_best(metric = "rmse")
```

```
# A tibble: 5 x 7
  num_breaks .metric .estimator  mean     n std_err .config
       <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
1          5 rmse    standard    40.3    10    1.27 Preprocessor1_Model1
2          4 rmse    standard    40.3    10    1.25 Preprocessor4_Model1
3          3 rmse    standard    40.6    10    1.26 Preprocessor3_Model1
4          9 rmse    standard    40.8    10    1.28 Preprocessor2_Model1
5          7 rmse    standard    40.8    10    1.28 Preprocessor5_Model1
```

Tuning determines that the best model uses 5 bins. We can use the `finalize_workflow()` function to fit the best model to the entire data set.

```
best_parameters <- select_best(tune_results, metric = "rmse")
final_model <- step_workflow %>%
  finalize_workflow(best_parameters) %>%
  fit(data)
```

Figure 15.3 shows the best fit of the regression model using a step function.

```
best_breaks <- (tune_results %>% show_best(metric = "rmse"))$num_breaks[1]
cuts <- tibble(breaks = quantile(data$age,
    probs = seq(0, 1, by = 1 / best_breaks)))
df %>%
  bind_cols(
    predict(final_model, new_data = df),
    predict(final_model, new_data = df, type = "conf_int")
  ) %>%
  ggplot(aes(x = age, y = .pred)) +
  geom_vline(aes(xintercept = breaks), data = cuts,
    color = "darkgreen", alpha = 0.5) +
  geom_point(aes(x = age, y = wage), data = data, alpha = 0.1) +
  geom_line() +
  geom_ribbon(aes(ymin = .pred_lower, ymax = .pred_upper), alpha = 0.2) +
  labs(x = "Age", y = "Wage")
```

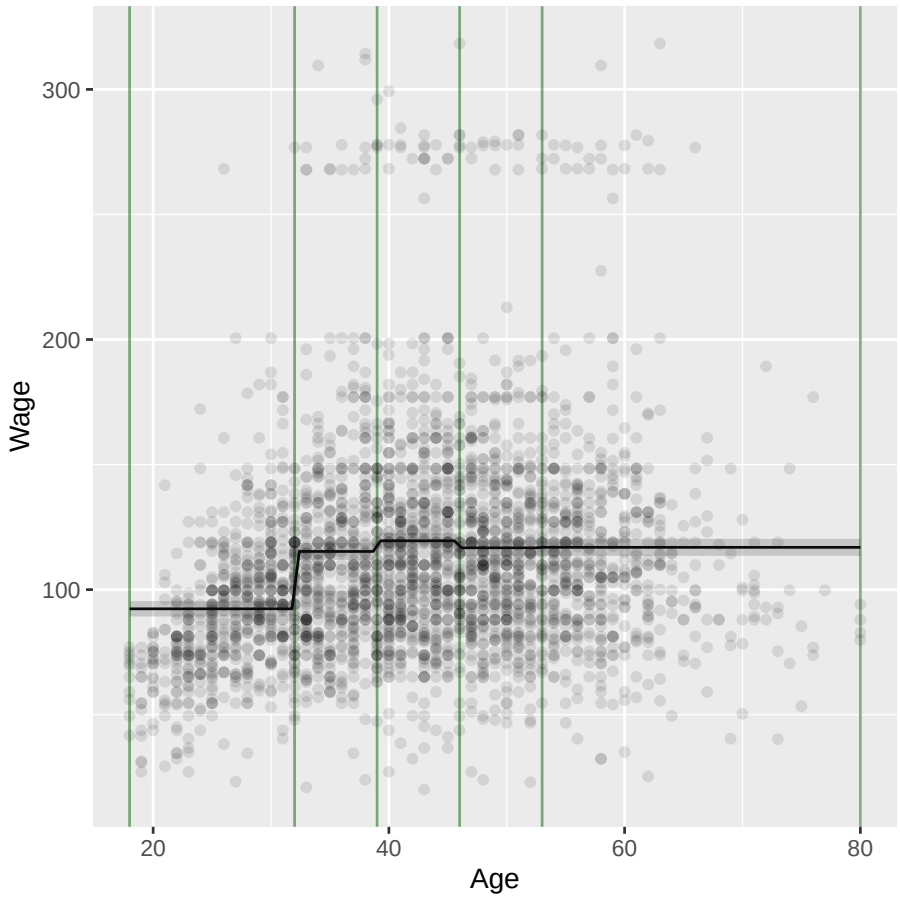


Figure 15.3: Stepwise regression model; the green lines show the bin boundaries.

The model developed in this section is different from the step function model described in Sections 7.2 and 7.8.1 of *An introduction to statistical learning* (James et al. 2021). In the book, the authors use `cut(age, 4)` to create a categorical variable with four levels. This leads to different boundaries than defined by the quantiles that the `step_discretize()` function uses. If you want to use defined boundaries, e.g. split into age groups of 10 years, you can use

```
step_cut(age, breaks=seq(20, 70, by=10),
    include_outside_range=TRUE)
```

The breaks are however not tunable in this case. Figure 15.4 shows the results of this model.

```
step_recipe <- recipe(wage ~ age, data = data) %>%
  step_cut(age, breaks = seq(20, 70, by = 10), include_outside_range = TRUE)
step_workflow <- workflow() %>%
  add_recipe(step_recipe) %>%
  add_model(model)
trained <- step_workflow %>% fit(data)
cuts <- tibble(breaks = seq(20, 70, by = 10))
df %>%
  bind_cols(
    predict(trained, new_data = df),
    predict(trained, new_data = df, type = "conf_int")
  ) %>%
  ggplot(aes(x = age, y = .pred)) +
  geom_vline(aes(xintercept = breaks), data = cuts,
    color = "darkgreen", alpha = 0.5) +
  geom_point(aes(x = age, y = wage), data = data, alpha = 0.1) +
  geom_line() +
  geom_ribbon(aes(ymin = .pred_lower, ymax = .pred_upper), alpha = 0.2) +
  labs(x = "Age", y = "Wage")
```

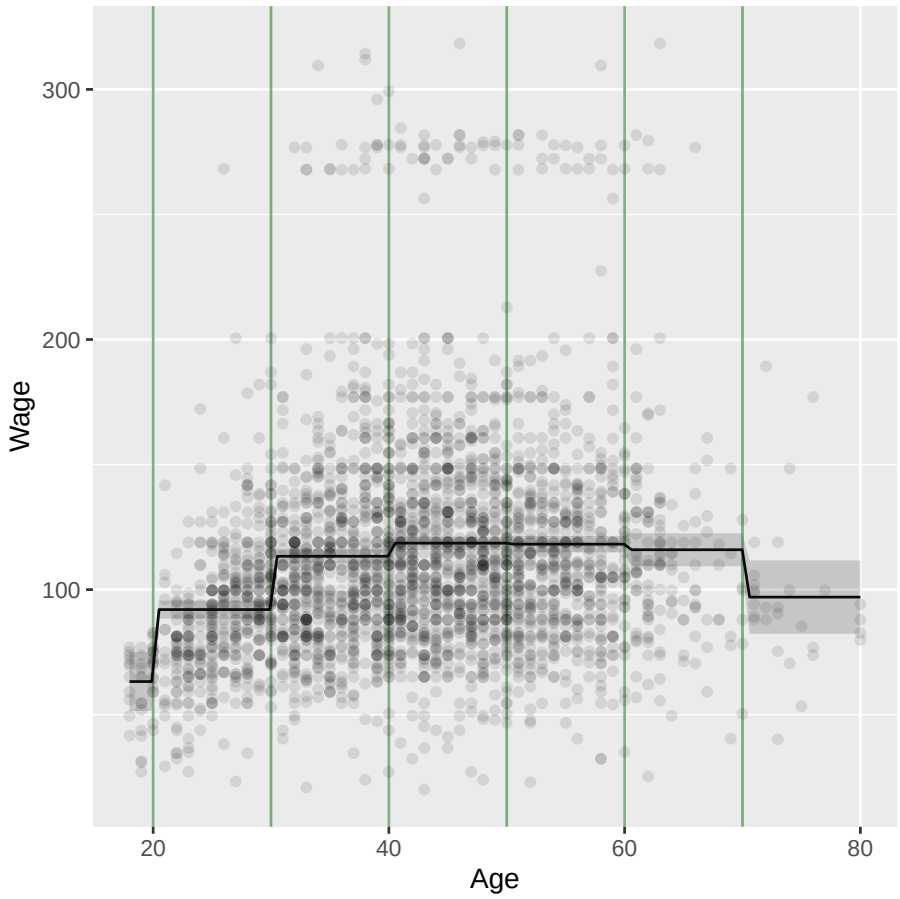


Figure 15.4: Stepwise regression model with fixed bin boundaries.

15.1.3 Spline regression

Next we will tune a model using a spline representation for `age`.

```
set.seed(123)
spline_recipe <- recipe(wage ~ age, data = data) %>%
  step_bs(age, degree = tune(), deg_free = tune())
spline_workflow <- workflow() %>%
  add_recipe(spline_recipe) %>%
  add_model(model)
tune_results <- tune_grid(spline_workflow, resamples = vfold_cv(data))
tune_results %>% show_best(metric = "rmse")
```

```
# A tibble: 5 x 8
```

```
  deg_free degree .metric .estimator  mean     n std_err .config
     <int>  <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
1        8      1 rmse    standard    39.7    10    1.28 Preprocessor06_Model1
2        7      2 rmse    standard    39.7    10    1.29 Preprocessor05_Model1
3       10      2 rmse    standard    39.8    10    1.29 Preprocessor07_Model1
4       11      2 rmse    standard    39.8    10    1.29 Preprocessor08_Model1
5        4      1 rmse    standard    39.8    10    1.30 Preprocessor03_Model1
```

```
best_parameters <- select_best(tune_results, metric = "rmse")
final_model <- spline_workflow %>%
  finalize_workflow(best_parameters) %>%
  fit(data)
```

Figure 15.5 shows the best fit of the spline regression model.

```
df %>%
  bind_cols(
    predict(final_model, new_data = df),
    predict(final_model, new_data = df, type = "conf_int")
  ) %>%
  ggplot(aes(x = age, y = .pred)) +
  geom_point(aes(x = age, y = wage), data = data, alpha = 0.1) +
  geom_line() +
  geom_ribbon(aes(ymin = .pred_lower, ymax = .pred_upper), alpha = 0.2) +
  labs(x = "Age", y = "Wage")
```

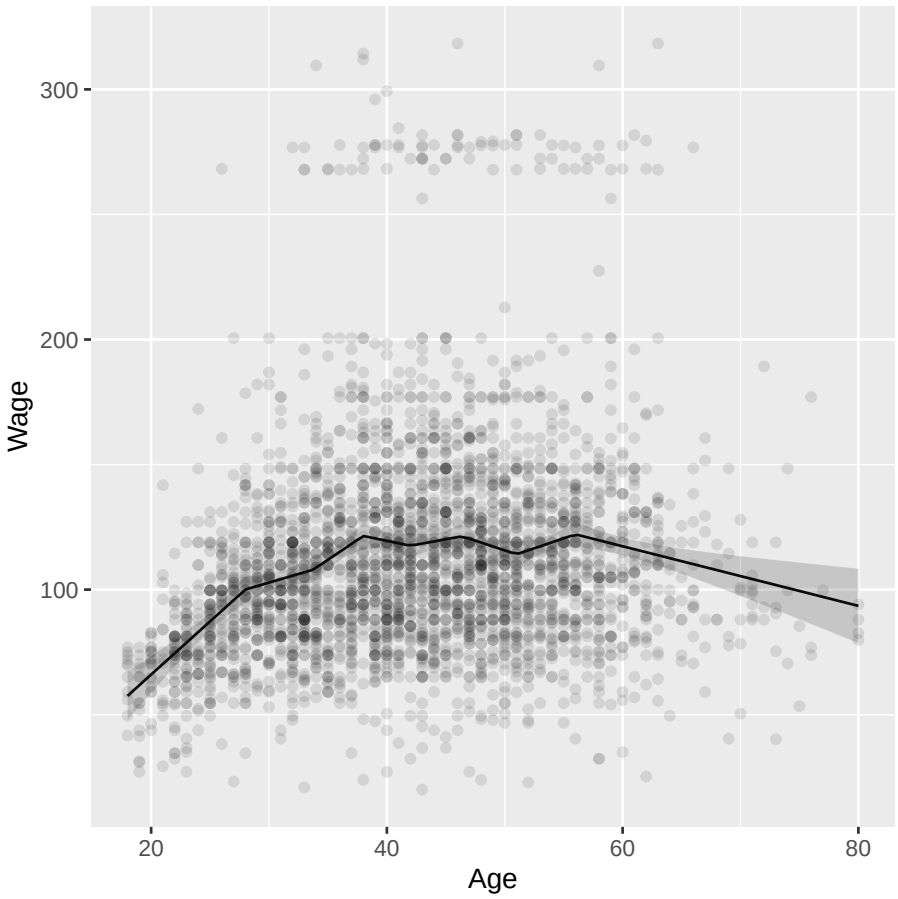


Figure 15.5: Spline regression model

Other examples of preprocessing steps that can be tuned for feature engineering are:

- `step_pca` using the `num_comp` parameter
- `step_nzv` to remove variables that are highly sparse and unbalanced using the `freq_cut` or `unique_cut` parameters

Todo

Compare Figure 15.5 with Figure 15.2. Which one do you prefer? Why?

15.2 Regularization

Regularization is used in several machine learning methods. In general, it is any approach that controls the complexity of a model. For linear models, there are two common types

of regularization. Both control the complexity of the model by penalizing the size of the coefficients.

L2-regularization (ridge penalty):

$$\text{Penalty}_{L2}(\beta, \lambda) = \lambda \sum_{j=1}^{p} |\beta_j|^2 = \beta^T \beta$$

L1-regularization (lasso penalty):

$$\text{Penalty}_{L1}(\beta, \lambda) = \lambda \sum_{j=1}^{p} |\beta_j|$$

Both approaches differ in how they penalize the size of the coefficients. The L2-regularization penalizes the sum of the squared coefficients, whereas the L1-regularization penalizes the sum of the absolute values of the coefficients. The added penalty causes the coefficients to shrink towards zero. The amount of shrinkage is controlled by the regularization parameter λ. The larger λ, the more the coefficients are shrunk towards zero. Figure 15.6 demonstrates the difference between the two regularization approaches. In L2 regularization, the coefficients are shrunk towards zero, but they are may never be exactly zero. In L1 regularization, the coefficients can be exactly zero. This means that L1 regularization can be used for feature selection.

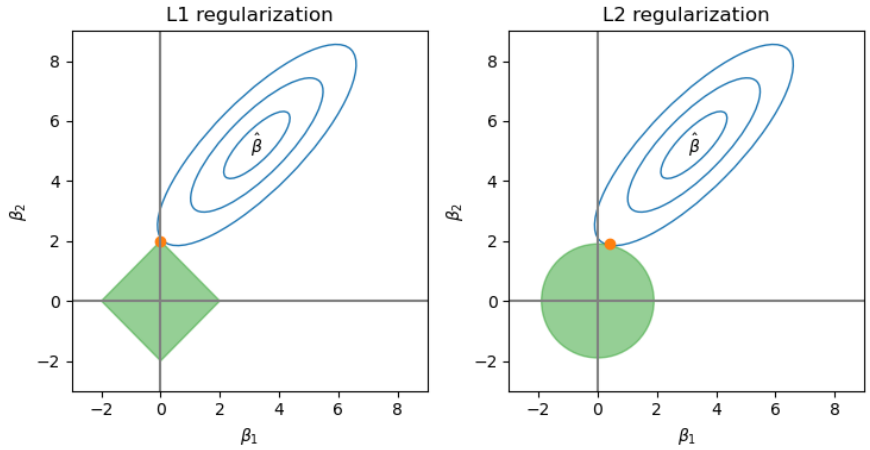


Figure 15.6: Contours of the error and constraint functions for lasso (left) and ridge (right) regularization. The green areas are the constraint regions, $|\beta_1| + |\beta_2| \leq t$ and $\beta_1^2 + \beta_2^2 \leq t^2$, while the blue ellipses are the contours of the RSS (residual sum of squares). The optimal coefficients are the points where the ellipses touch the constraint regions (red).

In some cases, it is useful to combine the L1 and L2 regularization. This is called elasticnet regularization.[1] The elasticnet penalty is a weighted combination of the L1 and L2 penalties. The weight α controls the relative contribution of the L1 and L2 penalties. When $\alpha = 0$, the elasticnet penalty is equivalent to the L2 penalty. When $\alpha = 1$, the elasticnet penalty is equivalent to the L1 penalty.

$$P_{elastic}(\beta) = \alpha P_{L1}(\beta, \lambda) + \frac{1}{2}(1 - \alpha) P_{L2} P(\beta, \lambda)$$

The `glmnet` package implements both L1 and L2 regularization as well as the elasticnet penalty. We've already seen in Section 14.4 how we can explore the `penalty` and `mixture` hyperparameters with *tidymodels*. See Chapter 21 for more details.

15.3 Feature selection

Many text books cover methods like *forward and backward stepwise selection* and *best subset selection*. The authors of *tidymodels* do not recommended these approaches for feature selection. It's unlikely that they will add these methods to *tidymodels* in the future. Instead, they suggest using regularization, in particular L1 regularization, or other methods. In fact, they work on a new package called `colino` that implements several approaches to feature selection.

The `colino` package is still in development and is not yet available on CRAN. You can find the development version on GitHub and install it with the following code.

```
if (!require(colino)) {
  devtools::install_github("stevenpawley/colino", force = TRUE)
}
library(colino)
```

The methods are all implemented as steps compatible with the `recipe` package.

> **Todo**
>
> Look through the package documentation at https://stevenpawley.github.io/colino/ to get an overview of the various methods.

The following code shows how to use the `step_select_forests()` function to select features based on variable importance derived from a random forest model. The actual model will be a k-nearest neighbor model.

[1] You can find different definitions of elasticnet regularization in the literature. Here, we use the definition from the `glmnet` package.

```
set.seed(123)
resamples <- vfold_cv(mtcars)
rec_select <- recipe(mpg ~ ., data = mtcars) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_select_forests(all_predictors(), outcome = "mpg", top_p = tune())
model <- nearest_neighbor(mode = "regression", neighbors = tune())
mtcars_workflow <- workflow() %>%
  add_recipe(rec_select) %>%
  add_model(model)
parameters <- extract_parameter_set_dials(mtcars_workflow)
parameters %>% knitr::kable()
```

name	id	source	component	component_object	object
neighbors	neighbors	model_spec	nearest_neighbor		integer , 1 , 15 , TRUE , TRUE , # Nearest Neighbors
top_p	top_p	recipe	step_select_forests	forests	integer, 1, 4, TRUE, TRUE, # Selected Predictors, function (object, x, ...) , {, check_param(object), rngs <- range_get(object, original = FALSE), if (!is_unknown(rngs$upper)), return(object),, $x_dims < -dim(x), if(is.null(x_dims)), cli :: cli_abort("Cannot determine number of columns. Is arg x == "integer" & is.null(object$trans)) {, rngs <- as.integer(rngs), }, range_set(object, rngs), }

The `step_select_forests()` function has various tunable parameters. Here, we use `top_p`, the number of most important features to keep.

Now we have everything we need to tune the workflow.[2]

```
tune_results <- tune_grid(mtcars_workflow,
  resamples = resamples,
  grid = grid_random(parameters, size = 50))
tune_results %>%
  show_best(metric = "rmse") %>%
  select(-.config) %>%
  knitr::kable()
```

[2] I initially ran this code using `size=10` for the random grid search. There was a large gap between the performance of the best model and the other models. This is an indication that the search was not exhaustive enough.

neighbors	top_p	.metric	.estimator	mean	n	std_err
6	4	rmse	standard	2.341911	10	0.2005014
8	4	rmse	standard	2.494542	10	0.2542292
1	4	rmse	standard	2.515561	10	0.2915867
9	4	rmse	standard	2.565404	10	0.2747478
3	3	rmse	standard	2.588270	10	0.3163073

The best model selected 4 features and 6 for the number of neighbors in the k-NN model. We can use the `finalize_workflow()` function to fit the best model to the entire data set and visualize the results, see Figure 15.7.

```
best_parameters <- select_best(tune_results, metric = "rmse")
best_workflow <- mtcars_workflow %>%
  finalize_workflow(best_parameters) %>%
  fit(mtcars)
```

```
mtcars %>%
  bind_cols(
    predict(best_workflow, new_data = mtcars)
  ) %>%
  ggplot(aes(x = mpg, y = .pred)) +
  geom_point() +
  geom_abline() +
  xlim(10, 35) + ylim(10, 35) +
  labs(x = "Observed mpg", y = "Predicted mpg")
```

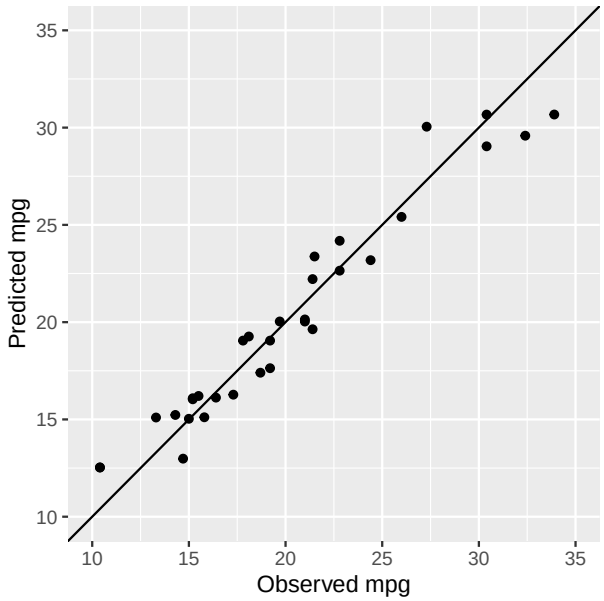


Figure 15.7: Feature selection using random forest variable importance

We can also extract the recipe and look at the results from the feature importance.

```
best_workflow %>%
  extract_recipe() %>%
  tidy(number = 2, type = "scores")
```

```
# A tibble: 10 x 3
   variable  score id
   <chr>     <dbl> <chr>
 1 wt       100    select_forests_iOjwA
 2 disp      80.3  select_forests_iOjwA
 3 cyl       74.8  select_forests_iOjwA
 4 hp        63.8  select_forests_iOjwA
 5 carb      12.8  select_forests_iOjwA
 6 drat       9.67 select_forests_iOjwA
 7 vs         4.65 select_forests_iOjwA
 8 qsec       4.37 select_forests_iOjwA
 9 am         3.50 select_forests_iOjwA
10 gear       0    select_forests_iOjwA
```

```
feature_scores <- best_workflow %>%
  extract_recipe() %>%
```

```
  tidy(number = 2, type = "scores")
kept_features <- feature_scores$variable[1:nfeatures]
kept_features
```

```
[1] "wt"   "disp" "cyl"  "hp"
```

The top-4 features used in the final model are wt, disp, cyl, hp.

15.4 Hyperparameter tuning

We already covered hyperparameter tuning in Chapter 14, so what else could there be said? Let's revisit the example from Chapter 13. In that chapter, we compared the performance of a k-nearest neighbor model and a logistic regression model to predict the probability of a customer taking a loan. Comparing the ROC curves showed strange behavior of the k-NN model. It's likely that our model requires further tuning.

15.4.1 Define the hyperparameter search space

Let's start with loading and preprocessing the data.

```
file <- "https://gedeck.github.io/DS-6030/datasets/UniversalBank.csv.gz"
data <- read_csv(file)
data <- data %>%
  select(-c(ID, `ZIP Code`)) %>%
  rename(
    Personal.Loan = `Personal Loan`,
    Securities.Account = `Securities Account`,
    CD.Account = `CD Account`
  ) %>%
  mutate(
    Personal.Loan = factor(Personal.Loan, labels = c("Yes", "No"),
      levels = c(1, 0)),
    Education = factor(Education,
      labels = c("Undergrad", "Graduate", "Advanced")),
  )
```

Next we repeat the model training from Chapter 13. We use 10-fold crossvalidation to estimate the performance of the models.

```
set.seed(1353)
folds <- vfold_cv(data, strata = Personal.Loan)

formula <- Personal.Loan ~ Age + Experience + Income + Family + CCAvg +
  Education + Mortgage + Securities.Account + CD.Account +
  Online + CreditCard

# Train the logistic regression model
logreg_wf <- workflow() %>%
  add_model(
    logistic_reg() %>% set_engine("glm")
  ) %>%
  add_formula(formula)
logreg_cv <- logreg_wf %>%
  fit_resamples(resamples = folds,
    control = control_resamples(save_pred = TRUE))
logreg_cv_predictions <- collect_predictions(logreg_cv)

# Train the nearest-neighbor model with 5 neighbors
nn5_wf <- workflow() %>%
  add_model(nearest_neighbor(neighbors = 5) %>%
      set_mode("classification") %>%
      set_engine("kknn")) %>%
  add_formula(formula)
nn5_cv <- nn5_wf %>%
  fit_resamples(resamples = folds,
    control = control_resamples(save_pred = TRUE))
nn5_cv_predictions <- collect_predictions(nn5_cv)
```

Figure 15.10 shows the ROC curves for the logistic regression and the 5-NN model. The logistic regression model performs better than the 5-NN model. The 5-NN model has a strange shape, which is likely due to the fact that we did not tune the model.

Let's tune the model with the default settings

```
# Tune the number of neighbors of the nearest-neighbor model
nn_wf <- workflow() %>%
  add_model(nearest_neighbor(neighbors = tune()) %>%
      set_mode("classification") %>%
      set_engine("kknn")) %>%
  add_formula(formula)
nn_default_tune <- tune_grid(nn_wf, resamples = folds)
autoplot(nn_default_tune)
```

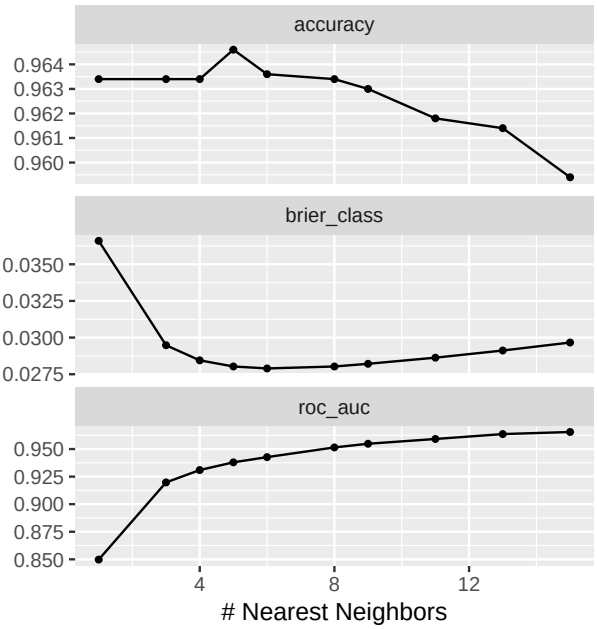


Figure 15.8: Tuning results for the k-NN model with default settings

Figure 15.8 shows the results of the tuning process. There are two interesting observations to point out. First, the selected number of neighbors depends on the metric. If we use `accuracy`, the optimal number of neighbors is 5, the same value we used in our initial model. For ROC AUC, the optimal number of neighbors is 15.

The second observation is that the `roc_auc` curve has not reached a maximum. This means that we have not explored the entire parameter space. We can increase the number of neighbors to explore the parameter space further.

```
parameters <- extract_parameter_set_dials(nn_wf) %>%
  update(neighbors = neighbors(c(1, 100)))

nn_bayes_tune <- tune_bayes(nn_wf, resamples = folds,
  param_info = parameters, iter = 25)
autoplot(nn_bayes_tune)
```

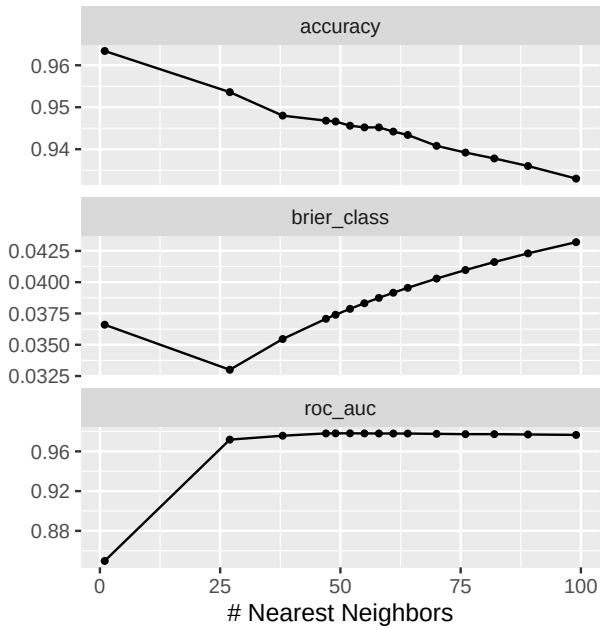


Figure 15.9: Tuning results for the k-NN model with a larger number of neighbors

```
optimal_nn_roc <- nn_bayes_tune %>%
  select_best(metric = "roc_auc")
```

This time, the hyperparameter search identified the optimal number of neighbors as 52.

```
cv_roc <- logreg_cv_predictions %>%
  roc_curve(truth = Personal.Loan, .pred_Yes, event_level = "first")
nn5_roc <- nn5_cv_predictions %>%
  roc_curve(truth = Personal.Loan, .pred_Yes, event_level = "first")

nn_default_auc <- nn_wf %>%
  finalize_workflow(select_best(nn_default_tune, metric = "roc_auc")) %>%
  fit_resamples(resamples = folds,
    control = control_resamples(save_pred = TRUE))
nn_default_auc_roc <- nn_default_auc %>%
  collect_predictions() %>%
  roc_curve(truth = Personal.Loan, .pred_Yes, event_level = "first")

nn_bayes_auc <- nn_wf %>%
  finalize_workflow(select_best(nn_bayes_tune, metric = "roc_auc")) %>%
  fit_resamples(resamples = folds,
    control = control_resamples(save_pred = TRUE))
```

```
nn_bayes_auc_roc <- nn_bayes_auc %>%
  collect_predictions() %>%
  roc_curve(truth = Personal.Loan, .pred_Yes, event_level = "first")

g1 <- ggplot() +
  geom_path(data = cv_roc, aes(x = 1 - specificity, y = sensitivity),
    color = "gray") +
  geom_path(data = nn5_roc,
    aes(x = 1 - specificity, y = sensitivity)) +
  geom_abline(lty = 2) +
  labs(title = "(a) Initial model (accuracy, k=5)")

g2 <- ggplot() +
  geom_path(data = cv_roc, aes(x = 1 - specificity, y = sensitivity),
    color = "gray") +
  geom_path(data = nn_default_auc_roc,
    aes(x = 1 - specificity, y = sensitivity)) +
  geom_abline(lty = 2) +
  labs(title = "(b) Default model (AUC, k=14)")

g3 <- ggplot() +
  geom_path(data = cv_roc, aes(x = 1 - specificity, y = sensitivity),
    color = "gray") +
  geom_path(data = nn_bayes_auc_roc,
    aes(x = 1 - specificity, y = sensitivity)) +
  geom_abline(lty = 2) +
  labs(title = "(c) Optimal model (AUC, k=48)")

g1 + g2 + g3
```

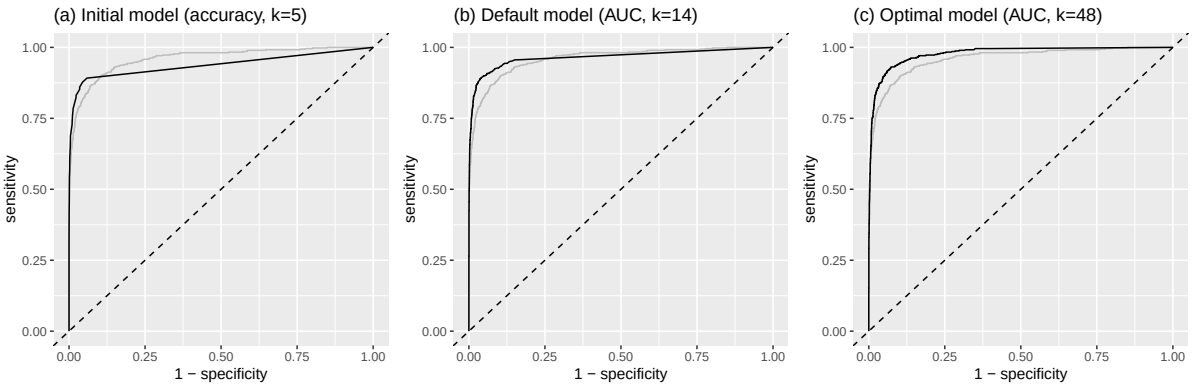


Figure 15.10: ROC curves for the logistic regression and the 5-NN model (left) and the tuned NN model (right)

Comparing the three ROC curves, we see that the fully tuned model, with sufficient exploration of the hyperparameter space, performs better than the initial model across the full data range.

> **Useful to know**
>
> Always check the tuning results using the `autoplot` function. If the maximum (or minimum) of your metric is not within the defined parameter space, you should increase the range of the hyperparameters.

Let's summarize the data in a table and Figure 15.11:

```
logreg_metrics <- collect_metrics(logreg_cv)
nn5_metrics <- collect_metrics(nn5_cv)
nn_default_metrics <- collect_metrics(nn_default_auc)
nn_bayes_metrics <- collect_metrics(nn_bayes_auc)

df <- bind_rows(
  logreg_metrics %>% mutate(model = "Logistic regression"),
  nn5_metrics %>% mutate(model = "Nearest neighbor (k=5)"),
  nn_default_metrics %>% mutate(model = "Nearest neighbor (k=14)"),
  nn_bayes_metrics %>% mutate(model = "Nearest neighbor (k=48)"),
)
df %>%
  select(model, mean, .metric) %>%
  pivot_wider(names_from = .metric, values_from = mean) %>%
  knitr::kable(digits = 3) %>%
  kableExtra::kable_styling(full_width = FALSE)
```

model	accuracy	brier_class	roc_auc
Logistic regression	0.958	0.032	0.961
Nearest neighbor (k=5)	0.965	0.028	0.938
Nearest neighbor (k=14)	0.959	0.030	0.965
Nearest neighbor (k=48)	0.946	0.038	0.978

```
df %>%
  mutate(
    model = factor(model,
      levels = rev(c(
        "Logistic regression",
        "Nearest neighbor (k=5)",
        "Nearest neighbor (k=14)",
        "Nearest neighbor (k=48)"))),
  ) %>%
  ggplot(aes(x = mean, y = model)) +
  geom_point() +
  geom_pointrange(aes(xmin = mean - std_err, xmax = mean + std_err)) +
  facet_wrap(~ .metric, scales = "free_x")
```

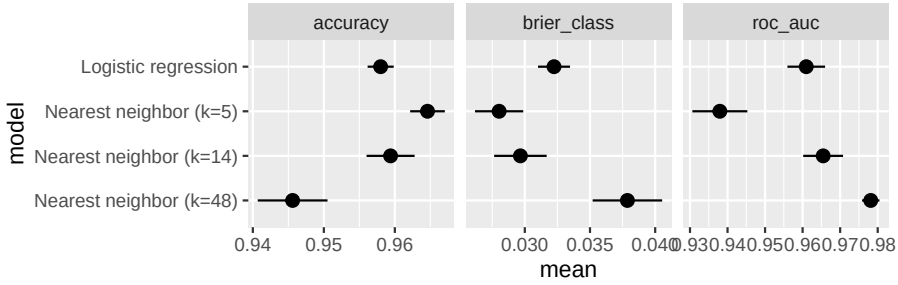


Figure 15.11: Comparison of the performance metrics for the different models

Tuning the hyperparameters of the k-NN model improved the ROC AUC to larger value than the logistic regression. The accuracy on the other hand dropped. However, as we learned in Chapter 11, accuracy is a performance metric that depends on the threshold used for classification.

15.4.2 Tune the threshold

Using the methods described in Section 11.1.2, we can tweak the thresholds of the logistic regression and the best k-NN model. In the following example, we use `model %>%`

`collect_predictions()` to use the out-of-fold predictions from cross-validation with the `probably::threshold_perf` method.

```
threshold_graph <- function(model) {
  performance <- probably::threshold_perf(
    model %>% collect_predictions(),
    Personal.Loan, .pred_Yes,
    thresholds = seq(0.05, 0.9, 0.01), event_level = "first",
    metrics = metric_set(accuracy, kap, bal_accuracy, f_meas)
  )
  max_values <- performance %>%
    arrange(desc(.threshold)) %>%
    group_by(.metric) %>%
    filter(.estimate == max(.estimate)) %>%
    filter(row_number() == 1)
  g <- ggplot(performance,
    aes(x = .threshold, y = .estimate, color = .metric)) +
    geom_line() +
    geom_vline(data = max_values,
      aes(xintercept = .threshold, color = .metric)) +
    scale_x_continuous(breaks = seq(0, 1, 0.1)) +
    coord_cartesian(ylim = c(0.5, 1)) +
    xlab("Threshold") + ylab("Metric value") +
    theme(legend.position = "inside",
      legend.position.inside = c(0.85, 0.75))
  return(g)
}
g1 <- threshold_graph(logreg_cv) +
  labs(title = "Logistic regression model")
g2 <- threshold_graph(nn_bayes_auc) +
  labs(title = "k-nearest neighbor model")

g1 + g2
```

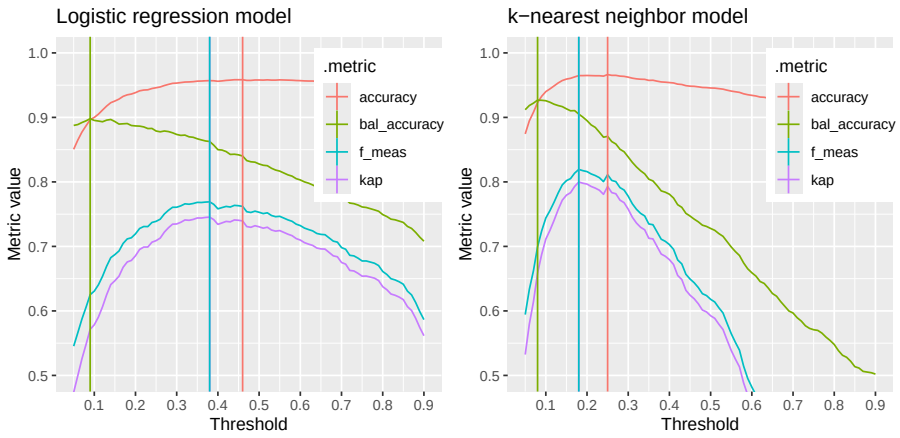


Figure 15.12: Metric values for different thresholds of the logistic regression and the k-NN model

Figure 15.12 shows the performance of the logistic regression and the k-NN model for different thresholds. For all metrics, the threshold dependent maxima are higher for k-NN compared to logistic regression.

> Useful to know
>
> In this section, we learned that it is important to:
>
> - sufficiently explore the hyperparameter space
> - use the right metric for model selection (ROC)
> - select the threshold after you identified the best model

15.5 The *one-standard-error* rule

Breiman et al (Breiman et al. 1984) suggested that instead of selecting the best model, we should select the simplest model within one standard error of the best model. This is called the *one-standard-error* rule. The idea is that the best model is likely to be overfitted and that a simpler model is likely to generalize better.

Let's demonstrate this with a ridge regression model to predict `mpg` in the `mtcars` dataset. First tune the model with a suitable range for the penalty parameter.

```
set.seed(123)
resamples <- vfold_cv(mtcars)
rec <- recipe(mpg ~ ., data = mtcars)
```

```
spec <- linear_reg(mode = "regression", penalty = tune(), mixture = 0) %>%
  set_engine("glmnet")
wf <- workflow() %>%
  add_recipe(rec) %>%
  add_model(spec)
parameters <- extract_parameter_set_dials(wf) %>%
  update(penalty = penalty(c(-2, 2)))
tune_results <- tune_grid(wf,
  resamples = resamples,
  grid = grid_regular(parameters, levels = 50))
```

The *one-standard-error* rule is implemented in the `select_by_one_std_err()` function. It is used in a similar way to the `select_best` function.

```
penalty_best <- select_best(tune_results, metric = "rmse")
penalty_1se <- select_by_one_std_err(tune_results, desc(penalty),
  metric = "rmse")
```

The implementation doesn't know what a *simpler* model is, so we need to tell it. In this case, we use the penalty parameter as a proxy for model complexity and inform the model that the complexity decreases with the penalty value.

Figure 15.13 shows the results. The best model is the one with the lowest RMSE. The *one-standard-error* rule adds one standard error to the minimum and then moves horizontally to the right until it crosses the curve. The penalty value to the left of the crossing is the selected parameter.

```
penalty_best <- show_best(tune_results, metric = "rmse", n = 1)
tune_results %>%
  collect_metrics() %>%
  filter(.metric == "rmse") %>%
  ggplot(aes(x = penalty, y = mean,
      ymax = mean + std_err, ymin = mean - std_err)) +
  geom_line() +
  geom_errorbar(color = "grey") +
  geom_point() +
  scale_x_log10() +
  coord_cartesian(ylim = c(2, 4)) +
  geom_vline(xintercept = penalty_best$penalty[1], color = "blue") +
  geom_vline(xintercept = penalty_1se$penalty[1], color = "red") +
  geom_hline(yintercept = penalty_best$mean[1] + penalty_best$std_err[1],
    color = "blue")
```

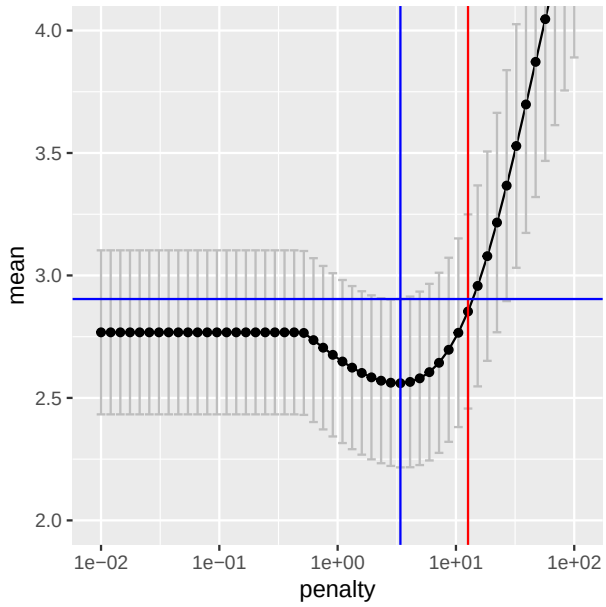


Figure 15.13: Tuning results for ridge regression model to predict mpg with the one-standard-error rule selection of penalty; the *best* penalty value is indicated by the blue line and the *one-standard-error* penalty value is indicated by the red line; the horizontal blue line demonstrates the rule.

The one-standard-error rule can only be applied if it is obvious how model complexity changes with an hyperparameter. In the example above, we used the penalty parameter as a proxy for model complexity. However, this is not always the case. For example, in a random forest model, the number of trees is not necessarily a good proxy for model complexity. In such cases, it is not clear how to apply the one-standard-error rule.

> **i** Further information
>
> - https://dials.tidymodels.org/ `dials` package
> - https://stevenpawley.github.io/colino/ `colino` package

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
knitr::include_graphics("images/model_workflow_tune.png")
```

```r
library(tidymodels)
library(tidyverse)
library(patchwork)
library(future)
plan(multisession, workers = parallel::detectCores(logical = FALSE))
# tidymodels is very picky about data types and will complain when we
# predict on new data if the age value is not an integer. We therefore
# convert here age to double
data <- ISLR2::Wage %>%
  mutate(age = as.double(age))

recipe(wage ~ age, data = data) %>%
  step_poly(hp) %>%
  step_discretize(hp) %>%
  step_cut(hp, breaks = tune()) %>%
  step_bs(hp) %>%
  tunable()
set.seed(123)
poly_recipe <- recipe(wage ~ age, data = data) %>%
  step_poly(age, degree = tune())
model <- linear_reg(mode = "regression") %>%
  set_engine("glm")
poly_workflow <- workflow() %>%
  add_recipe(poly_recipe) %>%
  add_model(model)
tune_results <- tune_grid(poly_workflow, resamples = vfold_cv(data))
tune_results %>% show_best(metric = "rmse")
best_parameters <- select_best(tune_results, metric = "rmse")
final_model <- poly_workflow %>%
  finalize_workflow(best_parameters) %>%
  fit(data)
df <- tibble(age = seq(min(data$age), max(data$age), length.out = 100))
df %>%
  bind_cols(
    predict(final_model, new_data = df),
    predict(final_model, new_data = df, type = "conf_int")
  ) %>%
  ggplot(aes(x = age, y = .pred)) +
  geom_point(aes(x = age, y = wage), data = data, alpha = 0.1) +
  geom_line() +
  geom_ribbon(aes(ymin = .pred_lower, ymax = .pred_upper), alpha = 0.2) +
  labs(x = "Age", y = "Wage")
```

```
set.seed(123)
step_recipe <- recipe(wage ~ age, data = data) %>%
  step_discretize(age, num_breaks = tune())
step_workflow <- workflow() %>%
  add_recipe(step_recipe) %>%
  add_model(model)
tune_results <- tune_grid(step_workflow, resamples = vfold_cv(data))
tune_results %>% show_best(metric = "rmse")
best_breaks <- (tune_results %>% show_best(metric = "rmse"))$num_breaks[1]
best_parameters <- select_best(tune_results, metric = "rmse")
final_model <- step_workflow %>%
  finalize_workflow(best_parameters) %>%
  fit(data)
best_breaks <- (tune_results %>% show_best(metric = "rmse"))$num_breaks[1]
cuts <- tibble(breaks = quantile(data$age,
    probs = seq(0, 1, by = 1 / best_breaks)))
df %>%
  bind_cols(
    predict(final_model, new_data = df),
    predict(final_model, new_data = df, type = "conf_int")
  ) %>%
  ggplot(aes(x = age, y = .pred)) +
  geom_vline(aes(xintercept = breaks), data = cuts,
    color = "darkgreen", alpha = 0.5) +
  geom_point(aes(x = age, y = wage), data = data, alpha = 0.1) +
  geom_line() +
  geom_ribbon(aes(ymin = .pred_lower, ymax = .pred_upper), alpha = 0.2) +
  labs(x = "Age", y = "Wage")
step_recipe <- recipe(wage ~ age, data = data) %>%
  step_cut(age, breaks = seq(20, 70, by = 10), include_outside_range = TRUE)
step_workflow <- workflow() %>%
  add_recipe(step_recipe) %>%
  add_model(model)
trained <- step_workflow %>% fit(data)
cuts <- tibble(breaks = seq(20, 70, by = 10))
df %>%
  bind_cols(
    predict(trained, new_data = df),
    predict(trained, new_data = df, type = "conf_int")
  ) %>%
  ggplot(aes(x = age, y = .pred)) +
  geom_vline(aes(xintercept = breaks), data = cuts,
```

```
    color = "darkgreen", alpha = 0.5) +
  geom_point(aes(x = age, y = wage), data = data, alpha = 0.1) +
  geom_line() +
  geom_ribbon(aes(ymin = .pred_lower, ymax = .pred_upper), alpha = 0.2) +
  labs(x = "Age", y = "Wage")
set.seed(123)
spline_recipe <- recipe(wage ~ age, data = data) %>%
  step_bs(age, degree = tune(), deg_free = tune())
spline_workflow <- workflow() %>%
  add_recipe(spline_recipe) %>%
  add_model(model)
tune_results <- tune_grid(spline_workflow, resamples = vfold_cv(data))
tune_results %>% show_best(metric = "rmse")
best_parameters <- select_best(tune_results, metric = "rmse")
final_model <- spline_workflow %>%
  finalize_workflow(best_parameters) %>%
  fit(data)
df %>%
  bind_cols(
    predict(final_model, new_data = df),
    predict(final_model, new_data = df, type = "conf_int")
  ) %>%
  ggplot(aes(x = age, y = .pred)) +
  geom_point(aes(x = age, y = wage), data = data, alpha = 0.1) +
  geom_line() +
  geom_ribbon(aes(ymin = .pred_lower, ymax = .pred_upper), alpha = 0.2) +
  labs(x = "Age", y = "Wage")
knitr::include_graphics("images/regularization.png")
if (!require(colino)) {
  devtools::install_github("stevenpawley/colino", force = TRUE)
}
library(colino)
set.seed(123)
resamples <- vfold_cv(mtcars)
rec_select <- recipe(mpg ~ ., data = mtcars) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_select_forests(all_predictors(), outcome = "mpg", top_p = tune())
model <- nearest_neighbor(mode = "regression", neighbors = tune())
mtcars_workflow <- workflow() %>%
  add_recipe(rec_select) %>%
  add_model(model)
parameters <- extract_parameter_set_dials(mtcars_workflow)
```

```
parameters %>% knitr::kable()
tune_results <- tune_grid(mtcars_workflow,
  resamples = resamples,
  grid = grid_random(parameters, size = 50))
tune_results %>%
  show_best(metric = "rmse") %>%
  select(-.config) %>%
  knitr::kable()
nfeatures <- (tune_results %>% show_best(metric = "rmse"))$top_p[1]
nneighbors <- (tune_results %>% show_best(metric = "rmse"))$neighbors[1]
best_parameters <- select_best(tune_results, metric = "rmse")
best_workflow <- mtcars_workflow %>%
  finalize_workflow(best_parameters) %>%
  fit(mtcars)
mtcars %>%
  bind_cols(
    predict(best_workflow, new_data = mtcars)
  ) %>%
  ggplot(aes(x = mpg, y = .pred)) +
  geom_point() +
  geom_abline() +
  xlim(10, 35) + ylim(10, 35) +
  labs(x = "Observed mpg", y = "Predicted mpg")
best_workflow %>%
  extract_recipe() %>%
  tidy(number = 2, type = "scores")
feature_scores <- best_workflow %>%
  extract_recipe() %>%
  tidy(number = 2, type = "scores")
kept_features <- feature_scores$variable[1:nfeatures]
kept_features
file <- "https://gedeck.github.io/DS-6030/datasets/UniversalBank.csv.gz"
data <- read_csv(file)
data <- data %>%
  select(-c(ID, `ZIP Code`)) %>%
  rename(
    Personal.Loan = `Personal Loan`,
    Securities.Account = `Securities Account`,
    CD.Account = `CD Account`
  ) %>%
  mutate(
    Personal.Loan = factor(Personal.Loan, labels = c("Yes", "No"),
```

```
      levels = c(1, 0)),
    Education = factor(Education,
      labels = c("Undergrad", "Graduate", "Advanced")),
  )
set.seed(1353)
folds <- vfold_cv(data, strata = Personal.Loan)

formula <- Personal.Loan ~ Age + Experience + Income + Family + CCAvg +
  Education + Mortgage + Securities.Account + CD.Account +
  Online + CreditCard

# Train the logistic regression model
logreg_wf <- workflow() %>%
  add_model(
    logistic_reg() %>% set_engine("glm")
  ) %>%
  add_formula(formula)
logreg_cv <- logreg_wf %>%
  fit_resamples(resamples = folds,
    control = control_resamples(save_pred = TRUE))
logreg_cv_predictions <- collect_predictions(logreg_cv)

# Train the nearest-neighbor model with 5 neighbors
nn5_wf <- workflow() %>%
  add_model(nearest_neighbor(neighbors = 5) %>%
      set_mode("classification") %>%
      set_engine("kknn")) %>%
  add_formula(formula)
nn5_cv <- nn5_wf %>%
  fit_resamples(resamples = folds,
    control = control_resamples(save_pred = TRUE))
nn5_cv_predictions <- collect_predictions(nn5_cv)
# Tune the number of neighbors of the nearest-neighbor model
nn_wf <- workflow() %>%
  add_model(nearest_neighbor(neighbors = tune()) %>%
      set_mode("classification") %>%
      set_engine("kknn")) %>%
  add_formula(formula)
nn_default_tune <- tune_grid(nn_wf, resamples = folds)
autoplot(nn_default_tune)
best_nn_roc <- nn_default_tune %>%
  select_best(metric = "roc_auc")
```

```
best_nn_accuracy <- nn_default_tune %>%
  select_best(metric = "accuracy")
parameters <- extract_parameter_set_dials(nn_wf) %>%
  update(neighbors = neighbors(c(1, 100)))

nn_bayes_tune <- tune_bayes(nn_wf, resamples = folds,
  param_info = parameters, iter = 25)
autoplot(nn_bayes_tune)
optimal_nn_roc <- nn_bayes_tune %>%
  select_best(metric = "roc_auc")
cv_roc <- logreg_cv_predictions %>%
  roc_curve(truth = Personal.Loan, .pred_Yes, event_level = "first")
nn5_roc <- nn5_cv_predictions %>%
  roc_curve(truth = Personal.Loan, .pred_Yes, event_level = "first")

nn_default_auc <- nn_wf %>%
  finalize_workflow(select_best(nn_default_tune, metric = "roc_auc")) %>%
  fit_resamples(resamples = folds,
    control = control_resamples(save_pred = TRUE))
nn_default_auc_roc <- nn_default_auc %>%
  collect_predictions() %>%
  roc_curve(truth = Personal.Loan, .pred_Yes, event_level = "first")

nn_bayes_auc <- nn_wf %>%
  finalize_workflow(select_best(nn_bayes_tune, metric = "roc_auc")) %>%
  fit_resamples(resamples = folds,
    control = control_resamples(save_pred = TRUE))
nn_bayes_auc_roc <- nn_bayes_auc %>%
  collect_predictions() %>%
  roc_curve(truth = Personal.Loan, .pred_Yes, event_level = "first")

g1 <- ggplot() +
  geom_path(data = cv_roc, aes(x = 1 - specificity, y = sensitivity),
    color = "gray") +
  geom_path(data = nn5_roc,
    aes(x = 1 - specificity, y = sensitivity)) +
  geom_abline(lty = 2) +
  labs(title = "(a) Initial model (accuracy, k=5)")

g2 <- ggplot() +
  geom_path(data = cv_roc, aes(x = 1 - specificity, y = sensitivity),
    color = "gray") +
```

```
  geom_path(data = nn_default_auc_roc,
    aes(x = 1 - specificity, y = sensitivity)) +
  geom_abline(lty = 2) +
  labs(title = "(b) Default model (AUC, k=14)")

g3 <- ggplot() +
  geom_path(data = cv_roc, aes(x = 1 - specificity, y = sensitivity),
    color = "gray") +
  geom_path(data = nn_bayes_auc_roc,
    aes(x = 1 - specificity, y = sensitivity)) +
  geom_abline(lty = 2) +
  labs(title = "(c) Optimal model (AUC, k=48)")

g1 + g2 + g3
logreg_metrics <- collect_metrics(logreg_cv)
nn5_metrics <- collect_metrics(nn5_cv)
nn_default_metrics <- collect_metrics(nn_default_auc)
nn_bayes_metrics <- collect_metrics(nn_bayes_auc)

df <- bind_rows(
  logreg_metrics %>% mutate(model = "Logistic regression"),
  nn5_metrics %>% mutate(model = "Nearest neighbor (k=5)"),
  nn_default_metrics %>% mutate(model = "Nearest neighbor (k=14)"),
  nn_bayes_metrics %>% mutate(model = "Nearest neighbor (k=48)"),
)
df %>%
  select(model, mean, .metric) %>%
  pivot_wider(names_from = .metric, values_from = mean) %>%
  knitr::kable(digits = 3) %>%
  kableExtra::kable_styling(full_width = FALSE)
df %>%
  mutate(
    model = factor(model,
      levels = rev(c(
        "Logistic regression",
        "Nearest neighbor (k=5)",
        "Nearest neighbor (k=14)",
        "Nearest neighbor (k=48)"))),
  ) %>%
  ggplot(aes(x = mean, y = model)) +
  geom_point() +
  geom_pointrange(aes(xmin = mean - std_err, xmax = mean + std_err)) +
```

```
  facet_wrap(~ .metric, scales = "free_x")
threshold_graph <- function(model) {
  performance <- probably::threshold_perf(
    model %>% collect_predictions(),
    Personal.Loan, .pred_Yes,
    thresholds = seq(0.05, 0.9, 0.01), event_level = "first",
    metrics = metric_set(accuracy, kap, bal_accuracy, f_meas)
  )
  max_values <- performance %>%
    arrange(desc(.threshold)) %>%
    group_by(.metric) %>%
    filter(.estimate == max(.estimate)) %>%
    filter(row_number() == 1)
  g <- ggplot(performance,
    aes(x = .threshold, y = .estimate, color = .metric)) +
    geom_line() +
    geom_vline(data = max_values,
      aes(xintercept = .threshold, color = .metric)) +
    scale_x_continuous(breaks = seq(0, 1, 0.1)) +
    coord_cartesian(ylim = c(0.5, 1)) +
    xlab("Threshold") + ylab("Metric value") +
    theme(legend.position = "inside",
      legend.position.inside = c(0.85, 0.75))
  return(g)
}
g1 <- threshold_graph(logreg_cv) +
  labs(title = "Logistic regression model")
g2 <- threshold_graph(nn_bayes_auc) +
  labs(title = "k-nearest neighbor model")

g1 + g2
set.seed(123)
resamples <- vfold_cv(mtcars)
rec <- recipe(mpg ~ ., data = mtcars)
spec <- linear_reg(mode = "regression", penalty = tune(), mixture = 0) %>%
  set_engine("glmnet")
wf <- workflow() %>%
  add_recipe(rec) %>%
  add_model(spec)
parameters <- extract_parameter_set_dials(wf) %>%
  update(penalty = penalty(c(-2, 2)))
tune_results <- tune_grid(wf,
```

```
  resamples = resamples,
  grid = grid_regular(parameters, levels = 50))
penalty_best <- select_best(tune_results, metric = "rmse")
penalty_1se <- select_by_one_std_err(tune_results, desc(penalty),
  metric = "rmse")
penalty_best <- show_best(tune_results, metric = "rmse", n = 1)
tune_results %>%
  collect_metrics() %>%
  filter(.metric == "rmse") %>%
  ggplot(aes(x = penalty, y = mean,
      ymax = mean + std_err, ymin = mean - std_err)) +
  geom_line() +
  geom_errorbar(color = "grey") +
  geom_point() +
  scale_x_log10() +
  coord_cartesian(ylim = c(2, 4)) +
  geom_vline(xintercept = penalty_best$penalty[1], color = "blue") +
  geom_vline(xintercept = penalty_1se$penalty[1], color = "red") +
  geom_hline(yintercept = penalty_best$mean[1] + penalty_best$std_err[1],
    color = "blue")
```

16 Stacking models

Boosting or bagging combines a series of models in ensembles to achieve a model performance of the ensemble that is better than the performance of individual models. Prominent examples are `RandomForest` or `xgboost`. In all of these cases, the individual models are of the same type, e.g. decision trees (see Section A.10, Section A.11, Section A.12).

Stacking is a similar concept that goes back to Wolpert (Wolpert 1992) who introduced the idea of stacked generalizations. A set of base models is combined using a *meta-model* that is trained to find the optimal combination of the base models. The models can (and should) be highly divers.

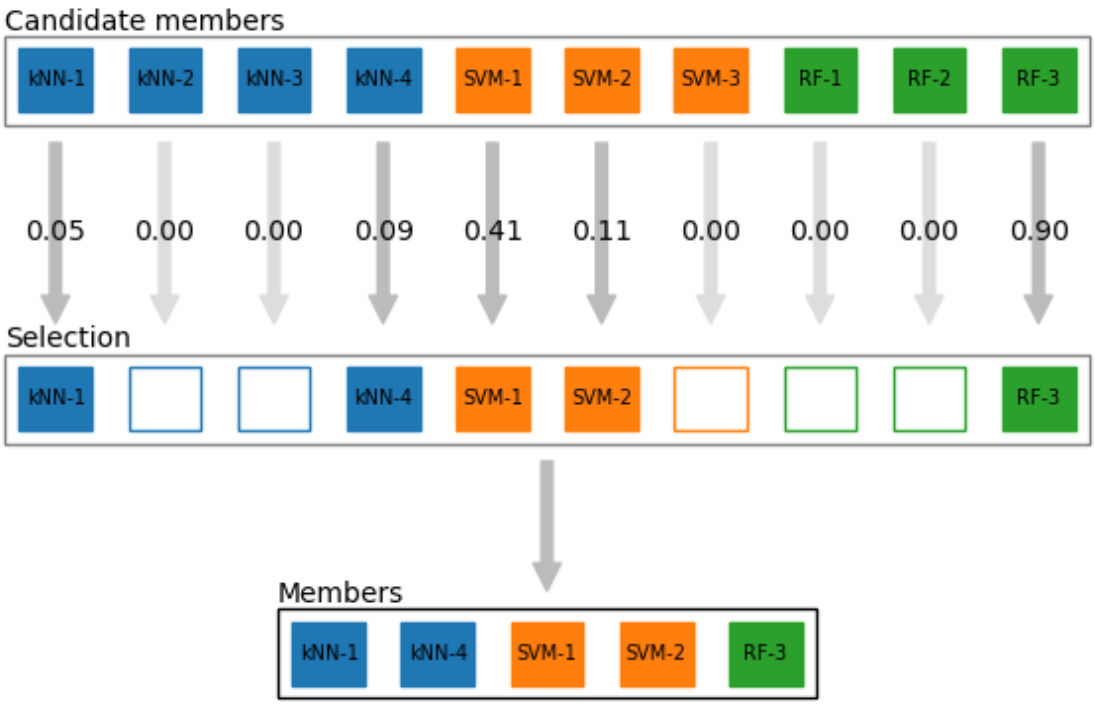


Figure 16.1: Concept of stacking Models

Figure 16.1 shows the concept of stacking. The base models are trained on the training data. The predictions of the base models are then used as input for the meta-model. The meta-model is trained on the training data and the predictions of the base models. The meta-model then combines the predictions of the base models to make the final prediction. Ideally, the meta-model uses some form of constraints that leads to selection of a subset of the candidate members.

There are many ways to implement stacking. In the *tidymodels* ecosystem, stacking is available using the `stacks` package. The `stacks` package uses L1 regularization to select the best combination of base models.

> Todo
>
> Read the articles on https://stacks.tidymodels.org/ to learn how to use the `stacks` package for
>
> - regression and
> - classification.

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
knitr::include_graphics("images/stacking-concept.png")
```

17 Model deployment

Building and training a model is only the first step in the lifecycle of a machine learning application. The next step is to deploy the model into production. This is the process of integrating the model into an existing production environment, so that it can be used to make predictions on new data.

There are many different ways of deploying a model and it very much depends on the infrastructure of your organization.

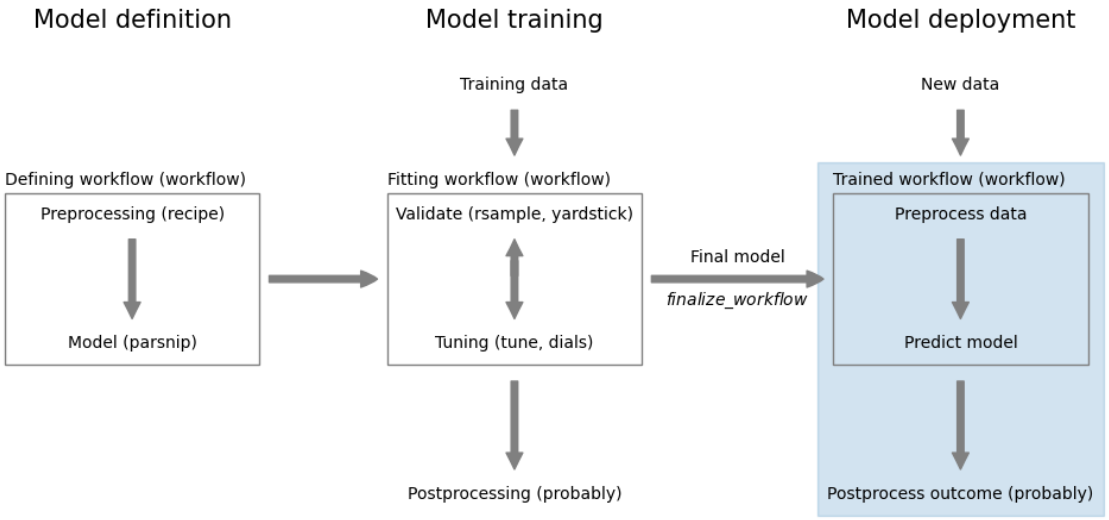


Figure 17.1: Modeling workflow

17.1 Model packaging and infrastructure

Probably the most important aspect of model deployment is to ensure that the model can be easily deployed and maintained in production. Using containers (e.g., Docker) can help to package the model with its dependencies and ensure consistency across environments.

Once you have a containerized model, you can easily deploy it on the most appropriate hardware. Try to use configuration tools like Kubernetes to manage the deployment and scaling of your model. This allows you to easily scale up or down based on demand and ensures that your model is always available to users. It also simplifies the process of updating the model.

An interesting approach to model packaging is to convert the model into the ONNX format (Open Neural Network Exchange). ONNX is an open format for representing machine learning models. It allows you to describe models in a common format that can be deployed on various platforms. You can for example load a model in ONNX format into the browser and access it easily from a web application. While there is basic support for ONNX in R, it is easier to develop the model in Python using `scikit-learn` and then convert it to ONNX format using the `skl2onnx` package.

17.2 Deployment Strategies

Consider if your model needs to provide predictions in real-time or if it can be used in batch mode. Real-time deployment requires the model to be available via an API and to provide predictions with low latency. Batch deployment, on the other hand, allows for more complex models and longer processing times, as predictions are made on a scheduled basis.

17.3 Monitoring and maintenance (post-deployment)

Once the model is deployed, the work is not over. You need to monitor the model's performance in production and maintain it over time. If new data becomes available, you may need to retrain the model to ensure that it continues to perform well.

Aim to continuously record performance metrics and monitor model degradation. The best way to do this is to setup an automated pipeline that retrains the model continuously and deploys the new model whenever it is beneficial.

17.4 R: the `vetiver` package

The `vetiver` package provides basic functionality to

- version and publish models
- deploy models into production
- monitor models in production

The documentation for `vetiver` takes you through the steps of deploying a model into production. You can find the documentation at https://vetiver.rstudio.com/.

To learn more about deploying models, look for resources on MLOps.

Part VI

Unsupervised learning

18 Dimensionality reduction

Load the packages we need for this chapter.

```
library(tidymodels)
library(embed)
library(GGally)
library(ggrepel)
library(kernlab)
library(dimRed)
library(RANN)
```

Load the penguin dataset.

```
penguins <- modeldata::penguins %>% drop_na()
```

18.1 Principal component analysis (PCA)

The `recipes` and `embed` packages have several implementations of principal component analysis (PCA).

- `step_pca`: classical PCA analysis that calculates all principal components
- `step_pca_truncated`: classical PCA analysis that calculates only the requested number of components
- `step_pca_sparse`: in PCA all predictors contribute to the principal components and will have non-zero coefficients; sparse PCA can produce principal components where not all predictors contribute (zero coefficients)

18.1.1 PCA

The `step_pca` (or `step_pca_truncted`) function converts numeric predictors to principal components as part of a recipe. We first define the recipe and then use `prep` and `bake` to get the transformed data.

```
pca_rec <- recipe(data = penguins, formula = ~ .) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_pca(all_numeric_predictors())

penguins_pca <- pca_rec %>%
  prep() %>%
  bake(new_data = NULL)
```

Figure 18.1 shows the first two principal components. We can see that the *gentoo* species is well separated from the other two by the first principal component. A combination of the first and second principal component separates *chinstrap* and *adelie* with some overlap.

```
penguins_pca %>%
  ggplot(aes(x = PC1, y = PC2, color = species, shape = species)) +
  geom_point()
```

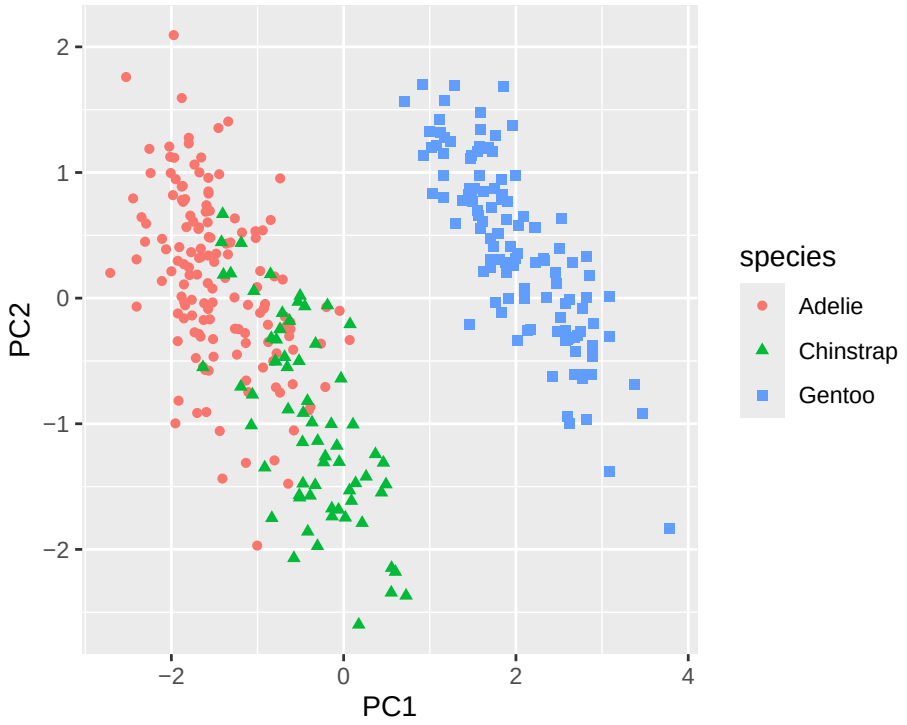


Figure 18.1: First two principal components of the penguin dataset. Points are colored by species

The `step_pca` function labels the principal components by default using `PC1` for the first principal component, `PC2` for the second, and so on; this is a common convention.

A PCA analysis usually also includes looking at the variance of the original dataset explained by the principal components. You have two options to extract this information from the `prep`-ed recipe. The first accesses the fitted PCA model from `prcomp`.

```
prep_pca_rec <- pca_rec %>% prep()

# Access the object from the underlying engine
summary(prep_pca_rec$steps[[2]]$res)
```

```
Importance of components:
                          PC1    PC2     PC3     PC4
Standard deviation     1.6569 0.8821 0.60716 0.32846
Proportion of Variance 0.6863 0.1945 0.09216 0.02697
Cumulative Proportion  0.6863 0.8809 0.97303 1.00000
```

While this is a reasonable approach, you might want to use the actual values for visualizations. In this case, we can use the `tidy` function. Figure 18.2 shows a scree plot created using these data.

```
explained_variance <-  tidy((pca_rec %>% prep())$steps[[2]],
  type = "variance")
perc_variance <- explained_variance %>%
  filter(terms == "percent variance")
cum_perc_variance <- explained_variance %>%
  filter(terms == "cumulative percent variance")

ggplot(explained_variance, aes(x = component, y = value)) +
  geom_bar(data = perc_variance, stat = "identity") +
  geom_line(data = cum_perc_variance) +
  geom_point(data = cum_perc_variance, size = 2) +
  labs(x = "Principal component", y = "Percent variance")
```

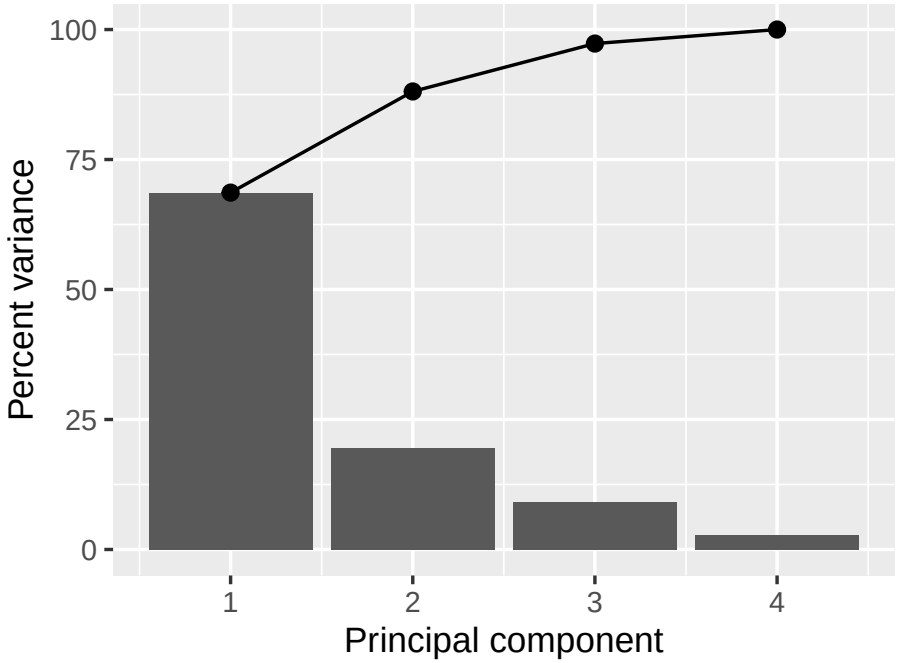


Figure 18.2: Scree plot

Another informative graph is a *biplot*.

```
pca <- pca_rec %>%
  prep()
loadings <-  tidy(pca$steps[[2]], type = "coef") %>%
  pivot_wider(id_cols = "terms", names_from = "component",
    values_from = "value")
scale <- 4
penguins_pca %>%
  ggplot(aes(x = PC1, y = PC2)) +
  geom_point(aes(color = species, shape = species)) +
  geom_segment(data = loadings,
    aes(xend = scale * PC1, yend = scale * PC2, x = 0, y = 0),
    arrow = arrow(length = unit(0.15, "cm"))) +
  geom_label_repel(data = loadings,
    aes(x = scale * PC1, y = scale * PC2, label = terms),
    hjust = "left", size = 2)
```

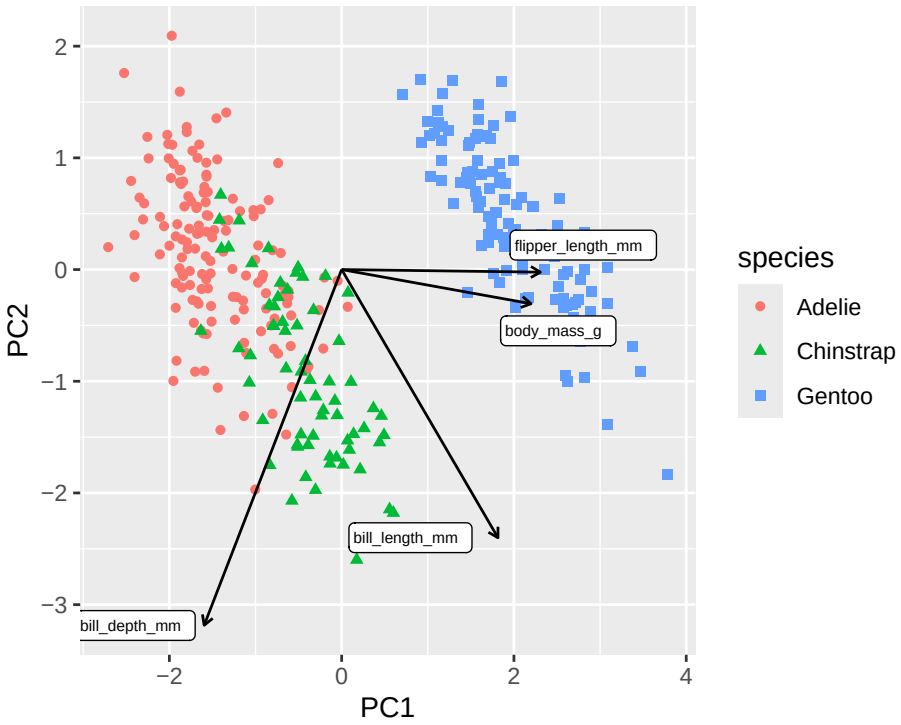


Figure 18.3: Biplot of the penguin dataset. The loadings are shown as arrows and the original variables as labels. For clarity, loadings were multipled by 4

The biplot in Figure 18.3 shows the PCA loadings, the cofficients of the linear combination of the original variables, overlaid onto the scatterplot of the transformed data. The loadings are shown as arrows and tell us that PC1 is mainly a linear combination of flipper length and body mass. PC2 is almost exclusively a linear combination of bill measurements.

18.1.2 Truncated PCA

Truncated PCA can be used if not all the components are needed. The result is identical to the normal PCA (see Figure 18.4). It is particularly useful for larger datasets as the computation will be faster.

```
pca_rec <- recipe(data = penguins, formula = ~ .) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_pca_truncated(all_numeric_predictors(), num_comp = 2)

penguins_pca <- pca_rec %>%
  prep() %>%
  bake(new_data = NULL)
```

```
penguins_pca %>%
  ggplot(aes(x = PC1, y = PC2, color = species, shape = species)) +
  geom_point()
```

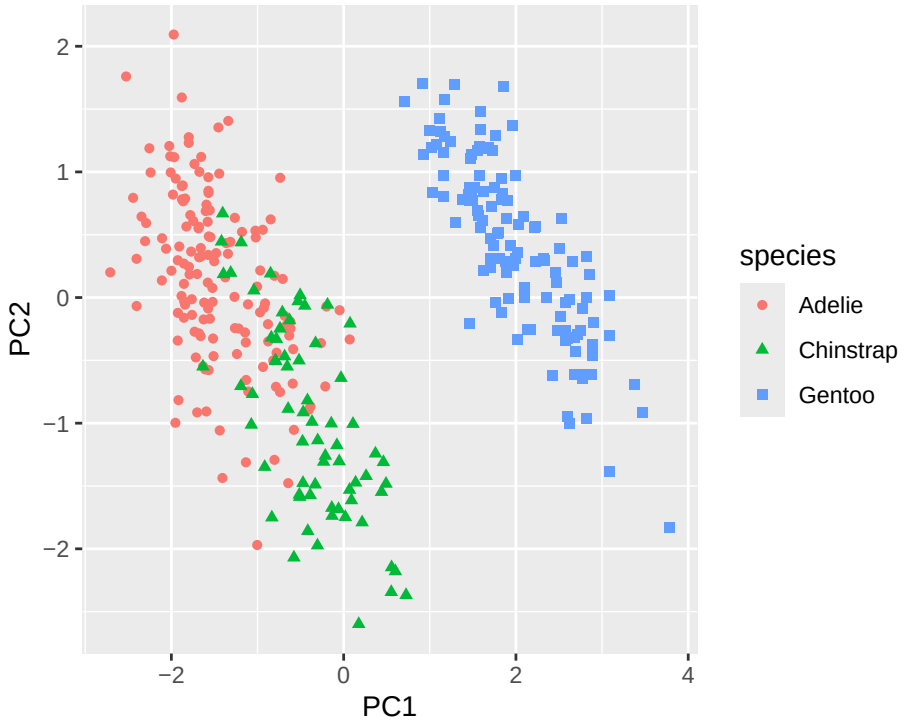


Figure 18.4: First two principal components of the penguin dataset determined using truncated PCA

18.1.3 Sparse principal component analysis (SPCA)

Even though PCA is independent of the dimensionality, high-dimensional datasets lead to principal components where each principal component is a linear combination of all the original variables. This makes it difficult to interpret the principal components. Sparse PCA is a method that creates principal components with sparse loadings. This is similar to what is happening in LASSO regression due to the L1 regularization. In fact, the sparse PCA problem can be reformulated as a LASSO problem (Zou, Hastie, and Tibshirani 2006).

The `step_pca_sparse` function from the `embed` package implements sparse PCA and can be used as a drop-in replacement for `step_pca` (see https://embed.tidymodels.org/reference/step_pca_sparse.html)

18.2 Kernel PCA

Kernel PCA is a non-linear extension of PCA. It uses a kernel function to map the data into a higher-dimensional space where it is linearly separable. The kernel PCA is then performed in this higher-dimensional space. The `recipes` package has a `step_kpca` function that can be used to perform kernel PCA. It's used in the same way as `step_pca`.

```
kpca_rec <- recipe(data = penguins, formula = ~ .) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_kpca(all_numeric_predictors(), num_comp = 2)

penguins_kpca <- kpca_rec %>%
  prep() %>%
  bake(new_data = NULL)

penguins_kpca %>%
  ggplot(aes(x = kPC1, y = kPC2, color = species, shape = species)) +
  geom_point()
```

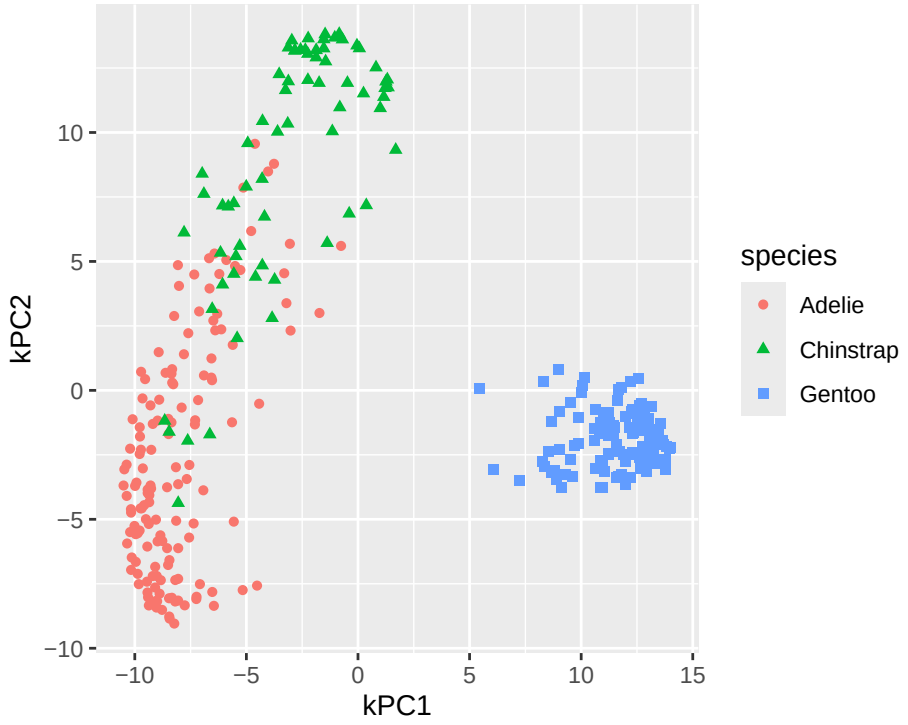


Figure 18.5: First two principal components of the penguin dataset determined using kernel PCA (default settings)

Figure 18.5 shows the resulting projection. In comparison to the linear PCA, the *adelie* and *chinstrap* species are better separated. The *gentoo* species is still well separated from the other two and more compact. To differentiate the kernel PCA result from PCA result, `step_kpca` labels the components using `kPC1`, `kPC2`, and so on. You can change the label prefix using the `prefix` argument.

The default for kernel PCA is to use the radial basis function (RBF) kernel. The `step_kpca` function has several other kernels implemented; see the `kernlab` package for more details. Figure 18.6 shows the resulting projection for kPCA using the polynomial kernel with degree 2. The projection is worse than the RBF kernel.

```
kpca_rec <- recipe(data = penguins, formula = ~ .) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_kpca(all_numeric_predictors(), num_comp = 2,
    options = list(kernel = "polydot", kpar = list(degree = 2)))

penguins_kpca <- kpca_rec %>%
  prep() %>%
  bake(new_data = NULL)

penguins_kpca %>%
  ggplot(aes(x = kPC1, y = kPC2, color = species, shape = species)) +
  geom_point()
```

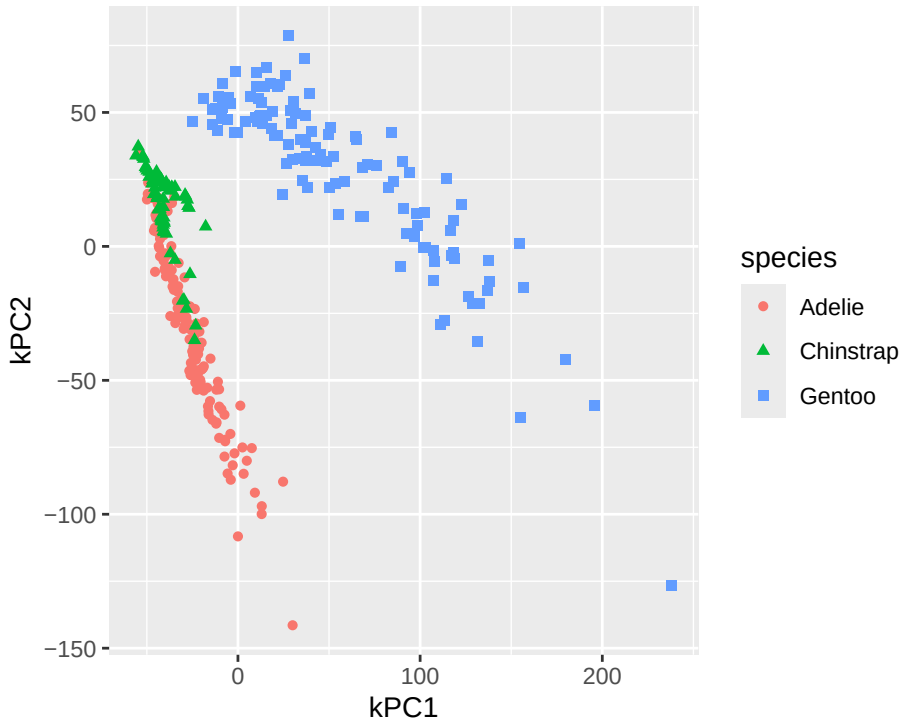


Figure 18.6: First two principal components of the penguin dataset determined using kernel PCA (polynomial kernel with degree 2)

18.3 UMAP

UMAP is a non-linear dimensionality reduction technique that is based on manifold learning. It is similar to t-SNE but is faster and can be used for larger datasets. UMAP also has the advantage over t-SNE that it allows to project new data points into the existing projection.

The `embed` package has a `step_umap` function that can be used to perform UMAP in a `recipe`. The projection creates new variables called `UMAP1`, `UMAP2`, etc. Figure 18.7 shows the resulting projection. The *adelie* and *chinstrap* species are well separated. The *gentoo* species is still well separated from the other two and more compact.

```
umap_rec <- recipe(data = penguins, formula = ~ .) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_umap(all_numeric_predictors(), num_comp = 2)

penguins_umap <- umap_rec %>%
  prep() %>%
  bake(new_data = NULL)
```

```
penguins_umap %>%
  ggplot(aes(x = UMAP1, y = UMAP2, color = species, shape = species)) +
  geom_point()
```

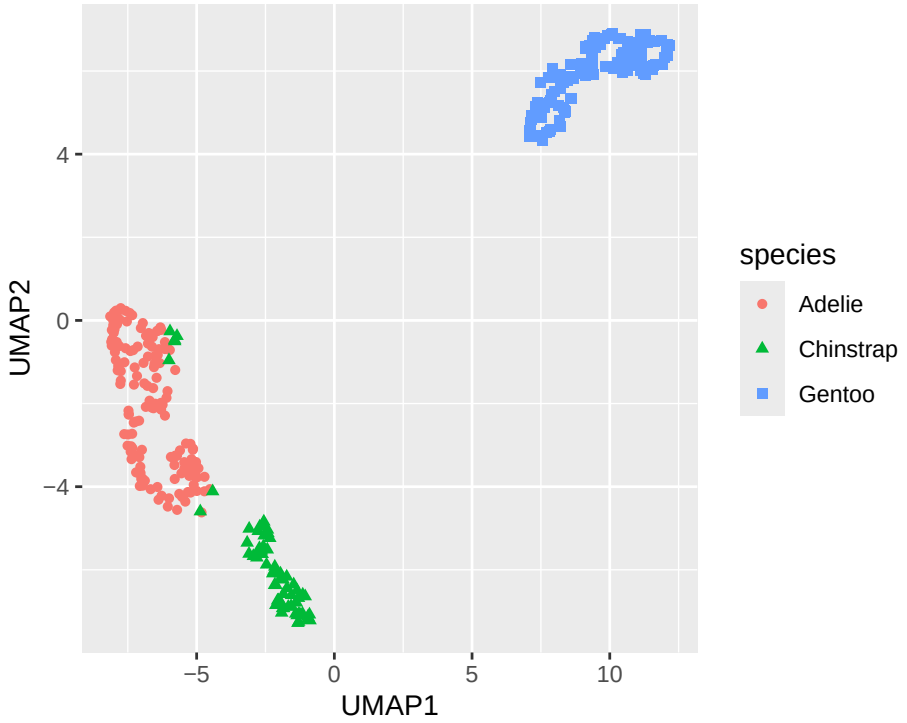


Figure 18.7: UMAP projection of the penguin dataset onto two dimensions (default settings)

UMAP has several parameters that can be tuned. The `min_dist` (default 0.01) parameter controls how tightly the clusters are packed. Increasing the value leads to looser clusters Figure 18.8 shows the projection with `min_dist=0.5`. It is a good idea to explore the effect of the `min_dist` parameter on the projection.

```
umap_rec <- recipe(data = penguins, formula = ~ .) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_umap(all_numeric_predictors(), num_comp = 2,
    min_dist = 0.5)

penguins_umap <- umap_rec %>%
  prep() %>%
  bake(new_data = NULL)
```

```
penguins_umap %>%
  ggplot(aes(x = UMAP1, y = UMAP2, color = species, shape = species)) +
  geom_point()
```

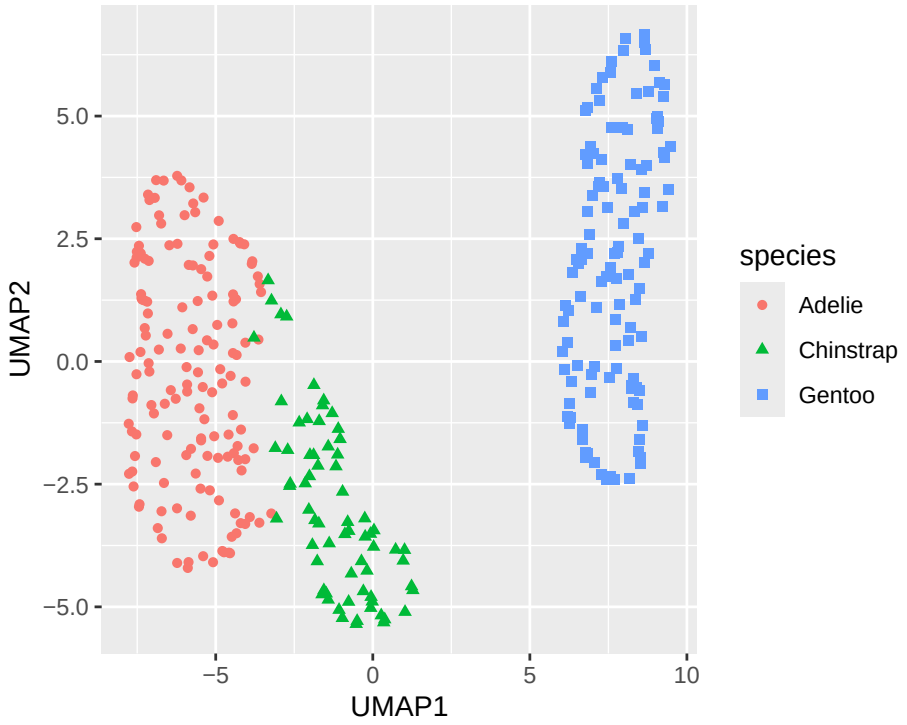


Figure 18.8: UMAP projection of the penguin dataset onto two dimensions (`min_dist=0.5`)

18.4 Isomap (multi-dimensional scaling, MDS)

Multi-dimensional scaling (MDS) is a family of dimensionality reduction techniques that tries to preserve the distances between the data points. Isomap is a version of MDS that uses nearest neighbors information to construct a network of neighbouring points and use this network to define a geodesic distance. which is then used in the mapping

The `recipes` package has a `step_isomap` function that can be used to perform Isomap in a `recipe`. The projection creates new variables called `Isomap1`, `Isomap2`, etc. Figure 18.9 shows the resulting projection.

```
isomap_rec <- recipe(data = penguins, formula = ~ .) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_isomap(all_numeric_predictors(), num_terms = 2, neighbors = 100)
```

```r
penguins_isomap <- isomap_rec %>%
  prep() %>%
  bake(new_data = NULL)
penguins_isomap %>%
  ggplot(aes(x = Isomap1, y = Isomap2, color = species, shape = species)) +
  geom_point()
```

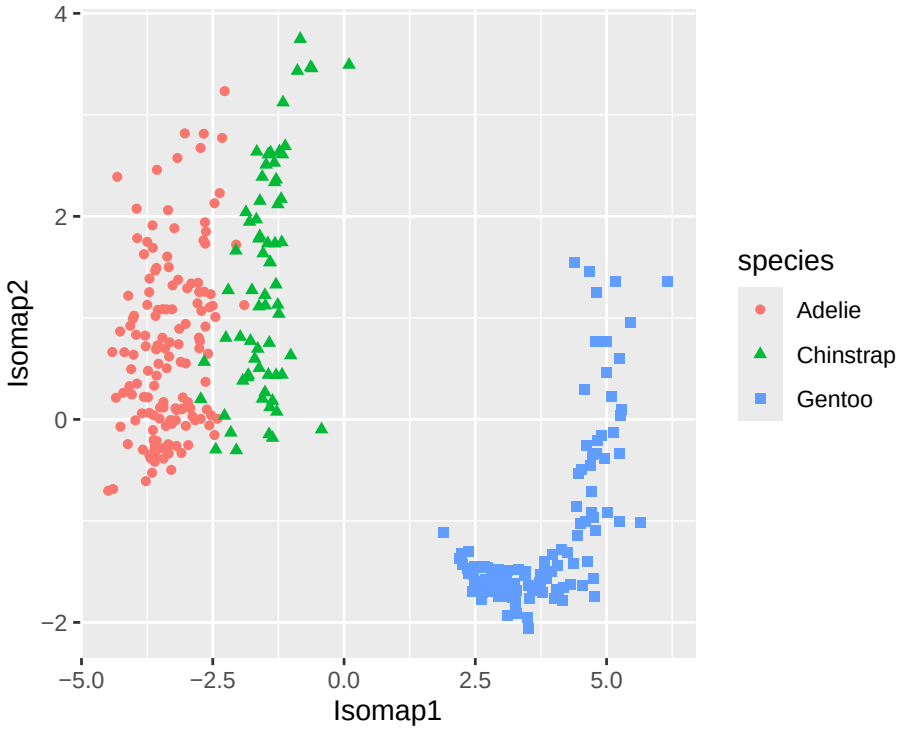


Figure 18.9: Isomap projection of the penguin dataset onto two dimensions (default settings)

18.5 Partial Least Squares (PLS)

Partial Least Squares (PLS) was developed initialy not as a dimensionality reduction technique but as a regression technique. It is similar to PCA but instead of maximizing the variance of the principal components, it maximizes the covariance between the predictors and the response. The `recipes` package has a `step_pls` function that can be used to perform PLS in a `recipe`. The projection creates new variables called `PLS1`, `PLS2`, etc.

```r
data <- datasets::mtcars %>%
  as_tibble(rownames = "car") %>%
  mutate(
    vs = factor(vs, labels = c("V-shaped", "straight")),
    am = factor(am, labels = c("automatic", "manual")),
  )

formula <- mpg ~ cyl + disp + hp + drat + wt + qsec + vs + am +
  gear + carb
pls_rec <- recipe(data = data, formula = formula) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_pls(all_numeric_predictors(), outcome = "mpg", num_comp = 2)

mtcars_pls <- pls_rec %>%
  prep() %>%
  bake(new_data = NULL)
mtcars_pls %>%
  ggplot(aes(x = PLS1, y = PLS2, color = mpg, size = mpg)) +
  geom_point()
```

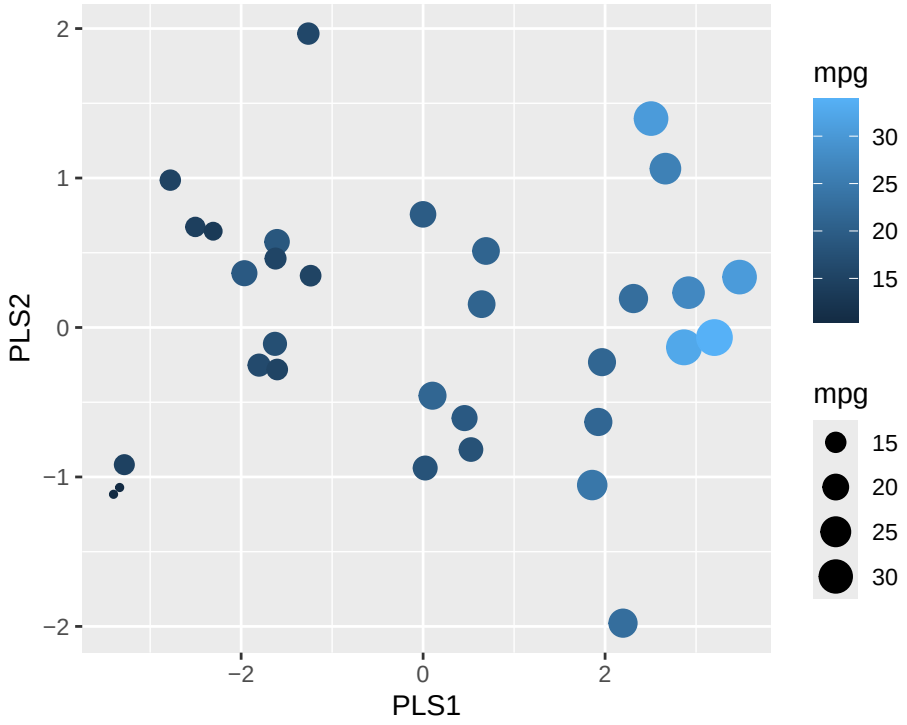


Figure 18.10: PLS projection of the penguin dataset onto two dimensions (default settings)

Figure 18.10 shows the resulting projection. We can see that PLS1 correlates well with the outcome.

> **Further information**
>
> - https://recipes.tidymodels.org/ `recipes` package
> - https://embed.tidymodels.org/ `embed` package

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
library(tidymodels)
library(embed)
library(GGally)
library(ggrepel)
library(kernlab)
library(dimRed)
library(RANN)
penguins <- modeldata::penguins %>% drop_na()
pca_rec <- recipe(data = penguins, formula = ~ .) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_pca(all_numeric_predictors())

penguins_pca <- pca_rec %>%
  prep() %>%
  bake(new_data = NULL)
penguins_pca %>%
  ggplot(aes(x = PC1, y = PC2, color = species, shape = species)) +
  geom_point()
prep_pca_rec <- pca_rec %>% prep()

# Access the object from the underlying engine
summary(prep_pca_rec$steps[[2]]$res)
explained_variance <-  tidy((pca_rec %>% prep())$steps[[2]],
  type = "variance")
perc_variance <- explained_variance %>%
  filter(terms == "percent variance")
cum_perc_variance <- explained_variance %>%
```

```
  filter(terms == "cumulative percent variance")

ggplot(explained_variance, aes(x = component, y = value)) +
  geom_bar(data = perc_variance, stat = "identity") +
  geom_line(data = cum_perc_variance) +
  geom_point(data = cum_perc_variance, size = 2) +
  labs(x = "Principal component", y = "Percent variance")
pca <- pca_rec %>%
  prep()
loadings <-  tidy(pca$steps[[2]], type = "coef") %>%
  pivot_wider(id_cols = "terms", names_from = "component",
    values_from = "value")
scale <- 4
penguins_pca %>%
  ggplot(aes(x = PC1, y = PC2)) +
  geom_point(aes(color = species, shape = species)) +
  geom_segment(data = loadings,
    aes(xend = scale * PC1, yend = scale * PC2, x = 0, y = 0),
    arrow = arrow(length = unit(0.15, "cm"))) +
  geom_label_repel(data = loadings,
    aes(x = scale * PC1, y = scale * PC2, label = terms),
    hjust = "left", size = 2)
pca_rec <- recipe(data = penguins, formula = ~ .) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_pca_truncated(all_numeric_predictors(), num_comp = 2)

penguins_pca <- pca_rec %>%
  prep() %>%
  bake(new_data = NULL)
penguins_pca %>%
  ggplot(aes(x = PC1, y = PC2, color = species, shape = species)) +
  geom_point()
kpca_rec <- recipe(data = penguins, formula = ~ .) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_kpca(all_numeric_predictors(), num_comp = 2)

penguins_kpca <- kpca_rec %>%
  prep() %>%
  bake(new_data = NULL)

penguins_kpca %>%
  ggplot(aes(x = kPC1, y = kPC2, color = species, shape = species)) +
```

```
  geom_point()
kpca_rec <- recipe(data = penguins, formula = ~ .) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_kpca(all_numeric_predictors(), num_comp = 2,
    options = list(kernel = "polydot", kpar = list(degree = 2)))

penguins_kpca <- kpca_rec %>%
  prep() %>%
  bake(new_data = NULL)

penguins_kpca %>%
  ggplot(aes(x = kPC1, y = kPC2, color = species, shape = species)) +
  geom_point()
umap_rec <- recipe(data = penguins, formula = ~ .) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_umap(all_numeric_predictors(), num_comp = 2)

penguins_umap <- umap_rec %>%
  prep() %>%
  bake(new_data = NULL)

penguins_umap %>%
  ggplot(aes(x = UMAP1, y = UMAP2, color = species, shape = species)) +
  geom_point()
umap_rec <- recipe(data = penguins, formula = ~ .) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_umap(all_numeric_predictors(), num_comp = 2,
    min_dist = 0.5)

penguins_umap <- umap_rec %>%
  prep() %>%
  bake(new_data = NULL)

penguins_umap %>%
  ggplot(aes(x = UMAP1, y = UMAP2, color = species, shape = species)) +
  geom_point()
isomap_rec <- recipe(data = penguins, formula = ~ .) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_isomap(all_numeric_predictors(), num_terms = 2, neighbors = 100)

penguins_isomap <- isomap_rec %>%
  prep() %>%
```

```
  bake(new_data = NULL)
penguins_isomap %>%
  ggplot(aes(x = Isomap1, y = Isomap2, color = species, shape = species)) +
  geom_point()
data <- datasets::mtcars %>%
  as_tibble(rownames = "car") %>%
  mutate(
    vs = factor(vs, labels = c("V-shaped", "straight")),
    am = factor(am, labels = c("automatic", "manual")),
  )

formula <- mpg ~ cyl + disp + hp + drat + wt + qsec + vs + am +
  gear + carb
pls_rec <- recipe(data = data, formula = formula) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_pls(all_numeric_predictors(), outcome = "mpg", num_comp = 2)

mtcars_pls <- pls_rec %>%
  prep() %>%
  bake(new_data = NULL)
mtcars_pls %>%
  ggplot(aes(x = PLS1, y = PLS2, color = mpg, size = mpg)) +
  geom_point()
```

19 Clustering

Tidymodels provides a framework for clustering with the `tidyclust` package. It currently supports the following clustering algorithms:

- k-means clustering (`k_means()`)
- hierarchical clustering (`hier_clust()`)

Clustering methods are defined similarly to predictive models in *tidymodels* (`parsnip`). This means each of the methods can use different engines and we can combine we can define clustering with a preprocessing recipe in a workflow.

Load the packages we need for this chapter.

```
library(tidymodels)
library(tidyclust)
library(kableExtra)
library(patchwork)
library(GGally)
library(DT)
```

Because tuning requires training many models, we also enable parallel computing.

```
library(future)
plan(multisession, workers = parallel::detectCores(logical = FALSE))
```

19.1 k-means clustering

The `k_means()` function is a wrapper around four different packages. Here is an example using the default `stats::kmeans` engine to cluster the `penguins` dataset into three clusters.

```
# k-means clustering uses a random starting point,
# so we set a seed for reproducibility
set.seed(123)
penguins <- modeldata::penguins %>% drop_na()
```

```
formula <- ~ bill_length_mm + bill_depth_mm + flipper_length_mm +
  body_mass_g
rec_penguins <- recipe(formula, data = penguins) %>%
  step_normalize(all_predictors())
kmeans_penguins <- k_means(num_clusters = 3) %>%
  set_engine("stats") %>%
  set_mode("partition")
kmeans_wf <- workflow() %>%
  add_recipe(rec_penguins) %>%
  add_model(kmeans_penguins)
```

Note that the preprocessing includes a normalization step. This is recommended so that all predictors have the same scale. The `kmeans_wf` object can be used to fit the model.

```
kmeans_model <- kmeans_wf %>% fit(data = penguins)
```

The `tidy` function gives us a concise overview of the results.

```
tidy(kmeans_model) %>% datatable(rownames = FALSE)
```

`file:////private/var/folders/_8/ms0ft4913k3290v7f0g_yfpc0000gn/T/RtmpAJ4hGD/file150bd510e624(`

Show 10 entries Search:

bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	size	withinss	cluster
-1.045235923738424	0.485894439397612	-0.8803700501844002	-0.7616077652288106	129	120.7030471030117	1
0.6710153036282146	0.8040533794691072	-0.2889117950289312	-0.3835267045110856	85	109.4813179674477	2
0.653774229864347	-1.101049747371146	1.160716294548293	1.099556063848393	119	139.4683666006648	3

Showing 1 to 3 of 3 entries Previous 1 Next

The resulting table has a row for each cluster and contains columns for the cluster centers, the number of observations in each cluster, and the within-cluster sum of squares. The cluster center coordinates are based on normalized data and therefore not directly comparable to the original data.

Useful to know

It is important to set a random seed for reproducibility. The numbering of clusters as well as cluster assignments of data points in rougly equal distance to multiple cluster centers can be different each time you run the code

We can also look at the cluster center coordinates using a *parallel coordinate* plot (see Figure 19.1).

```
tidy(kmeans_model) %>%
  pivot_longer(cols = c("bill_length_mm", "bill_depth_mm",
      "flipper_length_mm", "body_mass_g")) %>%
  ggplot(aes(x = name, y = value,
      group = cluster, color = cluster, shape = cluster)) +
  geom_point(size = 3) +
  geom_line() +
  labs(x = "", y = "Value at cluster center")
```

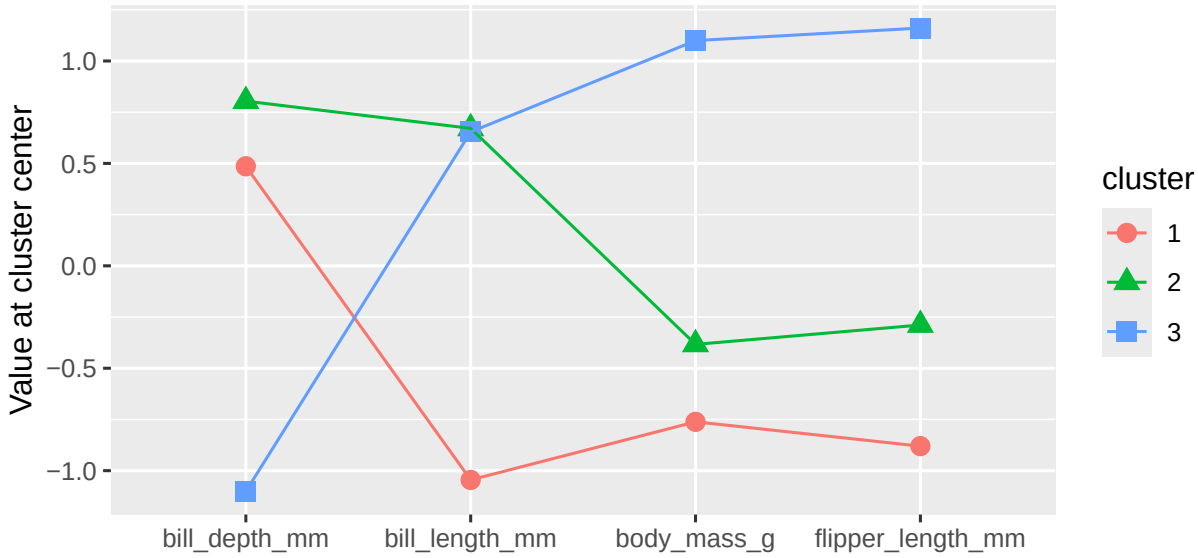


Figure 19.1: Cluster center values for each variable

Clusters 1 and 2 have similar characteristics and differ only with respect to bill length. Cluster 1 represents penguins with smaller bill lengths compared to the penguins in clusters 2 and 3. Cluster 3 is clearly different to the other two clusters and represents penguins with a larger body mass, longer flippers, and smaller bill depth.

We can use the `augment()` function to add the cluster assignments to the original or new data.

```
cl_penguins <- augment(kmeans_model, new_data = penguins)
datatable(cl_penguins %>% head(), rownames = FALSE)
```

```
file:////private/var/folders/_8/ms0ft4913k3290v7f0g_yfpc0000gn/T/RtmpAJ4hGD/file150bd4b61148:
```

Show 10 entries Search:

.pred_cluster	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
Cluster_1	Adelie	Torgersen	39.1	18.7	181	3750	male
Cluster_1	Adelie	Torgersen	39.5	17.4	186	3800	female
Cluster_1	Adelie	Torgersen	40.3	18	195	3250	female
Cluster_1	Adelie	Torgersen	36.7	19.3	193	3450	female
Cluster_1	Adelie	Torgersen	39.3	20.6	190	3650	male
Cluster_1	Adelie	Torgersen	38.9	17.8	181	3625	female

Showing 1 to 6 of 6 entries Previous 1 Next

Figure 19.2 shows a scatterplot matrix (`ggpairs`) of the penguin data with the cluster assignments indicated by color. The clusters are clearly separated in the scatterplot matrix. The `GGally` package provides a `ggpairs()` function that can be used to create such a plot.

```
cl_penguins %>%
  select(-c(species, island, sex)) %>%
  ggpairs(aes(color = .pred_cluster))
```

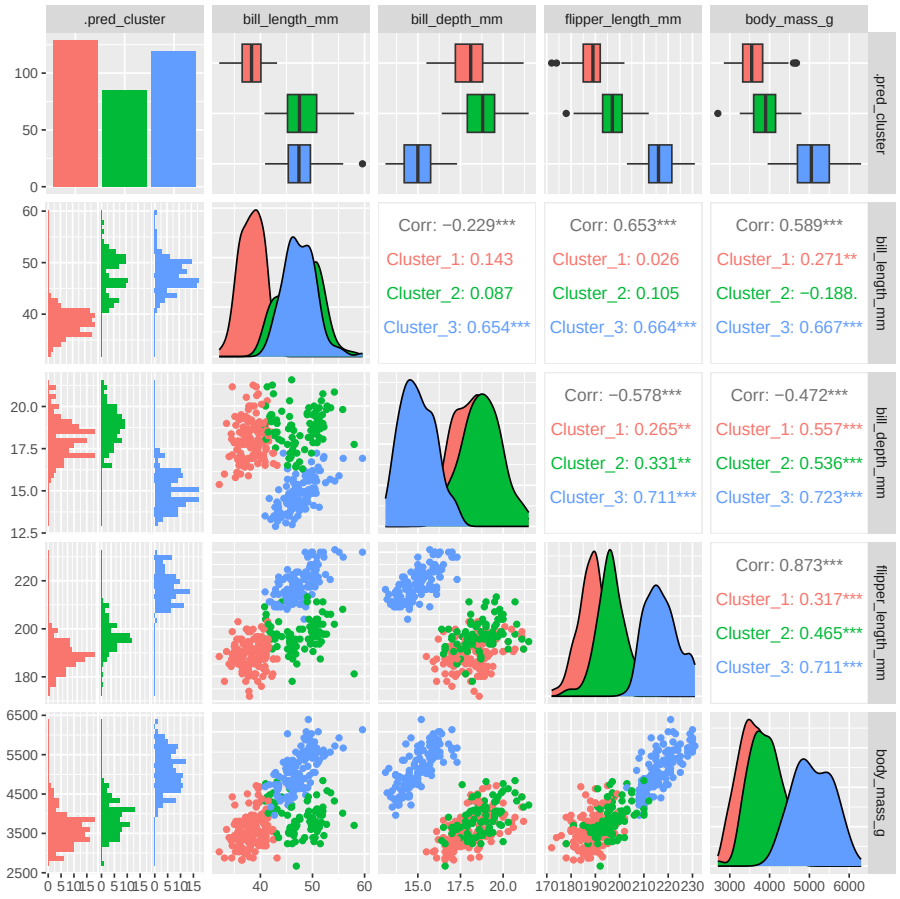


Figure 19.2: k-means clustering of penguins

It is interesting to compare the distribution of the other variables in the clusters.

```
plot_distribution <- function(variable) {
  g <- ggplot(cl_penguins, aes(fill = .data[[variable]],
      x = .pred_cluster)) +
    geom_bar() +
    theme(legend.position = "inside",
      legend.position.inside = c(0.74, 0.83)) +
    scale_y_continuous(limits = c(0, 200)) +
    labs(y = "", x = "")
  return(g)
}
g1 <- plot_distribution("species")
g2 <- plot_distribution("island")
g3 <- plot_distribution("sex")
```

```
g1 + g2 + g3
```

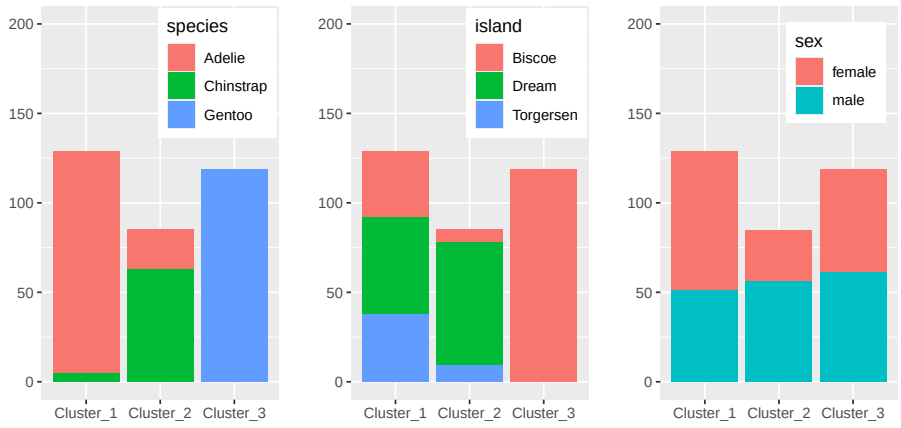


Figure 19.3: Distribution of other variables in the clusters

Figure 19.3 shows that the clusters separate the species well. The clusters also show some discrimination of islands. Cluster 3 contains only penguins from *Biscoe* and cluster 2 mostly penguins from *Dream*. Sex is not well separated by the clusters.

> Useful to know
>
> *k*-Means clustering requires numerical data. If your dataset contains factors, `tidyclust` will automatically convert these to indicator variables.[1]

19.2 Hierarchical clustering

The `tidyclust` package also provides hierarchical clustering. Using the same penguin dataset and the recipe from the previous section we can fit a hierarchical clustering as follows. In hierarchical clustering, points are combined based on distances. It is therefore recommended to normalize the data. At the time of writing, hierarchical clustering did not work as part of a workflow.[2] We therefore first preprocess the data and then perform the hierarchical clustering.

```
formula <- ~ bill_length_mm + bill_depth_mm + flipper_length_mm +
  body_mass_g
```

[1] I couldn't get this to work in a workflow
[2] If you get it to work, let me know

```
rec_penguins <- recipe(formula, data = penguins) %>%
  step_normalize(all_predictors())

norm_penguins <- rec_penguins %>%
  prep() %>%
  bake(new_data = penguins)

hier_penguins <- hier_clust(linkage_method = "complete",
  num_clusters = 3) %>%
  set_engine("stats") %>%
  set_mode("partition")

hier_model <- hier_penguins %>% fit(formula, data = norm_penguins)
```

We specify the number of clusters (`num_clusters`) in the `hier_clust` function. An alternative would be to define the height at which to cut the dendrogram using `cut_height`.

Currently, `tidyclust` only provides a wrapper around the `stats::hclust` function. The resulting clustering can be visualized by accessing the underlying `hier_model$fit` object.

```
hier_model$fit %>% plot()
```

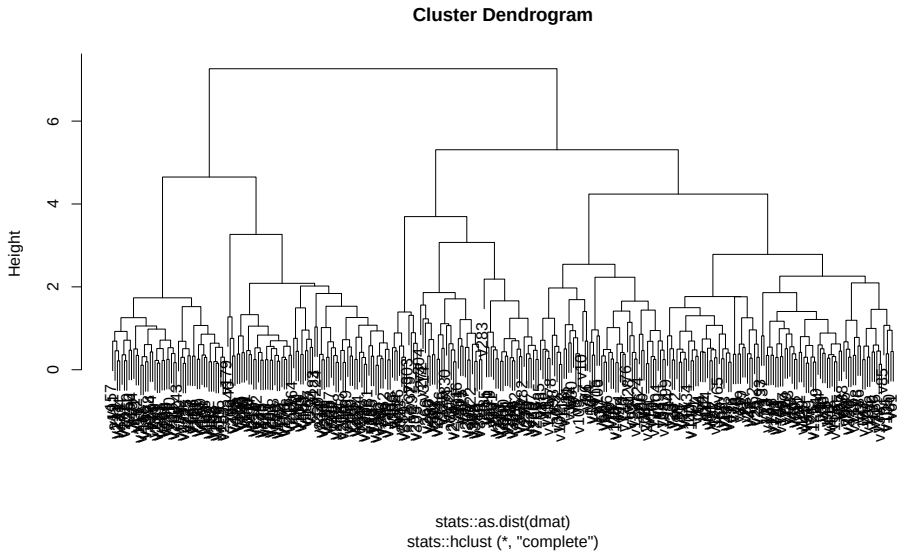


Figure 19.4: Cluster dendrogram for a hierarchical clustering of the penguins dataset using complete linkage

Figure 19.4 shows the dendrogram of the complete linkage clustering.

As we specified number of clusters, we can extract information about the resulting cluster.

```
cluster_assignment <- hier_model %>% extract_cluster_assignment()
centroids <- hier_model %>% extract_centroids()
```

```
centroids %>%
  pivot_longer(cols = c("bill_length_mm", "bill_depth_mm",
      "flipper_length_mm", "body_mass_g")) %>%
  ggplot(aes(x = name, y = value,
      group = .cluster, color = .cluster, shape = .cluster)) +
  geom_point(size = 3) +
  geom_line() +
  labs(x = "", y = "Value at cluster center")
```

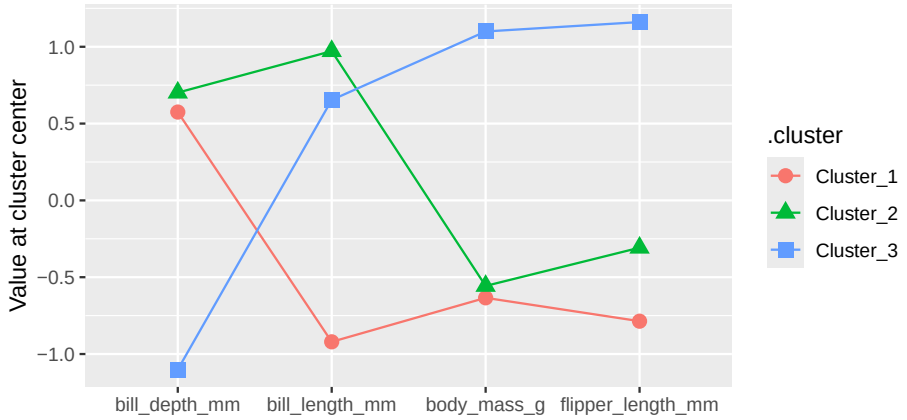


Figure 19.5: Cluster center values for each variable from hierarchical clustering

Figure 19.5 visualizes the co-ordinates of the cluster centroids. The results are comparable to the k-means clustering shown in Figure 19.1.

The resulting hiearchical clustering can also predict new data. It classifies a data point by distance to the nearest cluster centroid. In contrast to k-means clustering, you can specify the number of clusters or the cut height in the predict function.

```
pred_class <- hier_model %>%
  predict(new_data = norm_penguins, num_clusters = 4)
```

We can visualize the resulting cluster assignments in a pairs plot.

```
bind_cols(
  penguins,
  hier_model %>%
    predict(new_data = norm_penguins, num_clusters = 4),
) %>%
  select(-c(species, island,  sex)) %>%
  ggpairs(aes(color = .pred_cluster))
```

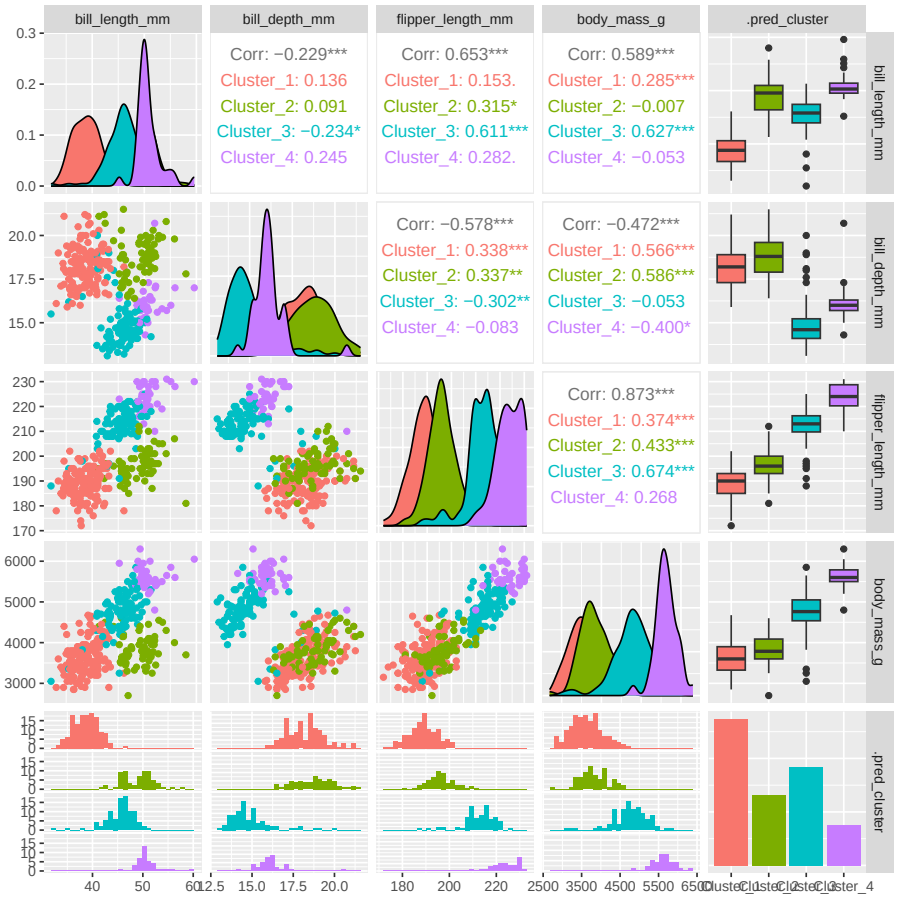


Figure 19.6: Hierarchical clustering of penguins

As can be seen in Figure 19.6 the additional split leads to the formation of the blue and purple clusters (compare to k-means Figure 19.2 for three clusters).

19.3 Determine the number of clusters

```
set.seed(123)
penguins <- modeldata::penguins %>%
  drop_na()

formula <- ~ bill_length_mm + bill_depth_mm + flipper_length_mm +
  body_mass_g
rec_penguins <- recipe(formula, data = penguins) %>%
  step_normalize(all_predictors())
kmeans_penguins <- k_means(num_clusters = tune()) %>%
  set_engine("stats") %>%
  set_mode("partition")
kmeans_wf <- workflow() %>%
  add_recipe(rec_penguins) %>%
  add_model(kmeans_penguins)
```

In order to tune the number of clusters, we set `num_clusters=tune()`. We can now run the `tune_cluster` function using a grid search over different values with cross-validation.

```
set.seed(4400)
folds <- vfold_cv(penguins, v = 2)
grid <- tibble(num_clusters = 1:10)
result <- tune_cluster(kmeans_wf, resamples = folds, grid = grid,
  metrics = cluster_metric_set(sse_within_total, silhouette_avg))
```

We can use the `collect_metrics` function to retrieve the cluster metrics for different cluster numbers. By default, `tidyclust` computes the total sum of squares (`sse_total`) and the within-cluster SSE (`sse_within_total`).

```
collect_metrics(result) %>% head()
```

```
# A tibble: 6 x 7
  num_clusters .metric          .estimator    mean     n std_err .config
         <int> <chr>            <chr>        <dbl> <int>   <dbl> <chr>
1            1 silhouette_avg   standard   NaN         0 NA      Preprocessor1_~
2            1 sse_within_total standard   662         2  2.00   Preprocessor1_~
3            2 silhouette_avg   standard     0.529     2  0.0105 Preprocessor1_~
4            2 sse_within_total standard   278.        2 18.4    Preprocessor1_~
5            3 silhouette_avg   standard     0.461     2  0.0188 Preprocessor1_~
6            3 sse_within_total standard   182.        2  1.65   Preprocessor1_~
```

`tune_cluster` also supports the `autoplot` function to visualize the variation of cluster metrics as a function of the tuning parameter.

```
autoplot(result)
```

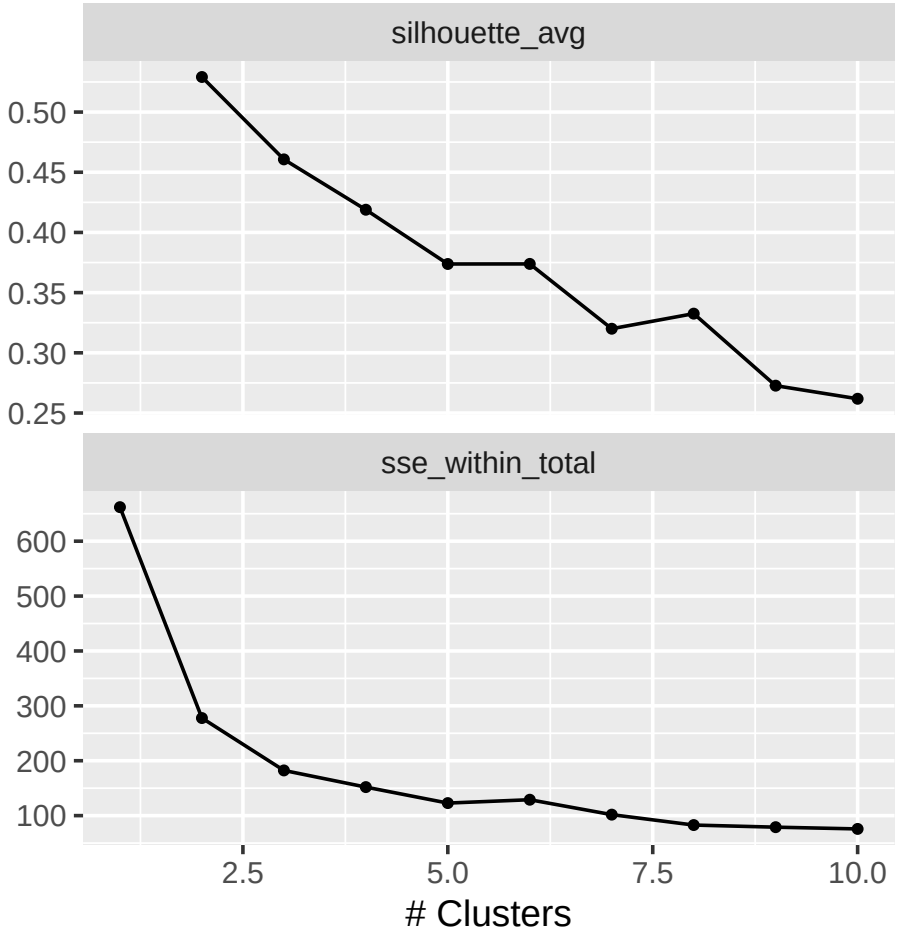


Figure 19.7: Cluster metrics as a function of number of clusters

The metrics curves are interpreted as follows to determine the *optimal* number of clusters. For the `sse_within_total` curve, we look for an *ellbow*, a point where the slope of the curve changes visibly. In our case, the ellbow is either at 2 or at 3. In the `silhouette_avg` curve, the *optimal* cluster number corresponds to the maximum of the curve, here 2.

Useful to know

To identify the *ellbow*, sometimes also called the *knee*, is highly subjective. Consider the graph in Figure 19.7 again.

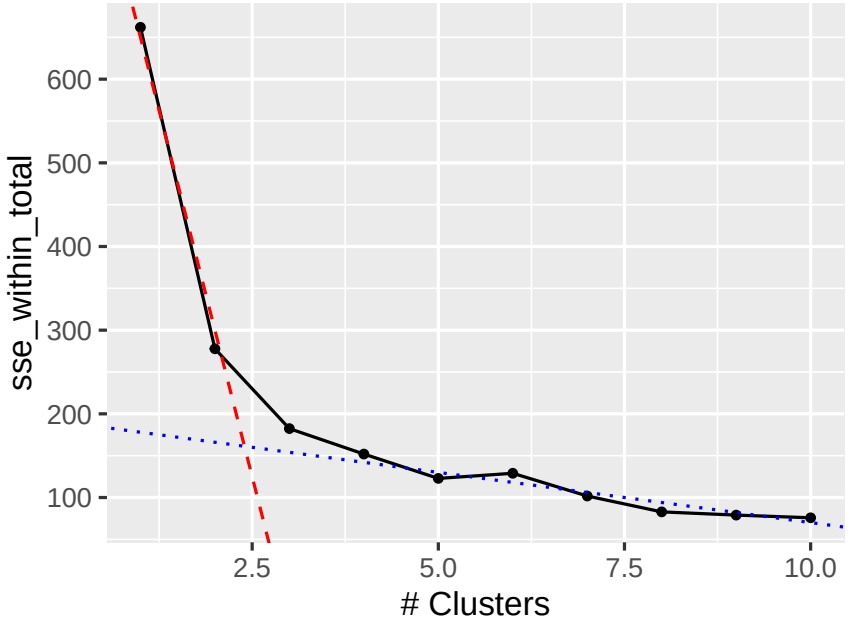


Figure 19.8: Using the ellbow method to select the number of clusters

In Figure 19.8, we added two lines manually. The red, dashed line highlights the initial decline in `sse_within_total` and the blue, dotted line the later, less rapid change. The two lines cross around 2.5 (the *ellbow*) which tells us that we should use 2 or 3 clusters.

While approaches like the *ellbow* method or metrics like the `silhouette_avg` can act as a guideline, the decision of how many clusters to keep is somewhat subjective and often depends more on the use case.

Once we have made a decision on the number of clusters, we can train a finalized model using the `finalize_model_tidyclust` or `finalize_workflow_tidyclust` methods.

```
best_params <- data.frame(num_clusters = 3)
model <- finalize_workflow_tidyclust(kmeans_wf, best_params)
model
```

```
== Workflow ====================================================================
Preprocessor: Recipe
Model: k_means()
```

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

* step_normalize()

-- Model ------------------------------------------------------------------------
K Means Cluster Specification (partition)

Main Arguments:
  num_clusters = 3

Computational engine: stats
```

> **i** Further information
>
> - https://tidyclust.tidymodels.org/ `tidyclust` package

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
library(tidymodels)
library(tidyclust)
library(kableExtra)
library(patchwork)
library(GGally)
library(DT)
library(future)
plan(multisession, workers = parallel::detectCores(logical = FALSE))
# k-means clustering uses a random starting point,
# so we set a seed for reproducibility
set.seed(123)
penguins <- modeldata::penguins %>% drop_na()

formula <- ~ bill_length_mm + bill_depth_mm + flipper_length_mm +
  body_mass_g
rec_penguins <- recipe(formula, data = penguins) %>%
```

```
  step_normalize(all_predictors())
kmeans_penguins <- k_means(num_clusters = 3) %>%
  set_engine("stats") %>%
  set_mode("partition")
kmeans_wf <- workflow() %>%
  add_recipe(rec_penguins) %>%
  add_model(kmeans_penguins)
kmeans_model <- kmeans_wf %>% fit(data = penguins)
tidy(kmeans_model) %>% datatable(rownames = FALSE)
tidy(kmeans_model) %>%
  pivot_longer(cols = c("bill_length_mm", "bill_depth_mm",
      "flipper_length_mm", "body_mass_g")) %>%
  ggplot(aes(x = name, y = value,
      group = cluster, color = cluster, shape = cluster)) +
  geom_point(size = 3) +
  geom_line() +
  labs(x = "", y = "Value at cluster center")
cl_penguins <- augment(kmeans_model, new_data = penguins)
datatable(cl_penguins %>% head(), rownames = FALSE)
cl_penguins %>%
  select(-c(species, island, sex)) %>%
  ggpairs(aes(color = .pred_cluster))
plot_distribution <- function(variable) {
  g <- ggplot(cl_penguins, aes(fill = .data[[variable]],
      x = .pred_cluster)) +
    geom_bar() +
    theme(legend.position = "inside",
      legend.position.inside = c(0.74, 0.83)) +
    scale_y_continuous(limits = c(0, 200)) +
    labs(y = "", x = "")
  return(g)
}
g1 <- plot_distribution("species")
g2 <- plot_distribution("island")
g3 <- plot_distribution("sex")

g1 + g2 + g3
formula <- ~ bill_length_mm + bill_depth_mm + flipper_length_mm +
  body_mass_g
rec_penguins <- recipe(formula, data = penguins) %>%
  step_normalize(all_predictors())
```

```
norm_penguins <- rec_penguins %>%
  prep() %>%
  bake(new_data = penguins)

hier_penguins <- hier_clust(linkage_method = "complete",
  num_clusters = 3) %>%
  set_engine("stats") %>%
  set_mode("partition")

hier_model <- hier_penguins %>% fit(formula, data = norm_penguins)
hier_model$fit %>% plot()
cluster_assignment <- hier_model %>% extract_cluster_assignment()
centroids <- hier_model %>% extract_centroids()
centroids %>%
  pivot_longer(cols = c("bill_length_mm", "bill_depth_mm",
      "flipper_length_mm", "body_mass_g")) %>%
  ggplot(aes(x = name, y = value,
      group = .cluster, color = .cluster, shape = .cluster)) +
  geom_point(size = 3) +
  geom_line() +
  labs(x = "", y = "Value at cluster center")
pred_class <- hier_model %>%
  predict(new_data = norm_penguins, num_clusters = 4)
bind_cols(
  penguins,
  hier_model %>%
    predict(new_data = norm_penguins, num_clusters = 4),
) %>%
  select(-c(species, island,  sex)) %>%
  ggpairs(aes(color = .pred_cluster))
set.seed(123)
penguins <- modeldata::penguins %>%
  drop_na()

formula <- ~ bill_length_mm + bill_depth_mm + flipper_length_mm +
  body_mass_g
rec_penguins <- recipe(formula, data = penguins) %>%
  step_normalize(all_predictors())
kmeans_penguins <- k_means(num_clusters = tune()) %>%
  set_engine("stats") %>%
  set_mode("partition")
kmeans_wf <- workflow() %>%
```

```
  add_recipe(rec_penguins) %>%
  add_model(kmeans_penguins)
set.seed(4400)
folds <- vfold_cv(penguins, v = 2)
grid <- tibble(num_clusters = 1:10)
result <- tune_cluster(kmeans_wf, resamples = folds, grid = grid,
  metrics = cluster_metric_set(sse_within_total, silhouette_avg))
collect_metrics(result) %>% head()
autoplot(result)
autoplot(result, metric = "sse_within_total") +
  geom_abline(intercept = 1000, slope = -350, linetype = "dashed",
    color = "red") +
  geom_abline(intercept = 190, slope = -12, linetype = "dotted",
    color = "blue")
best_params <- data.frame(num_clusters = 3)
model <- finalize_workflow_tidyclust(kmeans_wf, best_params)
model
```

Part VII

Model deep dives

20 Linear regression models

The DS-6030 course primarily focuses on predictive modeling. However, it is useful to know how to access the model parameters and interpret them for linear regression models. In this section, we will use the *tidymodels* framework to build a linear regression model and then look at the model parameters and diagnostics.

Load required libraries

```
library(tidyverse)
library(tidymodels)
library(kableExtra)
library(ggfortify)  # required for diagnostics plots
library(DT)
```

20.1 Build a linear regression model

We will use the example from Chapter 8 here.

```
data <- datasets::mtcars %>%
  as_tibble(rownames = "car") %>%
  mutate(
    vs = factor(vs, labels = c("V-shaped", "straight")),
    am = factor(am, labels = c("automatic", "manual")),
  )
formula <- mpg ~ cyl + disp + hp + drat + wt + qsec + vs + am +
  gear + carb
model <- linear_reg(engine = "lm", mode = "regression") %>%
  fit(formula, data = data)
```

20.2 Analyze model parameters

The `tidyverse` framework introduced the function `tidy` which is a generic function similar to `plot` or `predict`. This means, there are implementations for objects of different types. The

Table 20.1: Linear regression model parameters extracted using the tidy function

Show 10 entries Search:

term	estimate	std.error	statistic	p.value
(Intercept)	12.303	18.718	0.657	0.518
cyl	-0.111	1.045	-0.107	0.916
disp	0.013	0.018	0.747	0.463
hp	-0.021	0.022	-0.987	0.335
drat	0.787	1.635	0.481	0.635
wt	-3.715	1.894	-1.961	0.063
qsec	0.821	0.731	1.123	0.274
vsstraight	0.318	2.105	0.151	0.881
ammanual	2.520	2.057	1.225	0.234
gear	0.655	1.493	0.439	0.665

Showing 1 to 10 of 11 entries Previous 1 2 Next

`parsnip` package also provides an implementations of the `tidy` function that, works for fitted models. Using it with our linear regression model, extracts the model parameters. Table 20.1 shows the model parameters extracted with the `tidy` function.

```
model %>%
  tidy() %>%
  datatable(rownames = FALSE) %>%
  formatRound(
    columns = c("estimate", "std.error", "statistic", "p.value"),
    digits = 3
  )
```

file:////private/var/folders/_8/ms0ft4913k3290v7f0g_yfpc0000gn/T/RtmpgH01WX/file170375743b2a

Table 20.2: Linear regression model statistics extracted using the glance function

Show 10 entries Search:

r.squared	adj.r.squared	sigma	statistic	p.value	df	logLik	AIC	BIC	deviance	df.residual	nobs
0.869	0.807	2.650	13.932	0.000	10.000	-69.855	163.710	181.299	147.494	21.000	32.000

Showing 1 to 1 of 1 entries Previous 1 Next

Table 20.3: Plots created by 'which' argument

which	Plot
1	Residuals vs Fitted
2	Normal Q-Q
3	Scale-Location
4	Cook's distance
5	Residuals vs Leverage
6	Cooks's distance vs Leverage

20.3 Extract model statistics

The `broom` package provides the function `glance` to extract model statistics from a fitted linear regression model. Table 20.2 shows the model statistics for our case.

```
model %>%
  glance() %>%
  datatable(rownames = FALSE) %>%
  formatRound(columns = names(model %>% glance()), digits = 3)
```

```
file:////private/var/folders/_8/ms0ft4913k3290v7f0g_yfpc0000gn/T/RtmpgH01WX/file1703718c33d2/
```

20.4 Diagnostics plots

The `ggfortify` package provides implementations of the generic `autoplot` function that creates up to six diagnostics plots using `ggplot`. The plots are selected using the `which` argument. The default creates four plots (1, 2, 3, and 5).

20.4.1 Residuals vs Fitted

The argument `which=1` creates a graph of residuals vs fitted (see Figure 20.1). This type of plot is useful for all types of regression, not only linear regressions models.

```
autoplot(model, which = c(1, 2))
```

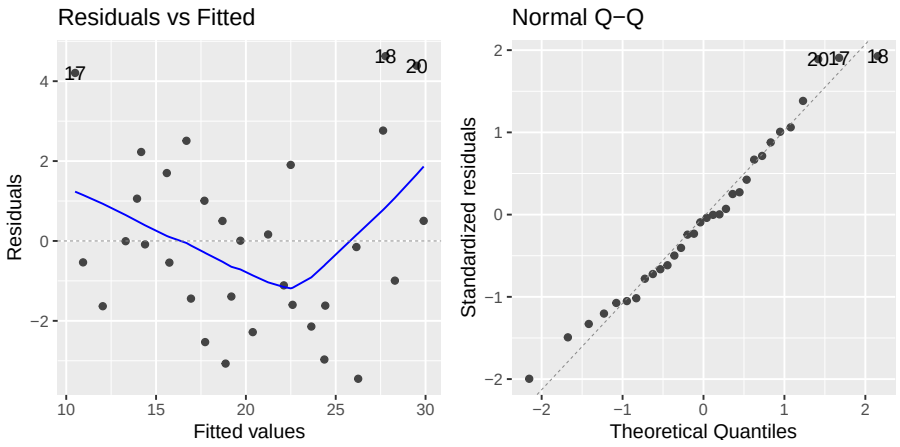


Figure 20.1: Diagnostics plots: residuals vs fitted (`which=1`) and Normal Q-Q plot (`which=2`)

The blue line is a smoother of the residuals. The line should be a flat line and show no trend in the data. Looking at the plot, we can see that there is some non-linearity in the residuals. This could be an indication to explore variable transformations or polynomial terms.

The plot can also reveal heteroscedasticity of the residuals. It is however more qualitative; the scale-location plot can show this better.

20.4.2 Normal Q-Q plot

The normal Q-Q plot looks at the distribution of the residuals. The normal Q-Q plot in Figure 20.1 shows that in our case, residuals are normally distributed.

20.4.3 Scale-location plot

```
autoplot(model, which = c(3, 4))
```

Figure 20.2: Diagnostics plots: scale location plot (`which=3`) and Cook's distance plot (`which=4`)

The scale location plot can reveal changes in variance of the residuals easier. Figure 20.2, shows that in our case, the variation doesn't change much.

20.4.4 Cook's distance plot

The Cook's distance measures the influence of a data point on the regression. Figure 20.2, shows that two data points stand out (9 and 29).

Some suggest that points with a Cook's distance greater than 1 should be looked at. None of our data points are flagged in this case. Others suggest $4/(n-p-1)$ as a threshold. For our model, $n = 32$ and $p = 10$, so $4/(n-p-1) = 4/(32-10-1) = 0.19$. This will flag 9 and 29 as points of interest.

20.4.5 Residuals vs Leverage

```
autoplot(model, which = c(5, 6))
```

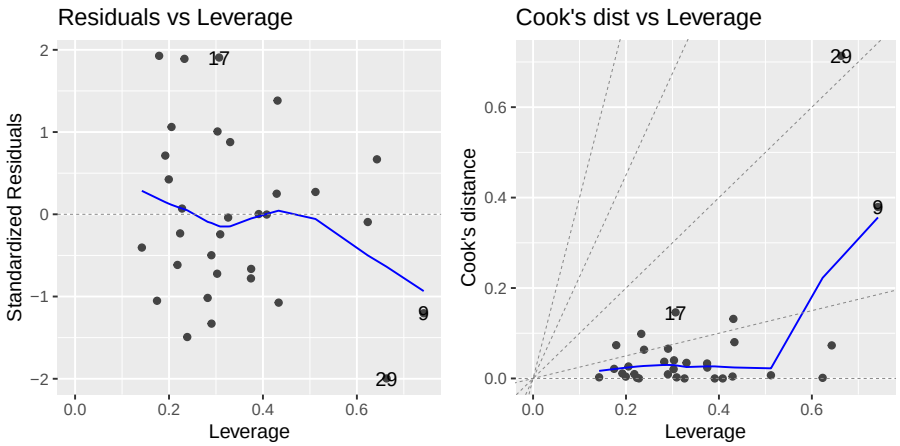


Figure 20.3: Diagnostics plots: residuals vs leverage (`which=5`) and Cook's distance vs leverage (`which=6`)

Leverage is another measure of how influential a data point is. The larger the leverage, the more the data point affects the regression. In our case (see Figure 20.3), no point stands out.

20.4.6 Cooks's distance vs Leverage

The final plot shows Cook's distance vs leverage (see Figure 20.3). Again, look for points with large Cook's distance or large leverage.

> **Further information**
>
> Additional information can be found in the following resources:
>
> - https://bookdown.org/dereksonderegger/571/7-Diagnostics-Chapter.html
> - Comprehensive slide deck by John Fox, the author of the book "Regression Diagnostics" https://socialsciences.mcmaster.ca/jfox/Courses/Brazil-2009/slides-handout.pdf

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
library(tidyverse)
```

```r
library(tidymodels)
library(kableExtra)
library(ggfortify)  # required for diagnostics plots
library(DT)
data <- datasets::mtcars %>%
  as_tibble(rownames = "car") %>%
  mutate(
    vs = factor(vs, labels = c("V-shaped", "straight")),
    am = factor(am, labels = c("automatic", "manual")),
  )
formula <- mpg ~ cyl + disp + hp + drat + wt + qsec + vs + am +
  gear + carb
model <- linear_reg(engine = "lm", mode = "regression") %>%
  fit(formula, data = data)
model %>%
  tidy() %>%
  datatable(rownames = FALSE) %>%
  formatRound(
    columns = c("estimate", "std.error", "statistic", "p.value"),
    digits = 3
  )
model %>%
  glance() %>%
  datatable(rownames = FALSE) %>%
  formatRound(columns = names(model %>% glance()), digits = 3)
tibble(
  which = c(1, 2, 3, 4, 5, 6),
  Plot = c("Residuals vs Fitted", "Normal Q-Q",
    "Scale-Location", "Cook's distance",
    "Residuals vs Leverage", "Cooks's distance vs Leverage")
) %>%
  kableExtra::kbl(caption = "Plots created by `which` argument") %>%
  kableExtra::kable_styling(full_width = FALSE)
autoplot(model, which = c(1, 2))
autoplot(model, which = c(3, 4))
autoplot(model, which = c(5, 6))
```

21 Regularized Generalized linear models (`glmnet`)

In linear regression, the outcome is a linear function of the predictor variables.

$$y = y_0 + c_1 x_1 + c_2 x_2 + c_3 x_3 + ...$$

where c_i are the coefficients and y_0 is the intercept.

Generalized linear models (GLMs) extend this idea by adding a link function to the outcome variable.

$$g(y) = y_0 + c_1 x_1 + c_2 x_2 + c_3 x_3 + ...$$

Here, g is the link function. The normal linear regression model uses the identity function $g(x) = x$. For logistic regression, the link function is $g(x) = \ln \frac{x}{1-x}$.

21.1 GLM implementation `glmnet`

The standard R distriution includes the `glm` function for fitting GLMs. The `glmnet` package extends this to include L1 and L2 regularization. To be more specific, it solves the following problem:

$$\min_{\beta_0,\beta} \frac{1}{N} \sum_{i=1}^{N} w_i l(y_i, \beta_0 + \beta^T x_i) + \lambda \left[\frac{1-\alpha}{2} \|\beta\|_2^2 + \alpha \|\beta\|_1 \right],$$

Here l is the negative log-likelihood contribution for observations. The weights w_i are used to account for sampling weights. The regularization parameters λ and α are controlling regularization. The λ value controls the strength of the penalty. The type of regularization is defined by the α value. For $\alpha = 0$, the penalty is L2 regularization (Euclidean norm $\|\beta\|_2^2$). For $\alpha = 1$, the penalty is L1 regularization (absolute value norm $\|\beta\|_1$). Setting α to a value between 0 and 1 results in a combination of L1 and L2 regularization. Hastie et al. write about the role of α:

> It is known that the ridge penalty shrinks the coefficients of correlated predictors towards each other while the lasso tends to pick one of them and discard the others. The elastic net penalty mixes these two: if predictors are correlated in groups, an $\alpha = 0.5$ tends to either select or leave out the entire group of features. This is a higher level parameter, and users might pick a value upfront or experiment with a

few different values. One use of α is for numerical stability; for example, the elastic net with $\alpha = 1 - \epsilon$ for some small $\epsilon > 0$ performs much like the lasso, but removes any degeneracies and wild behavior caused by extreme correlations.

21.2 `glmnet` in *tidymodels*

The `glmnet` package is accessible in the *tidymodels* framework. It can be selected as an engine for linear regression (see Section A.2.3) and logistic regression (see Section A.4.2).

The λ parameter in the elasticnet equation corresponds to the `penalty` argument. α is controlled by the `mixture` argument.

You can find example code for using `glmnet` in the *tidymodels* in Chapter @ef(model-tuning) and Section @(tune-one-standard-error). Here, we will cover functionality that is specific to `glmnet`.

Load the packages we need for this chapter.

```
library(tidymodels)
library(tidyverse)
library(ggfortify)
library(patchwork)
```

Let's start with training a Lasso model on the `mtcars` dataset.

```
set.seed(123)
rec <- recipe(mpg ~ ., data = mtcars)
spec <- linear_reg(mode = "regression", penalty = 1.0, mixture = 1) %>%
  set_engine("glmnet")
wf <- workflow() %>%
  add_recipe(rec) %>%
  add_model(spec)
wf_fit <- wf %>% fit(mtcars)
```

21.2.1 Coefficients - one of many

We can get the coefficients from the model using the `tidy` function.

```
tidy(wf_fit)
```

```
Attaching package: 'Matrix'

The following objects are masked from 'package:tidyr':

    expand, pack, unpack

Loaded glmnet 4.1-8
```

```
# A tibble: 11 x 3
   term        estimate penalty
   <chr>          <dbl>   <dbl>
 1 (Intercept)  35.3           1
 2 cyl          -0.872         1
 3 disp          0             1
 4 hp           -0.0101        1
 5 drat          0             1
 6 wt           -2.59          1
 7 qsec          0             1
 8 vs            0             1
 9 am            0             1
10 gear          0             1
11 carb          0             1
```

These are the coefficients of the model corresponding to the penalty of 1. We can see that most coefficients are 0. This is the effect of the L1 regularization.

Under the hood, `glmnet` is using a grid of λ values and fits the model for all of these values. We can get the coefficients for all examined λ values using the `extract_fit_engine` function.

```
tidy(wf_fit %>% extract_fit_engine())
```

```
# A tibble: 640 x 5
   term         step estimate lambda dev.ratio
   <chr>       <dbl>    <dbl>  <dbl>     <dbl>
 1 (Intercept)     1     20.1   5.15      0
 2 (Intercept)     2     21.6   4.69      0.129
 3 (Intercept)     3     23.2   4.27      0.248
 4 (Intercept)     4     24.7   3.89      0.347
 5 (Intercept)     5     26.0   3.55      0.429
 6 (Intercept)     6     27.2   3.23      0.497
```

```
 7 (Intercept)     7     28.4   2.95     0.554
 8 (Intercept)     8     29.4   2.68     0.601
 9 (Intercept)     9     30.3   2.45     0.640
10 (Intercept)    10     31.1   2.23     0.673
# i 630 more rows
```

Here are the results for a lambda value close to 1.

```
wf_fit %>%
  extract_fit_engine() %>%
  tidy() %>%
  filter(lambda < 1.1) %>%
  filter(lambda >= 1.0)
```

```
# A tibble: 4 x 5
  term         step estimate lambda dev.ratio
  <chr>       <dbl>    <dbl>  <dbl>     <dbl>
1 (Intercept)    18 35.1        1.06     0.805
2 cyl            18 -0.868      1.06     0.805
3 hp             18 -0.00965    1.06     0.805
4 wt             18 -2.56       1.06     0.805
```

The coefficients are the close to the ones we got from the `tidy` function. `glmnet` interpolates the coefficients for the λ values that are not in the grid.

21.2.2 Plotting the coefficients

Because the coefficients are all estimated for a grid of λ values, we can plot the coefficients as a function of λ. The `glmnet` package provides a function for this. Figure 21.1 shows the resulting plot.

```
opar <- par(mfrow = c(1, 2))
plot(wf_fit %>% extract_fit_engine(), label = TRUE)
plot(wf_fit %>% extract_fit_engine(), label = TRUE, xvar = "lambda")
par(opar)
```

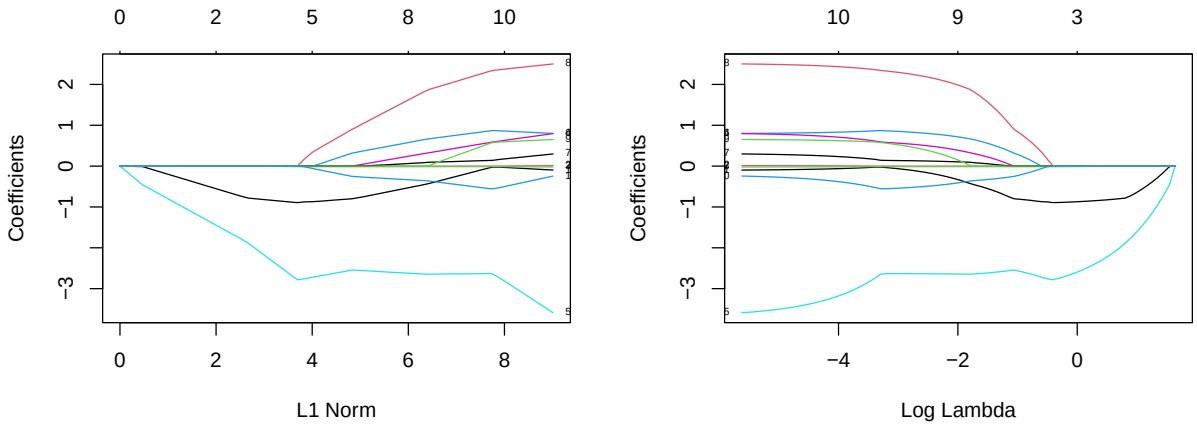


Figure 21.1: Coefficients for the `glmnet` model as a function of the penalty parameter λ.

Each line corresponds to the coefficient value for a given L1 norm of the coefficients. On the left side, all coefficients are 0 due to a very high penalty. As the penalty decreases, the L1 norm increases and we see coefficients different from zero. The numbers on the top of the plot are the number of non-zero coefficients.

> **Useful to know**
>
> Note that the logarithm of lambda used in the right plots is the natural logarithm. This means that for example log lambda of -2 corresponds to lambda of 0.135.

The `ggfortify` package provides a `autoplot` function for `glmnet` models that produces a clearer version of the coefficient values as a function of λ. Because the result is a `ggplot2` graph, you can make additional changes.

```
g1 <- autoplot(wf_fit %>% extract_fit_engine(), xvar = "norm")
g2 <- autoplot(wf_fit %>% extract_fit_engine(), xvar = "lambda")
g1 + g2
```

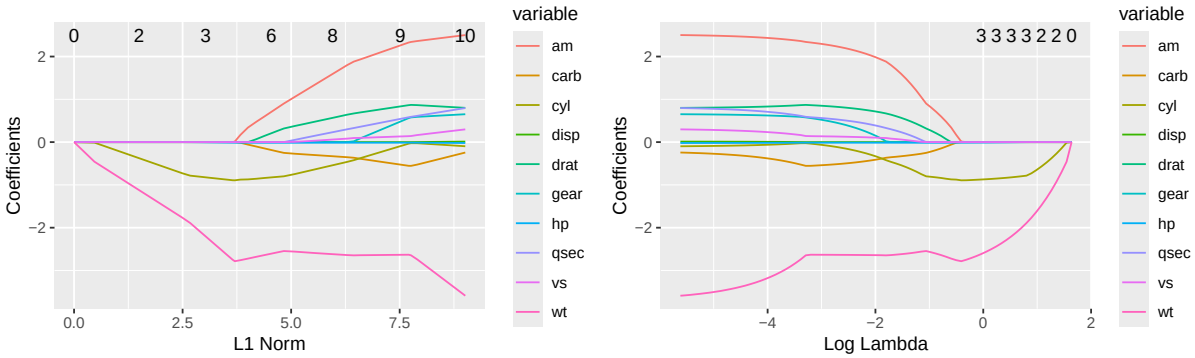


Figure 21.2: Coefficients for the `glmnet` model using the autoplot function from `ggfortify` as a function of L1 norm (left) or lambda (right)

Figure 21.2 shows the resulting plot. The left plot shows the coefficients as a function of the L1 norm. The right plot shows the coefficients as a function of the log of the λ value.

> **Further information**
>
> - `glmnet` introduction
> - Full details about the `glmnet` integration in `parsnip`

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
library(tidymodels)
library(tidyverse)
library(ggfortify)
library(patchwork)
set.seed(123)
rec <- recipe(mpg ~ ., data = mtcars)
spec <- linear_reg(mode = "regression", penalty = 1.0, mixture = 1) %>%
  set_engine("glmnet")
wf <- workflow() %>%
  add_recipe(rec) %>%
  add_model(spec)
wf_fit <- wf %>% fit(mtcars)
tidy(wf_fit)
```

```
tidy(wf_fit %>% extract_fit_engine())
wf_fit %>%
  extract_fit_engine() %>%
  tidy() %>%
  filter(lambda < 1.1) %>%
  filter(lambda >= 1.0)
opar <- par(mfrow = c(1, 2))
plot(wf_fit %>% extract_fit_engine(), label = TRUE)
plot(wf_fit %>% extract_fit_engine(), label = TRUE, xvar = "lambda")
par(opar)
g1 <- autoplot(wf_fit %>% extract_fit_engine(), xvar = "norm")
g2 <- autoplot(wf_fit %>% extract_fit_engine(), xvar = "lambda")
g1 + g2
```

22 Generalized additive models (GAM)

In linear regression, the outcome is a linear function of the predictor variables.

$$y = y_0 + c_1 x_1 + c_2 x_2 + c_3 x_3 + ...$$

where c_i are the coefficients and y_0 is the intercept.

Generalized linear models (GLMs) extend this idea by adding a link function to the outcome variable.

$$g(y) = y_0 + c_1 x_1 + c_2 x_2 + c_3 x_3 + ...$$

Here, g is the link function. The normal linear regression model uses the identity function $g(x) = x$. For logistic regression, the link function is $g(x) = \ln \frac{x}{1-x}$.

A generalized additive model (GAM) extends the concept even further and describes the outcome as a linear function of transformed predictor variables.

$$g(y) = y_0 + c_1 f_1(x_1) + c_2 f_2(x_2) + c_3 f_3(x_3) + ...$$

The functions f_i are usually smooth functions, such as splines, that are estimated from the data. The coefficients c_i are estimated using maximum likelihood estimation.

GAMs are useful when the relationship between the outcome and the predictor variables is not linear. For example, the relationship may be quadratic or sinusoidal. GAMs are also useful when the relationship is not known in advance and will be estimated from the data.

22.1 Specifying GAMs in formula notation

The GAM models extend the formula notation with special smoothing terms. Here is an example:

```
mpg ~ s(displacement) + s(horsepower) + s(weight) +
    acceleration + year
```

The `s()` function indicates that the variable is smoothed. The smoothing is done using splines. See the documentation for full details.

22.2 GAMs in *Tidymodels*

In *tidymodels*, GAMs are available with the `gen_additive_mod` function from the `parsnip` package. The only available engine that can be used is the `mgcv` package. The next section will demonstrate how to train GAM models in *tidymodels*. However, workflows requires a different approach from what we've seen so far.

```
library(tidymodels)
library(tidyverse)
library(patchwork)
```

22.3 Example: GAM for the `mpg` dataset

In the following, we use the `ISLR2::Auto` dataset to predict the fuel efficiency of cars. Load and preprocess the data:

```
auto <- ISLR2::Auto %>%
  as_tibble() %>%
  mutate(
    cylinders = as.factor(cylinders),
    origin = as.factor(origin),
  ) %>%
  select(-name)
```

The dataset contains 392 observations with 8 variables. The outcome variable is `mpg` and the predictor variables are `displacement`, `horsepower`, `weight`, `acceleration`, and `year`.

22.3.1 Utility functions

For convenience, we define a series of utility functions. Open the code block to see the implementations.

```
# create a residual plot
residual_plot <- function(model_fit, data, outcome) {
  result <- tibble(prediction = predict(model_fit, new_data = data)$.pred)
  result["residual"] <- data[outcome] - result["prediction"]
  g <- ggplot(result, aes(x = prediction, y = residual)) +
    geom_point() +
    geom_hline(yintercept = 0, linetype = "dashed") +
    geom_smooth(method = "loess", formula = "y ~ x") +
    labs(x = "Predicted mpg", y = "Residuals")
  return(g)
}

# collect and show model metrics
append_model_metrics <- function(model_metrics, model_fit, model_name) {
  model_metrics <- bind_rows(
    model_metrics,
    bind_cols(
      model = model_name,
      metrics(augment(model_fit, new_data = auto),
        truth = mpg, estimate = .pred)
    )
  )
  return(model_metrics)
}

show_metrics_table <- function(model_metrics) {
  return(
    model_metrics %>%
      pivot_wider(names_from = .metric, values_from = .estimate) %>%
      select(-.estimator) %>%
      knitr::kable(digits = 3) %>%
      kableExtra::kable_styling(full_width = FALSE)
  )
}
```

22.3.2 Linear regression model

```
formula <- mpg ~ displacement + horsepower + weight +
  acceleration + year
lm_model <- linear_reg() %>%
```

```
  set_engine("lm") %>%
  fit(formula, data = auto)
model_metrics <- append_model_metrics(tibble(), lm_model, "Linear model")
show_metrics_table(model_metrics)
```

model	rmse	rsq	mae
Linear model	3.409	0.809	2.619

The residual plot is shown in Figure 22.1.

```
residual_plot(lm_model, auto, "mpg")
```

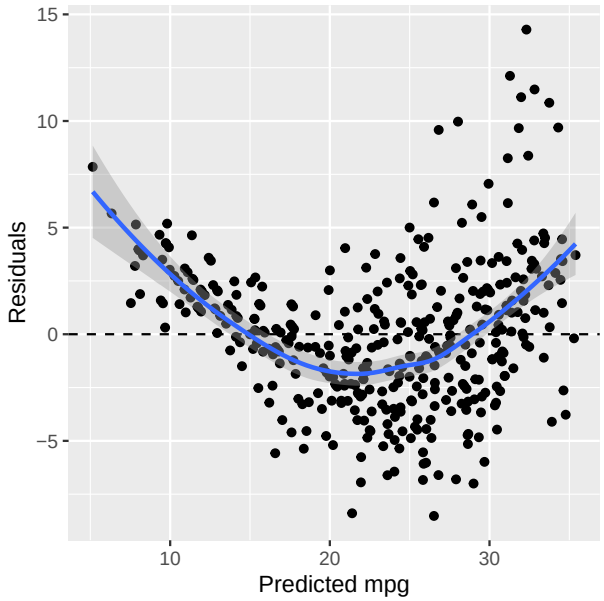


Figure 22.1: Residuals of the linear regression model

We observe two issues with the residuals vs. fit plot.

- The residuals are fanning out, i.e. they have a small spread for small values of the predicted mpg and a larger spread for larger values of the predicted mpg.
- The graph shows a non-linear relationship

Both observations indicate that the model is not a good fit for the data. An obvious approach would be to add quadratic terms to the model. GAMs are an alternative way of addressing this issue.

22.3.3 GAM with splines

We start by adding smoothers to the variable `displacement`, `horsepower`, `weight`, and `acceleration`.

```
gam_formula <- mpg ~ s(displacement) + s(horsepower) + s(weight) +
  s(acceleration) + year
gam_model <- gen_additive_mod() %>%
  set_engine("mgcv") %>%
  set_mode("regression") %>%
  fit(gam_formula, data = auto)

model_metrics <- append_model_metrics(model_metrics, gam_model, "GAM")
show_metrics_table(model_metrics)
```

model	rmse	rsq	mae
Linear model	3.409	0.809	2.619
GAM	2.835	0.868	2.084

The metrics show that the GAM model is a much better fit than the linear regression model. This is also obvious in the residual plot in Figure 22.2. The residuals are smaller and the non-linearity is less pronounced.

```
g1 <- residual_plot(lm_model, auto, "mpg") +
  labs(title = "Linear regression") +
  ylim(-10, 15)
g2 <- residual_plot(gam_model, auto, "mpg") +
  labs(title = "GAM") +
  ylim(-10, 15)
g1 + g2
```

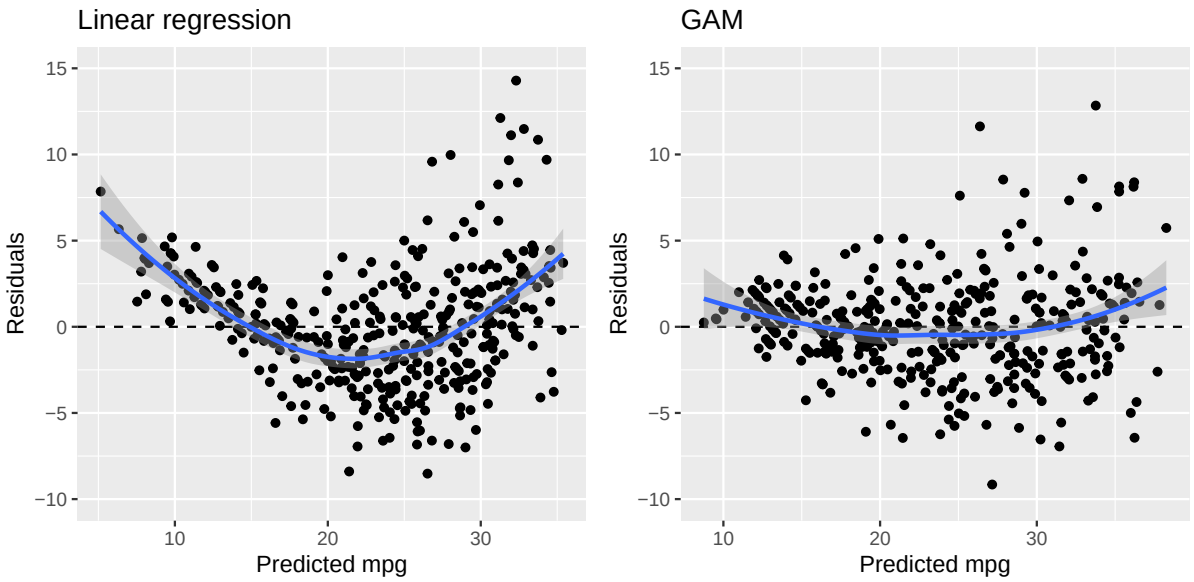


Figure 22.2: Residuals of the linear regression (left) and GAM (right) models

There is still some heteroskedasticity in the residuals.

22.3.4 GAM in workflows

The GAM model can also be used in a workflow. However, we can only use formulas in recipes that specify variables and not transformations. Instead, we need to use the `add_variables` method and specify the formula in the `add_model` function.

```
spec <- gen_additive_mod() %>%
  set_engine("mgcv") %>%
  set_mode("regression")

wf <- workflow() %>%
  add_variables(outcomes = c(mpg),
    predictors = c(displacement, horsepower, weight,
      acceleration, year)) %>%
  add_model(spec, formula = gam_formula)
wf_model <- wf %>% fit(data = auto)
model_metrics <- append_model_metrics(model_metrics, wf_model,
  "GAM-wf")
show_metrics_table(model_metrics)
```

model	rmse	rsq	mae
Linear model	3.409	0.809	2.619
GAM	2.835	0.868	2.084
GAM-wf	2.835	0.868	2.084

The metrics are, as expected, identical to the results from the previous GAM model and Figure 22.3 shows the same residual plot as before.

```
residual_plot(wf_model, auto, "mpg") +
  labs(title = "GAM")
```

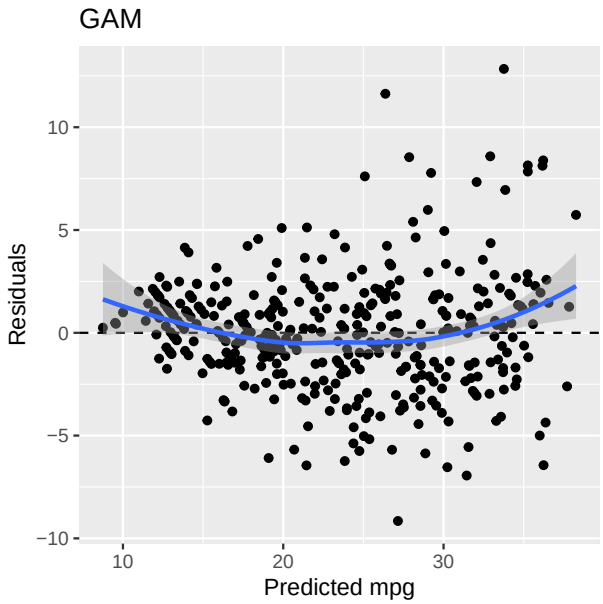


Figure 22.3: Residuals of the linear regression model trained using a workflow

However, using a workflow, we can also use a recipe and specify transformations to the variables. For example, we can use the Yeo-Johnson transformation to transform the variables as shown in the following example.

```
rec <- recipe(mpg ~ displacement + horsepower + weight +
    acceleration + year, data = auto) %>%
  step_YeoJohnson(all_numeric_predictors())
spec <- gen_additive_mod() %>%
  set_engine("mgcv") %>%
```

```
  set_mode("regression")

wf <- workflow() %>%
  add_recipe(rec) %>%
  add_model(spec, formula = gam_formula)
wf_model_2 <- wf %>% fit(data = auto)
model_metrics <- append_model_metrics(model_metrics, wf_model_2,
  "GAM-wf-2")
show_metrics_table(model_metrics)
```

model	rmse	rsq	mae
Linear model	3.409	0.809	2.619
GAM	2.835	0.868	2.084
GAM-wf	2.835	0.868	2.084
GAM-wf-2	2.800	0.871	2.075

The performance metrics are slightly better than before and if you analyze the residual plot, there is a hint of a reduction in the heteroskedasticity. Note however that the formula now refers to the transformed variables and not the original variables.

22.4 Using the plot function of the `mgcv` model

The `mgcv` package provides a `plot` function that can be used to visualize the components of the GAM model. The following code block shows how to use this function. It is important to explicitly load the `mgcv` package. Otherwise, the `plot` function will not be available. Note the argument `scale=0` which ensures that the plots are not scaled to the same y-axis. This is useful to see the individual components of the model.

```
library(mgcv)  # this is important to load the plot function
opar <- par(mfrow = c(2, 2))
plot(gam_model %>% extract_fit_engine(), scale = 0)
par(opar)
```

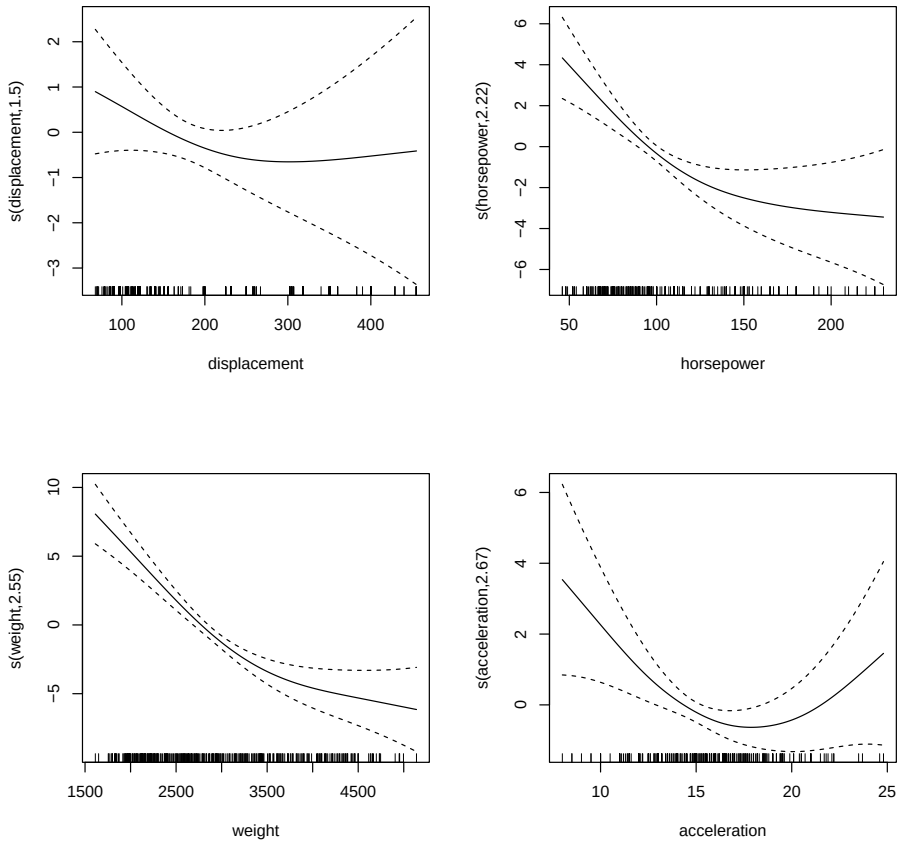


Figure 22.4: Plots of the GAM model

Figure 22.4 shows that the GAM model fits spline with for `displacement` and `horsepower` with considerable non-linearity.

We can also visualize the components from the workflow model. Figure 22.5 shows the results. As already mentioned, the formula refers to the transformed variables, so `displacment` now refers to the Yeo-Johnson transformed variable. Note, the more evenly distributed data points in the rug plot components.

```
library(mgcv)  # this is important to load the plot function
opar <- par(mfrow = c(2, 2))
plot(wf_model_2 %>% extract_fit_engine(), scale = 0)
par(opar)
```

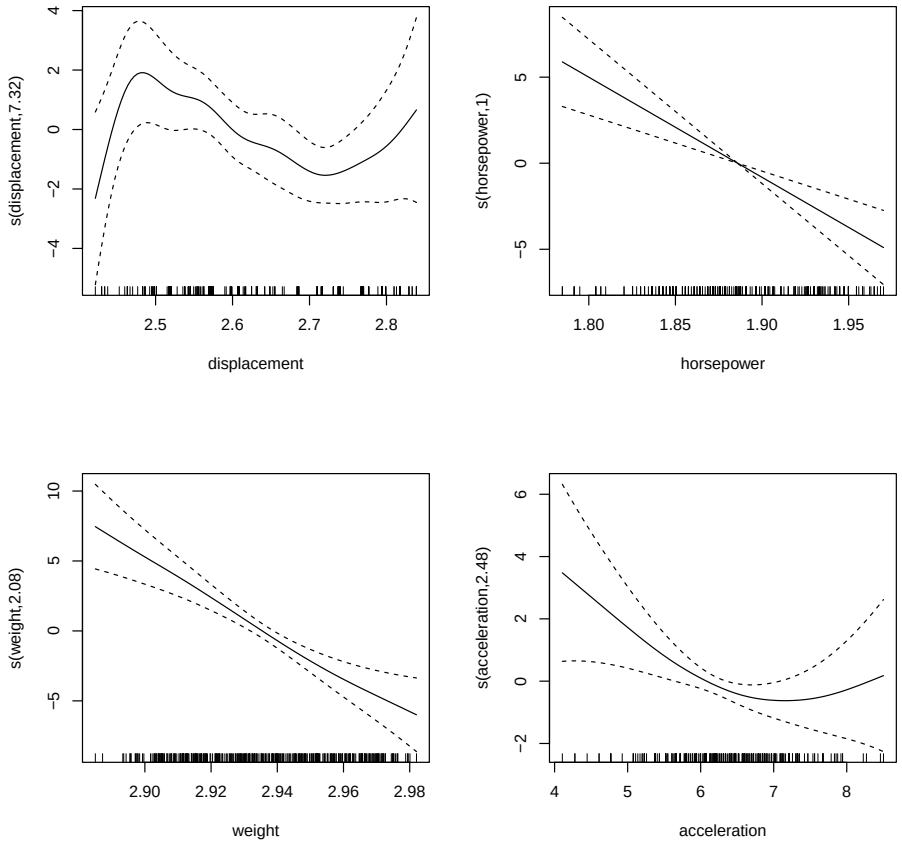


Figure 22.5: Plots of the GAM model

> **Further information**
>
> - `gen_additive_mod` reference
> - `mgcv` engine in parsnip
> - GAM formula notation
> - Wikipedia article on GAM

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
library(tidymodels)
```

```
library(tidyverse)
library(patchwork)
auto <- ISLR2::Auto %>%
  as_tibble() %>%
  mutate(
    cylinders = as.factor(cylinders),
    origin = as.factor(origin),
  ) %>%
  select(-name)
# create a residual plot
residual_plot <- function(model_fit, data, outcome) {
  result <- tibble(prediction = predict(model_fit, new_data = data)$.pred)
  result["residual"] <- data[outcome] - result["prediction"]
  g <- ggplot(result, aes(x = prediction, y = residual)) +
    geom_point() +
    geom_hline(yintercept = 0, linetype = "dashed") +
    geom_smooth(method = "loess", formula = "y ~ x") +
    labs(x = "Predicted mpg", y = "Residuals")
  return(g)
}

# collect and show model metrics
append_model_metrics <- function(model_metrics, model_fit, model_name) {
  model_metrics <- bind_rows(
    model_metrics,
    bind_cols(
      model = model_name,
      metrics(augment(model_fit, new_data = auto),
        truth = mpg, estimate = .pred)
    )
  )
  return(model_metrics)
}

show_metrics_table <- function(model_metrics) {
  return(
    model_metrics %>%
      pivot_wider(names_from = .metric, values_from = .estimate) %>%
      select(-.estimator) %>%
      knitr::kable(digits = 3) %>%
      kableExtra::kable_styling(full_width = FALSE)
  )
```

```
}
formula <- mpg ~ displacement + horsepower + weight +
  acceleration + year
lm_model <- linear_reg() %>%
  set_engine("lm") %>%
  fit(formula, data = auto)
model_metrics <- append_model_metrics(tibble(), lm_model, "Linear model")
show_metrics_table(model_metrics)
residual_plot(lm_model, auto, "mpg")
gam_formula <- mpg ~ s(displacement) + s(horsepower) + s(weight) +
  s(acceleration) + year
gam_model <- gen_additive_mod() %>%
  set_engine("mgcv") %>%
  set_mode("regression") %>%
  fit(gam_formula, data = auto)

model_metrics <- append_model_metrics(model_metrics, gam_model, "GAM")
show_metrics_table(model_metrics)
g1 <- residual_plot(lm_model, auto, "mpg") +
  labs(title = "Linear regression") +
  ylim(-10, 15)
g2 <- residual_plot(gam_model, auto, "mpg") +
  labs(title = "GAM") +
  ylim(-10, 15)
g1 + g2
spec <- gen_additive_mod() %>%
  set_engine("mgcv") %>%
  set_mode("regression")

wf <- workflow() %>%
  add_variables(outcomes = c(mpg),
    predictors = c(displacement, horsepower, weight,
      acceleration, year)) %>%
  add_model(spec, formula = gam_formula)
wf_model <- wf %>% fit(data = auto)
model_metrics <- append_model_metrics(model_metrics, wf_model,
  "GAM-wf")
show_metrics_table(model_metrics)
residual_plot(wf_model, auto, "mpg") +
  labs(title = "GAM")
rec <- recipe(mpg ~ displacement + horsepower + weight +
    acceleration + year, data = auto) %>%
```

```
  step_YeoJohnson(all_numeric_predictors())
spec <- gen_additive_mod() %>%
  set_engine("mgcv") %>%
  set_mode("regression")

wf <- workflow() %>%
  add_recipe(rec) %>%
  add_model(spec, formula = gam_formula)
wf_model_2 <- wf %>% fit(data = auto)
model_metrics <- append_model_metrics(model_metrics, wf_model_2,
  "GAM-wf-2")
show_metrics_table(model_metrics)
library(mgcv)  # this is important to load the plot function
opar <- par(mfrow = c(2, 2))
plot(gam_model %>% extract_fit_engine(), scale = 0)
par(opar)
library(mgcv)  # this is important to load the plot function
opar <- par(mfrow = c(2, 2))
plot(wf_model_2 %>% extract_fit_engine(), scale = 0)
par(opar)
```

23 Visualizing decision tree models

Decision tree models are popular because they are easy to interpret and to understand. This lead to the development of packages to facilitate the analysis of decision trees. Here, we will focus on the `ggparty` package and demonstrate some of its features using the `ISLR2::Carseats` data set.

Load required libraries

```
library(tidyverse)
library(tidymodels)
library(ggparty)
```

Prepare the ISLR2::Carseats data set for classification. The `Sales` variable is converted to a factor with two levels `Yes` and `No` based on the median value of `Sales` (7.49). The `Sales` variable is then removed from the data set.

```
carseats <- tibble(ISLR2::Carseats) %>%
  mutate(
    High = factor(ifelse(Sales <= median(Sales), "No", "Yes"))
  ) %>%
  dplyr::select(-c(Sales))
```

23.1 Classification Trees

We first train a model using the default settings.

```
model <- decision_tree(mode = "classification", engine = "rpart") %>%
  fit(High ~ ., data = carseats)
```

> Useful to know
>
> I've not been able to use the `ggparty` functionality with a decision tree model created using a workflow. If you use a workflow to tune a decision tree, you will need to create a separate decision tree model with the settings from the tuning.

23.1.1 Visualizing the tree (graph)

To use the `ggparty` visualization, we need to extract the `rpart` model and convert it into a format suitable for this package using the function `partykit::as.party(model$fit)`.

Figure 23.1 shows the tree visualization created by the `autoplot` implementation of the `party` object.

```
autoplot(partykit::as.party(model$fit))
```

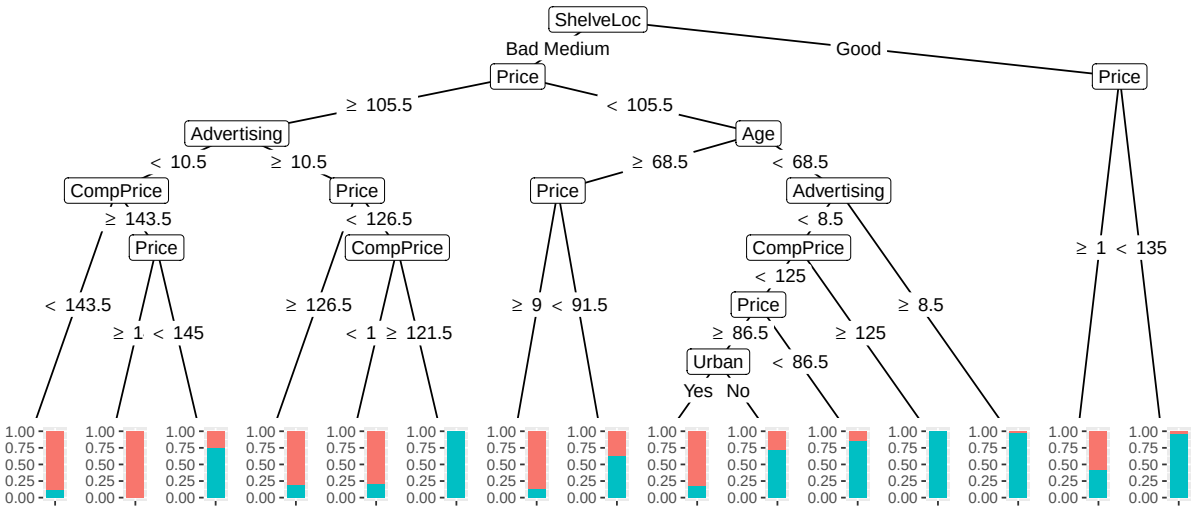


Figure 23.1: Decision tree visualization of default model (autoplot)

There are many options to customize the visualization. Figure 23.2 shows the tree visualization created by the `ggparty` implementation of the `party` object. The pie charts show the distribution of the two classes and the number of training data points in the terminal nodes.

```
ggparty::ggparty(partykit::as.party(model$fit)) +
  ggparty::geom_edge() +
  ggparty::geom_edge_label() +
  ggparty::geom_node_label(aes(label = splitvar), ids = "inner") +
  ggparty::geom_node_plot(gglist = list(
    geom_bar(aes(x = "", fill = High)),
    coord_polar("y"),
    theme_void()))
```

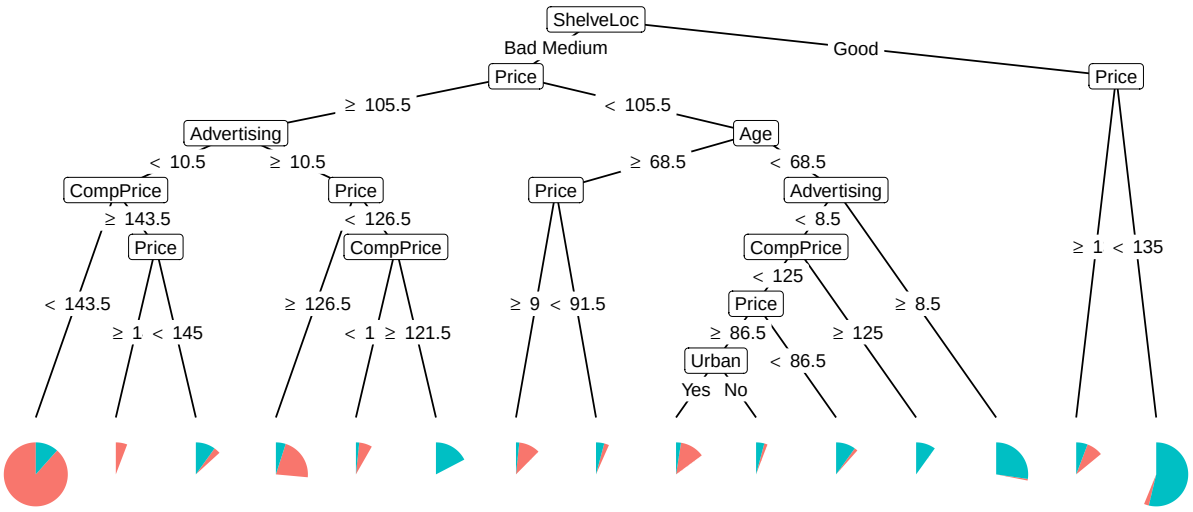


Figure 23.2: Decision tree visualization using pie charts

23.1.2 Visualizing the tree (text)

The conversion of the `rpart` model into a `party` object also allows us to print the tree as text in a format that is clearer than the default `rpart` output.

```
tree_carseats_party <- partykit::as.party(model$fit)
tree_carseats_party
```

```
Model formula:
High ~ CompPrice + Income + Advertising + Population + Price +
    ShelveLoc + Age + Education + Urban + US

Fitted party:
[1] root
|   [2] ShelveLoc in Bad, Medium
|   |   [3] Price >= 105.5
|   |   |   [4] Advertising < 10.5
|   |   |   |   [5] CompPrice < 143.5: No (n = 121, err = 11.6%)
|   |   |   |   [6] CompPrice >= 143.5
|   |   |   |   |   [7] Price >= 145: No (n = 7, err = 0.0%)
|   |   |   |   |   [8] Price < 145: Yes (n = 16, err = 25.0%)
|   |   |   [9] Advertising >= 10.5
|   |   |   |   [10] Price >= 126.5: No (n = 32, err = 18.8%)
```

```
|   |   |   |   [11] Price < 126.5
|   |   |   |   |   [12] CompPrice < 121.5: No (n = 10, err = 20.0%)
|   |   |   |   |   [13] CompPrice >= 121.5: Yes (n = 21, err = 0.0%)
|   |   [14] Price < 105.5
|   |   |   [15] Age >= 68.5
|   |   |   |   [16] Price >= 91.5: No (n = 15, err = 13.3%)
|   |   |   |   [17] Price < 91.5: Yes (n = 8, err = 37.5%)
|   |   |   [18] Age < 68.5
|   |   |   |   [19] Advertising < 8.5
|   |   |   |   |   [20] CompPrice < 125
|   |   |   |   |   |   [21] Price >= 86.5
|   |   |   |   |   |   |   [22] Urban in Yes: No (n = 18, err = 16.7%)
|   |   |   |   |   |   |   [23] Urban in No: Yes (n = 7, err = 28.6%)
|   |   |   |   |   |   [24] Price < 86.5: Yes (n = 14, err = 14.3%)
|   |   |   |   |   [25] CompPrice >= 125: Yes (n = 12, err = 0.0%)
|   |   |   |   [26] Advertising >= 8.5: Yes (n = 34, err = 2.9%)
|   [27] ShelveLoc in Good
|   |   [28] Price >= 135: No (n = 17, err = 41.2%)
|   |   [29] Price < 135: Yes (n = 68, err = 4.4%)

Number of inner nodes:    14
Number of terminal nodes: 15
```

Let's look at node `[7]`. The path is

- ShelveLoc in Bad, Medium
- Price >= 105.5
- Advertising < 10.5
- CompPrice >= 143.5
- Price >= 145

The distribution at that node is:

```
No (n = 7, err = 0.0%)
```

We predict that `High` is `No` with 0% error.

23.1.3 Visualizing the tree (rules)

The example above shows that `Price` occurred twice in the decision path. We can combine the two rules into one rule.

It can be difficult to extract the decision path for complex trees. The `rpart.plot` package has a function to convert the tree into a set of rules:

- ShelveLoc in Bad, Medium
- Advertising < 10.5
- CompPrice >= 143.5
- Price >= 145

The function `rpart.plot::rpart.rules` can be used to extract the rules from the tree.

```
rpart.plot::rpart.rules(model$fit, style="tallw")
```

Abbreviated output, the first rule corresponds to node `[7]`.

```
## High is 0.00 when
##               ShelveLoc is Bad or Medium
##               Price >= 145
##               Advertising < 10.5
##               CompPrice >= 144
##
## High is 0.12 when
##               ShelveLoc is Bad or Medium
##               Price >= 106
##               Advertising < 10.5
##               CompPrice < 144
##
## High is 0.13 when
##               ShelveLoc is Bad or Medium
##               Price is 92 to 106
##               Age >= 69
....
```

23.2 Regression Trees

We now train a regression model to predict `Sales` in the `ISLR2::Carseats` dataset using the default settings.

```
model <- decision_tree(mode = "regression", engine = "rpart") %>%
  fit(Sales ~ ., data = ISLR2::Carseats)
```

23.2.1 Visualizing the tree (graph)

Figure 23.3 shows the tree visualization created by the `autoplot` implementation of the `party` object.

```
ggparty::ggparty(partykit::as.party(model$fit), horizontal = TRUE) +
  ggparty::geom_edge() +
  ggparty::geom_edge_label() +
  ggparty::geom_node_label(aes(label = splitvar), ids = "inner") +
  ggparty::geom_node_plot(gglist = list(
    geom_histogram(aes(x = Sales), binwidth = 3),
    theme(
      axis.title.x = element_blank(),
      axis.title.y = element_blank())))
```

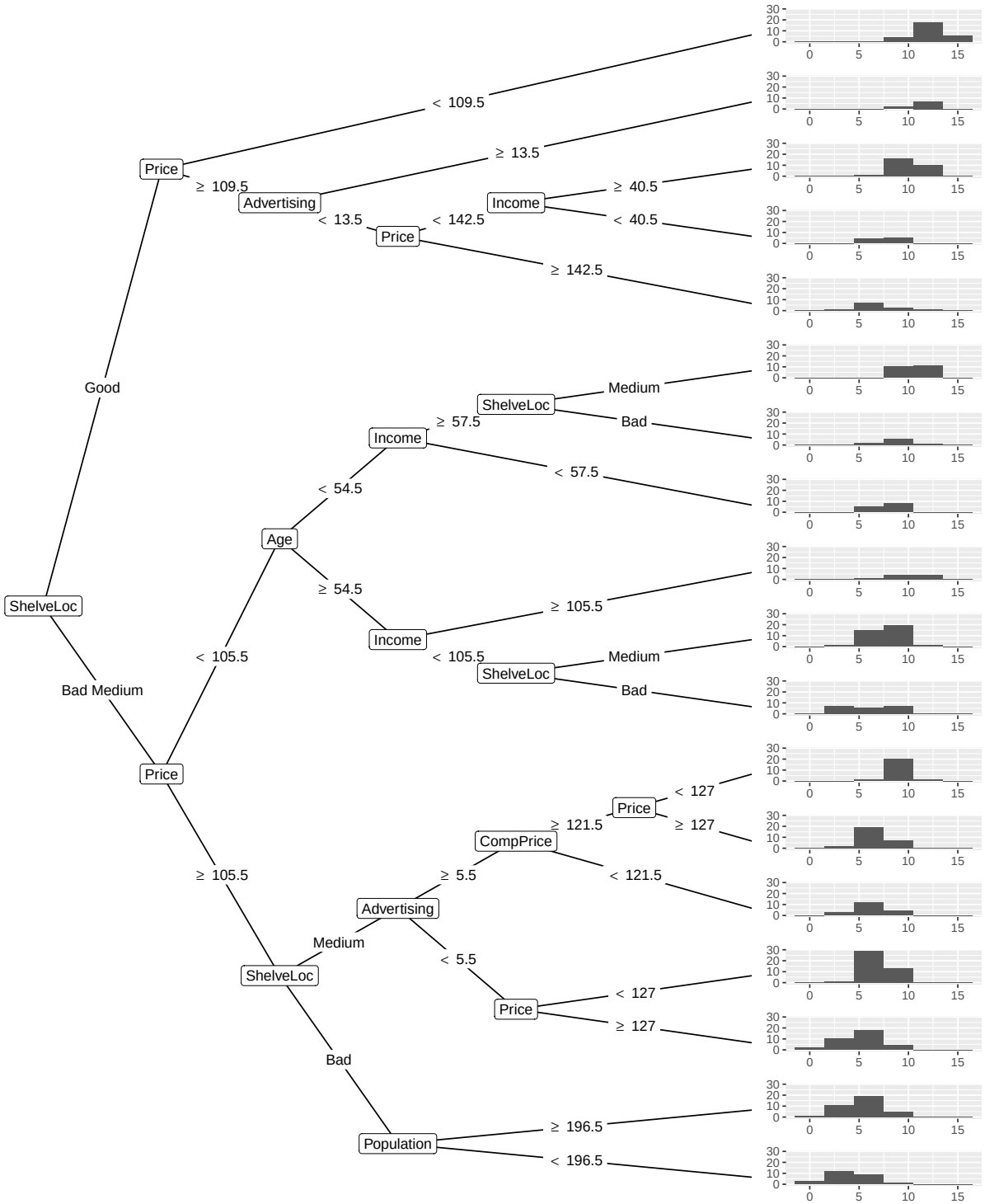


Figure 23.3: Decision tree visualization of regression model

Further information

There are many ways of customizing the visualization and you can find plenty of examples and resources on the internet. See the ggparty wiki.

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
library(tidyverse)
library(tidymodels)
library(ggparty)
carseats <- tibble(ISLR2::Carseats) %>%
  mutate(
    High = factor(ifelse(Sales <= median(Sales), "No", "Yes"))
  ) %>%
  dplyr::select(-c(Sales))
model <- decision_tree(mode = "classification", engine = "rpart") %>%
  fit(High ~ ., data = carseats)
autoplot(partykit::as.party(model$fit))
ggparty::ggparty(partykit::as.party(model$fit)) +
  ggparty::geom_edge() +
  ggparty::geom_edge_label() +
  ggparty::geom_node_label(aes(label = splitvar), ids = "inner") +
  ggparty::geom_node_plot(gglist = list(
    geom_bar(aes(x = "", fill = High)),
    coord_polar("y"),
    theme_void()))
tree_carseats_party <- partykit::as.party(model$fit)
tree_carseats_party
model <- decision_tree(mode = "regression", engine = "rpart") %>%
  fit(Sales ~ ., data = ISLR2::Carseats)
ggparty::ggparty(partykit::as.party(model$fit), horizontal = TRUE) +
  ggparty::geom_edge() +
  ggparty::geom_edge_label() +
  ggparty::geom_node_label(aes(label = splitvar), ids = "inner") +
  ggparty::geom_node_plot(gglist = list(
    geom_histogram(aes(x = Sales), binwidth = 3),
    theme(
```

```
      axis.title.x = element_blank(),
      axis.title.y = element_blank())))
```

24 Variable or feature importance

variable importance, also known as feature importance, is a measure of the influence of a feature on the prediction of a model. It helps you understand why a model makes a specific prediction and explain it. It can also help you to identify potential bias and errors in the model.

The approaches to calculate variable importance can be grouped into local and global measures of variable importance. Local measures are calculated for a single prediction, while global measures are calculated for the entire dataset.

In this chapter, we will focus on global measures of variable importance. We will use the `vip` package to calculate variable importance for a random forest model trained on the `mtcars` dataset. The `vip` package provides a unified interface to calculate variable importance for different models and datasets. It supports different methods to calculate variable importance, such as permutation-based importance, SHAP values, and others.

Load required libraries

```
library(tidyverse)
library(tidymodels)
library(kableExtra)
library(patchwork)
library(vip)
library(ranger)
library(pdp)
```

24.1 The `vip` package

The `vip` package provides a unified interface to calculate variable importance for different models and datasets.

24.2 Model specific measures of variable importance

24.2.1 Linear model

A common measure of variable importance in linear models is the t-statistic.

Let's start with a linear regression model trained on the `mtcars` dataset to predict `mpg`.

```
mtcars_rec <- recipe(mpg ~ ., data = mtcars) %>%
  step_normalize(all_numeric_predictors())
lm_fit <- workflow() %>%
  add_recipe(mtcars_rec) %>%
  add_model(linear_reg(mode = "regression")) %>%
  fit(mtcars)
```

To use the `vip` package, we need to extract the fit engine from the workflow and pass it to the `vip` function. Figure 24.1 shows the resulting graph:

```
lm_fit %>%
  extract_fit_engine() %>%
  vip()
```

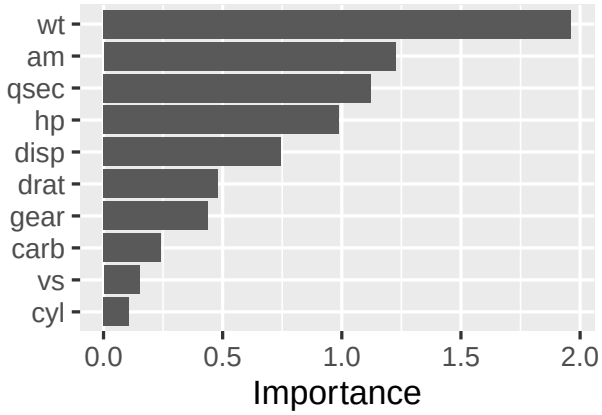


Figure 24.1: Variable importance for a linear regression model (t-statistic)

If we compare the graph with the coefficients of the linear model, we see that the graph shows the absolute values of the t-statistic of the coefficients.

```
lm_fit %>%
  extract_fit_engine() %>%
  summary() %>%
  pluck(coefficients)
```

```
             Estimate Std. Error    t value     Pr(>|t|)
(Intercept) 20.0906250  0.4684931 42.8835050 6.185024e-22
cyl         -0.1990240  1.8663298 -0.1066392 9.160874e-01
disp         1.6527522  2.2132353  0.7467585 4.634887e-01
hp          -1.4728757  1.4925162 -0.9868407 3.349553e-01
drat         0.4208515  0.8743992  0.4813036 6.352779e-01
wt          -3.6352668  1.8536038 -1.9611887 6.325215e-02
qsec         1.4671532  1.3059782  1.1234133 2.739413e-01
vs           0.1601576  1.0607063  0.1509915 8.814235e-01
am           1.2575703  1.0262499  1.2254035 2.339897e-01
gear         0.4835664  1.1017333  0.4389142 6.652064e-01
carb        -0.3221020  1.3386010 -0.2406258 8.121787e-01
```

The most influentials features are, unsurprisingly, `wt`, `am`, `qsec`, and `hp`.

24.2.2 Random forests

Let's begin with training a random forest model using the `ranger` package. By setting the `importance` argument, the `ranger` model will collect information about the effect of each feature on improving the model's performance at each split in the decision trees.

```
rf_spec <- rand_forest(mtry = 2, mode = "regression") %>%
  set_engine("ranger", importance = "impurity")
wf <- workflow() %>%
  add_recipe(mtcars_rec) %>%
  add_model(rf_spec)
rf_fit <- wf %>% fit(mtcars)

rf_fit %>%
  extract_fit_engine() %>%
  vip()
```

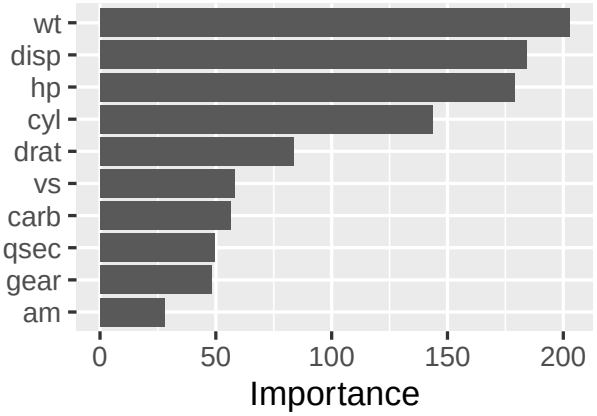


Figure 24.2: Variable importance for a linear regression model (random forest)

The result is shown in Figure 24.2. This time, different features are considered most important: `disp`, `wt`, `hp`, and `cyl`.

24.3 General approaches to calculate variable importance

While the model-specific measures of variable importance are useful, more general approaches were developed to determine variable importance for a wider range of models.

Figure 24.3 shows the variable importance calculated using the FIRM method (left) and the permutation method (right).

```
# extract the RF model from the workflow
rf_model <- rf_fit %>% extract_fit_engine()

# Define a prediction wrapper function
pfun <- function(object, newdata) {
  return(predict(object, data = newdata)$predictions)
}
vis_firm <- vi(rf_model, method = "firm", target = "mpg",
  metric = "rmse", pred_wrapper = pfun,
  train = bake(prep(mtcars_rec), new_data = NULL))

g1 <- vip(vis_firm) + labs(title = "FIRM")
vis_permute <- vi(rf_model, method = "permute", target = "mpg",
  metric = "rmse", nsim = 10, pred_wrapper = pfun,
  train = bake(prep(mtcars_rec), new_data = NULL))
```

```
g2 <- vip(vis_permute) + labs(title = "Permutation")
g1 + g2
```

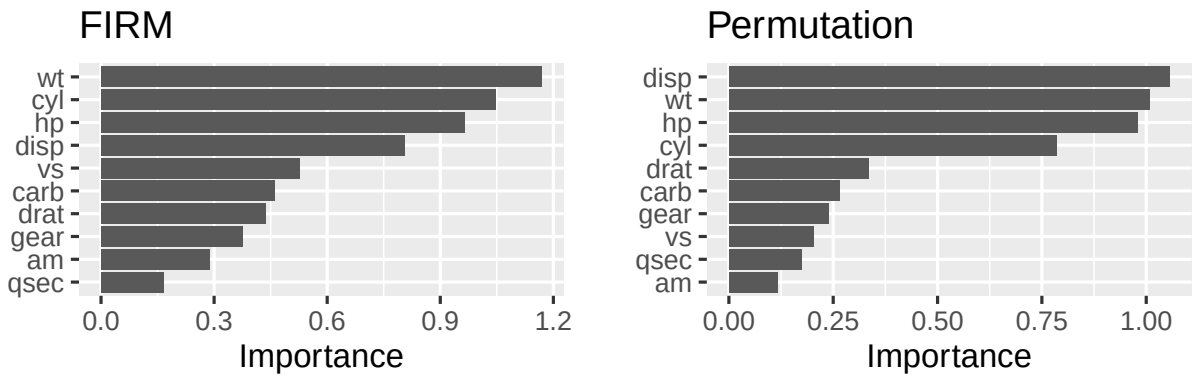


Figure 24.3: Variable importance determined using FIRM (left) and permutation approach (right)

The FIRM method looks at the effect of each feature on the prediction of the model. It is based on partial dependency plots. The effect of each predictor is determined by measuring the variation in the model's prediction when the feature is changed while keeping all other features constant. Figure 24.4 shows the partial dependency plots for each feature. The individual graphs are ordered in decreasing importance according to the FIRM method.

```
# combine effects into a single data frame
effects <- attr(vis_firm, which = "effects")
effect_data <- tibble()
for (name in vis_firm$Variable) {
  effect <- effects[[name]]
  effect_data <- bind_rows(
    effect_data,
    tibble(
      name = name,
      predictor = effect[[name]],
      yhat = effect$yhat
    )
  )
}
# order the predictors by FIRM importance
effect_data <- effect_data %>%
  mutate(name = factor(name, levels = vis_firm$Variable))
```

```
ggplot(effect_data, aes(x = predictor, y = yhat)) +
  geom_line() +
  facet_wrap(~name, ncol = 5)
```

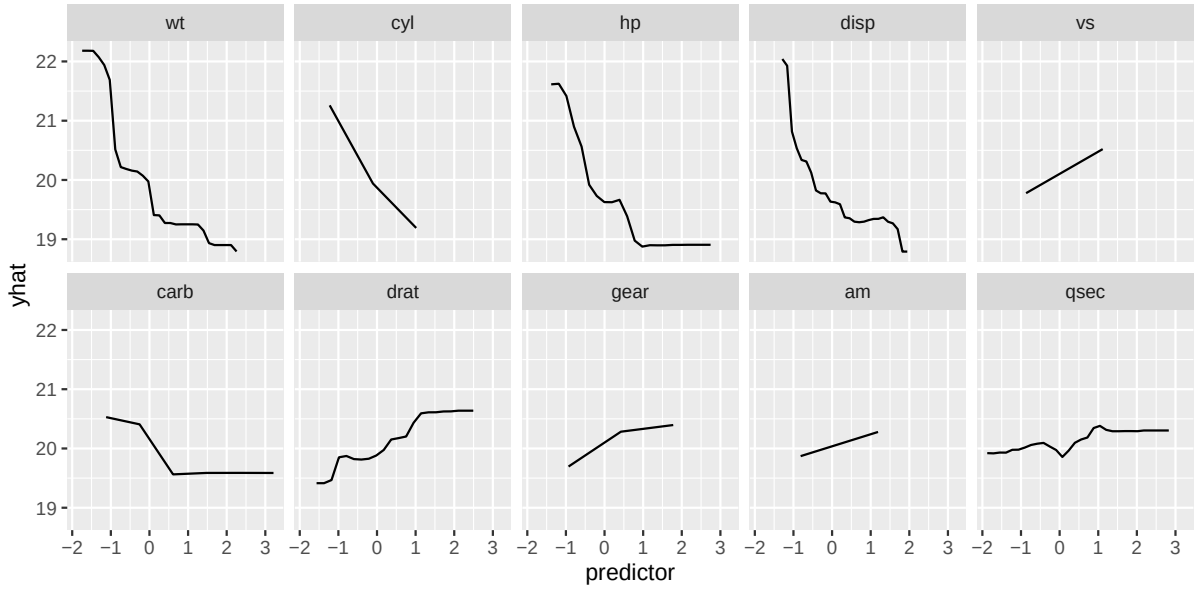


Figure 24.4: Partial dependency plots for each feature

The permutation method is a more straightforward approach. It calculates the variable importance by permuting the values of each feature and measuring the change in the model's performance. In our calculation of `vis_permute` we used `nsim=10` permutations. The individual results are shown in Figure 24.5.

```
g1 <- vip(vis_permute, geom = "boxplot", all_permutations = TRUE,
  jitter = TRUE)
g2 <- vip(vis_permute, all_permutations = TRUE, jitter = TRUE)
g1 + g2
```

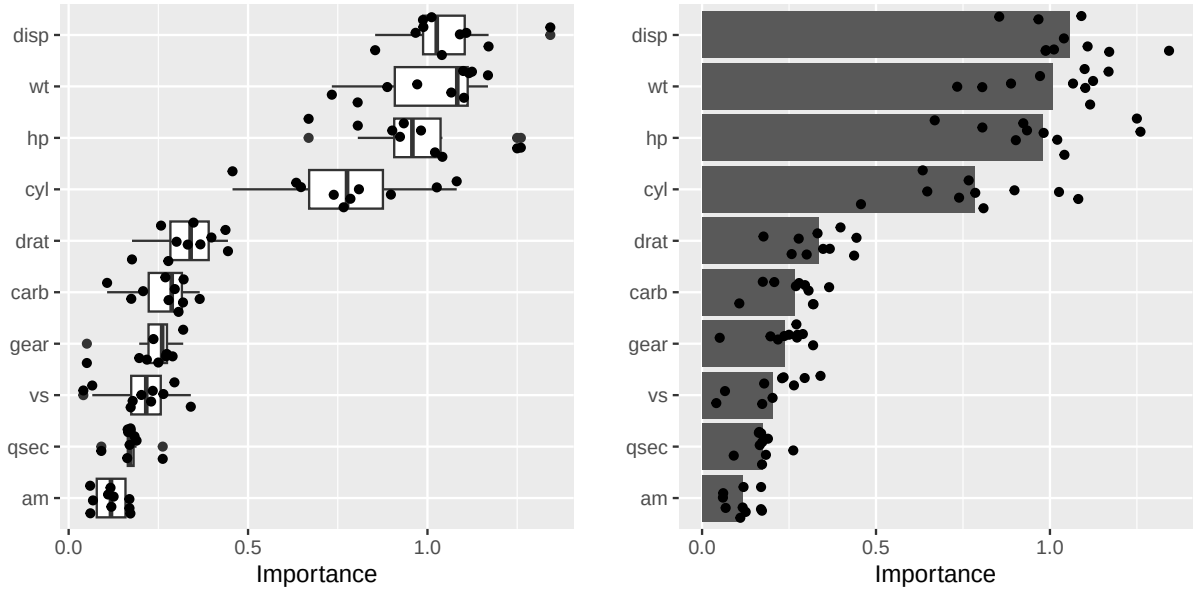


Figure 24.5: Different visualizations of the results of the permutation method

> **Further information**
>
> Additional information can be found in the following resources:
>
> - `vip` package documentation
> - `vip` overview

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
library(tidyverse)
library(tidymodels)
library(kableExtra)
library(patchwork)
library(vip)
library(ranger)
library(pdp)
mtcars_rec <- recipe(mpg ~ ., data = mtcars) %>%
```

```
  step_normalize(all_numeric_predictors())
lm_fit <- workflow() %>%
  add_recipe(mtcars_rec) %>%
  add_model(linear_reg(mode = "regression")) %>%
  fit(mtcars)
lm_fit %>%
  extract_fit_engine() %>%
  vip()
lm_fit %>%
  extract_fit_engine() %>%
  summary() %>%
  pluck(coefficients)
rf_spec <- rand_forest(mtry = 2, mode = "regression") %>%
  set_engine("ranger", importance = "impurity")
wf <- workflow() %>%
  add_recipe(mtcars_rec) %>%
  add_model(rf_spec)
rf_fit <- wf %>% fit(mtcars)

rf_fit %>%
  extract_fit_engine() %>%
  vip()
# extract the RF model from the workflow
rf_model <- rf_fit %>% extract_fit_engine()

# Define a prediction wrapper function
pfun <- function(object, newdata) {
  return(predict(object, data = newdata)$predictions)
}
vis_firm <- vi(rf_model, method = "firm", target = "mpg",
  metric = "rmse", pred_wrapper = pfun,
  train = bake(prep(mtcars_rec), new_data = NULL))

g1 <- vip(vis_firm) + labs(title = "FIRM")
vis_permute <- vi(rf_model, method = "permute", target = "mpg",
  metric = "rmse", nsim = 10, pred_wrapper = pfun,
  train = bake(prep(mtcars_rec), new_data = NULL))
g2 <- vip(vis_permute) + labs(title = "Permutation")
g1 + g2
# combine effects into a single data frame
effects <- attr(vis_firm, which = "effects")
effect_data <- tibble()
```

```
for (name in vis_firm$Variable) {
  effect <- effects[[name]]
  effect_data <- bind_rows(
    effect_data,
    tibble(
      name = name,
      predictor = effect[[name]],
      yhat = effect$yhat
    )
  )
}
# order the predictors by FIRM importance
effect_data <- effect_data %>%
  mutate(name = factor(name, levels = vis_firm$Variable))

ggplot(effect_data, aes(x = predictor, y = yhat)) +
  geom_line() +
  facet_wrap(~name, ncol = 5)
g1 <- vip(vis_permute, geom = "boxplot", all_permutations = TRUE,
  jitter = TRUE)
g2 <- vip(vis_permute, all_permutations = TRUE, jitter = TRUE)
g1 + g2
```

Part VIII

Examples

25 Model tuning

In this example, we will

- load and preprocess data
- define a workflow with tunable parameters
- tune the hyperparameters using Bayesian optimization
- train the final model
- evaluate the model using cross-validation and holdout data

using the Loan prediction dataset to illustrate the whole process.

Load the required packages:

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

Load and preprocess the data:

```
file <- "https://gedeck.github.io/DS-6030/datasets/loan_prediction.csv"
data <- read_csv(file, show_col_types = FALSE) %>%
  drop_na() %>%
  mutate(
    Gender = as.factor(Gender),
    Married = as.factor(Married),
    Dependents = gsub("\\+", "", Dependents) %>% as.numeric(),
    Education = as.factor(Education),
    Self_Employed = as.factor(Self_Employed),
    Credit_History = as.factor(Credit_History),
    Property_Area = as.factor(Property_Area),
    Loan_Status = factor(Loan_Status, levels = c("N", "Y"),
      labels = c("No", "Yes"))
  ) %>%
  select(-Loan_ID)
```

Split dataset into training and holdout data, prepare for cross-validation:

```
set.seed(123)
data_split <- initial_split(data, prop = 0.8, strata = Loan_Status)
train_data <- training(data_split)
holdout_data <- testing(data_split)

resamples <- vfold_cv(train_data, v = 10, strata = Loan_Status)
cv_metrics <- metric_set(roc_auc, accuracy)
cv_control <- control_resamples(save_pred = TRUE)
```

Define the recipe, the model specification (elasticnet logistic regression), and combine them into a workflow:

```
formula <- Loan_Status ~ Gender + Married + Dependents + Education +
  Self_Employed + ApplicantIncome + CoapplicantIncome + LoanAmount +
  Loan_Amount_Term + Credit_History + Property_Area
recipe_spec <- recipe(formula, data = train_data) %>%
  step_dummy(all_nominal(), -all_outcomes())

model_spec <- logistic_reg(engine = "glmnet", mode = "classification",
  penalty = tune(), mixture = tune())

wf <- workflow() %>%
  add_model(model_spec) %>%
  add_recipe(recipe_spec)
```

Tune the penalty and mixture hyperparameters using Bayesian hyperparameter optimization:

```
parameters <- extract_parameter_set_dials(wf) %>%
  update(penalty = penalty(c(-4, -1)))
tune_wf <- tune_bayes(wf, resamples = resamples, metrics = cv_metrics,
  param_info = parameters, iter = 25)
```

```
! No improvement for 10 iterations; returning current results.
```

The autoplot of the `tune_bayes` object (Figure 25.1) shows the ROC-AUC for different values of the penalty and mixture hyperparameters. We can see that the best `roc_auc` is obtained with penalty and mixture values inside the tuning range. We don't need to adjust the sampling ranges for the hyperparameters.

```
autoplot(tune_wf)
```

Figure 25.1: Autoplot shows the ROC-AUC for different values of the penalty and mixture hyperparameters.

Finalize the workflow:

```
best_parameter <- select_best(tune_wf, metric = "roc_auc")
best_wf <- finalize_workflow(wf, best_parameter)
```

The best `roc_auc` is obtained with a penalty of `best_parameter['penalty'] =` 0.0632250887522679 and a mixture of `best_parameter['mixture'] =` 0.760072101599665.

Use the tuned workflow for cross-validation and training the final model using the full dataset:

```
result_cv <-  fit_resamples(best_wf, resamples,
  metrics = cv_metrics, control = cv_control)
fitted_model <- best_wf %>% fit(train_data)
```

Estimate model performance using the cross-validation results and the holdout data:

Table 25.1: Model performance metrics

result	accuracy	roc_auc
Cross-validation	0.802	0.757
Holdout	0.835	0.741

```
cv_results <- collect_metrics(result_cv) %>%
  select(.metric, mean) %>%
  rename(.estimate = mean) %>%
  mutate(result = "Cross-validation")
holdout_predictions <- augment(fitted_model, new_data = holdout_data)
holdout_results <-  bind_rows(
  c(roc_auc(holdout_predictions, Loan_Status, .pred_Yes,
      event_level = "second")),
  c(accuracy(holdout_predictions, Loan_Status, .pred_class))
) %>%
  select(-.estimator) %>%
  mutate(result = "Holdout")
```

The performance metrics are summarized in the following table.

```
bind_rows(
  cv_results,
  holdout_results
) %>%
  pivot_wider(names_from = .metric, values_from = .estimate) %>%
  kableExtra::kbl(caption = "Model performance metrics", digits = 3) %>%
  kableExtra::kable_styling(full_width = FALSE)
```

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
library(tidyverse)
library(tidymodels)
file <- "https://gedeck.github.io/DS-6030/datasets/loan_prediction.csv"
data <- read_csv(file, show_col_types = FALSE) %>%
```

```
  drop_na() %>%
  mutate(
    Gender = as.factor(Gender),
    Married = as.factor(Married),
    Dependents = gsub("\\+", "", Dependents) %>% as.numeric(),
    Education = as.factor(Education),
    Self_Employed = as.factor(Self_Employed),
    Credit_History = as.factor(Credit_History),
    Property_Area = as.factor(Property_Area),
    Loan_Status = factor(Loan_Status, levels = c("N", "Y"),
      labels = c("No", "Yes"))
  ) %>%
  select(-Loan_ID)
set.seed(123)
data_split <- initial_split(data, prop = 0.8, strata = Loan_Status)
train_data <- training(data_split)
holdout_data <- testing(data_split)

resamples <- vfold_cv(train_data, v = 10, strata = Loan_Status)
cv_metrics <- metric_set(roc_auc, accuracy)
cv_control <- control_resamples(save_pred = TRUE)
formula <- Loan_Status ~ Gender + Married + Dependents + Education +
  Self_Employed + ApplicantIncome + CoapplicantIncome + LoanAmount +
  Loan_Amount_Term + Credit_History + Property_Area
recipe_spec <- recipe(formula, data = train_data) %>%
  step_dummy(all_nominal(), -all_outcomes())

model_spec <- logistic_reg(engine = "glmnet", mode = "classification",
  penalty = tune(), mixture = tune())

wf <- workflow() %>%
  add_model(model_spec) %>%
  add_recipe(recipe_spec)
parameters <- extract_parameter_set_dials(wf) %>%
  update(penalty = penalty(c(-4, -1)))
tune_wf <- tune_bayes(wf, resamples = resamples, metrics = cv_metrics,
  param_info = parameters, iter = 25)
autoplot(tune_wf)
best_parameter <- select_best(tune_wf, metric = "roc_auc")
best_wf <- finalize_workflow(wf, best_parameter)
result_cv <-  fit_resamples(best_wf, resamples,
  metrics = cv_metrics, control = cv_control)
```

```r
fitted_model <- best_wf %>% fit(train_data)
cv_results <- collect_metrics(result_cv) %>%
  select(.metric, mean) %>%
  rename(.estimate = mean) %>%
  mutate(result = "Cross-validation")
holdout_predictions <- augment(fitted_model, new_data = holdout_data)
holdout_results <-  bind_rows(
  c(roc_auc(holdout_predictions, Loan_Status, .pred_Yes,
      event_level = "second")),
  c(accuracy(holdout_predictions, Loan_Status, .pred_class))
) %>%
  select(-.estimator) %>%
  mutate(result = "Holdout")
bind_rows(
  cv_results,
  holdout_results
) %>%
  pivot_wider(names_from = .metric, values_from = .estimate) %>%
  kableExtra::kbl(caption = "Model performance metrics", digits = 3) %>%
  kableExtra::kable_styling(full_width = FALSE)
```

26 Threshold selection

In this example, we will

- load and preprocess data
- define a workflow
- use cross-validation to determine a threshold using the F-statistic
- train the final model
- evaluate the model using cross-validation and holdout data
- predict

using the Loan prediction dataset to illustrate the whole process.

Load the required packages:

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

Load and preprocess the data:

```
file <- "https://gedeck.github.io/DS-6030/datasets/loan_prediction.csv"
data <- read_csv(file, show_col_types = FALSE) %>%
  drop_na() %>%
  mutate(
    Gender = as.factor(Gender),
    Married = as.factor(Married),
    Dependents = gsub("\\+", "", Dependents) %>% as.numeric(),
    Education = as.factor(Education),
    Self_Employed = as.factor(Self_Employed),
    Credit_History = as.factor(Credit_History),
    Property_Area = as.factor(Property_Area),
    Loan_Status = factor(Loan_Status, levels = c("N", "Y"),
      labels = c("No", "Yes"))
  ) %>%
  select(-Loan_ID)
```

Split dataset into training and holdout data, prepare for cross-validation:

```
set.seed(123)
data_split <- initial_split(data, prop = 0.8, strata = Loan_Status)
train_data <- training(data_split)
holdout_data <- testing(data_split)

resamples <- vfold_cv(train_data, v = 10, strata = Loan_Status)
cv_metrics <- metric_set(roc_auc, accuracy)
cv_control <- control_resamples(save_pred = TRUE)
```

Define the recipe, the model specification (elasticnet logistic regression), and combine them into a workflow:

```
formula <- Loan_Status ~ Gender + Married + Dependents + Education +
  Self_Employed + ApplicantIncome + CoapplicantIncome + LoanAmount +
  Loan_Amount_Term + Credit_History + Property_Area
recipe_spec <- recipe(formula, data = train_data) %>%
  step_dummy(all_nominal(), -all_outcomes())

model_spec <- logistic_reg(engine = "glm", mode = "classification")

wf <- workflow() %>%
  add_model(model_spec) %>%
  add_recipe(recipe_spec)
```

Use the workflow for cross-validation and training the final model using the full dataset:

```
result_cv <-  fit_resamples(wf, resamples, metrics = cv_metrics,
  control = cv_control)
fitted_model <- wf %>% fit(train_data)
```

Estimate model performance using the cross-validation results and the holdout data:

```
cv_results <- collect_metrics(result_cv) %>%
  select(.metric, mean) %>%
  rename(.estimate = mean) %>%
  mutate(result = "Cross-validation", threshold = 0.5)
holdout_predictions <- augment(fitted_model, new_data = holdout_data)
holdout_results <-  bind_rows(
  c(roc_auc(holdout_predictions, Loan_Status, .pred_Yes,
      event_level = "second")),
  c(accuracy(holdout_predictions, Loan_Status, .pred_class))) %>%
```

```
  select(-.estimator) %>%
  mutate(result = "Holdout", threshold = 0.5)
```

```
performance <- probably::threshold_perf(
  result_cv %>% collect_predictions(),
  Loan_Status, .pred_Yes, seq(0.1, 0.9, 0.01), event_level = "second",
  metrics = metric_set(j_index, f_meas, kap))
```

```
max_values <- performance %>%
  arrange(desc(.threshold)) %>%
  group_by(.metric) %>%
  filter(.estimate == max(.estimate)) %>%
  filter(row_number() == 1)
ggplot(performance, aes(x = .threshold, y = .estimate, color = .metric)) +
  geom_line() +
  geom_vline(data = max_values, aes(xintercept = .threshold, color = .metric))
```

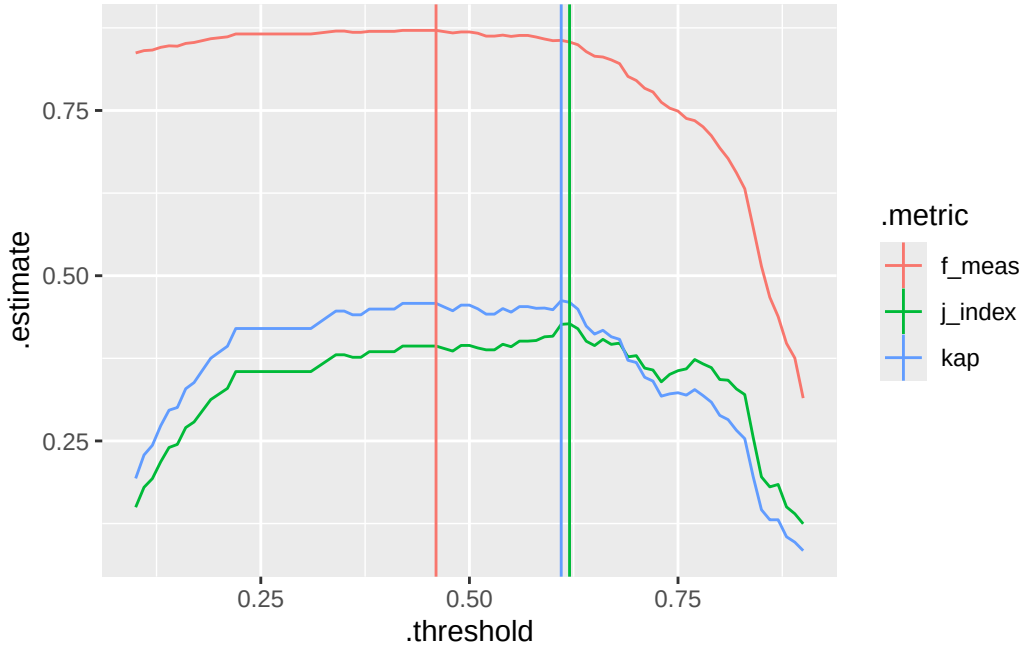


Figure 26.1: Performance metrics as a function of the classification threshold.

We decide to select the threshold that maximizes the F-measure:

```
threshold <- max_values %>%
  filter(.metric == "f_meas") %>%
  pull(.threshold)
```

We can now calculate the performance metrics using predictions at the selected threshold.

```
cv_predictions <- collect_predictions(result_cv) %>%
  mutate(
    .pred_class = factor(ifelse(.pred_Yes >= threshold, "Yes", "No"))
  )
cv_threshold_results <-  bind_rows(
  c(accuracy(cv_predictions, Loan_Status, .pred_class))
) %>%
  select(-.estimator) %>%
  mutate(result = "Cross-validation", threshold = threshold)
holdout_predictions <- augment(fitted_model, new_data = holdout_data) %>%
  mutate(
    .pred_class = factor(ifelse(.pred_Yes >= threshold, "Yes", "No"))
  )
holdout_threshold_results <-  bind_rows(
  c(accuracy(holdout_predictions, Loan_Status, .pred_class))
) %>%
  select(-.estimator) %>%
  mutate(result = "Holdout", threshold = threshold)
```

The performance metrics are summarized in the following table.

```
bind_rows(
  cv_results,
  holdout_results,
  cv_threshold_results,
  holdout_threshold_results,
) %>%
  pivot_wider(names_from = .metric, values_from = .estimate) %>%
  kableExtra::kbl(caption = "Model performance metrics", digits = 3) %>%
  kableExtra::kable_styling(full_width = FALSE)
```

We can see that the reduced threshold leads to a higher accuracy.

Table 26.1: Model performance metrics

result	threshold	accuracy	roc_auc
Cross-validation	0.50	0.799	0.752
Holdout	0.50	0.825	0.733
Cross-validation	0.46	0.802	NA
Holdout	0.46	0.835	NA

Code

The code of this chapter is summarized here.

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, autodep = TRUE,
  fig.align = "center")
library(tidyverse)
library(tidymodels)
file <- "https://gedeck.github.io/DS-6030/datasets/loan_prediction.csv"
data <- read_csv(file, show_col_types = FALSE) %>%
  drop_na() %>%
  mutate(
    Gender = as.factor(Gender),
    Married = as.factor(Married),
    Dependents = gsub("\\+", "", Dependents) %>% as.numeric(),
    Education = as.factor(Education),
    Self_Employed = as.factor(Self_Employed),
    Credit_History = as.factor(Credit_History),
    Property_Area = as.factor(Property_Area),
    Loan_Status = factor(Loan_Status, levels = c("N", "Y"),
      labels = c("No", "Yes"))
  ) %>%
  select(-Loan_ID)
set.seed(123)
data_split <- initial_split(data, prop = 0.8, strata = Loan_Status)
train_data <- training(data_split)
holdout_data <- testing(data_split)

resamples <- vfold_cv(train_data, v = 10, strata = Loan_Status)
cv_metrics <- metric_set(roc_auc, accuracy)
cv_control <- control_resamples(save_pred = TRUE)
formula <- Loan_Status ~ Gender + Married + Dependents + Education +
  Self_Employed + ApplicantIncome + CoapplicantIncome + LoanAmount +
  Loan_Amount_Term + Credit_History + Property_Area
```

```
recipe_spec <- recipe(formula, data = train_data) %>%
  step_dummy(all_nominal(), -all_outcomes())

model_spec <- logistic_reg(engine = "glm", mode = "classification")

wf <- workflow() %>%
  add_model(model_spec) %>%
  add_recipe(recipe_spec)
result_cv <-  fit_resamples(wf, resamples, metrics = cv_metrics,
  control = cv_control)
fitted_model <- wf %>% fit(train_data)
cv_results <- collect_metrics(result_cv) %>%
  select(.metric, mean) %>%
  rename(.estimate = mean) %>%
  mutate(result = "Cross-validation", threshold = 0.5)
holdout_predictions <- augment(fitted_model, new_data = holdout_data)
holdout_results <-  bind_rows(
  c(roc_auc(holdout_predictions, Loan_Status, .pred_Yes,
      event_level = "second")),
  c(accuracy(holdout_predictions, Loan_Status, .pred_class))) %>%
  select(-.estimator) %>%
  mutate(result = "Holdout", threshold = 0.5)
performance <- probably::threshold_perf(
  result_cv %>% collect_predictions(),
  Loan_Status, .pred_Yes, seq(0.1, 0.9, 0.01), event_level = "second",
  metrics = metric_set(j_index, f_meas, kap))
max_values <- performance %>%
  arrange(desc(.threshold)) %>%
  group_by(.metric) %>%
  filter(.estimate == max(.estimate)) %>%
  filter(row_number() == 1)
ggplot(performance, aes(x = .threshold, y = .estimate, color = .metric)) +
  geom_line() +
  geom_vline(data = max_values, aes(xintercept = .threshold, color = .metric))
threshold <- max_values %>%
  filter(.metric == "f_meas") %>%
  pull(.threshold)
cv_predictions <- collect_predictions(result_cv) %>%
  mutate(
    .pred_class = factor(ifelse(.pred_Yes >= threshold, "Yes", "No"))
  )
cv_threshold_results <-  bind_rows(
```

```
  c(accuracy(cv_predictions, Loan_Status, .pred_class))
) %>%
  select(-.estimator) %>%
  mutate(result = "Cross-validation", threshold = threshold)
holdout_predictions <- augment(fitted_model, new_data = holdout_data) %>%
  mutate(
    .pred_class = factor(ifelse(.pred_Yes >= threshold, "Yes", "No"))
  )
holdout_threshold_results <-  bind_rows(
  c(accuracy(holdout_predictions, Loan_Status, .pred_class))
) %>%
  select(-.estimator) %>%
  mutate(result = "Holdout", threshold = threshold)
bind_rows(
  cv_results,
  holdout_results,
  cv_threshold_results,
  holdout_threshold_results,
) %>%
  pivot_wider(names_from = .metric, values_from = .estimate) %>%
  kableExtra::kbl(caption = "Model performance metrics", digits = 3) %>%
  kableExtra::kable_styling(full_width = FALSE)
```

A Models

As we've seen in Chapter 8 and Chapter 10, a model requires first defining the model type (e.g. `linear_reg`) and then select a suitable engine (e.g. `lm`).

The following packages provide model engines for classification, regression, and censored regression compatible with the `parsnip` format:

- `parsnip`: parsnip.tidymodels.org
- `modeltime`: business-science.github.io/modeltime/ for time series forecasting

Use the command `show_engines(...)` to get an overview of all available engines for a given model type.

```
library(parsnip)
show_engines("linear_reg")
```

```
# A tibble: 8 x 2
  engine   mode
  <chr>    <chr>
1 lm       regression
2 glm      regression
3 glmnet   regression
4 stan     regression
5 spark    regression
6 keras    regression
7 brulee   regression
8 quantreg quantile regression
```

Some of the model types have common tunable parameters, e.g. the number of nearest neighbors in a k-nearest neighbor model. These parameters can be set when defining the model. `Parsnip` will translate these into the engine specific paramters. For example, the following code defines a k-nearest neighbor model with 5 neighbors and a distance weighting function:

```
nearest_neighbor(mode = "regression", neighbors = 5,
  weight_func = "triangular") %>%
  set_engine("kknn") %>%
  translate()
```

```
K-Nearest Neighbor Model Specification (regression)

Main Arguments:
  neighbors = 5
  weight_func = triangular

Computational engine: kknn

Model fit template:
kknn::train.kknn(formula = missing_arg(), data = missing_arg(),
    ks = min_rows(5, data, 5), kernel = "triangular")
```

The `translate` function returns information about how the actual engine is called. In this case, the `neighbors` parameter is mapped to `ks=min_rows(5, data, 5)` and `weight_func` is mapped to `kernel="triangular"`. This information will be useful when you want to understand more about the engine and read the documentation of the engine package.

In the following, we cover a selection of models and engines relevant for DS-6030. A full list of all available parsnip models can be found here:https://www.tidymodels.org/find/parsnip/

A.1 Non-informative model `null_model` (regression and classification)

While not an actual model, training and evaluating a non-informative model is a good baseline to compare other models against. A non-informative model always predicts the mean of the response variable for regression models and the most frequent class for classification models. See https://parsnip.tidymodels.org/reference/null_model.html for details.

```
null_model(mode = "regression") %>%
  set_engine("parsnip")
```

```
Null Model Specification (regression)

Computational engine: parsnip
```

```
null_model(mode = "classification") %>%
  set_engine("parsnip")
```

```
Null Model Specification (classification)

Computational engine: parsnip
```

A.2 Linear regression models `linear_reg` (regression)

See https://parsnip.tidymodels.org/reference/linear_reg.html for details.

A.2.1 `lm` engine (default)

```
linear_reg(mode = "regression") %>%
  set_engine("lm")
```

```
Linear Regression Model Specification (regression)

Computational engine: lm
```

No tunable parameters.

A.2.2 `glm` engine (generalized linear model)

The `glm` engine is a more flexible version of the `lm` engine. It allows to specify the distribution of the response variable (e.g. `gaussian` for linear regression, `binomial` for logistic regression, `poisson` for count data, etc.) and the link function (e.g. `identity` for linear regression, `logit` for logistic regression, `log` for count data, etc.).

```
linear_reg(mode = "regression") %>%
  set_engine("glm")
```

```
Linear Regression Model Specification (regression)

Computational engine: glm
```

No tunable parameters.

> Useful to know
>
> When using the `glm` or `glmnet` engine, you might come across this warning:
>
> ```
> Warning: glm.fit: fitted probabilities numerically 0 or 1
>     occurred from the logistic model
> ```
>
> In general, you can ignore it. It means that the model is very certain about the predicted

class.

A.2.3 `glmnet` engine (regularized linear regression)

```
linear_reg(mode = "regression") %>%
  set_engine("glmnet")
```

```
Linear Regression Model Specification (regression)

Computational engine: glmnet
```

`glmnet` supports L1 and L2 regularization. Here is an example with a mixture of L1 and L2 regularization (elastic net) and a regularization parameter of 0.01:

```
linear_reg(mode = "regression", penalty = 0.01, mixture = 0.5) %>%
  set_engine("glmnet")
```

```
Linear Regression Model Specification (regression)

Main Arguments:
  penalty = 0.01
  mixture = 0.5

Computational engine: glmnet
```

- See Chapter 21 for more details
- https://parsnip.tidymodels.org/reference/glmnet-details.html

A.3 Partial least squares regression `pls` (regression)

See https://parsnip.tidymodels.org/reference/pls.html for details.

A.3.1 `mixOmics` engine (default)

This engine requires installation of the `mixOmics` package. See http://mixomics.org/ for details. Use the following to install the package:

```
if (!require("BiocManager", quietly = TRUE))
  install.packages("BiocManager", repos = "http://cran.us.r-project.org")
if (!require("plsmod", quietly = TRUE))
  install.packages("plsmod", repos = "http://cran.us.r-project.org")

BiocManager::install("mixOmics")
```

Once the package is installed, you can use the `mixOmics` engine:

```
library(plsmod)

pls(mode = "regression") %>%
  set_engine("mixOmics")
```

```
PLS Model Specification (regression)

Computational engine: mixOmics
```

The engine has two tunable parameters, `num_comp` and `predictor_prop`.

A.4 Logistic regression models `logistic_reg` (classification)

See https://parsnip.tidymodels.org/reference/logistic_reg.html for details.

A.4.1 `glm` engine (default)

```
logistic_reg(mode = "classification") %>%
  set_engine("glm")
```

```
Logistic Regression Model Specification (classification)

Computational engine: glm
```

See comments above for `glm` engine in linear regression models (Section A.2.2).

A.4.2 `glmnet` engine (regularized logistic regression)

```
logistic_reg(mode = "classification") %>%
  set_engine("glmnet")
```

```
Logistic Regression Model Specification (classification)

Computational engine: glmnet
```

See comments above for `glmnet` engine in linear regression models. (Section A.2.3).

A.5 Nearest Neighbor models (classification and regression)

Nearest neighbor models can be used for classification and regression. It is therefore necessary to specify the mode of the model. You must choose classification or regression, i.e. `mode = "regression"` or `mode = "classification"`. See https://parsnip.tidymodels.org/reference/nearest_neighbor.html for details.

A.5.1 `kknn` engine (default)

`kknn` is currently the only supported engine. It supports both classification and regression. See https://parsnip.tidymodels.org/reference/kknn.html for details.

Use `mode` to specify either a classification or regression model:

```
nearest_neighbor(mode = "classification") %>%
  set_engine("kknn")
```

```
K-Nearest Neighbor Model Specification (classification)

Computational engine: kknn
```

```
nearest_neighbor(mode = "regression") %>%
  set_engine("kknn")
```

```
K-Nearest Neighbor Model Specification (regression)

Computational engine: kknn
```

The engine has several tunable parameters. Here is an example with a k-nearest neighbor model with 5 neighbors and a distance weighting function:

```
nearest_neighbor(mode = "regression", neighbors = 5,
  weight_func = "triangular") %>%
  set_engine("kknn")
```

```
K-Nearest Neighbor Model Specification (regression)

Main Arguments:
  neighbors = 5
  weight_func = triangular

Computational engine: kknn
```

A *triangular* weight function applies more weight to neighbors that are closer to the observation. There are other options; *rectangular* weights all neighbors equally.

See https://rdrr.io/cran/kknn/man/train.kknn.html for details about the `kknn` package.

If you cannot install the `kknn` package from CRAN, you can install it directly from GitHub:

```
install.packages("devtools")
devtools::install_github("KlausVigo/kknn")
```

A.6 Linear discriminant analysis `discrim_linear` (classification)

See https://parsnip.tidymodels.org/reference/discrim_linear.html for details.

A.6.1 `MASS` engine (default)

You will need to load the `discrim` package to use this engine.

```
library(discrim)

discrim_linear(mode = "classification") %>%
  set_engine("MASS")
```

```
Linear Discriminant Model Specification (classification)

Computational engine: MASS
```

No tunable parameters.

A.7 Quadratic discriminant analysis `discrim_quad` (classification)

See https://parsnip.tidymodels.org/reference/discrim_quad.html for details.

A.7.1 `MASS` engine (default)

You will need to load the `discrim` package to use this engine.

```
library(discrim)

discrim_quad(mode = "classification") %>%
  set_engine("MASS")
```

```
Quadratic Discriminant Model Specification (classification)

Computational engine: MASS
```

No tunable parameters.

A.8 Generalized additive models `gen_additive_mod` (regression and classification)

See https://parsnip.tidymodels.org/reference/gen_additive_mod.html for details.

A.8.1 `mgcv` engine (default)

You will need to load the `mgcv` package to use this engine.

```
library(mgcv)
```

```
Loading required package: nlme
```

```
This is mgcv 1.9-3. For overview type 'help("mgcv-package")'.
```

```
gen_additive_mod(mode = "regression") %>%
  set_engine("mgcv")
```

```
GAM Model Specification (regression)

Computational engine: mgcv
```

The model has two tuning parameters:

- `select_features` (default FALSE): if TRUE, the model will add a penalty term so that terms can be penalized to zero.
- `adjust_deg_free` (default 1): level of penalization; higher values lead to more penalization.

See Chapter 22 for more details.

A.9 Decision tree models `decision_tree` (classification, regression, and censored regression)

See https://parsnip.tidymodels.org/reference/decision_tree.html for details.

A.9.1 `rpart` engine (default)

```
decision_tree(mode = "regression") %>%
  set_engine("rpart")
```

```
Decision Tree Model Specification (regression)

Computational engine: rpart
```

The model has three tuning parameters:

- `tree_depth` (default 30): maximum depth of the tree
- `min_n` (default 2): minimum number of observations in a node
- `cost_complexity` (default 0.01): complexity parameter; higher values lead to simpler trees

A.9.2 `partykit` engine

```
decision_tree(mode = "classification") %>%
  set_engine("partykit")
```

```
! parsnip could not locate an implementation for `decision_tree` classification
  model specifications using the `partykit` engine.
i The parsnip extension package bonsai implements support for this
  specification.
i Please install (if needed) and load to continue.
```

```
Decision Tree Model Specification (classification)

Computational engine: partykit
```

The model has three tuning parameters:

- `tree_depth`: maximum depth of the tree, by default no restriction
- `min_n` (default 20): minimum number of observations in a node
- `mtry`: random number of predictors to try at each split, by default no restriction

The `partykit` engine requires installation of the `partykit` and `bonsai` packages.

A.10 Ensemble models I `bag_tree` (classification and regression)

See https://parsnip.tidymodels.org/reference/bag_tree.html for details.

A.10.1 `rpart` engine (default)

To use this engine, you need to load the `baguette` package.

```
library(baguette)

bag_tree(mode = "classification") %>%
  set_engine("rpart")
```

```
Bagged Decision Tree Model Specification (classification)

Main Arguments:
  cost_complexity = 0
  min_n = 2

Computational engine: rpart
```

The model has four tuning parameters:

- `tree_depth` (default 30): maximum depth of the tree
- `min_n` (default 2): minimum number of observations in a node
- `cp` (default 0.01): complexity parameter; higher values lead to simpler trees
- `class_cost` (default NULL): cost of misclassifying each class; if NULL, the cost is set to 1 for all classes

A.11 Ensemble models II `boost_tree` (classification and regression)

See https://parsnip.tidymodels.org/reference/boost_tree.html for details.

A.11.1 `xgboost` engine (default)

To use this engine, you need to have the `xgboost` package installed.

```
boost_tree(mode = "classification") %>%
  set_engine("xgboost")
```

```
Boosted Tree Model Specification (classification)

Computational engine: xgboost
```

The model has eight tuning parameters. Here is an example with a model with 100 trees, a learning rate of 0.1, and a maximum tree depth of 3:

```
boost_tree(mode = "classification", trees = 100, learn_rate = 0.1,
  tree_depth = 3) %>%
  set_engine("xgboost")
```

```
Boosted Tree Model Specification (classification)

Main Arguments:
  trees = 100
  tree_depth = 3
  learn_rate = 0.1

Computational engine: xgboost
```

For details see https://parsnip.tidymodels.org/reference/details_boost_tree_xgboost.html.

A.11.2 `lightgbm` engine

To use this engine, you need to have the `lightgbm` and the `bonsai` packages installed.

```
library(bonsai)
boost_tree(mode = "regression") %>%
  set_engine("lightgbm")
```

```
Boosted Tree Model Specification (regression)

Computational engine: lightgbm
```

This model has six tuning parameters. Here is an example with a model with 100 trees, a learning rate of 0.1, and a maximum tree depth of 3:

```
boost_tree(mode = "regression", trees = 100, learn_rate = 0.1,
  tree_depth = 3) %>%
  set_engine("lightgbm")
```

```
Boosted Tree Model Specification (regression)

Main Arguments:
  trees = 100
  tree_depth = 3
  learn_rate = 0.1

Computational engine: lightgbm
```

For details see https://parsnip.tidymodels.org/reference/details_boost_tree_lightgbm.html.

A.12 Ensemble models III `rand_forest` (classification and regression)

See https://parsnip.tidymodels.org/reference/rand_forest.html for details.

A.12.1 `ranger` engine (default)

To use this engine, you need to have the `ranger` package installed.

```
rand_forest(mode = "classification") %>%
  set_engine("ranger")
```

```
Random Forest Model Specification (classification)

Computational engine: ranger
```

The model has three tuning parameters. Here is an example with a model with 100 trees, a minimum node size of 5, and number of randomly selected predictors at each split to 3:

```
rand_forest(mode = "classification", trees = 100, min_n = 5, mtry = 3) %>%
  set_engine("ranger")
```

```
Random Forest Model Specification (classification)

Main Arguments:
  mtry = 3
```

```
  trees = 100
  min_n = 5

Computational engine: ranger
```

Default values are:

- `mtry`: number of randomly selected predictors at each split; default is the square root of the number of predictors
- `min_n`: minimum node size; default is 5 for regression and 10 for classification

If you want to extract information about variable importance from the model, you need to set `importance="impurity"` in the `ranger` engine:

```
rand_forest(mode = "classification", trees = 100, min_n = 5, mtry = 3) %>%
  set_engine("ranger", importance = "impurity")
```

```
Random Forest Model Specification (classification)

Main Arguments:
  mtry = 3
  trees = 100
  min_n = 5

Engine-Specific Arguments:
  importance = impurity

Computational engine: ranger
```

A.12.2 `randomForest` engine

To use this engine, you need to have the `randomForest` package installed.

```
rand_forest(mode = "regression") %>%
  set_engine("randomForest")
```

```
Random Forest Model Specification (regression)

Computational engine: randomForest
```

The `ranger` package is considerably faster than `randomForest`, so we recommend using `ranger` instead. However, if you want to use `randomForest`, here is an example with a model with 100 trees, a minimum node size of 5, and number of randomly selected predictors at each split to 3:

```
rand_forest(mode = "regression", trees = 100, min_n = 5, mtry = 3) %>%
  set_engine("randomForest")
```

```
Random Forest Model Specification (regression)

Main Arguments:
  mtry = 3
  trees = 100
  min_n = 5

Computational engine: randomForest
```

A.13 Support vector machines I `svm_linear` (classification and regression)

See https://parsnip.tidymodels.org/reference/svm_linear.html for details.

For SVM models, it is recommended to normalize the predictors to a mean of zero and a variance of one.

A.13.1 `LiblineaR` engine (default)

To use this engine, you need to have the `LiblineaR` package installed.

```
svm_linear(mode = "classification") %>%
  set_engine("LiblineaR")
```

```
Linear Support Vector Machine Model Specification (classification)

Computational engine: LiblineaR
```

The model has two tuning parameters. Here is an example with a model with a cost of 0.1 and a margin of 1:

```
svm_linear(mode = "classification", cost = 0.1, margin = 1) %>%
  set_engine("LiblineaR")
```

```
Linear Support Vector Machine Model Specification (classification)

Main Arguments:
  cost = 0.1
  margin = 1

Computational engine: LiblineaR
```

More details: https://parsnip.tidymodels.org/reference/details_svm_linear_LiblineaR.html

A.13.2 `kernlab` engine

To use this engine, you need to have the `kernlab` package installed.

```
svm_linear(mode = "regression") %>%
  set_engine("kernlab")
```

```
Linear Support Vector Machine Model Specification (regression)

Computational engine: kernlab
```

The model has two tuning parameters. Here is an example with a model with a cost of 0.1 and a margin of 1:

```
svm_linear(mode = "regression", cost = 0.1, margin = 1) %>%
  set_engine("kernlab")
```

```
Linear Support Vector Machine Model Specification (regression)

Main Arguments:
  cost = 0.1
  margin = 1

Computational engine: kernlab
```

The `margin` parameter is not used in regression models.

A.14 Support vector machines II `svm_poly` (classification and regression)

See https://parsnip.tidymodels.org/reference/svm_poly.html for details.

A.14.1 `kernlab` engine (default)

To use this engine, you need to have the `kernlab` package installed.

```
svm_poly(mode = "classification") %>%
  set_engine("kernlab")
```

```
Polynomial Support Vector Machine Model Specification (classification)

Computational engine: kernlab
```

The model has four tuning parameters. Here is an example with a model with a cost of 0.1, a scale_factor of 0.75, and a degree of 2:

```
svm_poly(mode = "classification", cost = 0.1, scale_factor = 0.75,
  degree = 2) %>%
  set_engine("kernlab")
```

```
Polynomial Support Vector Machine Model Specification (classification)

Main Arguments:
  cost = 0.1
  degree = 2
  scale_factor = 0.75

Computational engine: kernlab
```

For regression models, you can also tune the `margin` parameter.

A.15 Support vector machines III `svm_rbf` (classification and regression)

See https://parsnip.tidymodels.org/reference/svm_rbf.html for details.

A.15.1 `kernlab` engine (default)

To use this engine, you need to have the `kernlab` package installed.

```
svm_rbf(mode = "classification") %>%
  set_engine("kernlab")
```

```
Radial Basis Function Support Vector Machine Model Specification (classification)

Computational engine: kernlab
```

The model has three tuning parameters. Here is an example with a model with a cost of 0.1 and a rbf_sigma of 0.75:

```
svm_rbf(mode = "classification", cost = 0.1, rbf_sigma = 0.75) %>%
  set_engine("kernlab")
```

```
Radial Basis Function Support Vector Machine Model Specification (classification)

Main Arguments:
  cost = 0.1
  rbf_sigma = 0.75

Computational engine: kernlab
```

For regression models, you can also tune the `margin` parameter.

B Defining models using formulae

B.1 Linear models

In *R*, statistical models are usually defined using a formula. Here is an example:

```
y ~ x1 + x2 + x3
```

This formula describes a linear model of the form:

$$y = c_1 x1 + c_2 x2 + c_3 x3 + y_0$$

The outcome y is a linear combination of the predictors $x1$, $x2$, and $x3$ with coefficients c_1, c_2, and c_3. y_0 is a constant intercept. Note how the intercept is not included in the model definition. If you want to make the intercept explicit you can write the following formula where `1` represents the constant, but modeled intercept y_0.

```
y ~ x1 + x2 + x3 + 1
```

To exclude the intercept and fit a linear model without intercept, use one of the following options:

```
y ~ x1 + x2 + x3 - 1
y ~ x1 + x2 + x3 + 0
```

If your variable names contain spaces, you need to surround the name in the formula with the backtick character `` ` ``.

```
quality ~ `fixed acidity` + `volatile acidity` + chlorides
```

Here, "fixed acidity" and "volatile acidity" are names of variables or columns in a data frame.

You will frequently come across formulas like `y ~ .` in the following linear regression model:

```
model <- lm(y ~ ., data=df)
```

Here, the `.` stands for "*all columns not otherwise in the formula*". For example, if the tibble (or data frame) contains columns `a`, `b`, `c`, and `y`, then `y ~ .` is equivalent to the explicit formula `y ~ a + b + c`.

> While it might be tempting to use the shortcut `y ~ .`, it is better to be explicit and list all terms to ensure reproducibility. Even if we know exactly what is included in a dataset when we develop the model, data can change over time. For example, if data are downloaded from an external source and columns are added, model results will change. Another source of issues will be columns added to the dataset during exploratory data analysis.

> Useful to know
>
> With *Tidymodels*, it's possible to use the full capabilities of formulas described here only if you directly fit a model. With workflows, recipes only accept equations like the ones shown in this section. Interactions and transformations must be defined using preprocessing functions like `step_interact` or `step_sqrt`.[1]

B.2 Linear models with interactions

The models in the previous section included only main effects. It is easy to extend the model definition to include interaction terms. Interactions are identified using `:`, e.g. `a:b` represents interaction of `a` and `b`. Here is an example:

```
y ~ a + b + c + a:b
```

This represents the following linear model:

$$y = c_1 a + c_2 b + c_3 c + c_4 ab + y_0$$

The formula can be written in a more concise way using `*`:

```
y ~ a*b + c
```

[1] You will get the following error message when you define add a formula to a recipe that contains interactions or transformations: `! No in-line functions should be used here; use steps to define baking actions.`

The term `a*b` is expanded to `a + b + a:b`.

If you want to include interaction terms between several predictors, the following variations can be used:

```
y ~ (a + b + c)^2
y ~ (a + b + c)**2
```

It expands to `y ~ a + b + c + a:b + a:c + b:c`.

You can extend this expression to include interactions of more than two variables. For example:

```
y ~ (a + b + c)^3
```

is equivalent to

```
y ~ a + b + c + a:b + a:c + b:c + a:b:C
```

> You can only specify interactions term in `recipes` using the `step_interact` function. The function uses the same syntax, but interprets the formula slightly different. For example, you can specify `a*b` which is normally interpreted as `a + b + a:b`. `recipes` will add the interaction `a:b` but not the main effects. In practice, this makes no difference, as the main effects must be included in the recipe's formula and therefore will be present. See Section 7.7 for more details.

B.3 Linear models with transformations

Formula can also be used to define variable transformations. For example,

```
log(y) ~ a + log(x)
```

defines the following linear model:

$$\log y = c_1 a + c_2 \log x + y_0$$

Here, we take the logarithm for both the outcome y and the predictor x and train a linear model using the transformed variables.

Not all transformations can be easily expressed. Consider the following linear model:

$$y = c_1 x^2 + y_0$$

One could be tempted to write this as `y ~ x^2`. This is however interpreted as `y ~ x*x` which is equivalent to `y ~ x`. The correct way to formulate this model is to use the `I()` function.

```
y ~ I(x^2)
```

Whatever is inside the brackets, is evaluated as an expression.

Another case for using the `I()` function is this linear model:

$$y = c_1 a + c_2(b + c) + y_0$$

To express this in a formula use:

```
y ~ a + I(b + c)
```

> As we've seen for interactions, you can also only specify transformations in `recipes` using preprocessing functions. See Chapter 7 for more details and examples.

B.4 Miscellaneous

There are additional operators that you may come across. The `%in%` or/operator `expands`a/`bor`a %in% b`to`a + a:b‘.

The `-` operator allows to remove terms. E.g. the following formula are identical.

```
y ~ (a + b + c)^2 - a:b
y ~ a + b + c + a:b + a:c + b:c - a:b
y ~ a + b + c + a:c + b:C
```

> Useful to know
>
> This approach to define statistical models in *R*, was developed by Wilkinson and Rogers in 1973 (G. N. Wilkinson and Rogers 1973). It is also available in *Python* using the `patsy` package. `Patsy` is used by the `statsmodels` package and a few other *Python* packages.

C Markdown and R Markdown

Markdown allows creating formatted documents using a plain text formatting syntax. It is widely used across the web for writing content due to its simplicity and readability. Created in 2004, it uses simple, intuitive characters—like asterisks for bold or hashes for headers—to add structure, which is then parsed into HTML for web publishing, document creation, or note-taking. It is widely favored for its portability and speed. You likely have already used *Markdown* in various platforms, such as GitHub, Stack Overflow, or blogging sites. Simple formatting examples include:

- `**bold text**` for **bold text**
- `*italic text*` for *italic text*
- `# Header 1` for a top-level header
- `## Header 2` for a second-level header
- `- List item` for a bullet point

and so on. There are many resources online to learn about *Markdown* syntax, for example, the Markdown Guide.

In *R Markdown*, the code is executed and the results are included in the document. This is a very powerful way of writing a document. If you change the code, the document is updated automatically. This is very useful if you need to update a document regularly. For example, if you need to update a report every month with the latest data. For larger documents that require considerable computation time, caching can be used to speed up the process (see Section D.2).

C.1 General syntax

See the R Markdown documentation on the Posit website for a detailed description of the *R Markdown* syntax. Here, we cover only a few important aspects that should you help write documents using *R Markdown*.

C.2 Code chunks

In order to embed R code in a document, we use the following syntax:

````
```{r chunk-name}
x = 123
```
````

The three backticks define a code block. The `{r}` indicates that we want to execute the content using *R*. Here, we also name the code chunk `chunk-name`. This is optional but useful, and required if you want to reference figures.

C.3 Chunk options

There are many options to control the behavior of a code chunk. In this document, we used a subset of them. Use the following to avoid displaying warnings and messages:

````
```{r chunk-name}
#| warning: FALSE
#| message: FALSE
...
```
````

However, only add these options if you understand what the displayed warning or message means and you are sure that you want to hide it.

C.4 Figures

Figures come with a large number of options. The ones we used in this document are:

- `fig.cap`: Caption of the figure.
- `fig.width`: Width of the figure in inches.
- `fig.height`: Height of the figure in inches.
- `fig.align`: Alignment of the figure. Possible values are `left`, `right` and `center`.
- `dev`: Specifies the format of the figure. The default depends on the output format. For HTML, the default is `png`; for PDF it is `pdf`. If you create a PDF file and your graph contains a large number of points, it will be better to use `#| dev: png` to avoid a large file size and faster rendering of the PDF.

```{r fig-figure-demo}
#| fig.cap: "Figure caption shown below the plot"
#| fig.width: 4
#| fig.height: 4
#| fig.align: "center"
#| out.width: 75%
#| dev: "png"
plot(1:10)
```

This will create the following table with the caption shown below the plot. The width of the figure is 75% of the text width and figure is centered.

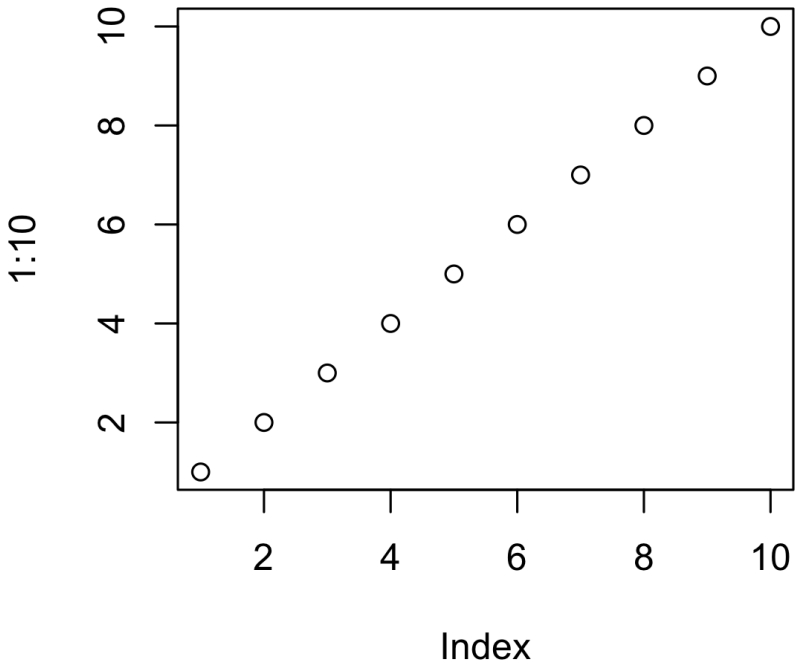


Figure C.1: Figure caption shown below the plot

In markdown, we can reference the Figure using `\@ref(fig:figure-demo)`. For example, Figure C.1 shows a plot of the numbers 1 to 10.

Useful to know

If your references don't show up, check if you knit to either `bookdown::pdf_document2` or `bookdown::html_document2`. The previous format for HTML documents (`html_document`) requires a different format for references.

Currently, knitting to PDF using markdown requires to have either `out.width` or `fig.align` specified. Otherwise, the resulting document will neither contain a figure caption nor will references work. You can also set `fig.align` as a global option:

```
knitr::opts_chunk$set(fig.align="center")
```

C.5 Referencing R variables in text

Sometimes, you might want to reference calculation results in your text. If you add the results manually, but later on change the calculations, the actual results and the text will no longer reference the same values. It is possible to embed results in text using markdown to avoid this situation. Here is an example:

```
rmse <- 1.2345678
```

We can reference the variable `rmse` in the text using:

> The RMSE value is `` `r round(rmse, 3)` ``.

The command `round(rmse, 3)` will be executed and the result shown in the text:

> The RMSE value is 1.235.

C.6 Troubleshooting

C.6.1 ! LaTeX Error: Unicode character ^^[(U+001B)

When you get this error message while knitting to LaTeX, set `warning` and `message` to `FALSE`.

````
```{r figure-demo}
#| warning: FALSE
#| message: FALSE
... code that ouptuts a warning that causes knitting to fail ...
```
````

Further information

- Overview of R markdown options: https://www.rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf

D Technical details

Once you start to develop models for larger datasets, you will realize that tuning models can take a long time. There are a few strategies to help speed up the process.

- Work with smaller datasets during development of your workflow until you get your *R* code right. Then switch to the full dataset. Chapter 12 shows how to create a representative subset of your data. This is a good strategy if you are working with a large dataset and you are not sure if your code is correct.
- Use parallel processing to speed up the tuning process. Approaches like cross-validation greatly benefit from this. Section D.1 shows how to do this. We made use of this approach in various chapters in this book. This is a good strategy that is useful for larger datasets.
- Use caching to speed up the tuning process. Section D.2 shows how to use caching to speed up the tuning process. This is a good strategy for when you are confident in your code and results, and you work on describing and discussing them in the text.

Load required packages:

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

D.1 Parallel processing

Modern computers no longer exist of a single CPU, but have multiple cores. You can use these cores to speed up the tuning process. *R* has a variety of packages to setup and use parallel processing on your computer. The package `future` is one of them. At the start of your *R* or _Rmarkdown file, you need to setup the parallel processing environment.

```
library(future)
plan(multisession,
  workers = parallel::detectCores(logical = FALSE))
```

After importing the `future` package, we create a cluster of workers. The number of workers is set to the number of cores on your computer. You can change this number to a lower number if you want to keep some cores available for other tasks or if your calculations runs out of memory (RAM).

```
file <- "https://gedeck.github.io/DS-6030/datasets/UniversalBank.csv.gz"
data <- read_csv(file) %>%
  select(-c(ID, `ZIP Code`)) %>%
  rename(
    Personal.Loan = `Personal Loan`,
    Securities.Account = `Securities Account`,
    CD.Account = `CD Account`
  ) %>%
  mutate(
    Personal.Loan = factor(Personal.Loan, labels = c("Yes", "No"),
      levels = c(1, 0)),
    Education = factor(Education,
      labels = c("Undergrad", "Graduate", "Advanced")),
  )

formula <- Personal.Loan ~ Income + Family + CCAvg + Education +
  Mortgage + Securities.Account + CD.Account + Online + CreditCard
recipe_rf <- recipe(formula, data = data)
model_rf <- rand_forest(mode = "classification",
  mtry = tune(), trees = tune()) %>%
  set_engine("ranger")
workflow_rf <- workflow() %>%
  add_recipe(recipe_rf) %>%
  add_model(model_rf)
parameters <- extract_parameter_set_dials(workflow_rf)
parameters <- parameters %>%
  update(
    mtry = mtry(c(2, 6)),
    trees = trees(c(100, 500))
  )
```

```
set.seed(1)
# repeat 10-fold cross-validation five times
resamples <- vfold_cv(data, v = 10, repeats = 5)

timings <- tibble()
for (number_cores in 1:20) {
  with(plan(multisession, workers = number_cores), {
```

```
    # train random forest model five times using cross-validation
    # and measure execution time
    start_time <- Sys.time()
    tune_rf <- tune_grid(
      workflow_rf,
      resamples = resamples,
      grid = grid_regular(parameters, levels = c(5, 2))
    )
    end_time <- Sys.time()
    time_taken <- round(end_time - start_time, 2)
    if (length(timings) == 0) {
      timings <- tibble(
        number_cores = number_cores,
        time_taken = time_taken
      )
    } else {
      timings <- timings %>%
        add_row(
          number_cores = number_cores,
          time_taken = time_taken
        )
    }
  })
}
timings$average_time_taken <- as.numeric(timings$time_taken) / 5
timings$ideal <- timings$average_time_taken[1] / timings$number_cores
```

We train $10 \times 5 \times 10 = 500$ models (10 tuning, 5 repeats, 10-fold cross-validation). Figure D.1 shows the reduction in tuning time when parallel processing is used.

```
ggplot(timings, aes(x = number_cores, y = average_time_taken)) +
  geom_line() +
  geom_line(aes(y = ideal), color = "grey") +
  geom_point() +
  geom_vline(
    xintercept = parallel::detectCores(logical = FALSE),
    color = "red") +
  xlab("Number of cores") +
  ylab("Average time required for tuning (seconds)")
```

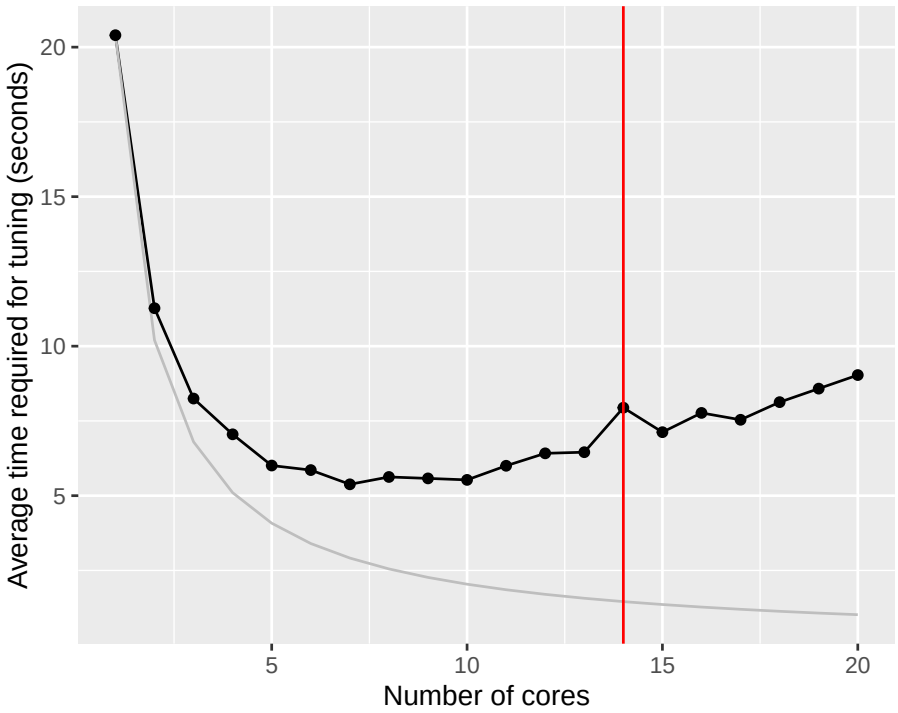


Figure D.1: Average time require for tuning a random forest model using different number of cores. The red line indicates the number of physical cores on the computer. The grey line the expected time if parallel execution would be perfect.

Without parallelization, tuning the random forest model takes about 20.4 seconds. Distributing the calculation on two cores, reduces the time to 11.3 seconds. If parallelization would be perfect, the time should be reduced to 10.2 seconds. There is overhead involved in parallelization and we can already see the effect of this here. Nevertheless, increasing the number of cores even more, brings the calculation time down to 5.4.

The improvement go through a minimum between 5 and 10 cores. After that, the average time required for tuning increases again slightly. The computer used for the graph was a M3 Macbook Pro with 14 cores.

On some computers, you might see hyperthreading. This is a technique to make the computer appear to have more cores than it actually has. This can be useful for tasks that require a lot of communication with other resources. However, for calculations like this, hyperthreading is not as efficient as physical cores.

Useful to know

When you use parallelization and you get error messages like this, your computer is running out of memory (RAM).

```
Error in unserialize(socklist[[n]]):
    error reading from connection
Error in serialize(data, node$con):
    error writing to connection
Error in `summary.connection()`: ! invalid connection
```

In this case, reduce the number of cores in the cluster. For example, set `plan(multisession, workers = 4)` to use only 4 cores.

Further information

The *tidymodels* package will already use parallelization if the a cluster is setup. However, if you want to parallelize your own code, you can use the `foreach` package. It's fairly easy to use. See the following links for examples: - https://cran.r-project.org/web/packages/doParallel/vignettes/gettingstartedParallel.pdf gives a short overview on how to use the `doParallel` package. - https://www.blasbenito.com/post/02_parallelizing_loops_with_r/ is a more in depth introduction into parallelization with the `foreach` package.

D.2 Caching

Running the code in the previous chapter takes more than 156 seconds. Executing the code every time this document needs to be recreated, will make working with the document insufferable. We can use caching to speed up the process. Caching is a technique to save the results of a calculation and reuse them later. The next time the code is executed, the results are loaded from the cache instead of being recalculated.

Using caching with *Rmarkdown* is straightforward. Above we used the `cache` option in the code chunk header to enable caching. Here is the markup for the code chunk that loads and prepares the data.

````
```{r cache-prepare-model}
#| message: FALSE
#| warning: FALSE
#| cache: TRUE
```
````

```
code to load and prepare the model

```
```

Note two points here. Firstly, the code chunk has a name. Secondly, the code chunk header contains the `cache: TRUE` option. The name identifies the cache. If you change the code inside the code chunk, the cache is invalidated and the code is executed again. If you don't specify a name, *Rmarkdown* will create a random name for the code chunk. This is not recommended as it can lead to issues later on.

We also use caching the code chunk that tunes the model.

```
```{r cache-time-tuning}
#| message: FALSE
#| warning: FALSE
#| cache: TRUE

code to tune the model

```
```

Like before, we use the `cache: TRUE` option which will cache the results of this code chunk too. Consider now the case where you knitted the document and both code chunks are cached and you make a change to the first code chunk which leads to a change of the data. When you knit the document again, this data are reloaded and preprocessed. The code chunk that tunes the model however was not changed and would in principle not be executed again. This would lead to invalid results. To avoid this, we use the `dependson` option in the code chunk header of the code chunk that tunes the model. The `dependson` option tells *Rmarkdown* that the code chunk that tunes the model depends on the code chunk that loads and prepares the model. If the code chunk that loads and prepares the model is executed, the code chunk that tunes the model will be executed too.

If you have a code chunk that depends on multiple previous code chunks, you can use a comma separated list of names in the `dependson` option. In this example, `test3` depends on `test1` and `test2`.

```
```{r test1}
#| cache: TRUE
x = 10
x
```


```{r test2}
```

```
#| cache: TRUE
#| dependson: test1
y = 2
y
```

```{r test3}
#| cache: TRUE
#| dependson: test1, test2
x + y
```

You can make a code chunk depend on all previous code chunks by using the `dependson=knitr::all_labels()` option.

Useful to know

Another option is to add the following code at the start of your document.

```{r}
knitr::opts_chunk$set(cache=TRUE, autodep=TRUE)
```

This will cache **all** code chunks in the document and automatically create the dependencies based on the variables used in and between code chunks.
If you use this approach together with parallelization, explicitly set the code that starts the cluster to not being cached.

```{r}
#| cache: FALSE
library(future)
plan(multisession,
    workers = parallel::detectCores(logical = FALSE))
```

Further information

- https://tune.tidymodels.org/articles/extras/optimizations.html more information on how to speed up your *tidymodels* code in particular with parallel processing.
- Caching has a lot more options that are not covered here. Check these sources for more information:

- https://r4ds.had.co.nz/r-markdown.html#caching
- https://bookdown.org/yihui/rmarkdown-cookbook/cache-path.html
- https://bookdown.org/yihui/rmarkdown-cookbook/cache-lazy.html
- https://yihui.org/knitr/options/#cache

References

Breiman, Leo, Jerome H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees*. Chapman & Hall.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2021. *An Introduction to Statistical Learning: With Applications in R*. Springer Texts in Statistics. New York, NY: Springer US. https://doi.org/10.1007/978-1-0716-1418-1.

Muschelli, John. 2020. "ROC and AUC with a Binary Predictor: A Potentially Misleading Metric." *Journal of Classification* 37 (3): 696–708. https://doi.org/10.1007/s00357-019-09345-1.

Scott, David W. 1992. *Multivariate Density Estimation: Theory, Practice, and Visualization*. 1st edition. New York: John Wiley & Sons.

Shmueli, Galit, Peter C. Bruce, Peter Gedeck, Inbal Yahav, and Nitin R. Patel. 2023. "Machine Learning for Business Analytics: Concepts, Techniques, and Applications in R, 2nd Edition | Wiley." *Wiley.com*. https://www.wiley.com/en-us/Machine+Learning+for+Business+Analytics%3A+Concepts%2C+Techniques%2C+and+Applications+in+R%2C+2nd+Edition-p-9781119835172.

Silverman, B. W. 1986. *Density Estimation for Statistics and Data Analysis*. Boca Raton: Chapman and Hall.

Trifonova, Oxana, Petr Lokhov, and A. I. Archakov. 2014. "Metabolic Profiling of Human Blood." *Biomeditsinskaya Khimiya* 60 (May): 281–94. https://doi.org/10.18097/pbmc20146003281.

Wickham, Hadley. 2011. "The Split-Apply-Combine Strategy for Data Analysis." *Journal of Statistical Software* 40 (April): 1–29. https://doi.org/10.18637/jss.v040.i01.

Wickham, Hadley, Mine Çetinkaya-Rundel, and Garrett Grolemund. 2023. "R for Data Science (2e)." https://r4ds.hadley.nz/.

Wilkinson, G. N., and C. E. Rogers. 1973. "Symbolic Description of Factorial Models for Analysis of Variance." *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 22 (3): 392–99. https://doi.org/10.2307/2346786.

Wilkinson, Leland. 2005. *The Grammar of Graphics*. Statistics and Computing. New York: Springer-Verlag. https://doi.org/10.1007/0-387-28695-0.

Wolpert, David H. 1992. "Stacked Generalization." *Neural Networks* 5 (2): 241–59. https://doi.org/10.1016/S0893-6080(05)80023-1.

Zou, Hui, Trevor Hastie, and Robert Tibshirani. 2006. "Sparse Principal Component Analysis." *Journal of Computational and Graphical Statistics* 15 (2): 265–86. https://www.jstor.org/stable/27594179.