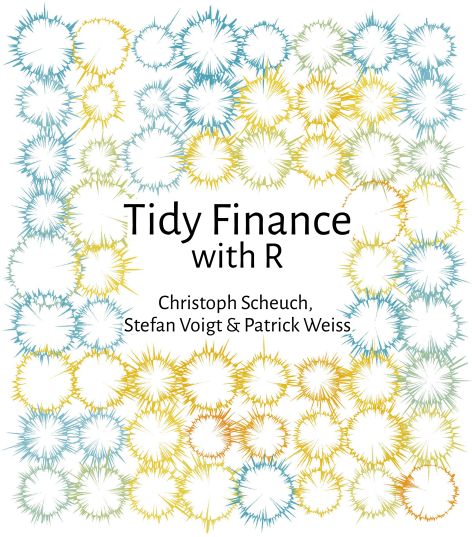
Christoph Scheuch (wikifolio Financial Technologies) and Stefan Voigt (University of Copenhagen and Danish Finance Institute) and Patrick Weiss (Vienna University of Economics and Business)

Tidy Finance with R

Contents

Preface



[1]

This website is the online version of *Tidy Finance with R*, a book currently under development and intended for eventual print release. The book is the result of a joint effort of Christoph Scheuch[2], Stefan Voigt[3], and Patrick Weiss[4].

We are grateful for any kind of feedback on *every* aspect of the book. So please get in touch with us via contact@tidy-finance.org[5] if you spot typos, discover any issues that deserve more attention, or if you have suggestions for additional chapters and sections. Additionally, let us know if you found the text helpful. We look forward to hearing from you!

[1]

[2] https://christophscheuch.github.io/

[3] https://voigtstefan.me/

[4] https://sites.google.com/view/patrick-weiss

[5] mailto:contact@tidy-finance.org

Why does this book exist?

Financial economics is a vibrant area of research, a central part of all businesses activities, and at least implicitly relevant for our everyday life. Despite its relevance for our society and a vast number of empirical studies of financial phenomenons, one quickly learns that the actual implementation is typically rather opaque. As graduate students, we were particularly surprised by the lack of public code for seminal papers or even textbooks on key concepts of financial economics. The lack of transparent code not only leads to numerous replication efforts (and their failures), but it also constitutes a waste of resources on problems that have already been solved by countless others in secrecy.

This book aims to lift the curtain on reproducible finance by providing a fully transparent code base for many common financial applications. We hope to inspire others to share their code publicly and take part in our journey towards more reproducible research in the future.

Who should read this book?

We write this book for three audiences:

- Students who want to acquire the basic tools required to conduct financial research ranging from undergrad to graduate level. The book's structure is simple enough such that the material is sufficient for self-study purposes.
- Instructors who look for materials to teach in empirical finance courses. We provide plenty of examples and (hopefully) intuitive explanations which can easily be adjusted or expanded.
- Data analysts or statisticians who work on issues pertaining to financial data and need practical tools to do so.

Our book is close in spirit to Regenstein Jr (2018) and Coqueret and Guida (2020) in that we provide fully reproducible code for useful applications and methods in finance.

The book "Reproducible Finance with R" provides an excellent introduction and discussion of different tools (based on the tidyverse, tidyquant, and xts) for standard applications in finance (e.g., how to compute returns and sample standard deviations of a time series of stock returns). Our book, in contrast, has a clear focus on

applications of state-of-the-art for academic research in finance. We thus fill a niche that allows aspiring researchers or instructors to rely on a well-designed code base.

The book "Machine Learning for Factor Investing" is a great compendium to our book with respect to applications related to return prediction and portfolio formation. The book primarily targets practitioners and has a hands-on focus. Our book, in contrast, relies on the typical databases used in financial research and focuses on the preparation of such datasets for academic applications. In addition, our chapter on machine learning focuses on factor selection instead of return prediction.

What will you learn?

The book is currently divided into 5 parts:

- Chapter 1 introduces you to important concepts around which our approach to Tidy Finance revolves.
- Chapter 2 provides tools to organize your data and prepare the most common data sets used in financial research: CRSP and Compustat. We reuse the data from this chapter in all following chapters.
- Chapters 3-7 deal with key concepts of empirical asset pricing such as beta estimation, portfolio sorts, and performance analysis.
- Chapters 8-9 apply machine learning methods to problems in factor selection and option pricing.
- Chapters 10-11 provide approaches for parametric, constrained portfolio optimization, and backtesting procedures.

Each chapter is self-contained and can be read individually. Yet the data chapter provides important background necessary for the data management in subsequent chapters. The number of chapters and covered content is subject to change as we will introduce additional material in the near future.

What won't you learn?

This book is about empirical work. While we assume only basic knowledge in statistics and econometrics, we do not provide detailed treatments of the underlying theoretical models or methods applied in this book. Instead, you find references to the seminal academic work in journal articles and to more detailed treatments. We believe that our comparative advantage is to provide a thorough implementation of

portfolio sorts, backtesting procedures, machine learning methods, or other related topics in empirical finance and enrich these implementations with discussions of the needy-greedy choices you face while conducting empirical analyses. We hence refrain from deriving theoretical models or discussing the statistical properties of well-established tools.

Why R?

We believe that R is among the best choices for a programming language in the area of finance. Some of our favorite features include:

- R is free and open-source so that you can use it in academic and professional contexts.
- A diverse and active online community works on a broad range of tools.
- A massive set of actively maintained packages for all kinds of applications exists, e.g., data manipulation, visualization, machine learning, etc.
- Powerful tools for communication, e.g., Rmarkdown and shiny, are readily available.
- RStudio is one of the best development environments for interactive data analysis.
- Strong foundations of functional programming are provided.
- Smooth integration with other programming languages, e.g., SQL, Python, C, C++, Fortran, etc.

For more information, we refer to Wickham et al. (2019).

Why tidy?

As you start working with data, you quickly realize that you spend a lot of time reading, cleaning, and transforming your data. In fact, it is often said that more than 80% of data analysis is spent on preparing data. By *tidying data*, we want to structure data sets to facilitate further analyses. As Wickham (2014) puts it:

> [T]idy datasets are all alike, but every messy dataset is messy in its own way. Tidy datasets provide a standardized way to link the structure of a dataset (its physical layout) with its semantics (its meaning).

In its essence, tidy data follows these three principles:

1. Every column is a variable.
2. Every row is an observation.
3. Every cell is a single value.

Throughout this book, we try to follow these principles as best as we can. If you want to learn more about tidy data principles in an informal manner, we refer you to this vignette[6].

In addition to the data layer, there are also tidy coding principles outlined in the tidy tools manifesto[7] that we try to follow:

1. Reuse existing data structures.
2. Compose simple functions with the pipe.
3. Embrace functional programming.
4. Design for humans.

In particular, we heavily draw on a set of packages called the `tidyverse`[8] (Wickham et al., 2019). The `tidyverse` is a consistent set of packages for all data analysis tasks, ranging from importing and wrangling to visualizing and modeling data with the same grammar. In addition to explicit tidy principles, the `tidyverse` has further benefits: (i) if you master one package, it is easier to master others, and (ii) the core packages are developed and maintained by the Public Benefit Company RStudio, Inc.

Throughout the book we use the pipe `|>`, a powerful tool to clearly express a sequence of operations. Readers familiar with the `tidyverse` may be used to the predecessor `%>%` by the `magrittr` package. For simple cases `|>` and `%>%` behave identically, however, we follow Hadley Wickhams note "the main advantage of `|>` is that it does less than `%>%`, i.e. it has what is important about the pipe with less of what is not important." For a more thorough discussion on the subtle differences, we refer to the second edition[9] of Wickham and Grolemund (2016).

Prerequisites

Before we continue, make sure you have all the software you need for this book:

[6] https://cran.r-project.org/web/packages/tidyr/vignettes/tidy-data.html
[7] https://tidyverse.tidyverse.org/articles/manifesto.html
[8] https://tidyverse.tidyverse.org/index.html
[9] https://r4ds.hadley.nz/workflow-pipes.html

- Install R and RStudio[10]. To get a walk-through of the installation for every major operating system, follow the steps outlined in this summary[11]. The whole process should be done in a few clicks. If you wonder about the difference: R is an open-source language and environment for statistical computing and graphics, free to download and use. While R runs the computations, RStudio is an integrated development environment that provides an interface by adding many convenient features and tools. We suggest doing all the coding in RStudio.
- Open RStudio and install the `tidyverse`[12]. Not sure how it works? You find helpful information on how to install packages in this brief summary[13].

If you are new to R, we recommend starting with the following sources:

- A very gentle and good introduction into the workings of R can be found in the form of the weighted dice project[14]. Once you are done setting up R on your machine, try to follow the instructions in this project.
- The main book on the `tidyverse`, Wickham and Grolemund (2016) is available online and for free: R for Data Science[15] by Hadley Wickham and Garrett Grolemund explains the majority of the tools we use in our book.
- If you are an instructor searching for effectively teach R and data science methods, we recommend to take a look on the excellent data science toolbox[16] by Mine Cetinkaya-Rundel[17].
- RStudio provides a range of excellent cheat sheets[18] with abundand information on how to use the `tidyverse` packages.

About the authors

We met at the Vienna Graduate School of Finance[19] from which each of us graduated with a different focus but a shared passion: coding with R. We continue to sharpen our R skills as part of our current occupations:

- Christoph Scheuch[20] is the Director of Product at the social trading platform wiki-

[10] https://rstudio-education.github.io/hopr/starting.html#starting
[11] https://rstudio-education.github.io/hopr/starting.html#starthng
[12] https://tidyverse.tidyverse.org/
[13] https://rstudio-education.github.io/hopr/packages2.html
[14] https://rstudio-education.github.io/hopr/project-1-weighted-dice.html
[15] https://r4ds.had.co.nz/introduction.html
[16] https://datasciencebox.org/
[17] https://mine-cr.com/about/
[18] https://www.rstudio.com/resources/cheatsheets/
[19] https://www.vgsf.ac.at/
[20] https://christophscheuch.github.io/

folio.com[21] where he is responsible for product planning, execution, and monitoring. He also manages a team of data scientists to analyze user behavior and develop new products.
- Stefan Voigt[22] is an Assistant Professor of Finance at the Department of Economics at the University in Copenhagen[23] and a research fellow at the Danish Finance Institute[24]. His research focuses on blockchain technology, high-frequency trading, and financial econometrics. Stefan teaches parts of this book in his courses on empirical finance.
- Patrick Weiss[25] is a Post-Doc at the Vienna University of Economics and Business[26]. His research centers around the intersection between asset pricing and corporate finance.

License

Colophon

This book was written in RStudio using bookdown. The website is hosted with github pages and automatically updated after every commit. The complete source is available from GitHub[29]. We generated all plots in this book using `ggplot2` and its classic dark-on-light theme (`theme_bw()`).

This version of the book was built with R version 4.2.1 (2022-06-23, Funny-Looking Kid) and the following packages:

[21] https://www.wikifolio.com/
[22] https://voigtstefan.me/
[23] https://www.economics.ku.dk/
[24] https://danishfinanceinstitute.dk/
[25] https://sites.google.com/view/patrick-weiss
[26] https://www.wu.ac.at/en/
[27] https://creativecommons.org/licenses/by-nc-sa/4.0/
[28] https://creativecommons.org/publicdomain/zero/1.0/
[29] www.github.com/voigtstefan/tidy_finance

Package	Version
alabama	2022.4-1
bookdown	0.27
broom	1.0.0
dbplyr	2.2.1
dplyr	1.0.9
forcats	0.5.1
frenchdata	0.2.0
furrr	0.3.0
ggplot2	3.3.6
glmnet	4.1-4
googledrive	2.0.0
hardhat	1.2.0
jsonlite	1.8.0
kableExtra	1.3.4
keras	2.9.0
knitr	1.39
lmtest	0.9-40
lubridate	1.8.0
purrr	0.3.4
quadprog	1.5-8
ranger	0.14.1
readr	2.1.2
readxl	1.4.0
renv	0.15.5
rlang	1.0.3
rmarkdown	2.14
RPostgres	1.4.4
RSQLite	2.2.14
sandwich	3.0-2
scales	1.2.0
slider	0.2.2
stringr	1.4.0
tibble	3.1.7
tidymodels	0.2.0
tidyquant	1.0.4
tidyr	1.2.0
tidyverse	1.3.1
timetk	2.8.1
wesanderson	0.3.6

Getting started

1

Introduction to Tidy Finance

The main aim of this chapter is to familiarize yourself with the `tidyverse`. We start by downloading and visualizing stock data before moving to a simple portfolio choice problem. These examples introduce you to our approach of *Tidy Finance*.

1.1 Working with stock market data

At the start of each session, we load the required packages. Throughout the entire book, we always use the package `tidyverse`. In this chapter, we also load the convenient `tidyquant` package to download price data. You typically have to install a package once before you can load it. In case you have not done this yet, call `install.packages("tidyquant")`. If you have trouble using `tidyquant`, check out the documentation[1].

```
library(tidyverse)
library(tidyquant)
```

We first download daily prices for one stock market ticker, e.g., *AAPL*, directly from the data provider Yahoo!Finance. To download the data, you can use the command `tq_get`. If you do not know how to use it, make sure you read the help file by calling `?tq_get`. We especially recommend taking a look at the documentation's examples section.

```
prices <- tq_get("AAPL",
  get = "stock.prices",
  from = "2000-01-01",
  to = "2022-03-30"
)
prices
```

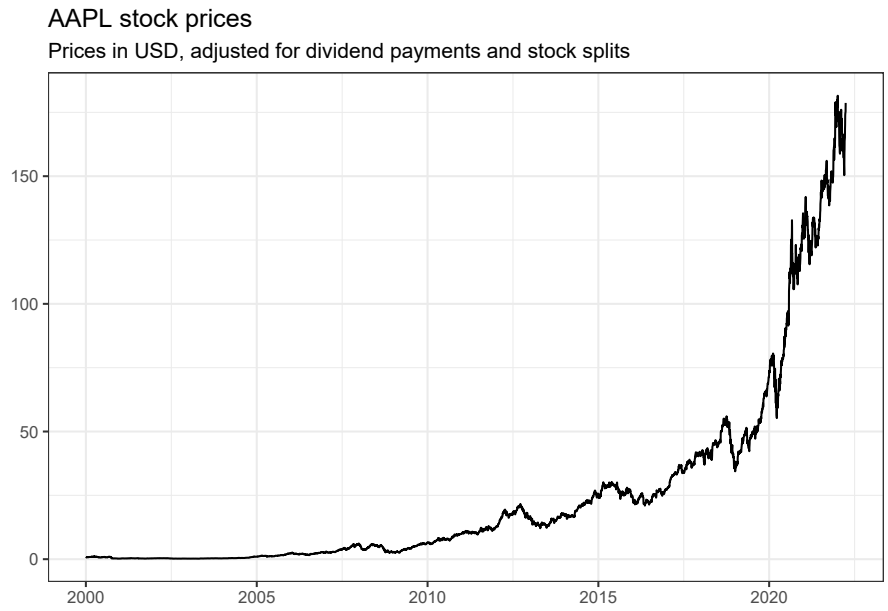
[1] https://cran.r-project.org/web/packages/tidyquant/vignettes/TQ01-core-functions-in-tidyquant.html#yahoo-finance

```
## # A tibble: 5,596 x 8
##   symbol date        open  high   low close    volume
##   <chr>  <date>     <dbl> <dbl> <dbl> <dbl>     <dbl>
## 1 AAPL   2000-01-03 0.936 1.00  0.908 0.999 535796800
## 2 AAPL   2000-01-04 0.967 0.988 0.903 0.915 512377600
## 3 AAPL   2000-01-05 0.926 0.987 0.920 0.929 778321600
## 4 AAPL   2000-01-06 0.948 0.955 0.848 0.848 767972800
## 5 AAPL   2000-01-07 0.862 0.902 0.853 0.888 460734400
## # ... with 5,591 more rows, and 1 more variable:
## #   adjusted <dbl>
```

`tq_get` downloads stock market data from Yahoo!Finance if you do not specify another data source. The function returns a tibble with eight quite self-explanatory columns: *symbol*, *date*, the market prices at the *open*, *high*, *low* and *close*, the daily *volume* (in number of traded shares), and the *adjusted* price in USD. The adjusted prices are corrected for anything that might affect the stock price after the market closes, e.g., stock splits and dividends. These actions affect the quoted prices, but they have no direct impact on the investors who hold the stock.

Next, we use `ggplot2` to visualize the time series of adjusted prices.

```
prices |>
  ggplot(aes(x = date, y = adjusted)) +
  geom_line() +
  labs(
    x = NULL,
    y = NULL,
    title = "AAPL stock prices",
    subtitle = "Prices in USD, adjusted for dividend payments and stock splits"
  )
```



Instead of analyzing prices, we compute daily returns defined as $(p_t - p_{t-1})/p_{t-1}$ where p_t is the adjusted day t price. The function `lag` computes the previous value in a vector.

```
returns <- prices |>
  arrange(date) |>
  mutate(ret = (adjusted - lag(adjusted)) / lag(adjusted)) |>
  select(symbol, date, ret)
returns
```

```
## # A tibble: 5,596 x 3
##   symbol date           ret
##   <chr>  <date>       <dbl>
## 1 AAPL   2000-01-03 NA
## 2 AAPL   2000-01-04 -0.0843
## 3 AAPL   2000-01-05  0.0146
## 4 AAPL   2000-01-06 -0.0865
## 5 AAPL   2000-01-07  0.0474
## # ... with 5,591 more rows
```

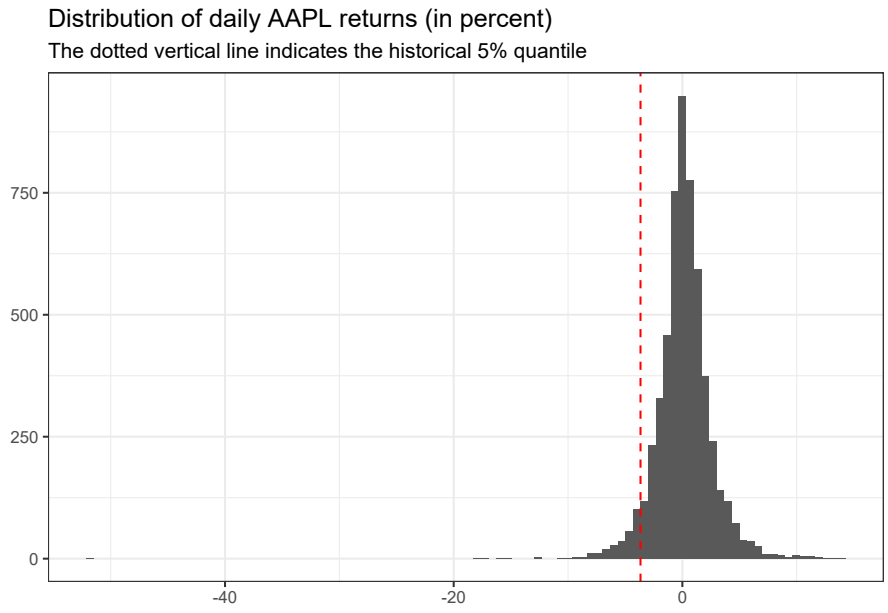
The resulting tibble contains three columns where the last contains the daily returns. Note that the first entry naturally contains `NA` because there is no previous price. Additionally, the computations require that the time series is ordered by date. Otherwise, `lag` would be meaningless.

For the upcoming examples, we remove missing values as these would require separate treatment when computing, e.g., sample averages. In general, however, make sure you understand why NA values occur and carefully examine if you can simply get rid of these observations.

```
returns <- returns |>
  drop_na(ret)
```

Next, we visualize the distribution of daily returns in a histogram. For convenience, we multiply the returns by 100 to get returns in percent for the visualizations. Additionally, we also add a dashed red line that indicates the 5% quantile of the daily returns to the histogram, which is a (crude) proxy for the worst return of the stock with a probability of at least 5%.

```
quantile_05 <- quantile(returns |> pull(ret) * 100, 0.05)

returns |>
  ggplot(aes(x = ret * 100)) +
  geom_histogram(bins = 100) +
  geom_vline(aes(xintercept = quantile_05),
    color = "red",
    linetype = "dashed"
  ) +
  labs(
    x = NULL,
    y = NULL,
    title = "Distribution of daily AAPL returns (in percent)",
    subtitle = "The dotted vertical line indicates the historical 5% quantile"
  )
```



Here, `bins = 100` determines the number of bins and hence implicitly the width of the bins. Before proceeding, make sure you understand how to use the geom `geom_vline()` to add a dotted red line that indicates the 5% quantile of the daily returns. A typical task before proceeding with *any* data is to compute summary statistics for the main variables of interest.

```
returns |>
  mutate(ret = ret * 100) |>
  summarize(across(
    ret,
    list(
      daily_mean = mean,
      daily_sd = sd,
      daily_min = min,
      daily_max = max
    )
  ))
```

```
## # A tibble: 1 x 4
##   ret_daily_mean ret_daily_sd ret_daily_min
##            <dbl>        <dbl>         <dbl>
## 1          0.129         2.52         -51.9
## # ... with 1 more variable: ret_daily_max <dbl>
```

We see that the maximum *daily* return was around 13.905 percent.
You can also compute these summary statistics for each year by imposing `group_by(year = year(date))`, where the call `year(date)` computes the year.

```
returns |>
  mutate(ret = ret * 100) |>
  group_by(year = year(date)) |>
  summarize(across(
    ret,
    list(
      daily_mean = mean,
      daily_sd = sd,
      daily_min = min,
      daily_max = max
    ),
    .names = "{.fn}"
  ))
```

```
## # A tibble: 23 x 5
##    year daily_mean daily_sd daily_min daily_max
##   <dbl>      <dbl>    <dbl>     <dbl>     <dbl>
## 1  2000     -0.346     5.49    -51.9      13.7
## 2  2001      0.233     3.93    -17.2      12.9
## 3  2002     -0.121     3.05    -15.0       8.46
## 4  2003      0.186     2.34     -8.14     11.3
## 5  2004      0.470     2.55     -5.58     13.2
## # ... with 18 more rows
```

In case you wonder: the additional argument `.names = "{.fn}"` in `across()` determines how to name the output columns. The specification is rather flexible and allows almost arbitrary column names, which can be useful for reporting.

1.2 Scaling up the analysis

As a next step, we generalize the code from before such that all the computations can handle an arbitrary vector of tickers (e.g., all constituents of an index). Following tidy principles, it is quite easy to download the data, plot the price time series, and tabulate the summary statistics for an arbitrary number of assets.

This is where the `tidyverse` magic starts: tidy data makes it extremely easy to generalize the computations from before to as many assets you like. The following code

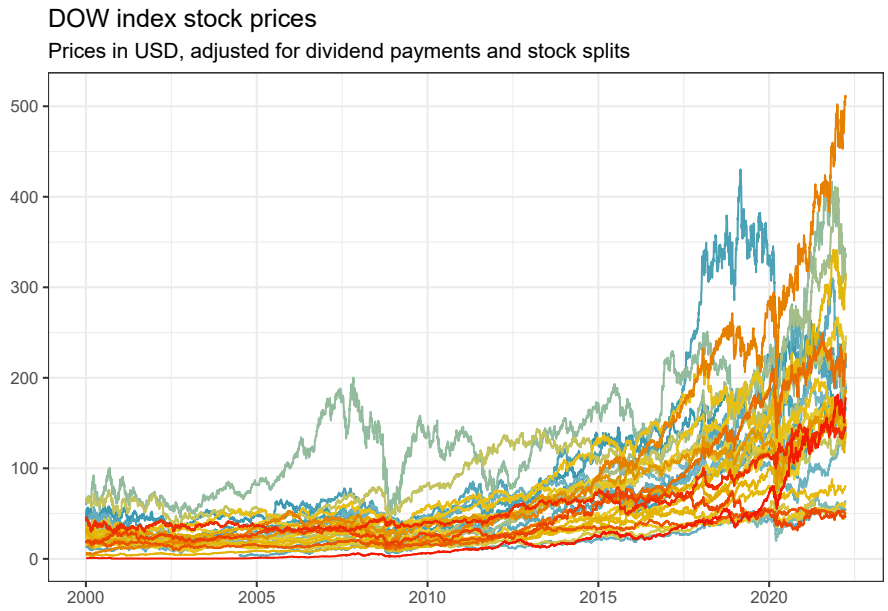
takes any vector of tickers, e.g., `ticker <- c("AAPL", "MMM", "BA")`, and automates the download as well as the plot of the price time series. In the end, we create the table of summary statistics for an arbitrary number of assets. We perform the analysis with data from all current constituents of the Dow Jones Industrial Average index[2].

```
ticker <- tq_index("DOW")
index_prices <- tq_get(ticker,
  get = "stock.prices",
  from = "2000-01-01",
  to = "2022-03-30"
)
```

The resulting file contains 159863 daily observations for in total 30 different corporations. The figure below illustrates the time series of downloaded *adjusted* prices for each of the constituents of the Dow Jones index. Make sure you understand every single line of code! (What are the arguments of `aes()`? Which alternative geoms could you use to visualize the time series? Hint: if you do not know the answers try to change the code to see what difference your intervention causes).

```
index_prices |>
  ggplot(aes(
    x = date,
    y = adjusted,
    color = symbol
  )) +
  geom_line() +
  labs(
    x = NULL,
    y = NULL,
    color = NULL,
    title = "DOW index stock prices",
    subtitle = "Prices in USD, adjusted for dividend payments and stock splits"
  ) +
  theme(legend.position = "none")
```

[2] https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average



Do you notice the small differences relative to the code we used before? `tq_get(ticker)` returns a tibble for several symbols as well. All we need to do to illustrate all tickers simultaneously is to include `color = symbol` in the `ggplot2` aesthetics. In this way, we generate a separate line for each ticker. Of course, there are simply too many lines on this graph to properly identify the individual stocks, but it illustrates the point well.

The same holds for stock returns. Before computing the returns, we use `group_by(symbol)` such that the `mutate` command is performed for each symbol individually. The same logic applies to the computation of summary statistics: `group_by(symbol)` is the key to aggregating the time series into ticker-specific variables of interest.

```
all_returns <- index_prices |>
  group_by(symbol) |>
  mutate(ret = adjusted / lag(adjusted) - 1) |>
  select(symbol, date, ret) |>
  drop_na(ret)

all_returns |>
  mutate(ret = ret * 100) |>
  group_by(symbol) |>
  summarize(across(
    ret,
    list(
```

```
      daily_mean = mean,
      daily_sd = sd,
      daily_min = min,
      daily_max = max
    ),
    .names = "{.fn}"
  ))
```

```
## # A tibble: 30 x 5
##   symbol daily_mean daily_sd daily_min daily_max
##   <chr>       <dbl>    <dbl>     <dbl>     <dbl>
## 1 AMGN       0.0484     1.98     -13.4      15.1
## 2 AXP        0.0572     2.30     -17.6      21.9
## 3 BA         0.0603     2.20     -23.8      24.3
## 4 CAT        0.0707     2.03     -14.5      14.7
## 5 CRM        0.124      2.68     -27.1      26.0
## # ... with 25 more rows
```

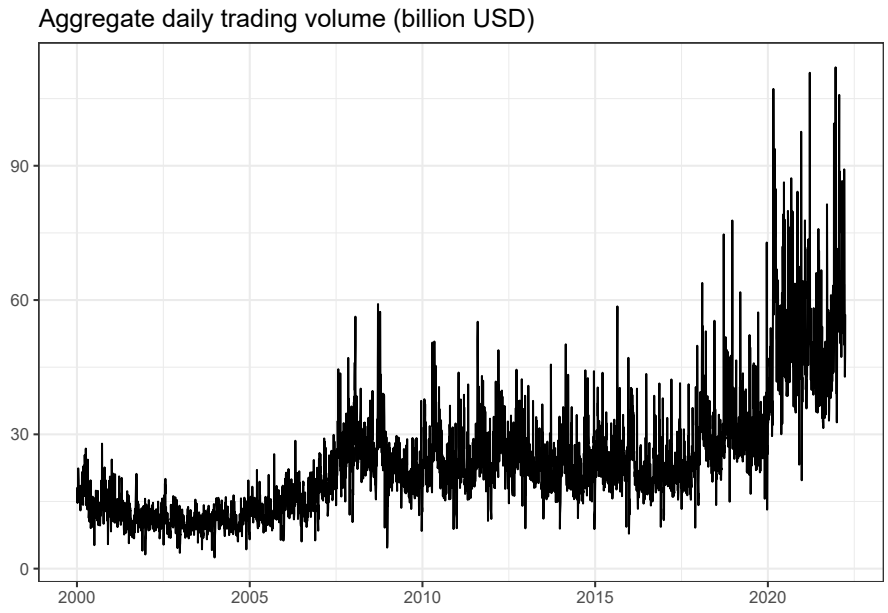
Note that you are now also equipped with all tools to download price data for *each* ticker listed in the S&P 500 index with the same number of lines of code. Just use `ticker <- tq_index("SP500")`, which provides you with a tibble that contains each symbol that is (currently) part of the S&P 500. However, don't try this if you are not prepared to wait for a couple of minutes because this is quite some data to download!

1.3 Other forms of data aggregation

Of course, aggregation across other variables than `symbol` can make sense as well. For instance, suppose you are interested in answering the question: are days with high aggregate trading volume likely followed by days with high aggregate trading volume? To provide some initial analysis on this question, we take the downloaded prices and compute aggregate daily trading volume for all Dow Jones constituents in USD. Recall that the column *volume* is denoted in the number of traded shares. Thus, we multiply the trading volume with the daily closing price to get a proxy for the aggregate trading volume in USD. Scaling by `1e9` denotes daily trading volume in billion USD.

```
volume <- index_prices |>
  mutate(volume_usd = volume * close / 1e9) |>
  group_by(date) |>
  summarize(volume = sum(volume_usd))
```

```
volume |>
  ggplot(aes(x = date, y = volume)) +
  geom_line() +
  labs(
    x = NULL, y = NULL,
    title = "Aggregate daily trading volume (billion USD)"
  )
```

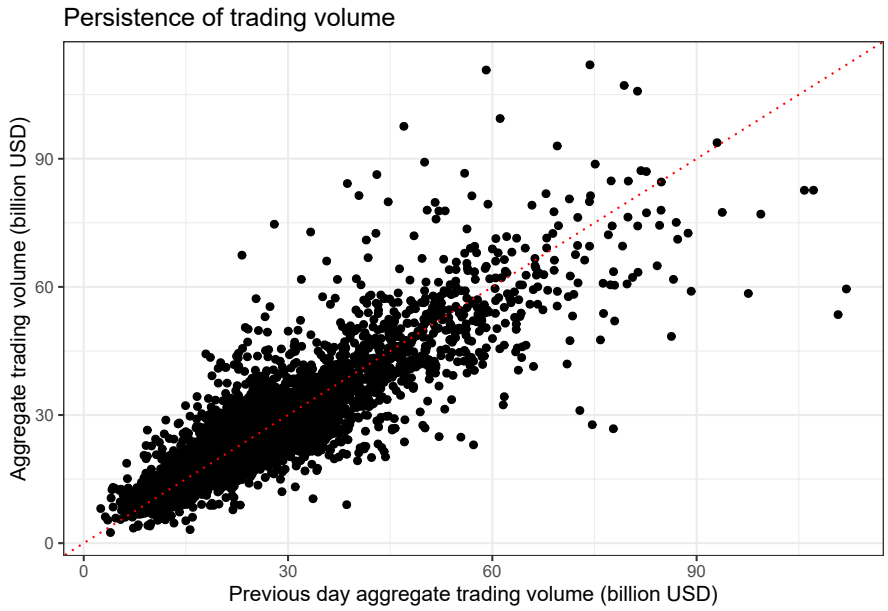


One way to illustrate the persistence of trading volume would be to plot volume on day t against volume on day $t-1$ as in the example below. We add a 45°-line to indicate a hypothetical one-to-one relation by `geom_abline`, addressing potential differences in the axes' scales.

```
volume |>
  ggplot(aes(x = lag(volume), y = volume)) +
  geom_point() +
  geom_abline(aes(intercept = 0, slope = 1),
    color = "red",
    linetype = "dotted"
  ) +
  labs(
    x = "Previous day aggregate trading volume (billion USD)",
    y = "Aggregate trading volume (billion USD)",
```

```
    title = "Persistence of trading volume"
  )
```

```
## Warning: Removed 1 rows containing missing values
## (geom_point).
```



Do you understand where the warning `## Warning: Removed 1 rows containing missing values (geom_point).` comes from and what it means? Purely eye-balling reveals that days with high trading volume are often followed by similarly high trading volume days.

1.4 Portfolio choice problems

In the previous part, we show how to download stock market data and inspect it with graphs and summary statistics. Now, we move to a typical question in Finance, namely, how to optimally allocate wealth across different assets. The standard framework for optimal portfolio selection considers investors that prefer higher future returns but dislike future return volatility (defined as the square root of the return variance): the *mean-variance investor*.

An essential tool to evaluate portfolios in the mean-variance context is the *efficient frontier*, the set of portfolios which satisfy the condition that no other portfolio exists with a higher expected return but with the same volatility (i.e., the risk). We compute and visualize the efficient frontier for several stocks. First, we extract each asset's *monthly* returns. In order to keep things simple, we work with a balanced panel and exclude tickers for which we do not observe a price on every single trading day since 2000.

```
index_prices <- index_prices |>
  group_by(symbol) |>
  mutate(n = n()) |>
  ungroup() |>
  filter(n == max(n)) |>
  select(-n)

returns <- index_prices |>
  mutate(month = floor_date(date, "month")) |>
  group_by(symbol, month) |>
  summarize(price = last(adjusted), .groups = "drop_last") |>
  mutate(ret = price / lag(price) - 1) |>
  drop_na(ret) |>
  select(-price)
```

Next, we transform the returns from a tidy tibble into a $(T \times N)$ matrix with one column for each of the N tickers to compute the covariance matrix Σ and also the expected return vector μ. We achieve this by using `pivot_wider()` with the new column names from the column `symbol` and setting the values to `ret`. We compute the vector of sample average returns and the sample variance-covariance matrix, which we consider as proxies for the parameters of future returns.

```
returns_matrix <- returns |>
  pivot_wider(
    names_from = symbol,
    values_from = ret
  ) |>
  select(-month)

sigma <- cov(returns_matrix)
mu <- colMeans(returns_matrix)
```

Then, we compute the minimum variance portfolio weights ω_{mvp} as well as the expected return $\omega_{\text{mvp}}'\mu$ and volatility $\sqrt{\omega_{\text{mvp}}'\Sigma\omega_{\text{mvp}}}$ of this portfolio. Recall that the minimum variance portfolio is the vector of portfolio weights that are the solution to

$$\omega_{\text{mvp}} = \arg\min w'\Sigma w \text{ s.t. } \sum_{i=1}^{N} w_i = 1.$$

It is easy to show analytically, that $\omega_{\text{mvp}} = \frac{\Sigma^{-1}\iota}{\iota'\Sigma^{-1}\iota}$ where ι is a vector of ones.

```
N <- ncol(returns_matrix)
iota <- rep(1, N)
mvp_weights <- solve(sigma) %*% iota
mvp_weights <- mvp_weights / sum(mvp_weights)

tibble(
  expected_ret = t(mvp_weights) %*% mu,
  volatility = sqrt(t(mvp_weights) %*% sigma %*% mvp_weights)
)
```

```
## # A tibble: 1 x 2
##   expected_ret[,1] volatility[,1]
##              <dbl>          <dbl>
## 1          0.00838         0.0314
```

Note that the *monthly* volatility of the minimum variance portfolio is of the same order of magnitude as the *daily* standard deviation of the individual components. Thus, the diversification benefits in terms of risk reduction are tremendous!

Next, we set out to find the weights for a portfolio that achieves three times the expected return of the minimum variance portfolio. However, mean-variance investors are not interested in any portfolio that achieves the required return but rather in the efficient portfolio, i.e., the portfolio with the lowest standard deviation. If you wonder where the solution ω_{eff} comes from: The efficient portfolio is chosen by an investor who aims to achieve minimum variance *given a minimum acceptable expected return* $\bar{\mu}$. Hence, their objective function is to choose ω_{eff} as the solution to

$$\omega_{\text{eff}}(\bar{\mu}) = \arg\min w' \Sigma w \text{ s.t. } w'\iota = 1 \text{ and } w'\mu \geq \bar{\mu}.$$

The code below implements the analytic solution to this optimization problem for a benchmark return $\bar{\mu}$ which we set to 3 times the expected return of the minimum variance portfolio. We encourage you to verify that it is correct.

```
mu_bar <- 3 * t(mvp_weights) %*% mu

C <- as.numeric(t(iota) %*% solve(sigma) %*% iota)
D <- as.numeric(t(iota) %*% solve(sigma) %*% mu)
E <- as.numeric(t(mu) %*% solve(sigma) %*% mu)

lambda_tilde <- as.numeric(2 * (mu_bar - D / C) / (E - D^2 / C))
efp_weights <- mvp_weights + lambda_tilde / 2 * (solve(sigma) %*% mu - D / C * solve(sigma) %*% iota)
```

1.5 The efficient frontier

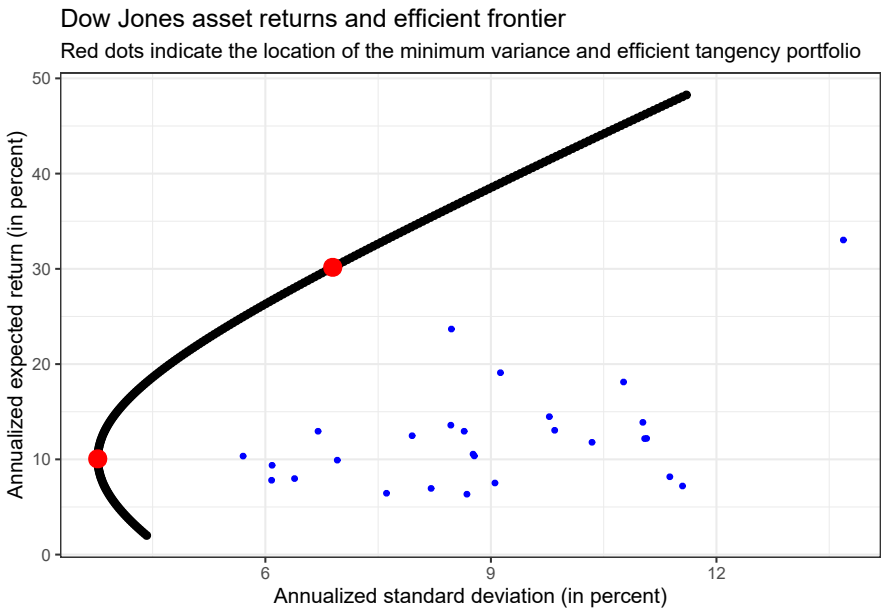
The two mutual fund separation theorem states that as soon as we have two efficient portfolios (such as the minimum variance portfolio and the efficient portfolio for another required level of expected returns like above), we can characterize the entire efficient frontier by combining these two portfolios. The code below implements the construction of the *efficient frontier*, which characterizes the highest expected return achievable at each level of risk. To understand the code better, make sure to familiarize yourself with the inner workings of the `for` loop.

```
c <- seq(from = -0.4, to = 1.9, by = 0.01)
res <- tibble(
  c = c,
  mu = NA,
  sd = NA
)

for (i in seq_along(c)) {
  w <- (1 - c[i]) * mvp_weights + (c[i]) * efp_weights
  res$mu[i] <- 12 * 100 * t(w) %*% mu
  res$sd[i] <- 12 * sqrt(100) * sqrt(t(w) %*% sigma %*% w)
}
```

Finally, it is simple to visualize the efficient frontier alongside the two efficient portfolios within one, powerful figure using `ggplot2`. We also add the individual stocks in the same call.

```
res |>
  ggplot(aes(x = sd, y = mu)) +
  geom_point() +
  geom_point( # locate the minimum variance and efficient portfolio
    data = res |> filter(c %in% c(0, 1)),
    color = "red",
    size = 4
  ) +
  geom_point( # locate the individual assets
    data = tibble(mu = 12 * 100 * mu, sd = 12 * 10 * sqrt(diag(sigma))),
    aes(y = mu, x = sd), color = "blue", size = 1
  ) +
  labs(
    x = "Annualized standard deviation (in percent)",
    y = "Annualized expected return (in percent)",
    title = "Dow Jones asset returns and efficient frontier",
```

```
    subtitle = "Red dots indicate the location of the minimum variance and efficient tangency portfolio"
)
```



The black line indicates the efficient frontier: the set of portfolios a mean-variance efficient investor would choose from. Compare the performance relative to the individual assets (the blue dots) - it should become clear that diversifying yields massive performance gains (at least as long as we take the parameters Σ and μ as given).

1.6 Exercises

1. Download daily prices for another stock market ticker of your choice from Yahoo!Finance with `tq_get` from the `tidyquant` package. Plot two time series of the ticker's un-adjusted and adjusted closing prices. Explain the differences.
2. Compute daily net returns for the asset and visualize the distribution of daily returns in a histogram. Also, use `geom_vline()` to add a dashed red line that indicates the 5% quantile of the daily returns within the histogram. Compute summary statistics (mean, standard deviation, minimum and maximum) for the daily returns

3. Take your code from before and generalize it such that you can perform all the computations for an arbitrary vector of tickers (e.g., `ticker <- c("AAPL", "MMM", "BA")`). Automate the download, the plot of the price time series, and create a table of return summary statistics for this arbitrary number of assets.
4. Consider the research question: Are days with high aggregate trading volume often also days with large absolute price changes? Find an appropriate visualization to analyze the question.
5. Compute monthly returns from the downloaded stock market prices. Compute the vector of historical average returns and the sample variance-covariance matrix. After you compute the minimum variance portfolio weights and the portfolio volatility and average returns, visualize the mean-variance efficient frontier. Choose one of your assets and identify the portfolio which yields the same historical volatility but achieves the highest possible average return.
6. In the portfolio choice analysis, we restricted our sample to all assets that were trading on every single day since 2000. How is such a decision a problem when you want to infer future expected portfolio performance from the results?
7. The efficient frontier characterizes the portfolios with the highest expected return for different levels of risk, i.e., standard deviation. Identify the portfolio with the highest expected return per standard deviation. Hint: the ratio of expected return to standard deviation is an important concept in Finance.

Financial data

2

Accessing & managing financial data

In this chapter, we propose a way to organize your financial data. Everybody, who has experience with data, is also familiar with storing data in various formats like CSV, XLS, XLSX, or other delimited value stores. Reading and saving data can become very cumbersome in the case of using different data formats, both across different projects, as well as across different programming languages. Moreover, storing data in delimited files often leads to problems with respect to column type consistency. For instance, date-type columns frequently lead to inconsistencies across different data formats and programming languages.

This chapter shows how to import different data sets. Specifically, our data comes from the application programming interface (API) of Yahoo!Finance, a downloaded standard CSV files, an XLSX file stored in a public Google drive repositories, and an SQL database connection. We store all the data in a **single** database, which serves as the only source of data in subsequent chapters.

First, we load the global packages that we use throughout this chapter. Later on, we load more packages in the sections where we need them.

```
library(tidyverse)
library(lubridate)
library(scales)
```

Moreover, we initially define the date range for which we fetch and store the financial data, making future data updates tractable. In case you need another time frame, you need to adjust these dates. Our data starts with 1960 since most asset pricing studies use data from 1962 on.

```
start_date <- as.Date("1960-01-01")
end_date <- as.Date("2020-12-31")
```

2.1 Fama-French data

We start by downloading some famous Fama-French factors (e.g., (Fama and French, 1993)) and portfolio returns commonly used in empirical asset pricing. For-

tunately, there is a neat package by Nelson Areal[1] that allows us to easily access the data: the `frenchdata` package provides functions to download and read data sets from Prof. Kenneth French finance data library[2].

```
library(frenchdata)
```

We can use the main function of the package to download monthly Fama-French factors. The set *3 Factors* includes the return time series of the market, size, and value factors alongside the risk-free rates. Note that we have to do some manual work to correctly parse all the columns and scale them appropriately as the raw Fama-French data comes in very unpractical data format. For precise descriptions of the variables, we suggest consulting Prof. Kenneth French finance data library[3] directly. If you are on the site, check the raw data files to appreciate the time saved by `frenchdata`.

```
factors_ff_monthly <- download_french_data("Fama/French 3 Factors")$subsets$data[[1]] %>%
  transmute(
    month = floor_date(ymd(paste0(date, "01")), "month"),
    rf = as.numeric(RF) / 100,
    mkt_excess = as.numeric(`Mkt-RF`) / 100,
    smb = as.numeric(SMB) / 100,
    hml = as.numeric(HML) / 100
  ) %>%
  filter(month >= start_date & month <= end_date)
```

It is straightforward to download the corresponding *daily* Fama-French factors with the same function.

```
factors_ff_daily <- download_french_data("Fama/French 3 Factors [Daily]")$subsets$data[[1]] %>%
  transmute(
    date = ymd(date),
    rf = as.numeric(RF) / 100,
    mkt_excess = as.numeric(`Mkt-RF`) / 100,
    smb = as.numeric(SMB) / 100,
    hml = as.numeric(HML) / 100
  ) %>%
  filter(date >= start_date & date <= end_date)
```

In a subsequent chapter, we also use the 10 monthly industry portfolios, so let us fetch that data, too.

[1] https://github.com/nareal/frenchdata/
[2] https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html
[3] https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html

```
industries_ff_monthly <- download_french_data("10 Industry Portfolios")$subsets$data[[1]] %>%
  mutate(month = floor_date(ymd(paste0(date, "01")), "month")) %>%
  mutate(across(where(is.numeric), ~ . / 100)) %>%
  select(month, everything(), -date) %>%
  filter(month >= start_date & month <= end_date)
```

It is worth taking a look at all available portfolio return time series from Kenneth French's homepage. You should check out the other sets by calling `frenchdata::get_french_data_list()`.

2.2 q-factors

In recent years, the academic discourse experienced the rise of alternative factor models, e.g., in the form of the (Hou et al., 2014) *q*-factor model. We refer to the extended background[4] information provided by the original authors for further information. The *q* factors can be downloaded directly from the authors' homepage from within `read_csv()`.

We also need to adjust this data. First, we discard information we will not use here. Then, we rename the columns with the "R_"-prescript using regular expressions and write all column names in lower case. You can try sticking to a consistent style for naming objects, which we try to illustrate here - the emphasis is on *try*. You can check out style guides available online, e.g., Hadley Wickham's `tidyverse` style guide[5].

```
factors_q_monthly <- read_csv("http://global-q.org/uploads/1/2/2/6/122679606/q5_factors_monthly_2020.csv") %>%
  mutate(month = as.Date(paste(year, month, "01", sep = "-"))) %>%
  select(-R_F, -R_MKT, -year) %>%
  rename_with(~ gsub("R_", "", .)) %>%
  rename_with(~ str_to_lower(.)) %>%
  mutate(across(-month, ~ . / 100)) %>%
  filter(month >= start_date & month <= end_date)
```

[4] http://global-q.org/background.html
[5] https://style.tidyverse.org/index.html

2.3 Macroeconomic predictors

Our next data source is a set of macroeconomic variables often used as predictors for the equity premium. (Welch and Goyal, 2008) comprehensively reexamine the performance of variables suggested by the academic literature to be good predictors of the equity premium. The authors host the data updated to 2020 on Amit Goyal's website[6]. Since the data is a .xlsx-file stored on a public Google drive location, we need additional packages to access the data directly from our R session. Therefore, we load `readxl` to read the .xlsx-file and `googledrive` for the Google drive connection.

```
library(readxl)
library(googledrive)
```

Usually, you need to authenticate if you interact with Google drive directly in R. Since the data is stored via a public link, we can proceed without any authentication.

```
drive_deauth()
```

The `drive_download()` function from the `googledrive` package allows us to download the data and store it locally.

```
drive_download("https://drive.google.com/file/d/1ACbhdnIy0VbCWgsnXkjcddiV8HF4feWv/view",
  path = "data/macro_predictors.xlsx"
)
```

Next, we read in the new data and transform the columns to the variables that we later use. You can consult the material on Amit Goyal's website[7] for the definitions of the variables and the transformations.

```
macro_predictors <- read_xlsx("data/macro_predictors.xlsx",
                              sheet = "Monthly") %>%
  mutate(month = ym(yyyymm)) %>%
  filter(month >= start_date & month <= end_date) %>%
  mutate(across(where(is.character), as.numeric)) %>%
  mutate(
    IndexDiv = Index + D12,
    logret = log(IndexDiv) - log(lag(IndexDiv)),
    Rfree = log(Rfree + 1),
    rp_div = lead(logret - Rfree, 1), # Future excess market return
```

[6] https://sites.google.com/view/agoyal145
[7] https://sites.google.com/view/agoyal145

```
    dp = log(D12) - log(Index), # Dividend Price ratio
    dy = log(D12) - log(lag(Index)), # Dividend yield
    ep = log(E12) - log(Index), # Earnings price ratio
    de = log(D12) - log(E12), # Dividend payout ratio
    tms = lty - tbl, # Term spread
    dfy = BAA - AAA # Default yield spread
  ) %>%
  select(month, rp_div, dp, dy, ep, de, svar,
    bm = `b/m`, ntis, tbl, lty, ltr,
    tms, dfy, infl
  ) %>%
  drop_na()
```

Finally, after reading in the macro predictors to our memory, we remove the raw data file from our temporary storage.

```
file.remove("data/macro_predictors.xlsx")
```

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

2.4 Setting up a database

Now that we have downloaded some data from the web into the memory of our R session, let us set up a database to store that information for future use. We will use the data stored in this database throughout the following chapters, but you could alternatively implement a different strategy and replace the respective code.

There are many ways to set up and organize a database, depending on the use case. For our purpose, the most efficient way is to use an SQLite[8] database, which is the C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine. Note that SQL[9] (Structured Query Language) is a standard language for accessing and manipulating databases, and it heavily inspired the `dplyr` functions. We refer to this tutorial[10] for more information on SQL.

There are two packages that make working with SQLite in R very simple: `RSQLite` embeds the SQLite database engine in R and `dbplyr` is the database back-end for `dplyr`. These packages allow to set up a database to remotely store tables and use these remote database tables as if they are in-memory data frames by automatically

[8] https://www.sqlite.org/index.html
[9] https://en.wikipedia.org/wiki/SQL
[10] https://www.w3schools.com/sql/sql_intro.asp

converting `dplyr` into SQL. Check out the RSQLite[11] and dbplyr vignettes[12] for more information.

```
library(RSQLite)
library(dbplyr)
```

A SQLite database is easily created - the code below is really all there is. Note that we use the `extended_types` option to enable date types when storing and fetching data, otherwise date columns are stored as integer values.

```
tidy_finance <- dbConnect(SQLite(), "data/tidy_finance.sqlite",
                          extended_types = TRUE)
```

Next, we create a remote table with the monthly Fama-French factor data.

```
factors_ff_monthly %>%
  dbWriteTable(tidy_finance, "factors_ff_monthly", ., overwrite = TRUE)
```

We can use the remote table as an in-memory data frame by building a connection via `tbl()`.

```
factors_ff_monthly_db <- tbl(tidy_finance, "factors_ff_monthly")
```

All `dplyr` calls are evaluated lazily, i.e., the data is not in the memory of our R session, and actually, the database does most of the work. You can see that by noticing that the output below does not show the number of rows. In fact, the following code chunk only fetches the top 10 rows from the database for printing.

```
factors_ff_monthly_db %>%
  select(month, rf)
```

```
## # Source:   SQL [?? x 2]
## # Database: sqlite 3.38.5 [C:\Users\ncj140\Dropbox\Projects\tidy_finance\data\tidy_finance.sqlite]
##   month          rf
##   <date>      <dbl>
## 1 1960-01-01 0.0033
## 2 1960-02-01 0.0029
## 3 1960-03-01 0.0035
## 4 1960-04-01 0.0019
## 5 1960-05-01 0.0027
## # ... with more rows
```

[11] https://cran.r-project.org/web/packages/RSQLite/vignettes/RSQLite.html
[12] https://db.rstudio.com/databases/sqlite/

If we want to have the whole table in memory, we need to `collect()` it. You will see that we regularly load the data into the memory in the next chapters.

```
factors_ff_monthly_db %>%
  select(month, rf) %>%
  collect()
```

```
## # A tibble: 732 x 2
##   month          rf
##   <date>      <dbl>
## 1 1960-01-01 0.0033
## 2 1960-02-01 0.0029
## 3 1960-03-01 0.0035
## 4 1960-04-01 0.0019
## 5 1960-05-01 0.0027
## # ... with 727 more rows
```

The last couple of code chunks are really all there is to organize a simple database! You can also share the SQLite database across devices and programming languages.

Before we move on to the next data source, let us also store the other four tables in our new SQLite database.

```
factors_ff_daily %>%
  dbWriteTable(tidy_finance, "factors_ff_daily", ., overwrite = TRUE)

industries_ff_monthly %>%
  dbWriteTable(tidy_finance, "industries_ff_monthly", ., overwrite = TRUE)

factors_q_monthly %>%
  dbWriteTable(tidy_finance, "factors_q_monthly", ., overwrite = TRUE)

macro_predictors %>%
  dbWriteTable(tidy_finance, "macro_predictors", ., overwrite = TRUE)
```

From now on, all you need to do to access data that is stored in the database is to follow three steps: (i) Establish the connection to the SQLite database, (ii) call the table you want to extract, and (iii) collect the data. For your convenience, the following steps show all you need in a compact fashion.

```
library(tidyverse)
library(RSQLite)
tidy_finance <- dbConnect(SQLite(), "data/tidy_finance.sqlite",
                          extended_types = TRUE)
factors_q_monthly <- tbl(tidy_finance, "factors_q_monthly")
factors_q_monthly <- factors_q_monthly %>% collect()
```

2.5 Accessing WRDS

Wharton Research Data Services (WRDS)[13] is the most widely used source for asset and firm-specific financial data used in academic settings. WRDS is a data platform that provides data validation, flexible delivery options, and access to many different data sources. The data at WRDS is also organized in an SQL database, although they use the PostgreSQL[14] engine. This database engine is just as easy to handle with R as SQLite. We use the `RPostgres` package to establish a connection to the WRDS database. Note that you could also use the `odbc` package to connect to a PostgreSQL database, but then you need to install the appropriate drivers yourself. `RPostgres` already contains a suitable driver.

```
library(RPostgres)
```

To establish a connection, you use the function `dbConnect()` with the following arguments. Note that you need to replace the `user` and `password` fields with your own credentials. We defined system variables for the purpose of this book because we obviously do not want to share our credentials with the rest of the world.

```
wrds <- dbConnect(
  Postgres(),
  host = "wrds-pgdata.wharton.upenn.edu",
  dbname = "wrds",
  port = 9737,
  sslmode = "require",
  user = Sys.getenv("user"),
  password = Sys.getenv("password")
)
```

The remote connection to WRDS is very useful. Yet, the database itself contains many different databases and tables. You can check the WRDS homepage to identify the table's name you are looking for (if you go beyond our exposition). Alternatively, you can also query the data structure with the function `dbSendQuery()`. If you are interested, there is an exercise below that is based on WRDS' tutorial on "Querying WRDS Data using R"[15]. Furthermore, the penultimate section of this chapter shows how to investigate the structure of databases.

[13] https://wrds-www.wharton.upenn.edu/

[14] https://www.postgresql.org/

[15] https://wrds-www.wharton.upenn.edu/pages/support/programming-wrds/programming-r/querying-wrds-data-r/

2.6 Downloading and preparing CRSP

The Center for Research in Security Prices (CRSP)[16] provides the most widely used data for US stocks. We use the `wrds` connection object that we just created to first access monthly CRSP return data. Actually, we need three tables to get the desired data: (i) the CRSP monthly security file,

```
msf_db <- tbl(wrds, in_schema("crsp", "msf"))
msf_db
```

```
## # Source:   table<"crsp"."msf"> [?? x 21]
## # Database: postgres  [pweiss@wrds-pgdata.wharton.upenn.edu:9737/wrds]
##   cusip    permno permco issuno hexcd hsiccd date
##   <chr>     <dbl>  <dbl>  <dbl> <dbl>  <dbl> <date>
## 1 68391610  10000   7952  10396     3   3990 1985-12-31
## 2 68391610  10000   7952  10396     3   3990 1986-01-31
## 3 68391610  10000   7952  10396     3   3990 1986-02-28
## 4 68391610  10000   7952  10396     3   3990 1986-03-31
## 5 68391610  10000   7952  10396     3   3990 1986-04-30
## # ... with more rows, and 14 more variables:
## #   bidlo <dbl>, askhi <dbl>, prc <dbl>, vol <dbl>,
## #   ret <dbl>, bid <dbl>, ask <dbl>, shrout <dbl>,
## #   cfacpr <dbl>, cfacshr <dbl>, altprc <dbl>,
## #   spread <dbl>, altprcdt <date>, retx <dbl>
```

(ii) the identifying information,

```
msenames_db <- tbl(wrds, in_schema("crsp", "msenames"))
msenames_db
```

```
## # Source:   table<"crsp"."msenames"> [?? x 21]
## # Database: postgres  [pweiss@wrds-pgdata.wharton.upenn.edu:9737/wrds]
##   permno namedt     nameendt   shrcd exchcd siccd
##    <dbl> <date>     <date>     <dbl>  <dbl> <dbl>
## 1  10000 1986-01-07 1986-12-03    10      3  3990
## 2  10000 1986-12-04 1987-03-09    10      3  3990
## 3  10000 1987-03-10 1987-06-11    10      3  3990
## 4  10001 1986-01-09 1993-11-21    11      3  4920
## 5  10001 1993-11-22 2004-06-09    11      3  4920
## # ... with more rows, and 15 more variables:
```

[16] https://crsp.org/

```
## #   ncusip <chr>, ticker <chr>, comnam <chr>,
## #   shrcls <chr>, tsymbol <chr>, naics <chr>,
## #   primexch <chr>, trdstat <chr>, secstat <chr>,
## #   permco <dbl>, compno <dbl>, issuno <dbl>,
## #   hexcd <dbl>, hsiccd <dbl>, cusip <chr>
```

and (iii) the delisting information.

```
msedelist_db <- tbl(wrds, in_schema("crsp", "msedelist"))
msedelist_db
```

```
## # Source:   table<"crsp"."msedelist"> [?? x 19]
## # Database: postgres  [pweiss@wrds-pgdata.wharton.upenn.edu:9737/wrds]
##   permno dlstdt     dlstcd nwperm nwcomp nextdt
##    <dbl> <date>      <dbl>  <dbl>  <dbl> <date>
## 1  10000 1987-06-11    560      0      0 1987-06-12
## 2  10001 2017-08-03    233      0      0 NA
## 3  10002 2013-02-15    231  35263   1658 NA
## 4  10003 1995-12-15    231  10569   8477 NA
## 5  10005 1991-07-11    560      0      0 1991-07-12
## # ... with more rows, and 13 more variables:
## #   dlamt <dbl>, dlretx <dbl>, dlprc <dbl>,
## #   dlpdt <date>, dlret <dbl>, permco <dbl>,
## #   compno <dbl>, issuno <dbl>, hexcd <dbl>,
## #   hsiccd <dbl>, cusip <chr>, acperm <dbl>,
## #   accomp <dbl>
```

We use the three remote tables to fetch the data we want to put into our local database. Just as above, the idea is that we let the WRDS database do all the work and just download the data that we actually need. We apply common filters and data selection criteria to narrow down our data of interest: (i) we keep only data in the time windows of interest, (ii) we keep only US-listed stocks as identified via share codes 10 and 11, and (iii) we keep only months with valid permno-specific information from `msenames`. In addition, we add delisting reasons and returns. You can read up in the great textbook of (Bali et al., 2016) (BEM) for an extensive discussion on the filters we apply in the code below.

```
crsp_monthly <- msf_db %>%
  filter(date >= start_date & date <= end_date) %>%
  inner_join(msenames_db %>%
    filter(shrcd %in% c(10, 11)) %>%
    select(permno, exchcd, siccd, namedt, nameendt), by = c("permno")) %>%
  filter(date >= namedt & date <= nameendt) %>%
  mutate(month = floor_date(date, "month")) %>%
```

```
  left_join(msedelist_db %>%
    select(permno, dlstdt, dlret, dlstcd) %>%
    mutate(month = floor_date(dlstdt, "month")), by = c("permno", "month")) %>%
  select(
    permno, # Security identifier
    date, # Date of the observation
    month, # Month of the observation
    ret, # Return
    shrout, # Shares outstanding (in thousands)
    altprc, # Last traded price in a month
    exchcd, # Exchange code
    siccd, # Industry code
    dlret, # Delisting return
    dlstcd # Delisting code
  ) %>%
  mutate(
    month = as.Date(month),
    shrout = shrout * 1000
  ) %>%
  collect()
```

Now, we have all the relevant monthly return data in memory and proceed with preparing the data for future analyses. We perform the preparation step at the current stage since we want to avoid executing the same mutations every time we use the data in subsequent chapters.

The first additional variable we create is market capitalization (`mktcap`). Note that we keep market cap in millions of US dollars just for convenience (we do not want to print huge numbers in our figures and tables). Moreover, we set zero market cap to missing as it makes conceptually little sense (i.e., the firm would be bankrupt).

```
crsp_monthly <- crsp_monthly %>%
  mutate(
    mktcap = abs(shrout * altprc) / 1000000,
    mktcap = if_else(mktcap == 0, as.numeric(NA), mktcap)
  )
```

The next variable we frequently use is the one-month *lagged* market capitalization. Lagged market capitalization is typically used to compute value-weighted portfolio returns, as we demonstrate in a later chapter. The most simple and consistent way to add a column with lagged market cap values is to add one month to each observation and then join the information to our monthly CRSP data.

```
mktcap_lag <- crsp_monthly %>%
  mutate(month = month %m+% months(1)) %>%
  select(permno, month, mktcap_lag = mktcap)

crsp_monthly <- crsp_monthly %>%
  left_join(mktcap_lag, by = c("permno", "month"))
```

If you wonder why we do not use the `lag()` function, e.g., via `crsp_monthly %>% group_by(permno) %>% mutate(mktcap_lag = lag(mktcap))`, take a look at the exercises.

Next, we follow BEM in transforming listing exchange codes to explicit exchange names.

```
crsp_monthly <- crsp_monthly %>%
  mutate(exchange = case_when(
    exchcd %in% c(1, 31) ~ "NYSE",
    exchcd %in% c(2, 32) ~ "AMEX",
    exchcd %in% c(3, 33) ~ "NASDAQ",
    TRUE ~ "Other"
  ))
```

Similarly, we transform industry codes to industry descriptions following BEM. Notice that there are also other categorizations of industries (e.g., by Eugene Fama and Kenneth French) that are commonly used.

```
crsp_monthly <- crsp_monthly %>%
  mutate(industry = case_when(
    siccd >= 1 & siccd <= 999 ~ "Agriculture",
    siccd >= 1000 & siccd <= 1499 ~ "Mining",
    siccd >= 1500 & siccd <= 1799 ~ "Construction",
    siccd >= 2000 & siccd <= 3999 ~ "Manufacturing",
    siccd >= 4000 & siccd <= 4899 ~ "Transportation",
    siccd >= 4900 & siccd <= 4999 ~ "Utilities",
    siccd >= 5000 & siccd <= 5199 ~ "Wholesale",
    siccd >= 5200 & siccd <= 5999 ~ "Retail",
    siccd >= 6000 & siccd <= 6799 ~ "Finance",
    siccd >= 7000 & siccd <= 8999 ~ "Services",
    siccd >= 9000 & siccd <= 9999 ~ "Public",
    TRUE ~ "Missing"
  ))
```

We also construct returns adjusted for delistings as described by BEM. After this transformation, we can drop the delisting returns and codes.

```
crsp_monthly <- crsp_monthly %>%
  mutate(ret_adj = case_when(
    is.na(dlstcd) ~ ret,
    !is.na(dlstcd) & !is.na(dlret) ~ dlret,
    dlstcd %in% c(500, 520, 580, 584) |
      (dlstcd >= 551 & dlstcd <= 574) ~ -0.30,
    dlstcd == 100 ~ ret,
    TRUE ~ -1
  )) %>%
  select(-c(dlret, dlstcd))
```

Next, we compute excess returns by subtracting the monthly risk-free rate provided by our Fama-French data. As we base all our analyses on the excess returns, we can drop adjusted returns and the risk-free rate from our tibble. Note that we ensure that excess returns are bounded by -1 from below as less than -100% return make conceptually no sense.

```
crsp_monthly <- crsp_monthly %>%
  left_join(factors_ff_monthly %>% select(month, rf), by = "month") %>%
  mutate(
    ret_excess = ret_adj - rf,
    ret_excess = pmax(ret_excess, -1)
  ) %>%
  select(-ret_adj, -rf)
```

Since excess returns and market capitalization are crucial for all our analyses, we can safely exclude all observations with missing returns or market capitalization.

```
crsp_monthly <- crsp_monthly %>%
  drop_na(ret_excess, mktcap, mktcap_lag)
```

Finally, we store the monthly CRSP file in our database.

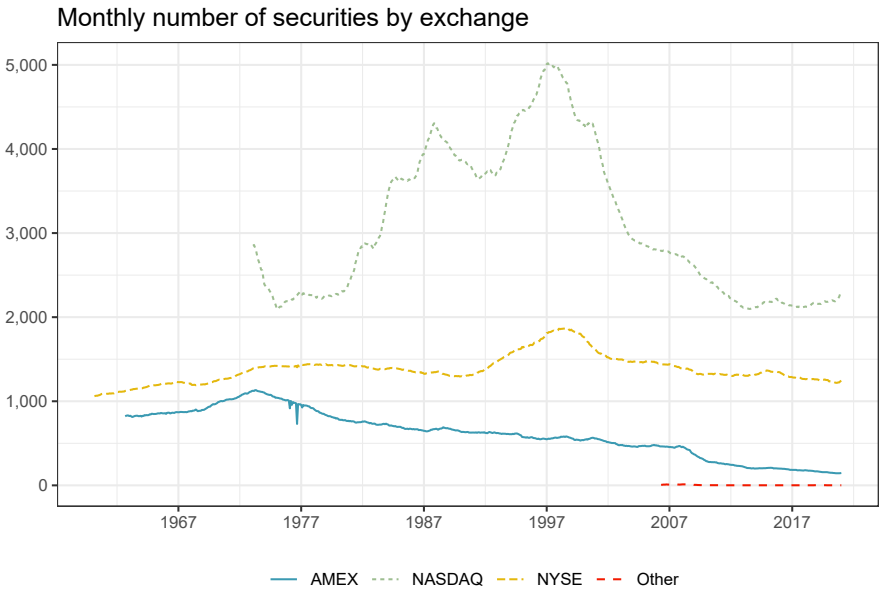
```
crsp_monthly %>%
  dbWriteTable(tidy_finance, "crsp_monthly", ., overwrite = TRUE)
```

2.7 First glimpse of the CRSP sample

Before we move on to other data sources, let us look at some descriptive statistics of the CRSP sample, which is our main source for stock returns.

The figure below shows the monthly number of securities by listing exchange over time. NYSE has the longest history in the data, but NASDAQ exhibits a considerable large number of stocks. The number of stocks on AMEX is decreasing steadily over the last couple of decades. By the end of 2020, there are 2300 stocks on NASDAQ, 1244 on NYSE, 147 on AMEX and only 1 belongs to the other category.

```
crsp_monthly %>%
  count(exchange, date) %>%
  ggplot(aes(x = date, y = n, color = exchange, linetype = exchange)) +
  geom_line() +
  labs(
    x = NULL, y = NULL, color = NULL, linetype = NULL,
    title = "Monthly number of securities by exchange"
  ) +
  scale_x_date(date_breaks = "10 years", date_labels = "%Y") +
  scale_y_continuous(labels = comma)
```



Next, we look at the aggregate market capitalization of the respective listing exchanges. To ensure that we look at meaningful data which is comparable over time, we adjust the nominal values for inflation. We use the familiar `tidyquant` package to fetch consumer price index (CPI) data from the Federal Reserve Economic Data (FRED)[17].

[17] https://fred.stlouisfed.org/series/CPIAUCNS

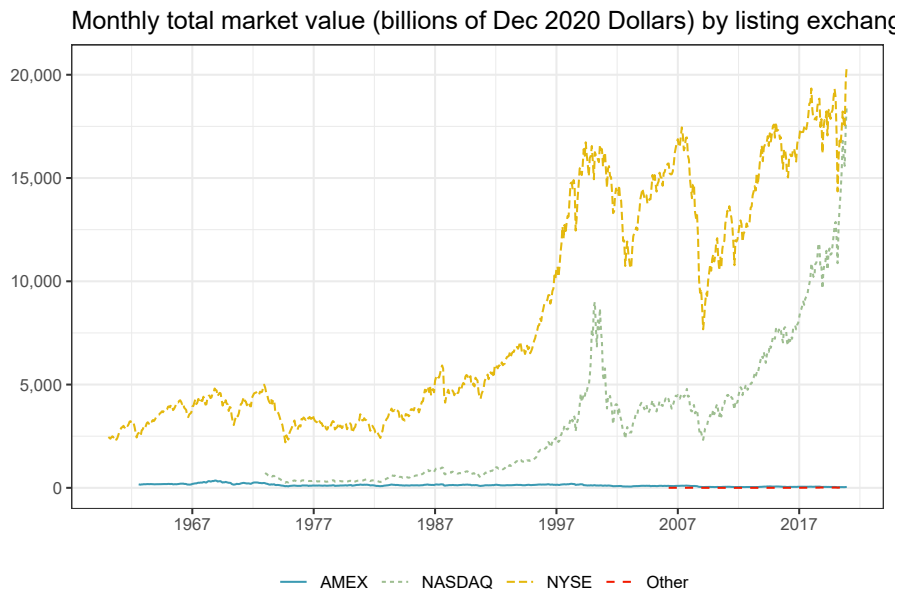
```
library(tidyquant)

cpi_monthly <- tq_get("CPIAUCNS",
  get = "economic.data",
  from = start_date, to = end_date
) %>%
  transmute(
    month = floor_date(date, "month"),
    cpi = price / price[month == max(crsp_monthly$month)]
  )
```

As the CPI data might come in handy at some point, we also put it into our local database.

```
cpi_monthly %>%
  dbWriteTable(tidy_finance, "cpi_monthly", ., overwrite = TRUE)
```

In fact, we can use the tables in our database to calculate aggregate market caps by listing exchange and plotting it just as if it were in memory. All values are in end of `year(end_date)` dollars to ensure inter-temporal comparability. NYSE-listed stocks have by far the largest market capitalization, followed by NASDAQ-listed stocks.

```
tbl(tidy_finance, "crsp_monthly") %>%
  left_join(tbl(tidy_finance, "cpi_monthly"), by = "month") %>%
  group_by(month, exchange) %>%
  summarize(
    securities = n_distinct(permno),
    mktcap = sum(mktcap, na.rm = TRUE) / cpi
  ) %>%
  collect() %>%
  mutate(month = as.Date(month)) %>%
  ggplot(aes(x = month, y = mktcap / 1000, color = exchange, linetype = exchange)) +
  geom_line() +
  labs(
    x = NULL, y = NULL, color = NULL, linetype = NULL,
    title = "Monthly total market value (billions of Dec 2020 Dollars) by listing exchange"
  ) +
  scale_x_date(date_breaks = "10 years", date_labels = "%Y") +
  scale_y_continuous(labels = comma)
```

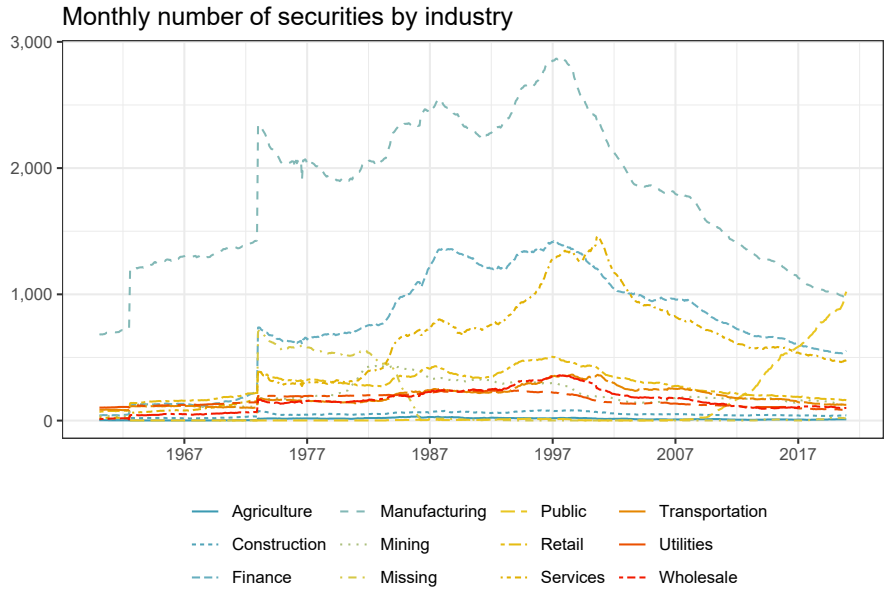


Of course, performing the computation in the database is not really meaningful because we already have all the required data in memory. The code chunk above is slower than performing the same steps on tables that are already in memory. However, we just want to illustrate that you can perform many things in the database before loading the data into your memory.

Next, we look at the same descriptive statistics by industry. The figure below plots the number of stocks in the sample for each of the SIC industry classifiers. For most of the sample period, the largest share of stocks is apparently in Manufacturing, albeit the number peaked somewhere in the 90s. The number of firms associated with public administration seems to be the only category on the rise in recent years, even surpassing Manufacturing at the end of our sample period.

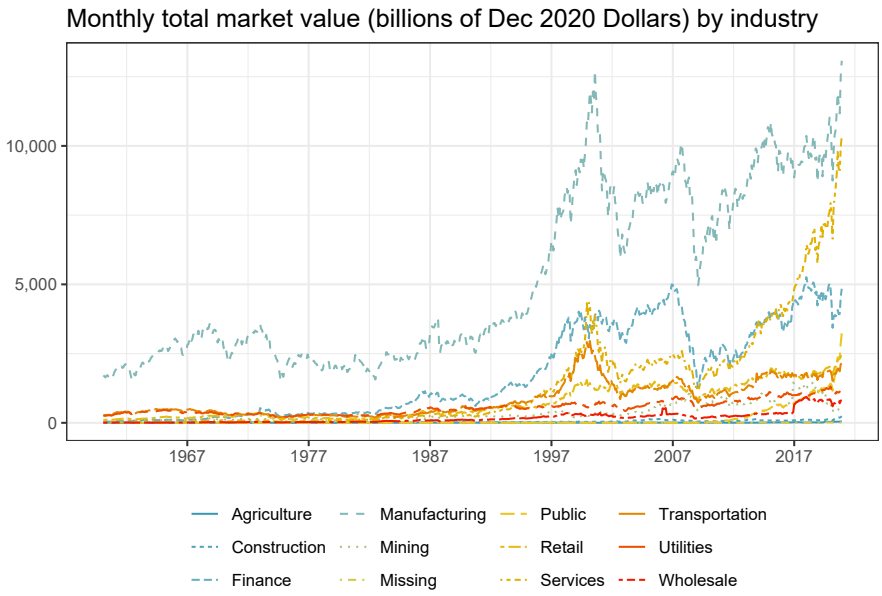
```
crsp_monthly_industry <- crsp_monthly %>%
  left_join(cpi_monthly, by = "month") %>%
  group_by(month, industry) %>%
  summarize(
    securities = n_distinct(permno),
    mktcap = sum(mktcap) / mean(cpi),
    .groups = "drop"
  )

crsp_monthly_industry %>%
  ggplot(aes(x = month, y = securities, color = industry, linetype = industry)) +
  geom_line() +
```

```
  labs(
    x = NULL, y = NULL, color = NULL, linetype = NULL,
    title = "Monthly number of securities by industry"
  ) +
  scale_x_date(date_breaks = "10 years", date_labels = "%Y") +
  scale_y_continuous(labels = comma)
```



We also compute the market value of all stocks belonging to the respective industries. All values are again in terms of billions of end of 2020 dollars. At all points in time, manufacturing firms comprise of the largest portion of market capitalization. Towards the end of the sample, however, financial firms and services begin to make up a substantial portion of the market value.

```
crsp_monthly_industry %>%
  ggplot(aes(x = month, y = mktcap / 1000, color = industry, linetype = industry)) +
  geom_line() +
  labs(
    x = NULL, y = NULL, color = NULL, linetype = NULL,
    title = "Monthly total market value (billions of Dec 2020 Dollars) by industry"
  ) +
  scale_x_date(date_breaks = "10 years", date_labels = "%Y") +
  scale_y_continuous(labels = comma)
```



2.8 Daily CRSP data

Before we turn to accounting data, we also want to provide a proposal for downloading daily CRSP data. While the monthly data from above typically fit into your memory and can be downloaded in a meaningful amount of time, this is usually not true for daily return data. The daily CRSP data file is substantially larger than monthly data and can exceed 20GB. This has two important implications: You cannot hold all the daily return data in your memory (hence it is not possible to copy the entire data set to your local database), and in our experience, the download usually crashes (or never stops) because it is too much data for the WRDS cloud to prepare and send to your R session.

There is a solution to this challenge. As with many 'big data' problems, you can split up the big task into several smaller tasks that are easy to handle. That is, instead of downloading data about many stocks all at once, download the data in small batches for each stock consecutively. Such operations can be implemented in `for` loops, where we download, prepare, and store the data for a single stock in each iteration. This operation might nonetheless take a couple of hours, so you have to be patient either way (we often run such code overnight). To keep track of the progress, you can use `txtProgressBar()`. Eventually, we end up with more than 68 million rows of daily return data. Note that we only store the identifying information that we ac-

tually need, namely `permno`, `date`, and `month` alongside the excess returns. We thus ensure that our local database contains only the data we actually use and that we can load the full daily data into our memory later.

```
dsf_db <- tbl(wrds, in_schema("crsp", "dsf"))
permnos <- tbl(tidy_finance, "crsp_monthly") %>%
  distinct(permno) %>%
  pull()

progress <- txtProgressBar(min = 0, max = length(permnos), initial = 0, style = 3)
for (j in 1:length(permnos)) {
  permno_sub <- permnos[j]
  crsp_daily_sub <- dsf_db %>%
    filter(permno == permno_sub &
      date >= start_date & date <= end_date) %>%
    select(permno, date, ret) %>%
    collect() %>%
    drop_na()

  if (nrow(crsp_daily_sub)) {
    crsp_daily_sub <- crsp_daily_sub %>%
      mutate(month = floor_date(date, "month")) %>%
      left_join(factors_ff_daily %>%
        select(date, rf), by = "date") %>%
      mutate(
        ret_excess = ret - rf,
        ret_excess = pmax(ret_excess, -1)
      ) %>%
      select(permno, date, month, ret_excess)

    if (j == 1) {
      overwrite <- TRUE
      append <- FALSE
    } else {
      overwrite <- FALSE
      append <- TRUE
    }

    crsp_daily_sub %>%
      dbWriteTable(tidy_finance, "crsp_daily", ., overwrite = overwrite, append = append)
  }
  setTxtProgressBar(progress, j)
}
close(progress)
```

```
crsp_daily_db <- tbl(tidy_finance, "crsp_daily")
```

2.9 Preparing Compustat data

Firm accounting data are an important source of information that we use in portfolio analyses in subsequent chapters. The commonly used source for firm financial information is Compustat provided by S&P Global Market Intelligence[18], which is a global data vendor that provides financial, statistical, and market information on active and inactive companies throughout the world. For US and Canadian companies, annual history is available back to 1950 and quarterly as well as monthly histories date back to 1962.

To access Compustat data, we can again tap WRDS, which hosts the `funda` table that contains annual firm-level information on North American companies.

```
funda_db <- tbl(wrds, in_schema("comp", "funda"))
```

We follow the typical filter conventions and pull only data that we actually need: (i) we get only industrial fundamental data (i.e., ignore financial services) (ii) in the standard format (i.e., consolidated information in standard presentation), and (iii) only data in the desired time window.

```
compustat <- funda_db %>%
  filter(
    indfmt == "INDL" &
      datafmt == "STD" &
      consol == "C" &
      datadate >= start_date & datadate <= end_date
  ) %>%
  select(
    gvkey, # Firm identifier
    datadate, # Date of the accounting data
    seq, # Stockholders' equity
    ceq, # Total common/ordinary equity
    at, # Total assets
    lt, # Total liabilities
    txditc, # Deferred taxes and investment tax credit
    txdb, # Deferred taxes
    itcb, # Investment tax credit
```

[18] https://www.spglobal.com/marketintelligence/en/

```
    pstkrv, # Preferred stock redemption value
    pstkl, # Preferred stock liquidating value
    pstk # Preferred stock par value
  ) %>%
  collect()
```

Next, we calculate the book value of preferred stock and equity inspired by the variable definition in Ken French's data library[19]. Note that we set negative or zero equity to missing as it makes conceptually little sense (i.e., the firm would be bankrupt).

```
compustat <- compustat %>%
  mutate(
    be = coalesce(seq, ceq + pstk, at - lt) +
      coalesce(txditc, txdb + itcb, 0) -
      coalesce(pstkrv, pstkl, pstk, 0),
    be = if_else(be <= 0, as.numeric(NA), be)
  )
```

We keep only the last available information for each firm-year group. Note that `datadate` defines the time the corresponding financial data refers to (e.g., annual report as of December 31, 2020). Therefore, `datadate` is not the date when data was made available to the public. Check out the exercises for more insights into the peculiarities of `datadate`.

```
compustat <- compustat %>%
  mutate(year = year(datadate)) %>%
  group_by(gvkey, year) %>%
  filter(datadate == max(datadate)) %>%
  ungroup()
```

With the last step, we are already done preparing the firm fundamentals. Thus, we can store them in our local database.

```
compustat %>%
  dbWriteTable(tidy_finance, "compustat", ., overwrite = TRUE)
```

[19] https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/Data_Library/variable_definitions.html

2.10 Merging CRSP with Compustat

Unfortunately, CRSP and Compustat use different keys to identify stocks and firms. CRSP uses `permno` for stocks, while Compustat uses `gvkey` to identify firms. Fortunately, a curated matching table on WRDS allows us to merge CRSP and Compustat, so we create a connection to the *CRSP-Compustat Merged* table (provided by CRSP).

```
ccmxpf_linktable_db <- tbl(wrds, in_schema("crsp", "ccmxpf_linktable"))
```

The linking table contains links between CRSP and Compustat identifiers from various approaches. However, we need to make sure that we keep only relevant and correct links, again following the description outlined in BEM. Note also that currently active links have no end date, so we just enter the current date via `Sys.Date()`.

```
ccmxpf_linktable <- ccmxpf_linktable_db %>%
  filter(linktype %in% c("LU", "LC") &
    linkprim %in% c("P", "C") &
    usedflag == 1) %>%
  select(permno = lpermno, gvkey, linkdt, linkenddt) %>%
  collect() %>%
  mutate(linkenddt = replace_na(linkenddt, Sys.Date()))
ccmxpf_linktable
```

```
## # A tibble: 31,770 x 4
##   permno gvkey  linkdt     linkenddt
##    <dbl> <chr>  <date>     <date>
## 1  25881 001000 1970-11-13 1978-06-30
## 2  10015 001001 1983-09-20 1986-07-31
## 3  10023 001002 1972-12-14 1973-06-05
## 4  10031 001003 1983-12-07 1989-08-16
## 5  54594 001004 1972-04-24 2022-07-08
## # ... with 31,765 more rows
```

We use these links to create a new table with a mapping between stock identifier, firm identifier, and month. We then add these links to the Compustat `gvkey` to our monthly stock data.

```
ccm_links <- crsp_monthly %>%
  inner_join(ccmxpf_linktable, by = "permno") %>%
  filter(!is.na(gvkey) & (date >= linkdt & date <= linkenddt)) %>%
  select(permno, gvkey, date)
```

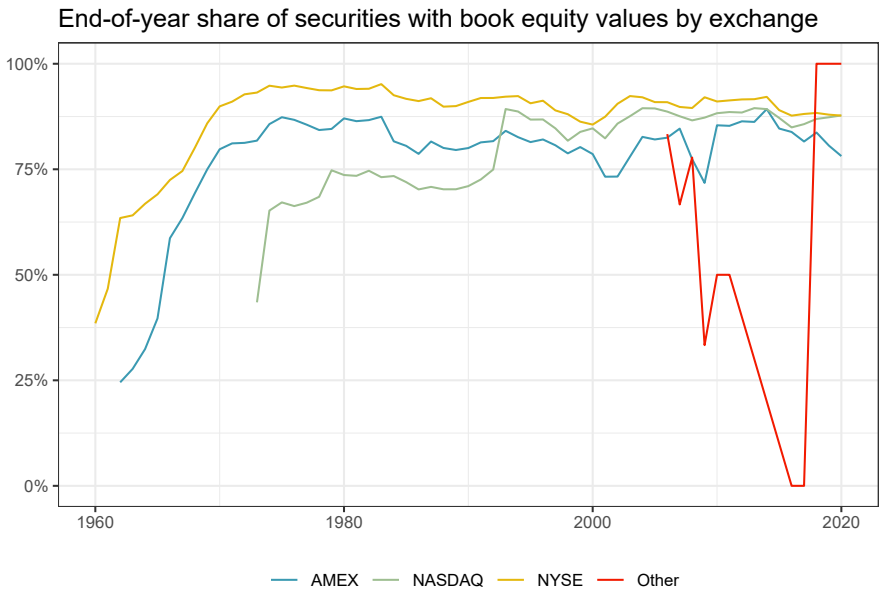
```
crsp_monthly <- crsp_monthly %>%
  left_join(ccm_links, by = c("permno", "date"))
```

As the last step, we update the previously prepared monthly CRSP file with the linking information in our local database.

```
crsp_monthly %>%
  dbWriteTable(tidy_finance, "crsp_monthly", ., overwrite = TRUE)
```

Before we close this chapter, let us look at an interesting descriptive statistic of our data. As the book value of equity plays a crucial role in many asset pricing applications, it is interesting to know for how many of our stocks this information is available. Hence, the figure below plots the share of securities with book equity values for each exchange. It turns out that the coverage is pretty bad for AMEX- and NYSE-listed stocks in the 60s but hovers around 80% for all periods thereafter. We can ignore the erratic coverage of securities that belong to the other category since there is only a handful of them anyway in our sample.

```
crsp_monthly %>%
  group_by(permno, year = year(month)) %>%
  filter(date == max(date)) %>%
  ungroup() %>%
  left_join(compustat, by = c("gvkey", "year")) %>%
  group_by(exchange, year) %>%
  summarize(share = n_distinct(permno[!is.na(be)]) / n_distinct(permno)) %>%
  ggplot(aes(x = year, y = share, color = exchange)) +
  geom_line() +
  labs(
    x = NULL, y = NULL, color = NULL, linetype = NULL,
    title = "End-of-year share of securities with book equity values by exchange"
  ) +
  scale_y_continuous(labels = percent) +
  coord_cartesian(ylim = c(0, 1))
```



2.11 Managing SQLite databases

Finally, at the end of our data chapter, we revisit the SQLite database itself. When you drop database objects such as tables or delete data from tables, the database file size remains unchanged because SQLite just marks the deleted objects as free and reserves their space for the future uses. As a result, the database file always grows in size.

To optimize the database file, you can run the `VACUUM` command in the database, which rebuilds the database and frees up unused space. You can execute the command in the database using the `dbSendQuery()` function.

```
dbSendQuery(tidy_finance, "VACUUM")
```

```
## <SQLiteResult>
##   SQL  VACUUM
##   ROWS Fetched: 0 [complete]
##        Changed: 0
```

The `VACUUM` command actually performs a couple of additional cleaning steps, which you can read up in this tutorial[20].

Apart from cleaning up, you might be interested in listing all the tables that are currently in your database. You can do this via the `dbListTables()` function.

```
dbListTables(tidy_finance)
```

```
## Warning: Closing open result set, pending rows
```

```
##  [1] "beta"                 "compustat"
##  [3] "cpi_monthly"          "crsp_daily"
##  [5] "crsp_monthly"         "factors_ff_daily"
##  [7] "factors_ff_monthly"   "factors_q_monthly"
##  [9] "industries_ff_monthly" "macro_predictors"
```

This function comes in handy if you are unsure about the correct naming of the tables in your database.

2.12 Some tricks for PostgreSQL databases

As we mentioned above, the WRDS database runs on PostgreSQL rather than SQLite. Finding the right tables for your data needs can be tricky in the WRDS PostgreSQL instance, as the tables are organized in schemas. If you wonder what the purpose of schemas is, check out this documetation[21]. For instance, if you want to find all tables that live in the `crsp` schema, you run

```
dbListObjects(wrds, Id(schema = "crsp"))
```

This operation returns a list of all tables that belong to the `crsp` family on WRSD, e.g. `<Id> schema = crsp, table = msenames`. Similarly, you can fetch a list of all tables that belong to the `comp` family via

```
dbListObjects(wrds, Id(schema = "comp"))
```

If you want to get all schemas, then run

[20] https://www.sqlitetutorial.net/sqlite-vacuum/
[21] https://www.postgresql.org/docs/9.1/ddl-schemas.html

```
dbListObjects(wrds)
```

2.13 Exercises

1. Download the monthly Fama-French factors manually from Ken French's data library[22] and read them in via `read_csv()`. Validate that you get the same data as via the `frenchdata` package.
2. Check out the structure of the WRDS database by sending queries in the spirit of "Querying WRDS Data using R"[23] and verify the output with `dbListObjects()`. How many tables are associated with CRSP? Can you identify what is stored within *msp500*?
3. Compute `mkt_cap_lag` using `lag(mktcap)` rather than joins as above. Filter out all the rows where the lag-based market capitalization measure is different from the one we computed above. Why are they different?
4. In the main part, we look at the distribution of market capitalization across exchanges and industries. Now, plot the average market capitalization of firms for each exchange and industry. What do you find?
5. `datadate` refers to the date to which the fiscal year of a corresponding firm refers to. Count the number of observations in Compustat by *month* of this date variable. What do you find? What does the finding suggest about pooling observations with the same fiscal year?
6. Go back to the original Compustat data in `funda_db` and extract rows where the same firm has multiple rows for the same fiscal year. What is the reason for these observations?
7. Repeat the analysis of market capitalization for book equity, which we computed from the Compustat data. Then, used the matched sample to plot book equity against market capitalization. How are these two variables related?

[22] https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html

[23] https://wrds-www.wharton.upenn.edu/pages/support/programming-wrds/programming-r/querying-wrds-data-r/

Asset pricing

3

Beta estimation

In this chapter, we introduce you to an important concept in financial economics: the exposure of an individual stock to changes in the market portfolio. According to the Capital Asset Pricing Model (CAPM), cross-sectional variation in expected asset returns should be a function of the covariance between the excess return of the asset and the excess return on the market portfolio. The regression coefficient of market returns on excess returns is usually called the market beta. In this chapter, we show an estimation procedure for the market betas. We do not go into details about the foundations of market beta but simply refer to any treatment of the CAPM[1] for further information. Instead, we provide details about all the functions that we use to compute the results. In particular, we leverage useful computational concepts: rolling-window estimation and parallelization.

We use the following packages throughout this chapter:

```
library(tidyverse)
library(RSQLite)
library(slider)
library(scales)
library(furrr)
```

3.1 Estimating beta using monthly returns

The estimation procedure is based on a rolling-window estimation where we may use either monthly or daily returns and different window lengths. First, let us start with loading the monthly data we prepared in the previous chapter from our `SQLite`-database introduced in our chapter on *"Accessing & managing financial data"*.

```
tidy_finance <- dbConnect(SQLite(), "data/tidy_finance.sqlite",
  extended_types = TRUE
)
```

[1] https://en.wikipedia.org/wiki/Capital_asset_pricing_model

```
crsp_monthly <- tbl(tidy_finance, "crsp_monthly") %>%
  collect()

factors_ff_monthly <- tbl(tidy_finance, "factors_ff_monthly") %>%
  collect()

crsp_monthly <- crsp_monthly %>%
  left_join(factors_ff_monthly, by = "month") %>%
  select(permno, month, industry, ret_excess, mkt_excess)
```

To estimate the CAPM equation

$$r_{i,t} - r_{f,t} = \alpha_i + \beta_i(r_{m,t} - r_{f,t}) + \varepsilon_{i,t}$$

we regress excess stock returns `ret_excess` on excess returns of the market portfolio `mkt_excess`. R provides a simple solution to estimate (linear) models with the function `lm()`. `lm()` requires a formula as input that is specified in a compact symbolic form. An expression of the form `y ~ model` is interpreted as a specification that the response `y` is modeled by a linear predictor specified symbolically by `model`. Such a model consists of a series of terms separated by `+` operators. In addition to standard linear models, `lm()` provides a lot of flexibility. You should check out the documentation for more information. To start, we restrict the data only to the time series of observations in CRSP that correspond to Apple's stock (i.e., to `permno` 14593 for Apple) and compute α_i as well as β_i.

```
fit <- lm(ret_excess ~ mkt_excess,
  data = crsp_monthly %>%
    filter(permno == "14593")
)

summary(fit)
```

```
##
## Call:
## lm(formula = ret_excess ~ mkt_excess, data = crsp_monthly %>%
##     filter(permno == "14593"))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -0.5167 -0.0610  0.0009  0.0643  0.3940
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.01051    0.00532    1.98    0.049 *
## mkt_excess   1.40081    0.11748   11.92   <2e-16 ***
```

```
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.115 on 478 degrees of freedom
## Multiple R-squared:  0.229,  Adjusted R-squared:  0.228
## F-statistic:  142 on 1 and 478 DF,  p-value: <2e-16
```

`lm()` returns an object of class `lm` which contains all information we usually care about with linear models. `summary()` returns an overview of the estimated parameters. `coefficients(fit)` would return only the estimated coefficients. The output above indicates that Apple moves excessively with the market as the estimated β_i is above one ($\widehat{\beta}_i = 1.4$).

3.2 Rolling-window estimation

After we estimated the regression coefficients on an example, we scale the estimation of β_i to a whole different level and perform rolling-window estimations for the entire CRSP sample. The following function implements the CAPM regression for a data frame (or a part thereof) containing at least `min_obs` observations to avoid huge fluctuations if the time series is too short. If the condition is violated, the function returns a missing value.

```
estimate_capm <- function(data, min_obs = 1) {
  if (nrow(data) < min_obs) {
    beta <- as.numeric(NA)
  } else {
    fit <- lm(ret_excess ~ mkt_excess, data = data)
    beta <- as.numeric(fit$coefficients[2])
  }
  return(beta)
}
```

Next, we define a function that does the rolling estimation. To perform the rolling-window estimation, we use the `slider` package of Davis Vaughan[2]. The `slide_period` function is able to handle months in its window input in a straightforward manner. We thus avoid using any time-series package (e.g., `zoo`) and converting the data to fit the package functions, but rather stay in the world of tibbles.

[2] https://github.com/DavisVaughan/slider

The following function takes input data and slides across the month vector, considering only a total of `months` months. The function essentially performs three steps: (i) combine all rows into a single data frame (which comes in handy in the case of daily data), (ii) compute betas by sliding across months, and (iii) return a tibble with months and corresponding beta estimates (again particularly useful in the case of daily data). As we demonstrate further below, we can also apply the same function to daily returns data.

```
roll_capm_estimation <- function(data, months, min_obs) {
  data <- bind_rows(data) %>%
    arrange(month)

  betas <- slide_period_vec(
    .x = data,
    .i = data$month,
    .period = "month",
    .f = ~ estimate_capm(., min_obs),
    .before = months - 1,
    .complete = FALSE
  )

  tibble(
    month = unique(data$month),
    beta = betas
  )
}
```

Before we attack the whole CRSP sample, let us focus on a couple of examples for well-known firms.

```
examples <- tribble(
  ~permno, ~company,
  14593, "Apple",
  10107, "Microsoft",
  93436, "Tesla",
  17778, "Berkshire Hathaway"
)
```

If we want to estimate rolling betas for Apple, we can use `mutate()`. We take a total of 5 years of data and require at least 48 months with return data to compute our betas. Check out the exercises if you want ot compute beta for different time periods.

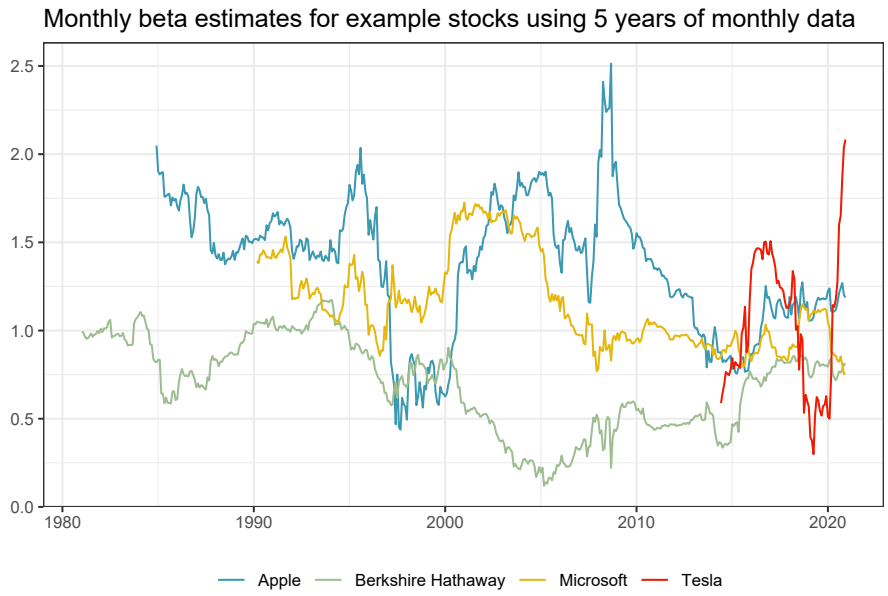
```
beta_example <- crsp_monthly %>%
  filter(permno == examples$permno[1]) %>%
  mutate(roll_capm_estimation(cur_data(), months = 60, min_obs = 48)) %>%
```

```
  drop_na()
beta_example
```

```
## # A tibble: 433 x 6
##   permno month      industry      ret_excess mkt_excess
##    <dbl> <date>     <chr>              <dbl>      <dbl>
## 1  14593 1984-12-01 Manufacturing     0.170      0.0184
## 2  14593 1985-01-01 Manufacturing    -0.0108     0.0799
## 3  14593 1985-02-01 Manufacturing    -0.152      0.0122
## 4  14593 1985-03-01 Manufacturing    -0.112     -0.0084
## 5  14593 1985-04-01 Manufacturing    -0.0467    -0.0096
## # ... with 428 more rows, and 1 more variable:
## #   beta <dbl>
```

It is actually quite simple to perform the rolling-window estimation for an arbitrary number of stocks, which we visualize in the following code chunk.

```
beta_examples <- crsp_monthly %>%
  inner_join(examples, by = "permno") %>%
  group_by(permno) %>%
  mutate(roll_capm_estimation(cur_data(), months = 60, min_obs = 48)) %>%
  ungroup() %>%
  select(permno, company, month, beta_monthly = beta) %>%
  drop_na()

beta_examples %>%
  ggplot(aes(x = month, y = beta_monthly, color = company)) +
  geom_line() +
  labs(
    x = NULL, y = NULL, color = NULL,
    title = "Monthly beta estimates for example stocks using 5 years of monthly data"
  )
```



3.3 Parallelized rolling-window estimation

Even though we could now just apply the function using `group_by()` on the whole CRSP sample, we advise against doing it as it is computationally quite expensive. Remember that we have to perform rolling-window estimations across all stocks and time periods. However, this estimation problem is an ideal scenario to employ the power of parallelization. Parallelization means that we split the tasks which perform rolling-window estimations across different workers (or cores on your local machine).

First, we `nest()` the data by `permno`. Nested data means we now have a list of `permno` with corresponding time series data.

```
crsp_monthly_nested <- crsp_monthly %>%
  nest(data = c(month, ret_excess, mkt_excess))
crsp_monthly_nested
```

```
## # A tibble: 29,203 x 3
##   permno industry      data
##    <dbl> <chr>         <list>
## 1  10000 Manufacturing <tibble [16 x 3]>
```

```
## 2  10001 Utilities     <tibble [378 x 3]>
## 3  10002 Finance       <tibble [324 x 3]>
## 4  10003 Finance       <tibble [118 x 3]>
## 5  10005 Mining        <tibble [65 x 3]>
## # ... with 29,198 more rows
```

Next, we ant to apply the `roll_capm_estimation()` function to each stock. This situation is an ideal use case for `map()`, which takes a list or vector as input and returns an object of the same length as the input. In our case, `map()` returns a single data frame with a time series of beta estimates for each stock. Therefore, we use `unnest()` to transform the list of outputs to a tidy data frame.

```
crsp_monthly_nested %>%
  inner_join(examples, by = "permno") %>%
  mutate(beta = map(data, ~ roll_capm_estimation(., months = 60, min_obs = 48))) %>%
  unnest(c(beta)) %>%
  select(permno, month, beta_monthly = beta) %>%
  drop_na()
```

```
## # A tibble: 1,362 x 3
##   permno month      beta_monthly
##    <dbl> <date>            <dbl>
## 1  10107 1990-03-01         1.39
## 2  10107 1990-04-01         1.38
## 3  10107 1990-05-01         1.43
## 4  10107 1990-06-01         1.43
## 5  10107 1990-07-01         1.45
## # ... with 1,357 more rows
```

However, instead, we want to perform the estimations of rolling betas for different stocks in parallel. We can use the flexibility of the `future` package, which we use to define how we want to perform the parallel estimation. If you have a Windows machine, it makes most sense to define `multisession`, which means that separate R processes are running in the background on the same machine to perform the individual jobs. If you check out the documentation of `plan()`, you can also see other ways to resolve the parallelization.

```
plan(multisession, workers = availableCores())
```

Using eight cores, the estimation for our sample of around 25k stocks takes around 20 minutes. Of course, you can speed up things considerably by having more cores available to share the workload or by having more powerful cores. Notice the difference in the code below? All you need to do is to replace `map()` with `future_map()`.

```
beta_monthly <- crsp_monthly_nested %>%
  mutate(beta = future_map(data, ~ roll_capm_estimation(., months = 60, min_obs = 48))) %>%
  unnest(c(beta)) %>%
  select(permno, month, beta_monthly = beta) %>%
  drop_na()
```

3.4 Estimating beta using daily returns

Before we provide some descriptive statistics of our beta estimates, we implement the estimation for the daily CRSP sample as well. Depending on the application, you might either use longer horizon beta estimates based on monthly data or shorter horizon estimates based on daily returns.

First, we load daily CRSP data. Note that the sample is large compared to the monthly data, so make sure to have enough memory available.

```
crsp_daily <- tbl(tidy_finance, "crsp_daily") %>%
  collect()
```

We also need the daily Fama-French market excess returns.

```
factors_ff_daily <- tbl(tidy_finance, "factors_ff_daily") %>%
  collect()
```

We make sure to keep only relevant data to save memory space. However, note that your machine might not have enough memory to read the whole daily CRSP sample. In this case, we refer you to the exercises and try working with loops as in chapter 2.

```
crsp_daily <- crsp_daily %>%
  inner_join(factors_ff_daily, by = "date") %>%
  select(permno, month, ret_excess, mkt_excess)
```

Just like above, we nest the data by `permno` for parallelization.

```
crsp_daily_nested <- crsp_daily %>%
  nest(data = c(month, ret_excess, mkt_excess))
```

This is what the estimation looks like for a couple of examples using `map()`. For the daily data, we use the same function as above but only take 3 months of data and require at least 50 daily return observations in these months. These restrictions help us to retrieve somehow smooth coefficient estimates.

```
crsp_daily_nested %>%
  inner_join(examples, by = "permno") %>%
  mutate(beta_daily = map(data, ~ roll_capm_estimation(., months = 3, min_obs = 50))) %>%
  unnest(c(beta_daily)) %>%
  select(permno, month, beta_daily = beta) %>%
  drop_na()
```

```
## # A tibble: 1,543 x 3
##   permno month      beta_daily
##    <dbl> <date>          <dbl>
## 1  10107 1986-05-01      0.898
## 2  10107 1986-06-01      0.906
## 3  10107 1986-07-01      0.822
## 4  10107 1986-08-01      0.900
## 5  10107 1986-09-01      1.01
## # ... with 1,538 more rows
```

For the sake of completeness, we tell our session again to use multiple workers for parallelization.

```
plan(multisession, workers = availableCores())
```

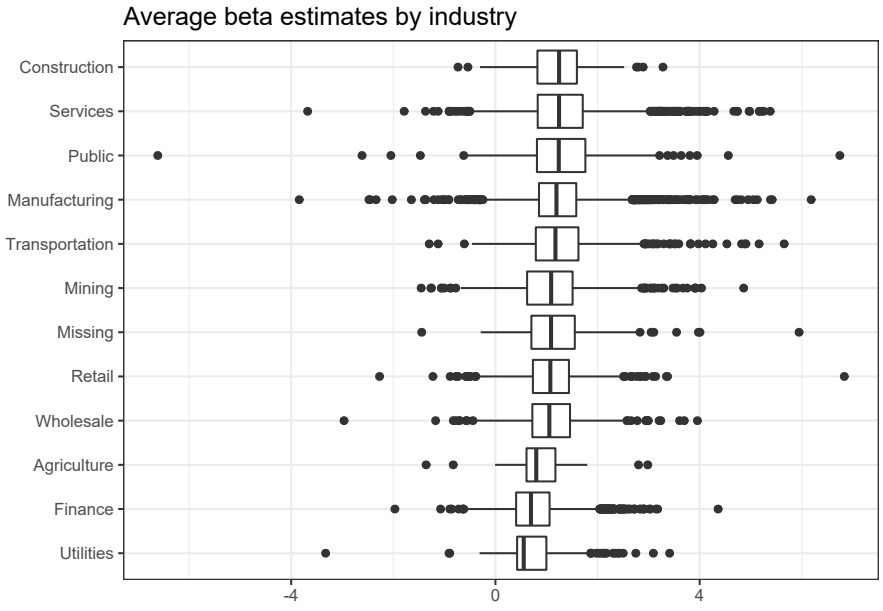
The code chunk for beta estimation using daily returns now looks very similar to the one for monthly data. The whole estimation takes around 30 minutes using eight cores and 32gb memory.

```
beta_daily <- crsp_daily_nested %>%
  mutate(beta_daily = future_map(data, ~ roll_capm_estimation(., months = 3, min_obs = 50))) %>%
  unnest(c(beta_daily)) %>%
  select(permno, month, beta_daily = beta) %>%
  drop_na()
```

3.5 Comparing beta estimates

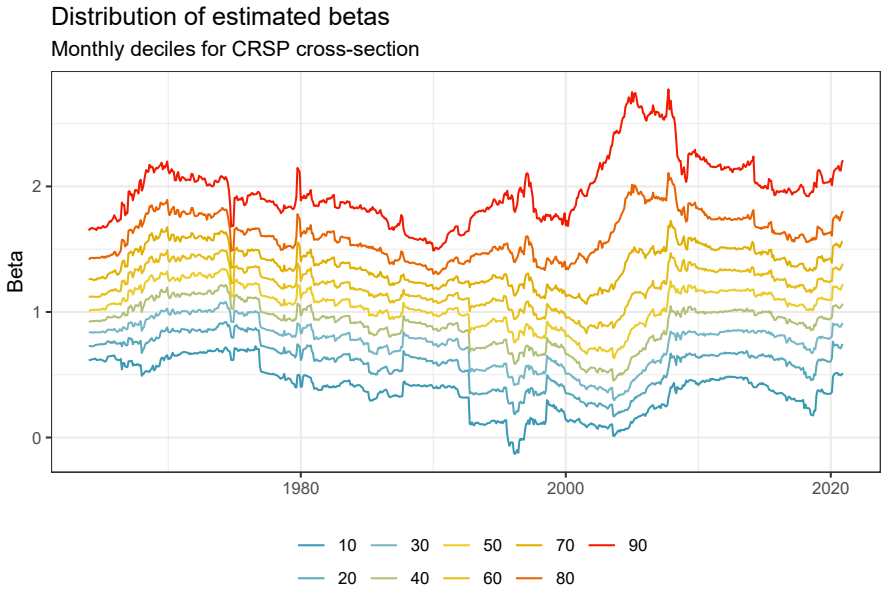
What is a typical value for stock betas? To get some feeling, we illustrate the dispersion of the estimated $\hat{\beta}_i$ across different industries and across time below. The first figure below shows that typical business models across industries imply different exposure to the general market economy. However, there are barely any firms that exhibit a negative exposure to the market factor.

```
crsp_monthly %>%
  left_join(beta_monthly, by = c("permno", "month")) %>%
  drop_na(beta_monthly) %>%
  group_by(industry, permno) %>%
  summarize(beta = mean(beta_monthly)) %>%
  ggplot(aes(x = reorder(industry, beta, FUN = median), y = beta)) +
  geom_boxplot() +
  coord_flip() +
  labs(
    x = NULL, y = NULL,
    title = "Average beta estimates by industry"
  )
```



Next, we illustrate the time-variation in the cross-section of estimated betas. The figure below shows the monthly deciles of estimated betas (based on monthly data) and indicates an interesting pattern: First, betas seem to vary over time in the sense that during some periods, there is a clear trend across all deciles. Second, the sample exhibits periods where the dispersion across stocks increases in the sense that the lower decile decreases and the upper decile increases, which indicates that for some stocks the correlation with the market increases while for others it decreases. Note also here: stocks with negative betas are an extremely rare exception.

```
beta_monthly %>%
  drop_na(beta_monthly) %>%
  group_by(month) %>%
  summarize(
    x = quantile(beta_monthly, seq(0.1, 0.9, 0.1)),
    quantile = 100 * seq(0.1, 0.9, 0.1),
    .groups = "drop"
  ) %>%
  ggplot(aes(x = month, y = x, color = as_factor(quantile))) +
  geom_line() +
  labs(
    x = NULL, y = "Beta", color = NULL,
    title = "Distribution of estimated betas",
    subtitle = "Monthly deciles for CRSP cross-section"
  )
```

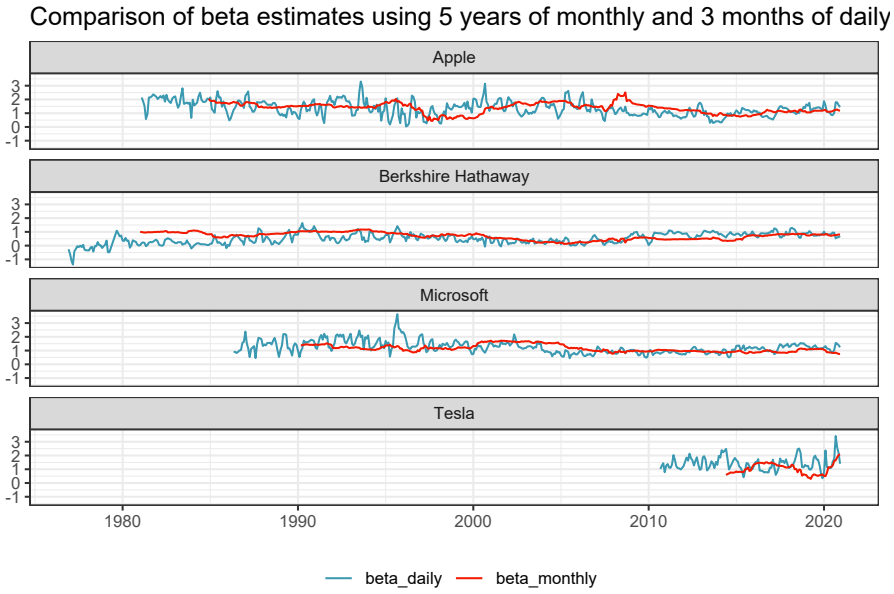


To compare the difference between daily and monthly data, we combine beta estimates to a single table. Then, we use the table to plot a comparison of beta estimates for our example stocks.

```
beta <- beta_monthly %>%
  full_join(beta_daily, by = c("permno", "month")) %>%
  arrange(permno, month)
```

```
beta %>%
  inner_join(examples, by = "permno") %>%
  pivot_longer(cols = c(beta_monthly, beta_daily)) %>%
  ggplot(aes(x = month, y = value, color = name)) +
  geom_line() +
  facet_wrap(~company, ncol = 1) +
  labs(
    x = NULL, y = NULL, color = NULL,
    title = "Comparison of beta estimates using 5 years of monthly and 3 months of daily data"
  )
```

```
## Warning: Removed 46 row(s) containing missing values
## (geom_path).
```



The estimates look as expected. As you can see, it really depends on the estimation window and data frequency how your beta estimates turn out.

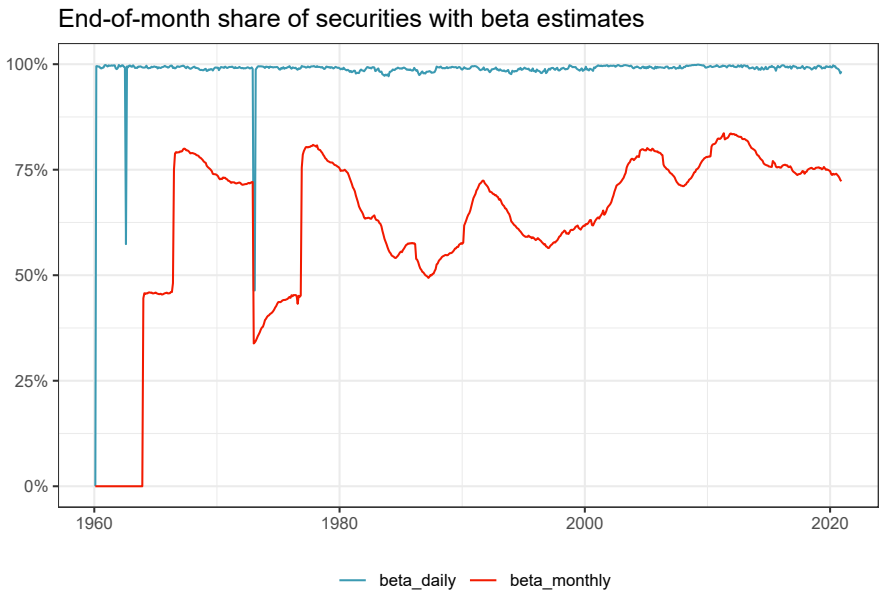
Finally, we write the estimates to our database such that we can use them in later chapters.

```
beta %>%
  dbWriteTable(tidy_finance, "beta", ., overwrite = TRUE)
```

Whenever you perform some kind of estimation, it also makes sense to do rough

plausibility tests. A possible check is to plot the share of stocks with beta estimates over time. This descriptive helps us discover potential errors in our data preparation or estimation procedure. For instance, suppose there was a gap in our output where we do not have any betas. In this case, we would have to go back and check all previous steps.

```
beta_long <- crsp_monthly %>%
  left_join(beta, by = c("permno", "month")) %>%
  pivot_longer(cols = c(beta_monthly, beta_daily))

beta_long %>%
  group_by(month, name) %>%
  summarize(share = sum(!is.na(value)) / n()) %>%
  ggplot(aes(x = month, y = share, color = name)) +
  geom_line() +
  scale_y_continuous(labels = percent) +
  labs(
    x = NULL, y = NULL, color = NULL,
    title = "End-of-month share of securities with beta estimates"
  ) +
  coord_cartesian(ylim = c(0, 1))
```



The figure above does not indicate any troubles, so let us move on to the next check.

We also encourage everyone to always look at the distributional summary statistics

of variables. You can easily spot outliers or weird distributions when looking at such tables.

```
beta_long %>%
  select(name, value) %>%
  drop_na() %>%
  group_by(name) %>%
  summarize(
    mean = mean(value),
    sd = sd(value),
    min = min(value),
    q05 = quantile(value, 0.05),
    q25 = quantile(value, 0.25),
    q50 = quantile(value, 0.50),
    q75 = quantile(value, 0.75),
    q95 = quantile(value, 0.95),
    max = max(value),
    n = n()
  )
```

```
## # A tibble: 2 x 11
##   name        mean    sd   min    q05   q25   q50   q75
##   <chr>      <dbl> <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl>
## 1 beta_daily 0.743 0.925 -43.7 -0.452 0.203 0.679  1.22
## 2 beta_mont~ 1.10  0.711 -13.0  0.123 0.631 1.03   1.47
## # ... with 3 more variables: q95 <dbl>, max <dbl>,
## #   n <int>
```

Finally, since we have two different estimators for the same theoretical object, we expect the estimators should be at least positively correlated (although not perfectly as the estimators are based on different sample periods).

```
beta %>%
  select(beta_daily, beta_monthly) %>%
  cor(., use = "complete.obs")
```

```
##              beta_daily beta_monthly
## beta_daily        1.000        0.322
## beta_monthly      0.322        1.000
```

Indeed, we find a positive correlation between our beta estimates. In the subsequent chapters, we mainly use the estimates based on monthly data as most readers should be able to replicate them due to potential memory limitations that might arise with the daily data.

3.6 Exercises

1. Compute beta estimates based on monthly data using 1, 3, and 5 years of data and impose a minimum number of observations of 10, 28, and 48 months with return data, respectively. How strongly correlated are the estimated betas?
2. Compute beta estimates based on monthly data using 5 years of data and impose different numbers of minimum observations. How does the share of permno-month observations with successful beta estimates vary across the different requirements? Do you find a high correlation across the estimated betas?
3. Instead of using `future_map()`, perform the beta estimation in a loop (using either monthly or daily data) for a subset of 100 permnos of your choice. Verify that you get the same results as with the parallelized code from above.
4. Filter out the stocks with negative betas. Do these stocks frequently exhibit negative betas, or do they resemble estimation errors?

4

Univariate portfolio sorts

In this chapter, we dive into portfolio sorts, one of the most widely used statistical methodologies in empirical asset pricing. The key application of portfolio sorts is to examine whether one or more variables can predict future excess returns. In general, the idea is to sort individual stocks into portfolios, where the stocks within each portfolio are similar with respect to a sorting variable, such as firm size. The different portfolios then represent well-diversified investments that differ in the level of the sorting variable. You can then attribute the differences in the return distribution to the impact of the sorting variable. We start by introducing univariate portfolio sorts (which sort based on only one characteristic). In a later chapter, we tackle bivariate sorting.

A univariate portfolio sort considers only one sorting variable $x_{t-1,i}$. Here, i denotes the stock and $t-1$ indicates that the characteristic is observable by investors at time t.
The objective is to assess the cross-sectional relation between $x_{t-1,i}$ and, typically, stock excess returns $r_{t,i}$ at time t as the outcome variable. To illustrate how portfolio sorts work, we use estimates for market betas from the previous chapter as our sorting variable.

The current chapter relies on the following set of packages.

```
library(tidyverse)
library(RSQLite)
library(lubridate)
library(sandwich)
library(lmtest)
library(scales)
```

4.1 Data preparation

We start with loading the required data from our `SQLite`-database introduced in our chapter on *"Accessing & managing financial data"*. In particular, we use the monthly

CRSP sample as our asset universe. Once we form our portfolios, we use the Fama-French factor returns to compute the risk-adjusted performance (i.e., alpha). `beta` is the tibble with market betas computed in the previous chapter.

```
tidy_finance <- dbConnect(SQLite(), "data/tidy_finance.sqlite",
  extended_types = TRUE
)

crsp_monthly <- tbl(tidy_finance, "crsp_monthly") %>%
  collect()

factors_ff_monthly <- tbl(tidy_finance, "factors_ff_monthly") %>%
  collect()

beta <- tbl(tidy_finance, "beta") %>%
  collect()
```

We keep only relevant data from the CRSP sample.

```
crsp_monthly <- crsp_monthly %>%
  left_join(factors_ff_monthly, by = "month") %>%
  select(permno, month, ret_excess, mkt_excess, mktcap_lag)
crsp_monthly
```

```
## # A tibble: 3,225,079 x 5
##   permno month      ret_excess mkt_excess mktcap_lag
##    <dbl> <date>          <dbl>      <dbl>      <dbl>
## 1  10000 1986-02-01    -0.262      0.0713       16.1
## 2  10000 1986-03-01     0.359      0.0488       12.0
## 3  10000 1986-04-01    -0.104     -0.0131       16.3
## 4  10000 1986-05-01    -0.228      0.0462       15.2
## 5  10000 1986-06-01    -0.0102     0.0103       11.8
## # ... with 3,225,074 more rows
```

4.2 Sorting by market beta

Next, we merge our sorting variable with the return data. We use the one-month *lagged* betas as a sorting variable to ensure that the sorts rely only on information available when we create the portfolios. To lag stock beta by one month, we add one month to the current date and join the resulting information with our return data. This procedure ensures that month t information is available in month $t + 1$. You

may be tempted to simply use a call such as `crsp_monthly %>% group_by(permno) %>% mutate(beta_lag = lag(beta))` instead. This procedure, however, does not work if there are non-explicit missing values in the time series.

```
beta_lag <- beta %>%
  mutate(month = month %m+% months(1)) %>%
  select(permno, month, beta_lag = beta_daily) %>%
  drop_na()

data_for_sorts <- crsp_monthly %>%
  inner_join(beta_lag, by = c("permno", "month"))
```

The first step to conduct portfolio sorts is to calculate periodic breakpoints that you can use to group the stocks into portfolios. For simplicity, we start with the median as the single breakpoint. We then compute the value-weighted returns for each of the two resulting portfolios, which means that the lagged market capitalization determines the weight in `weighted.mean()`.

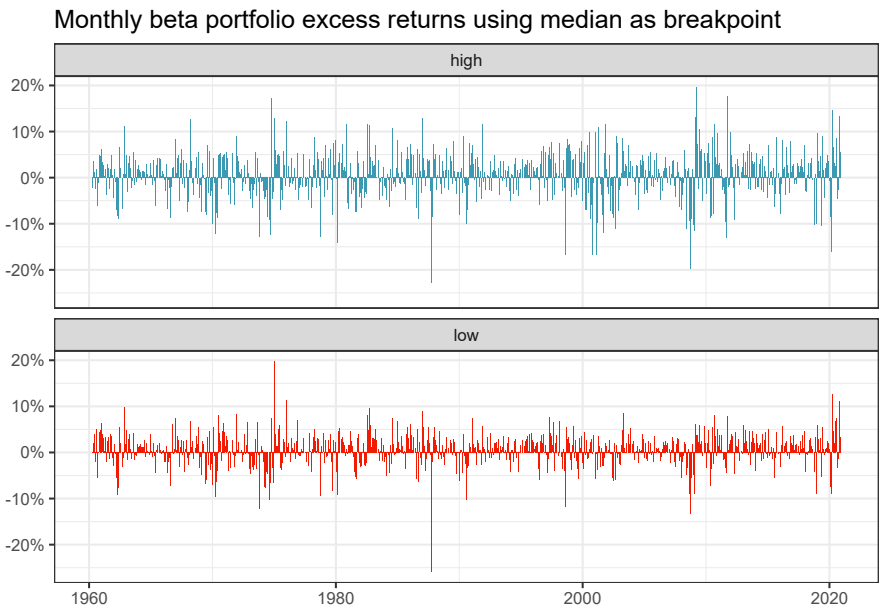
```
beta_portfolios <- data_for_sorts %>%
  group_by(month) %>%
  mutate(
    breakpoint = median(beta_lag),
    portfolio = case_when(
      beta_lag <= breakpoint ~ "low",
      beta_lag > breakpoint ~ "high"
    )
  ) %>%
  group_by(month, portfolio) %>%
  summarize(ret = weighted.mean(ret_excess, mktcap_lag), .groups = "drop")
```

4.3 Performance evaluation

The following figure shows the monthly excess returns of the two portfolios.

```
beta_portfolios %>%
  ggplot(aes(x = month, y = ret, fill = portfolio)) +
  geom_col() +
  facet_wrap(~portfolio, ncol = 1) +
  scale_y_continuous(labels = percent) +
  labs(
    x = NULL, y = NULL,
```

```
    title = "Monthly beta portfolio excess returns using median as breakpoint"
  ) +
  theme(legend.position = "none")
```



We can construct a long-short strategy based on the two portfolios: buy the high-beta portfolio and, at the same time, short the low-beta portfolio. Thereby, the overall position in the market is net-zero, i.e., you do not need to invest money to realize this strategy in the absence of frictions.

```
beta_longshort <- beta_portfolios %>%
  pivot_wider(month, names_from = portfolio, values_from = ret) %>%
  mutate(long_short = high - low) %>%
  left_join(factors_ff_monthly, by = "month")
```

We compute the average return and the corresponding standard error to test whether the long-short portfolio yields on average positive or negative excess returns. In the asset pricing literature, one typically uses Newey and West (1987) t-statistics to test the null hypothesis that average portfolio excess returns are equal to zero. To implement this test, we compute the average return via `lm()` and then employ the `coeftest` function.

```
model_fit <- lm(long_short ~ 1, data = beta_longshort)
coeftest(model_fit, vcov = NeweyWest, lag = 6)
```

```
##
## t test of coefficients:
##
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.000171   0.001005   -0.17     0.86
```

The results indicate that we cannot reject the null hypothesis of average returns being equal to zero. Our portfolio strategy using the median as a breakpoint hence does not yield any abnormal returns. Is this finding surprising if you reconsider the CAPM? It certainly is. The CAPM yields that the high beta stocks should yield higher expected returns. Our portfolio sort implicitly mimics an investment strategy that finances high beta stocks by shorting low beta stocks. Therefore, one should expect that the average excess returns yield a return that is above the risk-free rate.

4.4 Functional programming for portfolio sorts

Now we take portfolio sorts to the next level. We want to be able to sort stocks into an arbitrary number of portfolios. For this case, functional programming is very handy: we employ the curly-curly[1]-operator to give us flexibility concerning which variable to use for the sorting, denoted by `var`. We use `quantile()` to compute breakpoints for `n_portfolios`. Then, we assign portfolios to stocks using the `findInterval()` function. The output of the following function is a new column that contains the number of the portfolio to which a stock belongs.

```
assign_portfolio <- function(data, var, n_portfolios) {
  breakpoints <- data %>%
    summarize(breakpoint = quantile({{ var }},
      probs = seq(0, 1, length.out = n_portfolios + 1),
      na.rm = TRUE
    )) %>%
    pull(breakpoint) %>%
    as.numeric()

  data %>%
    mutate(portfolio = findInterval({{ var }},
      breakpoints,
      all.inside = TRUE
    )) %>%
    pull(portfolio)
}
```

[1] https://www.tidyverse.org/blog/2019/06/rlang-0-4-0/#a-simpler-interpolation-pattern-with-

We can use the above function to sort stocks into ten portfolios each month using lagged betas and compute value-weighted returns for each portfolio. Note that we transform the portfolio column to a factor variable because it provides more convenience for the figure construction below.

```
beta_portfolios <- data_for_sorts %>%
  group_by(month) %>%
  mutate(
    portfolio = assign_portfolio(
      data = cur_data(),
      var = beta_lag,
      n_portfolios = 10
    ),
    portfolio = as.factor(portfolio)
  ) %>%
  group_by(portfolio, month) %>%
  summarize(ret = weighted.mean(ret_excess, mktcap_lag), .groups = "drop")
```

4.5 More performance evaluation

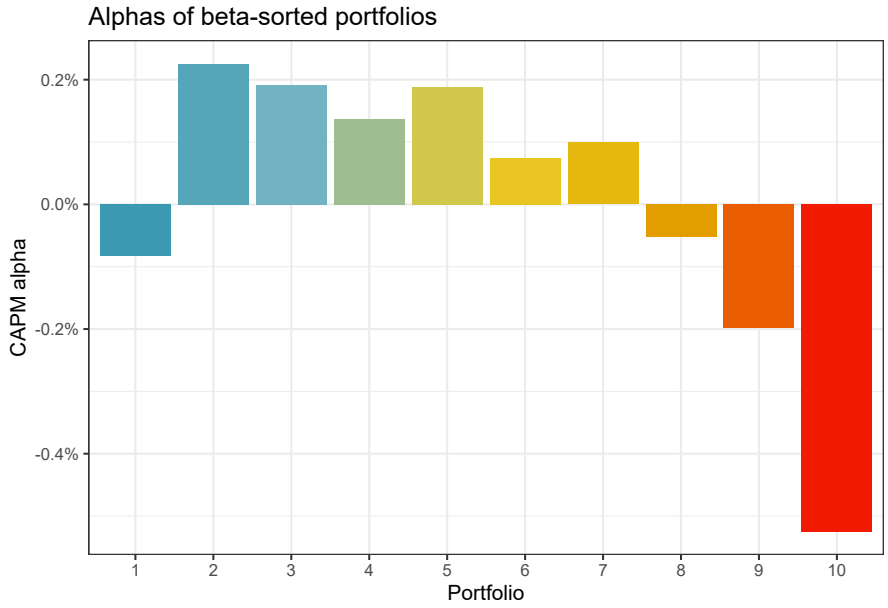
In the next step, we compute summary statistics for each beta portfolio. Namely, we compute CAPM-adjusted alphas, the beta of each beta portfolio, and average returns.

```
beta_portfolios_summary <- beta_portfolios %>%
  left_join(factors_ff_monthly, by = "month") %>%
  group_by(portfolio) %>%
  summarize(
    alpha = as.numeric(lm(ret ~ 1 + mkt_excess)$coefficients[1]),
    beta = as.numeric(lm(ret ~ 1 + mkt_excess)$coefficients[2]),
    ret = mean(ret)
  )
```

The figure below illustrates the CAPM alphas of beta-sorted portfolios. It shows that low beta portfolios tend to exhibit positive alphas, while high beta portfolios exhibit negative alphas.

```
beta_portfolios_summary %>%
  ggplot(aes(x = portfolio, y = alpha, fill = portfolio)) +
  geom_bar(stat = "identity") +
  labs(
```

```
    title = "Alphas of beta-sorted portfolios",
    x = "Portfolio",
    y = "CAPM alpha",
    fill = "Portfolio"
) +
scale_y_continuous(labels = percent) +
theme(legend.position = "None")
```

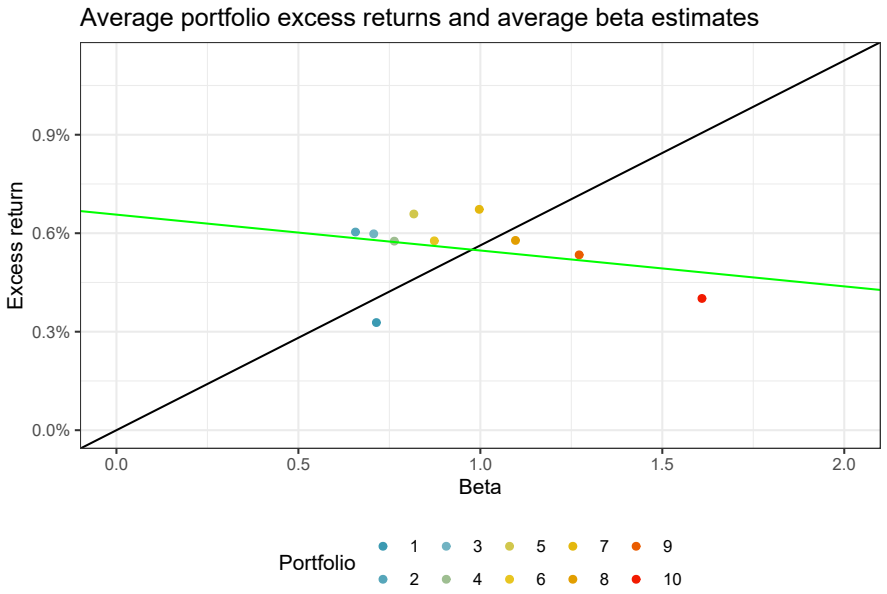


These results suggest a negative relation between beta and future stock returns, which contradicts the predictions of the CAPM. According to the CAPM, returns should increase with beta across the portfolios and risk-adjusted returns should be statistically indistinguishable from zero.

4.6 The security market line and beta portfolios

The CAPM predicts that our portfolios should lie on the security market line (SML). The slope of the SML is equal to the market risk premium and reflects the risk-return trade-off at any given time.

```
sml_capm <- lm(ret ~ 1 + beta, data = beta_portfolios_summary)$coefficients

beta_portfolios_summary %>%
  ggplot(aes(x = beta, y = ret, color = portfolio)) +
  geom_point() +
  geom_abline(intercept = 0, slope = mean(factors_ff_monthly$mkt_excess)) +
  geom_abline(intercept = sml_capm[1], slope = sml_capm[2], color = "green") +
  scale_y_continuous(labels = percent, limit = c(0, mean(factors_ff_monthly$mkt_excess) * 2)) +
  scale_x_continuous(limits = c(0, 2)) +
  labs(
    x = "Beta", y = "Excess return", color = "Portfolio",
    title = "Average portfolio excess returns and average beta estimates"
  )
```



To provide more evidence against the CAPM predictions, we again form a long-short strategy that buys the high-beta portfolio and shorts the low-beta portfolio.

```
beta_longshort <- beta_portfolios %>%
  ungroup() %>%
  mutate(portfolio = case_when(
    portfolio == max(as.numeric(portfolio)) ~ "high",
    portfolio == min(as.numeric(portfolio)) ~ "low"
  )) %>%
  filter(portfolio %in% c("low", "high")) %>%
```

```
  pivot_wider(month, names_from = portfolio, values_from = ret) %>%
  mutate(long_short = high - low) %>%
  left_join(factors_ff_monthly, by = "month")
```

Again, the resulting long-short strategy does not exhibit statistically significant returns.

```
coeftest(lm(long_short ~ 1, data = beta_longshort), vcov = NeweyWest)
```

```
##
## t test of coefficients:
##
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 0.000734   0.002483     0.3     0.77
```

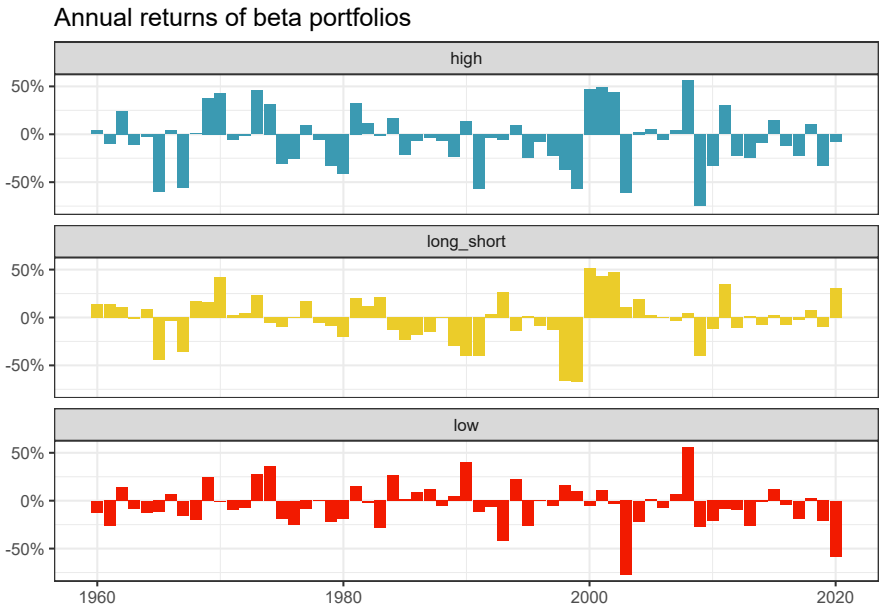
However, the long-short portfolio yields a statistically significant negative CAPM-adjusted alpha, although, controlling for the effect of beta, the average excess stock returns should be zero according to the CAPM. The results thus provide no evidence in support of the CAPM. The negative value has been documented as the so-called betting against beta factor (Frazzini and Pedersen (2014)). Betting against beta corresponds to a strategy that shorts high beta stocks and takes a (levered) long position in low beta stocks. If borrowing constraints prevent investors from taking positions on the SML they are instead incentivized to buy high beta stocks, which leads to a relatively higher price (and therefore lower expected returns than implied by the CAPM) for such high beta stocks. As a result, the betting-against-beta strategy earns from providing liquidity to capital constraint investors with lower risk aversion.

```
coeftest(lm(long_short ~ 1 + mkt_excess, data = beta_longshort), vcov = NeweyWest)
```

```
##
## t test of coefficients:
##
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.00441    0.00262   -1.69    0.092 .
## mkt_excess   0.89461    0.10214    8.76   <2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The plot below shows the annual returns of the extreme beta portfolios we are mainly interested in. The figure illustrates no consistent striking patterns over the last years - each portfolio exhibits periods with positive and negative annual returns.

```
beta_longshort %>%
  group_by(year = year(month)) %>%
  summarize(
    low = prod(1 + low),
    high = prod(1 + high),
    long_short = prod(1 + long_short)
  ) %>%
  pivot_longer(cols = -year) %>%
  ggplot(aes(x = year, y = 1 - value, fill = name)) +
  geom_col(position = "dodge") +
  facet_wrap(~name, ncol = 1) +
  theme(legend.position = "none") +
  scale_y_continuous(labels = percent) +
  labs(
    title = "Annual returns of beta portfolios",
    x = NULL, y = NULL
  )
```



Overall, this chapter shows how functional programming can be leveraged to form an arbitrary number of portfolios using any sorting variable and how to evaluate the performance of the resulting portfolios. In the next chapter, we dive deeper into the many degrees of freedom that arise in the context of portfolio analysis.

4.7 Exercises

1. Take the two long-short beta strategies based on different numbers of portfolios and compare the returns. Is there a significant difference in returns? How do the Sharpe ratios compare between the strategies? Find one additional portfolio evaluation statistic and compute it.
2. We plotted the alphas of the ten beta portfolios above. Write a function that tests these estimates for significance. Which portfolios have significant alphas?
3. The analysis here is based on betas from daily returns. However, we also computed betas from monthly returns. Re-run the analysis and point out differences in the results.
4. Given the results in this chapter, can you define a long-short strategy that yields positive abnormal returns (i.e., alphas)? Plot the cumulative excess return of your strategy and the market excess return for comparison.

5

Size sorts and p-hacking

In this chapter, we continue with portfolio sorts in a univariate setting. Yet, we consider firm size as a sorting variable, which gives rise to a well-known return factor: the size premium. The size premium arises from buying small stocks and selling large stocks. Prominently, Fama and French (1993) include it as a factor in their three-factor model. Apart from that, asset managers commonly include size as a key firm characteristic when making investment decisions.

We also introduce new choices in the formation of portfolios. In particular, we discuss listing exchanges, industries, weighting regimes, and periods. These choices matter for the portfolio returns and result in different size premiums. Exploiting these ideas to generate favorable results is called p-hacking. There is arguably a thin line between p-hacking and conducting robustness tests, our purpose here is simply to illustrate the substantial variation which can arise along the evidence generating process.

5.1 Data preparation

The chapter relies on the following set of packages.

```
library(tidyverse)
library(RSQLite)
library(lubridate)
library(sandwich)
library(lmtest)
library(scales)
library(furrr)
```

First, we retrieve the relevant data from our `SQLite`-database introduced in our chapter on *"Accessing & managing financial data"*. Firm size is defined as market equity in most asset pricing applications that we retrieve from CRSP. We further use the Fama-French factor returns for performance evaluation.

```
tidy_finance <- dbConnect(SQLite(), "data/tidy_finance.sqlite", extended_types = TRUE)

crsp_monthly <- tbl(tidy_finance, "crsp_monthly") %>%
  collect()

factors_ff_monthly <- tbl(tidy_finance, "factors_ff_monthly") %>%
  collect()
```

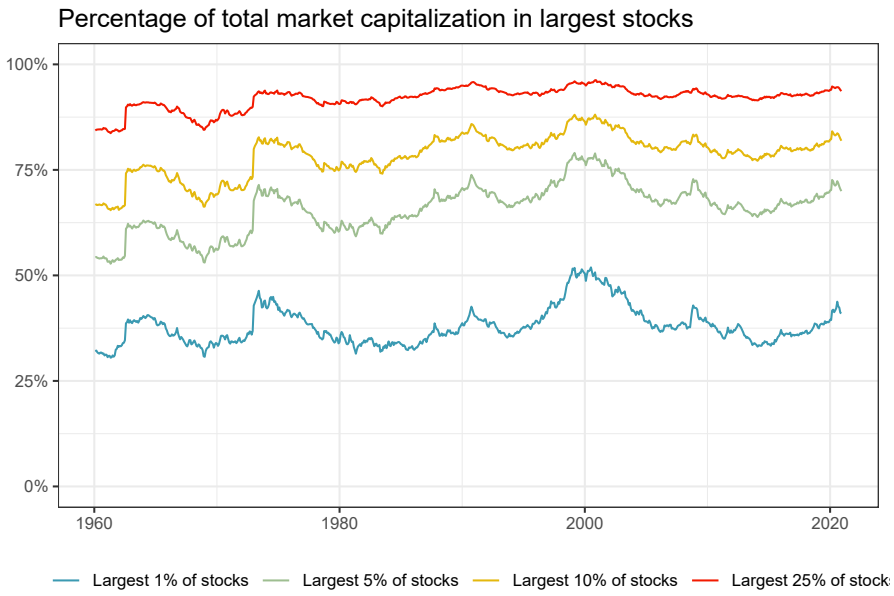
5.2 Size distribution

Before we build our size portfolios, we investigate the distribution of the variable *firm size*. Visualizing the data is a valuable starting point to understand the input to the analysis. The figure below shows the fraction of total market capitalization concentrated in the largest firm. To produce this graph, we create monthly indicators that track whether a stock belongs to the largest x% of the firms. Then, we aggregate the firms within each bucket and compute the buckets' share of total market capitalization.

The figure shows that the largest 1% of firms cover up to 50% of the total market capitalization, and holding just the 25% largest firms in the CRSP universe essentially replicates the market portfolio. The distribution of firm size thus implies that the largest firms of the market dominate many small firms whenever we use value-weighted benchmarks.

```
crsp_monthly %>%
  group_by(month) %>%
  mutate(
    top01 = if_else(mktcap >= quantile(mktcap, 0.99), 1L, 0L),
    top05 = if_else(mktcap >= quantile(mktcap, 0.95), 1L, 0L),
    top10 = if_else(mktcap >= quantile(mktcap, 0.90), 1L, 0L),
    top25 = if_else(mktcap >= quantile(mktcap, 0.75), 1L, 0L),
    total_market_cap = sum(mktcap)
  ) %>%
  summarize(
    `Largest 1% of stocks` = sum(mktcap[top01 == 1]) / total_market_cap,
    `Largest 5% of stocks` = sum(mktcap[top05 == 1]) / total_market_cap,
    `Largest 10% of stocks` = sum(mktcap[top10 == 1]) / total_market_cap,
    `Largest 25% of stocks` = sum(mktcap[top25 == 1]) / total_market_cap
  ) %>%
  pivot_longer(cols = -month) %>%
  mutate(name = factor(name, levels = c(
```

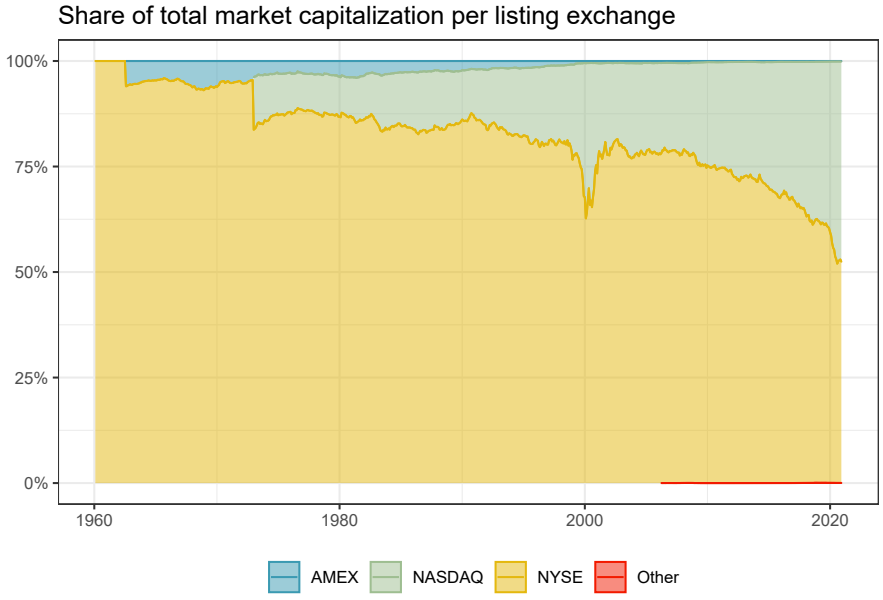
```
    "Largest 1% of stocks", "Largest 5% of stocks",
    "Largest 10% of stocks", "Largest 25% of stocks"
))) %>%
ggplot(aes(x = month, y = value, color = name)) +
geom_line() +
scale_y_continuous(labels = percent, limits = c(0, 1)) +
labs(
  x = NULL, y = NULL, color = NULL,
  title = "Percentage of total market capitalization in largest stocks"
)
```



Next, firm sizes also differ across listing exchanges. Stocks' primary listings were important in the past and are potentially still relevant today. The graph below shows that the New York Stock Exchange (NYSE) was and still is the largest listing exchange in terms of market capitalization. More recently, NASDAQ has gained relevance as a listing exchange. Do you know what the small peak in NASDAQ's market cap around the year 2000 was?

```
crsp_monthly %>%
  group_by(month, exchange) %>%
  summarize(mktcap = sum(mktcap)) %>%
  mutate(share = mktcap / sum(mktcap)) %>%
  ggplot(aes(x = month, y = share, fill = exchange, color = exchange)) +
  geom_area(
```

```
    position = "stack",
    stat = "identity",
    alpha = 0.5
  ) +
  geom_line(position = "stack") +
  scale_y_continuous(labels = percent) +
  labs(
    x = NULL, y = NULL, fill = NULL, color = NULL,
    title = "Share of total market capitalization per listing exchange"
  )
```



Finally, we consider the distribution of firm size across listing exchanges and create summary statistics. The pre-build function `summary()` does not include all statistics we are interested in, which is why we create the function `create_summary()` that adds the standard deviation and the number of observations. Then, we apply it to the most current month of our CRSP data on each listing exchange. We also add a row with `add_row()` with the overall summary statistics.

The resulting table shows that firms listed on NYSE are significantly larger on average than firms listed on the other exchanges. Moreover, NASDAQ lists the largest number of firms. This discrepancy between firm sizes across listing exchanges motivated researchers to form breakpoints exclusively on the NYSE sample and apply those breakpoints to all stocks. In the following, we use this distinction to update our portfolio sort procedure.

```
create_summary <- function(data, column_name) {
  data %>%
    select(value = {{ column_name }}) %>%
    summarize(
      mean = mean(value),
      sd = sd(value),
      min = min(value),
      q05 = quantile(value, 0.05),
      q25 = quantile(value, 0.25),
      q50 = quantile(value, 0.50),
      q75 = quantile(value, 0.75),
      q95 = quantile(value, 0.95),
      max = max(value),
      n = n()
    )
}

crsp_monthly %>%
  filter(month == max(month)) %>%
  group_by(exchange) %>%
  create_summary(mktcap) %>%
  add_row(crsp_monthly %>%
    filter(month == max(month)) %>%
    create_summary(mktcap) %>%
    mutate(exchange = "Overall"))
```

```
## # A tibble: 5 x 11
##   exchange   mean     sd      min     q05    q25    q50
##   <chr>     <dbl>  <dbl>    <dbl>   <dbl>  <dbl>  <dbl>
## 1 AMEX       283.  1298.     6.04    10.1 3.07e1 6.59e1
## 2 NASDAQ    8041. 74386.     4.65    27.3 1.34e2 4.85e2
## 3 NYSE     16427. 43130.     5.35   154.  9.16e2 3.34e3
## 4 Other    10061.    NA  10061.   10061.  1.01e4 1.01e4
## 5 Overall  10558. 63975.     4.65    31.0 1.85e2 8.72e2
## # ... with 4 more variables: q75 <dbl>, q95 <dbl>,
## #   max <dbl>, n <int>
```

5.3 Univariate size portfolios with flexible breakpoints

In the previous chapter, we construct portfolios with a varying number of portfolios and different sorting variables. Here, we extend the framework such that we

compute breakpoints on a subset of the data, for instance, based on selected listing exchanges. In published asset pricing articles, many scholars compute sorting breakpoints only on NYSE-listed stocks. These NYSE-specific breakpoints are then applied to the entire universe of stocks.

To replicate the NYSE-centered sorting procedure, we introduce `exchanges` as an argument in our `assign_portfolio()` function. The exchange-specific argument then enters in the filter `filter(grepl(exchanges, exchange))`. The function `grepl()` is part of a family of functions on *regular expressions*, which provide various functionalities to work and manipulate character strings. Here, we replace the character string stored in the column `exchange` with a binary variable that indicates if the string matches the pattern specified in the argument `exchanges`. For example, if `exchanges = 'NYSE'` is specified, only stocks from NYSE are used to compute the breakpoints. Alternatively, you could specify `exchanges = 'NYSE|NASDAQ|AMEX'`, which keeps all stocks listed on either of these exchanges. Overall, regular expressions are a powerful tool, and we only touch on a specific case here.

```
assign_portfolio <- function(n_portfolios,
                             exchanges,
                             data) {
  breakpoints <- data %>%
    filter(grepl(exchanges, exchange)) %>%
    summarize(breakpoint = quantile(
      mktcap_lag,
      probs = seq(0, 1, length.out = n_portfolios + 1),
      na.rm = TRUE
    )) %>%
    pull(breakpoint) %>%
    as.numeric()

  data %>%
    mutate(portfolio = findInterval(mktcap_lag, breakpoints, all.inside = TRUE)) %>%
    pull(portfolio)
}
```

5.4 Weighting schemes for portfolios

Apart from computing breakpoints on different samples, researchers often use different portfolio weighting schemes. So far, we weighted each portfolio constituent by its relative market equity of the previous period. This protocol is called *value-weighting*. The alternative protocol is *equal-weighting*, which assigns each stock's return the same weight, i.e., a simple average of the constituents' returns. Notice that

equal-weighting is difficult in practice as the portfolio manager needs to rebalance the portfolio monthly while value-weighting is a truly passive investment.

We implement the two weighting schemes in the function `compute_portfolio_returns()` that takes a logical argument to weight the returns by firm value. The statement `if_else(value_weighted, weighted.mean(ret_excess, mktcap_lag), mean(ret_excess))` generates value-weighted returns if `value_weighted = TRUE`. Additionally, the long-short portfolio is long in the smallest firms and short in the largest firms, consistent with research showing that small firms outperform their larger counterparts. Apart from these two changes, the function is similar to the procedure in the previous chapter.

```
compute_portfolio_returns <- function(n_portfolios = 10,
                                      exchanges = "NYSE|NASDAQ|AMEX",
                                      value_weighted = TRUE,
                                      data = crsp_monthly) {
  data %>%
    group_by(month) %>%
    mutate(portfolio = assign_portfolio(
      n_portfolios = n_portfolios,
      exchanges = exchanges,
      data = cur_data()
    )) %>%
    group_by(month, portfolio) %>%
    summarize(
      ret = if_else(value_weighted, weighted.mean(ret_excess, mktcap_lag), mean(ret_excess)),
      .groups = "drop_last"
    ) %>%
    summarize(size_premium = ret[portfolio == min(portfolio)] - ret[portfolio == max(portfolio)]) %>%
    summarize(size_premium = mean(size_premium))
}
```

To see how the function `compute_portfolio_returns()` works, we consider a simple median breakpoint example with value-weighted returns. We are interested in the effect of restricting listing exchanges on the estimation of the size premium. In the first function call, we compute returns based on breakpoints from all listing exchanges. Then, we computed returns based on breakpoints from NYSE-listed stocks.

```
ret_all <- compute_portfolio_returns(
  n_portfolios = 2,
  exchanges = "NYSE|NASDAQ|AMEX",
  value_weighted = TRUE,
  data = crsp_monthly
)
```

```
ret_nyse <- compute_portfolio_returns(
  n_portfolios = 2,
  exchanges = "NYSE",
  value_weighted = TRUE,
  data = crsp_monthly
)

tibble(Exchanges = c("all", "NYSE"), Premium = as.numeric(c(ret_all, ret_nyse)) * 100)
```

```
## # A tibble: 2 x 2
##   Exchanges Premium
##   <chr>       <dbl>
## 1 all         0.110
## 2 NYSE        0.181
```

The table shows that the size premium is more than 60% larger if we consider only stocks from NYSE to form the breakpoint each month. The NYSE-specific breakpoints are larger, and there are more than 50% of the stocks in the entire universe in the resulting small portfolio because NYSE firms are larger on average. The impact of this choice is not negligible.

5.5 P-hacking and non-standard errors

Since the choice of the exchange has a significant impact, the next step is to investigate the effect of other data processing decisions researchers have to make along the way. In particular, any portfolio sort analysis has to decide at least on the number of portfolios, the listing exchanges to form breakpoints, and equal- or value-weighting. Further, one may exclude firms that are active in the finance industry or restrict the analysis to some parts of the time series. All of the variations of these choices that we discuss here are part of scholarly articles published in the top finance journals. The intention of this application is to show that the different ways to form portfolios result in different estimated size premia. Despite the effects of this multitude of choices, there is no correct way. It should also be noted that none of the procedures is wrong, the aim is simply to illustrate the changes that can arise due to the variation in the evidence-generating process (Menkveld et al., 2021). From a malicious perspective, these modeling choices give the researcher multiple *chances* to find statistically significant results. Yet this is considered *p-hacking* which renders the statistical inference due to multiple testing invalid (Harvey et al., 2016).

Nevertheless, the multitude of options creates a problem since there is no single correct way of sorting portfolios. How should a researcher convince a reader that their

results do not come from a p-hacking exercise? To circumvent this dilemma, academics are encouraged to present evidence from different sorting schemes as *robustness tests* and report multiple approaches to show that a result does not depend on a single choice. Thus, the robustness of premiums is a key feature.

Below we conduct a series of robustness tests which could also be interpreted as a p-hacking exercise. To do so, we examine the size premium in different specifications presented in the table `p_hacking_setup`. The function `expand_grid()` produces a table of all possible permutations of its arguments. Notice that we use the argument `data` to exclude financial firms and truncate the time series.

```
p_hacking_setup <- expand_grid(
  n_portfolios = c(2, 5, 10),
  exchanges = c("NYSE", "NYSE|NASDAQ|AMEX"),
  value_weighted = c(TRUE, FALSE),
  data = rlang::parse_exprs('crsp_monthly; crsp_monthly %>% filter(industry != "Finance");
                             crsp_monthly %>% filter(month < "1990-06-01");
                             crsp_monthly %>% filter(month >="1990-06-01")')
)
p_hacking_setup
```

```
## # A tibble: 48 x 4
##   n_portfolios exchanges value_weighted data
##          <dbl> <chr>     <lgl>          <list>
## 1            2 NYSE      TRUE           <sym>
## 2            2 NYSE      TRUE           <language>
## 3            2 NYSE      TRUE           <language>
## 4            2 NYSE      TRUE           <language>
## 5            2 NYSE      FALSE          <sym>
## # ... with 43 more rows
```

To speed the computation up we parallelize the (many) different sorting procedures, as in Chapter 3. Finally, we report the resulting size premiums in descending order. There are indeed substantial size premia possible in our data, in particular when we use equal-weighted portfolios.

```
plan(multisession, workers = availableCores())

p_hacking_setup <- p_hacking_setup %>%
  mutate(size_premium = future_pmap(
    .l = list(
      n_portfolios,
      exchanges,
      value_weighted,
      data
```

```
    ),
    .f = ~ compute_portfolio_returns(
      n_portfolios = ..1,
      exchanges = ..2,
      value_weighted = ..3,
      data = rlang::eval_tidy(..4)
    )
  ))

p_hacking_results <- p_hacking_setup %>%
  mutate(data = map_chr(data, deparse)) %>%
  unnest(size_premium) %>%
  mutate(data = str_remove(data, "crsp_monthly %>% ")) %>%
  arrange(desc(size_premium))
p_hacking_results
```

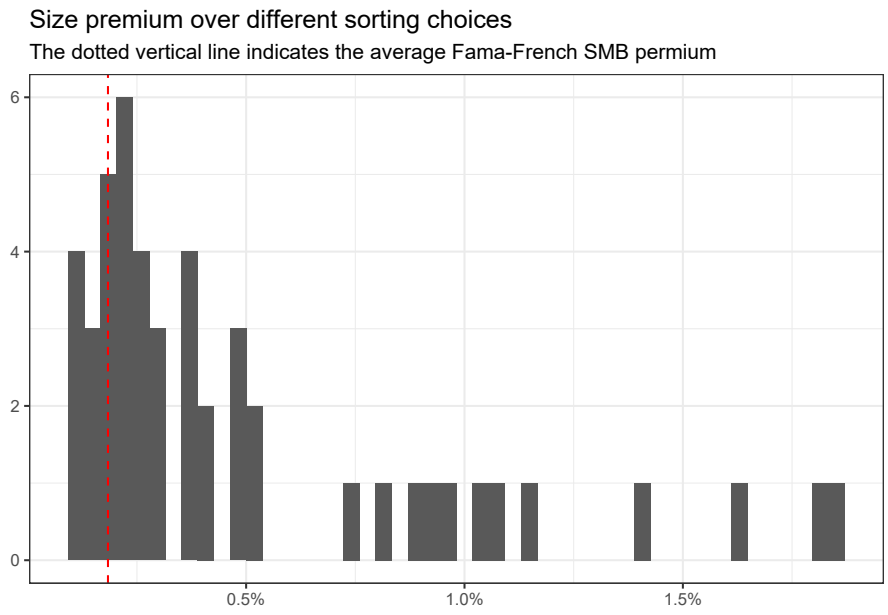
```
## # A tibble: 48 x 5
##   n_portfolios exchanges        value_weighted data
##          <dbl> <chr>            <lgl>          <chr>
## 1           10 NYSE|NASDAQ|AMEX FALSE          "filter~
## 2           10 NYSE|NASDAQ|AMEX FALSE          "filter~
## 3           10 NYSE|NASDAQ|AMEX FALSE          "crsp_m~
## 4           10 NYSE|NASDAQ|AMEX FALSE          "filter~
## 5           10 NYSE|NASDAQ|AMEX TRUE           "filter~
## # ... with 43 more rows, and 1 more variable:
## #   size_premium <dbl>
```

5.6 The size-premium variation

We provide a graph that shows the different premiums. This plot also shows the relation to the average Fama-French SMB (small minus big) premium used in the literature which we include as a dotted vertical line.

```
p_hacking_results %>%
  ggplot(aes(x = size_premium)) +
  geom_histogram(bins = nrow(p_hacking_results)) +
  labs(
    x = NULL, y = NULL,
    title = "Size premium over different sorting choices",
    subtitle = "The dotted vertical line indicates the average Fama-French SMB permium"
```

```
) +
geom_vline(aes(xintercept = mean(factors_ff_monthly$smb)),
  color = "red",
  linetype = "dashed"
) +
scale_x_continuous(labels = percent)
```



5.7 Exercises

1. We gained several insights on the size distribution above. However, we did not analyse the average size across exchanges and industries. Which exchanges/industries have the largest firms? Plot the average firm size for the three exchanges over time. What do you see?
2. We compute breakpoints but do not take a look at them in the exposition above. This might cover potential data errors. Plot the breakpoints for ten size portfolios over time. Then, take the difference between the two extreme portfolios and plot it. Describe your results.
3. The returns that we analyse above do not account for differences in the exposure to market risk, i.e., the CAPM beta. Change the function `com-`

`pute_portfolio_returns()` to output the CAPM alpha or beta instead of the average excess return.

4. While you saw the spread in returns from the p-hacking exercise, we did not show which choices led to the largest effects. Find a way to investigate which choice variable has the largest impact on the estimated size premium.
5. We computed several size premiums, but they do not follow the definition of Fama and French (1993). Which of our approaches comes closest to their SMB premium?

6

Value and bivariate sorts

This chapter extends univariate portfolio analysis to bivariate sorts which means that we assign stocks to portfolios based on two characteristics. Bivariate sorts are regularly used in the academic asset pricing literature. Yet, some scholars also use sorts with three grouping variables. Conceptually, portfolio sorts are easily applicable in higher dimensions.

We form portfolios on firm size and the book-to-market ratio. To calculate book-to-market ratios, accounting data is required which necessitates additional steps during portfolio formation. In the end, we demonstrate how to form portfolios on two sorting variables using so-called independent and dependent portfolio sorts.

The current chapter relies on this set of packages.

```
library(tidyverse)
library(RSQLite)
library(lubridate)
library(sandwich)
library(lmtest)
library(scales)
```

6.1 Data preparation

First, we load the necessary data from our `SQLite`-database introduced in our chapter on *"Accessing & managing financial data"*. We conduct portfolio sorts based on the CRSP sample but keep only the necessary columns in our memory. We use the same data sources for firm size as in the previous chapter.

```
tidy_finance <- dbConnect(SQLite(), "data/tidy_finance.sqlite",
                          extended_types = TRUE)

crsp_monthly <- tbl(tidy_finance, "crsp_monthly") %>%
  collect()
```

```
factors_ff_monthly <- tbl(tidy_finance, "factors_ff_monthly") %>%
  collect()

crsp_monthly <- crsp_monthly %>%
  left_join(factors_ff_monthly, by = "month") %>%
  select(permno, gvkey, month, ret_excess, mkt_excess,
         mktcap, mktcap_lag, exchange) %>%
  drop_na()
```

Further, we utilize accounting data. The most common source of accounting data is *Compustat*. We only need book equity data in this application, which we select from our database. Additionally, we convert the variable `datadate` to its monthly value, as we only consider monthly returns here and do not need to account for the exact date. To achieve this, we use the function `floor_date()`.

```
compustat <- tbl(tidy_finance, "compustat") %>%
  collect()

be <- compustat %>%
  select(gvkey, datadate, be) %>%
  drop_na() %>%
  mutate(month = floor_date(ymd(datadate), "month"))
```

6.2 Book-to-market ratio

A fundamental problem in handling accounting data is the *look-ahead bias* - we must not include data in forming a portfolio that is not public knowledge at the time. Of course, researchers have more information when looking into the past than agents had at that moment. However, abnormal excess returns from a trading strategy should not rely on an information advantage because the differential cannot be the result of informed agents' trades. Hence, we have to lag accounting information.

We continue to lag market capitalization and firm size by one month. Then, we compute the book-to-market ratio, which relates a firm's book equity to its market equity. Firms with high (low) book-to-market are called value (growth) firms. After matching the accounting and market equity information from the same month, we lag book-to-market by six months. This is a sufficiently conservative approach because accounting information is usually released well before six months pass. However, in the asset pricing literature, even longer lags are used as well.[1]

[1] The definition of a time lag is another choice a researcher has to make, similar to breakpoint choices as we describe in the section on p-hacking.

Having both variables, i.e., firm size lagged by one month and book-to-market lagged by six months, we merge these sorting variables to our returns using the `sorting_date`-column created for this purpose. The final step in our data preparation deals with differences in the frequency of our variables. Returns and firm size are recorded monthly. Yet the accounting information is only released on an annual basis. Hence, we only match book-to-market to one month per year and have eleven empty observations. To solve this frequency issue, we carry the latest book-to-market ratio of each firm to the subsequent months, i.e., we fill the missing observations with the most current report. This is done via the `fill()`-function after sorting by date and firm (which we identify by permno and gvkey) and on a firm basis (which we do by `group_by()` as usual). As the last step, we remove all rows with missing entries because the returns cannot be matched to any annual report.

```
me <- crsp_monthly %>%
  mutate(sorting_date = month %m+% months(1)) %>%
  select(permno, sorting_date, me = mktcap)

bm <- be %>%
  inner_join(crsp_monthly %>%
    select(month, permno, gvkey, mktcap), by = c("gvkey", "month")) %>%
  mutate(
    bm = be / mktcap,
    sorting_date = month %m+% months(6)
  ) %>%
  select(permno, gvkey, sorting_date, bm) %>%
  arrange(permno, gvkey, sorting_date)

data_for_sorts <- crsp_monthly %>%
  left_join(bm, by = c("permno", "gvkey", "month" = "sorting_date")) %>%
  left_join(me, by = c("permno", "month" = "sorting_date")) %>%
  select(permno, gvkey, month, ret_excess, mktcap_lag, me, bm, exchange)

data_for_sorts <- data_for_sorts %>%
  arrange(permno, gvkey, month) %>%
  group_by(permno, gvkey) %>%
  fill(bm) %>%
  drop_na()
```

The last step of preparation for the portfolio sorts is the computation of breakpoints. We continue to use the same function allowing for the specification of exchanges to use for the breakpoints. Additionally, we reintroduce the argument `var` into the function for defining different sorting variables via `curly-curly`.

```
assign_portfolio <- function(data, var, n_portfolios, exchanges) {
  breakpoints <- data %>%
    filter(exchange %in% exchanges) %>%
    summarize(breakpoint = quantile(
      {{ var }},
      probs = seq(0, 1, length.out = n_portfolios + 1),
      na.rm = TRUE
    )) %>%
    pull(breakpoint) %>%
    as.numeric()

  data %>%
    mutate(portfolio = findInterval({{ var }}, breakpoints, all.inside = TRUE)) %>%
    pull(portfolio)
}
```

After these data preparation steps, we present bivariate portfolio sorts on an independent and dependent basis.

6.3 Independent sorts

Bivariate sorts create portfolios within a two-dimensional space spanned by two sorting variables. It is then possible to assess the return impact of either sorting variable by the return differential from a trading strategy that invests in the portfolios at either end of the respective variables spectrum. We create a five-by-five matrix using book-to-market and firm size as sorting variables in our example below. We end up with 25 portfolios. Since we are interested in the *value premium* (i.e., the return differential between high and low book-to-market firms), we go long the five portfolios of the highest book-to-market firms and short the five portfolios of the lowest book-to-market firms. The five portfolios at each end are due to the size splits we employed alongside the book-to-market splits.

To implement the independent bivariate portfolio sort, we assign monthly portfolios for each of our sorting variables separately to create the variables `portfolio_bm` and `portfolio_bm`, respectively. Then, these separate portfolios are combined to the final sort stored in `portfolio_combined`. After assigning the portfolios, we compute the average return within each portfolio for each month. Additionally, we keep the book-to-market portfolio as it makes the computation of the value premium easier. The alternative would be to disaggregate the combined portfolio in a separate step. Notice that we weigh the stocks within each portfolio by their market capitalization, i.e., we decide to value-weight our returns.

```
value_portfolios <- data_for_sorts %>%
  group_by(month) %>%
  mutate(
    portfolio_bm = assign_portfolio(
      data = cur_data(),
      var = bm,
      n_portfolios = 5,
      exchanges = c("NYSE")
    ),
    portfolio_me = assign_portfolio(
      data = cur_data(),
      var = me,
      n_portfolios = 5,
      exchanges = c("NYSE")
    ),
    portfolio_combined = paste0(portfolio_bm, portfolio_me)
  ) %>%
  group_by(month, portfolio_combined) %>%
  summarize(
    ret = weighted.mean(ret_excess, mktcap_lag),
    portfolio_bm = unique(portfolio_bm),
    .groups = "drop"
  )
```

Equipped with our monthly portfolio returns, we are ready to compute the value premium. However, we still have to decide how to invest in the five high and the five low book-to-market portfolios. The most common approach is to weigh these portfolios equally, but this is yet another researcher's choice. Then, we compute the return differential between the high and low book-to-market portfolios and show the average value premium.

```
value_premium <- value_portfolios %>%
  group_by(month, portfolio_bm) %>%
  summarize(ret = mean(ret), .groups = "drop_last") %>%
  summarize(value_premium = ret[portfolio_bm == max(portfolio_bm)] -
              ret[portfolio_bm == min(portfolio_bm)])

mean(value_premium$value_premium * 100)
```

```
## [1] 0.328
```

The resulting annualized value premium is 3.936 percent.

6.4 Dependent sorts

In the previous exercise, we assigned the portfolios without considering the second variable in the assignment. This protocol is called independent portfolio sorts. The alternative, i.e., dependent sorts, creates portfolios for the second sorting variable within each bucket of the first sorting variable. In our example below, we sort firms into five size buckets, and within each of those buckets, we assign firms to five book-to-market portfolios. Hence, we have monthly breakpoints that are specific to each size group. The decision between independent and dependent portfolio sorts is another choice for the researcher. Notice that dependent sorts ensure an equal amount of stocks within each portfolio.

To implement the dependent sorts, we first create the size portfolios by calling `assign_portfolio()` with `var = me`. Then, we group our data again by month and by the size portfolio before assigning the book-to-market portfolio. The rest of the implementation is the same as before. Finally, we compute the value premium.

```
value_portfolios <- data_for_sorts %>%
  group_by(month) %>%
  mutate(portfolio_me = assign_portfolio(
    data = cur_data(),
    var = me,
    n_portfolios = 5,
    exchanges = c("NYSE")
  )) %>%
  group_by(month, portfolio_me) %>%
  mutate(
    portfolio_bm = assign_portfolio(
      data = cur_data(),
      var = bm,
      n_portfolios = 5,
      exchanges = c("NYSE")
    ),
    portfolio_combined = paste0(portfolio_bm, portfolio_me)
  ) %>%
  group_by(month, portfolio_combined) %>%
  summarize(
    ret = weighted.mean(ret_excess, mktcap_lag),
    portfolio_bm = unique(portfolio_bm),
    .groups = "drop"
  )

value_premium <- value_portfolios %>%
```

```
  group_by(month, portfolio_bm) %>%
  summarize(ret = mean(ret), .groups = "drop_last") %>%
  summarize(value_premium = ret[portfolio_bm == max(portfolio_bm)] -
              ret[portfolio_bm == min(portfolio_bm)])

mean(value_premium$value_premium * 100)
```

```
## [1] 0.265
```

The value premium from dependent sorts is 3.18 percent per year.

Overall, we show how to conduct bivariate portfolio sorts in this chapter. In one case, we sort the portfolios independently of each other. Yet we also discuss how to create dependent portfolio sorts. Along the line of the previous chapter, we see how many choices a researcher has to make to implement portfolio sorts, and bivariate sorts increase the number of choices.

6.5 Exercises

1. In the previous chapter, we examined the distribution of market equity. Repeat this analysis for book equity and the book-to-market ratio (alongside a plot of the breakpoints, i.e., deciles).
2. When we investigate the portfolios, we focus on the returns exclusively. However, it is also of interest to understand the characteristics of the portfolios. Write a function to compute the average characteristics for size and book-to-market across the 25 independently and dependently sorted portfolios.
3. As for the size premium, also the value premium constructed here does not follow Fama and French (1993). Implement a p-hacking setup as in the previous chapter to find a premium that comes closest to their HML premium.

7

Replicating Fama & French factors

The Fama and French three-factor model (see Fama and French (1993)) is a cornerstone of asset pricing. On top of the market factor represented by the traditional CAPM beta, the model includes the size and value factors. We introduce both factors in the previous chapter, and their definition remains the same. Size is the SMB factor (small-minus-big) that is long small firms and short large firms. The value factor is HML (high-minus-low) and is long in high book-to-market firms and short the low book-to-market counterparts. In this chapter, we also want to show the main idea of how to replicate these significant factors.

The current chapter relies on this set of packages.

```
library(tidyverse)
library(RSQLite)
library(lubridate)
```

7.1 Databases

We use CRSP and Compustat as data sources, as we need exactly the same variables to compute the size and value factors in the way Fama and French do it. Hence, there is nothing new below and we only load data from our `SQLite`-database introduced in our chapter on *"Accessing & managing financial data"*..

```
tidy_finance <- dbConnect(SQLite(), "data/tidy_finance.sqlite",
  extended_types = TRUE
)

crsp_monthly <- tbl(tidy_finance, "crsp_monthly") %>%
  collect()

factors_ff_monthly <- tbl(tidy_finance, "factors_ff_monthly") %>%
  collect()
```

```
compustat <- tbl(tidy_finance, "compustat") %>%
  collect()

data_ff <- crsp_monthly %>%
  left_join(factors_ff_monthly, by = "month") %>%
  select(
    permno, gvkey, month, ret_excess, mkt_excess,
    mktcap, mktcap_lag, exchange
  ) %>%
  drop_na()

be <- compustat %>%
  select(gvkey, datadate, be) %>%
  drop_na()
```

7.2 Data preparation

Yet when we start merging our data set for computing the premiums, there are a few differences to the previous chapter. First, Fama and French form their portfolios in June of year t, whereby the returns of July are the first monthly return for the respective portfolio. For firm size, they consequently use the market capitalization recorded for June. It is then held constant until June of year $t+1$.

Second, Fama and French also have a different protocol for computing the book-to-market ratio. They use market equity as of the end of year $t-1$ and the book equity reported in year $t-1$, i.e., the `datadate` is within the last year. Hence, the book-to-market ratio can be based on accounting information that is up to 18 months old. Market equity also does not necessarily reflect the same time point as book equity.

To implement all these time lags, we again employ the temporary `sorting_date`-column. Notice that when we combine the information, we want to have a single observation per year and stock since we are only interested in computing the breakpoints held constant for the entire year. We ensure this by a call of `distinct()` at the end of the chunk below.

```
me_ff <- data_ff %>%
  filter(month(month) == 6) %>%
  mutate(sorting_date = month %m+% months(1)) %>%
  select(permno, sorting_date, me_ff = mktcap)

me_ff_dec <- data_ff %>%
  filter(month(month) == 12) %>%
```

```
  mutate(sorting_date = ymd(paste0(year(month) + 1, "0701"))) %>%
  select(permno, gvkey, sorting_date, bm_me = mktcap)

bm_ff <- be %>%
  mutate(sorting_date = ymd(paste0(year(datadate) + 1, "0701"))) %>%
  select(gvkey, sorting_date, bm_be = be) %>%
  drop_na() %>%
  inner_join(me_ff_dec, by = c("gvkey", "sorting_date")) %>%
  mutate(bm_ff = bm_be / bm_me) %>%
  select(permno, sorting_date, bm_ff)

variables_ff <- me_ff %>%
  inner_join(bm_ff, by = c("permno", "sorting_date")) %>%
  drop_na() %>%
  distinct(permno, sorting_date, .keep_all = TRUE)
```

7.3 Portfolio sorts

Next, we construct our portfolios with an adjusted `assign_portfolio()` function. Fama and French rely on NYSE-specific breakpoints, they form two portfolios in the size dimension at the median and three portfolios in the dimension of book-to-market at the 30%- and 70%-percentiles, and they use independent sorts. The sorts for book-to-market require an adjustment to the previous function because the `seq()` we would produce does not produce the right breakpoints. Instead of `n_portfolios`, we now specify `percentiles`, which take the breakpoint-sequence as an object specified in the function's call. Specifically, we give `percentiles = c(0, 0.3, 0.7, 1)` to the function. Additionally, we perform an `inner_join()` with our return data to ensure that we only use traded stocks when computing the breakpoints as a first step.

```
assign_portfolio <- function(data, var, percentiles) {
  breakpoints <- data %>%
    filter(exchange == "NYSE") %>%
    summarize(breakpoint = quantile(
      {{ var }},
      probs = {{ percentiles }},
      na.rm = TRUE
    )) %>%
    pull(breakpoint) %>%
    as.numeric()

  data %>%
```

```
    mutate(portfolio = findInterval({{ var }}, breakpoints, all.inside = TRUE)) %>%
    pull(portfolio)
}

portfolios_ff <- variables_ff %>%
  inner_join(data_ff, by = c("permno" = "permno", "sorting_date" = "month")) %>%
  group_by(sorting_date) %>%
  mutate(
    portfolio_me = assign_portfolio(
      data = cur_data(),
      var = me_ff,
      percentiles = c(0, 0.5, 1)
    ),
    portfolio_bm = assign_portfolio(
      data = cur_data(),
      var = bm_ff,
      percentiles = c(0, 0.3, 0.7, 1)
    )
  ) %>%
  select(permno, sorting_date, portfolio_me, portfolio_bm)
```

Next, we merge the portfolios to the return data for the rest of the year. To implement this step, we create a new column `sorting_date` in our return data by setting the date to sort on to July of $t - 1$ if the month is June (of year t) or earlier or to July of year t if the month is July or later.

```
portfolios_ff <- data_ff %>%
  mutate(sorting_date = case_when(
    month(month) <= 6 ~ ymd(paste0(year(month) - 1, "0701")),
    month(month) >= 7 ~ ymd(paste0(year(month), "0701"))
  )) %>%
  inner_join(portfolios_ff, by = c("permno", "sorting_date"))
```

7.4 Fama and French factor returns

Equipped with the return data and the assigned portfolios, we can now compute the value-weighted average return for each of the six portfolios. Then, we form the Fama and French factors. For the size factor (i.e., SMB), we go long in the three small portfolios and short the three large portfolios by taking an average across either group. For the value factor (i.e., HML), we go long in the two high book-to-market portfolios and short the two low book-to-market portfolios, again weighting them equally.

```
factors_ff_monthly_replicated <- portfolios_ff %>%
  mutate(portfolio = paste0(portfolio_me, portfolio_bm)) %>%
  group_by(portfolio, month) %>%
  summarize(
    ret = weighted.mean(ret_excess, mktcap_lag), .groups = "drop",
    portfolio_me = unique(portfolio_me),
    portfolio_bm = unique(portfolio_bm)
  ) %>%
  group_by(month) %>%
  summarize(
    smb_replicated = mean(ret[portfolio_me == 1]) - mean(ret[portfolio_me == 2]),
    hml_replicated = mean(ret[portfolio_bm == 3]) - mean(ret[portfolio_bm == 1])
  )
```

7.5 Replication evaluation

In the previous section, we replicated the size and value premiums following the procedure outlined by Fama and French. However, we did not follow their procedure strictly. The final question is then: how close did we get? We answer this question by looking at the two time-series estimates in a regression analysis using `lm()`. If we did a good job, then we should see a non-significant intercept (rejecting the notion of systematic error), a coefficient close to 1 (indicating a high correlation), and an adjusted R-squared close to 1 (indicating a high proportion of explained variance).

```
test <- factors_ff_monthly %>%
  inner_join(factors_ff_monthly_replicated, by = "month") %>%
  mutate(
    smb_replicated = round(smb_replicated, 4),
    hml_replicated = round(hml_replicated, 4)
  )
```

The results for the SMB factor are quite convincing as all three criteria outlined above are met and the coefficient and R-squared are at 99%.

```
summary(lm(smb ~ smb_replicated, data = test))
```

```
##
## Call:
## lm(formula = smb ~ smb_replicated, data = test)
##
```

```
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.020320 -0.001501  0.000027  0.001519  0.014615
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)    -0.000143   0.000133   -1.07     0.28
## smb_replicated  0.996413   0.004418  225.55   <2e-16
##
## (Intercept)
## smb_replicated ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.00355 on 712 degrees of freedom
## Multiple R-squared:  0.986,  Adjusted R-squared:  0.986
## F-statistic: 5.09e+04 on 1 and 712 DF,  p-value: <2e-16
```

The replication of the HML factor is also a success, although at a slightly lower level with coefficient and R-squared around 95%.

```
summary(lm(hml ~ hml_replicated, data = test))
```

```
##
## Call:
## lm(formula = hml ~ hml_replicated, data = test)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.022250 -0.002933 -0.000101  0.002366  0.027475
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)    0.000294   0.000214    1.38     0.17
## hml_replicated 0.958849   0.007376  130.00   <2e-16
##
## (Intercept)
## hml_replicated ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0057 on 712 degrees of freedom
## Multiple R-squared:  0.96,   Adjusted R-squared:  0.96
```

```
## F-statistic: 1.69e+04 on 1 and 712 DF,  p-value: <2e-16
```

The evidence hence allows us to conclude that we did a relatively good job in replicating the original Fama-French premiums, although we cannot see their underlying code. From our perspective, a perfect match is only possible with additional information from the maintainers of the original data.

8

Fama-MacBeth Regressions

The regression approach of Fama and MacBeth (1973) is widely used in empirical asset pricing studies. Researchers use the two-stage regression approach to estimate risk premiums in various markets, but predominately in the stock market. Essentially, the two-step Fama-MacBeth regressions exploit a linear relationship between expected returns and exposure to (priced) risk factors. The basic idea of the regression approach is to project asset returns on factor exposures or characteristics that resemble exposure to a risk factor in the cross-section in each time period. Then, in the second step, the estimates are then aggregated across time to test if a risk factor is priced. In principle, Fama-MacBeth regressions can be used in the same way as portfolio sorts introduced in previous chapters. In this chapter, we present a simple implementation of Fama and MacBeth (1973) to introduce the concept of their regressions. We use individual stocks as test assets to estimate the risk premium associated with the three factors included in Fama and French (1993).

The Fama-MacBeth procedure is a simple two-step approach: The first step uses the exposures (characteristics) as explanatory variables in T cross-sectional regressions, i.e.,

$$r_{i,t+1} = \alpha_i + \lambda_t^M \beta_{i,t}^M + \lambda_t^{SMB} \beta_{i,t}^{SMB} + \lambda_t^{HML} \beta_{i,t}^{HML} + \epsilon_{i,t}, \text{ for each t.}$$

Here, we are interested in the compensation λ_t^f for the exposure to each risk factor $\beta_{i,t}^f$ at each time point, i.e., the risk premium. Note the terminology: $\beta_{i,t}^f$ is a asset-specific characteristic, e.g., a factor exposure or an accounting variable. *If* there is a linear relationship between expected returns and the characteristic in a given month, we expect the regression coefficient to reflect the relationship, i.e., $\lambda_t^f \neq 0$.

In the second step, the time-series average $\frac{1}{T}\sum_{t=1}^{T} \hat{\lambda}_t^f$ of the estimates $\hat{\lambda}_t^f$ can then be interpreted as the risk premium for the specific risk factor f. We follow Zaffaroni and Zhou (2022) and consider the standard cross-sectional regression to predict future returns. If the characteristics are replaced with time $t+1$ variables, the regression approach rather captures risk attributes.

Before we move to the implementation, we want to highlight that the characteristics, e.g., $\hat{\beta}_i^f$, are typically estimated in a separate step before applying the actual Fama-MacBeth methodology. You can think of this as a *step 0*. You might thus worry that the errors of $\hat{\beta}_i^f$ impact the risk premiums' standard errors. Measurement error in $\hat{\beta}_i^f$ indeed affects the risk premium estimates, i.e., they lead to biased estimates. The

literature provides adjustments for this bias (see, e.g., Shanken (1992), Kim (1995), Chen et al. (2015), among others) but also shows that the bias goes to zero as $T \to \infty$. We refer to Gagliardini et al. (2016) for an in-depth discussion also covering the case of time-varying betas. Moreover, if you plan to use Fama-MacBeth regressions with individual stocks: Hou et al. (2020) advocates using weighed-least squares to estimate the coefficients such that they are not biased towards small firms. Without this adjustment, the high number of small firms would drive the coefficient estimates.

8.1 Data preparation

The current chapter relies on this set of packages.

```
library(tidyverse)
library(RSQLite)
library(lubridate)
library(broom)
library(sandwich)
```

We illustrate Fama and MacBeth (1973) with the monthly CRSP sample and use three characteristics to explain the cross-section of returns: market capitalization, the book-to-market ratio, and the CAPM beta (i.e., the covariance of the excess stock returns with the market excess returns). We collect the data from our `SQLite`-database introduced in our chapter on *"Accessing & managing financial data"*.

```
tidy_finance <- dbConnect(SQLite(), "data/tidy_finance.sqlite",
  extended_types = TRUE
)

crsp_monthly <- tbl(tidy_finance, "crsp_monthly") %>%
  collect()

compustat <- tbl(tidy_finance, "compustat") %>%
  collect()

beta <- tbl(tidy_finance, "beta") %>%
  collect()
```

We use the Compustat and CRSP data to compute the book-to-market ratio and the (log) market capitalization. Furthermore, we also use the CAPM betas based on daily returns we computed in the previous chapters.

```
bm <- compustat %>%
  mutate(month = floor_date(ymd(datadate), "month")) %>%
  left_join(crsp_monthly, by = c("gvkey", "month")) %>%
  left_join(beta, by = c("permno", "month")) %>%
  transmute(gvkey,
            bm = be / mktcap,
            log_mktcap = log(mktcap),
            beta = beta_daily,
            sorting_date = month %m+% months(6))

data_fama_macbeth <- crsp_monthly %>%
  left_join(bm, by = c("gvkey", "month" = "sorting_date")) %>%
  group_by(permno) %>%
  arrange(month) %>%
  fill(c(beta, bm, log_mktcap), .direction = "down") %>%
  ungroup() %>%
  left_join(crsp_monthly %>%
              select(permno, month, ret_excess_lead = ret) %>%
              mutate(month = month %m-% months(1)),
            by = c("permno", "month")) %>%
  select(permno, month, ret_excess_lead, beta, log_mktcap, bm) %>%
  drop_na()
```

8.2 Cross-sectional regression

Next, we run the cross-sectional regressions with the characteristics as explanatory variables for each month.
We regress the returns of the test assets at a particular time point on each asset's characteristics. By doing so, we get an estimate of the risk premiums $\widehat{\lambda}_t^{F_f}$ for each point in time.

```
risk_premiums <- data_fama_macbeth %>%
  nest(data = -month) %>%
  mutate(estimates = map(.x = data,
                         ~tidy(lm(ret_excess_lead ~ . - permno, data = .x)))) %>%
  unnest(estimates)
```

8.3 Time-series aggregation

Now that we have the risk premiums' estimates for each period, we can average across the time-series dimension to get the expected risk premium for each characteristic. Similarly, we manually create the t-test statistics for each regressor, which we can then compare to usual critical values of 1.96 or 2.576 for two-tailed significance tests.

```
price_of_risk <- risk_premiums %>%
  group_by(factor = term) %>%
  summarise(
    risk_premium = mean(estimate)*100,
    t_statistic = mean(estimate) / sd(estimate) * sqrt(n())
  )
```

On a final note: It is common to adjust for autocorrelation when reporting standard errors of risk premiums. The typical procedure for this is computing Newey and West (1987) standard errors. One necessary input for Newey-West standard errors is a chosen bandwidth based on the number of lags employed for the estimation. While it seems that researchers often default on choosing a pre-specified lag length of 6 months, we instead recommend a data-driven approach. This automatic selection is advocated by Newey and West (1994) and available in the `sandwich` package thanks to Zeileis (2004). If you want to implement the apparent *default*, you can enforce `NeweyWest(., lag = 6, prewhite = FALSE)` in the code below.

```
regressions_for_newey_west <- risk_premiums %>%
  select(month, factor = term, estimate) %>%
  nest(data = c(month, estimate)) %>%
  mutate(
    model = map(data, ~ lm(estimate ~ 1, .)),
    mean = map(model, tidy)
  )

price_of_risk_newey_west <- regressions_for_newey_west %>%
  mutate(newey_west_se = map_dbl(model, ~ sqrt(NeweyWest(.)))) %>%
  unnest(mean) %>%
  mutate(t_statistic_newey_west = estimate / newey_west_se) %>%
  select(factor,
    risk_premium = estimate,
    t_statistic_newey_west
  )

left_join(price_of_risk,
```

```
          price_of_risk_newey_west %>% select(factor, t_statistic_newey_west),
          by = "factor")
```

```
## # A tibble: 4 x 4
##   factor      risk_premium t_statistic t_statistic_new~
##   <chr>              <dbl>       <dbl>            <dbl>
## 1 (Intercept)       1.62         5.09             4.07
## 2 beta             -0.0587      -0.790           -0.792
## 3 bm                0.177        3.48             2.95
## 4 log_mktcap       -0.114       -3.00            -2.51
```

Finally, let us interpret the results. Stocks with higher book-to-market ratios earn higher expected future returns, which is in line with the value premium. The negative value for log market capitalization reflects the size premium for smaller stocks. Lastly, the negative value for CAPM betas as characteristics is in line with the well-established betting against beta anomalies, indicating that investors with borrowing constraints tilt their portfolio towards high beta stocks to replicate a levered market portfolio (Frazzini and Pedersen, 2014).

8.4 Exercises

1. Download a sample of test assets from Kenneth French's homepage and reevaluate the risk premiums for industry portfolios instead of individual stocks.
2. Use individual stocks with weighted-least squares based on a firm's size as suggested by Hou et al. (2020). Then, repeat the Fama-MacBeth regressions without the weighting scheme adjustment but drop the smallest 20% of firms each month. Compare the results of the three approaches.
3. Implement a rolling-window regression for the time-series estimation of the factor exposure. Skip one month after each rolling period before including the exposures in the cross-sectional regression to avoid a look-ahead bias. Then, adapt the cross-sectional regression and compute the average risk premiums.

Modeling & machine learning

9

Factor selection via machine learning

The aim of this chapter is twofold. From a data science perspective, we introduce `tidymodels`, a collection of packages for modeling and machine learning (ML) using `tidyverse` principles. `tidymodels` comes with a handy workflow for all sorts of typical prediction tasks. From a finance perspective, we address the *factor zoo* (Cochrane, 2011). In previous chapters, we illustrate that stock characteristics such as size provide valuable pricing information in addition to the market beta. Such findings question the usefulness of the Capital Asset Pricing Model. In fact, during the last decades, financial economists "discovered" a plethora of additional factors which may be correlated with the marginal utility of consumption (and would thus deserve a prominent role for pricing applications). The search for factors that explain the cross section of expected stock returns has produced hundreds of potential candidates, as noted more recently by Harvey et al. (2016), McLean and Pontiff (2016) and Hou et al. (2020). Therefore, given the multitude of proposed risk factors, the challenge these days rather is: *Do we believe in the relevance of 300+ risk factors?*. During recent years, promising methods from the vast field of machine learning (ML) got applied to common finance applications. We refer to Mullainathan and Spiess (2017) for a treatment of ML from the perseptive of an econometrician, Nagel (2021) for an excellent review of ML practices in asset pricing, Easley et al. (2020) for ML applications in (high-frequency) market microstructure and Dixon et al. (2020) for a detailed treatment of all methodological aspects.

We introduce Lasso and Ridge regression as a special case of penalized regression models. Then, we explain the concept of cross-validation for model *tuning* with Elastic Net regularization as a popular example. We implement and showcase the entire cycle from model specification, training, and forecast evaluation within the `tidymodels` universe. While the tools can generally be applied to an abundance of interesting asset pricing problems, we apply penalized regressions for identifying macroeconomic variables and asset pricing factors that help explain a cross-section of industry portfolios.

9.1 Brief theoretical background

This is a book about *doing* empirical work in a tidy manner, and we refer to any of the many excellent textbook treatments of ML methods and especially penalized regressions for some deeper discussion (e.g., Hastie et al., 2009, Gareth et al. (2013), De Prado (2018)). Instead, we briefly summarize the idea of Lasso and Ridge regressions as well as the more general Elastic Net. Then, we turn to the fascinating question on *how* to implement, tune, and use such models with the `tidymodels` workflow.

To set the stage, we start with the definition of a linear model: suppose we have data $(y_t, x_t), t = 1, \dots, T$ where x_t is a $(K \times 1)$ vector of regressors and y_t is the response for observation t. The linear model takes the form $y_t = \beta' x_t + \varepsilon_t$ with some error term ε_t and has been studied in abundance. The well-known ordinary-least square (OLS) estimator for the $(K \times 1)$ vector β minimizes the sum of squared residuals and is then

$$\hat{\beta}^{\text{ols}} = \left(\sum_{t=1}^{T} x_t' x_t \right)^{-1} \sum_{t=1}^{T} x_t' y_t.$$

While we are often interested in the estimated coefficient vector $\hat{\beta}^{\text{ols}}$, ML is about the predictive performance most of the time. For a new observation $\tilde{x}_t$, the linear model generates predictions such that

$$\hat{y}_t = E\left(y | x_t = \tilde{x}_t\right) = \hat{\beta}^{\text{ols}\prime} \tilde{x}_t.$$

Is this the best we can do? Not really: Instead of minimizing the sum of squared residuals, penalized linear models can improve predictive performance by choosing other estimators $\hat{\beta}$ with lower variance than the estimator $\hat{\beta}^{\text{ols}}$. At the same time, it seems appealing to restrict the set of regressors to a few meaningful ones if possible. In other words, if K is large (such as for the number of proposed factors in the asset pricing literature), it may be a desirable feature to *select* reasonable factors and set $\hat{\beta}_k^{\text{ols}} = 0$ for some redundant factors.

It should be clear that the promised benefits of penalized regressions (reducing the mean squared error) come at a cost. In most cases, reducing the variance of the estimator introduces a bias such that $E\left(\hat{\beta}\right) \neq \beta$. What is the effect of such a bias-variance trade-off? To understand the implications, assume the following data-generating process for y:

$$y = f(x) + \varepsilon, \quad \varepsilon \sim (0, \sigma_\varepsilon^2)$$

While the properties of $\hat{\beta}^{\text{ols}}$ as an unbiased estimator may be desirable under some circumstances, they are certainly not if we consider predictive accuracy. For in-

stance, the mean-squared error (MSE) depends on our model choice as follow:

$$\begin{aligned} MSE &= E((y - \hat{f}(\mathbf{x}))^2) = E((f(\mathbf{x}) + \epsilon - \hat{f}(\mathbf{x}))^2) \\ &= \underbrace{E((f(\mathbf{x}) - \hat{f}(\mathbf{x}))^2)}_{\text{total quadratic error}} + \underbrace{E(\epsilon^2)}_{\text{irreducible error}} \\ &= E\left(\hat{f}(\mathbf{x})^2\right) + E\left(f(\mathbf{x})^2\right) - 2E\left(f(\mathbf{x})\hat{f}(\mathbf{x})\right) + \sigma_\varepsilon^2 \\ &= E\left(\hat{f}(\mathbf{x})^2\right) + f(\mathbf{x})^2 - 2f(\mathbf{x})E\left(\hat{f}(\mathbf{x})\right) + \sigma_\varepsilon^2 \\ &= \underbrace{\text{Var}\left(\hat{f}(\mathbf{x})\right)}_{\text{variance of model}} + \underbrace{E\left((f(\mathbf{x}) - \hat{f}(\mathbf{x}))\right)^2}_{\text{squared bias}} + \sigma_\varepsilon^2. \end{aligned}$$

While no model can reduce σ_ε^2, a biased estimator with small variance may have a lower mean squared error than an unbiased estimator.

9.1.1 Ridge regression

One biased estimator is known as Ridge regression. Hoerl and Kennard (1970) propose to minimize the sum of squared errors *while simultaneously imposing a penalty on the L_2 norm of the parameters $\hat{\beta}$*. Formally, this means that for a penalty factor $\lambda \geq 0$ the minimization problem takes the form $\min_\beta (y - X\beta)' (y - X\beta)$ s.t. $\beta'\beta \leq c$. Here, $X = (x_1 \dots x_T)'$ and $y = (y_1, \dots, y_T)'$. A closed-form solution for the resulting regression coefficient vector β^{ridge} exists:

$$\beta^{\text{ridge}} = (X'X + \lambda I)^{-1} X'y.$$

A couple of observations are worth noting: $\hat{\beta}^{\text{ridge}} = \hat{\beta}^{\text{ols}}$ for $\lambda = 0$ and $\hat{\beta}^{\text{ridge}} \to 0$ for $\lambda \to \infty$. Also for $\lambda > 0$, $(X'X + \lambda I)$ is non-singular even if $X'X$ is which means that $\hat{\beta}^{\text{ridge}}$ exists even if $\hat{\beta}$ is not defined. However, note also that the Ridge estimator requires careful choice of the hyperparameter λ which controls the *amount of regularization*: A larger value of λ implies *shrinkage* of the regression coefficient towards 0, a smaller value of λ reduces the bias of the resulting estimator.

Usually, X contains an intercept column with ones. As a general rule, the associated intercept coefficient is not penalized. In practice, this often implies that y is simply demeaned before computing $\hat{\beta}^{\text{ridge}}$.

What about the statistical properties of the Ridge estimator? First, the bad news is that $\hat{\beta}^{\text{ridge}}$ is a biased estimator of β. However, the good news is that (under homoscedastic error terms) the variance of the Ridge estimator is guaranteed to be *smaller* than the variance of the ordinary least square estimator. We encourage you to verify these two statements in the exercises. As a result, we face a trade-off: The Ridge regression sacrifices some bias to achieve a smaller variance than the OLS estimator.

9.1.2 Lasso

An alternative to Ridge regression is the Lasso (*l*east *a*bsolute *s*hrinkage and *s*election *o*perator). Similar to Ridge regression, the Lasso (Tibshirani, 1996) is a penalized and biased estimator. The main difference to Ridge regression is that Lasso does not only *shrink* coefficients but effectively selects variables by setting coefficients for *irrelevant* variables to zero. Lasso implements a L_1 penalization on the parameters such that:

$$\widehat{\beta}^{\text{Lasso}} = \arg\min_{\beta} (Y - X\beta)' (Y - X\beta) \text{ s.t. } \sum_{k=1}^{K} |\beta_k| < c.$$

There is no closed form solution for $\widehat{\beta}^{\text{Lasso}}$ in the above maximization problem but efficient algorithms exist (e.g., the R package `glmnet`). Like for Ridge regression, the hyperparameter λ has to be specified beforehand.

9.1.3 Elastic Net

The Elastic Net (Zou and Hastie, 2005) combines L_1 with L_2 penalization and encourages a grouping effect where strongly correlated predictors tend to be in or out of the model together. This more general framework considers the following optimization problem:

$$\widehat{\beta}^{\text{EN}} = \arg\min_{\beta} (Y - X\beta)' (Y - X\beta) + \lambda(1-\rho) \sum_{k=1}^{K} |\beta_k| + \frac{1}{2}\lambda\rho \sum_{k=1}^{K} \beta_k^2$$

Now, we have to chose two hyperparameters: the *shrinkage* factor λ and the *weighting parameter* ρ. The Elastic Net resembles Lasso for $\rho = 1$ and Ridge regression for $\rho = 0$. While the R package `glmnet` provides efficient algorithms to compute the coefficients of penalized regressions, it is a good exercise to implement Ridge and Lasso estimation on your own before you use the `glmnet` package or the `tidymodels` back-end.

9.2 Data preparation

To get started, we load the required packages and data. The main focus is on the workflow behind the amazing `tidymodels` package collection. Kuhn and Silge (2018) provide a thorough introduction into all `tidymodels` components.

```
library(RSQLite)
library(tidyverse)
library(tidymodels)
library(furrr)
```

```
library(glmnet)
library(broom)
library(timetk)
library(scales)
```

In this analysis, we use four different data sources that we load from our `SQLite`-database introduced in our chapter on *"Accessing & managing financial data"*. We start with two different sets of factor portfolio returns which have been suggested as representing practical risk factor exposure and thus should be relevant when it comes to asset pricing applications.

- The standard workhorse: monthly Fama-French 3 factor returns (market, small-minus-big, and high-minus-low book-to-market valuation sorts) defined in Fama and French (1992) and Fama and French (1993)
- Monthly q-factor returns from Hou et al. (2014). The factors contain the size factor, the investment factor, the return-on-equity factor, and the expected growth factor

Next, we include macroeconomic predictors which may predict the general stock market economy. Macroeconomic variables effectively serve as conditioning information such that their inclusion hints at the relevance of conditional models instead of unconditional asset pricing. We refer the interested reader to Cochrane (2009) on the role of conditioning information.

- Our set of macroeconomic predictors comes from the paper "A Comprehensive Look at The Empirical Performance of Equity Premium Prediction" (Welch and Goyal, 2008). The data has been updated by the authors until 2020 and contains monthly variables that have been suggested as good predictors for the equity premium. Some of the variables are the Dividend Price Ratio, Earnings Price Ratio, Stock Variance, Net Equity Expansion, Treasury Bill rate, and inflation

Finally, we need a set of *test assets*. The aim is to understand which of the plenty factors and macroeconomic variable combinations prove helpful in explaining our test assets' cross-section of returns.

- In line with many existing papers, we use monthly portfolio returns from 10 different industries according to the definition from Kenneth French's homepage[1] as test assets.

```
tidy_finance <- dbConnect(SQLite(),
  "data/tidy_finance.sqlite",
  extended_types = TRUE
```

[1] https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/Data_Library/det_10_ind_port.html

```
)

factors_ff_monthly <- tbl(tidy_finance, "factors_ff_monthly") |>
  collect() |>
  rename_with(~ paste0("factor_ff_", .), -month)

factors_q_monthly <- tbl(tidy_finance, "factors_q_monthly") |>
  collect() |>
  rename_with(~ paste0("factor_q_", .), -month)

macro_predictors <- tbl(tidy_finance, "macro_predictors") |>
  collect() |>
  rename_with(~ paste0("macro_", .), -month) |>
  select(-macro_rp_div)

industries_ff_monthly <- tbl(tidy_finance, "industries_ff_monthly") |>
  collect() |>
  pivot_longer(-month,
    names_to = "industry", values_to = "ret"
  ) |>
  mutate(industry = as_factor(industry))
```

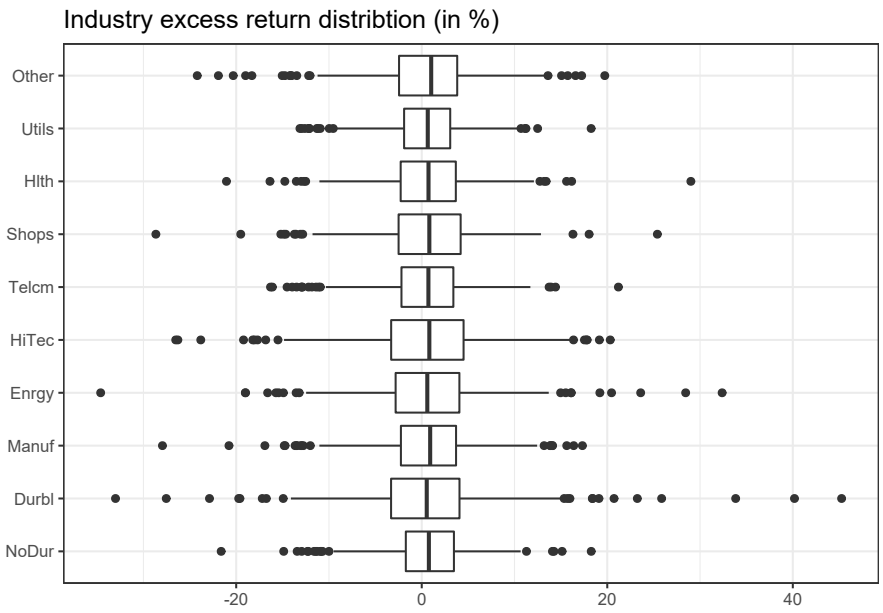
We combine all the monthly observations into one data frame.

```
data <- industries_ff_monthly |>
  left_join(factors_ff_monthly, by = "month") |>
  left_join(factors_q_monthly, by = "month") |>
  left_join(macro_predictors, by = "month") |>
  mutate(
    ret = ret - factor_ff_rf
  ) |>
  select(month, industry, ret, everything()) |>
  drop_na()
```

Our data contains 22 columns of regressors with the 13 macro variables and 8 factor returns for each month. The figure below provides summary statistics for the 10 monthly industry excess returns in percent.

```
data |>
  group_by(industry) |>
  mutate(ret = 100 * ret) |>
  ggplot(aes(x = industry, y = ret)) +
  geom_boxplot() +
  coord_flip() +
```

```
labs(
  x = NULL, y = NULL,
  title = "Industry excess return distribtion (in %)"
)
```



9.3 The tidymodels workflow

To illustrate penalized linear regressions, we employ the `tidymodels` collection of packages for modeling and ML using `tidyverse` principles. You can simply use `install.packages("tidymodels")` to get access to all the related packages. We recommend checking out the work of Max Kuhn and Julia Silge: They continuously write on the great book 'Tidy Modeling with R[2]' using tidy principles.

The `tidymodels` workflow encompasses the main stages of the modeling process: pre-processing of data, model fitting, and post-processing of results. As we demonstrate below, `tidymodels` provides efficient workflows that you can update with low effort.

Using the ideas of Ridge and Lasso regressions, the following example guides you through (i) pre-processing the data (data split and variable mutation), (ii) building

[2]https://www.tmwr.org/

models, (iii) fitting models, and (iv) tuning models to create the "best" possible predictions.

To start, we restrict our analysis to just one industry: Energy. We first split the sample into a *training* and a *test* set. For that purpose, `tidymodels` provides the function `initial_time_split` from the `rsample` package. The split takes the last 20% of the data as a test set, which is not used for any model tuning. We use this test set to evaluate the predictive accuracy in an out-of-sample scenario.

```
split <- initial_time_split(
  data |>
    filter(industry == "Enrgy") |>
    select(-industry),
  prop = 4 / 5
)
split
```

```
## <Training/Testing/Total>
## <517/130/647>
```

The object `split` simply keeps track of the observations of the training and the test set. We can call the training set with `training(split)`, while we can extract the test set with `testing(split)`.

9.3.1 Pre-process data

Recipes help you pre-process your data before training your model. Recipes are a series of pre-processing steps such as variable selection, transformation, or conversion of qualitative predictors to indicator variables. Each recipe starts with a `formula` that defines the general structure of the dataset and the role of each variable (regressor or dependent variable). For our dataset, our recipe contains the following steps before we fit any model:

- Our formula defines that we want to explain excess returns with all available predictors.
- We exclude the column *month* from the analysis.
- We include all interaction terms between factors and macroeconomic predictors.
- We demean and scale each regressor such that the standard deviation is one.

```
rec <- recipe(ret ~ ., data = training(split)) |>
  step_rm(month) |>
  step_interact(terms = ~ contains("factor"):contains("macro")) |>
  step_normalize(all_predictors()) |>
  step_center(ret, skip = TRUE)
```

A table of all available recipe steps can be found here[3]. As of 2022, more than 150 different processing steps are available! One important point: The definition of a recipe does not trigger any calculations yet but rather provides a *description* of the tasks to be applied. As a result, it is very easy to *reuse* recipes for different models and thus make sure that the outcomes are comparable as they are based on the same input. In the example above, it does not make a difference whether you use the input `data = training(split)` or `data = testing(split)`. All that matters at this early stage are the column names and types.

We can apply the recipe to any data with a suitable structure. The code below combines two different functions: `prep` estimates the required parameters from a training set that can be applied to other data sets later. `bake` applies the processed computations to new data.

```
tmp_data <- bake(prep(rec, training(split)), new_data = testing(split))
tmp_data
```

```
## # A tibble: 130 x 126
##   factor_ff_rf factor_ff_mkt_excess factor_ff_smb
##          <dbl>                <dbl>         <dbl>
## 1        -1.92                0.644        0.298
## 2        -1.88                1.27         0.387
## 3        -1.88                0.341        1.43
## 4        -1.88               -1.80        -0.0411
## 5        -1.88               -1.29        -0.627
## # ... with 125 more rows, and 123 more variables:
## #   factor_ff_hml <dbl>, factor_q_me <dbl>,
## #   factor_q_ia <dbl>, factor_q_roe <dbl>,
## #   factor_q_eg <dbl>, macro_dp <dbl>, macro_dy <dbl>,
## #   macro_ep <dbl>, macro_de <dbl>, macro_svar <dbl>,
## #   macro_bm <dbl>, macro_ntis <dbl>, macro_tbl <dbl>,
## #   macro_lty <dbl>, macro_ltr <dbl>, ...
```

Note that the resulting data contains the 130 observations from the test set and 126 columns. Why so many? Recall that the recipe states to compute every possible interaction term between the factors and predictors, which increases the dimension of the data matrix substantially.

You may ask at this stage: Why should I use a recipe instead of simply using the data wrangling commands such as `mutate` or `select`? `tidymodels` beauty is that a lot is happening under the hood. Recall, that for the simple scaling step, you actually have to compute the standard deviation of each column, then *store* this value, and apply the identical transformation to a different dataset, e.g., `testing(split)`. A prepped `recipe` stores these values and hands them on once you `bake` a novel dataset. Easy as pie with `tidymodels`, isn't it?

[3] https://www.tidymodels.org/find/recipes/

9.3.2 Build a model

Next, we can build an actual model based on our pre-processed data. In line with the definition above, we estimate regression coefficients of a Lasso regression such that we get

$$\hat{\beta}_{\lambda}^{\text{Lasso}} = \arg\min_{\beta} (Y - X\beta)' (Y - X\beta) + \lambda \sum_{k=1}^{K} |\beta_k|.$$

We want to emphasize that the `tidymodels` workflow for *any* model is very similar, irrespective of the specific model. As you will see further below, it is straightforward to fit Ridge regression coefficients and - later - Neural networks or Random forests with basically the same code. The structure is always as follows: create a so-called `workflow` and use the `fit` function. A table with all available model APIs is available here[4]. For now, we start with the linear regression model with a given value for the penalty factor λ. In the setup below, `mixture` denotes the value of ρ, hence setting `mixture = 1` implies the Lasso.

```
lm_model <- linear_reg(
  penalty = 0.0001,
  mixture = 1
) |>
  set_engine("glmnet", intercept = FALSE)
```

That's it - we are done! The object `lm_model` contains the definition of our model with all required information. Note that `set_engine("glmnet")` indicates the API character of the `tidymodels` workflow: Under the hood, the package `glmnet` is doing the heavy lifting, while `linear_reg` provides a unified framework to collect the inputs. The `workflow` ends with combining everything necessary for the (serious) data science workflow, namely, a recipe and a model.

```
lm_fit <- workflow() |>
  add_recipe(rec) |>
  add_model(lm_model)
lm_fit
```

```
## == Workflow ============================================
## Preprocessor: Recipe
## Model: linear_reg()
##
## -- Preprocessor ----------------------------------------
## 4 Recipe Steps
##
## * step_rm()
```

[4] https://www.tidymodels.org/find/parsnip/

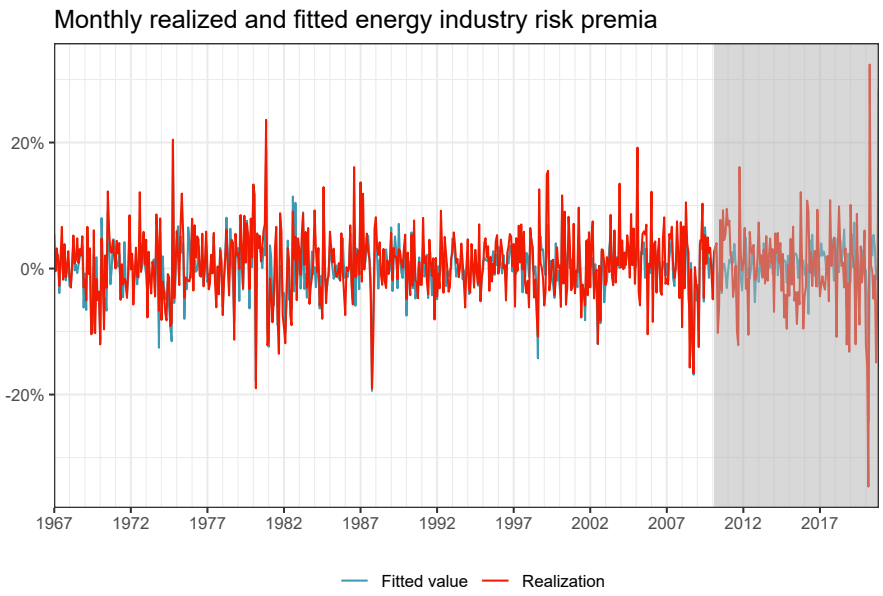
```
## * step_interact()
## * step_normalize()
## * step_center()
##
## -- Model ----------------------------------------------
## Linear Regression Model Specification (regression)
##
## Main Arguments:
##   penalty = 1e-04
##   mixture = 1
##
## Engine-Specific Arguments:
##   intercept = FALSE
##
## Computational engine: glmnet
```

9.3.3 Fit a model

With the `workflow` from above, we are ready to use `fit`. Typically, we use training data to fit the model. The training data is pre-processed according to our recipe steps, and the Lasso regression coefficients are computed. First, we focus on the predicted values $\widehat{y}_t = x_t \widehat{\beta}^{\text{Lasso}}$. The figure below illustrates the projections for the *entire* time series of the Energy industry portfolio returns. The grey area indicates the out-of-sample period, which we did not use to fit the model.

```
predicted_values <- lm_fit |>
  fit(data = training(split)) |>
  predict(data |> filter(industry == "Enrgy")) |>
  bind_cols(data |> filter(industry == "Enrgy")) |>
  select(month,
    "Fitted value" = .pred,
    "Realization" = ret
  ) |>
  pivot_longer(-month, names_to = "Variable")

predicted_values |>
  ggplot(aes(x = month, y = value, color = Variable)) +
  geom_line() +
  labs(
    x = NULL,
    y = NULL,
    color = NULL,
    title = "Monthly realized and fitted energy industry risk premia"
  ) +
```

```
  scale_x_date(
    breaks = function(x) seq.Date(from = min(x), to = max(x), by = "5 years"),
    minor_breaks = function(x) seq.Date(from = min(x), to = max(x), by = "1 years"),
    expand = c(0, 0),
    labels = date_format("%Y")
  ) +
  scale_y_continuous(
    labels = percent
  ) +
  annotate("rect",
    xmin = testing(split) |> pull(month) |> min(),
    xmax = testing(split) |> pull(month) |> max(),
    ymin = -Inf, ymax = Inf,
    alpha = 0.5, fill = "grey70"
  )
```

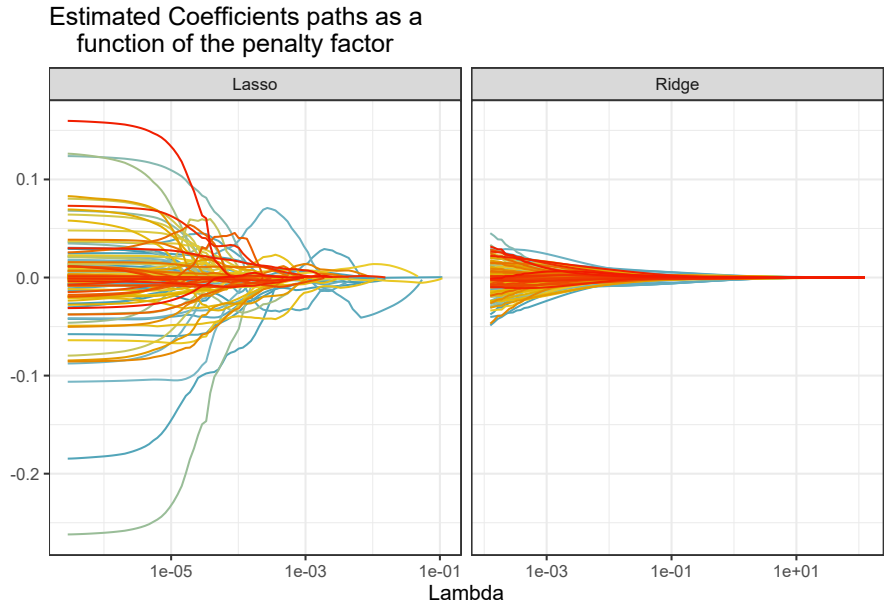


What do the estimated coefficients look like? To analyze these values and to illustrate the difference between the `tidymodels` workflow and the underlying `glmnet` package, it is worth computing the coefficients $\widehat{\beta}^{\text{Lasso}}$ directly. The code below estimates the coefficients for the Lasso and Ridge regression for the processed training data sample. Note that `glmnet` actually takes a vector `y` and the matrix of regressors X as input. Moreover, `glmnet` requires choosing the penalty parameter α, which corresponds to ρ in the notation above. When using the `tidymodels` model API, such details do not need consideration.

```
x <- tmp_data |>
  select(-ret) |>
  as.matrix()
y <- tmp_data |> pull(ret)

fit_lasso <- glmnet(
  x = x,
  y = y,
  alpha = 1,
  intercept = FALSE,
  standardize = FALSE,
  lambda.min.ratio = 0
)

fit_ridge <- glmnet(
  x = x,
  y = y,
  alpha = 0,
  intercept = FALSE,
  standardize = FALSE,
  lambda.min.ratio = 0
)
```

The objects `fit_lasso` and `fit_ridge` contain an entire sequence of estimated coefficients for multiple values of the penalty factor λ. The figure below illustrates the trajectories of the regression coefficients as a function of the penalty factor. Both Lasso and Ridge coefficients converge to zero as the penalty factor increases.

```
bind_rows(
  tidy(fit_lasso) |> mutate(Model = "Lasso"),
  tidy(fit_ridge) |> mutate(Model = "Ridge")
) |>
  rename("Variable" = term) |>
  ggplot(aes(x = lambda, y = estimate, color = Variable)) +
  geom_line() +
  scale_x_log10() +
  facet_wrap(~Model, scales = "free_x") +
  labs(
    x = "Lambda", y = NULL,
    title = "Estimated Coefficients paths as a
    function of the penalty factor"
  ) +
  theme(legend.position = "none")
```



One word of caution: The package `glmnet` computes estimates of the coefficients $\hat{\beta}$ based on numerical optimization procedures. As a result, the estimated coefficients for the special case[5] with no regularization ($\lambda = 0$) can deviate from the standard OLS estimates.

9.3.4 Tune a model

To compute $\hat{\beta}_{\lambda}^{\text{Lasso}}$, we simply imposed a value for the penalty hyperparameter λ. Model tuning is the process of optimally selecting such hyperparameters. `tidymodels` provides extensive tuning options based on so-called *cross-validation*. Again, we refer to any treatment of cross-validation to get a more detailed discussion of the statistical underpinnings. Here we focus on the general idea and the implementation with `tidymodels`.

The goal for choosing λ (or any other hyperparameter, e.g., ρ) is to find a way to produce predictors $\hat{Y}$ for an outcome Y that minimizes the mean squared prediction error MSPE $= E\left(\frac{1}{T}\sum_{t=1}^{T}(\hat{y}_t - y_t)^2\right)$. Unfortunately, the MSPE is not directly observable. We can only compute an estimate because our data is random and because we do not observe the entire population.

Obviously, if we train an algorithm on the same data that we use to compute the error, our estimate $\widehat{\text{MSPE}}$ would indicate way better predictive accuracy than what we can expect in real out-of-sample data. The result is called overfitting.

[5] https://parsnip.tidymodels.org/reference/glmnet-details.html

Cross-validation is a technique that allows us to alleviate this problem. We approximate the true MSPE as the average of many mean squared prediction errors obtained by creating predictions for K new random samples of the data, none of them used to train the algorithm $\frac{1}{K}\sum_{k=1}^{K}\frac{1}{T}\sum_{t=1}^{T}\left(\hat{y}_t^k - y_t^k\right)^2$. In practice, this is done by carving out a piece of our data and pretending it is an independent sample. We again divide the data into a training set and a test set. The MSPE on the test set is our measure for actual predictive ability, while we use the training set to fit models with the aim to find the *optimal* hyperparameter values. To do so, we further divide our training sample into (several) subsets, fit our model for a grid of potential hyperparameter values (e.g., λ), and evaluate the predictive accuracy on an *independent* sample. This works as follows:

1. Specify a grid of hyperparameters.
2. Obtain predictors $\widehat{y}_i(\lambda)$ to denote the predictors for the used parameters λ.
3. Compute
$$\text{MSPE}(\lambda) = \frac{1}{K}\sum_{k=1}^{K}\frac{1}{T}\sum_{t=1}^{T}\left(\hat{y}_t^k(\lambda) - y_t^k\right)^2$$
With K-fold cross-validation, we do this computation K times. Simply pick a validation set with $M = T/K$ observations at random and think of these as random samples $y_1^k, \ldots, y_{\tilde{T}}^k$, with $k = 1$.

How should you pick K? Large values of K are preferable because the training data better imitates the original data. However, larger values of K will have much higher computation time. `tidymodels` provides all required tools to conduct K-fold cross-validation. We just have to update our model specification and let `tidymodels` know which parameters to tune. In our case, we specify the penalty factor λ as well as the mixing factor ρ as *free* parameters. Note that it is simple to change an existing `workflow` with `update_model`.

```
lm_model <- linear_reg(
  penalty = tune(),
  mixture = tune()
) |>
  set_engine("glmnet")

lm_fit <- lm_fit |>
  update_model(lm_model)
```

For our sample, we consider a time-series cross-validation sample. This means that we tune our models with 20 random samples of length five years with a validation period of four years. For a grid of possible hyperparameters, we then fit the model for each fold and evaluate MSPE in the corresponding validation set. Finally, we select the model specification with the lowest MSPE in the validation set. First, we define the cross-validation folds based on our training data only.

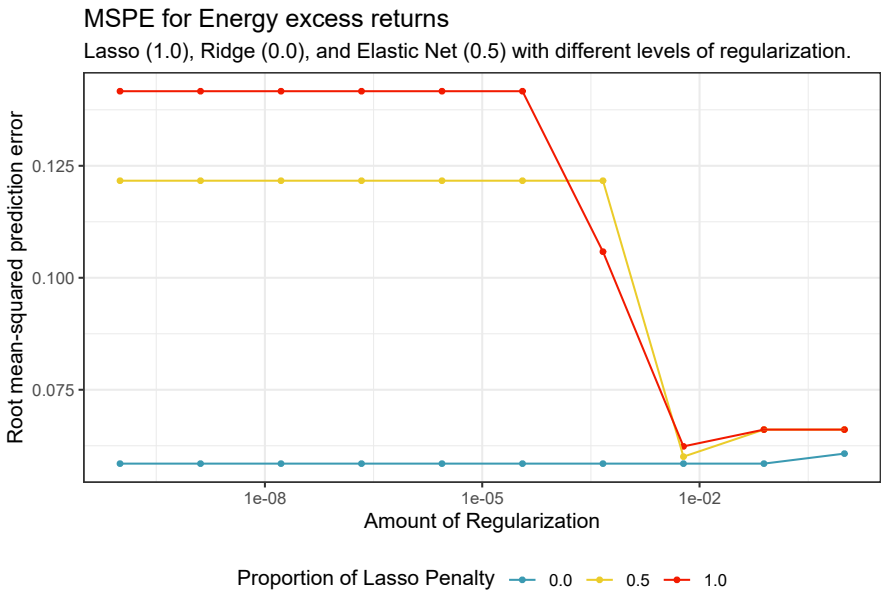
```
data_folds <- time_series_cv(
  data        = training(split),
  date_var    = month,
  initial     = "5 years",
  assess      = "48 months",
  cumulative  = FALSE,
  slice_limit = 20
)
```

Then, we evaluate the performance for a grid of different penalty values. `tidymodels` provides functionalities to construct a suitable grid of hyperparameters with `grid_regular`. The code chunk below creates a 10×3 hyperparameters grid. Then, the function `tune_grid` evaluates all the models for each fold.

```
lm_tune <- lm_fit |>
  tune_grid(
    resample = data_folds,
    grid = grid_regular(penalty(), mixture(), levels = c(10, 3)),
    metrics = metric_set(rmse)
  )
```

After the tuning process, we collect the evaluation metrics (the root mean-squared error in our example) to identify the *optimal* model. The figure illustrates the average validation set's root mean-squared error for each value of λ and ρ.

```
autoplot(lm_tune) +
  labs(
    y = "Root mean-squared prediction error",
    title = "MSPE for Energy excess returns",
    subtitle = "Lasso (1.0), Ridge (0.0), and Elastic Net (0.5) with different levels of regularization."
  )
```



The figure shows that the cross-validated mean squared prediction error drops for Lasso and Elastic Net and spikes afterward. For Ridge regression, the MSPE increases above a certain threshold. Recall that the larger the regularization, the more restricted the model becomes. Thus, we would choose the model with the lowest MSPE, which exhibits some intermediate level of regularization.

9.3.5 Parallelized workflow

Our starting point was the question: Which factors determine industry returns? To illustrate the entire workflow, we now run the penalized regressions for all ten industries. We want to identify relevant variables by fitting Lasso models for each industry returns time series. More specifically, we perform cross-validation for each industry to identify the optimal penalty factor λ. Then, we use the set of `finalize_*`-functions that take a list or tibble of tuning parameter values and update objects with those values. After determining the best model, we compute the final fit on the entire training set and analyze the estimated coefficients.

First, we define the Lasso model with one tuning parameter.

```
lasso_model <- linear_reg(
  penalty = tune(),
  mixture = 1
) |>
  set_engine("glmnet")
```

```
lm_fit <- lm_fit |>
  update_model(lasso_model)
```

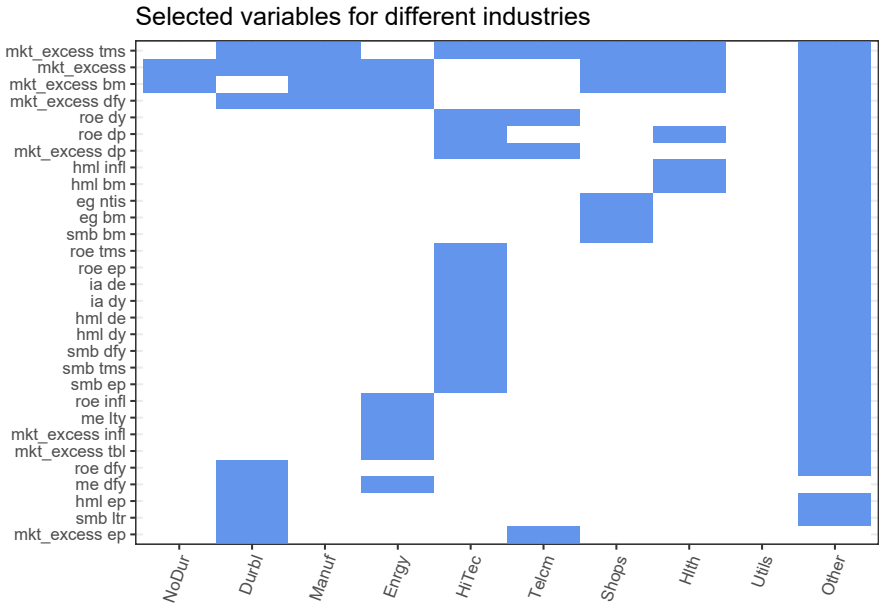
The following task can be easily parallelized to reduce computing time substantially. We use the parallelization capabilities of `furrr`. Note that we can also just recycle all the steps from above and collect them in a function.

```
select_variables <- function(input) {
  # Split into training and testing data
  split <- initial_time_split(input, prop = 4 / 5)

  # Data folds for cross-validation
  data_folds <- time_series_cv(
    data = training(split),
    date_var = month,
    initial = "5 years",
    assess = "48 months",
    cumulative = FALSE,
    slice_limit = 20
  )

  # Model tuning with the Lasso model
  lm_tune <- lm_fit |>
    tune_grid(
      resample = data_folds,
      grid = grid_regular(penalty(), levels = c(10)),
      metrics = metric_set(rmse)
    )

  # Finalizing: Identify the best model and fit with the training data
  lasso_lowest_rmse <- lm_tune |> select_by_one_std_err("rmse")
  lasso_final <- finalize_workflow(lm_fit, lasso_lowest_rmse)
  lasso_final_fit <- last_fit(lasso_final, split, metrics = metric_set(rmse))

  # Extract the estimated coefficients
  lasso_final_fit |>
    extract_fit_parsnip() |>
    tidy() |>
    mutate(
      term = gsub("factor_|macro_|industry_", "", term)
    )
}

# Parallelization
```

```
plan(multisession, workers = availableCores())

# Computation by industry
selected_factors <- data |>
  nest(data = -industry) |>
  mutate(selected_variables = future_map(data, select_variables,
    .options = furrr_options(seed = TRUE)
  ))
```

What has just happened? In principle, exactly the same as before but instead of computing the Lasso coefficients for one industry, we did it for ten in parallel. The final option `seed = TRUE` is required to make the cross-validation process reproducible. Now, we just have to do some housekeeping and keep only variables that Lasso does *not* set to zero. We illustrate the results in a heat map.

```
selected_factors |>
  unnest(selected_variables) |>
  filter(
    term != "(Intercept)",
    estimate != 0
  ) |>
  add_count(term) |>
  mutate(
    term = gsub("NA|ff_|q_", "", term),
    term = gsub("_x_", " ", term),
    term = fct_reorder(as_factor(term), n),
    term = fct_lump_min(term, min = 2),
    selected = 1
  ) |>
  filter(term != "Other") |>
  mutate(term = fct_drop(term)) |>
  complete(industry, term, fill = list(selected = 0)) |>
  ggplot(aes(industry,
    term,
    fill = as_factor(selected)
  )) +
  geom_tile() +
  scale_x_discrete(guide = guide_axis(angle = 70)) +
  scale_fill_manual(values = c("white", "cornflowerblue")) +
  theme(legend.position = "None") +
  labs(
    x = NULL, y = NULL,
    title = "Selected variables for different industries"
  )
```



The heat map conveys two main insights. First, we see a lot of white, which means that many factors, macroeconomic variables, and interaction terms are not relevant for explaining the cross-section of returns across the industry portfolios. In fact, only the market factor and the return-on-equity factor play a role for several industries. Second, there seems to be quite some heterogeneity across different industries. While not even the market factor is selected by Lasso for Utilities (which means the proposed model essentially just contains an intercept), many factors are selected for, e.g., High-Tech and Energy, but they do not coincide at all. In other words, there seems to be a clear picture that we do not need many factors, but Lasso does not provide conses across industries when it comes to pricing abilities.

9.4 Exercises

1. Write a function that requires three inputs, namely, `y` (a T vector), `x` (a $(T \times K)$ matrix), and `lambda` and then returns the **Ridge** estimator (a K vector) for a given penalization parameter λ. Recall that the intercept should not be penalized. Therefore, your function should indicate whether X contains a vector of ones as the first column, which should be exempt from the L_2 penalty.
2. Compute the L_2 norm ($\beta'\beta$) for the regression coefficients based on the

predictive regression from the previous exercise for a range of λ's and illustrate the effect of penalization in a suitable figure.

3. Now, write a function that requires three inputs, namely,`y` (a T vector), `X` (a $(T \times K)$ matrix), and 'lambda' and then returns the **Lasso** estimator (a K vector) for a given penalization parameter λ. Recall that the intercept should not be penalized. Therefore, your function should indicate whether X contains a vector of ones as the first column, which should be exempt from the L_1 penalty.
4. After you understand what Ridge and Lasso regressions are doing, familiarize yourself with the `glmnet()` package's documentation. It is a thoroughly tested and well-established package that provides efficient code to compute the penalized regression coefficients for Ridge and Lasso and for combinations, commonly called *Elastic Nets*.

10

Option pricing via machine learning

Machine learning (ML) is seen as a part of artificial intelligence. ML algorithms build a model based on training data in order to make predictions or decisions without being explicitly programmed to do so. While ML can be specified along a vast array of different branches, this chapter focuses on so-called supervised learning for regressions. The basic idea of supervised learning algorithms is to build a mathematical model for data that contains both the inputs and the desired outputs. In this chapter, we apply well-known methods such as random forests and neural networks to a simple application in option pricing. More specifically, we are going to create an artificial dataset of option prices for different values based on the Black-Scholes pricing equation for call options. Then, we train different models to *learn* how to price call options without prior knowledge of the theoretical underpinnings of the famous option pricing equation.

The roadmap is as follows: First, we briefly introduce regression trees, random forests, and neural networks. As the focus is on implementation, we leave a thorough treatment of the statistical underpinnings to other textbooks from authors with a real comparative advantage on these issues. We show how to implement random forests and deep neural networks with tidy principles using `tidymodels` or `tensorflow` for more complicated network structures.

In order to replicate the analysis regarding neural networks in this chapter, you have to install `TensorFlow` on your system, which requires administrator rights on your machine. Parts of this can be done from within R. Just follow these quick-start instructions[1].

Throughout this chapter, we need the following packages.

```
library(tidyverse)
library(tidymodels)
library(keras)
library(hardhat)
```

[1] https://tensorflow.rstudio.com/installation/

10.1 Regression trees and random forests

Regression trees are a popular ML approach for incorporating multiway predictor interactions. Trees are fully nonparametric and possess a logic that departs markedly from traditional regressions. Trees are designed to find groups of observations that behave similarly to each other. A tree "grows" in a sequence of steps. At each step, a new "branch" sorts the data leftover from the preceding step into bins based on one of the predictor variables. This sequential branching slices the space of predictors into rectangular partitions and approximates the unknown function $f(x)$ with the average value of the outcome variable within each partition.

We partition the predictor space into J non-overlapping regions, $R_1, R_2, \dots, R_J$. For any predictor x that falls within region R_j we estimate $f(x)$ with the average of the training observations, $\hat{y}_i$, for which the associated predictor x_i is also in R_j. Once we select a partition $\mathbf{x}$ to split in order to create the new partitions, we find a predictor j and value s that define two new partitions, called $R_1(j,s)$ and $R_2(j,s)$, which split our observations in the current partition by asking if x_j is bigger than s:

$$R_1(j,s) = \{\mathbf{x} \mid x_j < s\} \text{ and } R_2(j,s) = \{\mathbf{x} \mid x_j \geq s\}$$

To pick j and s, we find the pair that minimizes the residual sum of square (RSS):

$$\sum_{i:\, x_i \in R_1(j,s)} (y_i - y_{R_1})^2 + \sum_{i:\, x_i \in R_2(j,s)} (y_i - y_{R_2})^2$$

Note: Unlike the case for the sample variance, we do not scale by the number of elements $R_k(j,s)$! As in the chapter on penalized regressions, the first relevant question is: What are the hyperparameter decisions? Instead of a regularization parameter, trees are fully determined by the number of branches used to generate a partition (sometimes one specifies the minimum number of observations in each final branch instead of the maximum number of branches).

Models with a single tree suffer from high variance. Random forests address the shortcomings of decision trees. The goal is to improve the predictive performance and reduce instability by averaging multiple decision trees (a forest of trees constructed with randomness). A forest basically implies creating many regression trees and averaging their predictions. To assure that the individual trees are not the same, we use a bootstrap to induce randomness. More specifically, we build B decision trees $T_1, \dots, T_B$ using the training sample. For that purpose, we randomly select features to be included in the building of each tree. For each observation in the test set we then form a prediction $\hat{y} = \frac{1}{B} \sum_{i=1}^{B} \hat{y}_{T_i}$.

10.2 Neural networks

Roughly speaking, neural networks propagate information from an input layer, through one or multiple hidden layers, to an output layer. While the number of units (neurons) in the input layer is equal to the dimension of the predictors, the output layer usually consists of one neuron (for regression) or multiple neurons for classification. The output layer predicts the future data, similar to the fitted value in a regression analysis. Neural networks have theoretical underpinnings as "universal approximators" for any smooth predictive association (Hornik, 1991). Their complexity, however, ranks neural networks among the least transparent, least interpretable, and most highly parameterized ML tools.

Each neuron applies a nonlinear "activation function" f to its aggregated signal before sending its output to the next layer

$$x_k^l = f\left(\theta_0^k + \sum_{j=1}^{N^l} z_j \theta_{l,j}^k\right)$$

While the easiest case with $f(x) = \alpha + \beta x$ resembles linear regression, typical activation functions are sigmoid (i.e., $f(x) = (1 + e^{-x})^{-1}$) or ReLu (i.e., $f(x) = max(x, 0)$).

Neural networks gain their flexibility from chaining multiple layers together. Naturally, this imposes many degrees of freedom on the network architecture for which no clear theoretical guidance exists. The specification of a neural network requires, at a minimum, a stance on depth (number of hidden layers), the activation function, the number of neurons, the connection structure of the units (dense or sparse), and the application of regularization techniques to avoid overfitting. Finally, *learning* means to choose optimal parameters relying on numerical optimization, which often requires specifying an appropriate learning rate.

Despite the computational challenges, implementation in R is not tedious at all because we can use the API to `tensorflow`.

10.3 Option pricing

To apply ML methods in a relevant field of finance, we focus on option pricing. The application in its core is taken from Hull (2020). In its most basic form, call options give the owner the right but not the obligation to buy a specific stock (the underlying) at a specific price (the strike price K) at a specific date (the exercise date T). The

Black–Scholes price (Black and Scholes, 1973) of a call option for a non-dividend-paying underlying stock is given by

$$C(S,T) = \Phi(d_1)S - \Phi(d_1 - \sigma\sqrt{T})Ke^{-rT}$$

$$d_1 = \frac{1}{\sigma\sqrt{T}}\left[\ln\left(\frac{S}{K}\right) + \left(r_f + \frac{\sigma^2}{2}\right)T\right]$$

where $C(S,T)$ is the price of the option as a function of today's stock price of the underlying, S, with time to maturity$T$, r_f is the risk-free interest rate, and σ is the volatility of the underlying stock return. Φ is the cumulative distribution function of a standard normal random variable.

The Black-Scholes equation provides a way to compute the arbitrage-free price of a call option once the parameters S, K, r_f, T, and σ are specified (arguably, all parameters are easy to specify except for σ which has to be estimated). A simple R function allows computing the price as we do below.

```
black_scholes_price <- function(S = 50, K = 70, r = 0, T = 1, sigma = 0.2) {
  d1 <- (log(S / K) + (r + sigma^2 / 2) * T) / (sigma * sqrt(T))
  value <- S * pnorm(d1) - K * exp(-r * T) * pnorm(d1 - sigma * sqrt(T))
  return(value)
}
```

10.4 Learning Black-Scholes

We illustrate the concept of ML by showing how ML methods *learn* the Black-Scholes equation after observing some different specifications and corresponding prices without us revealing the exact pricing equation.

10.4.1 Data simulation

To that end, we start with simulated data. We compute option prices for call options for a grid of different combinations of times to maturity (`T`), risk-free rates (`r`), volatilities (`sigma`), strike prices (`K`), and current stock prices (`S`). In the code below, we add an idiosyncratic error term to each observation such that the prices considered do not exactly reflect the values implied by the Black-Scholes equation.

```
option_prices <- expand_grid(
  S = 40:60, # Stock price
  K = 20:90, # Strike price
  r = seq(from = 0, to = 0.05, by = 0.01), # Risk-free rate
  T = seq(from = 3 / 12, to = 2, by = 1 / 12), # Time to maturity
```

```
  sigma = seq(from = 0.1, to = 0.8, by = 0.1) # Volatility
) |>
  mutate(
    black_scholes = black_scholes_price(S, K, r, T, sigma),
    observed_price = map(black_scholes,
                         function(x) x + rnorm(2, sd = 0.15))
  ) |>
  unnest(observed_price)
```

The code above generates more than 1.5 million random parameter constellations. For each of these values, two *observed* prices reflecting the Black-Scholes prices are given and a random innovation term *pollutes* the observed prices.

Next, we split the data into a training set (which contains 1% of all the observed option prices) and a test set that will only be used for the final evaluation. Note that the entire grid of possible combinations contains 3148992 different specifications. Thus, the sample to learn the Black-Scholes price contains only 31489 observations and is therefore relatively small. In order to keep the analysis reproducible, we use `set.seed()`. A random seed specifies the start point when a computer generates a random number sequence and ensures that our simulated data is the same across different machines.

```
set.seed(420)
split <- initial_split(option_prices, prop = 1 / 100)
```

We process the training dataset further before we fit the different ML models. We define a `recipe` that defines all processing steps for that purpose. For our specific case, we want to explain the observed price by the five variables that enter the Black-Scholes equation. The *true* price (stored in `black_scholes`) should obviously not be used to fit the model. The recipe also reflects that we standardize all predictors to ensure that each variable exhibits a sample average of zero and a sample standard deviation of one.

```
rec <- recipe(observed_price ~ .,
  data = option_prices
) |>
  step_rm(black_scholes) |>
  step_normalize(all_predictors())
```

10.4.2 Single layer networks and random forests

Next, we show how to fit a neural network to the data. Note that this requires that `keras` is installed on your local machine. The function `mlp` from the package `parsnip` provides the functionality to initialize a single layer, feed-forward neural network.

The specification below defines a single layer feed-forward neural network with 15 hidden units. We set the number of training iterations to `epochs = 500`. The option `set_mode("regression")` specifies a linear activation function for the output layer.

```
nnet_model <- mlp(
  epochs = 500,
  hidden_units = 10,
) |>
  set_mode("regression") |>
  set_engine("keras", verbose = FALSE)
```

The `verbose=0` argument prevents logging the results. We can follow the straightforward `tidymodel` workflow as in the chapter before: Define a workflow, equip it with the recipe, and specify the associated model. Finally, fit the model with the training data.

```
nn_fit <- workflow() |>
  add_recipe(rec) |>
  add_model(nnet_model) |>
  fit(data = training(split))
```

Once you are familiar with the `tidymodel` workflow, it is a piece of cake to fit other models from the `parsnip` family. For instance, the model below initializes a random forest with 50 trees contained in the ensemble where we require at least 20 observations in a node.

```
rf_model <- rand_forest(
  trees = 50,
  min_n = 20
) |>
  set_engine("ranger") |>
  set_mode("regression")
```

Fitting the model follows exactly the same convention as for the neural network before.

```
rf_fit <- workflow() |>
  add_recipe(rec) |>
  add_model(rf_model) |>
  fit(data = training(split))
```

10.4.3 Deep neural networks

Note that while the `tidymodels` workflow is extremely convenient, more sophisticated *deep* neural networks are not supported yet (as of January 2022). For that reason, the

code snippet below illustrates how to initialize a sequential model with three hidden layers with 10 units per layer. The `keras` package provides a convenient interface and is flexible enough to handle different activation functions. The `compile` command defines the loss function with which the model predictions are evaluated.

```
model <- keras_model_sequential() |>
  layer_dense(units = 10, activation = "sigmoid",
              input_shape = 5) |>
  layer_dense(units = 10, activation = "sigmoid") |>
  layer_dense(units = 10, activation = "sigmoid") |>
  layer_dense(units = 1, activation = "linear") |>
  compile(
    loss = "mean_absolute_error"
  )
model
```

```
## Model: "sequential_1"
## _________________________________________________________
##  Layer (type)           Output Shape          Param #
## =========================================================
##  dense_5 (Dense)        (None, 10)            60
##  dense_4 (Dense)        (None, 10)            110
##  dense_3 (Dense)        (None, 10)            110
##  dense_2 (Dense)        (None, 1)             11
## =========================================================
## Total params: 291
## Trainable params: 291
## Non-trainable params: 0
## _________________________________________________________
```

To train the neural network, we provide the inputs (`x`) and the variable to predict (`y`) and then fit the parameters. Note the slightly tedious use of the method `extract_mold(nn_fit)`. Instead of simply using the *raw* data, we fit the neural network with the same processed data that is used for the single-layer feed-forward network. What is the difference to simply calling `x = training(data) |> select(-observed_price, -black_scholes)`? Recall that the recipe standardizes the variables such that all columns have unit standard deviation and zero mean. Further, it adds consistency if we ensure that all models are trained using the same recipe such that a change in the recipe is reflected in the performance of any model. A final note on a potentially irritating observation: Note that `fit()` alters the `keras` model - this is one of the few instances where a function in R alters the *input* such that after the function call the object `model` is not same anymore!

```
model |>
  fit(
    x = extract_mold(nn_fit)$predictors |> as.matrix(),
    y = extract_mold(nn_fit)$outcomes |> pull(observed_price),
    epochs = 500, verbose = FALSE
  )
```

10.4.4 Universal approximation

Before we evaluate the results, we implement one more model: In principle, any non-linear function can also be approximated by a linear model containing the input variables' polynomial expansions. To illustrate this, we first define a new recipe, `rec_linear`, which processes the training data even further. We include polynomials up to the tenth degree of each predictor and then add all possible pairwise interaction terms. The final recipe step, `step_lincomb`, removes potentially redundant variables (for instance, the interaction between r^4 and r^5 is the same as the term r^9). We fit a Lasso regression model with a pre-specified penalty term (consult the chapter on factor selection on how to tune the model hyperparameters).

```
rec_linear <- rec |>
  step_poly(all_predictors(),
            degree = 10,
            options = list(raw = T)) |>
  step_interact(terms = ~ all_predictors():all_predictors()) |>
  step_lincomb(all_predictors())

lm_model <- linear_reg(penalty = 0.01) |>
  set_engine("glmnet")

lm_fit <- workflow() |>
  add_recipe(rec_linear) |>
  add_model(lm_model) |>
  fit(data = training(split))
```

10.5 Prediction evaluation

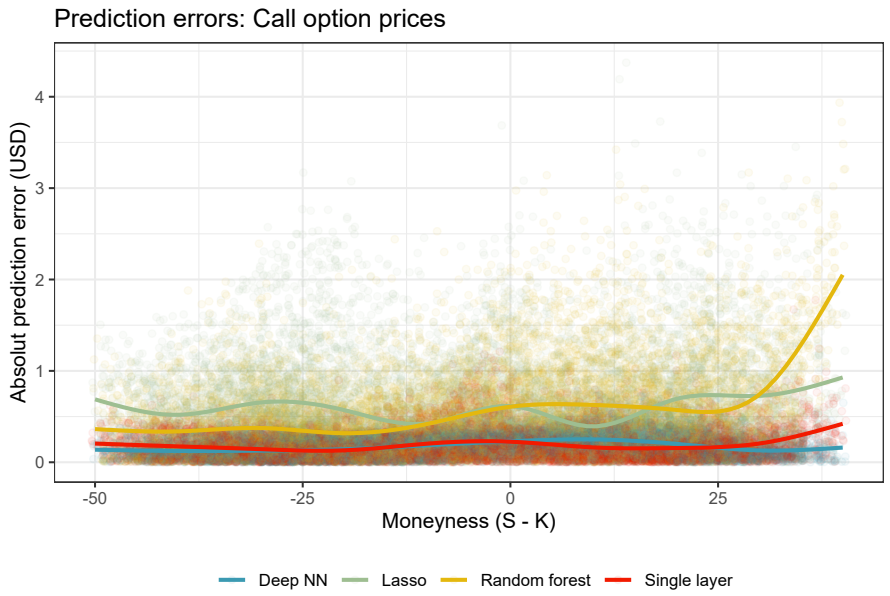
Finally, we collect all predictions to compare the *out-of-sample* prediction error evaluated on ten thousand new data points. Note that for the evaluation, we use the call to `extract_mold` to ensure that we use the same pre-processing steps for the testing

data across each model. We also use the somewhat advanced functionality in `hardhat::forge`, which provides an easy, consistent, and robust pre-processor at prediction time.

```
out_of_sample_data <- testing(split) |> slice_sample(n = 10000)

predictive_performance <- tibble("Deep NN" = model |>
    predict(forge(out_of_sample_data,
                  extract_mold(nn_fit)$blueprint)$predictors |> as.matrix()) |> as.vector()) |>
  bind_cols(nn_fit |>
    predict(out_of_sample_data)) |>
  rename("Single layer" = .pred) |>
  bind_cols(lm_fit |> predict(out_of_sample_data)) |>
  rename("Lasso" = .pred) |>
  bind_cols(rf_fit |> predict(out_of_sample_data)) |>
  rename("Random forest" = .pred) |>
  bind_cols(out_of_sample_data) |>
  pivot_longer("Deep NN":"Random forest", names_to = "Model") |>
  mutate(
    moneyness = (S - K),
    pricing_error = abs(value - black_scholes)
  )
```

In the lines above, we use each of the fitted models to generate predictions for the entire test data set of option prices. We evaluate the absolute pricing error as one possible measure of pricing accuracy, defined as the absolute value of the difference between predicted option price and the theoretical correct option price from the Black-Scholes model.

```
predictive_performance |>
  ggplot(aes(x = moneyness, y = pricing_error, color = Model)) +
  geom_jitter(alpha = 0.05) +
  geom_smooth(se = FALSE) +
  labs(
    x = "Moneyness (S - K)", color = NULL,
    y = "Absolut prediction error (USD)",
    title = "Prediction errors: Call option prices"
  )
```



The results can be summarized as follow: i) All ML methods seem to be able to *price* call options after observing the training test set. ii) The average prediction errors increase for far in-the money options, especially for the Single Layer neural network and Random Forests. ii) Random forest and the Lasso seem to perform consistently worse in prediction option prices than the Neural networks. iii) The complexity of the deep neural network relative to the single layer neural network does not result in better out-of-sample predictions.

10.6 Exercises

1. Write a function that takes `y` and a matrix of predictors `X` as inputs and returns a characterization of the relevant parameters of a regression tree with **1** branch.
2. Create a function that creates predictions for a new matrix of predictors 'newX´ based on the estimated regression tree.
3. Use the package `rpart` to *grow* a tree based on the training data and use the illustration tools in `rpart` to understand which characteristics the tree deems relevant for option pricing.
4. Make use of a training and a test set to choose the optimal depth (number of sample splits) of the tree.

5. Use 'keras' to initialize a sequential neural network that can take the predictors from the training dataset as input, contains at least one hidden layer, and generates continuous predictions. *This sounds harder than it is:* see a simple regression example here[2]. How many parameters does the neural network you aim to fit have?
6. Next, compile the object. It is important that you specify a loss function. Illustrate the difference in predictive accuracy for different architecture choices.

[2]https://tensorflow.rstudio.com/tutorials/beginners/basic-ml/tutorial_basic_regression/

Portfolio optimization

11

Parametric portfolio policies

In this chapter, we introduce different portfolio performance measures to evaluate and compare portfolio allocation strategies. For this purpose, we introduce a direct way to estimate optimal portfolio weights for large-scale cross-sectional applications. More precisely, the approach of Brandt et al. (2009) proposes to parametrize the optimal portfolio weights as a function of stock characteristics directly, instead of estimating the stock's expected return, variance, and covariances with other stocks in a prior step. We chose weights as a function of the characteristics which maximize the expected utility of the investor. This approach is feasible for large portfolio dimensions (such as the entire CRSP universe) and has been proposed by Brandt et al. (2009). The review paper Brandt (2010) provides an excellent treatment of related portfolio choice methods.

11.1 Data preparation

To get started, we load the required packages alongside the monthly CRSP file, which forms our investment universe. We load the data from our `SQLite`-database introduced in our chapter on *"Accessing & managing financial data"*.

```
library(tidyverse)
library(lubridate)
library(RSQLite)
```

```
tidy_finance <- dbConnect(SQLite(),
  "data/tidy_finance.sqlite",
  extended_types = TRUE
)

crsp_monthly <- tbl(tidy_finance, "crsp_monthly") |>
  collect()
```

To evaluate the performance of portfolios, we further use monthly market returns as a benchmark to compute CAPM alphas.

```
factors_ff_monthly <- tbl(tidy_finance, "factors_ff_monthly") |>
  collect()
```

Next, we retrieve some stock characteristics that have been shown to have an effect on the expected returns or expected variances (or even higher moments) of the return distribution. In particular, we record the lagged one-year return momentum (`momentum_lag`), defined as the compounded return between months $t-13$ and $t-2$ for each firm. The second characteristic is the firm's market equity (`size_lag`), defined as the log of the price per share times the number of shares outstanding. To construct the correct lagged values, we use the approach introduced in the chapter on *"Accessing & managing financial data"*.

```
crsp_monthly_lags <- crsp_monthly |>
  transmute(permno,
    month_12 = month %m+% months(12),
    mktcap
  )

crsp_monthly <- crsp_monthly |>
  inner_join(crsp_monthly_lags,
    by = c("permno", "month" = "month_12"),
    suffix = c("", "_12")
  )

data_portfolios <- crsp_monthly |>
  mutate(
    momentum_lag = mktcap_lag / mktcap_12,
    size_lag = log(mktcap_lag)
  ) |>
  drop_na(contains("lag"))
```

11.2 Parametric portfolio policies

The basic idea of parametric portfolio weights is as follows. Suppose that at each date t we have N_t stocks in the investment universe, where each stock i has a return of $r_{i,t+1}$ and is associated with a vector of firm characteristics $x_{i,t}$ such as time-series momentum or the market capitalization. The investor's problem is to choose portfolio weights $w_{i,t}$ to maximize the expected utility of the portfolio return:

$$\max_w E_t\left(u(r_{p,t+1})\right) = E_t\left[u\left(\sum_{i=1}^{N_t} w_{i,t} r_{i,t+1}\right)\right]$$

where $u(\cdot)$ denotes the utility function.

Where do the stock characteristics show up? We parameterize the optimal portfolio weights as a function of the stock characteristic $x_{i,t}$ with the following linear specification for the portfolio weights:

$$w_{i,t} = \bar{w}_{i,t} + \frac{1}{N_t}\theta' \hat{x}_{i,t},$$

where $\bar{w}_{i,t}$ is a stock's weight in a benchmark portfolio (we use the value-weighted or naive portfolio in the application below), θ is a vector of coefficients which we are going to estimate, and $\hat{x}_{i,t}$ are the characteristics of stock i, cross-sectionally standardized to have zero mean and unit standard deviation.

Intuitively, the portfolio strategy is a form of active portfolio management relative to a performance benchmark. Deviations from the benchmark portfolio are derived from the individual stock characteristics. Note that by construction the weights sum up to one as $\sum_{i=1}^{N_t} \hat{x}_{i,t} = 0$ due to the standardization. Moreover, the coefficients are constant across assets and over time. The implicit assumption is that the characteristics fully capture all aspects of the joint distribution of returns that are relevant for forming optimal portfolios.

We first implement cross-sectional standardization for the entire CRSP universe. We also keep track of (lagged) relative market capitalization `relative_mktcap`, which will represent the value-weighted benchmark portfolio, while `n` denotes the number of traded assets N_t, which we use to construct the naive portfolio benchmark.

```
data_portfolios <- data_portfolios |>
  group_by(month) |>
  mutate(
    n = n(),
    relative_mktcap = mktcap_lag / sum(mktcap_lag),
    across(contains("lag"), ~ (. - mean(.)) / sd(.)),
  ) |>
  ungroup() |>
  select(-mktcap_lag, -altprc)
```

11.3 Computing portfolio weights

Next, we move to identify optimal choices of θ. We rewrite the optimization problem together with the weight parametrization and can then estimate θ to maximize the objective function based on our sample

$$E_t\left(u(r_{p,t+1})\right) = \frac{1}{T}\sum_{t=0}^{T-1} u\left(\sum_{i=1}^{N_t}\left(\bar{w}_{i,t} + \frac{1}{N_t}\theta' \hat{x}_{i,t}\right) r_{i,t+1}\right).$$

The allocation strategy is straightforward because the number of parameters to estimate is small. Instead of a tedious specification of the N_t dimensional vector of expected returns and the $N_t(N_t + 1)/2$ free elements of the covariance matrix, all we need to focus on in our application is the vector θ. θ contains only two elements in our application - the relative deviation from the benchmark due to *size* and *momentum*.

To get a feeling for the performance of such an allocation strategy, we start with an arbitrary initial vector θ_0. The next step is to choose θ optimally to maximize the objective function. We automatically detect the number of parameters by counting the number of columns with lagged values.

```
n_parameters <- sum(grepl(
  "lag",
  colnames(data_portfolios)
))
theta <- rep(1.5, n_parameters)
names(theta) <- colnames(data_portfolios)[grepl(
  "lag",
  colnames(data_portfolios)
)]
```

The function `compute_portfolio_weights` below computes the portfolio weights $\bar{w}_{i,t} + \frac{1}{N_t}\theta' \hat{x}_{i,t}$ according to our parametrization for a given value θ_0 of θ. Everything happens within a single pipeline, hence we provide a short walkthrough.

We first compute `characteristic_tilt`, the tilting values $\frac{1}{N_t}\theta' \hat{x}_{i,t}$ which resemble the deviation from the benchmark portfolio. Next, we compute the benchmark portfolio `weight_benchmark`, which can be any reasonable set of portfolio weights. In our case, we choose either the value or equal-weighted allocation. `weight_tilt` completes the picture and contains the final portfolio weights `weight_tilt = weight_benchmark + characteristic_tilt` which deviate from the benchmark portfolio depending on the stock characteristics.

The final few lines go a bit further and implement a simple version of a no-short sale constraint. While it is generally not straightforward to ensure portfolio weight constraints via the parameterization, we simply normalize the portfolio weights such that they are enforced to be positive. Finally, we make sure that the normalized weights sum up to one again. We do so by

$$w_{i,t}^+ = \frac{\max(0, w_{i,t})}{\sum_{j=1}^{N_t} \max(0, w_{i,t})}.$$

The following function computes the optimal portfolio weights in the way just described.

```
compute_portfolio_weights <- function(theta,
                                      data,
                                      value_weighting = TRUE,
                                      allow_short_selling = TRUE) {
  data |>
    group_by(month) |>
    bind_cols(
      characteristic_tilt = data |>
        transmute(across(contains("lag"), ~ . / n)) |>
        as.matrix() %*% theta |> as.numeric()
    ) |>
    mutate(
      # Definition of benchmark weight
      weight_benchmark = case_when(
        value_weighting == TRUE ~ relative_mktcap,
        value_weighting == FALSE ~ 1 / n
      ),
      # Parametric portfolio weights
      weight_tilt = weight_benchmark + characteristic_tilt,
      # Short-sell constraint
      weight_tilt = case_when(
        allow_short_selling == TRUE ~ weight_tilt,
        allow_short_selling == FALSE ~ pmax(0, weight_tilt)
      ),
      # Weights sum up to 1
      weight_tilt = weight_tilt / sum(weight_tilt)
    ) |>
    ungroup()
}
```

In the next step we compute the portfolio weights for the arbitrary vector θ_0. In the example below, we use the value-weighted portfolio as a benchmark and allow negative portfolio weights.

```
weights_crsp <- compute_portfolio_weights(theta,
  data_portfolios,
  value_weighting = TRUE,
  allow_short_selling = TRUE
)
```

11.4 Portfolio performance

Are the computed weights optimal in any way? Most likely not, as we picked θ_0 arbitrarily. To evaluate the performance of an allocation strategy, one can think of many different approaches. In their original paper, Brandt et al. (2009) focus on a simple evaluation of the hypothetical utility of an agent equipped with a power utility function $u_\gamma(r) = \frac{(1+r)^\gamma}{1-\gamma}$, where γ is the risk aversion factor.

```
power_utility <- function(r, gamma = 5) {
  (1 + r)^(1 - gamma) / (1 - gamma)
}
```

It should be noted, that Gehrig et al. (2020) warn that in the leading case of constant relative risk aversion (CRRA) strong assumptions on the properties of the returns, the variables used to implement the parametric portfolio policy and the parameter space are necessary to obtain a well defined optimization problem.

No doubt, there are many other ways to evaluate a portfolio. The function below provides a summary of all kinds of interesting measures that can be considered relevant. Do we need all these evaluation measures? It depends: The original paper Brandt et al. (2009) only cares about expected utility to choose θ. However, if you want to choose optimal values that achieve the highest performance while putting some constraints on your portfolio weights, it is helpful to have everything in one function.

```
evaluate_portfolio <- function(weights_crsp,
                               full_evaluation = TRUE) {
  evaluation <- weights_crsp |>
    group_by(month) |>
    summarize(
      return_tilt = weighted.mean(ret_excess, weight_tilt),
      return_benchmark = weighted.mean(ret_excess, weight_benchmark)
    ) |>
    pivot_longer(-month,
      values_to = "portfolio_return",
      names_to = "model"
    ) |>
    group_by(model) |>
    left_join(factors_ff_monthly, by = "month") |>
    summarize(tibble(
      "Expected utility" = mean(power_utility(portfolio_return)),
      "Average return" = 100 * mean(12 * portfolio_return),
      "SD return" = 100 * sqrt(12) * sd(portfolio_return),
      "Sharpe ratio" = mean(portfolio_return) / sd(portfolio_return),
```

```
      "CAPM alpha" = coefficients(lm(portfolio_return ~ mkt_excess))[1],
      "Market beta" = coefficients(lm(portfolio_return ~ mkt_excess))[2]
    )) |>
    mutate(model = gsub("return_", "", model)) |>
    pivot_longer(-model, names_to = "measure") |>
    pivot_wider(names_from = model, values_from = value)

  if (full_evaluation) {
    weight_evaluation <- weights_crsp |>
      select(month, contains("weight")) |>
      pivot_longer(-month, values_to = "weight", names_to = "model") |>
      group_by(model, month) |>
      transmute(tibble(
        "Absolute weight" = abs(weight),
        "Max. weight" = max(weight),
        "Min. weight" = min(weight),
        "Avg. sum of negative weights" = -sum(weight[weight < 0]),
        "Avg. fraction of negative weights" = sum(weight < 0) / n()
      )) |>
      group_by(model) |>
      summarize(across(-month, ~ 100 * mean(.))) |>
      mutate(model = gsub("weight_", "", model)) |>
      pivot_longer(-model, names_to = "measure") |>
      pivot_wider(names_from = model, values_from = value)
    evaluation <- bind_rows(evaluation, weight_evaluation)
  }
  return(evaluation)
}
```

Let us take a look at the different portfolio strategies and evaluation measures.

```
evaluate_portfolio(weights_crsp) |> print(n = Inf)
```

```
## # A tibble: 11 x 3
##    measure                           benchmark     tilt
##    <chr>                                 <dbl>    <dbl>
##  1 Expected utility                   -2.49e-1 -0.262
##  2 Average return                      6.86e+0 -0.604
##  3 SD return                           1.53e+1 21.0
##  4 Sharpe ratio                        1.29e-1 -0.00831
##  5 CAPM alpha                          1.08e-4 -0.00574
##  6 Market beta                         9.92e-1  0.927
##  7 Absolute weight                     2.46e-2  0.0631
##  8 Max. weight                         3.52e+0  3.65
```

```
##  9 Min. weight                          2.78e-5 -0.145
## 10 Avg. sum of negative weights         0       78.0
## 11 Avg. fraction of negative weights    0       49.4
```

The value-weighted portfolio delivers an annualized return of more than 6 percent and clearly outperforms the tilted portfolio, irrespective of whether we evaluate expected utility, the Sharpe ratio or the CAPM alpha. We can conclude the market beta is close to one for both strategies (naturally almost identically 1 for the value-weighted benchmark portfolio). When it comes to the distribution of the portfolio weights, we see that the benchmark portfolio weight takes less extreme positions (lower average absolute weights and lower maximum weight). By definition, the value-weighted benchmark does not take any negative positions, while the tilted portfolio also takes short positions.

11.5 Optimal parameter choice

Next, we move to a choice of θ that actually aims to improve some (or all) of the performance measures. We first define a helper function `compute_objective_function`, which we then pass to an optimizer.

```
compute_objective_function <- function(theta,
                                       data,
                                       objective_measure = "Expected utility",
                                       value_weighting = TRUE,
                                       allow_short_selling = TRUE) {
  processed_data <- compute_portfolio_weights(
    theta,
    data,
    value_weighting,
    allow_short_selling
  )

  objective_function <- evaluate_portfolio(processed_data,
    full_evaluation = FALSE
  ) |>
    filter(measure == objective_measure) |>
    pull(tilt)

  return(-objective_function)
}
```

You may wonder why we return the negative value of the objective function. This

is simply due to the common convention for optimization procedures to search for minima as a default. By minimizing the negative value of the objective function, we get the maximum value as a result. In its most basic form, R optimization relies on the function `optim`. As main inputs, the function requires an initial guess of the parameters and the objective function to minimize. Now, we are fully equipped to compute the optimal values of $\hat{\theta}$, which maximize the hypothetical expected utility of the investor.

```
optimal_theta <- optim(
  par = theta,
  compute_objective_function,
  objective_measure = "Expected utility",
  data = data_portfolios,
  value_weighting = TRUE,
  allow_short_selling = TRUE
)

optimal_theta$par
```

```
## momentum_lag     size_lag
##        0.189       -2.007
```

The resulting values of $\hat{\theta}$ are easy to interpret intuitively. Expected utility increases by tilting weights from the value-weighted portfolio towards smaller stocks (negative coefficient for size) and towards past winners (positive value for momentum).

11.6 More model specifications

How does the portfolio perform for different model specifications? For this purpose, we compute the performance of a number of different modeling choices based on the entire CRSP sample. The next code chunk performs all the heavy lifting.

```
full_model_grid <- expand_grid(
  value_weighting = c(TRUE, FALSE),
  allow_short_selling = c(TRUE, FALSE),
  data = list(data_portfolios)
) |>
  mutate(optimal_theta = pmap(
    .l = list(
      data,
```

```
      value_weighting,
      allow_short_selling
    ),
    .f = ~ optim(
      par = theta,
      compute_objective_function,
      data = ..1,
      objective_measure = "Expected utility",
      value_weighting = ..2,
      allow_short_selling = ..3
    )$par
  ))
```

Finally, we can compare the results. The table below shows summary statistics for all possible combinations: equal- or value-weighted benchmark portfolio, with or without short-selling constraints, and tilted towards maximizing expected utility.

```
performance_table <- full_model_grid |>
  mutate(
    processed_data = pmap(
      .l = list(
        optimal_theta,
        data,
        value_weighting,
        allow_short_selling
      ),
      .f = ~ compute_portfolio_weights(..1, ..2, ..3, ..4)
    ),
    portfolio_evaluation = map(processed_data,
      evaluate_portfolio,
      full_evaluation = TRUE
    )
  ) |>
  select(
    value_weighting,
    allow_short_selling,
    portfolio_evaluation
  ) |>
  unnest(portfolio_evaluation)

performance_table |>
  rename(
    " " = benchmark,
    Optimal = tilt
  ) |>
```

```
  mutate(
    value_weighting = case_when(
      value_weighting == TRUE ~ "VW",
      value_weighting == FALSE ~ "EW"
    ),
    allow_short_selling = case_when(
      allow_short_selling == TRUE ~ "",
      allow_short_selling == FALSE ~ "(no s.)"
    )
  ) |>
  pivot_wider(
    names_from = value_weighting:allow_short_selling,
    values_from = " ":Optimal,
    names_glue = "{value_weighting} {allow_short_selling} {.value} "
  ) |>
  select(
    measure,
    `EW    `,
    `VW    `,
    sort(contains("Optimal"))
  ) |>
  print(n = 11, max_extra_cols = 7)
```

```
## # A tibble: 11 x 7
##    measure             `EW    ` `VW    ` `VW  Optimal `
##    <chr>                  <dbl>    <dbl>          <dbl>
##  1 Expected utility    -0.250   -2.49e-1       -0.247
##  2 Average return      10.5      6.86e+0       14.7
##  3 SD return           20.3      1.53e+1       20.6
##  4 Sharpe ratio         0.149    1.29e-1        0.206
##  5 CAPM alpha           0.00231  1.08e-4        0.00649
##  6 Market beta          1.13     9.92e-1        1.01
##  7 Absolute weight      0.0246   2.46e-2        0.0379
##  8 Max. weight          0.0246   3.52e+0        3.34
##  9 Min. weight          0.0246   2.78e-5       -0.0327
## 10 Avg. sum of negati~  0        0             27.9
## 11 Avg. fraction of n~  0        0             38.8
## # ... with 3 more variables:
## #   `VW (no s.) Optimal ` <dbl>, `EW  Optimal ` <dbl>,
## #   `EW (no s.) Optimal ` <dbl>
```

The results indicate that the average annualized Sharpe ratio of the equal-weighted portfolio exceeds the Sharpe ratio of the value-weighted benchmark portfolio. Nevertheless, starting with the weighted value portfolio as a benchmark and tilting optimally with respect to momentum and small stocks yields the highest Sharpe ratio

across all specifications. Imposing no short-sale constraints does not improve the performance of the portfolios in our application.

11.7 Exercises

1. How do the estimated parameters $\hat{\theta}$ and the portfolio performance change if your objective is to maximize the Sharpe ratio instead of the hypothetical expected utility?
2. The code above is very flexible in the sense that you can easily add new firm characteristics. Construct a new characteristic and evaluate the corresponding coefficient $\hat{\theta}_i$.
3. Tweak the function `optimal_theta`such that you can impose additional performance constraints in order to determine $\hat{\theta}$ which maximizes expected utility under the constraint that the market beta is below 1.
4. Does the portfolio performance resemble a realistic out-of-sample backtesting procedure? Verify the robustness of the results by first estimating $\hat{\theta}$ based on *past data* only. Then, use more recent periods to evaluate the actual portfolio performance.
5. By formulating the portfolio problem as a statistical estimation problem, you can easily obtain standard errors for the coefficients of the weight function. Brandt et al. (2009) provide the relevant derivations in their paper in Equation (10). Implement a small function that computes standard errors for $\hat{\theta}$.

12

Constrained optimization and backtesting

In this chapter, we conduct portfolio backtesting in a more realistic setting by including transaction costs and investment constraints such as no-short-selling rules. We start with *standard* mean-variance efficient portfolios. Then, we introduce further constraints step-by-step. Numerical constrained optimization is performed by the packages `quadprog` (for quadratic objective functions such as in typical mean-variance framework) and `alabama` (for more general, non-linear objectives and constraints).

```
library(tidyverse)
library(RSQLite)
library(quadprog)
library(alabama)
```

12.1 Data preparation

We start by loading the required data from our `SQLite`-database introduced in our chapter on *"Accessing & managing financial data"*. For simplicity, we restrict our investment universe to the monthly Fama-French industry portfolio returns in the following application.

```
tidy_finance <- dbConnect(SQLite(), "data/tidy_finance.sqlite",
  extended_types = TRUE
)

industry_returns <- tbl(tidy_finance, "industries_ff_monthly") |>
  collect()

industry_returns <- industry_returns |>
  select(-month)
```

12.2 Recap of portfolio choice

A common objective for portfolio optimization is to find mean-variance efficient portfolio weights, i.e., the allocation which delivers the lowest possible return variance for a given minimum level of expected returns. In the most extreme case, where the investor is only concerned about portfolio variance, she may choose to implement the minimum variance portfolio (MVP) weights which are given by the solution to

$$w_{\text{mvp}} = \arg\min w'\Sigma w \text{ s.t. } w'\iota = 1$$

where Σ is the $(N \times N)$ covariance matrix of the returns. The optimal weights ω_{mvp} can be found analytically and are $\omega_{\text{mvp}} = \frac{\Sigma^{-1}\iota}{\iota'\Sigma^{-1}\iota}$. In terms of code, the math is equivalent to the following.

```
Sigma <- cov(industry_returns)
w_mvp <- solve(Sigma) %*% rep(1, ncol(Sigma))
w_mvp <- as.vector(w_mvp / sum(w_mvp))
```

Next, consider an investor who aims to achieve minimum variance *given a required expected portfolio return* $\bar{\mu}$ such that she chooses

$$w_{\text{eff}}(\bar{\mu}) = \arg\min w'\Sigma w \text{ s.t. } w'\iota = 1 \text{ and } \omega'\mu \geq \bar{\mu}.$$

It can be shown (see Exercises) that the portfolio choice problem can equivalently be formulated for an investor with mean-variance preferences and risk aversion factor γ. The investor aims to choose portfolio weights such that

$$w^*_\gamma = \arg\max w'\mu - \frac{\gamma}{2}w'\Sigma w \quad s.t. w'\iota = 1.$$

The solution to the optimal portfolio choice problem is:

$$\omega^*_\gamma = \frac{1}{\gamma}\left(\Sigma^{-1} - \frac{1}{\iota'\Sigma^{-1}\iota}\Sigma^{-1}\iota\iota'\Sigma^{-1}\right)\mu + \frac{1}{\iota'\Sigma^{-1}\iota}\Sigma^{-1}\iota.$$

Empirically, this classical solution imposes many problems. In particular, the estimates of μ_t are noisy over short horizons, the $(N \times N)$ matrix Σ_t contains $N(N-1)/2$ distinct elements and thus, estimation error is huge. Even worse, if the asset universe contains more assets than available time periods $(N > T)$, the sample covariance matrix is no longer positive definite such that the inverse Σ^{-1} does not exist anymore. On top of the estimation uncertainty, *transaction costs* are a major concern. Rebalancing portfolios is costly, and, therefore, the optimal choice should depend on the investor's current holdings.

12.3 Estimation uncertainty and transaction costs

The empirical evidence regarding the performance of a mean-variance optimization procedure in which you simply plug in some sample estimates $\hat{\mu}_t$ and $\hat{\Sigma}_t$ can be summarized rather briefly: mean-variance optimization performs poorly! The literature discusses many proposals to overcome these empirical issues. For instance, one may impose some form of regularization of Σ, rely on Bayesian priors inspired by theoretical asset pricing models, or use high-frequency data to improve forecasting. One unifying framework that works easily, effectively (even for large dimensions), and is purely inspired by economic arguments is an ex-ante adjustment for transaction costs (Hautsch and Voigt, 2019).

Assume that returns are from a multivariate normal distribution such that $p_t(r_{t+1}|\mathcal{M}) = N(\mu, \Sigma)$. Additionally, we assume quadratic transaction costs which penalize rebalancing such that

$$\nu\left(\omega_{t+1}, \omega_{t^+}, \beta\right) := \nu_t\left(\omega_{t+1}, \beta\right) = \frac{\beta}{2}\left(\omega_{t+1} - \omega_{t^+}\right)'\left(\omega_{t+1} - \omega_{t^+}\right),$$

with cost parameter $\beta > 0$ and $\omega_{t^+} := \omega_t \circ (1 + r_t)/\iota'(\omega_t \circ (1 + r_t))$. Note that ω_{t^+} differs mechanically from ω_t due to the returns in the past period.
Then, the optimal portfolio choice is

$$\begin{aligned}\omega_{t+1}^* &:= \arg\max_{\omega \in \mathbb{R}^N, \iota'\omega = 1} \omega'\mu - \nu_t(\omega, \omega_{t^+}, \beta) - \frac{\gamma}{2}\omega'\Sigma\omega \\ &= \arg\max_{\omega \in \mathbb{R}^N, \iota'\omega = 1} \omega'\mu^* - \frac{\gamma}{2}\omega'\Sigma^*\omega,\end{aligned}$$

where

$$\mu^* := \mu + \beta\omega_{t^+} \quad \text{and} \quad \Sigma^* := \Sigma + \frac{\beta}{\gamma} I_N.$$

As a result, adjusting for transaction costs implies a standard mean-variance optimal portfolio choice with adjusted return parameters Σ^* and μ^*:

$$\omega_{t+1}^* = \frac{1}{\gamma}\left(\Sigma^{*-1} - \frac{1}{\iota'\Sigma^{*-1}\iota}\Sigma^{*-1}\iota\iota'\Sigma^{*-1}\right)\mu^* + \frac{1}{\iota'\Sigma^{*-1}\iota}\Sigma^{*-1}\iota.$$

An alternative formulation of the optimal portfolio can be derived as follows:

$$\omega_{t+1}^* = \arg\max_{\omega \in \mathbb{R}^N, \iota'\omega = 1} \omega'\left(\mu + \beta\left(\omega_{t^+} - \frac{1}{N}\iota\right)\right) - \frac{\gamma}{2}\omega'\Sigma^*\omega.$$

The optimal weights correspond to a mean-variance portfolio where the vector of expected returns is such that assets that currently exhibit a higher weight are considered as delivering a higher expected return.

12.4 Optimal portfolio choice

The function below implements the efficient portfolio weight in its general form, allowing for transaction costs (conditional on the holdings *before* reallocation). For $\beta = 0$, the computation resembles the standard mean-variance efficient framework. `gamma` denotes the coefficient of risk aversion,`beta` is the transaction cost parameter and `w_prev` are the weights before rebalancing.

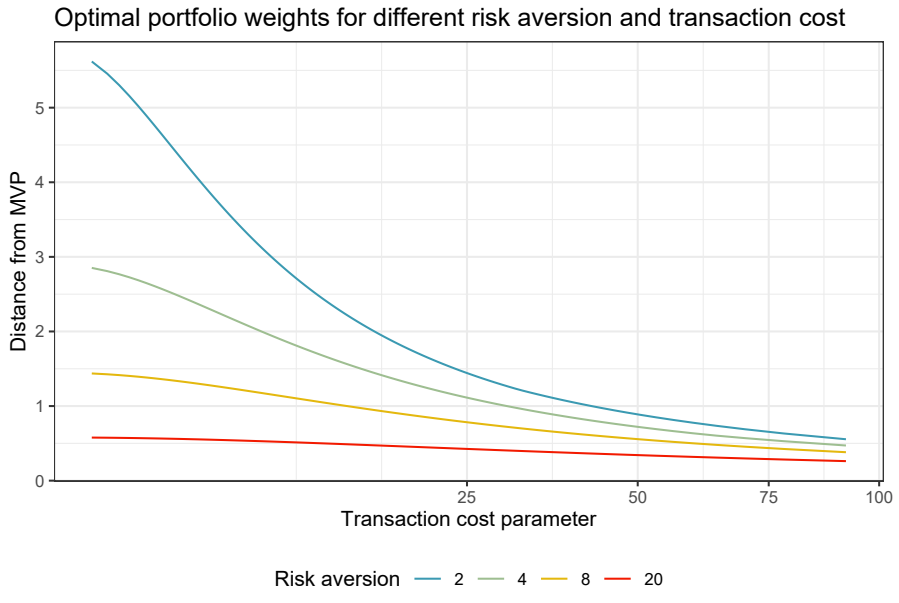
```
compute_efficient_weight <- function(Sigma,
                                     mu,
                                     gamma = 2,
                                     beta = 0, # transaction costs
                                     w_prev = 1 / ncol(Sigma) * rep(1, ncol(Sigma))) {
  iota <- rep(1, ncol(Sigma))
  Sigma_processed <- Sigma + beta / gamma * diag(ncol(Sigma))
  mu_processed <- mu + beta * w_prev

  Sigma_inverse <- solve(Sigma_processed)

  w_mvp <- Sigma_inverse %*% iota
  w_mvp <- as.vector(w_mvp / sum(w_mvp))
  w_opt <- w_mvp + 1 / gamma *
    (Sigma_inverse - 1 / sum(Sigma_inverse) * Sigma_inverse %*% iota %*% t(iota) %*% Sigma_inverse) %*%
      mu_processed
  return(as.vector(w_opt))
}

mu <- colMeans(industry_returns)
compute_efficient_weight(Sigma, mu)
```

```
##  [1]  1.428  0.270 -1.302  0.375  0.308 -0.152  0.544
##  [8]  0.472 -0.167 -0.776
```

What is the effect of transaction costs or different levels of risk aversion on the optimal portfolio choice? The following few lines of code analyze the distance between the MVP and the portfolio implemented by the investor for different values of the transaction cost parameter β and risk aversion γ.

```
transaction_costs <- expand_grid(
  gamma = c(2, 4, 8, 20),
  beta = 20 * qexp((1:99) / 100)
) |>
```

```
  mutate(
    weights = map2(
      .x = gamma,
      .y = beta,
      ~ compute_efficient_weight(Sigma,
        mu,
        gamma = .x,
        beta = .y / 10000,
        w_prev = w_mvp
      )
    ),
    concentration = map_dbl(weights, ~ sum(abs(. - w_mvp)))
  )

transaction_costs |>
  mutate(`Risk aversion` = as_factor(gamma)) |>
  ggplot(aes(x = beta, y = concentration, color = `Risk aversion`)) +
  geom_line() +
  scale_x_sqrt() +
  labs(
    x = "Transaction cost parameter",
    y = "Distance from MVP",
    title = "Optimal portfolio weights for different risk aversion and transaction cost"
  )
```



The figure shows that the initial portfolio is always the (sample) MVP and that the higher the transaction costs parameter β, the smaller is the rebalancing from the initial portfolio (which we always set to the MVP weights in this example). In addition, if risk aversion γ increases, the efficient portfolio is closer to the MVP weights such that the investor desires less rebalancing from the initial holdings.

12.5 Constrained optimization

Next, we introduce constraints to the above optimization procedure. Very often, typical constraints such as short-selling restrictions prevent analytical solutions for optimal portfolio weights. However, numerical optimization allows computing the solutions to such constrained problems. For the purpose of mean-variance optimization, we rely on the `solve.QP()` function from the package `quadprog`.

The function `solve.QP()` delivers numerical solutions to quadratic programming problems of the form $\min(-\mu\omega + 1/2\omega'\Sigma\omega)$ s.t. $A'\omega >= b_0$.

The function takes one argument (`meq`) for the number of equality constraints. Therefore, the above matrix A is simply a vector of ones to ensure that the weights sum up to one. In the case of short-selling constraints, the matrix A is of the form

$$A = \begin{pmatrix} 1 & 1 & ... & 1 \\ 1 & 0 & ... & 0 \\ 0 & 1 & ... & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & ... & 1 \end{pmatrix} \qquad b_0 = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

Before we dive into unconstrained optimization, we revisit the *unconstrained* problem and replicate the analytical solutions for the minimum variance and efficient portfolio weights from above. We verify that the output is equal to the above solution. Note that we round to the first six digits to avoid differences at higher digits that might arise due to inherent imprecision of numerical estimation procedures. As just discussed, we set `Amat` to a matrix with a column of ones and `bvec` to 1 to enforce the constraint that weights must sum up to one. `meq=1` means that one (out of one) constraints must be satisfied with equality.

```
n_industries <- ncol(industry_returns)

w_mvp_numerical <- solve.QP(
  Dmat = Sigma,
  dvec = rep(0, n_industries),
  Amat = cbind(rep(1, n_industries)),
  bvec = 1,
```

```
  meq = 1
)

all(round(w_mvp, 6) == round(w_mvp_numerical$solution, 6))
```

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

```
w_efficient_numerical <- solve.QP(
  Dmat = 2 * Sigma,
  dvec = mu,
  Amat = cbind(rep(1, n_industries)),
  bvec = 1,
  meq = 1
)

all(round(compute_efficient_weight(Sigma, mu), 6) == round(w_efficient_numerical$solution, 6))
```

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

For more complex optimization routines, this optimization task view[1] provides an overview of the wast optimization landscape in R.

Next, we approach problems where no analytical solutions exist. First, we additionally impose short-sale constraints, which implies N inequality constraints if the form $w_i >= 0$.

```
w_no_short_sale <- solve.QP(
  Dmat = 2 * Sigma,
  dvec = mu,
  Amat = cbind(1, diag(n_industries)),
  bvec = c(1, rep(0, n_industries)),
  meq = 1
)
w_no_short_sale$solution
```

```
##  [1]  6.34e-01 -1.91e-17  1.20e-16 -3.47e-18  6.78e-18
##  [6] -6.24e-17  1.02e-01  2.64e-01  3.38e-22 -2.22e-16
```

`solve.QP` is fast because it benefits from a very clear structure with a quadratic objective and linear constraints. However, optimization typically requires more flexibility. As an example, we show how to compute optimal weights, subject to the so-called regulation T-constraint[2], which requires that the sum of all absolute portfolio

[1] https://cran.r-project.org/web/views/Optimization.html
[2] https://en.wikipedia.org/wiki/Regulation_T

weights is smaller than 1.5. The constraint implies an initial margin requirement of 50% and, therefore, also a non-linear objective function. Thus, we can no longer rely on `solve.QP()`. Instead, we rely on the package `alabama`, which requires a separate definition of objective and constraint functions.

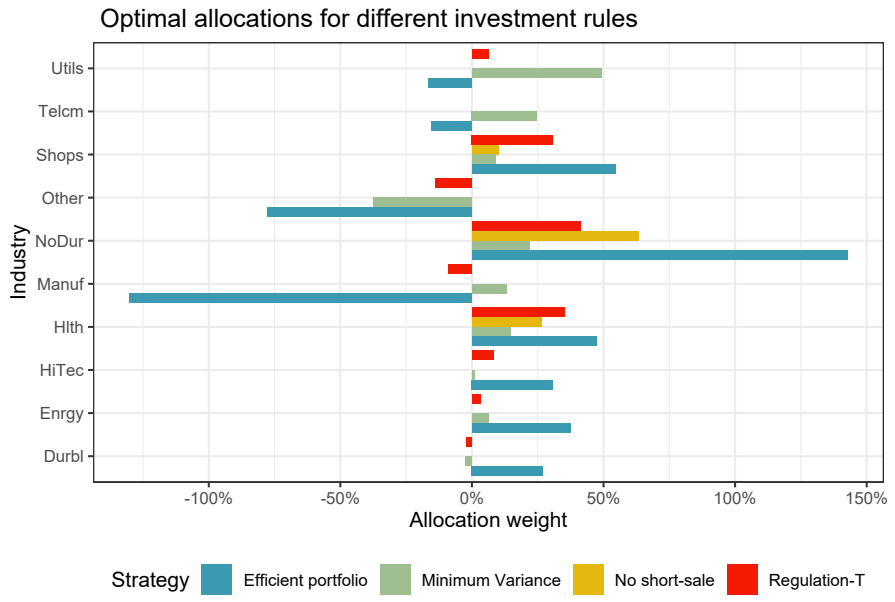
```
initial_weights <- 1 / n_industries * rep(1, n_industries)
objective <- function(w, gamma = 2) -t(w) %*% (1 + mu) + gamma / 2 * t(w) %*% Sigma %*% w
inequality_constraints <- function(w, reg_t = 1.5) {
  return(reg_t - sum(abs(w)))
}
equality_constraints <- function(w) {
  return(sum(w) - 1)
}

w_reg_t <- constrOptim.nl(
  par = initial_weights,
  hin = inequality_constraints,
  fn = objective,
  heq = equality_constraints,
  control.outer = list(trace = FALSE)
)
w_reg_t$par
```

```
##  [1]  4.11e-01 -2.07e-02 -9.00e-02  3.37e-02  8.03e-02
##  [6] -2.25e-08  3.08e-01  3.52e-01  6.25e-02 -1.37e-01
```

The figure below shows the optimal allocation weights across all 10 industries for the four different strategies considered so far: minimum variance, efficient portfolio with $\gamma = 2$, efficient portfolio with short-sale constraints, and the Regulation-T constrained portfolio.

```
tibble(
  `No short-sale` = w_no_short_sale$solution,
  `Minimum Variance` = w_mvp,
  `Efficient portfolio` = compute_efficient_weight(Sigma, mu),
  `Regulation-T` = w_reg_t$par,
  Industry = colnames(industry_returns)
) |>
  pivot_longer(-Industry,
    names_to = "Strategy"
  ) |>
  ggplot(aes(
    fill = Strategy,
    y = value,
```

```
    x = Industry
)) +
geom_bar(position = "dodge", stat = "identity") +
coord_flip() +
labs(
  y = "Allocation weight",
  title = " Optimal allocations for different investment rules"
) +
scale_y_continuous(labels = scales::percent)
```



Before we move on, we want to propose a final allocation strategy, which reflects a somewhat more realistic structure of transaction costs instead of the quadratic specification used above. The function below computes efficient portfolio weights while adjusting for L_1 transaction costs $\beta \sum_{i=1}^{N} |(w_{i,t+1} - w_{i,t^+})|$. No closed-form solution exists, and we rely on non-linear optimization procedures.

```
compute_efficient_weight_L1_TC <- function(mu,
                                           Sigma,
                                           gamma = 2,
                                           beta = 0,
                                           initial_weights = 1 / ncol(sigma) * rep(1, ncol(sigma))) {
  objective <- function(w) -t(w) %*% mu + gamma / 2 * t(w) %*% Sigma %*% w + (beta / 10000) / 2 * sum(abs(w - in
```

```
  w_optimal <- constrOptim.nl(
    par = initial_weights,
    fn = objective,
    heq = function(w) {
      sum(w) - 1
    },
    control.outer = list(trace = FALSE)
  )

  w_optimal$par
}
```

12.6 Out-of-sample backtesting

For the sake of simplicity, we committed one fundamental error in computing portfolio weights above. We used the full sample of the data to determine the optimal allocation. To implement this strategy at the beginning of the 2000s, you will need to know how the returns will evolve until 2020. While interesting from a methodological point of view, we cannot evaluate the performance of the portfolios in a reasonable out-of-sample fashion. We do so next in a backtesting application for three strategies. For the backtest, we recompute optimal weights just based on past available data.

```
window_length <- 120
periods <- nrow(industry_returns) - window_length

beta <- 50
gamma <- 2

performance_values <- matrix(NA,
  nrow = periods,
  ncol = 3
) # A matrix to collect all returns
colnames(performance_values) <- c("raw_return", "turnover", "net_return")

performance_values <- list(
  "MV (TC)" = performance_values,
  "Naive" = performance_values,
  "MV" = performance_values
)
```

```
w_prev_1 <- w_prev_2 <- w_prev_3 <- rep(
  1 / n_industries,
  n_industries
)
```

We also define two helper functions: one to adjust the weights due to returns and one for performance evaluation, where we compute realized returns net of transaction costs.

```
adjust_weights <- function(w, next_return) {
  w_prev <- 1 + w * next_return
  as.numeric(w_prev / sum(as.vector(w_prev)))
}

evaluate_performance <- function(w, w_previous, next_return, beta = 50) {
  raw_return <- as.matrix(next_return) %*% w
  turnover <- sum(abs(w - w_previous))
  net_return <- raw_return - beta / 10000 * turnover
  c(raw_return, turnover, net_return)
}
```

The lines above define the general setup. We consider 120 periods from the past to update the parameter estimates before recomputing portfolio weights. Then, we update portfolio weights which is costly and affects the performance. The portfolio weights determine the portfolio return. A period later, the current portfolio weights have changed and form the foundation for transaction costs incurred in the next period. We consider three different competing strategies: the mean-variance efficient portfolio, the mean-variance efficient portfolio with ex-ante adjustment for transaction costs, and the naive portfolio, which allocates wealth equally across the different assets.

The following code chunk performs rolling-window estimation. In each period, the estimation window contains the returns available up to the current period. Note that we use the sample moments, but you might use more advanced estimators in practice.

```
for (p in 1:periods) {
  returns_window <- industry_returns[p:(p + window_length - 1), ]
  next_return <- industry_returns[p + window_length, ] |> as.matrix()

  Sigma <- cov(returns_window)
  mu <- 0 * colMeans(returns_window)

  # Transaction-cost adjusted portfolio
  w_1 <- compute_efficient_weight_L1_TC(
```

```
    mu = mu,
    Sigma = Sigma,
    beta = beta,
    gamma = gamma,
    initial_weights = w_prev_1
  )

  performance_values[[1]][p, ] <- evaluate_performance(w_1,
    w_prev_1,
    next_return,
    beta = beta
  )

  w_prev_1 <- adjust_weights(w_1, next_return)

  # Naive portfolio
  w_2 <- rep(1 / n_industries, n_industries)

  performance_values[[2]][p, ] <- evaluate_performance(
    w_2,
    w_prev_2,
    next_return
  )

  w_prev_2 <- adjust_weights(w_2, next_return)

  # Mean-variance efficient portfolio (w/o transaction costs)
  w_3 <- compute_efficient_weight(
    Sigma = Sigma,
    mu = mu,
    gamma = gamma
  )

  performance_values[[3]][p, ] <- evaluate_performance(
    w_3,
    w_prev_3,
    next_return
  )

  w_prev_3 <- adjust_weights(w_3, next_return)
}
```

Finally, we get to the evaluation of the portfolio strategies *net-of-transaction costs*. Note that we compute annualized returns and standard deviations.

```
performance <- lapply(performance_values, as_tibble) |>
  bind_rows(.id = "strategy")

performance |>
  group_by(strategy) |>
  summarize(
    Mean = 12 * mean(100 * net_return),
    SD = sqrt(12) * sd(100 * net_return),
    `Sharpe ratio` = if_else(Mean > 0,
      Mean / SD,
      NA_real_
    ),
    Turnover = 100 * mean(turnover)
  )
```

```
## # A tibble: 3 x 5
##   strategy   Mean    SD `Sharpe ratio` Turnover
##   <chr>     <dbl> <dbl>          <dbl>    <dbl>
## 1 MV       -0.637  12.4         NA     214.
## 2 MV (TC)  12.1    15.1          0.802   0.0311
## 3 Naive    12.1    15.1          0.801   0.229
```

The results clearly speak against mean-variance optimization. Turnover is huge when the investor only considers her portfolio's expected return and variance. Effectively, the mean-variance portfolio generates a *negative* annualized return after adjusting for transaction costs. At the same time, the naive portfolio turns out to perform very well. In fact, the performance gains of the transaction-cost adjusted mean-variance portfolio are small. The out-of-sample Sharpe ratio is slightly higher than for the naive portfolio. Note the extreme effect of turnover penalization on turnover: *MV (TC)* effectively resembles a buy-and-hold strategy which only updates the portfolio once the estimated parameters $\hat{\mu}_t$ and $\hat{\Sigma}_t$indicate that the current allocation is too far away from the optimal theoretical portfolio.

12.7 Exercises

1. We argue that an investor with a quadratic utility function with certainty equivalent

$$\max_w CE(w) = \omega'\mu - \frac{\gamma}{2}\omega'\Sigma\omega \text{ s.t. } \iota'\omega = 1$$

faces an equivalent optimization problem to a framework where portfolio weights are chosen with the aim to minimize volatility given a pre-

specified level or expected returns,

$$\min_{w} \omega' \Sigma \omega \text{ s.t. } \omega' \mu = \bar{\mu} \text{ and } \iota' \omega = 1.$$

Proof that there is an equivalence between the optimal portfolio weights in both cases.

2. Consider the portfolio choice problem for transaction-cost adjusted certainty equivalent maximization with risk aversion parameter γ

$$\omega_{t+1}^* := \arg \max_{\omega \in \mathbb{R}^N, \iota' \omega = 1} \omega' \mu - \nu_t(\omega, \beta) - \frac{\gamma}{2} \omega' \Sigma \omega$$

where Σ and μ are (estimators of) the variance-covariance matrix of the returns and the vector of expected returns. Assume for now that transaction costs are quadratic in rebalancing **and** proportional to stock illiquidity such that

$$\nu_t(\omega, \times) := \frac{\beta}{2} (\omega - \omega_{t^+})' B (\omega - \omega_{t^+})$$

where $B = \text{diag}(ill_1, \dots, ill_N)$ is a diagonal matrix where $ill_1, \dots, ill_N$. Derive a closed-form solution for the mean-variance efficient portfolio ω_{t+1}^* based on the transaction cost specification above. Discuss the effect of illiquidity ill_i on the individual portfolio weights relative to an investor that myopically ignores transaction costs in her decision.

3. Use the solution from the previous exercise to update the function `compute_efficient_weight` such that you can compute optimal weights conditional on a matrix B with illiquidity measures.
4. Illustrate the evolution of the *optimal* weights from the naive portfolio to the efficient portfolio in the mean-standard deviation diagram.
5. Is it always optimal to choose the same β in the optimization problem than the value used in evaluating the portfolio performance? In other words: Can it be optimal to choose theoretically sub-optimal portfolios based on transaction cost considerations that do not reflect the actual incurred costs? Evaluate the out-of-sample Sharpe ratio after transaction costs for a range of different values of imposed β values.

A

Cover design

```
library(tidyverse)
library(RSQLite)
library(wesanderson)

tidy_finance <- dbConnect(SQLite(), "data/tidy_finance.sqlite", extended_types = TRUE)
mfac <- tbl(tidy_finance, "factors_ff_daily") %>%
  collect()

cp <- coord_polar(direction = -1, clip = "on")
cp$is_free <- function() TRUE

plot_data <- mfac %>%
  select(date, mkt_excess) %>%
  group_by(year = lubridate::floor_date(date, "year")) %>%
  mutate(group_id = cur_group_id())

plot_data <- plot_data %>%
  mutate(
    group_id = if_else(group_id >= 28, group_id + 4, group_id + 0),
    group_id = if_else(group_id >= 36, group_id + 4, group_id + 0),
    group_id = if_else(group_id >= 44, group_id + 4, group_id + 0)
  ) %>%
  bind_rows(plot_data %>%
    filter(group_id %in% c(28:31, 36:39, 44:47)) %>%
    mutate(mkt_excess = NA)) %>%
  group_by(group_id) %>%
  mutate(
    day = 2 * pi * (1:n()) / 252,
    ymin = pmin(1 + mkt_excess, 1),
    ymax = pmax(1 + mkt_excess, 1),
    vola = sd(mkt_excess)
  ) %>%
  filter(year >= "1961-01-01")

colors <- wes_palette("Zissou1", n_groups(plot_data), type = "continuous")
```

```
levels <- plot_data %>%
  distinct(group_id, vola) %>%
  arrange(vola) %>%
  pull(vola)

plot <- plot_data %>%
  mutate(vola = factor(vola, levels = levels)) %>%
  ggplot() +
  aes(x = day, y = mkt_excess, group = group_id, fill = vola) +
  cp +
  geom_ribbon(aes(
    ymin = ymin,
    ymax = ymax,
    fill = as_factor(vola)
  ), alpha = 0.90) +
  theme_void() +
  facet_wrap(~group_id, ncol = 8, scales = "free") +
  theme(
    strip.text.x = element_blank(),
    legend.position = "None",
    panel.spacing = unit(-5, "lines")
  ) +
  scale_fill_manual(values = colors)

# ggsave(
#   plot = plot, width = 8, height = 9,
#   filename = "cover.jpg", bg = "white"
# )
```

Bibliography

Bali, T. G., Engle, R. F., and Murray, S. (2016). *Empirical Asset Pricing: The Cross Section of Stock Returns*. John Wiley & Sons, Inc., Hoboken, New Jersey.

Black, F. and Scholes, M. (1973). The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654.

Brandt, M. W. (2010). Portfolio choice problems. In Ait-Sahalia, Y. and Hansen, L. P., editors, *Handbook of Financial Econometrics: Tools and Techniques*, volume 1 of *Handbooks in Finance*, pages 269–336. North-Holland, San Diego.

Brandt, M. W., Santa-Clara, P., and Valkanov, R. (2009). Parametric portfolio policies: Exploiting characteristics in the cross-section of equity returns. *The Review of Financial Studies*, 22(9):3411–3447.

Chen, H.-Y., Lee, A. C., and Lee, C.-F. (2015). Alternative errors-in-variables models and their applications in finance research. *The Quarterly Review of Economics and Finance*, 58:213–227.

Cochrane, J. H. (2009). *Asset pricing: Revised edition*. Princeton university press.

Cochrane, J. H. (2011). Presidential address: Discount rates. *The Journal of Finance*, 66(4):1047–1108.

Coqueret, G. and Guida, T. (2020). *Machine Learning for Factor Investing: R Version*. Chapman and Hall/CRC.

De Prado, M. L. (2018). *Advances in financial machine learning*. John Wiley & Sons.

Dixon, M. F., Halperin, I., and Bilokon, P. (2020). *Machine learning in Finance*. Springer.

Easley, D., López de Prado, M., O'Hara, M., and Zhang, Z. (2020). Microstructure in the Machine Age. *The Review of Financial Studies*, 34(7):3316–3363.

Fama, E. F. and French, K. R. (1992). The cross-section of expected stock returns. *The Journal of Finance*, 47(2):427–465.

Fama, E. F. and French, K. R. (1993). Common risk factors in the returns on stocks and bonds. *Journal of Financial Economics*, 33(1):3–56.

Fama, E. F. and MacBeth, J. D. (1973). Risk, return, and equilibrium: Empirical tests. *Journal of Political Economy*, 81(3):607–636.

Frazzini, A. and Pedersen, L. H. (2014). Betting against beta. *Journal of Financial Economics*, 111(1):1–25.

Gagliardini, P., Ossola, E., and Scaillet, O. (2016). Time-varying risk premium in large cross-sectional equity data sets. *Econometrica*, 84(3):985–1046.

Gareth, J., Daniela, W., Trevor, H., and Robert, T. (2013). *An introduction to statistical learning: with applications in R*. Spinger.

Gehrig, T., Sögner, L., and Westerkamp, A. (2020). Making portfolio policies work. *Working Paper*.

Harvey, C. R., Liu, Y., and Zhu, H. (2016). ... and the cross-section of expected returns. *The Review of Financial Studies*, 29(1):5–68.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition.

Hautsch, N. and Voigt, S. (2019). Large-scale portfolio allocation under transaction costs and model uncertainty. *Journal of Econometrics*, 212(1):221–240.

Hoerl, A. E. and Kennard, R. W. (1970). Ridge regression: Applications to nonorthogonal problems. *Technometrics*, 12(1):69–82.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257.

Hou, K., Xue, C., and Zhang, L. (2014). Digesting Anomalies: An Investment Approach. *The Review of Financial Studies*, 28(3):650–705.

Hou, K., Xue, C., and Zhang, L. (2020). Replicating anomalies. *The Review of Financial Studies*, 33(5):2019–2133.

Hull, J. C. (2020). *Machine Learning in Business An Introduction to the World of Data Science*. Independently published.

Kim, D. (1995). The errors in the variables problem in the cross-section of expected stock returns. *The Journal of Finance*, 50(5):1605–1634.

Kuhn, M. and Silge, J. (2018). *Tidy Modeling with R*. John Wiley & Sons.

McLean, R. D. and Pontiff, J. (2016). Does academic research destroy stock return predictability? *The Journal of Finance*, 71(1):5–32.

Menkveld, A. J., Dreber, A., Holzmeister, F., Huber, J., Johannesson, M., Kirchler, M., Neusüss, S., Razen, M., and Weitzel, U. (2021). Non-standard errors. *Working Paper*.

Mullainathan, S. and Spiess, J. (2017). Machine learning: An applied econometric approach. *Journal of Economic Perspectives*, 31(2):87–106.

Nagel, S. (2021). *Machine learning in asset pricing*. Princeton University Press.

Newey, W. K. and West, K. D. (1987). Hypothesis testing with efficient method of moments estimation. *International Economic Review*, pages 777–787.

Newey, W. K. and West, K. D. (1994). Automatic lag selection in covariance matrix estimation. *The Review of Economic Studies*, 61(4):631–653.

Regenstein Jr, J. K. (2018). *Reproducible finance with R: Code flows and shiny apps for portfolio analysis*. Chapman and Hall/CRC.

Shanken, J. (1992). On the estimation of beta-pricing models. *The Review of Financial Studies*, 5(1):1–33.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288.

Welch, I. and Goyal, A. (2008). A Comprehensive Look at The Empirical Performance of Equity Premium Prediction. *The Review of Financial Studies*, 21(4).

Wickham, H. (2014). Tidy data. *Journal of Statistical Software*, 59(1):1–23.

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., Takahashi, K., Vaughan, D., Wilke, C., Woo, K., and Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686.

Wickham, H. and Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data*. " O'Reilly Media, Inc.".

Zaffaroni, P. and Zhou, G. (2022). Asset pricing: Cross-section predictability. *Working Paper*.

Zeileis, A. (2004). Econometric computing with hc and hac covariance matrix estimators. *Journal of Statistical Software*, 11(10):1–17.

Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, 67(2):301–320.