Machine Learning for Factor Investing

Guillaume Coqueret and Tony Guida

2019-12-13 PRELIMINARY & VERY INCOMPLETE

To Leslie and Selin.

Contents

List of Tables

List of Figures

1 Preface

1.1 Foreword

By definition, many topics and references will have escaped our scrutiny. Our intent is to progressively improve the content of the book. We will be grateful to any comment that helps correct or update the monograph. Thank you for sending your feedback at guillaume.coqueret(at)gmail.com.

https://github.com/josephmisiti/awesome-machine-learning/blob/master/books.md

1.2 What this book is not about

This book deals with machine learning tools and their applications in factor investing. The topics we discussed are related to other themes that will not be covered in the monographs. These themes include:

- applications of ML in **other financial fields**, such as fraud detection or credit scoring. We refer to Ngai et al. (2011) and Baesens et al. (2015) for general purpose fraud detection, to Bhattacharyya et al. (2011) for a focus on credit cards and to Ravisankar et al. (2011) and Abbasi et al. (2012) for studies on fraudulent financial reporting. On the topic of credit scoring, Wang et al. (2011) and Brown and Mues (2012) provide overviews of methods and some empirical results.

- **use cases of alternative datasets** that show how to leverage textual data from social media, satellite imagery, or credit card logs to predict sales, earning reports, and, ultimately, future returns. The literature on this topic is still emerging (see, e.g., Blank et al. (2019), Jha (2019) and Ke et al. (2019)) but will likely blossom in the near future.

- **technical details** of machine learning tools. While we do provide some insights on specificities of some approaches (those we believe are important), the purpose of the book is not to serve as reference manual on statistical learning. We refer to Hastie et al. (2009) for a general treatment on the subject, to Du and Swamy (2013) and Goodfellow et al. (2016) for monographs on neural networks particularly and to Sutton and Barto (2018) for a tour in reinforcement learning.

1.3 The targeted audience

Who should read this book? This book is intended for two audiences. First, postgraduate students who wish to pursue their studies in quantitative finance with a view towards investment and asset management. The second target are professionals from the money management industry who either seek to pivot towards allocation methods that are based on machine learning or are simply interested in these new tools and want to upgrade their set of competences. To a lesser extent, the book can serve to scholars or researchers who need a manual with a broad spectrum of references both on recent asset pricing issues and machine learning algorithms applied to money management.

The book assumes basic knowledge in **algebra** (matrix manipulation), **analysis** (function differentiation, gradients), **optimization** (first and second order conditions), and **statistics** (distributions, moments, tests). A minimal **financial culture** is also required: simple notions like stocks, accounting quantities (e.g., book value) will not be defined in this book.

1.4 How this book is structured

The book can be divided into four parts.

The first part gathers preparatory material like notations (Chapter 2) and introductory remarks (Chapter 3). The next two chapters are very important. The first one (Chapter 4) outlines the economic foundations (theoretical and empirical) of factor investing and sums up recent literature. The second one (Chapter 5) deals with data preparation. It briefly recalls the basic tips and warns about major issues.

The second part of the book is dedicated to predictive algorithms in supervised learning. Those are the most common tools that are used to forecast financial quantities (returns, volatilities, Sharpe ratios, etc.). They range from penalized regressions (Chapter 6), to tree methods (Chapter 7), neural networks (Chapter 8), support vector machines (Chapter 9) and Bayesian approaches (Chapter 10).

The next portion of the book bridges the gap between the tools and their application in finance. Chapter 11 details how to assess and improve the ML engines defined beforehand. Chapter 12 explains how models can be combined and often why that may not be a good idea. Finally, one of the most important chapters (number 13) reviews the critical steps of portfolio backtesting and mentions the frequent mistakes that are often encountered at this stage.

The end of the book covers a range of advanced topics connected to machine learning more specifically. The first one is interpretability. ML models are often considered to be black boxes and this raises trust issues: how and why should one trust ML-based predictions? Chapter 14 is intended to present methods that help understand what is happening under the hood. Chapter 15 is focused on causality, which is a much more powerful concept than correlation. Most ML tools rely on correlation-like patterns and it is important to underline the benefits and techniques related to causality. Chapters 16 and 17 are dedicated to non supervised methods. The latter can be useful, but their financial applications should be wisely and cautiously motivated. Laslty, the final chapter (18) introduces standard approaches for the treatment of textual data.

1.5 Companion website

www.mlfactor.com copy paste code easily.

1.6 Why R?

The supremacy of Python as the ML programming language is a widespread belief. This is because almost all applications of deep learning (which is as of 2020 one of the most fashionable branches of ML) are coded in Python via Tensorflow or Pytorch. The fact is that **R** has a **lot** to offer as well. First of all, let us not forget that one of the most influencial textbooks in ML (Hastie et al. (2009)) is written by statisticians who code in R. Moreover, many statistics-orientated algorithms (e.g. BARTs in Section 10.5) are primarily coded in R and not Python. The R offering in Bayesian packages in general (`https://cran.r-project.org/web/views/Bayesian.html`) and in Bayesian learning in particular is probably unmatched.

There are currently several ML frameworks available in R.

- **caret**: `https://topepo.github.io/caret/index.html`, a compilation of more than 200 ML models;
- **tidymodels**: `https://github.com/tidymodels`, a collection of packages for ML workflow;
- **rtemis**: `https://rtemis.netlify.com`, a general purpose package for ML and visualization;
- **mlr3**: `https://mlr3.mlr-org.com/index.html`, also a simple framework for ML models;
- **h2o**: `https://github.com/h2oai/h2o-3/tree/master/h2o-r`, a large set of tools provided by h2o (coded in Java);
- **Open ML**: `https://github.com/openml/openml-r`, the R version of the OpenML (www.openml.org) community.

Moreover, via the *reticulate* package, it is possible (but not always easy) to benefit from Python tools as well. The most prominent example is the adaptation of the *tensorflow* and *keras* libraries to R. Thus, some very advanced Python material is readily available to R users. This is also true for other resources, like Stanford's CoreNLP library (in Java) which was adapted to R in the package *coreNLP* (which we will not use in this book).

Package	Purpose	Chapter(s)
adabag	Boosted trees	7
BART	Bayesian additive trees	10
broom	Tidy regression output	XXX
CAM	Causal Additive Models	XXX
caTools	AUC curves	11
breakDown	Breakdown interpretability	14
dummies	One-hot encoding	8
e1071	Support Vector Machines	9
factoextra	PCA visualization	16XXX
FNN	Nearest Neighbors detection	16XXX
ggpubr	Combining plots	11
glmnet	Penalized regressions	6
iml	Interpretability tools	14
keras	Neural networks	8
lime	Interpretability	14
lmtest	Granger causality	15
lubridate	Handling dates	All (or many)
MlBayesOpt	Bayesian hyperparameter tuning	11
naivebayes	Naive Bayes classifier	10
quadprog	Quadratic programming	12
quantmod	Data extraction	4, 12
randomForest	Random forests	7
ReinforcementLearning	Reinforcement Learning	17
rpart and *rpart.plot*	Simple decision trees	7
spBayes	Bayesian linear regression	10
tidyverse	Environment for data science, data wrangling	All
xgboost	Boosted trees	7
xtable	Table formatting	4

TABLE 1.1: List of all packages used in the book.

1.7 Coding instructions

One of the purposes of the book is to propose a large scale tutorial of ML applications in financial predictions and portfolio selection. Thus, one keyword is **REPRODUCIBILITY**!

R and RStudio. Coding requirements, tidyverse / dplyr+tidyr, filter, select, arrange, spread, gather. R libraries Install packages install.packages("nameofthepackage") Short chunks Comments ## output Big tutorial so most of the chunks depend on previously defined variables. When replicating parts of the code, make sure that the environment includes all relevant variables.

Of all of these packages (or collections thereof), the **tidyverse** and **lubridate** are compulsory in almost all sections of the book.

1.8 Acknowledgements

The core of the book was prepared for a lecture given by one of the authors to students of Masters Degrees in Finance at EMLYON Business School and at the Imperial College Business School in the Spring of 2019. We thank Eric André, Bertand Tavin, and Aurélie Brossard for friendly reviews; Christophe Dervieux for his help with bookdown; John Kimmel for making this happen and Jonathan Regenstein for his availability, no matter the topic.

1.9 Future developments

The fields of machine learning and factor modelling are developing at a fast pace. The content of this book will always constitute a solid background but it is naturally destined to obsolescence. As much as we can, we will update it with the latest ongoing research.[1] Until then, all errors are ours.

[1] We again thank any reader for sharing useful references and sending them to guillaume.coqueret(at)gmail.com.

2

Notations and data

2.1 Notations

Bold notations indicate vectors and matrices. We use capital letters for matrices and lower case letters for vectors. $\mathbf{v}'$ and $\mathbf{M}'$ denote the transposes of $\mathbf{v}$ and $\mathbf{M}$. $\mathbf{M} = [m]_{i,j}$, where i is the row index and j the column index.

We will work with two notations in parallel. The first one is the pure machine learning notation in which the **labels** (also called **output** or **dependent** variables) $\mathbf{y} = y_i$ are approximated by functions of features $\mathbf{X}_i = (x_{i,1}, \dots, x_{i,K})$. The dimension of the feature matrix $\mathbf{X}$ is $I \times K$: there are I **instances**, **records**, or **observations** and each one of them has K **attributes**, **features**, **inputs**, or **predictors** which will serve as **independent** and **explanatory** variables (all these terms will be used interchangeably). Sometimes, to ease notations, we will write $\mathbf{x}_i$ for one instance (one row) of $\mathbf{X}$ or $\mathbf{x}_k$ for one (feature) vector of $\mathbf{X}$.

The second field is finance and will directly relate to the first. We will often work with discrete returns $r_{t,n} = p_{t,n}/p_{t-1,n} - 1$ computed from price data. Here t is the time index and n the asset index. Unless specified otherwise, the return is always computed over one period, though this period can sometimes be one month or one year. Whenever confusion is possible, we will specify other notations for returns.

In line with our previous conventions, the number of return dates will be T and the number of assets, N. The features or characteristics of assets will be denoted with $x_{t,n}^{(k)}$: it is the time-t value of the k^{th} attribute of firm or asset n. Moreover, $\mathbf{r}_t$ stands for all returns at time t while $\mathbf{r}_n$ stand for all returns of asset n. Often, returns will play the role of the dependent variable, or label (in ML terms). For the riskless asset, we will use the notation $r_{t,f}$.

The link between the two notations will most of the time be the following. One instance i will consist one one couple (t, n) of one particular date and one particular firm (if the data is perfectly rectangular with no missing field, $I = T \times N$). The label will usually be some performance measure of the firm computed over some future period while the features will consist of the firm attributes at time t. Hence, the purpose of the machine learning engine in factor investing will be to determine the model that maps the time-t characteristics of firms into their future performance.

In terms of canonical matrices: $\mathbf{I}_N$ will denote the $(N \times N)$ identity matrix.

From the probabilistic literature, we employ the expectation operator $\mathbb{E}[\cdot]$ and the conditional expectation $\mathbb{E}_t[\cdot]$, where the corresponding filtration $\mathcal{F}_t$ corresponds to all information available at time t. More precisely, $\mathbb{E}_t[\cdot] = \mathbb{E}[\cdot|\mathcal{F}_t]$. $\mathbb{V}[\cdot]$ will denote the variance operator. Depending on the context, probabilities will be written simply P, but sometimes we will use the heavier notation $\mathbb{P}$. Probability distribution functions (pdfs) will be denoted with lowercase letters (f) and cumulative distribution functions (cdfs) with uppercase letters (F). We will write equality in distribution as $X \overset{d}{=} Y$, which is equivalent to $F_X(z) = F_Y(z)$ for all z on the support of the variables.

Sometimes, asymptotic behaviours will be characterized with the usual Landau notation $o(\cdot)$ and $O(\cdot)$. The symbol $\propto$ refers to proportionality: $x \propto y$ means that x is proportional to y. With respect to derivatives, we use the standard notation $\frac{\partial}{\partial x}$ when differentiating with respect to x. We resort to the symbol ∇ when all derivatives are computed (gradient vector).

Finally, we turn to functions. We list a few below:
- $1_{\{x\}}$: the indicator function of the condition x, which is equal to one if x is true and to zero otherwise. - $\phi(\cdot)$ and $\Phi(\cdot)$ are the standard Gaussian pdf and cdf.
- card$(\cdot)$ is the cardinal function: it evaluates the number of elements in a given set.
- $\lfloor \cdot \rfloor$ is the integer part function.
- for a real number x, $[x]^+$ is the positive part of x, that is $\max(0, x)$.
- tanh$(\cdot)$ is the hyperbolic tangent: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$.
- ReLu$(\cdot)$ is the rectified linear unit: $\text{ReLu}(x) = \max(0, x)$.
- s$(\cdot)$ will be the softmax function: $s(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^J e^{x_j}}$, where the subscript i refers to the i^{th} element of the output.

2.2 Dataset

Throughout the book, we will illustrate the concepts we present with examples of implementation based on a single financial dataset. This dataset comprises information on 1,207 stocks listed in the US (possibly originating from Canada or Mexico). The time range starts in November 1998 and ends in March 2019. For each point in time, 96 characteristics describe the firms in the sample. These attributes cover a wide range of topics:

- valuation (earning yields, accouting ratios)
- profitability and quality (return on equity)
- momentum and technical analysis (12-1 monthly returns, relative strength index)
- risk (volatilities)
- estimates (earnings-per-share)
- volume and liquidity (share turnover)

The sample is not perfectly rectangular: there are no missing points but the number of firms and their attributes is not constant through time. This makes the computations in the backtest more tricky, but also more realistic.

```
library(tidyverse)          # Activate the data science package
library(lubridate)          # Activate the date management package
load("data_ml.RData")                              # Load the data
data_ml <- data_ml %>%
    filter(date > "1999-12-31",                # Keep the date with sufficient data points
           date < "2019-01-01") %>%
    arrange(stock_id, date)                    # Order the data
data_ml[1:6, 1:6]                              # Sample values
```

```
## # A tibble: 6 x 6
##   stock_id date       Advt_12M_Usd Advt_3M_Usd Advt_6M_Usd Asset_Turnover
##      <int> <date>            <dbl>       <dbl>       <dbl>          <dbl>
## 1        1 2000-01-31         0.41        0.39        0.42           0.19
## 2        1 2000-02-29         0.41        0.39        0.4            0.19
## 3        1 2000-03-31         0.4         0.37        0.37           0.2
## 4        1 2000-04-30         0.39        0.36        0.37           0.2
## 5        1 2000-05-31         0.4         0.42        0.4            0.2
## 6        1 2000-06-30         0.41        0.47        0.42           0.21
```

The data has 99 columns and 268336 rows. The first two columns indicate the stock identifier and the date. The points are sampled at the monthly frequency. There are four immediate labels in the dataset: R1M_Usd, R3M_Usd, R6M_Usd and R12M_Usd, which correspond to the 1 month, 3 month, 6 month and 12 month future/forward returns of the stocks. These labels are located in the last 4 columns. We provide their descriptive statistics below.

In anticipation for future models, we keep the name of the predictors in memory. In addition, we also keep a much shorter list of predictors.

The original labels are numerical and will be used for regression exercises, that is, when the objective is to predict a scalar real number. Sometimes, the exercises can be different and the purpose is to forecast a category, like “buy”, “hold” or “sell”. In order to be able to perform this type of analysis, we create additional labels that are categorical.

```
data_ml <- data_ml %>%
    group_by(date) %>%                                   # Group by date
    mutate(R1M_Usd_C = R1M_Usd > median(R1M_Usd),        # Create the categorical labels
           R12M_Usd_C = R1M_Usd > median(R12M_Usd)) %>%
    ungroup() %>%
    mutate_if(is.logical, as.factor)
```

The labels are binary: they are equal to 1 (true) if the original return is above that of the median return over the considered period and to 0 (false) if not. Hence, at each point in time, half of the sample is either equal to zero or one: some stocks overperforms and others underperform.

In machine learning, models are estimated on one portion of data (training set) and then tested on another portion of the data (testing set) to assess their quality. We split our sample accordingly.

```
separation_date <- as.Date("2014-01-15")
training_sample <- filter(data_ml, date < separation_date)
testing_sample <- filter(data_ml, date >= separation_date)
```

We also keep in memory a few key variables, like the list of asset identifiers and a rectangular version of returns. For simplicity, in the computation of the latter, we shrink the investment universe to keep only the stocks for which we have the maximum number of points.

```
stock_ids <- levels(as.factor(data_ml$stock_id)) # A list of all stock_ids
stock_days <- data_ml %>%                        # Compute the number of data points per stock
    group_by(stock_id) %>% summarize(nb = n())
stock_ids_short <- stock_ids[which(stock_days$nb == max(stock_days$nb))] # Stocks with full data
```

```
returns <- data_ml %>%                          # Compute returns, in matrix format, in 3 steps:
    filter(stock_id %in% stock_ids_short) %>%   # 1. Filtering the data
    select(date, stock_id, R1M_Usd) %>%         # 2. Keep returns along with dates & firm names
    spread(key = stock_id, value = R1M_Usd)     # 3. Put in matrix shape
```

3

Introduction

3.1 Context

The blossoming of machine learning in factor investing has it source at the confluence of three favorable developments: data availability, computational capacity, and economic groundings.

First, the data. Nowadays, classical providers, such as Bloomberg and Reuters have seen their playing field invaded by niche players and aggregation platforms.[1] In addition, high-frequency data and derivative quotes have become mainstream. Hence, firm-specific attributes are easy and often cheap to compile. This means that the size of $\mathbf{X}$ in (3.1) is now sufficiently large to be plugged into ML algorithms. The order of magnitude (in 2019) that can be reached is the following: a few hundred monthly observations over several thousand stocks (US listed at least) covering a few hundred attributes. This makes a dataset of dozens of millions of points. While it is a reasonably high figure, we highlight that the chronological depth is probably the weak point and will remain so for decade to come because accounting figures are only released on a quarterly basis. Needless to say that this drawback does not hold for high-frequency strategies.

Second, computational power, both through hardware and software. Storage and processing speed are no technical hurdles anymore and models can even be run on the cloud thanks to services hosted by major actors (Amazon, Microsoft, IBM and Google) and by smaller players (Rackspace, Techila). On the software side, open source has become the norm, funded by corporations (TensorFlow & Keras by Google, Pytorch by Facebook, h2o, etc.), universities (Scikit-Learn by INRIA, NLPCore by Stanford, NLTK by UPenn) or individual or groups of researchers (caret, xgboost, tidymodels to list but a pair of frameworks). Consequently, ML is no longer the private turf of a handful of expert computer scientist, but is on the contrary accessible to anyone willing to learn and code.

Finally, economic framing. Machine learning applications in finance were initially introduced by computer scientists and information system experts (e.g., Braun and Chandler (1987), White (1988)) and exploited shortly after by academics in financial economics (Bansal and Viswanathan (1993)). Nonlinear relationships then became more mainstream in asset pricing (Freeman and Tse (1992), Bansal et al. (1993)). These contributions started to pave the way for the more brute-force approaches that have blossomed since the 2010 decade.

In the synthetic proposal of Arnott et al. (2019b), the first piece of advice is to rely on a model that makes sense economically. While we agree with this stance, the only assumption that we make in this book is that future returns depend on firm characteristics. The relationship between these features and performance is largely unknown and probably time-varying. This

[1] We refer to https://alternativedata.org/data-providers/[2] for a list of alternative data providers. Moreover, we recall that Quandl, an alt-data hub was acquired by Nasdaq in December 2018.

is why ML can be useful: to detect some hidden patterns beyond the documented asset pricing anomalies.

$$\mathbf{y} = f(\mathbf{X}) + \epsilon \tag{3.1}$$

3.2 Portfolio construction: the workflow

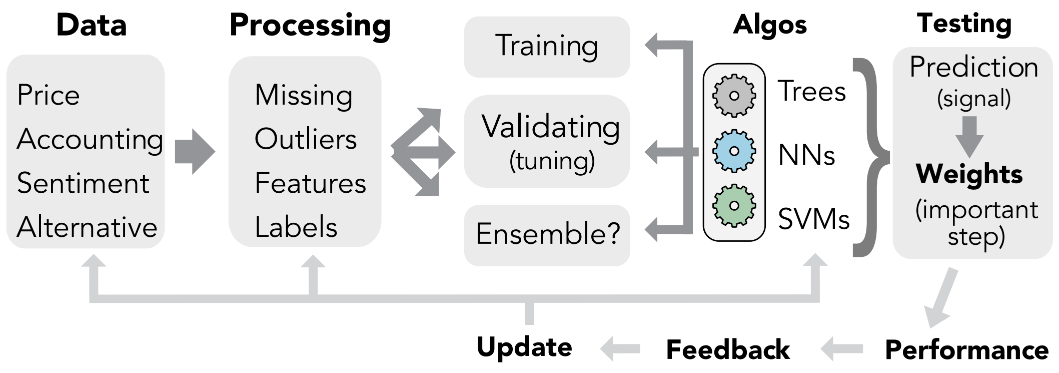


FIGURE 3.1: Simplified workflow in portfolio construction.

3.3 Machine Learning is no Magic Wand

By definition, the curse of predictions is that they rely on past data to infer patterns about subsequent fluctuations. The more or less explicit hope of any forecaster is that the past will turn out to be a good approximation of the future. Needless to say, this is a pious wish: in general, predictions fare badly. Surprisingly, this does not depend much on the sophistication of the econometric tool. In fact, heuristic guesses are often hard to beat.

Hard to translate computer vision or textual ML into financial predictions.

To illustrate this sad truth, the baseline algorithms that we detail in chapters 6 to 8 yield at best mediocre results.

The attentive reader will have noticed that…

4

Factor investing and asset pricing anomalies

Asset pricing anomalies are the foundations of factor investing. In this chapter the aim is twofold:

- present simple ideas and concepts: basic factor models, common empirical facts (time-varying nature of returns and risk premia);

- provide the reader with articles that go much deeper.

The purpose of this chapter is not to provide a full treatment of the many topics related to factor investing. Rather, it is intended to give a broad overview and cover the essential themes so that the reader is guided towards the relevant references. As such, it can serve as a short, non-exhaustive, review of the literature. The subject of factor modelling in finance is incredibly vast and the number of papers dedicated to it is substantial and still rapidly increasing.

Several monographs are already dedicated to the topic of style allocation (a synonym of factor investing) To cite but a few, we mention:

- Ilmanen (2011): an exhaustive excursion into risk premia, across many asset classes, with a large spectrum of descriptive statistics (across factors and periods),

- Ang (2014): covers factor investing with a strong focus on the money management industry,

- Bali et al. (2016), very complete book on the cross-section of signals with statistical analyses (univariate metrics, correlations, persistence, etc.),

- Jurczenko (2017): a tour on various topics given by field experts (factor purity, predictability, selection vs weighting, factor timing, etc.).

Finally, we mention a few wide-scope papers on this topic: Goyal (2012), Cazalet and Roncalli (2014) and Baz et al. (2015).

4.1 Introduction

The topic of factor investing, though a decades-old academic theme, has gained traction concurrently with the rise of Equity Traded Funds (ETFs) as vectors of investment. Both have gathered momentum in the 2010 decade. Not so surprisingly, the feedback loop between practical financial engineering and academic research has stimulated both sides in a mutually beneficial manner. Practitioners rely on key scholarly findings (e.g., asset pricing anomalies) while researchers dig deeper into pragmatic topics (e.g., factor exposure or transaction costs). Recently, researchers have also tried to quantify and qualify the impact of factor indices on financial markets. For instance, Krkoska and Schenk-Hoppé (2019) analyze herding behaviors while Cong and Xu (2019) show that the introducting of composite securities increase volatilty and cross-asset correlations.

The core aim of factor models is to understand the **drivers of asset prices**. Broadly speaking, the rationale behind factor investing is that the financial performance of firms depend on factors, whether they be latent and inobservable, or related to intrinsic characteristics (like accounting ratios for instance). As such, factor models can be viewed as special cases of the arbitrage pricing theory (APT) of Ross (1976), which assumes that the return on an asset n can be modelled as a linear combination of underlying factors f_k:

$$r_{t,n}=\alpha_n+\sum_{k=1}^K\beta_{n,k}f_{t,k}+\epsilon_{t,n}, \tag{4.1}$$

where the usual econometric constraints on linear models hold: $\mathbb{E}[\epsilon_{t,n}]=0$, $\text{cov}(\epsilon_{t,n},\epsilon_{t,m})=0$ for $n\neq m$ and $\text{cov}(f_n,\epsilon_n)=0$. If such factors do exist, then they are in contradiction with the cornerstone model in asset pricing: the capital asset pricing model (CAPM) of Sharpe (1964), Lintner (1965) and Mossin (1966). Indeed, according to the CAPM, the only driver of returns is the market portfolio. This explains why factors are also called 'anomalies'.

Empirical evidence of asset pricing anomalies has accumulated since the dual publication of Fama and French (1992) and Fama and French (1993). This seminal work has paved the way for a blossoming stream of literature that has its meta-studies (e.g., Green et al. (2013), Harvey et al. (2016) and McLean and Pontiff (2016)). The regression (4.1) can be evaluated once (unconditionally) or sequentially over different time frames. In the latter case, the parameters (coefficient estimates) change and the models are thus called *conditional* (we refer to Ang and Kristensen (2012) and to Cooper and Maio (2019) for recent results on this topic as well as for a detailed review on the related research). Conditional models are more flexible because they acknowledge that the drivers of asset prices may not be constant, which seems like a reasonable postulate.

4.2 Detecting anomalies

Obviously, a crucial step is to be able to identify an anomaly and the complexity of this task should not be underestimated. Given the publication bias towards positive results (see, e.g., Harvey (2017) in Finance), researchers are often tempted to report partial results that are sometimes invalidated by further studies. The need for replication is therefore high and many findings have no tomorrow (Linnainmaa and Roberts (2018)). Some researcher document fading effects because of publication: once the anomaly becomes public, agents invest in it, which pushes prices up and the anomaly disappears. McLean and Pontiff (2016) document this effect in the US but Jacobs and Müller (2019) find that all other countries experience sustained post-publication factor returns. With a different methodology, Chen and Zimmermann (2019) introduce a publication bias adjustment for returns and the authors note that this (negative) adjustment is in fact rather small. Penasse (2018) recommends the notion of *alphadecay* to study the persistence or attenuation of anomalies.

The destruction of factor premia may due to herding (Krkoska and Schenk-Hoppé (2019)) and could be accelerated by the democritization of so-called smart-beta products (Equity Traded Funds (ETFs) notably) that allow investors to directly invest in particular styles (value, low volatility, etc.).

This subsection was inspired from Baker et al. (2017) and Harvey and Liu (2017).

4.2.1 Simple portfolio sorts

This is the most common procedure and the one used in Fama and French (1992). The idea is simple. On one date,

1. rank firms according to a particular criterion (e.g., size, book-to-market ratio);

2. form $J \geq 2$ portfolios (i.e. homogeneous groups) consisting of the same number of stocks according to the ranking (usually, $J = 2$, $J = 3$, $J = 5$ or $J = 10$ portfolios are built, based on the median, terciles, quintiles or deciles of the criterion);

3. the weight of stocks inside the portfolio is either uniform (equal weights), or proportional to market capitalisation;
4. at a future date (usually one month), report the returns of the portfolios.
   Then, iterate the procedure until the chronological end of the sample is reached.

The outcome is a time-series of portfolio returns r_t^j for each grouping j. An anomaly is identified if the t-test between the first ($j = 1$) and the last group ($j = J$) unveils a significant difference in average returns. A strong limitation of this approach is that the sorting criterion could have a non monotonous impact on returns and the simple t-statistic would not detect it. Several articles address this concern: Patton and Timmermann (2010) and Romano and Wolf (2013) for instance.

Instead of focusing on only one criterion, it is possible to group asset according to more characteristics. The original paper Fama and French (1992) also combines market capitalization with book-to-market ratios. Each characteristic is divided into 10 buckets, which makes 100 portfolios in total. Beyond data availability, there is no upper bound on the number of features that can be included the sorting process. In fact, some authors investigate more complex sorting algorithms that can manage a potentially large number of characteristics (see e.g., Feng et al. (2019b) and Bryzgalova et al. (2019b)).

Finally, we refer to Ledoit et al. (2018) for refinements that take into account the covariance structure of asset returns and to Cattaneo et al. (2019) for a theoretical study on the statistical properties of the sorting procedure. Notably, thelatter paper discusses the optimal number of portfolios.

In the code and Figure 4.1 below, we compute size portfolios (equally weighted: above versus below the median capitalization). According to the size anomaly, the firms with below median market cap should earn higher returns on average. This is verified whenever the red bar in the plot is above the green one (it happens most of the time).

```
data_ml %>%
    group_by(date) %>%
    mutate(large = Mkt_Cap_12M_Usd > median(Mkt_Cap_12M_Usd)) %>% # Creates the cap sort
    ungroup() %>%                                                  # Ungroup
    mutate(year = lubridate::year(date)) %>%                # Creates a year variable
    group_by(year, large) %>%                               # Analyze by year & cap
    summarize(avg_return = mean(R1M_Usd)) %>%               # Compute average return
    ggplot(aes(x = year, y = avg_return, fill = large)) +   # Plot!
    geom_col(position = "dodge") +                          # Bars side-to-side
    theme(legend.position = c(0.8, 0.2)) +                  # Legend location
    coord_fixed(124) +                                      # x/y aspect ratio
    scale_fill_manual(values=c("#F87E1F", "#0570EA"))       # Colors
```

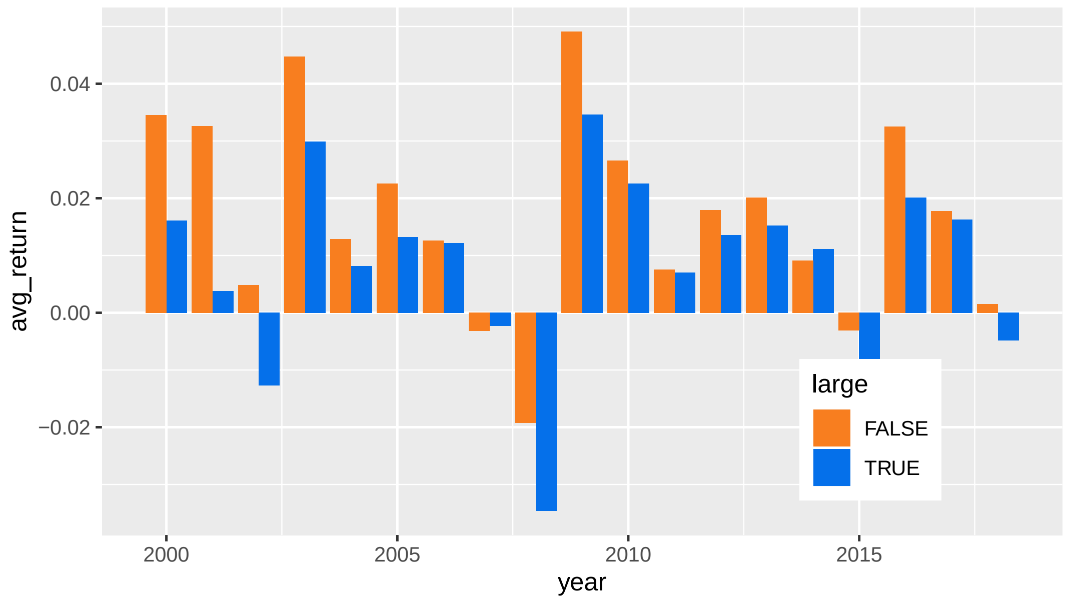


FIGURE 4.1: The size factor: average returns of small (=FALSE) versus large (=TRUE) firms.

4.2.2 Factors

The construction of so-called factors follows the same lines as above. Portfolios are based on one characteristic and the factor is a long-short ensemble of one extreme portfolio minus the opposite extreme (small minus large for the size factor or high book-to-market ratio minus low book-to-market ratio for the value factor). Sometimes, subtleties include forming bivariate sorts and aggregating several portfolios together, as in the original contribution of Fama and French (1993). The most common factors are listed below, along with a few references. We refer to the books listed at the beginning of the chapter for a more exhaustive treatment of factor idiosyncrasies. For most anomalies, theoretical justifications have been brought forward, whether risk-based or behavioural. We list the most frequently cited factors below:

- size (**SMB** = small firms minus large firms): Banz (1981), Fama and French (1992), Fama and French (1993), Van Dijk (2011), Asness et al. (2018) and Astakhov et al. (2019).

- value (**HM** = high minus low: undervalued minus 'growth' firms): Fama and French (1992), Fama and French (1993), Asness et al. (2013).

- momentum (**WML** = winners minus looser): Jegadeesh and Titman (1993), Asness et al. (2013). The winners are the assets that have experienced the highest returns over the last year (sometimes the computation of the return is truncated to omit the last month). Cross-sectional momentum is linked, but not equivalent, to time-series momentum (trend following), see e.g., Moskowitz et al. (2012) and Lempérière et al. (2014).

- profitability (**RMW** = robust minus weak profits): Fama and French (2015), Bouchaud et al. (2019). In the former reference, profitability is measured as (revenues - (cost and expenses))/equity.

- investment (**CMA** = conservative minus aggressive): Fama and French (2015), Hou et al. (2015). Investment is measures via the growth of total assets (divided by total assets). Aggressive firms are those that experience the largest growth in assets.

- low 'risk' (sometimes: **BAB** = betting against beta): Ang et al. (2006), Baker et al. (2011), Frazzini and Pedersen (2014), Boloorforoosh et al. (2019). In this case, the computation of risk changes from one article to the other (simple volatility, market beta, idiosyncratic volatility, etc.)

With the notable exception of the low risk premium, the most mainstream anomalies are kept and updated in the data library of Kenneth French (`https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html`). Of course, the computation of the factors follows a particular set of rules, but they are generally accepted in the academic sphere. Another source of data is the AQR repository: `https://www.aqr.com/Insights/Datasets`.

Below, we import data from Ken French's data library. We will use it later on in the chapter.

```r
library(quantmod)                         # Package for data extraction
library(xtable)                           # Package for LaTeX exports
min_date <- "1963-07-31"                  # Start date
max_date <- "2019-11-28"                  # Stop date
temp <- tempfile()
KF_website <- "http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/"
KF_file <- "ftp/F-F_Research_Data_5_Factors_2x3_CSV.zip"
link <- paste0(KF_website,KF_file)        # Link of the file
download.file(link, temp, quiet = TRUE)   # Download!
FF_factors <- read_csv(unz(temp, "F-F_Research_Data_5_Factors_2x3.CSV"),
                       skip = 3) %>%          # Check the number of lines to skip!
    rename(date = X1, MKT_RF = `Mkt-RF`) %>%  # Change the name of the first column
    mutate_at(vars(-date), as.numeric) %>%                 # Convert values to number
    mutate(date = ymd(parse_date_time(date, "%Y%m"))) %>%  # Date in right format
    mutate(date = rollback(date + months(1)))              # End of month date
FF_factors <- FF_factors %>% mutate(MKT_RF = MKT_RF / 100, # Scale returns
                                    SMB = SMB / 100,
                                    HML = HML / 100,
                                    RMW = RMW / 100,
                                    CMA = CMA / 100,
                                    RF = RF/100) %>%
    filter(date >= min_date, date <= max_date)             # Finally, keep only recent points
knitr::kable(head(FF_factors),  booktabs = TRUE,
             caption = "Sample of monthly factor values.") # A look at the data (see table)
```

Posterior to the discovery of these stylised facts, some contributions have aimed at building theoretical models that capture these properties. We cite a handful below:

- size and value: Berk et al. (1999), Daniel et al. (2001b), Barberis and Shleifer (2003), Gomes et al. (2003), Carlson et al. (2004), arnott2014can;

TABLE 4.1: Sample of monthly factor values.

date	MKT_RF	SMB	HML	RMW	CMA	RF
1963-07-31	-0.0039	-0.0047	-0.0083	0.0066	-0.0115	0.0027
1963-08-31	0.0507	-0.0079	0.0167	0.0039	-0.0040	0.0025
1963-09-30	-0.0157	-0.0048	0.0018	-0.0076	0.0024	0.0027
1963-10-31	0.0253	-0.0129	-0.0010	0.0275	-0.0224	0.0029
1963-11-30	-0.0085	-0.0084	0.0171	-0.0045	0.0222	0.0027
1963-12-31	0.0183	-0.0189	-0.0012	0.0008	-0.0030	0.0029

- momentum: Johnson (2002), Grinblatt and Han (2005), Vayanos and Woolley (2013), Choi and Kim (2014).

In addition, recent bridges have been built between risk-based factor representations and behavioural theories. We refer essentially to Barberis et al. (2016) and Daniel et al. (2019) and the references therein.

While these factors (i.e., long/short portfolios) exhibit time-varying risk-premia, it is well-documented (and accepted) that they deliver positive returns over long horizons. We refer to Gagliardini et al. (2016) and to the survey Gagliardini et al. (2019), as well as to the related bibliography for technical details on estimation procedures of risk premia and the corresponding empirical results. A large sample study that documents regime changes in factor premia was also carried out by Ilmanen et al. (2019).

In Figure 4.2, we plot the average monthly return aggregated over each calendar year for five common factors. The risk free rate (which is not a factor per se) is the most stable while the market factor (aggregate market returns minus the risk-free rate) is the most volatile. This makes sense because it is the only long equity factor among the five series.

```
FF_factors %>%
    mutate(date = year(date)) %>%                       # Turn date into year
    gather(key = factor, value = value, - date) %>%     # Put in tidy shape
    group_by(date, factor) %>%                          # Group by year and factor
    summarise(value = mean(value)) %>%                  # Compute average return
    ggplot(aes(x = date, y = value, color = factor)) +  # Plot
    geom_line() + coord_fixed(500)                      # Fix x/y ratio
```

The individual attributes of investor who allocate towards particular factors is a blossoming topic. We list a few references below, even though, they somewhat lie out of the scope of this book. Betermier et al. (2017) show that value investors are older, wealthier and face lower income risk compared to growth investors: they are those in the best position to take financial risks. The study Cronqvist et al. (2015b) leads to different conclusions: it finds that the propensity to invest in value versus growth assets has roots in genetics and in life events (the latter effect being confirmed in Cocco et al. (2019) and the former being further detailed in a more general context in Cronqvist et al. (2015a)). Psychological traits can also explain some factors: when agents extrapolates, they are likely to fuel momentum (this topic is thoroughly reviewed in Barberis (2018)). Micro- and macro-economic consequences of these preferences are detailed in Bhamra and Uppal (2019). To conclude this paragraph, we mention that theoretical models have also been proposed that link agents' preferences and beliefs to market anomalies (see for instance Barberis et al. (2019)).

Finally, we highlight the need of replicability of factor premia. As is shown by Linnainmaa

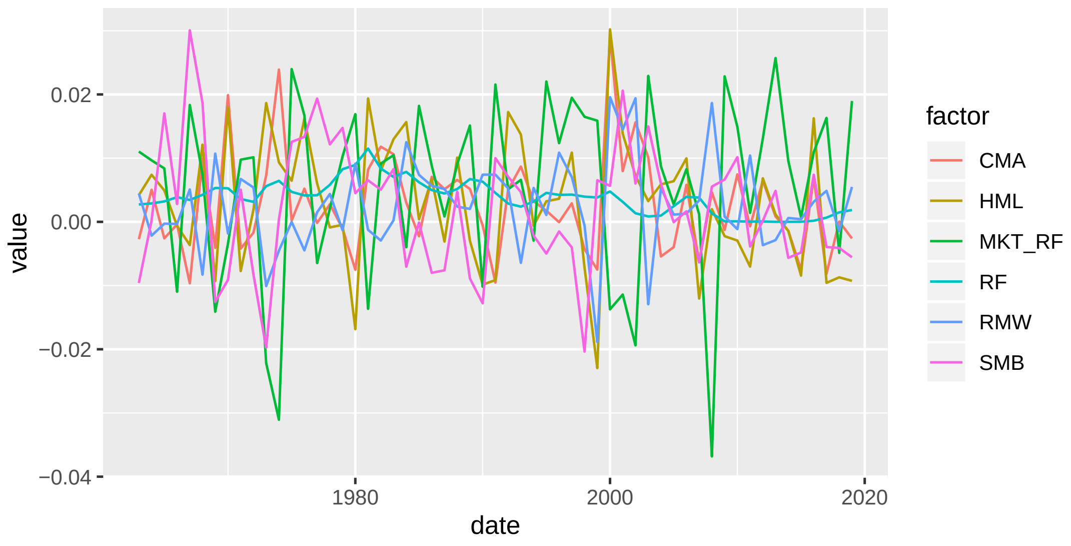


FIGURE 4.2: Average returns of common anomalies (1963-2020). Source: Ken French library.

and Roberts (2018) and Hou et al. (2019), many proclaimed factors are in fact very much data-dependent and often fail to deliver sustained profitability when the investment universe is altered.

Campbell Harvey, in a series of papers, tried to synthesize the research on factors: Harvey et al. (2016), Harvey and Liu (2017), Harvey and Liu (2019). His work underlines the need to set high bars for an anomaly to be called a 'true' factor. Increasing thresholds for p-values is only a partial answer as it is always possible to resort to data snooping in order to find an optimized strategy that will fail out-of-sample but that will deliver a t-statistic larger than three (or even four). Harvey (2017) recommends to resort to a Bayesian approach which blends data-based significance with a prior into a so-called Bayesanised p-value (see subsection below).

Following this work, researchers have continued to explore the richness of this zoo. Bryzgalova et al. (2019a) propose a tractable Bayesian estimation of large-dimensional factor models and evaluate all possible combinations of more than 50 factors, yielding an incredibly large number of coefficients. This combined with a Bayesianized Fama and MacBeth (1973) procedure allows to distinguish between pervasive and superfluous factors.

4.2.3 Predictive regressions, sorts, and p-value issues

For simplicity, assume a simple form:

$$\mathbf{r} = a + b\mathbf{x} + \mathbf{e}, \tag{4.2}$$

where the vector $\mathbf{r}$ stacks all returns of all stocks and $\mathbf{x}$ is a lagged variable so that the regression is indeed predictive. If the estimate $\hat{b}$ is significant given a specified threshold, then it can be tempting to conclude that $\mathbf{x}$ does a good job at predicting returns. Hence, long-short portfolios related to extreme values of $\mathbf{x}$ (mind the sign of $\hat{b}$) are expected to generate profits. This is unfortunately often false because $\hat{b}$ gives information on the *past* ability of $\mathbf{x}$ to forecast returns. What happens in the future may be another story.

Statistical tests are also used for portfolio sorts. Assume two extreme portfolios are expected to yield very different average returns (like very small cap versus very large cap, or strong winners versus bad losers). The portfolio returns are written r_t^+ and r_t^-. The simplest test for the mean is $t = \sqrt{T}\frac{m_{r_+}-m_{r_-}}{\sigma_{r_+-r_-}}$, where T is the number of points and $m_{r_\pm}$ denote the means of returns and $\sigma_{r_+-r_-}$ is the standard deviation of the difference between the two series, i.e., the volatility of the long/short portfolio. In short, the statistic can be viewed as a scaled Sharpe ratio (though usually these ratios are computed for long-only portfolios) and can in turn be used to compute p-values to assess the robustness of an anomaly. As is shown in Linnainmaa and Roberts (2018) and Hou et al. (2019), many factors discovered by reasearchers fail to survive in out-of-sample tests.

One reason why people are overly optimistic about anomalies they detect is the widespread reverse interpretation of the p-value. Often, it is thought of as the probability of one hypothesis (e.g., my anomaly exists) given the data. In fact, it's the opposite: it's the likelihood of your data sample, knowing that the anomaly holds.

$$\begin{aligned} p-\text{value} &= P[D|H] \\ \text{target prob.} &= P[H|D] = \frac{P[D|H]}{P[D]} \times P[H], \end{aligned}$$

where H stands for hypothesis and D for data. The equality in the second row is a plain application of Bayes' identity: the interesting probability is in fact a transform of the p-value.

Two articles (at least) discuss this idea. Harvey (2017) introduces **Bayesianized *p*-values**:

$$\text{Bayesianized } p-\text{value} = \text{Bpv} = e^{-t^2/2} \times \frac{\text{prior}}{1+e^{-t^2/2}\times \text{prior}}, \tag{4.3}$$

where t is the t-statistic obtained from the regression (i.e., the one that defines the p-value) and prior is the analyst's estimation of the odds that the hypothesis (anomaly) is true. The prior is coded as follows. Suppose there is a p% chance that the null holds (i.e (1-p)% for the anomaly). The odds are coded as $p/(1-p)$. Thus, if the t-statistic is equal to 2 (corresponding to a p-value of 5% roughly) and the prior odds are equal to 6, then the Bpv is equal to $e^{-2}\times 6 \times (1+e^{-2}\times 6)^{-1} \approx 0.448$ and there is a 44.8% chance that the null is true. This interpretation stands in sharp contrast with the original p-value which cannot be viewed as a probability that the null holds. Of course, one drawback is that the level of the prior is crucial and solely user-specified.

The work of Chinco et al. (2019b) is very different but shares some key concepts, like the introduction of Bayesian priors in regression outputs. They show that introducing an L^2 constraint on the predictive regression (see the ridge regression in Chapter 6) amounts to introduce views on what the true distribution of b is. The stronger the constraint, the more the estimate $\hat{b}$ will the shrunk towards zero. One key idea in their work is the introduction of a distribution of the true b across many anomalies. It is assumed to be Gaussian and centered. The interesting parameter is the standard deviation: the larger it is, the more significant anomalies are discovered. Notably, the authors show that this parameter changes through time and we refer to the original paper for more details on this subject.

4.2.4 Fama-Macbeth regressions

Another detection method was proposed by Fama and MacBeth (1973) through a two-stage regression analysis of risk premia. The first stage is a simple estimation of the relationship

(4.1): the regressions are run on a stock-by-stock basis over the corresponding time-series. The resulting estimates $\hat{\beta}_{i,k}$ are then plugged into a second series of regressions:

$$r_{t,n} = \gamma_{t,0} + \sum_{k=1}^{K} \gamma_{t,k}\hat{\beta}_{n,k} + \varepsilon_{t,n}, \tag{4.4}$$

which are ran date-by-date on the cross-section of assets.[1] Theoretically, the betas would be known and the regression would be run on the $\beta_{n,k}$ instead of their estimated values. The $\hat{\gamma}_{t,k}$ estimate the premia of factor k at time t. Under suitable distributional assumptions over the $\varepsilon_{t,n}$, statistical tests can be perform to determine whether these premia are significant or not. Typically, the statistic on the time-aggregated (average) premia $\hat{\gamma}_k = \frac{1}{T}\sum_{t=1}^{T}\hat{\gamma}_{t,k}$:

$$t_k = \frac{\hat{\gamma}_k}{\hat{\sigma_k}/\sqrt{T}}$$

is often used in pure Gaussian contexts to assess whether or not the factor is significant ($\hat{\sigma}_k$ is the standard deviation of the $\hat{\gamma}_{t,k}$).

We refer to Jagannathan and Wang (1998) and Petersen (2009) for technical discussions on the biases and losses in accuracy that can be induced by standard OLS estimations. Moreover, as the $\hat{\beta}_{i,k}$ in the second-pass regression are *estimates*, a second level of errors can arise (the so-called errors in variables). The interested reader will find some extensions and solutions in Shanken (1992), Ang et al. (2018) and Jegadeesh et al. (2019).

Below, we perform Fama and MacBeth (1973) regressions on our sample. We start by thr first pass: individual estimation of betas. We build a dedicated function below and use some functional programming to automate the process.

```
nb_factors <- 5                                                    # Number of factors
data_FM <- left_join(data_ml %>%                                   # Join the 2 datasets
                         select(date, stock_id, R1M_Usd) %>%       # (with returns...
                         filter(stock_id %in% stock_ids_short),    # ... over some stocks)
                     FF_factors,
                     by = "date") %>%
    mutate(R1M_Usd = lag(R1M_Usd)) %>%                             # Lag returns
    na.omit() %>%                                                  # Remove missing points
    spread(key = stock_id, value = R1M_Usd)
models <- lapply(paste0("`", stock_ids_short,
                        '` ~  MKT_RF + SMB + HML + RMW + CMA'),            # Model spec
                 function(f){ lm(as.formula(f), data = data_FM,            # Call lm(.)
                                 na.action="na.exclude") %>%
                         summary() %>%                                     # Gather the output
                         "$"(coef) %>%                                     # Keep only coefs
                         data.frame() %>%                                  # Convert to dataframe
                         select(Estimate)}                                 # Keep the estimates
                 )
betas <- matrix(unlist(models), ncol = nb_factors + 1, byrow = T) %>%      # Extract the betas
    data.frame(row.names = stock_ids_short)                                # Format: row names
colnames(betas) <- c("Constant", "MKT_RF", "SMB", "HML", "RMW", "CMA")     # Format: col names
knitr::kable(head(betas %>% round(3)),  booktabs = TRUE,
             caption = "Sample of beta values (row numbers are stock IDs).") # Betas (table)
```

[1]Originally, Fama and MacBeth (1973) work with the market beta only: $r_{t,n} = \alpha_n + \beta_n r_{t,M} + \epsilon_{t,n}$ and the second pass included nonlinear terms: $r_{t,n} = \gamma_{n,0} + \gamma_{t,1}\hat{\beta}_n + \gamma_{t,2}\hat{\beta}_n^2 + \gamma_{t,3}\hat{s}_n + \eta_{t,n}$, where the $\hat{s}_n$ are risk estimates for the assets that are not related to the betas. It is then possible to perform asset pricing tests to infer some properties. For instance, test whether betas have a linear influence on returns or not ($\mathbb{E}[\gamma_{t,2}] = 0$), or test the validity of the CAPM (which implies $\mathbb{E}[\gamma_{t,0}] = 0$).

TABLE 4.2: Sample of beta values (row numbers are stock IDs).

	Constant	MKT_RF	SMB	HML	RMW	CMA
1	0.008	1.438	0.525	0.567	1.006	-0.301
3	-0.002	0.791	1.085	0.854	0.220	-0.423
4	0.005	0.364	0.305	-0.072	0.589	0.267
7	0.006	0.414	0.682	0.284	0.294	0.119
9	0.005	0.811	0.683	1.046	0.009	0.148
11	0.000	0.937	0.120	0.494	-0.246	0.086

TABLE 4.3: Sample of reformatted beta values (ready for regression).

	MKT_RF	SMB	HML	RMW	CMA	2000-01-31	2000-02-29	2000-03-31
1	1.438	0.525	0.567	1.006	-0.301	-0.036	0.263	0.031
3	0.791	1.085	0.854	0.220	-0.423	0.077	-0.024	0.018
4	0.364	0.305	-0.072	0.589	0.267	-0.016	0.000	0.153
7	0.414	0.682	0.284	0.294	0.119	-0.009	0.027	0.000
9	0.811	0.683	1.046	0.009	0.148	0.032	0.076	-0.025
11	0.937	0.120	0.494	-0.246	0.086	0.144	0.258	0.049

We then reformat these betas to prepare the second pass. Each line corresponds to one asset: the first 5 columns are the estimated factor loadings and the remaining ones are the asset returns (date by date).

```r
loadings <- betas %>%                               # Start from loadings (betas)
    select(-Constant) %>%                           # Remove constant
    data.frame()                                    # Convert to dataframe
ret <- returns %>%                                  # Start from returns
    select(-date) %>%                               # Keep the returns only
    data.frame(row.names = returns$date) %>%        # Set row names
    t()                                             # Transpose
FM_data <- cbind(loadings, ret)                     # Aggregate both
knitr::kable(head(FM_data[,1:8] %>% round(3)),  booktabs = TRUE,    # The betas (see table)
             caption = "Sample of reformatted beta values (ready for regression).")
```

We observe that the values of the first column (market betas) revolve around one, which is what we would expect. Finally, we are ready for the second round of regressions.

```r
models <- lapply(paste("`", returns$date, "`", ' ~  MKT_RF + SMB + HML + RMW + CMA', sep = ""),
function(f){ lm(as.formula(f), data = FM_data) %>%                         # Call lm(.)
                         summary() %>%                                     # Gather the output
                         "$"(coef) %>%                                     # Keep only the coefs
                         data.frame() %>%                                  # Convert to dataframe
                         select(Estimate)}                                 # Keep only estimates
                 )
gammas <- matrix(unlist(models), ncol = nb_factors + 1, byrow = T) %>%     # Switch to dataframe
    data.frame(row.names = returns$date)                                   # & set row names
colnames(gammas) <- c("Constant", "MKT_RF", "SMB", "HML", "RMW", "CMA")    # Set col names
knitr::kable(head(gammas %>% round(3)),  booktabs = TRUE,    # The gammas (see table)
             caption = "Sample of gamma (premia) values.")
```

Visually, the estimated premia are also very volatile. We plot their estimated values for the market, SMB and HML factors.

TABLE 4.4: Sample of gamma (premia) values.

	Constant	MKT_RF	SMB	HML	RMW	CMA
2000-01-31	-0.031	0.037	0.227	-0.157	-0.276	0.044
2000-02-29	0.020	0.081	-0.134	0.050	0.089	-0.027
2000-03-31	0.007	-0.011	-0.016	0.054	0.036	0.039
2000-04-30	0.127	-0.132	-0.104	0.088	0.117	-0.002
2000-05-31	0.042	-0.005	0.075	-0.113	-0.080	-0.045
2000-06-30	0.028	-0.029	-0.019	0.054	0.045	0.017

```
gammas %>%                                                       # Take gammas:
    select(MKT_RF, SMB, HML) %>%                                 # Select 3 factors
    bind_cols(date = data_FM$date) %>%                           # Add date
    gather(key = factor, value = gamma, -date) %>%               # Put in tidy shape
    ggplot(aes(x = date, y = gamma, color = factor)) +           # Plot
    geom_line() + facet_grid( factor~. ) +                       # Lines & facets
    scale_color_manual(values=c("#F87E1F", "#0570EA", "#F81F40")) + # Colors
    coord_fixed(980)                                             # Fix x/y ratio
```

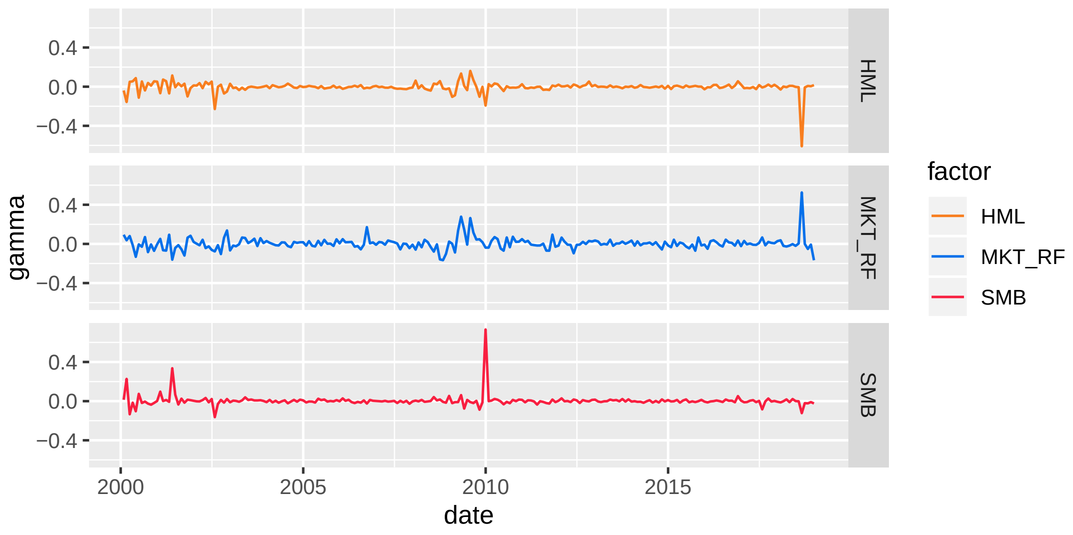


FIGURE 4.3: Time-series plot of gammas (premia) in Fama-Macbeth regressions

The two spikes at the end of the sample signal potential colinearity issues: two factors seem to compensate in an unclear aggregate effect. This underlines the usefulness of penalized estimates (see Chapter 6).

4.2.5 Factor competition

The core purpose of factors is to explain the cross-section of stock returns. For theoretical and practical reasons, it is preferable if redundancies within factors are avoided. Indeed, redundancies imply collinearity which is known to perturb estimates (Belsley et al. (2005)). In addition, when asset managers decompose the performance of their returns into factors, overlap between factors yield exposures that are less interpretable often because positive and negative exposures compensate each other spuriously.

A simple protocol to sort out redundant factors is to run regressions of each factor against all others:

$$f_{t,k} = a_k + \sum_{j \neq k} \delta_{k,j} f_{t,j} + \epsilon_{t,k}. \tag{4.5}$$

The interesting metric is then the test statistic associated to the estimation of a_k. If a_k is significantly different from zero, then the cross-section of (other) factors fails to explain exhaustively the average return of factor k. Otherwise, the return of the factor can be captured be exposures to the other factors and is thus redundant.

One mainstream application of this technique was performed in Fama and French (2015), in which the authors show that the HML factor is redundant when taking into account four other factors (Market, SMB, RMW and CMA). Below, we reproduce their analysis on an updated sample. We start our analysis directly with the database maintained by Kenneth French.

We can run the regressions that determine the redundancy of factors via the procedure defined in Equation (4.5).

```
factors <- c("MKT_RF", "SMB", "HML", "RMW", "CMA")
models <- lapply(paste(factors, ' ~  MKT_RF + SMB + HML + RMW + CMA-',factors),
 function(f){ lm(as.formula(f), data = FF_factors) %>%               # Call lm(.)
                         summary() %>%                               # Gather the output
                         "$"(coef) %>%                               # Keep only the coefs
                         data.frame() %>%                            # Convert to dataframe
                         filter(rownames(.) == "(Intercept)") %>%    # Keep only the Intercept
                         select(Estimate,`Pr...t..`)}                # Keep the coef & p-value
                 )
alphas <- matrix(unlist(models), ncol = 2, byrow = T) %>%       # Switch from list to dataframe
    data.frame(row.names = factors)
# alphas # To see the alphas (optional)
```

We obtain the vector of α values from equation (4.5). Below, we format these figures along with p-value thresholds and export them in a summary table. The significance levels of coefficients is coded as follows: $0 < (***) < 0.001 < (**) < 0.01 < (*) < 0.05$.

```
results <- matrix(NA, nrow = length(factors), ncol = length(factors) + 1)   # Coefs
signif  <- matrix(NA, nrow = length(factors), ncol = length(factors) + 1)   # p-values
for(j in 1:length(factors)){
    form <- paste(factors[j],
                  ' ~  MKT_RF + SMB + HML + RMW + CMA-',factors[j])         # Build model
    fit <- lm(form, data = FF_factors) %>% summary()                        # Estimate model
    coef <- fit$coefficients[,1]                                            # Keep coefficients
    p_val <- fit$coefficients[,4]                                           # Keep p-values
    results[j,-(j+1)] <- coef                                               # Fill matrix
    signif[j,-(j+1)] <- p_val
}
signif[is.na(signif)] <- 1                                                  # Kick out NAs
results <- results %>% round(3)  %>% data.frame()                           # Basic formatting
results[signif<0.001] <- paste(results[signif<0.001]," (***)")              # 3 star signif
results[signif>0.001&signif<0.01] <-                                        # 2 star signif
    paste(results[signif>0.001&signif<0.01]," (**)")
results[signif>0.01&signif<0.05] <-                                         # 1 star signif
    paste(results[signif>0.01&signif<0.05]," (*)")

results <- cbind(as.character(factors), results)                            # Add dep. variable
```

TABLE 4.5: Factor competition among the Fama French 2015 five factors. The sample starts in 1963-07 and ends in 2019-11. The regressions are run on monthly returns.

Dep. Variable	Intercept	MKT_RF	SMB	HML	RMW	CMA
MKT_RF	0.008 (***)	NA	0.252 (***)	0.095	-0.378 (***)	-0.916 (***)
SMB	0.003 (**)	0.13 (***)	NA	0.066	-0.434 (***)	-0.128
HML	0	0.026	0.035	NA	0.143 (***)	1.011 (***)
RMW	0.004 (***)	-0.099 (***)	-0.221 (***)	0.137 (***)	NA	-0.291 (***)
CMA	0.002 (***)	-0.112 (***)	-0.03	0.453 (***)	-0.136 (***)	NA

```r
colnames(results) <- c("Dep. Variable","Intercept", factors)              # Add column names
knitr::kable(results,  booktabs = TRUE,
             caption = paste0("Factor competition among the Fama French 2015 five factors. ",
                              "The sample starts in ",substr(min_date,1,7),
                              " and ends in ", substr(max_date,1,7),
                              ". The regressions are run on monthly returns. "
                      )
      ) %>%
    kable_styling(font_size = 9)
```

We confirm that the HML factor remains redundant when the four other are present in the asset pricing model. The figures we obtain are very close to the ones in the original paper (Fama and French (2015)), which makes sense, since we only add 5 years to their initial sample.

At a more macro-level, researchers also try to figure out which models (i.e., combinations of factors) are the most likely, given the data empirically observed (and possibly given priors formulated by the econometrician). For instance, this stream of literature seeks to quantify to which extent the 3 factor model of Fama and French (1993) outperforms the 5 factors in Fama and French (2015). In this direction, De Moor et al. (2015) introduce a novel computation for p-values that compare the relative likelihood that two models pass a zero-alpha test. More generally, the Bayesian method of Barillas and Shanken (2018) was subsequently improved by Chib et al. (2019).

4.2.6 Advanced techniques

The ever increasing number of factors combined to their importance in asset management has led researchers to craft more subtle methods in order to "organise" the so-called *factor zoo* and more importantly, to detect spurious anomalies and compare different asset pricing model specifications. We list a few of them below.

- Feng et al. (2019a) combine LASSO selection with Fama-MacBeth regressions to test if new factor models are worth it. They quantify the gain of adding one new factor to a set of predefined factors and show that many factors reported in paper published in the 2010 decade do not add much incremental value;

- Harvey and Liu (2017) (in a similar vein) use bootstrap on orthogonalised factors. They make the case that correlations among predictors is a major issue and their method aims at solving this problem. Their lengthy procedure seeks to test if maximal additional contribution of a candidate variable is significant;

- Fama and French (2018) compare asset pricing models through squared maximum Sharpe ratios;
- Giglio and Xiu (2018) estimate factor risk premia using a three-pass method based on principal component analysis;
- Pukthuanthong et al. (2018) disentangle priced and non-priced factors via a combination of principal component analysis and Fama and MacBeth (1973) regressions.
- Gospodinov et al. (2019) warn against factor misspecification (when spurious factors are included in the list of regressors). Traded factors (*resp.* macro-economic factors) seem more likely (*resp.* less likely) to yield robust identifications (see also Bryzgalova (2019)).

There is obviously no infaillible method, but the number of contributions in the field highlights the need for robustness. This is evidently a major concern when crafting investment decisions based on factor intuitions. One major hurdle for short-term strategies is the likely time-varying feature of factors. We refer for instance to Ang and Kristensen (2012) and Cooper and Maio (2019) for practical results and to Gagliardini et al. (2016) and Ma et al. (2018)) for more theoretical treatments (with additional empirical results).

4.3 Factors or characteristics?

The decomposition of returns into linear factor models is convenient because of its simple interpretation. There is nonetheless a debate in the academic literature about whether firm returns are indeed explained by exposure to macro-economic factors or simply by the characteristics of firms. In their early study, Lakonishok et al. (1994) argue that one explanation of the value premium comes from incorrect extrapolation of past earning growth rates. Investors are overly optimistic about firms subject to recent profitability. Consequently, future returns are (also) driven by the core (accounting) features of the firm. The question is then to disentangle which effect is the most pronounced when explaining returns: characteristics versus exposures to macro-economic factors?

In their seminal contribution on this topic, Daniel and Titman (1997) provide evidence in favour of the former (two follow-up papers are Daniel et al. (2001a) and Daniel and Titman (2012)). They show that firms with high book-to-market ratios or small capitalisations display higher average returns, even if they are negatively loaded on the HML or SMB factors. Therefore, it seems that it is indeed the intrinsic characteristics that matter, and not the factor exposure. For further material on characteristics' role in return explanation or prediction, we refer to the following contributions:

- Section 2.5.2. in Goyal (2012) surveys pre-2010 results on this topic;
- Chordia et al. (2015) find that characteristics explain a larger proportion of variation in estimated expected returns than factor loadings;
- Kozak et al. (2018) reconcile factor-based explanations of premia to a theoretical model in which some agents' demands are sentiment driven;

- Han et al. (2019) show with penalised regressions that 20 to 30 characteristics (out of 94) are useful for the prediction of monthly returns of US stocks. Their methodology is interesting: they regress returns against characteristics to build forecasts and then regress the returns on the forecast to assess if the forecasts are reliable. The latter regression uses a LASSO-type penalization (see Chapter 6) so that useless characteristics are excluded from the model. The penalization is extended to elasticnet in Rapach and Zhou (2019).

- both Kelly et al. (2019) and Kim et al. (2019) estimate models in which *factors* are *latent* but loadings (betas) and possibly alphas depend on characteristics. In contrast, Lettau and Pelger (1XXX) and Lettau and Pelger (2018) estimate latent factors without any link to particular characteristics (and provide large sample asymptotic properties of their methods).

- in the same vein as Hoechle et al. (2018), Gospodinov et al. (2019) and Bryzgalova (2019) and discuss potential errors that arise when working with portfolio sorts that yield long-short returns. The authors show that in some cases, tests based on this procedure may be deceitful. This happens when the characteristic chosen to perform the sort is correlated with an external (unobservable) factor. They propose a novel regression-based approach aimed at bypassing this problem.

More recently and in a separate stream of literature, Koijen and Yogo (2019) have introduced a demand model in which investors for their portfolios according to their preferences towards particular firm characteristics. They show that this allows them to mimic the portfolios of large institutional investors. In their model, aggregate demands (and hence, prices) are directly linked to characteristics, not to factors. In a follow-up paper, Koijen et al. (2019) show that a few set of characteristics suffice to predict future returns. They also show that, based on institutional holdings from the UK and the US, the largest investors are those who are the most influencial in the formation of prices. In a similar vein, Betermier et al. (2019) derive an elegant (theoretical) general equilibrium model that generates some well documented anomalies (size, book-to-market). Finally, in Martin and Nagel (2019), characteristics influence returns via the role they play in the predictability of dividend growth. This paper discussed the asymptotic case when the number of assets and the number of characteristics are proportional and both increase to infinity.

4.4 The link with machine learning

Given the exponential increase in data availability, the obvious temptation of any asset manager is to try to infer future returns from the abundance of attributes available at the firm level. We allude to classical data like accounting ratios and to alternative data, like sentiment. This task is precisely the aim of Machine Learning. Given a large set of predictor variables ($\mathbf{X}$), the goal is to predict a proxy for future performance $\mathbf{y}$ through a model of the form (3.1).

Some attempts toward this direction have already been made (e.g., Brandt et al. (2009), Ammann et al. (2016), Martin Utrera et al. (2018)), but not with any ML intent or focus.

4.4.1 A short list of recent references

Independently of a characteristics-based approach, ML applications in Finance have blossomed, initially working with price data only and later on integrating firm characteristics as predictors. We cite a few references below, grouped by methodological approach:

- penalised quadratic programming: Goto and Xu (2015), Ban et al. (2016) and Perrin and Roncalli (2019),
- regularised predictive regressions: Rapach et al. (2013) and Chinco et al. (2019a),
- support vector machines: Cao and Tay (2003) (and the references therein),
- model comparison and/or aggregation: Kim (2003), Huang et al. (2005), Matías and Reboredo (2012), Reboredo et al. (2012), Dunis et al. (2013), Gu et al. (2018) and Guida and Coqueret (2018b). The latter two more recent articles work with a large cross-section of characteristics.

We provide more detailed lists for tree-based methods, neural networks and reinforcement learning techniques in Chapters 7, 8 and 17, respectively. Moreover, we refer to Ballings et al. (2015) for a comparison of classifiers and to Henrique et al. (2019) for a survey on ML-based forecasting techniques.

4.4.2 Explicit connexions with asset pricing models

The first and obvious link between factor investing and asset pricing is (average) return prediction. The main canonical academic reference is Gu et al. (2018). Let us first write the general equation and then comment on it:

$$r_{t+1,n} = g(\mathbf{x}_{t,n}) + \epsilon_{t+1}. \tag{4.6}$$

The interesting discussion lies in the differences between the above model and that of Equation (4.1). The first obvious difference is the introduction of the nonlinear function g: indeed, there is no reason (beyond simplicity and interpretability) why we should restrict the model to linear relationships. One early reference for nonlinearities in asset pricing kernels is Bansal and Viswanathan (1993).

More importantly, the second difference between (4.6) and (4.1) is the shift in the time index. Indeed, from an investor's perspective, the interest is to be able to *predict* some information about the structure of the cross-section of assets. Explaining asset returns with synchronous factors is not useful because the realization of factor values are not known in advance. Hence, if one seeks to extract value from the model, there needs to be a time interval between the observation of the state space (which we call $\mathbf{x}_{t,n}$) and the occurrence of the returns. Once the model $\hat{g}$ is estimated, the time-t (measurable) value $g(\mathbf{x}_{t,n})$ will give a forecast for the (average) future returns. These predictions can then serve as signal in the crafting of portfolio weights (see Chapter 13 for more on that topic).

While most studies do work with returns on the l.h.s. of (4.6), there is no reason why other indicators should not be used. Returns are straightforward and simple to compute, but they could very well be replaced by more sophisticated metrics, like the Sharpe ratio, for instance. The firms' features would then be used to predict a risk-adjusted performance rather than simple returns.

Beyond the explicit form of Equation (4.6), several other ML-related tools can also be used to estimate asset pricing models. This can be achieved in several ways, some of which we list below.

First, one mainstream problems in asset pricing is to characterize the stochastic discount factor (SDF) M_t, which satisfies $\mathbb{E}_t[M_{t+1}(r_{t+1,n} - r_{t+1,f})] = 0$ for any asset n (see Cochrane (2009)). This equation is a natural playing field for the generalized method of moment (Hansen (1982)): M_t must be such that

$$\mathbb{E}[M_{t+1}R_{t+1,n}g(V_t)] = 0, \tag{4.7}$$

where the instrumental variables V_t are $\mathcal{F}_t$-measurable (i.e., are known at time t) and the capital $R_{t+1,n}$ denotes the excess return of asset n. In order to reduce and simplify the estimation problem, it is customary to define the SDF as a portfolio of assets (see Chapter 3 in Back (2010)). In Chen et al. (2019), the authors use a generative adversarial network (GAN, see Section 8.6.1) to estimate the weights of the portfolios that are the closest to satisfy (4.7) under a strongly penalizing form.

A second approach is to try to model asset returns as linear combinations of factors, just as in (4.1). We write in compact notation

$$r_{t,n} = \alpha_n + \boldsymbol{\beta}_{t,n}'\mathbf{f}_t + \epsilon_{t,n},$$

and we allow the loadings $\boldsymbol{\beta}_{t,n}$ to be time-dependent. The trick is then to introduce the firm characteristics in the above equation. Traditionally, the characteristics are present in the definition of factors (as in the seminal definition of Fama and French (1993)). The decomposition of the return is made according to the exposition of the firm's return to these factors constructed according to market size, accounting ratios, past performance etc. Given the exposures, the performance of the stock is attributed to particular style profiles (e.g., small stock, or value stock, etc.).

Habitually, the factors are heuristic portfolios constructed from simple rules like thresholding. For instance, firms below the 1/3 quantile in book-to-market are growth firms and those above the 2/3 quantile are the value firms. A value factor can then be defined by the long-short portfolio of these two sets, with uniform weights. Note that Fama and French (1993) use a more complex approach which also take market capitalization into account both in the weighting scheme and also in the composition of the portfolios.

One of the advances enabled by machine learning is to automate the construction of the factors. It is for instance the approach of Feng et al. (2019b). Instead of building the factors heuristically, the authors optimize the construction to maximize the fit in the cross-section of returns. The optimization is performed via a relatively deep feed-forward neural network and the feature space is lagged so that the relationship is indeed predictive, as in Equation (4.6). Theoretically, the resulting factors help explain a substantially larger porportion of the in-sample variance in the returns. The prediction ability of the model depends on how well it generalizes out-of-sample.

A third approach is that of Kelly et al. (2019) (though the statistical treatment is not machine learning per se).[2] Their idea is the opposite: factors are latent (unobserved) and it is the betas (loadings) that depend on the characteristics. This allows many degrees of freedom because in $r_{t,n} = \alpha_n + (\boldsymbol{\beta}_{t,n}(\mathbf{x}_{t-1,n}))'\mathbf{f}_t + \epsilon_{t,n}$, only the characteristics $\mathbf{x}_{t-1,n}$ are known and both the factors $\mathbf{f}_t$ and the functional forms $\boldsymbol{\beta}_{t,n}(\cdot)$ must be estimated. In their article, Kelly et al. (2019) work with a linear form, which is naturally more tractable.

Lastly, a fourth approach (introduced in Gu et al. (2019)) goes even further and combines two neural network architectures. The first neural network takes characteristics $\mathbf{x}_{t-1}$ as

[2] In the same spirit, see also Lettau and Pelger (1XXX) and Lettau and Pelger (2018).

inputs and generates factor loadings $\boldsymbol{\beta}_{t-1}(\mathbf{x}_{t-1})$. The second network transforms returns $\mathbf{r}_t$ into factor values $\mathbf{f}_t(\mathbf{r}_t)$ (in Feng et al. (2019b)). The aggregate model can then be written:

$$\mathbf{r}_t = \boldsymbol{\beta}_{t-1}(\mathbf{x}_{t-1})'\mathbf{f}_t(\mathbf{r}_t) + \boldsymbol{\epsilon}_t. \tag{4.8}$$

The above specification is quite special because the output (on the left hand side) is also present as input (in the right hand side). In machine learning, autoencoders (see Section 8.6.2) share the same property. Their aim, just like in principal component analysis, is to find a parsimonious nonlinear representation form for a dataset (in this case: returns). In Equation (4.8), the input is $\mathbf{r}_t$ and the output function is $\boldsymbol{\beta}_{t-1}(\mathbf{x}_{t-1})'\mathbf{f}_t(\mathbf{r}_t)$. The aim is to minimize the difference between the two just as is any regression-like model.

Autoencoders are neural networks which have outputs as close as possible to the inputs with an objective of dimensional reduction. The innovation in Gu et al. (2019) is that the pure autoencoder part is merged with a vanilla perceptron used to model the loadings. The structure of the neural network is summarized below.

$$\left.\begin{array}{rcl} \text{returns } (\mathbf{r}_t) & \overset{NN_1}{\longrightarrow} & \text{factors } (\mathbf{f}_t = NN_1(\mathbf{r}_t)) \\ \text{characteristics } (\mathbf{x}_{t-1}) & \overset{NN_2}{\longrightarrow} & \text{loadings } (\boldsymbol{\beta}_{t-1} = NN_2(\mathbf{x}_{t-1})) \end{array}\right\} \longrightarrow \text{returns } (r_t)$$

A simple autoencoder would consist in only the first line of the model.

As a conclusion of this chapter, it appears undeniable that the intersection between the two fields of asset pricing and machine learning offers a rich variety of applications. The literature is already exhaustive and it is often hard to disentangle the noise from the great ideas in the continuous flow of publications on these topics. Practice and implementation is the only way forward to extricate value from hype.

4.5 Coding exercises

1. Compute annual returns of the growth versus value portfolios, that is, the average return of firms with above median price-to-book ratio (Pb variable in the dataset).

2. Same exercise, but compute the monthly returns and plot the value (through time) of the corresponding portfolios.

3. Instead of a unique threshold, compute simply sorted portfolios based on quartiles of market capitalization.

5

Data preprocessing

The methods we describe in this chapter are driven by financial applications. For an introduction to non-financial data processing, we recommend two references: Chapter 3 from the general purpose ML book Boehmke and Greenwell (2019) and the monograph on this dedicated subject: Kuhn and Johnson (2019).

5.1 Know your data

The first step, as in any quantitative study, is obviously to make sure the data is trustworthy, i.e., comes from a reliable provider. The landscape in financial data provision is vast to say the least: some providers are well established (e.g., Bloomberg, Thomson-Reuters, Datastream, CRSP, Morningstar), some are more recent (e.g., Capital IQ, Ravenpack) and some focus on alternative data niches (see `https://alternativedata.org/data-providers/` for an exhaustive list). Unfortunately, and to the best of our knowledge, no study has been published that evaluates a large spectrum of these providers in terms of data reliability.

The second step is to have a look at **summary statistics**: ranges (minimum and maximum values), and averages and medians. Histograms or plots of time-series carry of course more information but cannot be analyzed properly in high dimensions. They are nonetheless sometimes useful to track local patterns or errors for a given stock and a particular feature. Beyond first order moments, second order quantities (variances and covariances/correlations) also matter because they help spot colinearities. When two features are highly correlated, problems may arise in some models (e.g., simple regressions, see Section 16.1).

Often, the number of predictors is so large that it is unpractical to look at these simple metrics. A minimal verification is recommended. To further ease the analysis:

- focus on a subset of predictors, e.g., the ones linked to the most common factors (market-capitalization, price-to-book or book-to-market, momentum (past returns), profitability, asset growth, volatility);
- track outliers in the summary statistics (when the maximum/median or median/minimum ratios seem suspicious).

More importantly, when seeking to work with supervised learning (as we will do most of the time), the link of some features with the dependent variable can be further characterized by the smoothed **conditional average** because it shows how the features impact the label. The use of the conditional average has a deep theoretical grounding. Suppose there is only one feature X and that we seek a model $Y = f(X) + \text{error}$, where variables are real-valued. The function f that minimizes the average squared error $\mathbb{E}[(Y - f(X))^2]$ is the so-called regression function (see Section 2.4 in Hastie et al. (2009)):

$$f(x) = \mathbb{E}[Y|X = x]. \tag{5.1}$$

In Figure 5.1, we plot two illustrations of this function when the dependent variable is the one month ahead return. The first one pertains to the average market capitalization over the past year and the second to the volatility over the past year as well. Both predictors have been uniformized (see Section 5.4.2 below) so that their values are uniformly distributed in the cross-section of assets for any given time period. Thus, the range of features is $[0, 1]$ and is shown on the x-axis of the plot. The grey corridors around the lines show 95% level confidence interval for the computation of the mean. Essentially, it is narrow when both i) many data points are available and ii) these points are not too dispersed.

```
data_ml %>%                                                     # From dataset:
  ggplot(aes(y = R1M_Usd)) +                                    # Plot
  geom_smooth(aes(x = Mkt_Cap_12M_Usd, color = "Market Cap")) + # Cond. Exp. Mkt_cap
  geom_smooth(aes(x = Vol1Y_Usd, color = "Volatility")) +       # Cond. Exp. Vol
  scale_color_manual(values=c("#F87E1F", "#0570EA")) +          # Change color
  coord_fixed(10)                                               # Change x/y ratio
```

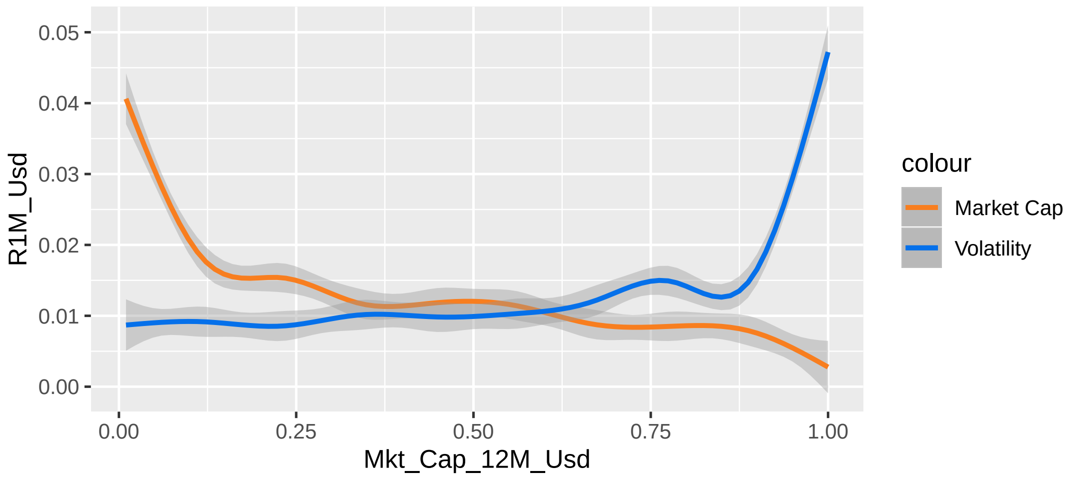


FIGURE 5.1: Conditional expecations: average returns as smooth functions of features.

The two variables have a close to monotonic impact on future returns. Returns, on average, decrease with market capitalization (thereby corroborating the so-called *size* effect). The reverse pattern is less pronounced for volatility: the curve is rather flat is the first half of volatility scores and progressively increases, especially over the last quintile of volatility values (thereby contradicting the low-vol anomaly).

5.2 Missing data

Similarly to any empirical discipline, portfolio management is bound to face missing data issues. The topic is well known and several books detail solutions to this problem (e.g., Allison (2001), Enders (2010), Little and Rubin (2014) and Van Buuren (2018)). While researchers continuously propose new methods to cope with absent points (Honaker and King (2010) or Che et al. (2018) to cite but a few), we believe that a simple, heuristic, treatment is usually sufficient as long as some basic cautious safeguards are enforced.

First of all, there are mainly two ways to deal with missing data: removal and imputation. Removal is agnostic but costly, especially if one whole instance is eliminated because of only one missing feature. Imputation is often prefered but relies on some underlying and potentially erroneous assumption.

A simplified classification of imputation is the following:

- A basic imputation choice is the median (or mean) of the feature for the stock over the past available values. If there is a trend in the time series, this will nonetheless alter the trend. Relatedly, this method is forward looking.

- In time-series contexts with views towards backtesting, the most simple imputation comes from previous values: if x_t is missing, replace it with x_{t-1}. This makes sense most of the time because past values are all that is available and are by definition backward looking. However, in some particular cases, this be a very bad choice (see words of caution below).
- Medians and means can also be computed over the cross-section of assets. This roughly implies that the missing feature value will be relocated in the bulk of observed values. When many values are missing, this creates an atom in the distribution of the feature and alters the original distribution. One advantage is that this imputation is not forward looking.

- Many techniques rely on some modelling assumptions for the data generating process. We refer to nonparametric approaches (Stekhoven and Bühlmann (2011) and Shah et al. (2014), which both rely on random forests, see Chapter 7), Bayesian imputation (Schafer (1999)), maximum likelihood approaches (Enders (2001), Enders (2010)), interpolation or extrapolation and nearest neighbor algorithms (García-Laencina et al. (2009)). More generally, the four books cited at the begining of the subsection detail many such imputation processes. Advanced techniques are much more demanding computationally.

A few words of caution:

- Interpolation should be avoided at all cost. Accounting values or ratios that are released every quarter must never be linearly interpolated for the simple reason that this is forward looking. If numbers are disclosed in January and April, then interpolating February and March requires the knowledge of the April figure, which, in live trading will not be known. Resorting to past values is a better way to go.

- Nevertheless, there are some feature types for which imputation from past values should be avoided. First of all, returns should not be replicated. By default, a superior choice is to set missing return indicators to zero (which is often close to the average or the median). A good indicator that can help the decision is the persistence of the feature through time. If it is highly autocorrelated (and the time-series plot create a smooth curve, like for marjet capitalization), then imputation from the past can make sense. If not, then it should be avoided.

- There are some cases that can require more attention. Let us consider the following fictitious sample of dividend yield:

Date	Original yield	Replacement value
2015-02	NA	(preceding (if it exists)

Date	Original yield	Replacement value
2015-03	0.02	untouched (none)
2015-04	NA	0.02 (previous)
2015-05	NA	0.02 (previous)
2015-06	NA	$\leftarrow$ **Problem**!

In this case, the yield is released quarterly, in March, June, September, etc. But in June, the value is missing. The problem is that we cannot know if it is missing because of a genuine data glitch, or because the firm simply did not pay any dividends in June. Thus, imputation from past value may be erroneous here. There is no perfect solution but a decision must nevertheless be taken. For dividend data, three options are:
1) Keep the previous value.
2) Extrapolate from previous observations (this is very different from **inter**polation): for instance, evaluate a trend and pursue that trend. 3) Set the value to zero. This is tempting but may be sub-optimal due to dividend smoothing practices from executives (see for instance Leary and Michaely (2011) and Chen et al. (2012) for details on the subject). For persistent time-series, the first two options are probably better.

Tests can be perform to evaluate the relative performance of each option. It is also important to **remember** these design choices. There are so many of them that they are easy to forget. Keeping track of them is obviously compulsory. In the ML pipeline, the scripts pertaining to data preparation are often key because they do not serve only once!

5.3 Outlier detection

The topic of outlier detection is also well documented and has its own surveys (Hodge and Austin (2004), Chandola et al. (2009) and Gupta et al. (2014)) and a few dedicated books (Aggarwal (2013) and Rousseeuw and Leroy (2005), though the latter is very focused on regression analysis).

Again, incredibly sophisticated methods may require a lot of effort for possibly limited gain. Simple heuristic methods, as long as they are documented in the process may suffice. They often rely on 'hard' thresholds:

- for one given feature (possibly filtered in time), any point outside the interval $[\mu-m\sigma,\mu+m\sigma]$ can be deemed an outlier. Here μ is the mean of the sample and σ the standard deviation. The multiple value m usually belongs to the set $\{3,5,10\}$, which is of course arbitrary.
- likewise, if the largest value is above m times the second-to-largest, then is can also be classified as an outlier (the same reasoning applied for the other side of the tail).
- finally, for a given small threshold q, any value outside the $[q,1-q]$ quantile range can be considered outliers.

This latter idea was popularized by winsorization. Winsorizing amounts to setting to $x^{(q)}$ all values below $x^{(q)}$ and to $x^{(1-q)}$ all values above $x^{(1-q)}$. The winsorised variable $\tilde{x}$ is:

$$\tilde{x}_i=\left\{ \begin{array}{lll} x_i & \text{if } x_i \in [x^{(q)},x^{(1-q)}] & \text{(unchanged)} \\ x^{(q)} & \text{if } x_i < x^{(q)} & \\ x^{(1-q)} & \text{if } x_i > x^{(1-q)} & \end{array} \right. .$$

The range for q is usually $(0.5\%, 5\%)$ with 1% and 2% being the most often used.

The winsorization stage must be performed on a feature-by-feature and a data-by-date basis. However, keeping a time-series perspective is also useful. For instance, a 800B$ market capitalization may seems out of range, except when looking at the history of Apple's capitalization.

We conclude this subsection by recalling that *true* outliers (i.e, extreme points that are not due to data extraction errors) are valuable because they are likely to carry important information.

5.4 Feature engineering

Feature engineering is a very important step of the portfolio construction process. Computer scientistic often refer to the saying "garbage in, garbage out". It is thus paramount to prevent the ML engine of the allocation to be trained on ill-designed variables. We invite the interested reader to have a look at the recent work of Kuhn and Johnson (2019) on this topic. The (shorter) academic reference is Guyon and Elisseeff (2003).

5.4.1 Feature selection

The first step is selection. Given a large set of predictors, it seems a sound idea to filter out unwanted or redundant exogenous variables. Heuristically, simple methods include:

- computing the correlation matrix of all features and making sure that no (absolute) value is above a threshold (0.7 is a common value) so that redundant variables do not pollute the learning engine;

- carrying out a linear regression and removing the non significant variables (e.g., those with p-value above 0.05).

Both these methods are somewhat reductive and overlook nonlinear relationships. Another approach would be to fit a decision tree (or a random forest) and retain only the features that have a high variable importance. These topics will be developed in Chapter 7.

5.4.2 Scaling the predictors

The premise of the need to pre-process the data comes from the large variety of scales in financial data:

- returns are most of the time smaller than one in absolute value;
- stock volatility lies usually between 5% and 80%;
- market capitalisation is expressed in million or billion units of a particular currency;
- accounting values as well;
- accounting ratios have inhomogeneous units;
- synthetic attributes like sentiment also have their idiosyncrasies.

While it is widely considered that monotonic transformation of the features have a marginal impact on prediction outcomes, Galili and Meilijson (2016) show that this is not always the case (see also section 5.8.2). Hence, the choice of normalisation may in fact very well matter.

If we write x_i for the raw input and $\tilde{x}_i$ for the transformed data, common scaling practices include:

- **standardization**: $\tilde{x}_i = (x_i - m_x)/\sigma_x$, where m_x and σ_x are the mean and standard deviation of x, respectively;
- **min-max** rescaling over [0,1]: $\tilde{x}_i = (x_i - \min(\mathbf{x}))/(\max(\mathbf{x}) - \min(\mathbf{x}))$;
- **min-max** rescaling over [-1,1]: $\tilde{x}_i = 2\frac{x_i - \min(\mathbf{x})}{\max(\mathbf{x}) - \min(\mathbf{x})} - 1$;
- **uniformization**: $\tilde{x}_i = F_{\mathbf{x}}(x_i)$, where $F_{\mathbf{x}}$ is the empirical c.d.f. of $\mathbf{x}$. In this case, the vector $\tilde{\mathbf{x}}$ is defined to follow a uniform distribution over [0,1].

Sometimes, it is possible to apply a logarithmic transform of variables with both large values (market capitalization) and large outliers. The scaling can come after this transformation. Obviously, this technique is prohibited for features with negative values.

It is often advised to scale inputs so that they range in [0,1] before sending them through the training of neural networks for instance. The dataset that we use in this book is based on variables that have been uniformized. In factor investing, the scaling of features must be **operated separately for each date and each feature**. This point is critical. It makes sure that for every rebalancing date, the predictors will have a similar shape and do carry information on the cross-section of stocks.

Scaling features across dates should be proscribed. Take for example the case of market capitalization. On the long run (market crashes notwithstanding), this feature increases through time. Thus, scaling across dates, would lead to small values at the beginning of the sample and large values at the end of the sample. This would completely alter and dilute the cross-sectional content of the features.

5.5 Labelling

5.5.1 Simple labels

There are several ways to define labels when constructing portfolio policies. Of course, the finality is the portfolio weight, but it is rarely considered as the best choice for the label.[1]

Usual labels in factor investing are the following:

- raw asset returns;
- future relative returns (versus some benchmark: market-wide, or sector-based for instance);
- the probability of positive return (or of return above a specified threshold);
- the probability of outperforming a benchmark;

[1] Some methodologies do map firm attributes into final weights, e.g., Brandt et al. (2009) and Ammann et al. (2016) but these are outside the scope of the book.

- the binary version of the above: YES (outperforming) versus NO (underperforming);
- risk-adjusted versions of the above: Sharpe ratios, information ratios (see Section 13.3).

As we will discuss later in this chapter, these choices still leave room for additional degrees of freedom. Should the labels be rescaled, just like features are processed? What is the best time horizon on which to compute performance metrics?

5.5.2 Categorical labels

In a typical ML analysis, when y is a proxy for future performance, the ML engine will try to minimize some distance between the predicted value and the realized values. For mathematical convenience, the sum of squared error (L^2 norm) is used because it has the simplest derivative and makes gradient descent accessible and easy to compute.

Sometimes, it can be interesting not to focus on raw performance proxies, like returns or Sharpe ratios, but on investment decisions - which can be derived from these proxies. A simple example (decision rule) is the following:

$$y_{t,i}=\left\{ \begin{array}{rll} -1 & \text{if} & \hat{r}_{t,i} < r_- \\ 0 & \text{if} & \hat{r}_{t,i} \in [r,r_+] \\ +1 & \text{if} & \hat{r}_{t,i} > r_+ \end{array} \right. ,$$

where $\hat{r}_{t,i}$ is the performance proxy and $r_\pm$ are the decision thresholds. When the predicted performance is below r_-, the decision is -1 (e.g., *sell*), when it is above r_+, the decision is +1 (e.g., *buy*) and when it is in the middle (the model is neither very optimistic nor very pessimistic), then the decision is neutral (e.g., *hold*). The performance proxy can of course be relative to some benchmark so that the decision is directly related to this benchmark. The thresholds $r_\pm$ should be chosen such that the three categories are relatively balanced, that is, have a comparable number of instances.

In this case, the final output can be both considered as categorical or numerical because it belongs to an important subgroup of categorical variables: the ordered categorical (**ordinal**) variables. If y is taken as a number, then the usual regression tools apply. The transformation of the initial output into a new format is similar to what is performed when engineering features.

When y is treated as non-ordered (**nominal**) categorical variable, then a new layer of processing is required because ML tools only work with numbers. Hence, the categories must be recoded into digits. The mapping that is most often used is called '**one-hot encoding**'. The vector of classes is split in a sparse matrix in which each column is dedicated to one class. The matrix is filled with zeros and ones. A one is allocated to the column corresponding to the class of the instance. We provide a simple illustration in the table below.

Initial data		One-hot encoding	
Position	Sell	Hold	Buy
buy	0	0	1
buy	0	0	1
hold	0	1	0
sell	1	0	0
buy	0	0	1

In classification tasks, the output has a larger dimension. For each instance, it gives the probability of belonging to each class assigned by the model. As we will see in Chapters 7 and 8, this is easily handled via the softmax function.

5.5.3 The triple barrier method

We conclude this section with an advanced labelling technique mentioned in De Prado (2018). The idea is to consider the full dynamics of a trading strategy and not a simple performance proxy. The rationale for this extension is that often money managers implement P&L triggers that cash in when gains are sufficient or opt out to stop their losses. Upon inception of the strategy, three barriers are fixed (see Figure 5.2):

- one above the current level of the asset (majenta line), which measures a reasonable expected profit;
- one below the current level of the asset (cyan line), which acts as a stop-loss signal to prevent large negative returns;
- and finally, one that fixes the horizon of the strategy after which it will be terminated (black line).

If the strategy hits the first (*resp.* second) barrier, the output is +1 (*resp.* -1) and if it hits the last barrier, the output is equal to zero or to some linear interpolation (between -1 and +1) that represents the position of the terminal value relative to the two horizontal barriers. Computationally, this method is **much** more demanding as it evaluates a whole trajectory for each instance. It is nonetheless considered as more realistic because trading strategies are often accompanied with automatic triggers such as stop-loss, etc.

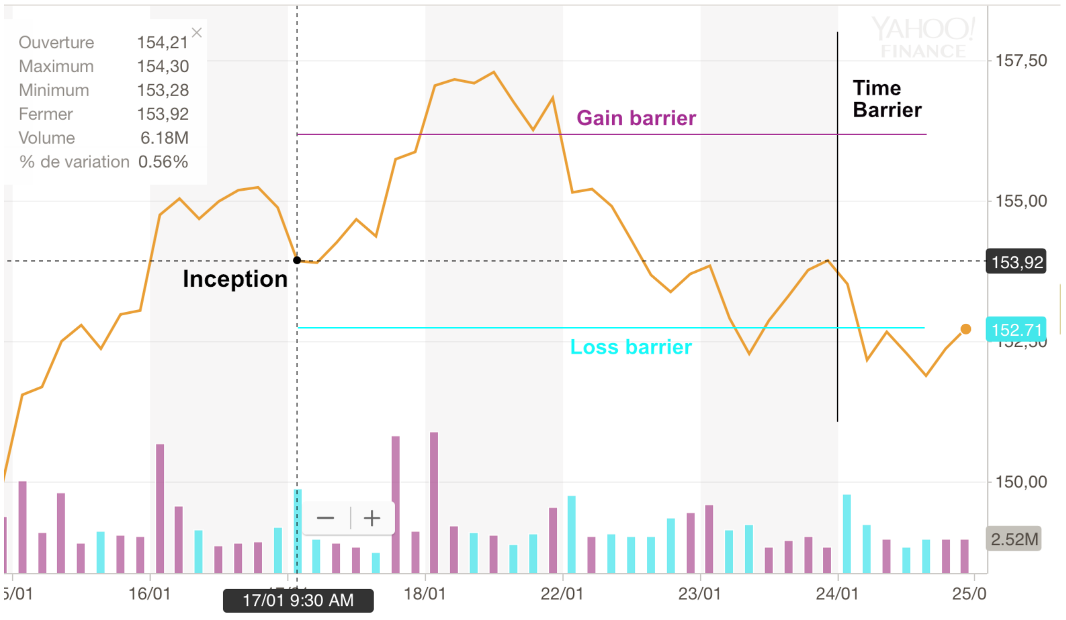


FIGURE 5.2: Illustration of the triple barrier method.

5.5.4 Filtering the sample

One of the main challenges in Machine Learning is to extract as much **signal** as possible. By signal, we mean patterns that will hold out-of-sample. Intuitively, it may seem reasonable to think that the more data we gather, the more signal we can extract. This is in fact false in all generality because more data also means more noise. Surprisingly, filtering the training samples can improve performance. This idea was for example implemented successfully in Fu et al. (2018), Guida and Coqueret (2018a) and Guida and Coqueret (2018b).

In our paper Coqueret and Guida (2019), we investigate why smaller samples may lead to superior out-of-sample accuracy for a particular type of ML algorithm: decision trees (see Chapter 7). We focus on a particular kind of filter: we exclude the labels (i.e., returns) that are not extreme and retain the 20% values that are the smallest and the 20% that are the largest (the bulk of the distribution is removed). In doing so, we alter the structure of trees in two ways:
- when the splitting points are altered, they are always closer to the center of the distribution of the splitting variable (i.e., the resulting clusters are more balanced);
- the choice of splitting variables is (sometimes) pushed towards the features that have a monotonous impact on the label.

These two properties are desirable. The first reduces the risk of fitting to small groups of instances that may be spurious. The second gives more importance to features that appear globally more relevant in explaning the returns. However, the filtering must not be too intense. If, instead of retaining 20% of each tail of the predictor, we keep just 10%, then the loss in signal becomes too severe and the performance deteriorates.

5.5.5 Return horizons

This subsection deals with one of the least debated issue in factor-based machine learning models.

Jegadeesh and Titman (1993) HORIZONS ??????? XXXX

5.6 Discussion on persistence

While we have separated the steps of feature engineering and labelling in two different subsections, it is probably wiser to consider them jointly. One important property of the dataset processed by the ML algorithm should be the consistency of persistence between features and labels. Intuitively, the autocorrelation patterns between the label $y_{t,n}$ (future performance) and the features $x_{t,n}^{(k)}$ should not be too distant.

One problematic example is when the dataset is sampled at the monthly frequency (not unusual in the money management industry) with the labels being monthly returns and the features being risk-based or fundamental attributes. In this case, the label is very weakly autocorrelated, while the features are often highly autocorrelated. In this situation, most sophisticated forecasting tools will arbitrage between features which will probably result in a lot of noise. In linear predictive models, this configuration is known to generate bias in estimates (see the study of Stambaugh (1999) and the review by Gonzalo and Pitarakis (2018)).

Among other more technical options, there are two simple solutions when facing this issue. Either introduce autocorrelation into the label, or remove it from the features. Both are rather easy econometrically:

- to increase the autocorrelation of the label, compute performance over longer time ranges. For instance, when working with monthly data, considering annual or biennial returns will do the trick.
- to get rid of autocorrelation, the shortest route is to resort to difference/variations: $\Delta x_{t,n}^{(k)} = x_{t,n}^{(k)} - x_{t-1,n}^{(k)}$. One advantage of this procedure is that it makes sense, economically: variations in features may be better drivers of performance, compared to raw levels.

The crucial choice is whether to work *mostly* with persistent or oscillating variables. A mix between the two in the feature space is also possible, but both types should be well represented.

5.7 Extensions

5.7.1 Transforming features

The feature space can easily be augmented through simple operations. One of them is lagging, that is, considering older values of features and assuming some memory effect for their impact on the label. This is naturally useful mostly if the features are oscillating (adding a layer of memory on persistent features can be somewhat redudant). New variables are defined by $\breve{x}_{t,n}^{(k)} = x_{t-1,n}^{(k)}$.

In some cases (e.g., insufficient number of features), it is possible to consider ratios or products between features. Accounting ratios like price-to-book, book-to-market, debt-to-equity are examples of functions of raw features that make sense. The gains brought by a larger spectrum of features are not obvious. The risk of overfitting increases, just like in a simple linear regression adding variables mechanically increases the R^2. The choices must make sense, economically.

Another way to increase the feature space (mentioned above) is to consider variations. Variations in sentiment, variations in book-to-market ratio, etc., can be relevant predictors because sometimes, the change is more important than the level. In this case, a new predictor is $\breve{x}_{t,n}^{(k)} = x_{t,n}^{(k)} - x_{t-1,n}^{(k)}$.

5.7.2 Macro-economic variables

Finally, we discuss a very important topic. The data should never be seperated from the context it comes from (its environment). In classical financial terms, this means that a particular model is likely to depend on the overarching situation which is often proxied by macro-economic indicators. One way to take this into account at the data level is simply to multiply the feature by a exogenous indicator z_t and in this case, the new predictor is

$\breve{x}_{t,n}^{(k)} = z_t \times x_{t,n}^{(k)}$. This technique is used by Gu et al. (2018) who use 8 economic indicators (plus the original predictors ($z_t = 1$)). This increase the feature space ninefold.

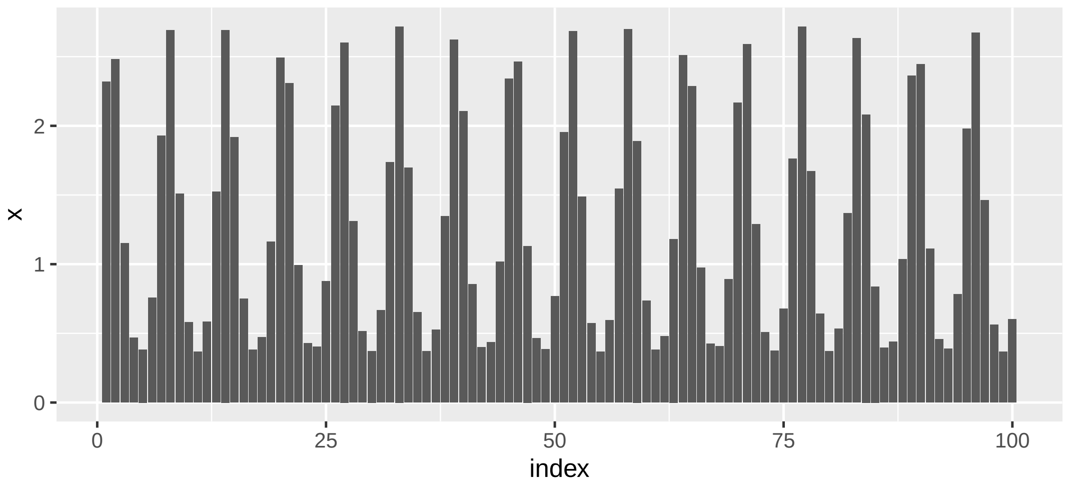
conditional feature engineering XXX regime dependent
XXXXXXX features released at different frequencies

5.8 Code and results

5.8.1 Impact of rescaling: graphical representation

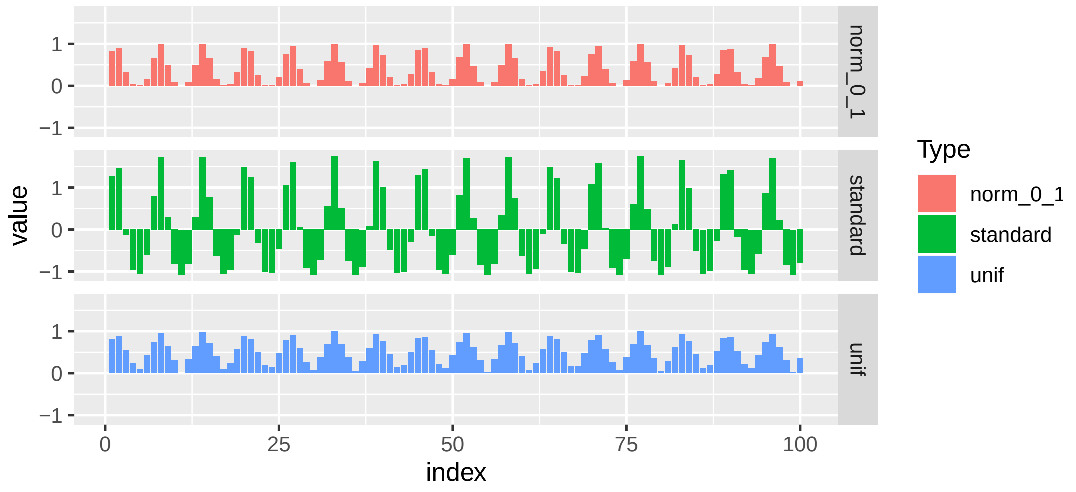
We start with a simple illustration of the different scaling methods. We generate an arbitrary series and then rescale it. The series is not random so that each time the code chunk is executed, the output remains the same.

```
Length <- 100                                 # Length of the sequence
x <- exp(sin(1:Length))                       # Original data
data <- data.frame(index = 1:Length, x = x)   # Data framed into dataframe
ggplot(data, aes(x = index, y = x)) + geom_bar(stat = "identity") # Plot
```



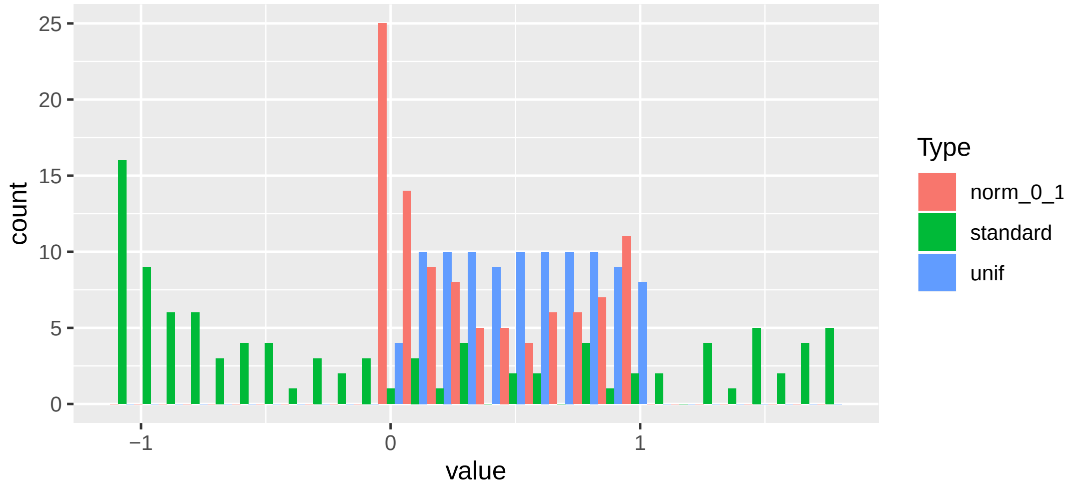
We define and plot the scaled variables below.

```
norm_unif <-  function(v){  # This is a function that uniformalises a vector.
    v <- v %>% as.matrix()
    return(ecdf(v)(v))
}

norm_0_1 <-  function(v){  # This is a function that uniformalises a vector.
    return((v-min(v))/(max(v)-min(v)))
}

data_norm <- data.frame(                        # Formatting the data
    index = 1:Length,                           # Index of point/instance
    standard = (x - mean(x)) / sd(x),           # Standardisation
    norm_0_1 = norm_0_1(x),                     # [0,1] reduction
    unif = norm_unif(x)) %>%                    # Uniformisation
    gather(key = Type, value = value, -index)   # Putting in tidy format
ggplot(data_norm, aes(x = index, y = value, fill = Type)) +   # Plot!
    geom_bar(stat = "identity") +
    facet_grid(Type~.)          # This option creates 3 concatenated graphs to ease comparison
```



Finally, we look at the histogram of the newly created variables.

```
ggplot(data_norm, aes(x = value, fill = Type)) + geom_histogram(position = "dodge")
```



With respect to shape, the green and red distributions are close to the original one. It is only the support that changes: the min/max rescaling ensures all values lie in the $[0, 1]$ interval. In both cases, the smallest values (on the left) display a spike in distribution. By construction, this spike disappears under the uniformisation: the points are evenly distributed over the unit interval.

TABLE 5.3: Sample data for a toy example.

firm	date	cap	return	cap_0_1	cap_u
1	1	10	0.06	0.000	0.333
1	2	50	0.01	0.364	0.667
1	3	100	-0.06	1.000	1.000
2	1	15	-0.03	0.026	0.667
2	2	10	0.00	0.000	0.333
2	3	15	0.02	0.000	0.333
3	1	200	-0.04	1.000	1.000
3	2	120	-0.02	1.000	1.000
3	3	80	0.00	0.765	0.667

5.8.2 Impact of rescaling: toy example

To illustrate the impact of chosing one particular rescaling method,[2] we build a simple dataset, comprising 3 firms and 3 dates.

```r
firm <- c(rep(1,3), rep(2,3), rep(3,3))             # Firms (3 lines for each)
date <- rep(c(1,2,3),3)                             # Dates
cap <- c(10, 50, 100,                               # Market capitalisation
         15, 10, 15,
         200, 120, 80)
return <- c(0.06, 0.01, -0.06,                      # Return values
            -0.03, 0.00, 0.02,
            -0.04, -0.02,0.00)
data_toy <- data.frame(firm, date, cap, return)     # Aggregation of data
data_toy <- data_toy %>%                            # Transformation of data
    group_by(date) %>%
    mutate(cap_0_1 = norm_0_1(cap), cap_u = norm_unif(cap))
knitr::kable(data_toy, digits = 3,                  # Display the data
             caption = "Sample data for a toy example.")
```

Let's briefly comment on this synthetic data. We assume that dates are ordered chronologically and far away: each date stands for a year or the beginning of a decade, but the (forward) returns are computed on a monthly basis. The first firm is hugely successful and multiplies its cap ten times over the periods. The second firms remains stable cap-wise, while the third one plummets. If we look at 'local' future returns, they are strongly negatively related to size for the first and third firms. For the second one, there is no clear pattern.

Date-by-date, the analysis is fairly similar, though slightly nuanced.

1. On date 1, the smallest firm has the largest return and the two others have negative returns.

2. On date 2, the biggest firm has a negative return while the two smaller firms do not.

3. On date 3, returns are decreasing with size.

While the relationship is not always perfectly monotonous, there seems to be a link between

[2]For a more thorough technical discussion on the impact of feature engineering, we refer to Galili and Meilijson (2016).

TABLE 5.4: Regression output when the independent var. comes from min-max rescaling

term	estimate	std.error	statistic	p.value
(Intercept)	0.0162778	0.0137351	1.185121	0.2746390
cap_0_1	-0.0497032	0.0213706	-2.325777	0.0529421

TABLE 5.5: Regression output when the independent var. comes from uniformisation

term	estimate	std.error	statistic	p.value
(Intercept)	0.06	0.0198139	3.028170	0.0191640
cap_u	-0.10	0.0275162	-3.634219	0.0083509

size and return and typically, investing in the smallest firm would be a very good strategy with this sample.

Now let us look at the output of simple regressions.

```
lm(return ~ cap_0_1, data = data_toy) %>% # First regression (min-max rescaling)
    broom::tidy() %>%
    knitr::kable(caption = 'Regression output when the independent var. comes
                 from min-max rescaling')
```

```
lm(return ~ cap_u, data = data_toy) %>%   # Second regression (uniformised feature)
    broom::tidy() %>%
    knitr::kable(caption = 'Regression output when the independent var. comes from uniformisation')
```

In terms of *p*-**value** (last column), the first estimation for the cap coefficient is above 5% (in Table 5.4) while the second is below 1% (in Table 5.5). One possible explanation for this discrepancy is the standard deviation of the variables. The deviations are equal to 0.47 and 0.29 for cap_0 and cap_u, respectively. Values like market capitalisations can have very large ranges and are thus subject to substantial deviations (even after scaling). Working with uniformised variables reduces dispersion and can help solve this problem.

Note that this is a **double-edged sword**: while it can help avoid **false negatives**, it can also lead to **false positives**.

5.9 Coding exercises

6

Penalized regressions and sparse hedging for minimum variance portfolios

In this chapter, we introduce the widespread concept of regularisation for linear models. There are in fact several possible applications for these models. The first one is straightforward: resort to penalizations to improve the robustness of factor-based predictive regressions. The outcome can then be used to fuel an allocation scheme. For instance, Han et al. (2019) and Rapach and Zhou (2019) use penalized regressions to improve stock return prediction when combining forecasts that emanate from individual characteristics.

Similar ideas can be devloped for macroeconomic predictions for instance, as in Uematsu and Tanaka (2019). The second application stems from a lesser known results which originates from Stevens (1998). It links the weights of optimal mean-variance portfolios to particular cross-sectional regressions. The idea is then different and the purpose is to improve the quality of mean-variance driven portfolio weights. We present the two approach below after an introduction on regularization techniques for linear models.

Other examples of financial applications of penalization can be found in d'Aspremont (2011), Ban et al. (2016) and Kremer et al. (2019). In any case, the idea is the same as in the seminal paper Tibshirani (1996): standard (unconstrained) optimization programs may lead to noisy estimates, thus adding a structuring constraint helps remove some noise (at the cost of a possible bias). For instance, Kremer et al. (2019) use this concept to build more robust mean-variance (Markowitz (1952)) portfolios.

6.1 Penalised regressions

6.1.1 Simple regressions

The ideas behind linear models are at least two centuries old (Legendre (1805) is an early reference on least squares optimization). Given a matrix of predictors $\mathbf{X}$, we seek to decompose the output vector $\mathbf{y}$ as a linear function of the columns of $\mathbf{X}$ (written $\mathbf{X}\boldsymbol{\beta}$) plus an error term $\boldsymbol{\epsilon}$: $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$.

The best choice of $\boldsymbol{\beta}$ is naturally the one that minimizes the error. For analytical tractability, it is the sum of squared error that is minimized: $L = \boldsymbol{\epsilon}'\boldsymbol{\epsilon}$. The loss L is called the sum of squared residuals (SSR). In order to find the optimal $\boldsymbol{\beta}$, it is imperative to differentiate this loss L with respect to $\boldsymbol{\beta}$ because the first order condition requires that the gradient be equal to zero:

$$\begin{aligned}\nabla_{\boldsymbol{\beta}} L &= \frac{\partial}{\partial \boldsymbol{\beta}}(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})'(\mathbf{y}-\mathbf{X}\boldsymbol{\beta}) = \frac{\partial}{\partial \boldsymbol{\beta}}\boldsymbol{\beta}'\mathbf{X}'\mathbf{X}\boldsymbol{\beta} - 2\mathbf{y}'\mathbf{X}\boldsymbol{\beta} \\ &= 2\mathbf{X}'\mathbf{X}\boldsymbol{\beta} - 2\mathbf{X}'\mathbf{y}\end{aligned}$$

so that the first order condition $\nabla_{\boldsymbol{\beta}} = \mathbf{0}$ is satisfied if

$$\boldsymbol{\beta}^* = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}, \tag{6.1}$$

which is known as the standard **ordinary least squares** (OLS) solution of the linear model. If the matrix $\mathbf{X}$ has dimensions $I \times K$, then the $\mathbf{X}'\mathbf{X}$ can only be inverted if the number of rows I is strictly superior to the number of columns K. In some cases, that may not hold: there are more predictors than instances and there is no unique value of $\boldsymbol{\beta}$ that minimizes the loss. If $\mathbf{X}'\mathbf{X}$ is nonsingular (or positive definite), then the second order condition ensures that $\boldsymbol{\beta}^*$ yields a global minimum for the loss L (the second order derivative of L with respect to $\boldsymbol{\beta}$ is $\mathbf{X}'\mathbf{X}$).

Up to now, we have made no distributional assumption on any of the above quantities. Standard assumptions are the following:
- $\mathbb{E}[\mathbf{y}|\mathbf{X}] = \mathbf{X}\boldsymbol{\beta}$: **linear shape for the regression function**;
- $\mathbb{E}[\boldsymbol{\epsilon}|\mathbf{X}] = \mathbf{0}$: errors are **independent of predictors**;
- $\mathbb{E}[\boldsymbol{\epsilon}\boldsymbol{\epsilon}'|\mathbf{X}] = \sigma^2\mathbf{I}$: **homoscedasticity**: errors are uncorrelated and have identical variance;
- the ϵ_i are normally distributed.

Under these hypotheses, it is possible to perform statistical tests related to the $\hat{\boldsymbol{\beta}}$ coefficients. We refer to Chapters 2 to 4 in Greene (2018) for a thorough treatment on linear models as well as to Chapter 5 of the same book for details on the corresponding tests.

6.1.2 Forms of penalizations

Penalised regressions have been popularised since the seminal work of Tibshirani (1996). The idea is to impose a constraint on the coefficients of the regression, namely that their total magnitude be restrained. In his original paper, Tibshirani (1996) proposes to estimate the following model (LASSO):

$$y_i = \sum_{j=1}^J \beta_j x_{i,j} + \epsilon_i, \quad i = 1, \dots, I, \quad \text{s.t.} \quad \sum_{j=1}^J |\beta_j| < \delta, \tag{6.2}$$

for some strictly positive constant δ. Under least square minimisation, this amounts to solve the Lagrangian formulation:

$$\min_{\beta} \left\{ \sum_{i=1}^I \left(y_i - \sum_{j=1}^J \beta_j x_{i,j} \right)^2 + \lambda \sum_{j=1}^J |\beta_j| \right\}, \tag{6.3}$$

for some value $\lambda > 0$. This specification seems close to the ridge regression (L^2 regularisation), which is in fact anterior to the Lasso:

$$\min_{\beta} \left\{ \sum_{i=1}^I \left(y_i - \sum_{j=1}^J \beta_j x_{i,j} \right)^2 + \lambda \sum_{j=1}^J \beta_j^2 \right\}, \tag{6.4}$$

which is equivalent to estimating the following model

$$y_i = \sum_{j=1}^{J} \beta_j x_{i,j} + \epsilon_i, \quad i = 1, \dots, I, \quad \text{s.t.} \quad \sum_{j=1}^{J} \beta_j^2 < \delta, \tag{6.5}$$

but the outcome is in fact quite different, which justifies a separate treatment. Mechanically, as λ increases (or as δ in (6.5) *decreases*), the coefficients of the ridge regression all slowly decrease in magnitude towards zero. In the case of the LASSO, the convergence is somewhat more brutal as some coefficients shrink to zero very quickly. For λ sufficiently large, only one coefficient will remain nonzero, while in the ridge regression, the zero value is only reached asymptotically for all coefficients.

To depict the difference between the Lasso and the ridge regression, let us consider the case of $K = 2$ predictors which is shown in Figure 6.1. The optimal unconstrained solution $\boldsymbol{\beta}^*$ is pictured in red in the middle of the space. The problem is naturally that it does not satisfy the imposed conditions. These constraints are shown in light grey: they take the shape of a square $|\beta_1| + |\beta_2| \le \delta$ in the case of the Lasso and a circle $\beta_1^2 + \beta_2^2 \le \delta$ for the ridge regression. In order to satisfy these constraints, the optimization need to look in the vicinity of $\boldsymbol{\beta}^*$ by allowing for larger error levels. These error levels are shown as orange ellipsoids in the figure. When the requirement on the error is loose enough, one ellipsoid touches the acceptable boundary (in grey) and this is where the constrained solution is located.

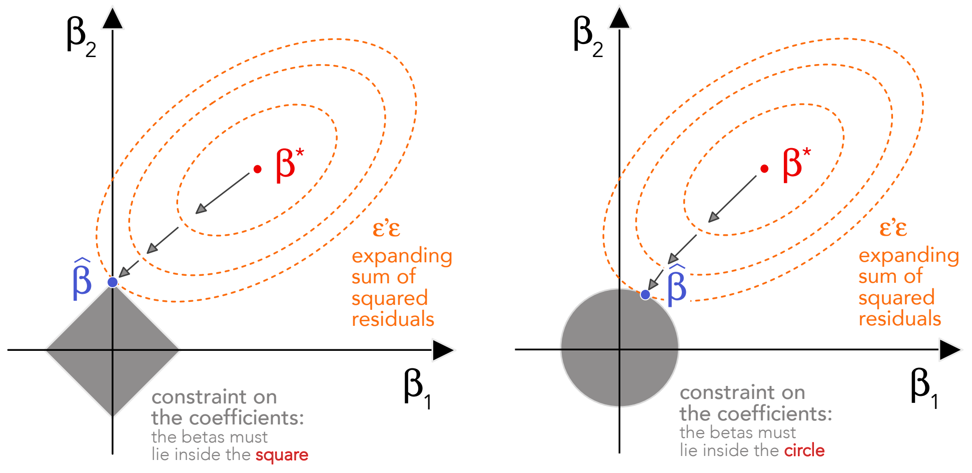

β2
β*
ε'ε
expanding sum of squared residuals
β̂
β1
constraint on the coefficients:
the betas must lie inside the square
β2
β*
ε'ε
expanding sum of squared residuals
β̂
β1
constraint on the coefficients:
the betas must lie inside the circle

FIGURE 6.1: Schematic view of Lasso versus ridge regressions.

Both methods work when the number of exogenous variables surpasses that of observations, i.e., in the case where classical regressions are ill-defined. This is easy to see in the case of the ridge regression for which the OLS solution is simply

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}'\mathbf{X} + \lambda \mathbf{I}_N)^{-1}\mathbf{X}'\mathbf{Y}.$$

The additional term $\lambda \mathbf{I}_N$ compared to Equation (6.1) ensures that the inverse matrix is well-defined whenever $\lambda > 0$. As λ increases, the magnitudes of the $\hat{\beta}_i$ decrease, which explains why penalizations are sometimes referred to as **shrinkage** methods.

Zou and Hastie (2005) propose to benefit from the best of both worlds when combining both penalisations in a convex manner (elasticnet):

$$y_i = \sum_{j=1}^J \beta_j x_{i,j} + \epsilon_i, \quad \text{s.t.} \quad (1-\alpha)\sum_{j=1}^J |\beta_j| + \alpha \sum_{j=1}^J \beta_j^2 < \delta, \quad i = 1, \dots, N, \tag{6.6}$$

which is the solved as

$$\min_{\beta} \left\{ \sum_{i=1}^I \left(y_i - \sum_{j=1}^J \beta_j x_{i,j} \right)^2 + \lambda \left(\alpha \sum_{j=1}^J |\beta_j| + (1-\alpha) \sum_{j=1}^J \beta_j^2 \right) \right\}, \tag{6.7}$$

The main advantage of the LASSO compared to the ridge regression is its selection capability. Indeed, given a very large number of variables (or predictors), the LASSO will progressively rule-out those that are the least relevant. The elasticnet preserves this selection ability and Zou and Hastie (2005) argue that in some cases, it is even more effective that the LASSO. The parameter $\alpha \in [0,1]$ tunes the smoothness of convergence (of the coefficients) towards zero. The closer α is to zero the smoother the convergence.

6.1.3 Illustrations

We start with simple illustrations of penalized regressions. We start with the LASSO. The original implementation by the authors is in R, which is practical. The syntax is slightly different, compared to usual linear models. The illustrations are run on the whole dataset. First, we estimate the coefficients. By default, the function chooses a large array of penalization values so that the results for different intensities can be shown immediately.

```
library(glmnet)
y_penalized <- data_ml$R1M_Usd                               # Dependent variable
x_penalized <- data_ml %>% select(features) %>% as.matrix() # Predictors
fit_lasso <- glmnet(x_penalized, y_penalized, alpha = 1)
```

Once the coefficients are computed, they require some wrangling before plotting.

```
lasso_res <- summary(fit_lasso$beta)                       # Extract LASSO coefs
lambda <- fit_lasso$lambda                                 # Values of the penalisation constant
lasso_res$Lambda <- lambda[lasso_res$j]                    # Put the labels where they belong
lasso_res$Feat <- features[lasso_res$i] %>% as.factor()    # Add names of variables to the output
lasso_res[1:120,] %>%                                      # Take the first 150 estimates
    ggplot(aes(x = Lambda, y = x, color = Feat)) +         # Plot!
    geom_line() + coord_fixed(0.25) +                      # Change aspect ratio of graph
    theme(legend.text = element_text(size = 8))            # Reduce legend font size
```

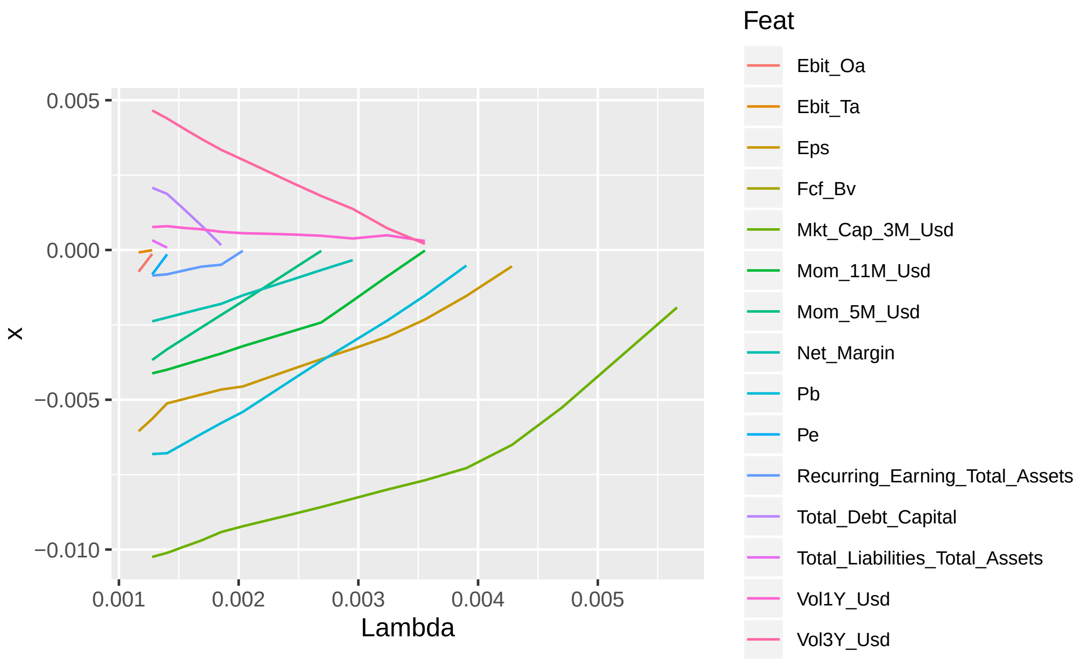


FIGURE 6.2: LASSO model. The dependent variable is the 1 month ahead return.

The graph plots the evolution of coefficients as the penalization intensity, λ, increases. For some characteristics, like Ebit_Ta (in orange), the convergence to zero is rapid. Other variables resist the penalization longer, like Mkt_Cap_3M_Usd, which is the last one to vanish. Essentially, this means that at the first order, this variable is an important driver of future 1 month returns - in our sample. Moreover, the negative sign of its coefficient are a confirmation (again, in this sample) of the size anomaly, according to which small firms experience higher future returns compared to their larger counterparts.

Next, we turn to ridge regressions.

```
fit_ridge <- glmnet(x_penalized, y_penalized, alpha = 0)
ridge_res <- summary(fit_ridge$beta)                              # Extract ridge coefs
lambda <- fit_ridge$lambda                                        # Values of penalisation constant
ridge_res$Feat <- features[ridge_res$i] %>% as.factor()
ridge_res$Lambda <- lambda[ridge_res$j]                           # Put labels where they belong
ridge_res %>%
    filter(Feat %in% levels(droplevels(lasso_res$Feat[1:120]))) %>%  # Keep the same features as above
    ggplot(aes(x = Lambda, y = x, color = Feat)) +                # Plot!
    geom_line() + scale_x_log10() + coord_fixed(45) +            # Aspect ratio of + logscale
    theme(legend.text = element_text(size = 8))
```

In the above graph, the convergence to zero is much smoother. We underline that the x-axis (penalization intensities) have a log-scale. This allows to see the early patterns (close to zero, to the left) more clearly. As in the previous figure, the Mkt_Cap_3M_Usd predictor clearly dominates, with again large negative coefficients. Nonetheless, as λ increases, its domination over the other predictor fades.

By definition, the elasticnet will produce curves that behave like a blend of the two above approaches. Nonetheless, as long as $\alpha > 0$, the selective property of the LASSO will be preserved: some features will see their coefficients shrink rapidly to zero. In fact, the strength

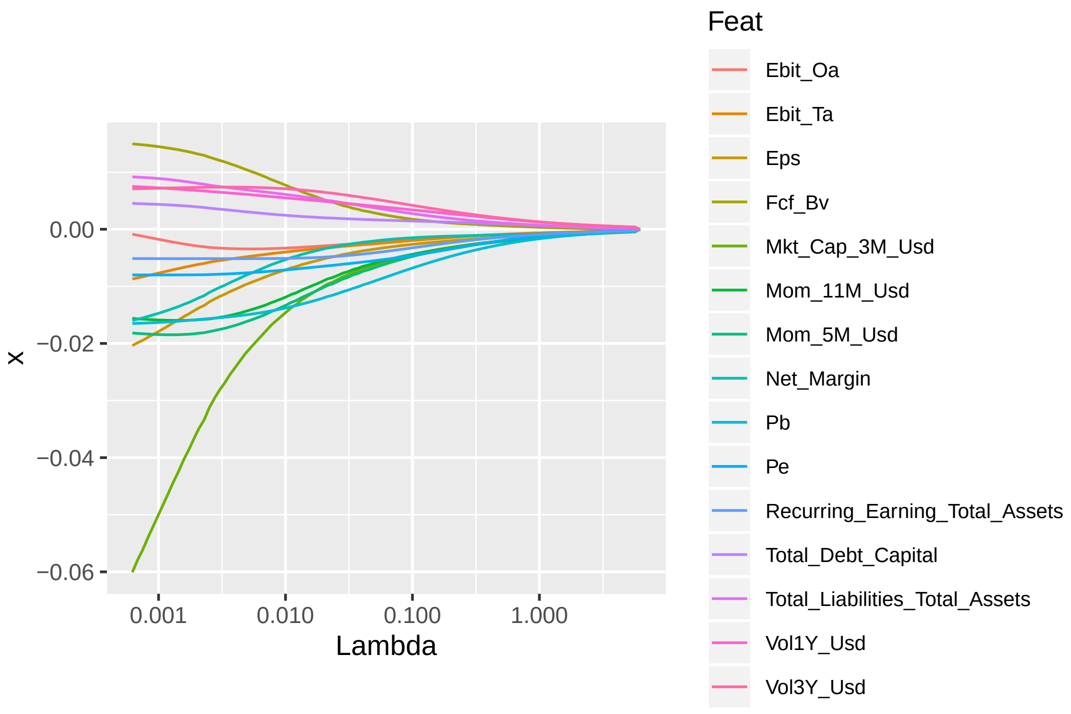


FIGURE 6.3: Ridge regression. The dependent variable is the 1 month ahead return.

of the LASSO is such that a balanced mix of the two penalization is not reached at $\alpha = 1/2$, but rather at a much smaller value (possibly below 0.1).

6.2 Sparse hedging for minimum variance portfolios

6.2.1 Presentation and derivations

The idea of constructing sparse portfolios is not new per se (see, e.g., Brodie et al. (2009), Fastrich et al. (2015)) and the link with the selective property of the LASSO is rather straightforward in classical quadratic programs. Note that the choice of the L^1 norm is imperative because when enforcing a simple L^2 norm, the diversification of the portfolio increases (see Coqueret (2015)).

The idea behind this section stems from Goto and Xu (2015) but the cornerstone result was first published by Stevens (1998) and we present it below. We provide details because the derivations are not commonplace in the literature.

In usual mean-variance allocations, one core ingredient is the inverse covariance matrix of assets $\mathbf{\Sigma}^{-1}$. For instance, the maximum Sharpe ratio portfolio is given by

$$\mathbf{w}^{\text{MSR}} = \frac{\mathbf{\Sigma}^{-1}\boldsymbol{\mu}}{\mathbf{1}'\mathbf{\Sigma}^{-1}\boldsymbol{\mu}}, \tag{6.8}$$

where μ is the vector of expected (excess) returns. Taking $\mu = \mathbf{1}$ yields the minimum variance portfolio, which is agnostic in terms of the first moment of expected returns (and, as such, usually more robust than most alternatives).

If we decompose the matrix $\mathbf{\Sigma}$ into:

$$\mathbf{\Sigma} = \begin{bmatrix} \sigma^2 & \mathbf{c}' \\ \mathbf{c} & \mathbf{C} \end{bmatrix},$$

classical partitioning results (e.g., Schur complements) imply

$$\mathbf{\Sigma}^{-1} = \begin{bmatrix} (\sigma^2 - \mathbf{c}'\mathbf{C}^{-1}\mathbf{c})^{-1} & -(\sigma^2 - \mathbf{c}'\mathbf{C}^{-1}\mathbf{c})^{-1}\mathbf{c}'\mathbf{C}^{-1} \\ -(\sigma^2 - \mathbf{c}'\mathbf{C}^{-1}\mathbf{c})^{-1}\mathbf{C}^{-1}\mathbf{c} & \mathbf{C}^{-1} + (\sigma^2 - \mathbf{c}'\mathbf{C}^{-1}\mathbf{c})^{-1}\mathbf{C}^{-1}\mathbf{c}\mathbf{c}'\mathbf{C}^{-1} \end{bmatrix}.$$

We are interested in the first line, which has 2 components: the factor $(\sigma^2 - \mathbf{c}'\mathbf{C}^{-1}\mathbf{c})^{-1}$ and the line vector $\mathbf{c}'\mathbf{C}^{-1}$. $\mathbf{C}$ is the covariance matrix of assets 2 to N and $\mathbf{c}$ is the covariance between the first asset and all other assets. The first line of $\mathbf{\Sigma}^{-1}$ is

$$(\sigma^2 - \mathbf{c}'\mathbf{C}^{-1}\mathbf{c})^{-1} \begin{bmatrix} 1 & \underbrace{-\mathbf{c}'\mathbf{C}^{-1}}_{N-1 \text{ terms}} \end{bmatrix}. \tag{6.9}$$

We now consider an alternative setting. We regress the returns of the first asset on those of all other assets:

$$r_{1,t} = a_1 + \sum_{n=2}^{N} \beta_{1|n} r_{n,t} + \epsilon_t, \quad \text{i.e.,} \quad \mathbf{r}_1 = a_1 \mathbf{1}_T + \mathbf{R}_{-1}\beta_1 + \epsilon_1, \tag{6.10}$$

where $\mathbf{R}_{-1}$ gathers the returns of all assets except the first one. The OLS estimator for β_1 is

$$\hat{\beta}_1 = \mathbf{C}^{-1}\mathbf{c}, \tag{6.11}$$

and this is the partitioned formed (when a constant is included to the regression) stemming from the Frisch-Waugh-Lovell theorem (see chapter 3 in Greene (2018)).

In addition,

$$(1 - R^2)\sigma^2_{\mathbf{r}_1} = \sigma^2_{\mathbf{r}_1} - \mathbf{c}'\mathbf{C}^{-1}\mathbf{c} = \sigma^2_{\epsilon_1}. \tag{6.12}$$

The proof of this last fact is given below.

With $\mathbf{X}$ being the concatenation of $\mathbf{1}_T$ with returns $\mathbf{R}_{-1}$ and with $\mathbf{y} = \mathbf{r}_1$, the classical expression of the R^2 is

$$R^2 = 1 - \frac{\epsilon'\epsilon}{T\sigma^2_Y} = 1 - \frac{\mathbf{y}'\mathbf{y} - \hat{\beta}'\mathbf{X}'\mathbf{X}\hat{\beta}}{T\sigma^2_Y} = 1 - \frac{\mathbf{y}'\mathbf{y} - \mathbf{y}'\mathbf{X}\hat{\beta}}{T\sigma^2_Y},$$

with fitted values $\mathbf{X}\hat{\beta} = \hat{a}_1\mathbf{1}_T + \mathbf{R}_{-1}\mathbf{C}^{-1}\mathbf{c}$. Hence,

$$\begin{aligned}
T\sigma^2_{\mathbf{r}_1}R^2 &= T\sigma^2_{\mathbf{r}_1} - \mathbf{r}_1'\mathbf{r}_1 + \hat{a}_1\mathbf{1}_T'\mathbf{r}_1 + \mathbf{r}_1'\mathbf{R}_{-1}\mathbf{C}^{-1}\mathbf{c} \\
T(1-R^2)\sigma^2_{\mathbf{r}_1} &= \mathbf{r}_1'\mathbf{r}_1 - \hat{a}_1\mathbf{1}_T'\mathbf{r}_1 - \left(\tilde{\mathbf{r}}_1 + \frac{\mathbf{1}_T\mathbf{1}_T'}{T}\mathbf{r}_1\right)'\left(\tilde{\mathbf{R}}_{-1} + \frac{\mathbf{1}_T\mathbf{1}_T'}{T}\mathbf{R}_{-1}\right)\mathbf{C}^{-1}\mathbf{c} \\
T(1-R^2)\sigma^2_{\mathbf{r}_1} &= \mathbf{r}_1'\mathbf{r}_1 - \hat{a}_1\mathbf{1}_T'\mathbf{r}_1 - T\mathbf{c}'\mathbf{C}^{-1}\mathbf{c} - \mathbf{r}_1'\frac{\mathbf{1}_T\mathbf{1}_T'}{T}\mathbf{R}_{-1}\mathbf{C}^{-1}\mathbf{c} \\
T(1-R^2)\sigma^2_{\mathbf{r}_1} &= \mathbf{r}_1'\mathbf{r}_1 - \frac{(\mathbf{1}_T'\mathbf{r}_1)^2}{T} - T\mathbf{c}'\mathbf{C}^{-1}\mathbf{c} \\
(1-R^2)\sigma^2_{\mathbf{r}_1} &= \sigma^2_{\mathbf{r}_1} - \mathbf{c}'\mathbf{C}^{-1}\mathbf{c}
\end{aligned}$$

where in the fourth equality we have plugged $\hat{a}_1 = \frac{\mathbf{1}'_T}{T}(\mathbf{r}_1 - \mathbf{R}_{-1}\mathbf{C}^{-1}\mathbf{c})$. Note: there is probably a simpler proof - see e.g. Section 3.5 in Greene (2018).

Combining (6.9), (6.11) and (6.12), we get that the first line of $\mathbf{\Sigma}^{-1}$ is equal to

$$\frac{1}{\sigma^2_{\epsilon_1}} \times \left[1 \quad -\hat{\boldsymbol{\beta}}'_1\right]. \tag{6.13}$$

Given the first line of $\mathbf{\Sigma}^{-1}$, it suffices to multiply by $\boldsymbol{\mu}$ to get the portfolio weight in the first asset (up to a scaling constant).

There is a nice economic intuition behind the above results which justifies the term "sparse hedging". We take the case of the minimum variance portfolio, for which $\boldsymbol{\mu} = \mathbf{1}$. In Equation (6.10), we try to explain the return of asset 1 with that of all other assets. In the above equation, up to a scaling constant, the portfolio has a unit position in the first asset and $-\hat{\boldsymbol{\beta}}_1$ positions in all other assets. Hence, the purpose of all other assets is clearly to hedge the return of the first one. In fact, these positions are aimed at minimizing the squared errors of the aggregate portfolio for asset one (these errors a exactly ϵ_1). Moreover, the scaling factor $\sigma^{-2}_{\epsilon_1}$ is also simple to interpret: the more we trust the regression output (because of a small $\sigma^2_{\epsilon_1}$), the more we invest in the hedging portfolio of the asset.

This reasoning is easily generalized for any line of $\mathbf{\Sigma}^{-1}$, which can be obtained by regressing the returns of asset i on the returns of all other assets. If the allocation scheme has the form (6.8) for given values of $\boldsymbol{\mu}$, then the pseudo-code for the sparse portfolio strategy is the following.

At each date (which we omit for notational convenience),

- For all stocks i,

  1. estimate the elastic-net regression over the $t = 1, \dots, T$ samples to get the i^{th} line of $\hat{\mathbf{\Sigma}}^{-1}$:

  $$\left[\hat{\mathbf{\Sigma}}^{-1}\right]_{i,\cdot} = \underset{\boldsymbol{\beta}_{i|}}{\text{argmin}} \left\{ \sum_{t=1}^T \left(r_{i,t} - a_i + \sum_{n \neq i}^N \beta_{i|n} r_{n,t} \right)^2 + \lambda\alpha ||\boldsymbol{\beta}_{i|}||_1 + \lambda(1-\alpha)||\boldsymbol{\beta}_{i|}||_2^2 \right\}$$

  2. to get the weights of asset i, we compute the μ-weighted sum: $w_i = \sigma^{-2}_{\epsilon_i}\left(\mu_i - \sum_{j\neq i} \beta_{i|j}\mu_j\right)$

The introduction of the penalization norms is the new ingredient, compared to the original approach of Stevens (1998). The benefits are twofold: first, introducing constraints yields weights that are more robust and less subject to errors in the estimates of μ; second, because of sparsity, weights are more stable, less leveraged and thus the strategy is less impacted by transaction costs. Before we turn to numerical applications, we mention a more direct route to the estimation of a robust inverse covariance matrix: the Graphical LASSO. We refer to the original article Friedman et al. (2008) for more details on this subject.

6.2.2 Example

The interest of sparse hedging portfolios is to proposed a robust approach to the estimation of minimum variance policies. Indeed, since the vector of expected returns $\boldsymbol{\mu}$ is usually very noisy, a simple solution is to adopt an agnostic view by setting $\boldsymbol{\mu} = \mathbf{1}$. In order to test the added value of the sparsity constraint, we must resort to a full backtest. In doing so, we anticipate the content of chapter 13.

We first prepare the variables. Sparse portfolios are based on returns only; we thus base our analysis on the dedicated variable in matrix/rectangular format (*returns*) which were created at the end of Chapter 2.

Then, we initialize the output variables: portfolio weights and portfolio returns. We want to compare three strategies: an equally weighted (EW) benchmark of all stocks, the classical global minimum variance portfolio (GMV) and the sparse-hedging approach to minimum variance.

```r
t_oos <- returns$date[returns$date > separation_date] %>%            # Out-of-sample dates
    unique() %>%                                                     # Remove duplicates
    as.Date(origin = "1970-01-01")                                   # Transform in date format
Tt <- length(t_oos)                                                  # Nb of dates, avoid T
nb_port <- 3                                                         # Nb of portfolios/strats.
portf_weights <- array(0, dim = c(Tt, nb_port, ncol(returns) - 1))  # Initial portf. weights
portf_returns <- matrix(0, nrow = Tt, ncol = nb_port)               # Initial portf. returns
```

Next, because it is the purpose of this section, we isolate the computation of the weights of sparse-hedging portfolios. In the case of minimum variance portfolios, when $\boldsymbol{\mu} = \mathbf{1}$, the weight in asset 1 will simply be the sum of all terms in Equation (6.13) and the other weights have similar forms.

```r
weights_sparsehedge <- function(returns, alpha, lambda){  # The parameters are defined here
    w <- 0                                                # Initiate weights
    for(i in 1:ncol(returns)){                            # Loop on the assets
        y <- returns[,i]                                  # Dependent variable
        x <- returns[,-i]                                 # Independent variable
        fit <- glmnet(x,y, family = "gaussian", alpha = alpha, lambda = lambda)
        err <- y-predict(fit, x)                          # Prediction errors
        w[i] <- (1-sum(fit$beta))/var(err)                # Output: weight of asset i
    }
    return(w / sum(w))                                    # Normalisation of weights
}
```

In order to benchmark our strategy, we define a meta-weighting function that embeds three strategies: the EW benchmarks (1), the classical GMV (2) and the sparse-hedging minimum variance (3). For the GMV, since there are much more assets than date, the covariance matrix is singular. Thus, we as a small heuristic shrinkage term. For a more rigorous treatment of this technique, ce refer to the original article Ledoit and Wolf (2004) and to the recent improvements mentioned in Ledoit and Wolf (2017). In short, we use $\hat{\boldsymbol{\Sigma}} = \boldsymbol{\Sigma}_S + \delta \boldsymbol{I}$ for some small constant δ (equal to 0.001 in the code below).

```r
weights_multi <- function(returns,j, alpha, lambda){
    N <- ncol(returns)
    if(j == 1){                                 # j = 1 => EW
        return(rep(1/N,N))
    }
    if(j == 2){                                 # j = 2 => Minimum Variance
        sigma <- cov(returns) + 0.01 * diag(N)  # Covariance matrix + regularizing term
        w <- solve(sigma) %*% rep(1,N)          # Inverse & multiply
        return(w / sum(w))                      # Normalize
    }
    if(j == 3){                                 # j = 3 => Penalised / elasticnet
```

```
        w <- weights_sparsehedge(returns, alpha, lambda)
    }
}
```

Finally, we proceed to the backtesting loop. Given the number of assets, the execution of the loop takes a few minutes. At the end of the loop, we compute the standard deviation of portfolio returns (monthly volatility). This is the key indicator as minimum variance seeks to minimize this particular metric.

```
for(t in 1:length(t_oos)){                                            # Loop = rebal. dates
    temp_data <- returns %>%                                          # Data for weights
        filter(date < t_oos[t]) %>%                                   # Expand. window
        select(-date) %>%
        as.matrix()
    realised_returns <- returns %>%                                   # OOS returns
        filter(date ==  t_oos[t]) %>%
        select(-date)
    for(j in 1:nb_port){                                              # Looping over strats
        portf_weights[t,j,] <- weights_multi(temp_data, j, 0.1, 0.1)       # Hard-coded params!
        portf_returns[t,j] <- sum(portf_weights[t,j,] * realised_returns)  # Portf. returns
    }
}
apply(portf_returns, 2, sd)  # Portfolio volatilities (monthly scale)
```

```
## [1] 0.04180422 0.03350424 0.02672169
```

The aim of the sparse hedging restrictions is to provide a better estimate of the covariance structure of assets so that the estimation of minimum variance portfolio weights is more accurate. From the above exercise, we see that the monthly volatility is indeed reduced when building covariance matrices based on sparse hedging relationships. This is not the case if we use the shrunk sample covariance matrix because there is probably too much noise in the estimates of correlations between assets. Working with daily returns would likely improve the quality of the estimates. But the above backtest shows that the penalized methodology performs well even when the number of observations (dates) is small compared to the number of assets.

6.3 Predictive regressions

6.3.1 Literature review and principle

The topic of predictive regressions sits on a collection of very interesting articles. One influential contribution is Stambaugh (1999), where the authors show the perils of regressions in which the independent variables are autocorrelated. In this case, the usual OLS estimate is biased and must therefore be corrected.

A second important topic pertains to the time-dependence of the coefficients in predictive regressions. One contribution in this direction is Dangl and Halling (2012), where latent variables and Bayesian selection. More recently Kelly et al. (2019) also use latent variables to model the cross-section of stock returns. The time-varying nature of coefficients of predictive regressions is further documented by Henkel et al. (2011) for short term returns. Lastly, Farmer et al. (2019) introduce the concept of pockets of predictability: assets or markets experiences different phases; in some phases they are predictable and in some others, they aren't. Pockets are measured both by the number of days that a t-statistic is above a particular threshold and by the magnitude of the R^2 over the considered period.

The introduction of penalization within predictive regressions goes back at least to Rapach et al. (2013), where they are used to assess lead-lag relationships between US markets and other international stock exchanges. More recently, Chinco et al. (2019a) use LASSO regressions to forecast high frequency returns based on past returns (in the cross-section) at various horizons. They report statistically significant gains. Han et al. (2019) and Rapach and Zhou (2019) use LASSO and elasticnet regressions (respectively) to improve forecast combinations and single out the characteristics that matter when explaining stock returns.

These contributions underline the relevant of the overlap between predictive regressions and penalized regressions. In simple machine-learning based asset pricing, we often seek to build models such as that of Equation (4.6). If we stick to a linear relationship and add penalization terms, then the model becomes:

$$r_{t+1,n} = \alpha_n + \sum_{k=1}^K \beta_n^k f^k_{t,n} + \epsilon_{t+1,n}, \quad \text{s.t.} \quad (1-\alpha)\sum_{j=1}^J |\beta_j| + \alpha \sum_{j=1}^J \beta_j^2 < \theta$$

where we use $f^k_{t,n}$ or $x^k_{t,n}$ interchangeably and θ is some penalization intensity. Again, one of the aim of the regularization is to generate more robust estimates. If the patterns extracted hold out of sample, then

$$\hat{r}_{t+1,n} = \hat{\alpha}_n + \sum_{k=1}^K \hat{\beta}_n^k f^k_{t,n},$$

will be a relatively reliable proxy of future performance.

6.3.2 Code and results

Given the form of our dataset, implementing penalized predictive regressions is easy.

```
y_penalized_train <- training_sample$R1M_Usd                              # Dependent variable
x_penalized_train <- training_sample %>% select(features) %>% as.matrix() # Predictors
fit_pen_pred <- glmnet(x_penalized_train, y_penalized_train, alpha = 0.1, lambda = 0.1)
```

We then report two key performance measures: the mean squared error and the hit ratio. A detailed account of metrics is given later in the book (Chapter **??**).

```
x_penalized_test <- testing_sample %>% select(features) %>% as.matrix()    # Predictors
mean((predict(fit_pen_pred, x_penalized_test) - testing_sample$R1M_Usd)^2) # MSE
```

```
## [1] 0.03699696
```

```
mean(predict(fit_pen_pred, x_penalized_test) * testing_sample$R1M_Usd > 0) # Hit ratio
```

```
## [1] 0.5460346
```

From an investor's standpoint, the MSE (or even the mean absolute error) are hard to interpret because it complicated to map them mentally into some intuitive financial indicator. In this perspective, the hit ratio is more natural. It tells the proportion of correct signs achieved by the predictions. If the investor is long in positive signals and short in negative ones, the hit ratio indicates the proportion of 'correct' bets (the positions that go in the expected direction). A natural threshold is 50% but because of transaction costs, 51% of accurate forecasts probably won't be profitable. The figure 0.5460346 can be deemed a relatively good hit ratio, though not a very impressive one.

6.4 Coding exercises

1. On the test sample, evaluate the impact of the two elastic net parameters on out-of-sample accuracy.

2.

7

Tree-based methods

Classification and regression trees are simple yet powerful clustering algorithms popularised by the monograph Breiman et al. (1984). Decision trees and their extensions are known to be quite efficient forecasting tools when working on tabular data. A large proportion of winning solutions in ML contests (especially on the Kaggle webiste[1]) resort to improvements of trees.

Recently, the surge in Machine Learning applications in Finance has led to multiple publications that use trees in portfolio allocation problems. A long though not exhaustive list includes: Ballings et al. (2015), Patel et al. (2015a), Patel et al. (2015b), Moritz and Zimmermann (2016), Krauss et al. (2017), Gu et al. (2018), Guida and Coqueret (2018a), Coqueret and Guida (2019) and Simonian et al. (2019). One notable contribution is Bryzgalova et al. (2019b) in which the authors create factors from trees by sorting portfolios via simple trees, which they call *Asset Pricing Trees*.

In this chapter, we review the methodologies associated to trees and their applications in portfolio choice.

7.1 Simple trees

7.1.1 Principle

Decision trees seek to partition datasets into homogeneous clusters. Given an exogenous variable **Y** and features **X**, trees iteratively split the sample into groups (usually two at a time) which are as homogeneous in **Y** as possible. The splits are made according to one variable within the set of features. A short word on nomenclature: when **Y** consists of real numbers, we talk about *regression trees* and when **Y** is categorical, we use the term *classification trees*.

Before formalising this idea, we illustrate this process in Figure 7.1. There are 12 stars with three features: color, size and complexity (number of branches).

[1]See www.kaggle.com.

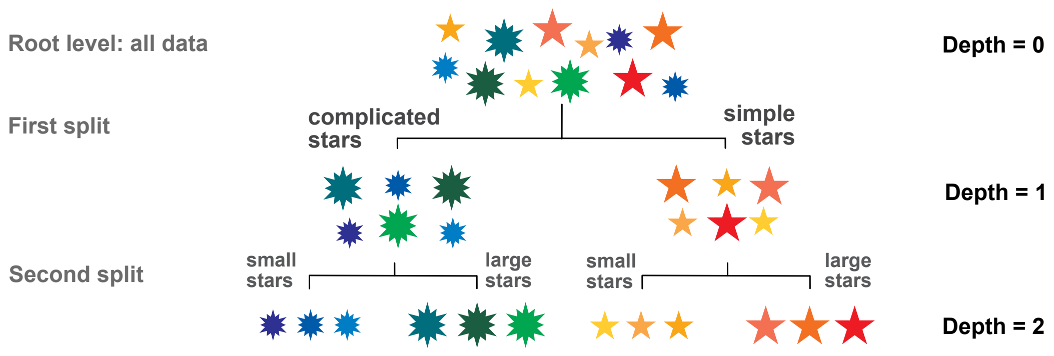


FIGURE 7.1: Elementary tree scheme: visualization of the splitting process.

The dependent variable is the color (let's consider the wavelength associated to the color for simplicity). The first split is made according to size or complexity. Clearly, complexity is the better choice: complicated stars are blue and green while simple stars are yellow, orange and red. Splitting according to size would have mixed blue and yellow stars (small ones) and green and orange stars (large ones).

The second step is to split the two clusters one level further. Since only one variable (size) is relevant, the secondary splits are straightforward. In the end, our stylised tree has four consistent clusters. The analogy with factor investing is simple: the color represents performance: red for high performance and blue for mediocre performance. The features (size and complexity of stars) are replaced by firm-specific attributes, such as capitalization, accounting ratios, etc. Hence, the purpose of the exercise is to find the characteristics that allow to split firms into the ones that will perform well versus those likely to fare more poorly.

We now turn to the technical construction of regression trees. We follow the standard literature as exposed in Breiman et al. (1984) or in Chapter 9 of Hastie et al. (2009). Given a sample of $(y_i,\mathbf{x}_i)$ of size I, a *regression* tree seeks the splitting points that minimize the total variation of the y_i inside the two child clusters. In order to do that, it proceeds in two steps. First, it finds, for each feature $x_i^{(k)}$, the best splitting point (so that the clusters are homogeneous in $\mathbf{Y}$). Second, it selects the feature that achieves the highest level of homogeneity.

Homogeneity in regression trees is closely linked to variance. Since we want the y_i inside each cluster to be similar, we seek to minimize their variability inside each cluster and then sum the two figures. We cannot sum the variances because this would not take into account the relative sizes of clusters. Hence, we work with *total* variation, which is the variance times the number of elements in the clusters.

Below, the notation is a bit heavy because we resort to superscripts k (the index of the feature), but it is largely possible to ignore these superscripts to ease understanding. The first step is to find the best split for each feature, that is, solve $\underset{c^{(k)}}{\text{argmin}}\, V_I^{(k)}(c^{(k)})$ with

$$V_I^{(k)}(c^{(k)}) = \underbrace{\sum_{x_i^{(k)}<c^{(k)}} \left(y_i - m_I^{k,-}(c^{(k)})\right)^2}_{\text{Total dispersion of first cluster}} + \underbrace{\sum_{x_i^{(k)}>c^{(k)}} \left(y_i - m_I^{k,+}(c^{(k)})\right)^2}_{\text{Total dispersion of second cluster}}, \quad (7.1)$$

where $m_I^{k,-}(c^{(k)}) = \frac{1}{\#\{i,x_i^{(k)}<c^{(k)}\}}\sum_{\{x_i^{(k)}<c^{(k)}\}} y_i$ and $m_I^{k,+}(c^{(k)}) =$

$\frac{1}{\#\{i,x_i^{(k)}>c^{(k)}\}}\sum_{\{x_i^{(k)}>c^{(k)}\}}y_i$ are the average values of Y, conditional on $X^{(k)}$ being smaller or larger than c. The cardinal function $\#\{\cdot\}$ counts the number of instances of its argument. For feature k, the optimal split $c^{k,*}$ is thus the one for which the total dispersion over the two subgroups is the smallest.

The optimal splits satisfy $c^{k,*} = \underset{c^{(k)}}{\text{argmin}}\ V_I^{(k)}(c^{(k)})$. Of all the possible splitting variables, the tree will choose the one that minimizes the total dispersion: $k^* = \underset{k}{\text{argmin}}\ V_I^{(k)}(c^{k,*})$.

After one split is performed, the procedure continues on the two newly formed clusters. There are several criteria that can determine when to stop the splitting process (see Section 7.1.3). One simple criterion is to fix a maximum number of levels (the depth) for the tree. A usual condition is to impose a minimum gain that is expected for each split. If the reduction in dispersion after the split is only marginal and below a specified threshold, then the split is not executed. For further technical discussions on decision trees, we refer for instance to section 9.2.4 of Hastie et al. (2009).

When the tree is built (trained), a prediction for new instances is easy to make. Given its feature values, the instance ends up in one leaf of the tree. Each leaf has an average value for the label: this is the predicted outcome. Of course, this only works when the label is numerical. We discuss below the changes that occur when it is categorical.

7.1.2 Further details on classification

Classification exercises are somewhat more complex than regression tasks. The most obvious difference is the loss function which must take into account the fact that the final output is not a simple number, but a vector. The output $\tilde{\mathbf{y}}_i$ has as many elements as there are categories in the label and each element is the probability that the instance belong to the corresponding category.

For instance, if there are 3 categories: *buy*, *hold* and *sell*, then each instance would have a label with as many columns as there are classes. Following our example, one label would be (1,0,0) for a *buy* position for instance. We refer to Section 5.5.2 for a introduction on this topic.

Inside a tree, labels are aggregated at each cluster level. A typical output would look like (0.6,0.1,0.3): they are the proportions of each class represented within the cluster. In this case, the cluster has 60% of *buy*, 10% of *hold* and 30% of *sell*.

The loss function must take into account this multidimensionality of the label. When building trees, since the aim is to favor homogeneity, the loss penalizes outputs that are not concentrated towards one class. Indeed, facing a diversified output of (0.3,0.4,0.3) is much harder to handle than the concentrated case of (0.8,0.1,0.1).

The algorithm is thus seeking purity: it searches a splitting criterion that will lead to clusters that are as pure as possible, i.e., with one very dominant class, or a few dominant classes. There are several metrics proposed by the literature and all are based on the proportions coded in the output. If there are J classes, we denote these proportions with p_j. The usual loss functions are:

- the Gini impurity index: $1-\sum_{j=1}^J p_j^2$;

- the misclassification error: $1-\max_j p_j$;
- entropy: $-\sum_{j=1}^J \log(p_j)p_j$.

The Gini index is nothing but one minus the Herfindahl index which measures the diversification of a portfolio. Trees seek partitions that are the least diversified. The minimum value of the Gini index is zero and reached when one $p_j=1$ and all others are equal to zero. The maximum value is equal to $1-1/J$ and is reached when all $p_j=1/J$. Similar relationships hold for the other two losses. One drawback of the misclassification error is its lack of differentiability which explains why the other two options are often favored.

Once the tree is grown, new instances automatically belong to one final leaf. This leaf is associated to the proportions of classes it nests.

7.1.3 Pruning criteria

When building a tree, the splitting process can be pursued until the full tree is grown, that is, when:
- when all instances belong to separate leaves, and/or
- when all leaves comprise instances that cannot be further segregated based on the current set of features.

At this stage, the splitting process cannot be pursued.

Obviously, fully grown trees often lead to almost perfect fits when the predictors are relevant, numerous and numerical. Nonetheless, the fine grained idiosyncrasies of the training sample are of little interest for out-of-sample predictions. For instance, being able to perfectly match the patterns of 2000 to 2006 will probably not be very interesting in 2007 to 2009. The most reliable sections of the trees are those closest to the root because they embed large portions of the data: the average values in the early clusters are trustworthy because the are computed on a large number of observations. The first splits are those that matter the most because they highlight the most general patterns. The deepest splits only deal with the peculiarities of the sample.

Thus, it is imperative to limit the size of the tree. There are several ways to prune the tree and all depend on some particular criteria. We list a few of them below:

- Impose a minimum number of instances for each terminal node (leaf). This ensures that each final cluster is composed by a sufficient number of observations. Hence, the average value of the label will be reliable because calculated on a large amount of data.
- Similarly, it can be imposed that a cluster has a minimal size before even considering any further split. This criterion is of course related to the one above.
- Require a certain threshold of improvement in the fit. If a split does not sufficiently reduce the loss, then it can be deemed unnecessary. The user specifies a small number $\epsilon>0$ and a split is only validated if the loss obtained post-split is smaller than $1-\epsilon$ times the loss before the split.
- Limit the depth of the tree. The depth is defined as the overal maximum number of splits between the root and any leaf of the tree.

In the example below, we implement all of these criteria at the same time, but usually, two of them at most should suffice.

7.1.4 Code and interpretation

We start with a simple tree and its interpretation. The label is the future 1 month return and the features are all predictors available in the sample. The tree is trained on the full sample.

```
## Loading required package: rpart
```

```
library(rpart)              # Tree package
library(rpart.plot)         # Tree plot package
formula <- paste("R1M_Usd ~", paste(features, collapse = " + ")) # Defines the model
formula <- as.formula(formula)                                    # Forcing formula object
fit_tree <- rpart(formula,
             data = data_ml,     # Data source: full sample
             minbucket = 1500,   # Min nb of obs required in each terminal node (leaf)
             minsplit = 4000,    # Min nb of obs required to continue splitting
             cp = 0.0001,        # Precision: smaller = more leaves
             maxdepth = 3        # Maximum depth (i.e. tree levels)
             )
rpart.plot(fit_tree)             # Plot the tree
```

There usually exists a convention in the representation of trees. At each node, a condition describes the split with a boolean expression. If the expression is true, then the instance goes to the left cluster, if not, it goes to the right cluster. Given the whole sample, the initial split in this tree (Figure 7.2) is performed according to market capitalization. If the 6 month

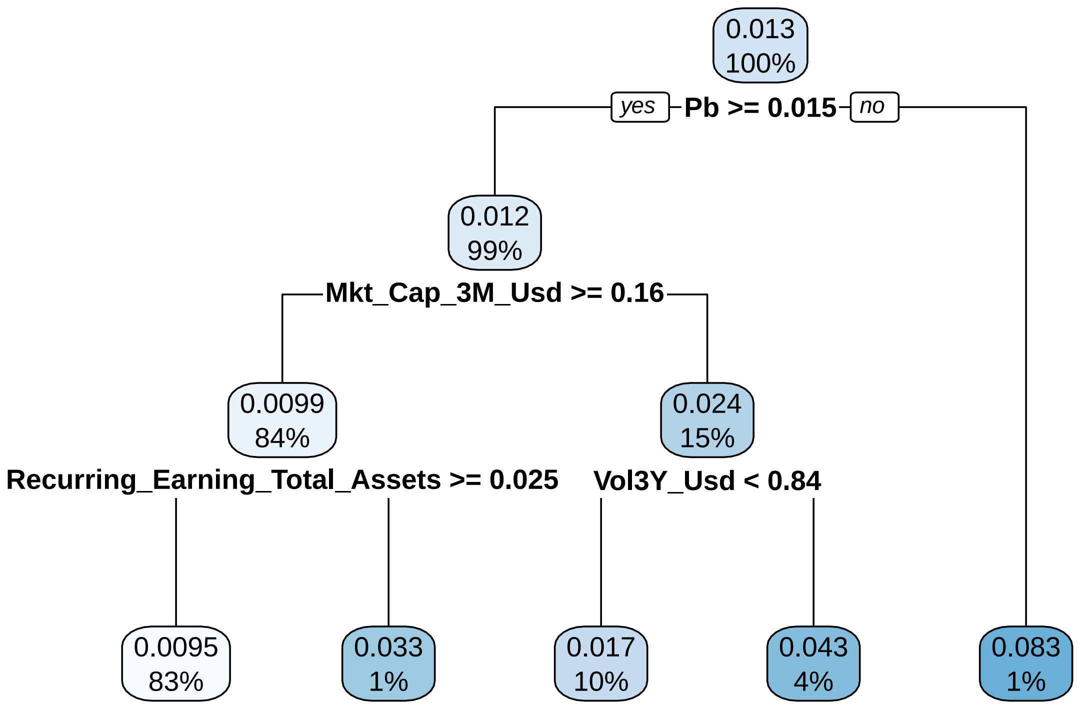


FIGURE 7.2: Simple tree. The dependent variable is the 1 month future return.

market capitalization score (or value) of the instance is above 0.025, then the instance is placed in the left bucket, otherwise, it goes in the right bucket.

At each node, there are two important metrics. The first one is the average value of the label in the cluster and the second one is the proportion of instances in the cluster. At the top of the tree, all instances (100%) are present and the average 1 month future return is 1.3%. One level below, the left cluster is by far the most crowded, with roughly 98% of observations averaging a 1.2% return. The right cluster is much smaller (2%) but concentrates instances with a much higher average return (6.1%).

The splitting process continues similarly at each node until some condition is satisfied (typically here: the maximum depth is reached). A color codes the average return: from white (low return) to blue (high return). The cluster with the lowest average return consists of firms that satisfy *all* the following criteria:

- have a 6 month market capitalization score above 0.025;
- have a 3 month market capitalization score above 0.19;
- have a score of recurring earnings over total assets above 0.025.

Notice that one peculiarity of trees is their possible heterogeneity in cluster sizes. Sometimes, a few clusters gather almost all of the observations while a few small groups embed some outliers. This is not a favorable property of trees as small groups are more likely to be flukes and may fail to generalize out-of-sample.

This is why we imposed restrictions during the construction of the tree. The first one (minbucket = 1500 in the code) imposes that each cluster consists of at least 1500 instances. The second one (minsplit) further imposes that a cluster comprises at least 4000 observation in order to pursue the splitting process. The cp = 0.0001 parameter in the code requires any split to reduce the loss below 0.9999 times its original value before the split. Finally, the maximum depth of three essentially means that there are at most three splits between the root of the tree and any terminal leaf.

The complexity of the tree (measured by the number of terminal leaves) is a decreasing function of minbucket, minsplit and cp and an increasing function of maximum depth.

Once the model has be trained (i.e., the tree is grown), a prediction for any instance is the average value of the label within the cluster where the instance should land.

```
predict(fit_tree, data_ml[1:5,]) # Test (prediction) on the first five instances of the sample
```

```
##           1           2           3           4           5 
## 0.009529823 0.009529823 0.009529823 0.009529823 0.009529823
```

Given the figure, we immediately conclude that the first three instances belong to the second cluster (starting from the left) and the last two belong to the fourth.

As a verification of the first split, we plot the smoothed average of future returns, conditionally on market capitalization, past return and price-to-book scores.

```
data_ml %>% ggplot() +
    stat_smooth(aes(x = Mkt_Cap_6M_Usd, y = R1M_Usd, color = "Market Cap"), se = FALSE) +
    stat_smooth(aes(x = Pb, y = R1M_Usd, color = "Price-to-Book"), se = FALSE) +
    stat_smooth(aes(x = Mom_Sharp_11M_Usd, y = R1M_Usd, color = "Momentum"), se = FALSE) +
    xlab("Predictor")
```

Indeed, we acknowledge a strong impact of market capitalization for the smallest firms: they appear to earn a very high return (close to +5% on a monthly basis). The pattern is much more pronounced compared to other well documented anomalies namely the value premium and the (cross-sectional) momentum effect.

Finally, we assess the predictive quality of a single tree on the testing set (the tree is grown on the training set). We use a deeper tree, with a maximum depth of five.

```
fit_tree2 <- rpart(formula,
             data = training_sample,     # Data source: training sample
             minbucket = 1500,           # Min nb of obs required in each terminal node (leaf)
             minsplit = 4000,            # Min nb of obs required to continue splitting
             cp = 0.0001,                # Precision: smaller = more leaves
             maxdepth = 5                # Maximum depth (i.e. tree levels)
             )
mean((predict(fit_tree2, testing_sample) - testing_sample$R1M_Usd)^2) # MSE
```

```
## [1] 0.03700039
```

```
mean(predict(fit_tree2, testing_sample) * testing_sample$R1M_Usd > 0) # Hit ratio
```

```
## [1] 0.5416619
```

The mean squared error is usually hard to interpret. It's not easy to map an error on returns into the impact on investment decisions. The hit ratio is a more intuitive indicator because it evaluates the proportion of correct guesses (and hence profitable investments). Obviously, it

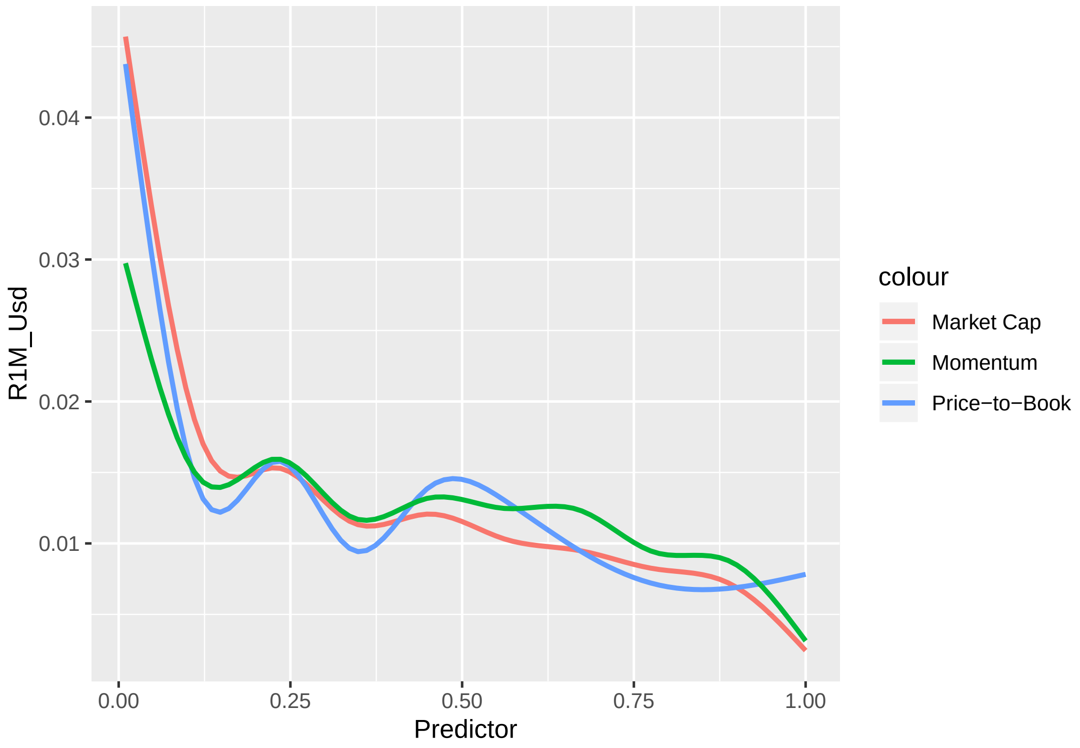


FIGURE 7.3: Average of 1 month future returns, conditionally on market capitalization and price-to-book scores

is not perfect: 55% of small gains can be mitigated by 45% of large losses. Nonetheless, it is a popular metric and moreover it corresponds to the usual accuracy measure often computed in binary classification exercises. Here, an accuracy of 0.5416619 is satisfactory. Even if any number above 50% may seem valuable, it must not be forgottent that transaction costs will curtail benefits. Hence, the benchmark threshold is probably at least at 52%.

7.2 Random forests

While trees give intuitive representations of relationships between **Y** and **X**, they can be improved via the simple idea of ensembles (which is discussed both more generally and in more details in Chapter 12).

7.2.1 Principle

One intuitive extension of predicting tools is a *combination* of such tools. Naturally, it is not obvious upfront which individual model is the best, hence a combination seems a reasonable path towards the diversification of prediction errors (when they are not too correlated). Some theoretical foundations of model combinations were laid out in Schapire (1990).

More practical considerations were proposed later in Ho (1995) and more importantly in Breiman (2001) which is the major reference for random forests. There are two ways to create multiple predictors from simple trees and random forests combines both:

- first, the model can be trained on similar yet different datasets. One way to achieve this is via bootstrap: the instances are resampled with or without replacement, yielding new training data each time a new tree is built.
- second, the data can be altered by curtailing the number of predictors. Alternative models are built based on different sets of features. The user chooses how many features to retain and then the algorithm selects these features randomly at each try.

Hence, it becomes simple to grow many different trees and the ensemble is simply a weigthed combination of all trees. Usually, equal weights are used, which is an agnostic and robust choice. We illustrate the idea of simple combinations (also referred to as bagging) in Figure 7.4 below. The terminal prediction is simply the mean of all intermediate predictions.

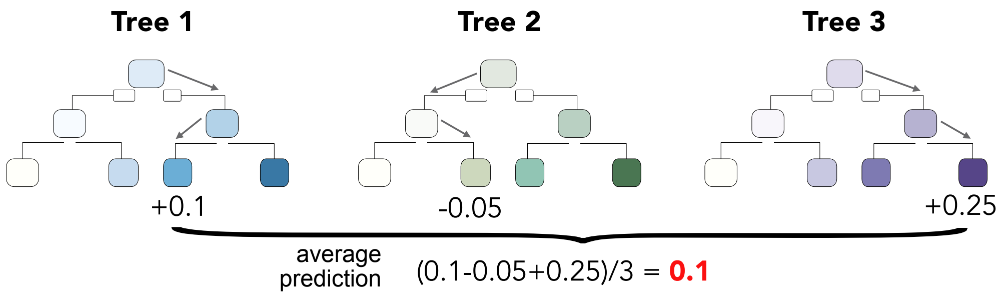


FIGURE 7.4: Combining tree outputs via random forests.

Random forests, because they are built on the idea of bootstrapping are more efficient than simple trees. They are used by Ballings et al. (2015), Patel et al. (2015a), Krauss et al. (2017), and Huck (2019) and they are shown to perform very well in these papers. The original theoretical properties of random forests are demonstrated in Breiman (2001) for classification trees and are fairly straightforward (it's likely similar formulations hold for other ensemble types). The inaccuracy of the aggregation (as measured by the so-called generalization error) is bounded by $\bar{\rho}(1-s^2)/s^2$, where
- s is the strength (average quality) of the individual classifiers and
- $\bar{\rho}$ is the average correlation between the learners.

Notably, Breiman (2001) also shows that as the number of trees grows to infinity, the inaccuracy converges to some finite number which explains why random forests are not prone to overfitting.

While the original paper of Breiman (2001) is dedicated to classification models, many articles have since then tackled the problem of regression trees. We refer the interested reader to Biau (2012) and to Scornet et al. (2015). Finally, further results on classifying ensembles can be obtained in Biau et al. (2008) and we mention the short survey paper Denil et al. (2014) which sums up recent results in this field.

7.2.2 Code and results

Several implementations of random forests exist. For simplicity, we choose to work with the original R library, but another choice could be the one developed by h2o, which is a highly efficient meta-environment for machine learning (coded in Java).

The syntax of randomForest follows that of many ML libraries. The full list of options for some random forest implementations is prohibitively large.[2] Below, we train a model and exhibit the predictions for the first 5 instances of the testing sample.

```
library(randomForest)
set.seed(42)                            # Sets the random seed
fit_RF <- randomForest(formula,         # Same formula as for simple trees!
                 data = training_sample,    # Data source: training sample
                 sampsize = 10000,      # Size of (random) sample for each tree
                 replace = FALSE,       # Is the sampling done with replacement?
                 nodesize = 250,        # Minimum size of terminal cluster
                 ntree = 40,            # Nb of random trees
                 mtry = 30              # Nb of predictive variables for each tree
    )
predict(fit_RF, testing_sample[1:5,])   # Prediction over the first 5 test instances
```

```
##            1            2            3            4            5
##  0.009787728  0.012507087  0.008722386  0.009398814 -0.011511758
```

One first comment is that each instance has its own prediction, which contrasts with the outcome of simple tree-based outcomes. Combining many trees leads to tailored forecasts. Note that the second line of the chunk freezes the random number generation. Indeed, random forests are by construction contingent on the arbitrary combinations of instances and features that are chosen to build the individual learners.

In the above example, each individual learner (tree) is built on 10,000 randomly chosen instances (without replacement) and each terminal leaf (cluster) must comprise at least 240

[2]See, e.g., http://docs.h2o.ai/h2o/latest-stable/h2o-r/docs/reference/h2o.randomForest.html

elements (observations). In total, 40 trees are aggregated and each tree is constructed based on 30 randomly chosen predictors (out of the whole set of features).

Unlike for simple trees, it is not possible to simply illustrate the outcome of the learning process. It could be possible to extract all 40 trees but a synthetic visualization is out-of-reach. A simplified view can be obtained via variable importance, as is discussed in Section 14.1.1.

Finally, we can assess the accuracy of the model.

```
mean((predict(fit_RF, testing_sample) - testing_sample$R1M_Usd)^2) # MSE
```

```
## [1] 0.03698197
```

```
mean(predict(fit_RF, testing_sample) * testing_sample$R1M_Usd > 0) # Hit ratio
```

```
## [1] 0.5370186
```

The MSE is smaller than 4% and the hit ratio is close to 54%, which is reasonably above both 50% and 52% thresholds.

Let's see if we can improve the hit ratio by resorting to a classification exercise. We start by training the model on a new formula (the label is R1M_Usd_C).

```
formula_C <- paste("R1M_Usd_C ~", paste(features, collapse = " + ")) # Defines the model
formula_C <- as.formula(formula_C)                                   # Forcing formula object
fit_RF_C <- randomForest(formula_C,         # New formula!
                 data = training_sample,    # Data source: training sample
                 sampsize = 20000,          # Size of (random) sample for each tree
                 replace = FALSE,           # Is the sampling done with replacement?
                 nodesize = 250,            # Minimum size of terminal cluster
                 ntree = 40,                # Number of random trees
                 mtry = 30                  # Number of predictive variables for each tree
    )
```

We can then assess the proportion of correct (binary) guesses.

```
mean(predict(fit_RF_C, testing_sample) == testing_sample$R1M_Usd_C) # Hit ratio
```

```
## [1] 0.4977353
```

The accuracy is disappointing. There are two potential explanations for this (beyond the possibility of very different patterns in the training and testing sets). The first one is the sample size, which may be too small. The original training set has more than 200,000 observations, hence we retain only one in 10 in the above training specification. We are thus probably sideline relevant information and the cost can be heavy. The second reason is the number of predictors, which is set to 30, i.e., one third of the total at our disposal. Unfortunately, this leaves room for the algorithm to pick less pertinent predictors. The default numbers of predictors chosen by the routines are $\sqrt{p}$ and $p/3$ for classification and regression tasks, respectively. Here p is the total number of features.

7.3 Boosted trees: Adaboost

The idea of boosting is slightly more advanced compared to agnostic aggregation. In random forest, we hope that the diversification through many trees will improve the overall quality of the model. In boosting, it is sought to iteratively improve the model whenever a new tree is added. There are many ways to boost learning and we present two that can easily be implemented with trees. The first one (Adaboost, for adaptive boosting) improves the learning process by progressively focusing on the instances that yield the largest errors. The second one (xgboost) is a flexible algorithm in which each new tree is only focused on the minimization of the training sample loss.

7.3.1 Methodology

The origins of adaboost go back to Freund and Schapire (1997), Freund and Schapire (1996) and for the sake of completeness, we also mention the book dedicated on boosting Schapire and Freund (2012). Extensions of these ideas are proposed in Friedman et al. (2000) (the so-called real Adaboost algorithm) and in Drucker (1997) (for regression analysis). Theoretical treatments were derived by Breiman et al. (2004).

We start by directly stating the general structure of the algorithm:

- set equal weights $w_i = I^{-1}$;
- For $m = 1, \dots, M$ do:
    1. Find a learner l_m that minimizes the weighted loss $\sum_{i=1}^{I} w_i L(l_m(\mathbf{x}_i), \mathbf{y}_i)$;
    2. Compute a learner weight
    $$a_m = f_a(\mathbf{w}, l_m(\mathbf{x}), \mathbf{y}); \tag{7.2}$$
    3. Update the instance weights
    $$w_i \leftarrow w_i e^{f_w(l_m(\mathbf{x}_i), \mathbf{y}_i)}; \tag{7.3}$$
    4. Normalize the w_i to sum to one;
- The output for instance $\mathbf{x}_i$ is a simple function of $\sum_{m=1}^{M} a_m l_m(\mathbf{x}_i)$,

$$\tilde{y}_i = f_y\left(\sum_{m=1}^{M} a_m l_m(\mathbf{x}_i)\right). \tag{7.4}$$

Let us comment on the steps of the algorithm. The formulation holds for many variations of Adaboost and we will specify the functions f_a and f_w below.

1. The first step seeks to find a learner (tree) l_m that minimizes a weighted loss. Here the base loss function L essentially depends on the task (regression versus classification).
2. The second and third steps are the most interesting because they are the heart of Adaboost: they define the way the algorithm adapts sequentially. Because the purpose is to aggregate models, a more sophisticated approach compared

to uniform weights for learners is a tailored weight for each learner. A natural property (for f_a) should be that a learner that yields a smaller error should have a larger weight because it is more accurate.

3. The third step is to change the weights of observations. In this case, because the model aims at improving the learning process, f_w is constructed to give more weight on observations for which the current model does not do a good job (i.e., generates the largest errors). Hence, the next learner will be incentivized to pay more attention on these pathological cases.
4. The third step is a simple scaling procedure.

In Table 7.1, we detail two examples of weighting functions used in the literature. For the original Adaboost (Freund and Schapire (1996), Freund and Schapire (1997)), the label is binary with values +1 and -1 only. The second example stems from Drucker (1997) and is dedicated to regression analysis (with real-valued label). The interested reader can have a look at other possibilities in Schapire (2003) and Ridgeway et al. (1999).

TABLE 7.1: Examples of functions for Adaboost-like algorithms.

	Bin. classif. (orig. Adaboost)	Regression (Drucker (1997))
Individual error	$\epsilon_i=\mathbf{1}_{\{y_i\neq l_m(\mathbf{x}_i)\}}$	$\epsilon_i=\frac{\|y_i-l_m(\mathbf{x}_i)\|}{\underset{i}{\max}\|y_i-l_m(\mathbf{x}_i)\|}$
Weight of learner via f_a	$f_a=\log\left(\frac{1-\epsilon}{\epsilon}\right)$,with $\epsilon=I^{-1}\sum_{i=1}^I w_i\epsilon_i$	$f_a=\log\left(\frac{1-\epsilon}{\epsilon}\right)$,with $\epsilon=I^{-1}\sum_{i=1}^I w_i\epsilon_i$
Weight of instances via $f_w(i)$	$f_w=f_a\epsilon_i$	$f_w=f_a\epsilon_i$
Output function via f_y	$f_y(x)=\text{sign}(x)$	weighted median of predictions

Let us comment on the original Adaboost specification. The basic error term $\epsilon_i=\mathbf{1}_{\{y_i\neq l_m(\mathbf{x}_i)\}}$ is a dummy number indicating if the prediction is correct (we recall only two values are possible, +1 and -1). The average error $\epsilon\in[0,1]$ is simply a weighted average of individual errors and the weight of the m^{th} learner defined in Equation (7.2) is given by $a_m=\log\left(\frac{1-\epsilon}{\epsilon}\right)$. The function $x\mapsto\log((1-x)x^{-1})$ decreases on $[0,1]$ and switches sign (from positive to negative) at $x=1/2$. Hence, when the average error is small, the learner has a large positive weight but when the error becomes large, the learner can even obtain a negative weight. Indeed, the threshold $\epsilon>1/2$ indicated that the learner is wrong more than 50% of the time. Obviously, this indicates a problem and the learner should even be discarded.

The change in instance weights follows a similar logic. The new weight is proportional to $w_i\left(\frac{1-\epsilon}{\epsilon}\right)^{\epsilon_i}$. If the prediction is right and $\epsilon_i=0$, the weight is unchanged. If the prediction is wrong and $\epsilon_i=1$, the weight is adjusted depending on the aggregate error ϵ. If the error is small and the learner efficient ($\epsilon<1/2$), then $(1-\epsilon)/\epsilon>1$ and the weight of the instance increases. This means that for the next round, the learner will have to focus more on instance i.

Lastly, the final prediction of the model corresponds to the sign of the weighted sums of individual predictions: if the sum is positive, the model will predict +1 and it will yield -1 otherwise.[3] The odds of a zero sum are negligible. In the case of numerical labels, the process is slightly more complicated and we refer to Section 3, step 8 of Drucker (1997) for more details on how to proceed.

We end this presentation with one word on instance weighting. There are two ways to deal with this topic. The first one works at the level of the loss functions. For regression trees, Equation (7.1) would naturally generalize to

$$V_N^{(k)}(c^{(k)}, \mathbf{w}) = \sum_{x_i^{(k)} < c^{(k)}} w_i \left(y_i - m_N^{k,-}(c^{(k)})\right)^2 + \sum_{x_i^{(k)} > c^{(k)}} w_i \left(y_i - m_N^{k,+}(c^{(k)})\right)^2,$$

and hence an instance with a large weight w_i would contribute more to the dispersion of its cluster. For classification objectives, the alteration is more complex and we refer to Ting (2002) for one example of an instance-weighted tree-growing algorithm. The idea is closely linked to the alteration of the misclassification risk via a loss matrix (see Section 9.2.4 in Hastie et al. (2009)).

The second way to enforce instance weighting is via random sampling. If instances have weights w_i, then the training of learners can be performed over a sample that is randomly extracted with distribution equal to w_i. In this case, an instance with a larger weight will have more chances to be represented in the training sample. The original adaboost algorithm relies on this method.

7.3.2 Illustration

Below, we test an implementation of the original adaboost classifier. As such, we work with the R1M_Usd_C variable and change the model formula. The computational cost of adaboost is high on large datasets, thus we work with a smaller sample and we only impose three iterations.

```
library(fastAdaboost)                                                   # Adaboost package
subsample <- (1:52000)*4                                                # Target small sample
fit_adaboost_C <- adaboost(formula_C,                                   # Model spec.
                          data = data.frame(training_sample[subsample,]), # Data source
                          nIter = 3)                                    # Number of trees
```

Finally, we evaluate the performance of the classifier.

```
mean(testing_sample$R1M_Usd_C == predict(fit_adaboost_C, testing_sample)$class)
```

```
## [1] 0.5028202
```

The accuracy (as evaluated by the hit ratio) is clearly not satisfactory. One reason for this may be the restrictions we enforced for the training (smaller sample and only three trees).

[3]The Real Adaboost of Friedman et al. (2000) has a different output: the probability of belonging to a particular class.

7.4 Boosted trees: extreme gradient boosting

The ideas behind tree boosting were popularized, among others, by Mason et al. (2000), Friedman (2001), and Friedman (2002). In this case, the combination of learners (prediction tools) is not agnostic as in random forest, but adapted (or optimized) at the learner level. At each step s, the sum of models $M_S = \sum_{s=1}^{S-1} m_s + m_S$ is such that the last learner m_S was precisely designed to reduce the loss of M_S on the training sample.

Below, we follow closely the original work of Chen and Guestrin (2016) because their algorithm yields incredibly accurate predictions and also because it is highly customizable. It is their implementation that we use in our empirical section. The other popular alternative is lightgbm (see Ke et al. (2017)). What XGBoost seeks to minimise is the objective

$$O = \underbrace{\sum_{i=1}^{I} \text{loss}(y_i, \tilde{y}_i)}_{\text{error term}} \quad + \quad \underbrace{\sum_{j=1}^{J} \Omega(T_j)}_{\text{regularisation term}} \quad .$$

The first term (over all instances) measures the distance between the true label and the output from the model. The second term (over all trees) penalises models that are too complex.

For simplicity, we propose the full derivation with the simplest loss function $\text{loss}(y, \tilde{y}) = (y - \tilde{y})^2$, so that:

$$O = \sum_{i=1}^{I} \left(y_i - m_{J-1}(\mathbf{x}_i) - T_J(\mathbf{x}_i)\right)^2 + \sum_{j=1}^{J} \Omega(T_j).$$

7.4.1 Managing Loss

Let us assume that we have already built all trees T_j up to $j = 1, \dots, J-1$ (and hence model M_{J-1}): how to choose tree T_J optimally? We rewrite

$$\begin{aligned}
O &= \sum_{i=1}^{I} \left(y_i - m_{J-1}(\mathbf{x}_i) - T_J(\mathbf{x}_i)\right)^2 + \sum_{j=1}^{J} \Omega(T_j) \\
&= \sum_{i=1}^{I} \left\{y_i^2 + m_{J-1}(\mathbf{x}_i)^2 + T_J(\mathbf{x}_i)^2\right\} + \sum_{j=1}^{J-1} \Omega(T_j) + \Omega(T_J) \quad \text{(squared terms + penalisation)} \\
&\quad - 2\sum_{i=1}^{I} \{y_i m_{J-1}(\mathbf{x}_i) + y_i T_J(\mathbf{x}_i) - m_{J-1}(\mathbf{x}_i) T_J(\mathbf{x}_i))\} \quad \text{(cross terms)} \\
&= \sum_{i=1}^{I} \left\{-2y_i T_J(\mathbf{x}_i) + 2m_{J-1}(\mathbf{x}_i) T_J(\mathbf{x}_i)) + T_J(\mathbf{x}_i)^2\right\} + \Omega(T_J) + c
\end{aligned}$$

All terms known at step J (i.e., indexed by $J-1$) vanish because they do not enter the optimisation scheme. The are embedded in the constant c.

Things are fairly simple with quadratic loss. For more complicated loss functions, Taylor expansions are used (see the original paper).

7.4.2 Penalisation

In order to go any further, we need to specify the way the penalisation works. For a given tree T, we specify its structure by $T(x) = w_{q(x)}$, where w is the output value of some leaf and $q(\cdot)$ is the function that maps an input to its final leaf. This encoding is illustrated in Figure 7.5. The function q indicates the path, while the vector $\mathbf{w} = w_i$ codes the terminal leaf values.

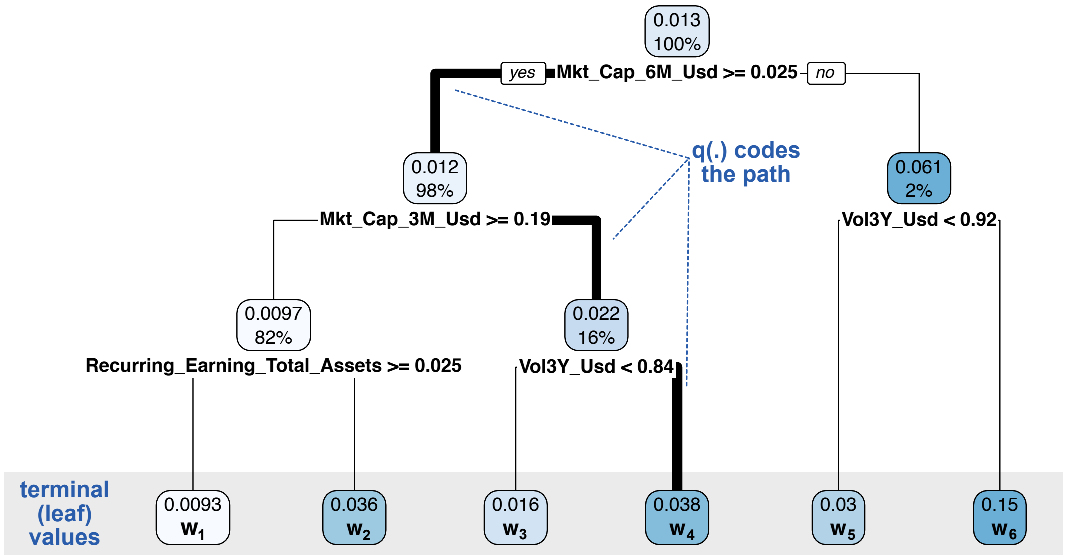


FIGURE 7.5: Coding a decision tree

We write $l = 1, \dots, L$ for the indices of the leafs of the tree. In XGBoost, complexity is defined as:

$$\Omega(T) = \gamma L + \frac{\lambda}{2}\sum_{l=1}^{L} w_l^2,$$

where

- the first term penalises the **total number of leaves**;
- the second term penalises the **magnitude of output values** (this helps reduce variance).

The first penalization reduces the depth of the tree while the second shrinks the size of the adjustments that will come from the latest tree.

7.4.3 Aggregation

We aggregate both sections of the objective (loss and penalization). We write I_l for the set of the indices of the instances belonging to leaf l. Then,

$$\begin{aligned}
O &= 2\sum_{i=1}^{I}\left\{-y_iT_J(\mathbf{x}_i)+m_{J-1}(\mathbf{x}_i)T_J(\mathbf{x}_i))+\frac{T_J(\mathbf{x}_i)^2}{2}\right\}+\gamma L+\frac{\lambda}{2}\sum_{l=1}^{L}w_l^2\\
&= 2\sum_{i=1}^{I}\left\{-y_iw_{q(\mathbf{x}_i)}+m_{J-1}(\mathbf{x}_i)w_{q(\mathbf{x}_i)})+\frac{w^2_{q(\mathbf{x}_i)}}{2}\right\}+\gamma L+\frac{\lambda}{2}\sum_{l=1}^{L}w_l^2\\
&= 2\sum_{l=1}^{L}\left(w_l\sum_{i\in I_l}(-y_i+m_{J-1}(\mathbf{x}_i))+\frac{w_l^2}{2}\sum_{i\in I_l}\left(1+\frac{\lambda}{2}\right)\right)+\gamma L
\end{aligned}$$

The function is of the form $aw_l+\frac{b}{2}w_l^2$, which has minimum values $-\frac{a^2}{2b}$ at point $w_l=-a/b$. Thus, writing $\#(.)$ for the cardinal function that counts the number of items in a set,

$$\rightarrow\quad w_l^*=\frac{\sum_{i\in I_l}(y_i-m_{J-1}(\mathbf{x}_i))}{\left(1+\frac{\lambda}{2}\right)\#\{i\in I_l\}},\text{ so that} \tag{7.5}$$

$$O_L(q)=-\frac{1}{2}\sum_{l=1}^{L}\frac{\left(\sum_{i\in I_l}(y_i-m_{J-1}(\mathbf{x}_i))\right)^2}{\left(1+\frac{\lambda}{2}\right)\#\{i\in I_l\}}+\gamma L,$$

where we added the dependence of the objective both in q (structure of tree) and L (number of leaves). Indeed, the meta-shape of the tree remains to be determined.

7.4.4 Tree structure

Final problem: the **tree structure**! Let us take a step back. In the construction of a simple regression tree, the output value at each node is equal to the average value of the label within the node (or cluster). When adding a new tree in order to reduce the loss, the nodes values must be computed completely differently, which is the purpose of Equation (7.5).

Nonetheless, the growing of the iterative trees follows similar lines as simple trees. Features must be tested in order to pick the one that minimizes the objective for each given split. The final question is then: what's the best depth and when to stop growing the tree? The method is to

- proceed node by node;
- for each node, look at whether a split is useful (in terms of objective) or not:

$$\text{Gain}=\frac{1}{2}\left(\text{Gain}_L+\text{Gain}_R+\text{Gain}_O\right)-\gamma$$

- each gain is computed with respect to the instances in each bucket (cluster):

$$\text{Gain}_{\mathcal{X}}=\frac{\left(\sum_{i\in I_{\mathcal{X}}}(y_i-m_{J-1}(\mathbf{x}_i))\right)^2}{\left(1+\frac{\lambda}{2}\right)\#\{i\in I_{\mathcal{X}}\}},$$

  where $I_{\mathcal{X}}$ is the set of instances within cluster $\mathcal{X}$.

Gain_O is the original gain (no split) and Gain_L and Gain_R are the gains of the left and right cluster, respectively. One work about the $-\gamma$ adjustment in the above formula: there is one unit of new leaves (two new minus one old)! This makes a one leaf difference, hence $\Delta L = 1$ and the penalization intensity for each new leaf is equal to γ.

Lastly, we underline the fact that XGBoost also applies a learning rate: each new tree is scaled by a factor η, with $\eta \in (0, 1]$. This is very useful because a pure aggregation of 100 optimized trees is the best way to overfit the training sample.

7.4.5 Extensions

Several additional features are available to further prevent boosted trees to overfit. Indeed, given a sufficiently large number of trees, the aggregation is able to match the training sample very well, but may fail to generalize well out-of-sample.

Following the pioneering work of Srivastava et al. (2014), the DART (Dropout for Additive Regression Trees) model was proposed by Rashmi and Gilad-Bachrach (2015). The idea is to omit a specified number of trees during training. The trees that are removed from the model are chosen randomly. The full specifications can be found at `https://xgboost.readthedocs.io/en/latest/tutorials/dart.html`.

Monotonicity constraints are another element that is featured both in xgboost and lightgbm. Sometimes, it is expected that one particular feature has a monotonous impact on the label. For instance, if one deeply believes in momentum, then past returns should have an increasing impact on future returns (in the cross-section of stocks).

Given the recursive nature of the splitting algorithm, it is possible to choose when to perform a split (according to a particular variable) and when not to. In Figure 7.6, we show how the algorithm proceeds. All splits are performed according to the same feature. For the first split, things are easy because it suffices to verify that the averages of each cluster are ranked in the right direction. Things are more complicated for the splits that occur below. Indeed, the average values set by all above splits matter as they give bounds for acceptable values for the future average values in lower splits. If a split violates these bounds, then it is overlooked and another variable will be chosen instead.

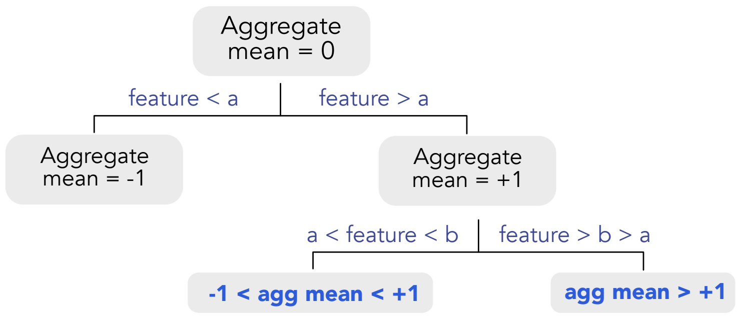


FIGURE 7.6: Imposing monotonic constraints. The constraints are shown in bold blue in the bottom leaves.

7.4.6 Code and results

In this section, we train a model using the *XGBoost* library. Other options include *catboost*, *gbm*, *lightgbm*, and *h2o*'s own version of boosted machines. Unlike many other packages, the XGBoost function requires a particular syntax and dedicated formats. The first step is thus to encapsulate the data accordingly.

Moreover, because training times can be long, we shorten the training sample as advocated in Coqueret and Guida (2019). We retain only the 40% most extreme observations (in terms of label values: top 20% and bottom 20%) and work with the small subset of features. In all coding sections dedicated to boosted trees in this book, the models will be trained with only 7 features.

```r
library(xgboost)                                            # The package for boosted trees
train_features_xgb <- training_sample %>%
    filter(R1M_Usd < quantile(R1M_Usd, 0.2) |
               R1M_Usd > quantile(R1M_Usd, 0.8)) %>%            # Extreme values only!
    select(features_short) %>% as.matrix()                  # Independent variable
train_label_xgb <- training_sample %>%
    filter(R1M_Usd < quantile(R1M_Usd, 0.2) |
               R1M_Usd > quantile(R1M_Usd, 0.8)) %>%
    select(R1M_Usd) %>% as.matrix()                         # Dependent variable
train_matrix_xgb <- xgb.DMatrix(data = train_features_xgb,
                                label = train_label_xgb)        # XGB format!
```

The second (optional) step is to determine the monotonicity constraints that we want to impose. For simplicity, we will only enforce three constraints on

1. market capitalization (negative, because large firms have smaller returns under the size anomaly);
2. price-to-book ratio (negative, because overvalued forms also have smaller returns under the value anomaly);
3. past annual returns (positive, because winner outperform losers under the momentum anomaly).

```r
mono_const <- rep(0, length(features))                     # Initialize the vector
mono_const[which(features == "Mkt_Cap_12M_Usd")] <- (-1) # Decreasing in market cap
mono_const[which(features == "Pb")] <- (-1)              # Decreasing in price-to-book
mono_const[which(features == "Mom_11M_Usd")] <- 1        # Increasing in past return
```

The third step is to train the model on the formatted training data.

```r
fit_xgb <- xgb.train(data = train_matrix_xgb,     # Data source
              eta = 0.3,                          # Learning rate
              objective = "reg:linear",           # Objective function
              max_depth = 4,                      # Maximum depth of trees
              lambda = 1,                         # Penalisation of leaf values
              gamma = 0.1,                        # Penalisation of number of leaves
              nrounds = 30,                       # Number of trees used (rather low here)
              monotone_constraints = mono_const,  # Monotonicity constraints
              verbose = 0                         # No comment from the algo
    )
```

Finally, we evaluate the performance of the model. Note that before that, a proper formatting of the testing sample is required.

```
xgb_test <- testing_sample %>%                                # Test sample => XGB format
    select(features_short) %>%
    as.matrix()
mean((predict(fit_xgb, xgb_test) - testing_sample$R1M_Usd)^2) # MSE
```

```
## [1] 0.03804396
```

```
mean(predict(fit_xgb, xgb_test) * testing_sample$R1M_Usd > 0) # Hit ratio
```

```
## [1] 0.5047146
```

The performance is comparable to those observed for other predictive tools. As a final exercise, we show one implementation of a classification task under XGBoost. Only the label changes. In XGBoost, labels must be coded with integer number, starting at zero exactly. In R, factors are numerically coded as integers numbers starting from one, hence the mapping is simple.

```
train_label_C <- training_sample %>%
    filter(R1M_Usd < quantile(R1M_Usd, 0.2) |          # Either low 20% returns
               R1M_Usd > quantile(R1M_Usd, 0.8)) %>%   # Or top 20% returns
    select(R1M_Usd_C)
train_matrix_C <- xgb.DMatrix(data = train_features_xgb,
                              label = as.numeric(train_label_C == "TRUE")) # XGB format!
```

When working with categories, the loss function is usually the softmax function (see Section 2.1).

```
fit_xgb_C <-  xgb.train(data = train_matrix_C,  # Data source (pipe input)
              eta = 0.8,                        # Learning rate
              objective = "multi:softmax",      # Objective function
              num_class = 2,                    # Number of classes
              max_depth = 4,                    # Maximum depth of trees
              nrounds = 10,                     # Number of trees used
              verbose = 0                       # No warning message
    )
```

We can then proceed to the assessment of the quality of the model. We adjust the prediction to the value of the true label and count the proportion of accurate forecasts.

```
mean(predict(fit_xgb_C, xgb_test) + 1 == as.numeric(testing_sample$R1M_Usd_C)) # Hit ratio
```

```
## [1] 0.495613
```

Consistently with the previous classification attempts, the results are underwhelming, as if switching to binary labels incurred a loss of information.

7.4.7 Instance weighting

In the computation of the aggregate loss, it is possible to introduce some flexibility and assign weight to instances:

$$O=\underbrace{\sum_{i=1}^I\mathcal{W}_i\times \text{loss}(y_i,\tilde{y}_i)}_{\text{weighted error term}} \quad + \quad \underbrace{\sum_{j=1}^J\Omega(T_j)}_{\text{regularisation term (unchanged)}} \quad .$$

In factor investing these weights can very well depend on the feature values ($\mathcal{W}_i=\mathcal{W}_i(\mathbf{x}_i)$). For instance, for one particular characteristic $\mathbf{x}^k$, weights can be increasing thereby giving more importance to assets with high values of this characteristic (e.g., value stocks are favored compared to growth stocks). One other option is to increase weights when the values of the characteristic become more extreme (deep value and deep growth stocks have larger weight). If the features are uniform, the weights can simply be $\mathcal{W}_i(x_i^k)\propto|x_i^k-0.5|$: firms with median value 0.5 have zero weight and as the feature value shifts towards 0 or 1, the weight increases. Specifying weights on instances biases the learning process just like views introduced à la Black and Litterman (1992) influence the asset allocation process. The difference is that the nudge is performed well ahead of the portfolio choice problem.

In xgboost, the implementation instance weighting is done very early, in the definition of the xgb.DMatrix:

```
inst_weights <- runif(nrow(train_features_xgb))             # Random weights
inst_weights <- inst_weights / sum(inst_weights)            # Normalization
train_matrix_xgb <- xgb.DMatrix(data = train_features_xgb,
                                label = train_label_xgb,
                                weight = inst_weights)      # Weights!
```

Then, in the subsequent stages, the optimization will be performed with these hard-coded weights. The splitting points can be altered (via the total weighted loss in clusters) and the terminal weight values (7.5) are also impacted.

7.5 Discussion

We end this chapter by a discussion on the choice of predictive engine with a view towards portfolio construction. As recalled in Chapter 3, the ML signal is just one building stage of construction of the investment strategy. At some point, this signal must be translated into portfolio weights.

From this perspective, simple trees appear suboptimal. Tree depth are usually set between 3 and 6. This implies between 8 and 64 terminal leaves at most, with possibly very unbalanced clusters. The likelihood of having one cluster with 20% to 30% of the sample is high. This means that when it comes to predictions, roughly 20% to 30% of the instances will be given the same value.

On the other side of the process, portfolio policies commonly have a fixed number of assets. Thus, having assets with equal signal does not permit to discriminate and select a subset to be included in the portfolio. For instance, if the policy requires exactly 100 stocks and 105 stocks have the same signal, the signal cannot be used for selection purposes. It would have to be combined with exogenous information such as the covariance matrix in a mean-variance type allocation.

Overall, this is one reason to prefer aggregate models. When the number of learners is sufficiently large (5 is almost enough), the predictions for assets will be unique and tailored to these assets. It then becomes possible to discriminate via the signal and select only those assets that have the most favorable signal. In practice, random forests and boosted trees are probably the best choices.

7.6 Coding exercises

8 Neural networks

Neural networks (NN) are an immensely rich and complicated topic. In this chapter, we introduce the simple ideas and concept behind the most simple architectures of NN. For more exhaustive treatments on NN idiosyncracies, we refer to the monographs Haykin (2009), Du and Swamy (2013) and Goodfellow et al. (2016). The latter is available freely online: www.deeplearningbook.org. For a practical introduction, we recommend the great book of Chollet (2017).

For starters, we briefly comment on the qualification "neural network". Most experts agree that the term is not very well chosen, as NN have little to do with how the human brain works (of which we know not that much). This explains why they are often referred to as "artificial neural networks" - we do not use the adjective for notational simplicity. Because we consider it more appropriate, we recall the definition of NN given by François Chollet: "*chains of differentiable, parameterised geometric functions, trained with gradient descent (with gradients obtained via the chain rule)*".

Early references of neural networks in finance are Bansal and Viswanathan (1993) and Eakins et al. (1998). Both have very different goals. In the first one, the authors aims to estimate a nonlinear form for the pricing kernel. In the second one, the purpose is to identify and quantify relationships between institutional investments in stocks and the attributes of the firms (an early contribution towards factor investing). An early review (Burrell and Folarin (1997)) lists financial applications of NN during the 1990s decade.

The pure predictive ability of NN in financial markets is a popular subject and we cite for example Kimoto et al. (1990), Enke and Thawornwong (2005), Zhang and Wu (2009) and Guresen et al. (2011).[1] This list is very far from exhaustive. More recent studies on neural networks include:

- Feng et al. (2019b) use neural network to find factors that are the best at explaining the cross-section of stock returns.

- Gu et al. (2018) map firm attributes and macro-economic variables into future returns. This creates a strong predictive tool that is able to forecast future returns very accurately.
- Chen et al. (2019) estimate the pricing kernel with a complex neural network structure including a generative adversarial network. This again gives crucial information on the structure of expected stock returns and can be used for portfolio construction (by building an accurate maximum Sharpe ratio policy).

[1]Neural networks have also been recently applied to derivatives pricing and hedging, see for instance the work of Buehler et al. (2019) and the survey by Ruf and Wang (2019).

8.1 The original perceptron

The origins of NN go back at least to Rosenblatt (1958). Its aim is binary classification. For simplicity, let us assume that the output is $\{0 = \text{do not invest}\}$ versus $\{1 = \text{invest}\}$ (e.g., derived from return, negative versus positive). Given the current nomenclature, a perceptron can be defined as an activated linear mapping. The model is the following:

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x}'\mathbf{w} + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

The vector of weights $\mathbf{w}$ scales the variables and the bias b shifts the decision barrier. Given values for b and w_i, the error is $\epsilon_i = y_i - 1_{\left\{\sum_{j=1}^J x_{i,j}w_j + w_0 > 0\right\}}$. As is customary, we set $b = w_0$ and add an initial constant column to x: $x_{i,0} = 1$, so that $\epsilon_i = y_i - 1_{\left\{\sum_{j=0}^J x_{i,j}w_j\right\}}$. In contrast to regressions, perceptrons do not have closed-form solutions. The optimal weights can only be approximated. Just like for regression, one way to derive good weights is to minimize the sum of squared errors. To this purpose, the simplest way to proceed is to

1. compute the current model value at point $\mathbf{x}_i$: $\tilde{y}_i = 1_{\left\{\sum_{j=0}^J w_j x_{i,j} > 0\right\}}$,
2. adjust the weight vector: $w_j \leftarrow w_j + \eta(y_i - \tilde{y}_i)x_{i,j}$,

which amounts to shifting the weights in the *right* direction. Just like for tree methods, the scaling factor η is the learning rate. A large η will imply large shifts: learning will be rapid but convergence may be slow or may even not occur. A small η is usually preferable as it helps reduce the risk of overfitting.

In Figure 8.1, we illustrate this mechanism. The initial model (dashed grey line) was trained on 7 points (3 red and 4 blue). A new black point comes in.

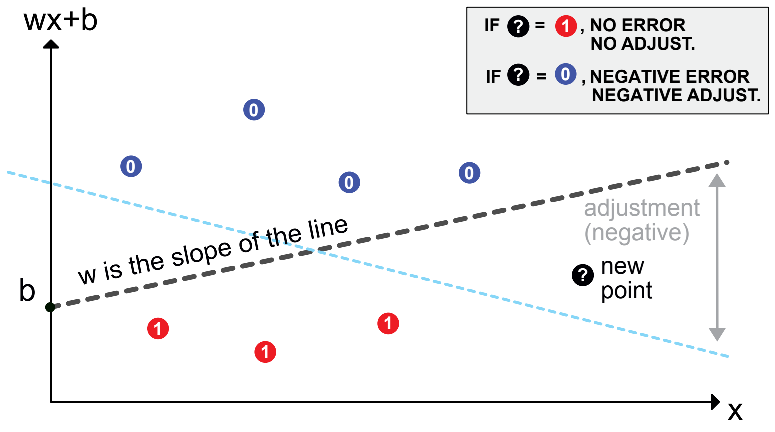

wx+b
IF ? = 1 , NO ERROR NO ADJUST.
IF ? = 0 , NEGATIVE ERROR NEGATIVE ADJUST.
0
0
0
0
w is the slope of the line
adjustment (negative)
b
? new point
1
1
1
x

FIGURE 8.1: Scheme of a perceptron.

- if the point is red, there is no need for adjustment: it is labelled correctly as it lies on the right side of the border.

- if the point is blue, then the model needs to be updated appropriately. Given the rule mentioned above, this means adjusting the slope of the line downwards. Depending on η, the shift will be sufficient to change the classification of the new point - or not.

At the time of its inception, the perceptron was an immense breakthrough which received an intense media coverage (see Olazaran (1996) and Anderson and Rosenfeld (2000)). Its rather simple structure was progressively generalized to networks (combinations) of perceptrons. Each one of them is a simple unit and units are gathered into layers. The next section describes the organization of simple multilayer perceptrons.

8.2 Multilayer perceptron

8.2.1 Introduction and notations

A perceptron can be viewed as a linear model to which is applied a particular function: the Heaviside (step) function. Other choices of functions are naturally possible. In the NN jargon, they are called activation functions. Their purpose is to introduce nonlinearity in otherwise very linear models.

Just like for random forest, the idea behind neural networks is to combine perceptron-like building blocks. A popular representation of neural networks is shown in Figure 8.2. This scheme is overly simplistic. It hides what is really going on: there is a perceptron in each green circle and each output is activated by some function before it is sent to the final output aggregation.

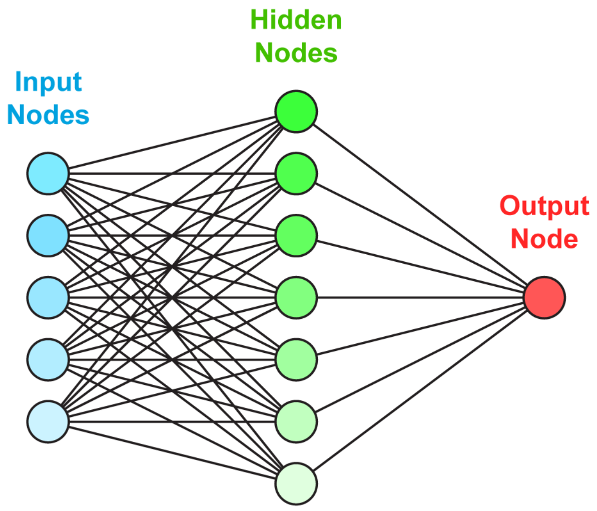


FIGURE 8.2: Simplified scheme of a multi-layer perceptron.

A more faithful account of what is going on is laid out in Figure 8.3.

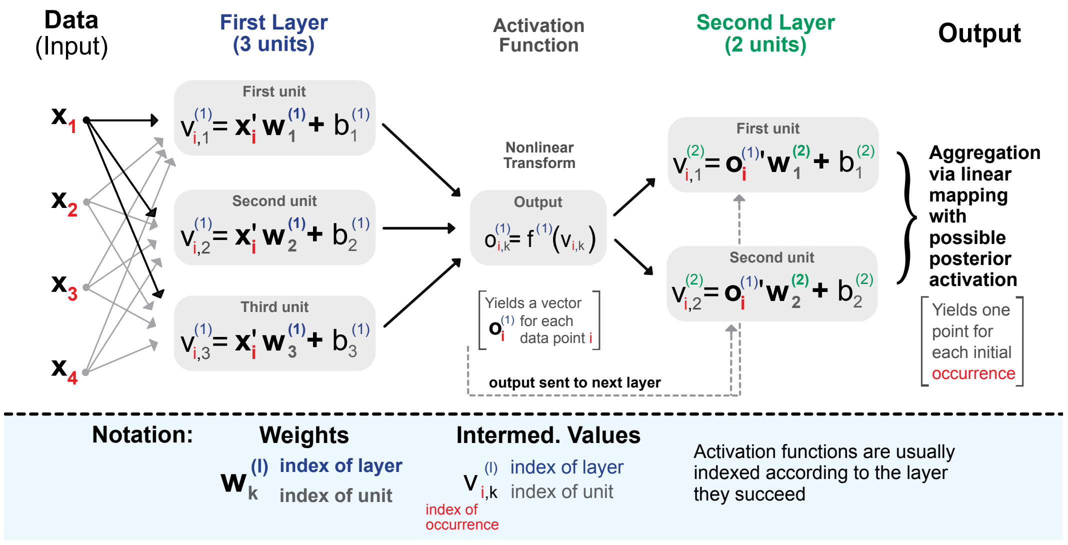

Data (Input)
First Layer (3 units)
Activation Function
Second Layer (2 units)
Output
x_1
x_2
x_3
x_4
First unit
$v^{(1)}_{i,1} = x_i' w^{(1)}_1 + b^{(1)}_1$
Second unit
$v^{(1)}_{i,2} = x_i' w^{(1)}_2 + b^{(1)}_2$
Third unit
$v^{(1)}_{i,3} = x_i' w^{(1)}_3 + b^{(1)}_3$
Nonlinear Transform
Output
$o^{(1)}_{i,k} = f^{(1)}(v_{i,k})$
Yields a vector $o^{(1)}_i$ for each data point i
output sent to next layer
First unit
$v^{(2)}_{i,1} = o^{(1)\prime}_i w^{(2)}_1 + b^{(2)}_1$
Second unit
$v^{(2)}_{i,2} = o^{(1)\prime}_i w^{(2)}_2 + b^{(2)}_2$
Aggregation via linear mapping with possible posterior activation
Yields one point for each initial occurrence
Notation:
Weights
$w^{(l)}_k$
(l) index of layer
index of unit
Intermed. Values
$v^{(l)}_{i,k}$
index of layer
index of unit
index of occurrence
Activation functions are usually indexed according to the layer they succeed

FIGURE 8.3: Detailed scheme of a perceptron with 2 intermediate layers.

Before we proceed with comments, we introduce some notation that will be used thoughout the chapter.

- The data is separated into a matrix $\mathbf{X} = x_{i,j}$ of features and a vector of output values $\mathbf{y} = y_i$. $\mathbf{x}$ or $\mathbf{x}_i$ denotes one line of $\mathbf{X}$.
- A neural network will have $L \geq 1$ layers and for each layer l, the number of units is $U_l \geq 1$.
- The weights for unit k located in layer l are denoted with $\mathbf{w}_k^{(l)} = w_{k,j}^{(l)}$ and the corresponding biases $b_k^{(l)}$. The length of $\mathbf{w}_k^{(l)}$ is equal to U_{l-1}. k refers to the location of the unit in layer l while j to the unit in layer $l-1$.
- Outputs (post activation) are denoted $o_{i,k}^{(l)}$ for instance i, layer l and unit k.

The process is the following. When entering the network, the data goes though the initial linear mapping:

$$v_{i,k}^{(1)} = \mathbf{x}_i'\mathbf{w}_k^{(1)} + b_k^{(1)}, \text{for } l = 1, \quad k \in [1, U_1],$$

and this linear transformation will be repeated (with different weights) for each layer of the network:

$$v_{i,k}^{(l)} = (\mathbf{o}_i^{(l-1)})'\mathbf{w}_k^{(l)} + b_k^{(l)}, \text{for } l \geq 2, \quad k \in [1, U_l].$$

The connexions between the layers are the so-called outputs, which are basically the linear mappings to which the activation functions have been applied. The output of layer l is the input of layer $l+1$.

$$o_{i,k}^{(l)} = f^{(l)}\left(v_{i,k}^{(l)}\right).$$

Finally, the terminal stage aggregates the outputs from the last layer:

$$\tilde{y}_i = f^{(L+1)}\left((\mathbf{o}_i^{(L)})'\mathbf{w}^{(L+1)} + b^{(L+1)}\right).$$

In the forward propagation of the input, the activation function naturally play an important role. In Figure 8.4, we plot the most usual activation function used by neural network libraries.

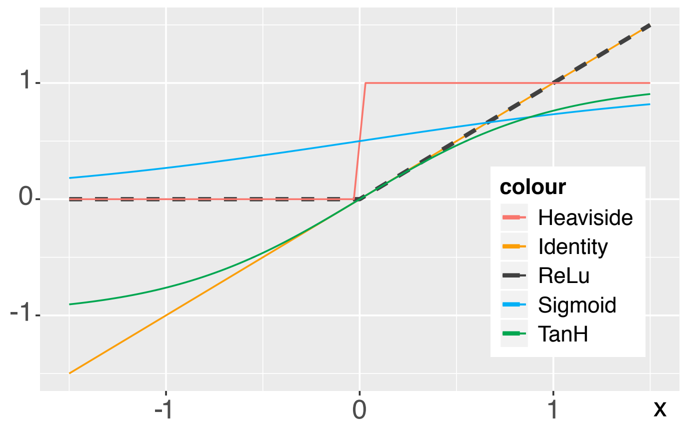


FIGURE 8.4: Plot of most common activation functions.

Let us rephrase the process through the lens of factor investing. The input $\mathbf{x}$ are the characteristics of the firms. The first step is to multiply their value by weights and add a bias. This is performed for all the units of the first layer. The output, which is a linear combination of the input is then transformed by the activation function. Each unit provides one value and all of these values are fed to the second layer following the same process. This is iterated until the end of the network. The purpose of the last layer is to yield an output shape that corresponds to the label: if the label is numerical, the output is a single number, if it is categorical, then usually it is a vector with length equal to the number of categories.

It is possible to use a final activation function after the output. This can have a huge importance on the result. Indeed, if the labels are returns, applying a sigmoid function at the very end will be disastrous because the sigmoid is always positive.

8.2.2 Universal approximation

One reason neural networks work well is that they are *universal approximators*. Given any bounded continuous function, there exists a one-layer network that can approximate this function up to arbitrary precision (see Cybenko (1989) and for early references, and section 4.2 in Du and Swamy (2013) and section 6.4.1 in Goodfellow et al. (2016) for more exhaustive lists of papers and Guliyev and Ismailov (2018) for recent results).

Formally, a one layer perceptron is defined by

$$f_n(\mathbf{x})=\sum_{l=1}^n c_l\phi(\mathbf{x}\mathbf{w}_l+\mathbf{b}_l)+c_0,$$

where ϕ is a (non constant) bounded continuous function. Then, for any $\epsilon>0$, it is possible to find one n such that for any continuous function f on the unit hypercube $[0,1]^d$,

$$|f(\mathbf{x})-f_n(\mathbf{x})|<\epsilon, \quad \forall \mathbf{x}\in[0,1]^d.$$

This result is rather intuitive: it suffices to add units to the layer to improve the fit. The process is more or less analogous to polynomial approximation, though some subtleties arise depending on the properties of the activations functions (boundedness, smoothness, convexity, etc.). We refer to Costarelli et al. (2016) for a survey on this topic.

The raw results on universal approximation imply that any well behaved function f can be approached sufficiently closely by a simple neural network, as long as the number of units can be arbitrarily large. Now, they do not directly relate to the learning phase, i.e., when the model is optimized with respect to a particular dataset. In a series of papers (Barron (1993) and Barron (1994) notably), Barron gives a much more precise characterization of what neural networks can achieve. In Barron (1993) it is for instance proved a more precise verion of universal approximation: for particular neural networks (with sigmoid activation), $\mathbb{E}[(f(\mathbf{x})-f_n(\mathbf{x}))^2]\le c_f/n$, which gives a speed of convergence related to the size of the network. In the expectation, the random term is $\mathbf{x}$: this corresponds to the case where the data is considered to be a sample of i.i.d. observations of a fixed distribution (this is the most common assumption in machine learning).

Below, we state one important result that is easy to interpret; it is taken from Barron (1994).

In the sequel, f_n corresponds to a possibly penalized neural network with only one intermediate layer with n units and sigmoid activation function. Moreover, both the supports of the predictors and the label are assumed to be bounded (which is not a major constraint). The most important metric in a regression exercise is the mean squared error (MSE) and the main result is a bound (in order of magnitude) on this quantity. For N randomly sampled i.i.d. points $y_i=f(x_i)+\epsilon_i$ on which f_n is trained, the best possible empirical MSE behaves like

$$\mathbb{E}\left[(f(x)-f_n(x))^2\right]=\underbrace{O\left(\frac{c_f}{n}\right)}_{\text{size of network}}+\underbrace{O\left(\frac{nK\log(N)}{N}\right)}_{\text{size of sample}}, \tag{8.1}$$

where K is the dimension of the input (number of columns) and c_f is a constant that depends on the generator function f. The above quantity provides a bound on the error that can be achieved by the best possible neural network given a dataset of size N.

There are clearly two components in the decomposition of this bound. The first one pertains to the complexity of the network. Just as in the original unviversal approximation theorem, the error decreases with the number of units in the network. But this is not enough! Indeed, the sample size is of course a key driver in the quality of learning (of i.i.d. observations). The second component of the bound indicates that the error decreases at a slightly slower pace with respect to the number of observations ($\log(N)/N$). and is linear in the number of units and the size of the input. This clearly underlines the link (trade-off?) between sample size and model complexity: having a very complex model is useless if the sample is small just like a simple model will not catch the fine relationships in a large dataset.

Overall, a neural network is a possibly very complicated function with a lot of parameters. In linear regressions, it is possible to increase the fit by spuriously adding exogenous variables. In neural networks, it suffices to increase the number of parameters by arbitrarily adding units to the layer(s). This is of course a very bad idea because high-dimensional networks will mostly capture the particularities of the sample they are trained on.

8.2.3 Learning via back-propagation

Just like for tree methods, neural networks are trained by minimizing some loss function subject to some penalization:

$$O = \sum_{i=1}^{I} \text{loss}(y_i, \tilde{y}_i) + \text{penalization},$$

where $\tilde{y}_i$ are the values obtained by the model and y_i are the *true* values of the instances. A simple requirement that eases computation is that the loss function be differentiable. The most common choices are the squared error for regression tasks and cross-entropy for classification tasks. We discuss the technicalities of classification in the next subsection.

The training of a neural network amounts to alter the weights (and biases) of all units in all layers so that O defined above is the smallest possible. To ease the notation and given that the y_i are fixed, let us write $D(\tilde{y}_i(\mathbf{W})) = \text{loss}(y_i, \tilde{y}_i)$, where $\mathbf{W}$ denotes the entirety of weights and biases in the network. The updating of the weights will be performed via gradient descent, i.e., via

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial D(\tilde{y}_i)}{\partial \mathbf{W}}. \tag{8.2}$$

This mechanism is the most classical in the optimization literature and we illustrate it in Figure 8.5. We highlight the possible suboptimality of large learning rates. In the diagram, the descent associated with the high η will oscillate around the optimal point whereas the one related to the small eta will converge more directly.

The complicated task in the above equation is to compute the gradient (derivative) which tell in which direction the adjustment should be done. The problem is that the successive nested layers and associated activations require many iterations of the chain rule for differentiation.

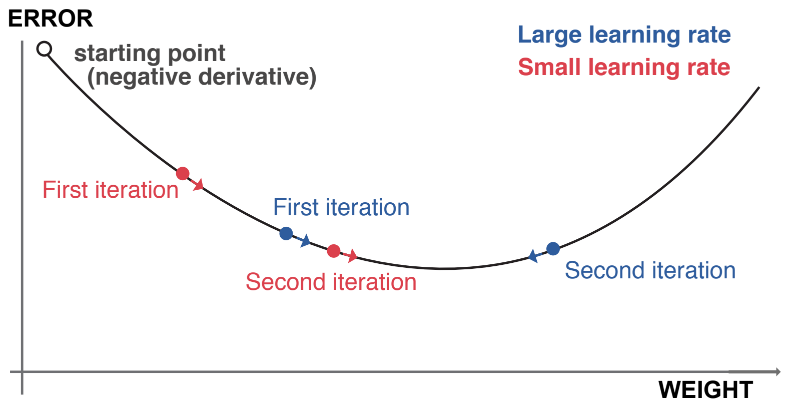


FIGURE 8.5: Outline of gradient descent.

The most common way to approximate a derivative is probably the finite difference method. Under the usual assumptions (the loss is twice differentiable), the centered difference satisfies:

$$\frac{\partial D(\tilde{y}_i(w_k))}{\partial w_k} = \frac{D(\tilde{y}_i(w_k + h)) - D(\tilde{y}_i(w_k - h))}{2h} + O(h^2),$$

where $h > 0$ is some arbitrarily small number. Inspite of its apparent simplicity, this method is costly computationally because it requires a number of operations of the magnitude of the number of weights.

Luckily, there is a small trick that can considerably ease and speed up the computation. The idea is to simply follow the chain rule and recycle terms along the way. Let us start by recalling

$$\tilde{y}_i = f^{(L+1)}\left((\mathbf{o}_i^{(L)})'\mathbf{w}^{(L+1)} + b^{(L+1)}\right) = f^{(L+1)}\left(b^{(L+1)} + \sum_{k=1}^{U_L} w_k^{(L+1)} o_{i,k}^{(L)}\right),$$

so that if we differentiate with the most immediate weights and biases, we get:

$$\frac{\partial D(\tilde{y}_i)}{\partial w_k^{(L+1)}} = D'(\tilde{y}_i)\left(f^{(L+1)}\right)'\left(b^{(L+1)} + \sum_{k=1}^{U_L} w_k^{(L+1)} o_{i,k}^{(L)}\right) o_{i,k}^{(L)} \tag{8.3}$$

$$= D'(\tilde{y}_i)\left(f^{(L+1)}\right)'\left(v_{i,k}^{(L+1)}\right) o_{i,k}^{(L)} \tag{8.4}$$

$$\frac{\partial D(\tilde{y}_i)}{\partial b^{(L+1)}} = D'(\tilde{y}_i)\left(f^{(L+1)}\right)'\left(b^{(L+1)} + \sum_{k=1}^{U_L} w_k^{(L+1)} o_{i,k}^{(L)}\right). \tag{8.5}$$

This is the easiest part. We must now go back one layer and this can only be done via the chain rule. To access layer L, we recall identity $v_{i,k}^{(L)} = (\mathbf{o}_i^{(L-1)})'\mathbf{w}_k^{(L)} + b_k^{(L)} = b_k^{(L)} + \sum_{j=1}^{U_L} o_{i,j}^{(L-1)} w_{k,j}^{(L)}$. We can then proceed:

$$\frac{\partial D(\tilde{y}_i)}{\partial w_{k,j}^{(L)}} = \frac{\partial D(\tilde{y}_i)}{\partial v_{i,k}^{(L)}} \frac{\partial v_{i,k}^{(L)}}{\partial w_{k,j}^{(L)}} = \frac{\partial D(\tilde{y}_i)}{\partial v_{i,k}^{(L)}} o_{i,j}^{(L-1)} \tag{8.6}$$

$$= \frac{\partial D(\tilde{y}_i)}{\partial o_{i,k}^{(L)}} \frac{\partial o_{i,k}^{(L)}}{\partial v_{i,k}^{(L)}} o_{i,j}^{(L-1)} = \frac{\partial D(\tilde{y}_i)}{\partial o_{i,k}^{(L)}} (f^{(L)})'(v_{i,k}^{(L)}) o_{i,j}^{(lL1)} \tag{8.7}$$

$$= \underbrace{D'(\tilde{y}_i)\left(f^{(L+1)}\right)'\left(v_{i,k}^{(L+1)}\right)}_{\text{computed above!}} w_k^{(L+1)} (f^{(L)})'(v_{i,k}^{(L)}) o_{i,j}^{(L-1)}, \tag{8.8}$$

where as we show in the last line, one part of the derivative was already computed in the previous step (Equation (8.4)). Hence, we can recycle this number and only focus on the right part of the expression.

The magic of the so-called backpropagation is that this will hold true for each step of the differentiation. When computing the gradient for weights and biases in layer l, there will be two parts: one that can be recycled from previous layers and another, local part, which depends only on the values and activation function of the current layer. A nice illustration of this process is given by the Google developer team: `https://google-developers.appspot.com/machine-learning/crash-course/backprop-scroll/`

When the data is formatted using tensors, it is possible to resort to vectorization so that the number of calls is limited to an order of the magnitude of the number of nodes (units) in the network.

The backpropagation algorithm can be summarized as follows. Given a sample of points (possibly just one):

1. the data flows from left as is described in Figure 8.3;
2. this allows the computation of the error or loss function;
3. all derivatives of this function (w.r.t. weights and biases) are computed, starting from the last layer and diffusing to the left (hence the term backpropagation);
4. all weights and biases can be updated to take the sample points into account (the model is adjusted to reduce the loss/error stemming from these points).

This operation can be performed any number of times with different sample sizes. We discuss this issue in Section 8.3.

The learning rate η can be refined. One option to reduce overfitting is to impose that after each epoch, the intensity of the update decreases. One possible parametric form is $\eta = \alpha e^{-\beta t}$, where t is the epoch and $\alpha, \beta > 0$. One further sophistication is to resort to so-called *momentum* (which originates from Polyak (1964)):

$$\begin{aligned} \mathbf{W}_{t+1} &\leftarrow \mathbf{W}_t - \mathbf{m}_t \quad \text{with} \\ \mathbf{m}_t &\leftarrow \eta \frac{\partial D(\tilde{y}_i)}{\partial \mathbf{W}_t} + \gamma \mathbf{m}_{t-1}, \end{aligned} \tag{8.9}$$

where t is the index of the weight update. The idea of momentum is to speed up the convergence by including a memory term of the last adjustment ($\mathbf{m}_{t-1}$) and going in the same direction in the current update. The parameter γ is often taken to be 0.9.

More complex and enhanced method have progressively developed:
- Nesterov (1983) improves the momentum term by forecasting the future shift in parameters;
- Adagrad (Duchi et al. (2011)) uses a different learning rate for each parameter;
- Adadelta (Zeiler (2012)) and Adam (Kingma and Ba (2014)) combine the ideas of Adagrad and momentum.

Lastly, in some degenerate case, some gradients may explode and push weights far from their optimal values. In order to avoid this phenomenon, learning libraries implement gradient clipping. The user specifies a maximum magnitude for gradients, usually expressed as a norm. Whenever the gradient surpasses this magnitude, it is rescaled to reach the authorized threshold. Thus, the direction remains the same, but the adjustment is smaller.

8.2.4 Further details on classification

In decision trees, the ultimate goal is to create homogeneous clusters, and the process to reach this goal was outlined in the previous chapter. For neural networks, things work differently because the objective is explicitly to minimize the error between the prediction $\tilde{\mathbf{y}}_i$ and a target label $\mathbf{y}_i$. Again, here $\mathbf{y}_i$ is a vector full of zeros with only one *one* denoting the class of the instance.

Facing a classification problem, the trick is to use an appropriate activation function at the very end of the network. The dimension of the terminal output of the network should be equal to J (number of classes to predict), and if, for simplicity, we write $\mathbf{x}_i$ for the values of this output, the most commonly used activation is the so-called *softmax* function:

$$\tilde{\mathbf{y}}_i = s(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^J e^{x_j}}.$$

The justification of this choice is straightforward: it can take any value as input (over the real line) and it sums to one over any (finite-valued) output. Similarly as for trees, this yields a 'probability' vector over the classes. Often, the chosen loss is a generalization of the entropy used for trees. Given the target label $\mathbf{y}_i = (y_{i,1}, \dots, y_{i,L}) = (0, 0, \dots, 0, 1, 0, \dots, 0)$ and the predicted output $\tilde{\mathbf{y}}_i = (\tilde{y}_{i,1}, \dots, \tilde{y}_{i,L})$, the cross-entropy is defined as

$$\text{CE}(\mathbf{y}_i, \tilde{\mathbf{y}}_i) = -\sum_{j=1}^J \log(\tilde{y}_{i,j}) y_{i,j}. \tag{8.10}$$

Basically, it is a proxy of the dissimilarity between its two arguments. One simple interpretation is the following. For the nonzero label value, the loss is $-\log(\tilde{y}_{i,l})$, while for all others, it is zero. In the log, the loss will be minimal if $\tilde{y}_{i,l} = 1$, which is exactly what we seek (i.e., $y_{i,l} = \tilde{y}_{i,l}$). In applications, this best case scenario will not happen, and the loss will simply increases when $\tilde{y}_{i,l}$ drifts away downwards from one.

8.3 How deep should we go? And other practical issues

Beyond the ones presented in the previous sections, the user faces many degrees of freedom when building a neural network. We present a few classical choices that are available when constructing and training neural networks.

8.3.1 Architectural choices

Arguably, the first choice pertains to the structure of the network. Beyond the dichotomy feed-forward versus recurrent (see Section 8.5), the immediate question is: how big (or how deep) the networks should be. First of all, let us calculate the number of parameters (i.e., weights plus biases) are estimated (optimized) in a network.

- For the first layer, this gives $(U_0 + 1)U_1$ parameters, where U_0 is the number of columns in $\mathbb{X}$ (i.e., number of explanatory variables) and U_1 is the number of units in the layer.
- For layer $l \in [2, L]$, the number of parameters is $(U_{l-1} + 1)U_l$.
- For the final output, there are simply $U_L + 1$ parameters.
- In total, this means the total number of values to optimise is

$$\mathcal{N} = \left(\sum_{l=1}^L (U_{l-1} + 1)U_l \right) + U_L + 1$$

As in any model, the number of parameters should be much smaller than the number of instances. There is no fixed ratio, but it is preferable if the sample size is *at least* ten times larger than the number of parameters. Below a ratio of 5, the risk of overfitting is high. Given the amount of data readily available, this constraint is seldom an issue, unless one wishes to work with a very large network.

The number of layers in current financial applications rarely exceed three or four. The number of units per layer is often chosen to follow the geometric pyramid rule (see, e.g., Masters (1993)). If there are L hidden layers, with I features in the input and O dimensions in the output (for regression tasks, $O=1$), then, for the k^{th} layer, a rule of thumb for the number of units is

$$U_k \approx \left\lfloor O\left(\frac{I}{O}\right)^{\frac{L+1-k}{L+1}}\right\rfloor.$$

If there is only one intermediate layer, the recommended proxy is the integer part of $\sqrt{IO}$. If not, the network starts with many units and the number of unit decreases exponentially towards the output size.

Several studies have shown that very large architectures do not always perform better than more shallow ones (e.g., Gu et al. (2018) and Orimoloye et al. (2019) for high frequency data, i.e., not factor-based). As a rule of thumb, a maximum of three hidden layers seem to be sufficient for prediction purposes.

8.3.2 Frequency of weight updates and learning duration

In the expression (8.2), it is implicit that the computation is performed for one given instance. If the sample size is very large (hundreds of thousands or millions of instances), updating the weights according to each point is computationally too costly. The updating is then performed on groups of instances which are called batches. The sample is (randomly) split into batches of fixed sizes and each update is performed following the rule:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial \sum_{i\in \text{batch}} D(\tilde{y}_i)/\text{card}(\text{batch})}{\partial \mathbf{W}}. \tag{8.11}$$

The change in weights is computed over the average loss computed over all instances in the batch. The terminology for training includes:

- **epoch**: one epoch is reached when each instance of the sample has contributed to the update of the weights (i.e., the training). Often, training a NN requires several epochs and up to a few dozen.
- **batch size**: the batch size is the number of samples used for one single update of weights.
- **iterations**: the number of iterations can mean alternatively the ratio of sample size divided by batch size or this ratio multiplied by the number of epochs. It's either the number of weight updates required to reach one epoch or the total number of updates during the whole training.

When the batch is equal to only one instance, the method is referred to as 'stochastic gradient descent' (SGD): the instance can be chosen randomly. When the batch size is strictly above one and below the total number of instances, the learning is performed via 'mini' batches.

8.3.3 Penalizations and dropout

At each level (layer), it is possible to enforce constraints or penalizations on the weights (and biases). Just as for tree methods, this helps slow down the learning to prevent overfitting on the training sample. Penalizations are enforced directly on the loss function and the objective function takes the form

$$O=\sum_{i=1}^I \text{loss}(y_i,\tilde{y}_i)+\sum_k \lambda_k||\mathbf{W}_k||_1+\sum_j\delta_k||\mathbf{W}_j||_2^2,$$

where the subscripts k and j pertain to the weights to which the L^1 and (or) L^2 penalization is applied.

In addition, specific constraints can be enforced on the weights directly during the training. Typically, two types of constraints are used:

- norm constraints: a maximum norm is fixed for the weight vectors or matrices;
- non-negativity constraint: all weights must be positive or zero.

Lastly, another (somewhat exotic) way to reduce the risk of overfitting is simply to reduce the size (number of parameters) of the model. Srivastava et al. (2014) propose to omit units during training (hence the term '**dropout**'). The weights of randomly chosen units are set to zero during training. All links from and to the unit are ignored, which mechanically shrinks the network. In the testing phase, all units are back, but the values (weights) must be scaled to account for the missing activations during the training phase.

8.4 Code samples and comments for vanilla MLP

There are several frameworks and libraries that allow robust and flexible constructions of neural networks. Among them, Keras and Tensorflow (developed by Google) are probably the most used at the time we write this book (PyTorch, from Facebook, is one alternative). For simplicity and because we believe it is the best choice, we implement the NN with Keras (which is the high level API of Tensorflow, see `https://www.tensorflow.org`). The original Python implementation is reference on `https://keras.io`, and the details for the R version can be found here: `https://keras.rstudio.com`. We recommend a thorough installation before proceeeding. Because the native versions of Tensorflow and Keras are written in Python (and accessed by R via the *reticulate* package), a running version of Python is required below. To install Keras, please follow the instructions provided at `https://keras.rstudio.com`.

In this section, we provide an detailed (though far from exhaustive) account of how to train neural networks with Keras. For the sake of completeness, we proceed in two steps. The first one relates to a very simple regression exercise. Its purpose is to get the reader familiar with the syntax of Keras. In the second step, we lay out many of the options proposed by keras to perform a classification exercise. With these two examples, we thus cover most of the mainstream topics falling under the umbrella of feed-forward multilayered perceptrons.

8.4.1 Regression example

Before we head to the core of the NN, a short stage of data preparation is required. Just as for penalized regressions (glmnet package) and boosted trees (xgboost package), the data must be sorted into four parts which are the combination of two dichotomies: training versus testing and labels versus features. We define the corresponding variables below. For simplicity, the first example is a regression exercise. A classification task will be detailed below.

```
NN_train_features <- select(training_sample, features) %>% as.matrix() # Matrix = important
NN_train_labels <- training_sample$R1M_Usd
NN_test_features <- select(testing_sample, features) %>% as.matrix()   # Matrix = important
NN_test_labels <- testing_sample$R1M_Usd
```

In Keras, the training of neural networks is performed through three steps:

1. Defining the structure/architecture of the network;

2. Setting the loss function and learning process (options on the updating of weights)
3. Train by specifying the batch sizes and number of rounds (epochs)

We start with a very simple architecture with two hidden layers.

```
library(keras)
# install_keras() # To complete installation
model <- keras_model_sequential()
model %>%   # This defines the structure of the network, i.e. how layers are organized
    layer_dense(units = 16, activation = 'relu', input_shape = ncol(NN_train_features)) %>%
    layer_dense(units = 8, activation = 'sigmoid') %>%
    layer_dense(units = 1) # No activation means linear activation: f(x) = x.
```

The definition of the structure is very intuitive and uses the *sequential* syntax in which one input is iteratively transformed by a layer until the last iteration which gives the output. Each layer depends on two parameters: the number of layers and the activation function that is applied to the output of the layer. One important point is the input_shape parameter for the first layer. It is required for the first layer and is equal to the number of features. For the subsequent layers, the input_shape is dictated by the number of units of the previous layer; hence it is not required. The activations that are currently available are listed on https://keras.io/activations/.

```
model %>% compile(                             # Model specification
    loss = 'mean_squared_error',               # Loss function
    optimizer = optimizer_rmsprop(),           # Optimisation method (weight updating)
    metrics = c('mean_absolute_error')         # Output metric
)
summary(model)                                 # Model architecture
```

```
## ___________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## ===========================================================================
## dense (Dense)                    (None, 16)                    1504
## ___________________________________________________________________________
## dense_1 (Dense)                  (None, 8)                     136
## ___________________________________________________________________________
## dense_2 (Dense)                  (None, 1)                     9
## ===========================================================================
## Total params: 1,649
## Trainable params: 1,649
## Non-trainable params: 0
## ___________________________________________________________________________
```

The summary of the model lists the layers in their order from input to output (forward pass). Because we are working with 93 features, the number of parameters for the first layer (16 units) is 93 plus one (for the bias) multiplied by 16, which makes 1504. For the second layer, the number of inputs is equal to the size of the output from the previous layer (16). Hence given the the second layer has 8 units, the total number of parameters is (16+1)*8 = 136.

We set the loss function to the standard mean squared error. Other losses are listed on https://keras.io/losses/: some of them work only for regressions (MSE, MAE) and others only for classification (categorical cross entropy, see Equation (8.10)). The RMS propragation optimizer is the classical mini-batch backpropagation implementation. For other weight updating algorithms, we refer to https://keras.io/optimizers/. The metric is the function used to assess the quality of the model. It can be different from the loss: for instance, using entropy for training and accuracy as the performance metric.

The final stage fits the model to the data and requires some additional training parameters:

```
fit_NN <- model %>%
    fit(NN_train_features,                                        # Training features
        NN_train_labels,                                          # Training labels
        epochs = 10, batch_size = 512,                            # Training parameters
        validation_data = list(NN_test_features, NN_test_labels) # Test data
)
plot(fit_NN)                                                                # Plot, evidently!
```

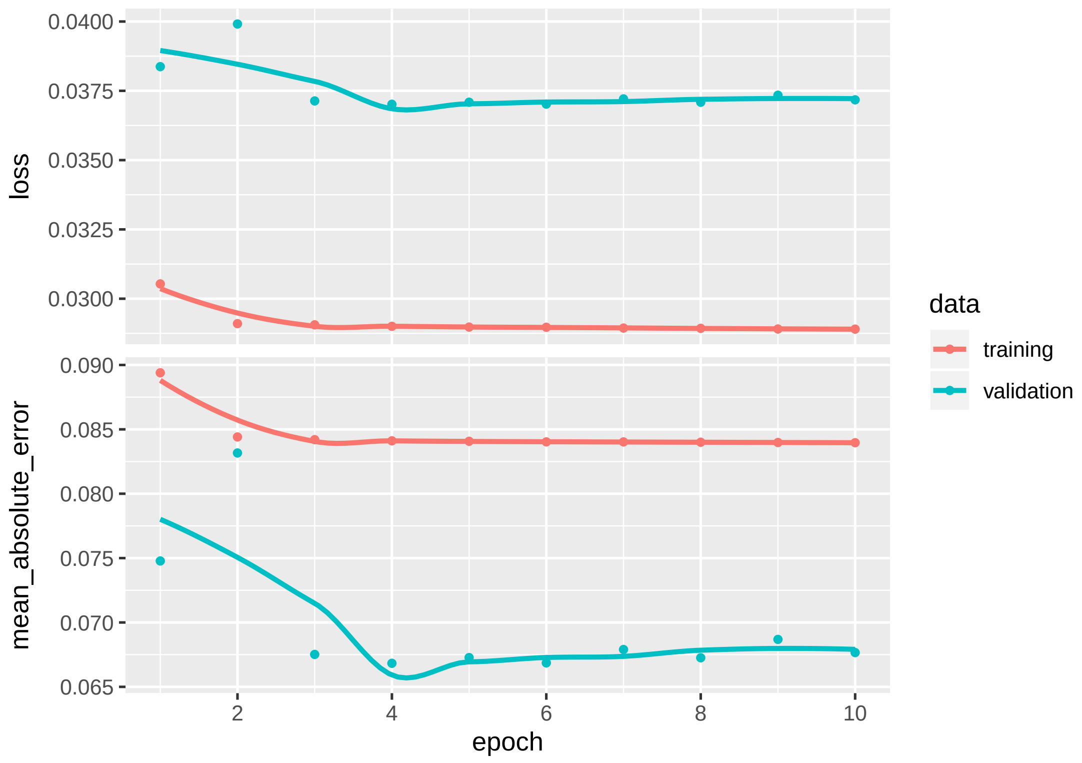


FIGURE 8.6: Output from a trained neural network (regression task)

In keras, the plot of the trained model shows four different curves (shown here in Figure 8.6). The top graph displays the improvement (or lack thereof) in loss as the number of epochs increases. Usually, the algorithm starts by learning rapidly and then converges to a point where any additional epoch does not improve. In the example above, this point arrives rather quickly because it is hard to notice any gain beyond the fourth epoch. The two colors show the performance on the two samples: the training sample and the testing sample. By construction, the loss will always improve (even marginally) on the training sample. When the impact is negligible on the testing sample (the curve is flat, as is the case here), the model fails to generalize out-of-sample: the gains obtained by training on the original sample do not translate to gains on previously unseen data, thus, the model seems to be learning noise.

The second graph shows the same behavior but computed using the metric function. The correlation (in absolute terms) between the two curves (loss and metric) is usually high. If one of them is flat, the other should be as well.

In order to obtain the parameters of the model, the user can call get_weights(model)[2] We do not execute the code here because the size of the output is much too large as their are thousands of weights.

Finally, from a practical point of view, the prediction is obtained via the usual predict() function. We use this function below on the testing sample to calculate the hit ratio.

```
mean(predict(model, NN_test_features) * NN_test_labels > 0) # Hit ratio
```

```
## [1] 0.5429438
```

Again, the hit ratio lies between 50% and 55%, which *seems* reasonably good. Most of the time, neural networks have their weights initialized randomly. Hence, two independently trained networks with the same architecture and same training data may well lead to very different predictions and performance! One way to bypass this issue is to freeze the random number generator. Models can also be easily exchanged by loading weights via the set_weights() function.

8.4.2 Classification example

We pursue our exploration of neural networks with a much more detailed example. The aim is to carry out a classification task on the binary label R1M_Usd_C. Before we proceed, we need to format the label properly. To this purpose, we resort to one-hot encoding (see Section 5.5.2).

```
library(dummies)                                          # Package for one-hot encoding
NN_train_labels_C <- training_sample$R1M_Usd_C %>% dummy() # One-hot encoding of the label
NN_test_labels_C <- testing_sample$R1M_Usd_C %>% dummy()   # One-hot encoding of the label
```

The labels NN_train_labels_C and NN_test_labels_C have two columns: the first flags the instances with above median return and the second flags those with below median returns. Note that we do not alter the feature variables: they remain unchanged. Below, we set the structure of the networks with many additional features compared to the first one.

[2]In case of package conflicts, use keras::get_weights(model). Indeed, another package in the machine learning landscape, *yardstick*, uses the function name "get_weights".

```
model_C <- keras_model_sequential()
model_C %>%   # This defines the structure of the network, i.e. how layers are organized
    layer_dense(units = 16, activation = 'tanh',               # Nb units & activation
                input_shape = ncol(NN_train_features),         # Size of input
                kernel_initializer = "random_normal",          # Initialization of weights
                kernel_constraint = constraint_nonneg()) %>%   # Weights should be nonneg
    layer_dropout(rate = 0.25) %>%                             # Dropping out 25% units
    layer_dense(units = 8, activation = 'elu',                 # Nb units & activation
                bias_initializer = initializer_constant(0.2),  # Initialization of biases
                kernel_regularizer = regularizer_l2(0.01)) %>% # Penalization of weights
    layer_dense(units = 2, activation = 'softmax')             # Softmax for categorical output
```

Before we start commenting on the many options used above, we highlight that keras models, unlike many R variables, are mutable objects. This means that any piping %>% after calling a model will alter it. Hence, successive trainings do not start from scratch but from the result of the previous training.

First, the options used above and below were chosen as illustrative examples and do not serve to particularly improve the quality of the model. The first change compared to Section 8.4.1 are the activation functions. The first two are simply new cases while the third one (for the output layer) is imperative. Indeed, since the goal is classification, the dimension of the output must be equal to the number of categories of the labels. The activation that yields a multivariate is the softmax function. Note that we must also specify the number of classes (categories) in the terminal layer.

The second major innovation are options pertaining to parameters. One family of options deals with the initialization of weights and biases. In keras, weights are referred to as the 'kernel'. The list of initializers is quite long and we suggest the interest reader has a look at the keras reference (https://keras.io/initializers/). Most of them are random, but some of them are constant.

Another family of options are the constraints and norm penalization that are applied on the weights and biases during training. In the above example, the weights of the first layer are coerced to be nonnegative while the weights of the second layer see their magnitude penalized by a factor (0.01) times their L^2 norm.

Lastly, the final novelty is the dropout layer (see Section 8.3.3) between the first and second layers. According to this layer, one fourth of the units in the first layer will be (randomly) omitted during training.

The specification of the training is outlined below.

```
model_C %>% compile(                                 # Model specification
    loss = 'binary_crossentropy',                    # Loss function
    optimizer = optimizer_adam(lr = 0.005,           # Optimisation method (weight updating)
                               beta_1 = 0.9,
                               beta_2 = 0.95),
    metrics = c('categorical_accuracy')              # Output metric
)
summary(model_C)                                     # Model structure
```

```
## ___________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## ===========================================================================
## dense_3 (Dense)                  (None, 16)                    1504
## ___________________________________________________________________________
## dropout (Dropout)                (None, 16)                    0
## ___________________________________________________________________________
## dense_4 (Dense)                  (None, 8)                     136
## ___________________________________________________________________________
## dense_5 (Dense)                  (None, 2)                     18
## ===========================================================================
## Total params: 1,658
## Trainable params: 1,658
## Non-trainable params: 0
## ___________________________________________________________________________
```

Here again, many changes have been made: all levels have been revised. The loss is now the crossentropy. Since we work with two categories, we resort to a specific choice (binary crossentropy) but the more general form is categorical_crossentropy and works for any number of classes (strictly above 1). The optimizer is also different and allows for several parameters and we refer to Kingma and Ba (2014). Simply put, the two beta parameters control decay rates for exponentially-weighted moving averages used in the update of weights. The two averages are estimates for the first and second moment of the gradient and can be exploited to increase the speed of learning. The performance metric in the above chunk is the categorical accuracy. In multiclass classification, the accuracy is defined as the average accuracy over all classes and all predictions. Since a prediction for one instance is a vector of weights, the 'terminal' prediction is the class that is associated with the largest weight. The accuracy then measures the proportion of times when the prediction is equal to the realized value.

Finally, we proceed with the training of the model.

```
fit_NN_C <- model_C %>%
    fit(NN_train_features,                                  # Training features
        NN_train_labels_C,                                  # Training labels
        epochs = 20, batch_size = 512,                      # Training parameters
        validation_data = list(NN_test_features,
                               NN_test_labels_C),           # Test data
        verbose = 0,                                        # No comments from algo
        callbacks = list(
            callback_early_stopping(monitor = "val_loss",   # Early stopping:
                                    min_delta = 0.001,      # Improvement threshold
                                    patience = 3,           # Nb epochs with no improvmt
                                    verbose = 0             # No warnings
                                    )
        )
    )
plot(fit_NN_C)
```

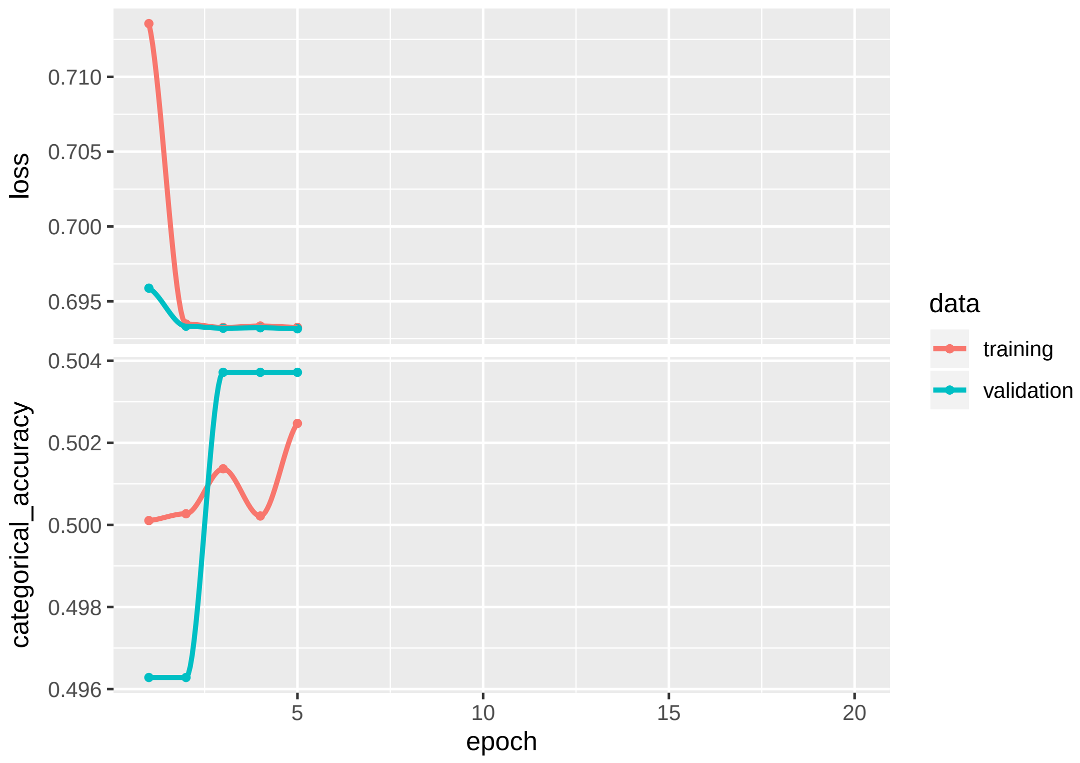


FIGURE 8.7: Output from a trained neural network (classification task) with early stopping

There is only one major difference here compared to the previous training call. In keras, callbacks are functions that can be used at given stages of the learning process. In the above example, we use one such function to stop the algorithm when no progress has been made for some time.

When datasets are large, the training can be long, especially when batch sizes are small and/or the number of epochs is high. It is not guaranteed that going to the full number of epochs is useful, as the loss or metric functions may be plateauing much sooner. Hence, can be very convenient to stop the process if no improvement is achieved during a specified time frame. We set the number of epochs to 20, but the process will likely stop before that.

In the above code, the improvement is focused on validation accuracy ("val_acc"; one alternative is "val_loss"). The min_delta value sets the minimum improvement that needs to be attained for the algorithm to continue. Therefore, unless the validation accuracy gains 0.001 points at each epoch, the training will stop. Nevertheless, some flexibility is introduced via the patience parameter, which in our case asserts that the hatling decision is made only after three consecutive epochs with no improvement. In the option, the verbose parameter dictates the amount of comments that is made by the function. For simplicity, we do not want any comments, hence this value is set to zero.

In Figure 8.7, the two graphs yield very different curves. One reason for that is the scale of the second graph. The range of accuracies is very narrow. Any change in this range does not represent much variation overall. The pattern is relatively clear on the training sample: the loss decreases while the accuracy improves. Unfortunately, this does not translate to the testing sample which indicates that the model does not generalize well out-of-sample.

8.5 Recurrent networks

8.5.1 Presentation

Multilayer perceptrons are feedforward networks because the data flows from left to right with no looping in between. For some particular tasks with sequential linkages (e.g., time-series or speech recognition), it might be useful to keep track of what happened with the previous sample (i.e., there is a natural ordering). One simple way to model 'memory' would be to consider the following network with only one intermediate layer:

$$\tilde{y}_i = f^{(y)}\left(\sum_{j=1}^{U_1} h_{i,j} w_j^{(y)} + b^{(2)}\right)$$

$$h_i = f^{(h)}\left(\sum_{k=1}^{U_0} x_{i,k} w_k^{(h,1)} + b^{(1)} + \underbrace{\sum_{k=1}^{U_1} w_k^{(h,2)} h_{i-1,k}}_{\text{memory part}}\right)$$

These kinds of models are often referred to as Elman (1990) models or to Jordan (1997) models if in the latter case h_{i-1} is replaced by y_{i-1} in the computation of h_i. Both types of models fall under the overarching umbrella of Recurrent Neural Networks (RNNs).

The h_i is usually called the state or the hidden layer. The training of this model is complicated and must be done by unfolding the network over all instances to obtain a simple feed-forward network and train it regularly. We illustrate the unfolding principle in Figure 8.8. It shows a very deep network. The first input impacts the first layer and then the second one via h_1 and all following layers in the same fashion. Likewise, the second inputs impacts all layers except the first and each instance $i-1$ is going to impact the output $\tilde{y}_i$ and all outputs $\tilde{y}_j$ for $j \geq i$. In Figure 8.8, the parameters that are trained are shown in blue. They appear many times, in fact, at each level of the unfolded network.

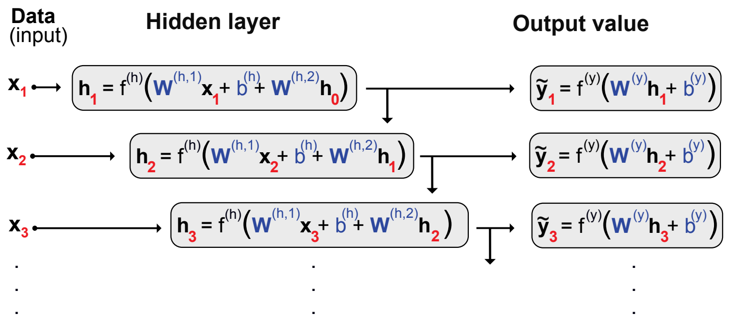


FIGURE 8.8: Unfolding a recurrent network.

The main problem with the above architecture is the loss of memory induced by **vanishing gradients**. Because of the depth of the model, the chain rule used in the backpropagation with imply a large number of products of derivatives of activation functions. Now, as is shown in Figure 8.4, these functions are very smooth and their derivatives are most of the time smaller than one (in absolute value). Hence, multiplying many numbers smaller than one leads to very small figures: beyond some layers, the learning does not propagate because the adjustments are too small.

One way to prevent this progressive discounting of the memory was introduced in Hochreiter and Schmidhuber (1997) (Long Short Term Memory - LSTM model). This model was subsequently simplified by the authors Chung et al. (2015) and we present this more parcimonious model below. The Gated Recurrent Unit is a slightly more complicated version of the vanilla recurrent network defined above. It has the following representation:

$$\begin{aligned}
\tilde{y}_i &= z_i\tilde{y}_{i-1} + (1-z_i)\tanh\left(\mathbf{w}_y'\mathbf{x}_i + b_y + u_y r_i \tilde{y}_{i-1}\right) \quad && \text{output (prediction)} \\
z_i &= \text{sig}(\mathbf{w}_z'\mathbf{x}_i + b_z + u_z\tilde{y}_{i-1}) && \text{`update gate'} \in (0,1) \\
r_i &= \text{sig}(\mathbf{w}_r'\mathbf{x}_i + b_r + u_r\tilde{y}_{i-1}) && \text{`reset gate'} \in (0,1).
\end{aligned}$$

In compact form, this gives

$$\tilde{y}_i = \underbrace{z_i}_{\text{weight}} \underbrace{\tilde{y}_{i-1}}_{\text{past value}} + \underbrace{(1-z_i)}_{\text{weight}} \underbrace{\tanh\left(\mathbf{w}_y'\mathbf{x}_i + b_y + u_y r_i \tilde{y}_{i-1}\right)}_{\text{candidate value (classical RNN)}},$$

where the z_i decides the optimal mix between the current and past values. For the candidate value, r_i decides which amount of past/memory to retain. r_i is commonly referred to as the '*reset gate*' and z_i to the '*update gate*'.

There are some subtleties in the training of a recurrent network. Indeed, because of the chaining between the instances, each batch must correspond to a coherent time-series. A logical choice is thus one batch per asset with instances (logically) chronologically ordered. Lastly, one option in some frameworks is to keep some memory between the batches by passing the final value of $\tilde{y}_i$ to the next batch (for which it will be $\tilde{y}_0$). This is often referred to as the stateful mode and should be considered meticulously. It does not seem desirable in a portfolio prediction setting if the batch size corresponds to all observations for each asset: there is no particular link between assets. If the dataset is divided into several parts for each given asset, then the training must be handled very cautiously.

Reccurrent networks and LSTM especially have been found to be good forecasting tools in financial contexts (see, e.g., Fischer and Krauss (2018) and Wang et al. (2019)).

8.5.2 Code and results

Recurrent networks are theoretically more complicated compared to multilayered perceptrons. In practice, they are also more challenging in their implementation. Indeed, the serial linkages require more attention compare to feedforward architectures. In an asset pricing framework, we must separate the assets because the stock-specific time series cannot be bundled together. The learning will be sequential, one stock at a time.

The dimensions of variables are crucial. In keras, they are defined for RNNs as:

1. The size of the batch. In our case, it will be the number of assets. Indeed, the recurrence relationship holds at the asset level, hence each asset will represent a new batch on which the model will learn.

2. The timesteps. In our case, it will simply be the number of dates.

3. The number of features. In our case, there is only one possible figure: the number of predictors.

For simplicity and in order to reduce computation times, we will use the same subset of stocks as that from Section 6.2.2. This yields a perfectly rectangular dataset in which all dates have the same number of observations.

First, we create some new, intermediate variables.

```r
data_rnn <- data_ml %>%                                # Dedicated dataset
    filter(stock_id %in% stock_ids_short)
training_sample_rnn <- filter(data_rnn, date < separation_date)
testing_sample_rnn <- filter(data_rnn, date > separation_date)
nb_stocks <- length(stock_ids_short)                   # Nb stocks
nb_feats <- length(features)                           # Nb features
nb_dates_train <- nrow(training_sample) / nb_stocks    # Nb training dates (size of sample)
nb_dates_test <- nrow(testing_sample) / nb_stocks      # Nb testing dates
```

Then, we construct the variables we will pass as arguments. We recall that the data file was ordered first by stocks and then by date (see Section 2.2).

```r
train_features_rnn <- array(NN_train_features,         # Formats the training data into array
                            dim = c(nb_dates_train, nb_stocks, nb_feats)) %>% # Tricky order
    aperm(c(2,1,3))                                    # The order is: stock, date, feature
test_features_rnn <- array(NN_test_features,           # Formats the testing data into array
                            dim = c(nb_dates_test, nb_stocks, nb_feats)) %>%  # Tricky order
    aperm(c(2,1,3))                                    # The order is: stock, date, feature
train_labels_rnn <- as.matrix(NN_train_labels) %>%
    array(dim = c(nb_dates_train, nb_stocks, 1)) %>% aperm(c(2,1,3))
test_labels_rnn <- as.matrix(NN_test_labels) %>%
    array(dim = c(nb_dates_test, nb_stocks, 1)) %>% aperm(c(2,1,3))
```

Finally, we move towards the training part. For simplicity, we only consider a simple RNN with only one layer. The structure is outlined below. In terms of recurrence structure, we pick a Gated Recurrent Unit.

```r
model_RNN <- keras_model_sequential() %>%
    layer_gru(units = 16,                             # Nb units in hidden layer
              batch_input_shape = c(nb_stocks,        # Dimensions = tricky part!
                                    nb_dates_train,
                                    nb_feats),
              activation = 'tanh',                    # Activation function
              return_sequences = TRUE) %>%            # Return all the sequence
    layer_dense(units = 1)                            # Final aggregation layer
model_RNN %>% compile(
    loss = 'mean_squared_error',                      # Loss = quadratic
    optimizer = optimizer_rmsprop(),                  # Backprop
    metrics = c('mean_absolute_error')                # Output metric MAE
)
```

There are many options available for recurrent layers. For GRUs, we refer to the keras documentation `https://keras.rstudio.com/reference/layer_gru.html`. We comment briefly on the option return_sequences which we activate. In many cases, the output is simply the terminal value of the sequence. If we do not require all of the sequence to be returned, we will face a problem in the dimensionality because the label is indeed a full sequence. Once the structure is determined, we can move forward to the training stage.

```
fit_RNN <- model_RNN %>% fit(train_features_rnn,    # Training features
                  train_labels_rnn,                 # Training labels
                  epochs = 10,                      # Number of rounds
                  batch_size = nb_stocks,           # Length of sequences
                  verbose = 0)                      # No comments
plot(fit_RNN)
```

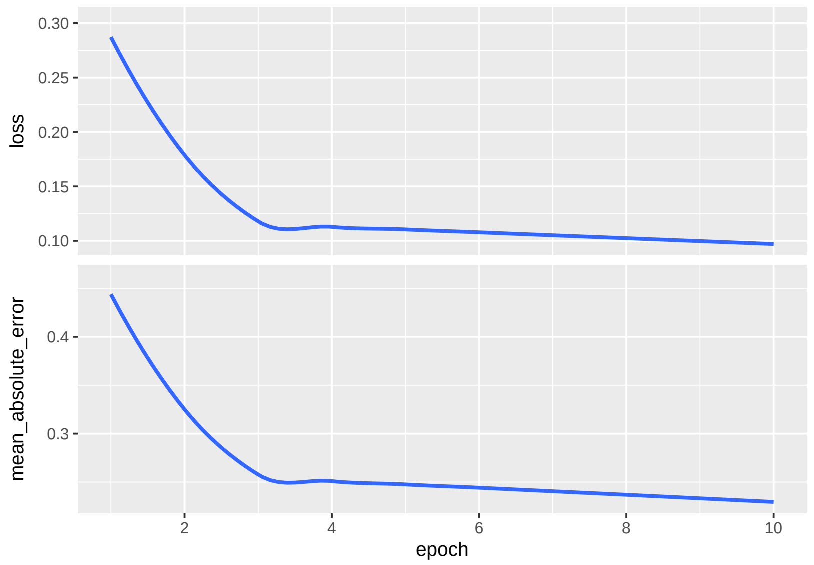


FIGURE 8.9: Output from a trained recurrent neural network (regression task)

Compared to our previous models, the major difference both in the ouptut (the graph on Figure 8.9) and the input (the code) is the absence of validation (or testing) data. One reason for that is because keras is very restrictive on RNNs and imposes that both the training and testing samples share the same dimensions. In our situation this is obviously not the case, hence we must bypass this obstacle by duplicating the model.

```
new_model <- keras_model_sequential() %>%
    layer_gru(units = 16,
              batch_input_shape = c(nb_stocks,          # New dimensions
                                    nb_dates_test,
                                    nb_feats),
              activation = 'tanh',                      # Activation function
              return_sequences = TRUE) %>%              # Passing last state to next batch
    layer_dense(units = 1)                              # Output dimension
new_model %>% keras::set_weights(keras::get_weights(model_RNN))
```

Finally, once the new model is ready - and with the matching dimensions, we can push forward to predicting the test values. We resort to the predict() function and immediately compute the hit ratio obtained by the model.

```
pred_rnn <- predict(new_model, test_features_rnn, batch_size = nb_stocks) # Predictions
mean(c(t(as.matrix(pred_rnn))) * test_labels_rnn > 0)          # Hit ratio
```

```
## [1] 0.5017483
```

The hit ratio is close to 50%, hence the model does hardly better than coin tossing.

8.6 Other common architectures

In this section, we present other network structures. Because they are less mainstream and often harder to implement, we do not propose code examples and stick to theoretical introductions.

8.6.1 Generative adversarial networks

The idea of Generative Adversarial Networks (GANs) is to improve the accuracy of a classical neural network by trying to fool it. This very popular idea was introduced by Goodfellow et al. (2014). Imagine you are an expert in Picasso paintings and that you boast about being able to easily recognize any piece of work from the painter. One way to refine your skill is to test them against a counterfeiter. A true expert should be able to discriminate between a true original Picasso and one emanating from a forger. This is the principle of GANs.

GANs consist in two neural networks: the first one tries to learn and the second one tries to fool the first (induce it into error). Just like in the example above, there are also two sets of data: one ($\mathbf{x}$) is true (or correct), stemming from a classical training sample and the other one ($\mathbf{z}$) is fake and generated by the counterfeiter network.

In the GAN nomenclature the networks that learns is D because it's suppose to discriminate while the forger is G because it generates false data. In their original formulation, GANs are aimed at classifying. To ease the presentation, we keep this scope. The discriminant network has a simple (scalar) output: the probability that its input comes from true data (versus fake data). The input of G is some arbitrary noise and its output has the same shape/form as the input of D.

We state the theoretical formula of a GAN directly and comment on it below. D and G play the following minimax game:

$$\min_G \max_D \; \{\mathbb{E}[\log(D(\mathbf{x}))] + \mathbb{E}[\log(1 - D(G(\mathbf{z})))]\}\,. \tag{8.12}$$

First, let us decompose this expression in its two parts (the optimizers). The first part (i.e. the first max) is the classical one: the algorithm seeks to maximize the probability of assigning the correct label to all examples it seeks to classify. As is done in economics and finance, the program does not maximize $D(\mathbf{x})$ itself on average, but rather a functional form (like a utility function).

On the left side, since the expectation is driven by $\mathbf{x}$, the objective must be increasing in the output. On the right side, where the expectation is taken over the fake instances, the right classification is the opposite, i.e., $1 - D(G(\mathbf{z})$.

The second, overarching, part seeks to minimize the performance of the algorithm on the simulated data: it aims at shrinking the odds that D finds out that the data is indeed corrupt. A summarized version of the structure of the network is provided below.

In ML-based asset pricing, the most notable application of GANs was introduced in Chen et al. (2019). Their aim is to make use of the method of moment expression

$$\mathbb{E}[M_{t+1} r_{t+1,n} g(I_t, I_{t,n})] = 0,$$

$$\left.\begin{array}{r}\text{training sample} = \mathbf{x} = \text{true data} \\ \text{noise} = \mathbf{z} \overset{G}{\to} \quad \text{fake data}\end{array}\right\} \overset{D}{\to} \text{output} = \text{probability for label}$$

FIGURE 8.10: Scheme of a GAN.

which is an application of Equation (4.7) where the instrumental variables $I_{t,n}$ are firm-dependent (e.g., characteristics and attributes) while the I_t are macro-economic variables (aggregate dividend yield, volatility level, credit spread, term spread, etc.). The trick is to model the SDF as an unknown combination of assets $M_{t+1} = 1 - \sum_{n=1}^N w(I_t, I_{t,n})r_{t+1,n}$. The primary discriminatory network (D) is the one that approximates the SDF via the weights $w(I_t, I_{t,n})$. The secondary generative network is the one that creates the moment condition through $g(I_t, I_{t,n})$ in the above equation.

The full specification of the network is given by the program:

$$\min_w \max_g \sum_{j=1}^N \left\| \mathbb{E}\left[\left(1 - \sum_{n=1}^N w(I_t, I_{t,n})r_{t+1,n}\right) r_{t+1,j}\, g(I_t, I_{t,j})\right]\right\|^2$$

The asset pricing equations (moments) are not treated as equalities but as a relationship that is approximated. The network defined by $\mathbf{w}$ is the asset pricing modeler and tries to determine the best possible model while the network defined by $\mathbf{g}$ seeks to find the worst possible conditions so that the model performs badly. We refer to the original article for the full specification of both networks. In their empirical section, Chen et al. (2019) report that adopting a strong structure driven by asset pricing imperatives add values compared to a pure predictive 'vanilla' approach such as the one detailed in Gu et al. (2018). The out-of-sample behavior of decile sorted portfolios (based on the model's prediction) display a monotonous patter with respect to the order of the deciles.

8.6.2 Auto-encoders

In the recent literature auto-encoders are used in Huck (2019) (portfolio management), and Gu et al. (2019) (asset pricing).
Autoencoders are a strange family of neural networks because they are classified among non-supervised algorithms. In the supervised jargon, their label is equal to the input. Like GANS, autoencoders consist of two networks, though the structure is very different: the first network encodes the input into some intermediary output (usually called the code) and the second network decodes the code into a modified version of the input.

$$\begin{array}{ccccc}\mathbf{x} & \xrightarrow{E} & \mathbf{z} & \xrightarrow{D} & \mathbf{x}' \\ \text{input} & \text{encoder} & \text{code} & \text{decoder} & \text{modified input}\end{array}$$

Because auto-encoders do not belong to the large family of supervised algorithms, we postpone their presentation to Section 16.2.3.

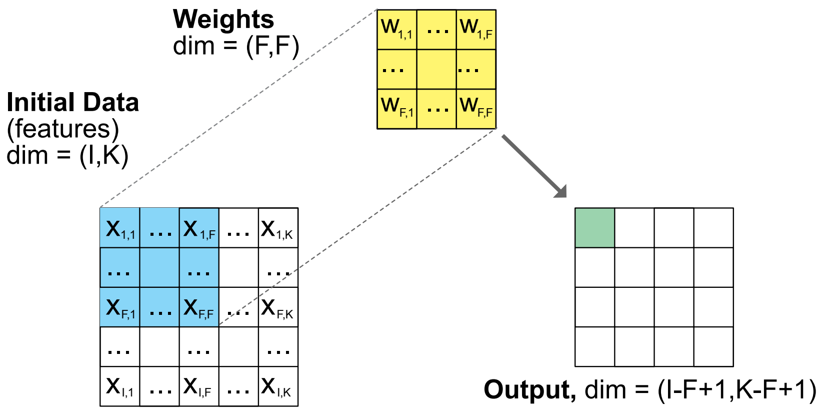
8.6.3 A word on convolutional networks

Neural networks gained popularity during the 2010 decade thanks to a series of successes in computer vision competitions. The algorithms behind these advances are convolutional neural networks (CNN). While they may seem a surprising choice for financial predictions, several teams of researchers in the Computer Science field have proposed approaches that rely on this variation of neural networks (Chen et al. (2016), Loreggia et al. (2016), Dingli and Fournier (2017), Tsantekidis et al. (2017), Hoseinzade and Haratizadeh (2019)). Hence, we briefly present the principle in this final section on neural networks. We lay out the presentation for CNNs of dimension two, but they are can also be used in dimension one or three.

The reason why CNNs are useful is because they allow to progressively reduce the dimension of a large dataset by keeping local information. An image is a rectangle of pixels. Each pixel is usually coded via three layers, one for each color: red, blue and green. But to keep things simple, let's just consider one layer of, say 1,000 by 1,000 pixels with one value for each pixel. In order to analyze the content of this image, a **convolutional layer** will simplify by scanning the values using rectangles with arbitrary weights.

Figure 8.11 sketches this process (it is strongly inspired by Hoseinzade and Haratizadeh (2019)). The original data is a matrix $(I \times K)$ $x_{i,k}$ and the weights are also a matrix $w_{j,l}$ of size $(J \times L)$ with $J < I$ and $L < K$. The scanning transforms each rectangle of size $(J \times L)$ into one real number. Hence, the output has a smaller size: $(I - J + 1) \times (K - L + 1)$. If $I = K = 1,000$ and $J = L = 201$, then the output has dimension (800×800) which is already much smaller. The output values are given by

$$o_{i,k} = \sum_{j=1}^{J} \sum_{l=1}^{L} w_{j,l} x_{i+j-1,k+l-1}.$$



Weights
dim = (F,F)
$W_{1,1}$ … $W_{1,F}$
… …
$W_{F,1}$ … $W_{F,F}$
Initial Data
(features)
dim = (I,K)
$X_{1,1}$ … $X_{1,F}$ … $X_{1,K}$
… … …
$X_{F,1}$ … $X_{F,F}$ … $X_{F,K}$
… … …
$X_{I,1}$ … $X_{I,F}$ … $X_{I,K}$
Output, dim = (I-F+1,K-F+1)

FIGURE 8.11: Scheme of a convolutional unit. Note: the dimensions are general and do not correspond to the number of squares.

Iteratively reducing the dimension of the output via sequences of convolutional layers like the one presented above would be costly in computation and could give rise to overfitting because the number of weights would be incredibly large. In order to efficiently reduce the size of outputs, **pooling layers** are often used. The job of pooling units is to simplify matrices by reducing them to a simple metric such as the minimum, maximum or average value of the matrix:

$$o_{i,k} = f(x_{i+j-1,k+l-1}, 1 \leq j \leq J, 1 \leq l \leq L),$$

where f is the minimum, maximum or average value. We show examples of pooling in Figure 8.12 below. In order to increase the speed of compression, it is possible to add a stride to omit cells. A stride value of v will perform the operation only every v value and hence bypass intermediate steps. In Figure 8.12, the two cases on the left do not resort to pooling, hence the reduction in dimension is exactly equal to the size of the pooling size. When stride is into action (right pane), the reduction is more marked. From a 1,000 by 1,000 input, a 2-by-2 pooling layer with stride 2 will yield a 500 by 500 output: the dimension is shrinked four-fold, as in the right scheme of Figure 8.12 .

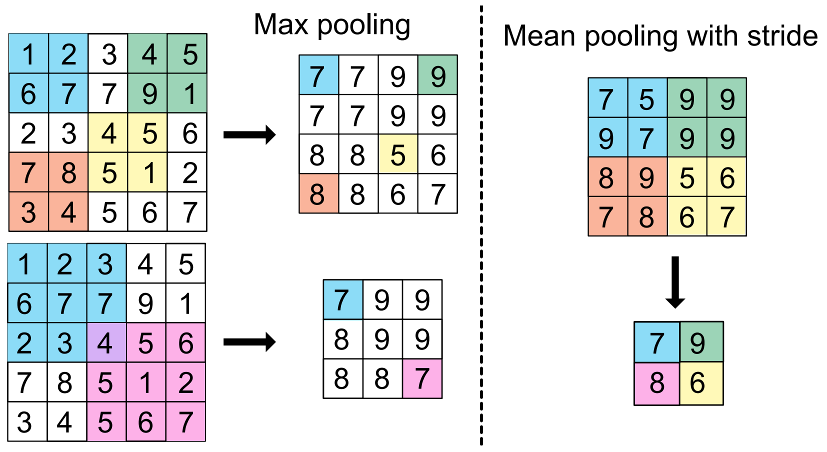


FIGURE 8.12: Scheme of pooling units.

With these tools in hand, it is possible to build new predictive tools. In Hoseinzade and Haratizadeh (2019), predictors such as price quotes, technical indicators and macro-economic data are fed to a complex neural network with 6 layers in order to predict the sign of price variations. While this is clearly an interesting computer science exercise, the deep economic motivation behind this choice of architecture remain unclear.

8.6.4 Advanced architectures

The superiority of neural networks in tasks related to computer vision and natural language processing is now well established. However, in many ML tournaments in the 2010 decade, neural networks have often been surpassed by tree-based models when dealing with tabular data. This puzzle encouraged researchers to construct novel NN structures that are better suited to tabular databases. Examples include Arik and Pfister (2019) and Popov et al. (2019) but their ideas lie outside the scope of this book. The interested reader can have a look at the original papers.

8.7 Coding exercises

9

Support vector machines

While the origins of support vector machines (SVM) are old (and go back to Vapnik and Lerner (1963)), their modern treatment was initiated in Boser et al. (1992) and Cortes and Vapnik (1995) (binary classification) and Drucker et al. (1997) (regression). We refer to `http://www.kernel-machines.org/books` for an exhaustive bibliography on their theoretical and empirical properties. SVMs have been very popular between since their creation among the machine learning community. Nonetheless, other tools (neural networks especially) have gained popularity and progressively replaced SVMs in many applications like computer vision notably.

9.1 SVM for classification

As is often the case in machine learning, it is easier to explain a complex tool through an illustration with binary classification. In fact, sometimes, it is originally how the tool was designed (e.g., for the perceptron). Let us consider a simple example in the plane , that is, with two features. In Figure 9.1, the goal is to find a model that correctly classifies points: filled circles versus empty squares.

A model consists of two weights $\mathbf{w} = (w_1, w_2)$ that load on the variables and create a natural linear separation in the plane. In the example above, we show three separations. The red one

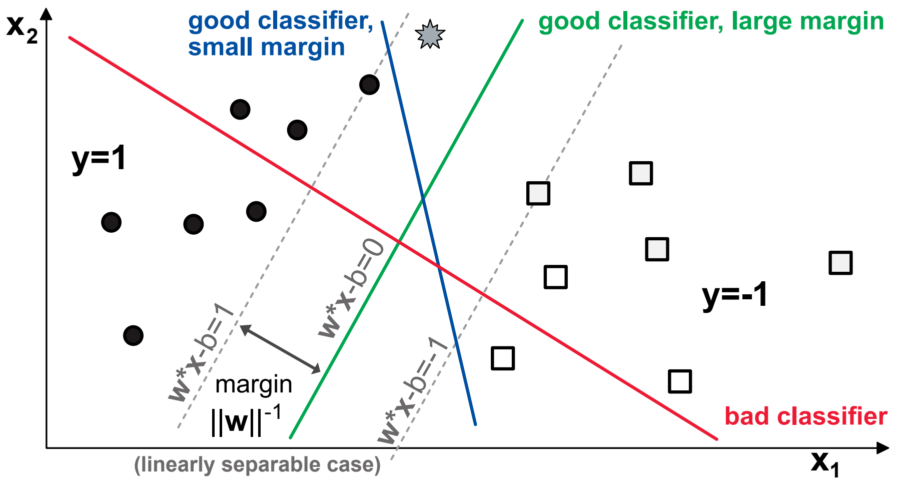

x2
good classifier, small margin
good classifier, large margin
y=1
y=-1
w*x-b=1
w*x-b=0
w*x-b=-1
margin
||w||^-1
bad classifier
(linearly separable case)
x1

FIGURE 9.1: Diagram of binary classification with support vectors

is not a good classifier because there are circles and squares above and beneath it. The blue line is a good classifier: all circles are to its left and all squares to its right. Likewise, the green line achieves a perfect classification score. Yet, there is a notable difference between the two.

The grey star at the top of the graph is a mystery point and given its location, if the data pattern holds, it should be a circle. The blue model fails to recognize it a such while the green one succeeds. The interesting features of the scheme are those that we have not mentioned yet, that is, the grey dotted lines. These lines represent the no-man's land in which no observation falls when the green model is enforced. In this area, each strip above and below the green line can be viewed as a margin of error for the model. Typically, the grey star is located inside this margin.

The two margins are computed as the parallel lines that maximize the distance between the model and the closest points that are correctly classified (on both sides). These points are called support vectors, which justifies the name of the technique. Obviously, the green model has a greater margin than the blue one. The core idea of SVMs is to maximize the margin, under the constraint that the classifier does not make any mistake. Said differently, SVMs try to pick the most robust model among all those that yield a correct classification.

More formally, if we numerically define circles as +1 and squares as -1, any 'good' linear model is expected to satisfy:

$$\begin{cases} \sum_{k=1}^K w_k x_{i,k} + b \geq +1 & \text{when } y_i = +1 \\ \sum_{k=1}^K w_k x_{i,k} + b \leq -1 & \text{when } y_i = -1, \end{cases} \tag{9.1}$$

which can be summarized in compact form $y_i \times \left(\sum_{k=1}^K w_k x_{i,k} + b\right) \geq 1$. Now, the margin between the green model and a support vector on the dashed grey line is equal to $||\mathbf{w}|| = \sqrt{\sum_{k=1}^K w_k^2}$. This value comes from the fact that the distance between a point (x_0, y_0) and a line parametrized by $ax + by + c = 0$ is equal to $d = \frac{|ax_0+by_0+c|}{\sqrt{a^2+b^2}}$. In hte case of the model defined above ((9.1)), the numerator is equal to 1 and the norm is that of $\mathbf{w}$. Thus, the final problem is the following:

$$\underset{\mathbf{w},b}{\text{argmin}} \ ||\mathbf{w}|| \ \text{ s.t. } y_i \left(\sum_{k=1}^K w_k x_{i,k} + b\right) \geq 1. \tag{9.2}$$

Naturally, this problem becomes infeasible whenever the condition cannot be satisfied, that is, when a simple line cannot perfectly separate the labels, no matter the choice of coefficients. This is the most common configuration and datasets are then called logically *not linearly separable*. This complicates the process but it is possible to resort to a trick. The idea is to introduce some flexbility in (9.1) by adding correction variables that allow the conditions to be met:

$$\begin{cases} \sum_{k=1}^K w_k x_{i,k} + b \geq +1 - \xi_i & \text{when } y_i = +1 \\ \sum_{k=1}^K w_k x_{i,k} + b \leq -1 + \xi_i & \text{when } y_i = -1, \end{cases} \tag{9.3}$$

where the novelties, the ξ_i are positive so-called 'slack' variables that make the conditions feasible. They are illustrated in Figure 9.2. In this new configuration, there is no simple linear model that can perfectly discriminate between the two classes.

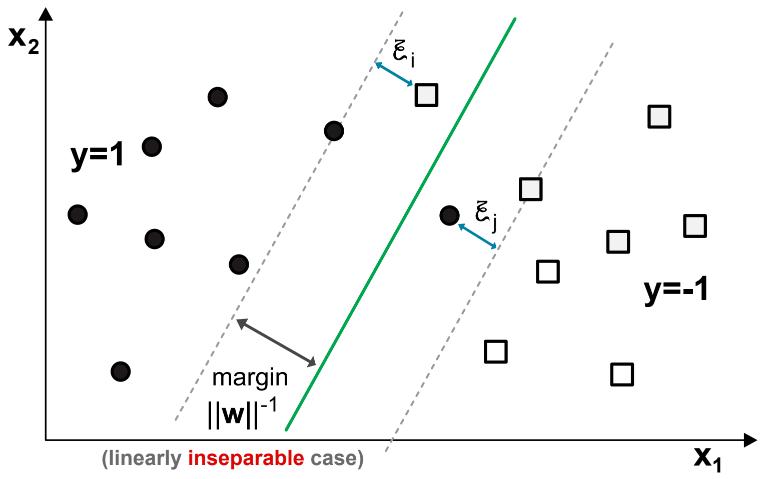


FIGURE 9.2: Diagram of binary classification with SVM: linearly inseparable data.

The optimization program then becomes

$$\underset{\mathbf{w},b,\boldsymbol{\xi}}{\text{argmin}}\ ||\mathbf{w}|| + C\sum_{i=1}^{I}\xi_i \ \text{ s.t. } \left\{ y_i\left(\sum_{k=1}^{K} w_k\phi(x_{i,k}) + b\right) \geq 1-\xi_i \ \text{ and } \ \xi_i \geq 0,\ \forall i \right\}, \tag{9.4}$$

where the parameter $C > 0$ tunes the cost of mis-classification: as C increases, errors become more penalizing. In addition, the program can be generalized to nonlinear models, via the kernel ϕ which is applied to the input points $x_{i,k}$. Nonlinear kernels can help cope with patterns that are more complex than straight lines (see Figure 9.3). Common kernels can be polynomial, radial or sigmoid. The solution is found using more or less standard techniques for constrained quadratic programs. Once the weights $\mathbf{w}$ and bias b a set via training, a prediction for a new vector $\mathbf{x}_j$ is simply made by computing $\sum_{k=1}^{K} w_k\phi(x_{j,k})+b$ and choosing the class based on the sign of the expression.

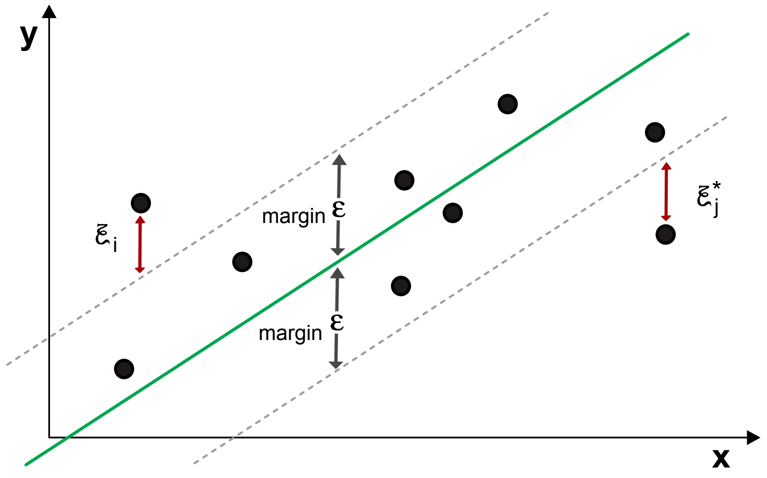


FIGURE 9.3: Examples of nonlinear kernels.

9.2 SVM for regression

The ideas of classification SVM can be transposed to regression exercises. One general formulation is the following

$$\underset{\mathbf{w},b,\boldsymbol{\xi}}{\text{argmin}} \; ||\mathbf{w}|| + C\sum_{i=1}^{I}(\xi_i + \xi_i^*) \tag{9.5}$$

$$\text{s.t.} \; \sum_{k=1}^{K} w_k\phi(x_{i,k}) + b - y_i \leq \epsilon + \xi_i \tag{9.6}$$

$$y_i - \sum_{k=1}^{K} w_k\phi(x_{i,k}) - b \leq \epsilon + \xi_i^* \tag{9.7}$$

$$\xi_i, \xi_i^* \geq 0, \; \forall i, \tag{9.8}$$

and it is illustrated in Figure 9.4. The user specifies a margin ϵ and the model will try to find the linear (up to kernal transformation) relationship between the labels y_i and the input $\mathbf{x}_i$. Just as in the classification task, if the data points are inside the strip, the slack variables ξ_i and ξ_i^* are set to zero. When the points violate the threshold, the objective function (first line of the code) is penalized. Note that setting a large ϵ leaves room for more error. Once the model has been trained, a prediction for $\mathbf{x}_j$ is simply $\sum_{k=1}^{K} w_k\phi(x_{j,k}) + b$.

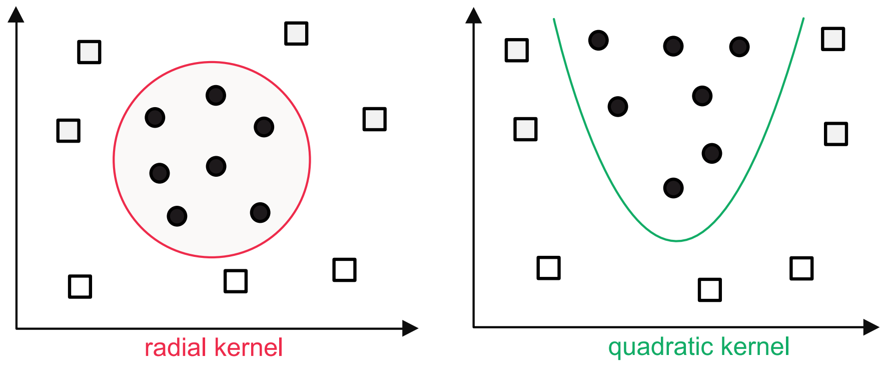


FIGURE 9.4: Examples of nonlinear kernels.

The models laid out in this section are a preview of the universe of SVM engines. One reference library that is coded in C and C++ is LIBSVM and it is widely used by many other programming languages. The interested reader can have a look at the corresponding article Chang and Lin (2011) for more details on the SVM zoo (a more recent November 2019 version is also available online).

9.3 Practice

In R the LIBSVM library is exploited in several packages. One of them, *e1071*, is a good choice because it also nests many other interesting functions, especially a naive Bayes classifier that we will use below.

In the implementation of LIBSVM, the package requires to specify the label and features separately. For this reason, we recycle the variables used for the boosted trees. Moreover, the training being slow, we perform it on a subsample of these sets (first thousand instances).

```r
library(e1071)
fit_svm <- svm(y = train_label_xgb[1:1000],      # Train label
               x = train_features_xgb[1:1000,],  # Training features
               type = "eps-regression",          # SVM task type (see LIBSVM documentation)
               kernel = "radial",                # SVM kernel (or: linear, polynomial, sigmoid)
               epsilon = 0.1,                    # Width of strip for errors
               gamma = 0.5,                      # Constant in the radial kernel
               cost = 0.1)                       # Slack variable penalisation
mean((predict(fit_svm,
              select(testing_sample,features_short)) - testing_sample$R1M_Usd)^2) # MSE
```

```
## [1] 0.03839085
```

```r
mean(predict(fit_svm,
             select(testing_sample,features_short)) * testing_sample$R1M_Usd > 0) # Hit ratio
```

```
## [1] 0.5222197
```

The results are slightly better than those of the boosted trees. All parameters are completely arbitrary, especially the choice of the kernel. We finally turn to a classification example.

```r
library(e1071)
fit_svm_C <- svm(y = training_sample$R1M_Usd_C[1:1000],   # Train label
               x = training_sample[1:1000,] %>%
                   select(features),                      # Training features
               type = "C-classification",                 # SVM task type (see LIBSVM documentation)
               kernel = "sigmoid",                        # SVM kernel
               gamma = 0.5,                               # Parameter in the sigmoid kernel
               coef0 = 0.3,                               # Parameter in the sigmoid kernel
               cost = 0.2)                                # Slack variable penalisation
mean(predict(fit_svm_C, select(testing_sample,features)) == testing_sample$R1M_Usd_C) # Accuracy
```

```
## [1] 0.5008973
```

The arbitrariness in our choice of the parameters may explain why the predictive accuracy is so poor.

9.4 Coding exercises

10

Bayesian methods

This section is dedicated to the subset of machine learning that makes prior assumptions on parameters. Before we explain how Bayes' theorem can be applied to simple building blocks in machine learning, we introduce some notations and concept in the subsection below. Good references for Bayesian analysis are Gelman et al. (2013) and Kruschke (2014). The latter, like the present book, illustrates the concepts with many lines of R code.

10.1 The Bayesian framework

Up to now, the models that have been presented rely on data only. This approach is often referred to as '*frequentist*'. Given one dataset, a frequentist will extract (i.e., estimate) a unique set of optimal parameters and consider it to be the best model. Bayesians, on the other hand, consider datasets as a snapshots of reality and for them, parameters are thus random! Instead of estimating one value for parameters, they a more ambitious and try to determine the whole distribution of the parameter.

In order to outline how that can be achieved, we introduce basic notations and results. The foundational concept in Bayesian analysis is the conditional probability. Given two random sets (or events) A and B, we define the probability of A knowing B (or, conditionally on B) as

$$P[A|B] = \frac{P[A \cap B]}{P[B]},$$

that is, the probability of the intersection between the two sets divided by the probability of B. Likewise, the probability that both events occur is equal to $P[A \cap B] = P[A]P[B|A]$. Given n disjoint events A_i, $i = 1, ...n$ such that $\sum_{i=1}^n P(A_i) = 1$, then for any event B, the law of total probabilities is (or implies)

$$P(B) = \sum_{i=1}^n P(B \cap A_i) = \sum_{i=1}^n P(B|A_i)P(A_i).$$

Given this expression, we can formulate a general version of Bayes' theorem:

$$P(A_i|B) = \frac{P(A_i)P(B|A_i)}{P(B)} = \frac{P(A_i)P(B|A_i)}{\sum_{i=1}^n P(B|A_i)P(A_i)}. \tag{10.1}$$

Endowed with this result, we can move forward to the core topic of this section, which is the estimation of some parameter $\boldsymbol{\theta}$ (possibly a vector) given a dataset, which we denote with $\mathbf{y}$) (following the conventions from Gelman et al. (2013)).

In Bayesian analysis, one sophistication (compared to a frequentist approach) comes from the fact that the data is not almighty. The distribution of the parameter $\boldsymbol{\theta}$ will be a mix between some prior distribution set by the statistician (the user, the analyst) and the empirical distribution from the data. More precisely, a simple application of Bayes' formula yields

$$p(\boldsymbol{\theta}|\mathbf{y}) = \frac{p(\boldsymbol{\theta})p(\mathbf{y}|\boldsymbol{\theta})}{p(\mathbf{y})} \propto p(\boldsymbol{\theta})p(\mathbf{y}|\boldsymbol{\theta}). \tag{10.2}$$

The interpretation is immediate: the distribution of $\boldsymbol{\theta}$ knowing the data $\mathbf{y}$ is proportional to the distribution of $\boldsymbol{\theta}$ times the distribution of $\mathbf{y}$ knowing $\boldsymbol{\theta}$. The term $p(\mathbf{y})$ is often omitted because it is simply a scaling number that ensures that the density sums or integrates to one.

We use a slightly different notation between Equation (10.1) and Equation (10.2). In the former, P denotes a true probability, i.e., it is a number. In the latter, p stands for the whole probability density function of $\boldsymbol{\theta}$ or $\mathbf{y}$.

The whole purpose of Bayesian analysis is to compute the so-called *posterior* distribution $p(\boldsymbol{\theta}|\mathbf{y})$ via the *prior* distribution $p(\boldsymbol{\theta})$ and the *likelihood function* $p(\mathbf{y}|\boldsymbol{\theta})$. Priors are sometimes qualified as informative, weakly informative or uninformative, depending on the degree to which the user is confident on the relevance and robustness of the prior. The simplest way to define a non-informative prior is to set a constant (uniform) distribution over some realistic intervals.

The most challenging part is usually the likelihood function. The easiest way to solve the problem is to resort to a specific distribution (possibly a parametric family) for the distribution of the data and then consider that obsevations are i.i.d., just as in a simple maximum likelihood inference. If we assume that new parameters for the distributions are gathered into $\boldsymbol{\lambda}$, then the likelihood can be written as

$$p(\mathbf{y}|\boldsymbol{\theta},\boldsymbol{\lambda}) = \prod_{i=1}^{I} f_{\boldsymbol{\lambda}}(y_i;\boldsymbol{\beta}), \tag{10.3}$$

but in this case the problem because slightly more complex because adding new parameters change the posterior distribution to $p(\boldsymbol{\theta},\boldsymbol{\lambda}|\mathbf{y})$. The user must find out the joint distribution of $\boldsymbol{\theta}$ and $\boldsymbol{\lambda}$ - given $\mathbf{y}$).

10.2 Bayesian sampling

10.2.1 Gibbs sampling

One adjacent field of applications of Bayes' theorem is simulation. Suppose we want to simulate the multivariate distribution of a random vector $\mathbf{X}$ given by its density $p = p(x_1,\dots,x_J)$. Often, the full distribution is complex, but its marginals are more accessible. Indeed, they are simpler because they depend on only one variable (when all other values are known):

$$p(X_j = x_j|X_1 = x_1,\dots,X_{j-1} = x_{j-1},X_{j+1} = x_{j+1},\dots,X_J = x_J) = p(X_j = x_j|\mathbf{X}_{-j} = \mathbf{x}_{-j}),$$

where we use the compact notation $\mathbf{X}_{-j}$ for all variables except X_j. One way to generate samples with law p is the following and relies both on the knowledge of the conditionals

$p(x_j|\mathbf{x}_{-j})$ and on the notion of Markov Chain Monte Carlo. The process is iterative and assumes that it is possible to draw samples of the aforementioned conditionals. We write x_j^m for the m^{th} sample of the j^{th} variable (X_j). The simulation starts with a prior (or fixed, or random) sample $\mathbf{x}^0 = (x_1^0, \dots, x_J^0)$. Then, for a sufficiently large number of times, say T, new samples are drawn according to

$$\begin{aligned}
x_1^{m+1} &= p(X_1|X_2 = x_2^m, \dots, X_J = x_J^m);\\
x_2^{m+1} &= p(X_2|X_1 = x_1^{m+1}, X_3 = x_3^m, \dots, X_J = x_J^m);\\
&\dots\\
x_J^{m+1} &= p(X_J|X_1 = x_1^{m+1}, X_2 = x_2^{m+1}, \dots, X_{J-1} = x_{J-1}^{m+1}).
\end{aligned}$$

The important detail is that after each line, the value of the variable is updated. Hence, in the second line, X_2 is sampled with the knowledge of $X_1 = x_1^{m+1}$ and in the last line, all variable except X_J have been updated to their $m+1^{th}$ state. The above algorithm is called Gibbs sampling. It relates to Markov chains because each new iteration depends only on the previous one.

Under some technical assumptions, as T increases, the distrbution of $\mathbf{x}_T$ converges to that of p. The conditions under which the convergence occurs have been widely discussed in series of articles in the 1990s. The interested reader can have a look for instance at Tierney (1994), Roberts and Smith (1994), as well as at section 11.7 of Gelman et al. (2013).

Sometimes, the full distribution is complex and the conditional laws are hard to determine and to sample. Then, a more general method, called Metropolis-Hastings, can be used that relies on the rejection method for the simulation of random variables. For the sake of brevity, we omit the presentation here, but the methods are outlined in section 11.2 of Gelman et al. (2013) and in Chapter 7 of Kruschke (2014).

10.2.2 Metropolis-Hastings sampling

The Gibbs algorithm can be considered as a particular case of the Metropolis-Hastings (MH) method, which, is its simplest version, was introduced in Metropolis and Ulam (1949). The premise is similar: the aim is to simulate random variables that follow $p(\mathbf{x})$ with the ability to sample from of a simpler form $p(\mathbf{x}|\mathbf{y})$ which gives the probability of the future state $\mathbf{x}$, given the past one $\mathbf{y}$.

Once an initial value for $\mathbf{x}$ has been sampled ($\mathbf{x}_0$), each new iteration (m) of the simulation takes place in three stages:

1. generate a candidate value $\mathbf{x}'_{m+1}$ from $p(\mathbf{x}|\mathbf{x}_m)$;
2. compute the acceptance ratio $\alpha = \min\left(\frac{p(\mathbf{x}'_{m+1})p(\mathbf{x}_m|\mathbf{x}'_{m+1})}{p(\mathbf{x}_m)p(\mathbf{x}'_{m+1}|\mathbf{x}_m)}\right)$
3. pick $\mathbf{x}_{m+1} = \mathbf{x}'_{m+1}$ with probability α or stick with the previous value ($\mathbf{x}_{m+1} = \mathbf{x}_m$) with probability $1 - \alpha$

The interpretation of the acceptance ratio is not straightforward in the general case. When the sampling generator is symmetric ($p(\mathbf{x}|\mathbf{y}) = (\mathbf{y}|\mathbf{x})$), the candidate is always chosen whenever $p(\mathbf{x}'_{m+1}) \geq p(\mathbf{x}_m)$. If the reverse condition holds, then it is retained with odds equal to $p(\mathbf{x}'_{m+1})/p(\mathbf{x}_m)$, which is the ratio of likelihoods. The more likely the new proposal, the higher the odds of retaining it.

Often, the first simulations are discarded in order to leave time to the chain to converge to a high probability region. This procedure (often called 'burn in') ensures that the first retained samples are located in a zone that is likely, that is: they are more representative of the law we are trying to simulate.

10.3 Bayesian linear regression

Because Bayesian concepts are rather abstract, it is useful to illustrate the theoretical notions with a simple example. In a linear model, $y_i = \mathbf{x}_i\mathbf{b} + \epsilon_i$ and it is often statistically assumed that the ϵ_i are i.i.d. and normally distributed with zero mean and variance σ^2. Hence, Equation (10.3) translates into

$$p(\boldsymbol{\epsilon}|\mathbf{b},\sigma) = \prod_{i=1}^{I} \frac{e^{-\frac{\epsilon_i^2}{2\sigma}}}{\sigma\sqrt{2\pi}} = (\sigma\sqrt{2\pi})^{-I}e^{-\sum_{i=1}^{I}\frac{\epsilon_i^2}{2\sigma^2}}.$$

In a regression analysis, the data is given both by $\mathbf{y}$ and by $\mathbf{X}$, hence both are reported in the notations. Simply acknowledging that $\boldsymbol{\epsilon} = \mathbf{y} - \mathbf{X}\mathbf{b}$, we get

$$\begin{aligned} p(\mathbf{y},\mathbf{X}|\mathbf{b},\sigma) &= \prod_{i=1}^{I} \frac{e^{-\frac{\epsilon_i^2}{2\sigma}}}{\sigma\sqrt{2\pi}} = (\sigma\sqrt{2\pi})^{-I}e^{-\sum_{i=1}^{I}\frac{\left(y_i-\mathbf{x}_i'\mathbf{b}\right)^2}{2\sigma^2}} = (\sigma\sqrt{2\pi})^{-I}e^{-\frac{(\mathbf{y}-\mathbf{X}\mathbf{b})'(\mathbf{y}-\mathbf{X}\mathbf{b})}{2\sigma^2}} \\ &= \underbrace{(\sigma\sqrt{2\pi})^{-I}e^{-\frac{(\mathbf{y}-\mathbf{X}\hat{\mathbf{b}})'(\mathbf{y}-\mathbf{X}\hat{\mathbf{b}})}{2\sigma^2}}}_{\text{depends on }\sigma,\text{ not }\mathbf{b}} \times \underbrace{e^{-\frac{(\mathbf{b}-\hat{\mathbf{b}})'\mathbf{X}'\mathbf{X}(\mathbf{b}-\hat{\mathbf{b}})}{2\sigma^2}}}_{\text{depends on both }\sigma,\text{ and }\mathbf{b}}, \end{aligned} \tag{10.4}$$

where the last line is very convenient because it expresses the law as a function of the difference $\mathbf{b} - \hat{\mathbf{b}}$, where $\hat{\mathbf{b}} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$ seems like a natural choice for the mean of $\mathbf{b}$ - though in the end this vector is chosen for the simple form it produces.

The above expression is the frequentist (data-based) block of the posterior. If we want to obtain a tractable expression for the posterior, we need to find a prior that has a form that will combine well with the likelihood. These forms are called *conjugate priors*. A natural candidate for the right part (that depends on both $\mathbf{b}$ and σ) is the multivariate Gaussian density:

$$p[\mathbf{b}|\sigma] = \sigma^{-k}e^{-\frac{(\mathbf{b}-\mathbf{b}_0)'\boldsymbol{\Lambda}_0(\mathbf{b}-\mathbf{b}_0)}{2\sigma^2}}, \tag{10.5}$$

where we are obliged to condition with respect to σ. The density has prior mean $\mathbf{b}_0$ and prior covariance matrix $\boldsymbol{\Lambda}_0^{-1}$. This prior gets us one step closer to the posterior because

$$\begin{aligned} p[\mathbf{b},\sigma|\mathbf{y},\mathbf{X}] &\propto p[\mathbf{y},\mathbf{X}|\mathbf{b},\sigma]p[\mathbf{b},\sigma] \\ &\propto p[\mathbf{y},\mathbf{X}|\mathbf{b},\sigma]p[\mathbf{b}|\sigma]p[\sigma]. \end{aligned} \tag{10.6}$$

In order to fully specify the cascade of probability, we need to take care of σ and set a density of the form

$$p[\sigma^2] \propto (\sigma^2)^{-1-a_0}e^{-\frac{b_0}{2\sigma^2}}, \tag{10.7}$$

which is close to that of the left part of (10.4). This corresponds to an inverse gamma distribution for the variance with prior parameters a_0 and b_0 (this scalar notation is not optimal because it can be confused with the prior mean $\mathbf{b}_0$ so we must pay extra attention).

Now, we can simplify $p[\mathbf{b},\sigma|\mathbf{y},\mathbf{X}]$ with (10.4), (10.5) and (10.7):

$$\begin{aligned} p[\mathbf{b},\sigma|\mathbf{y},\mathbf{X}] &\propto (\sigma\sqrt{2\pi})^{-I}\sigma^{-2(1+a_0)}e^{-\frac{(\mathbf{y}-\mathbf{X}\hat{\mathbf{b}})'(\mathbf{y}-\mathbf{X}\hat{\mathbf{b}})}{2\sigma^2}} \times e^{-\frac{(\mathbf{b}-\hat{\mathbf{b}})'\mathbf{X}'\mathbf{X}(\mathbf{b}-\hat{\mathbf{b}})}{2\sigma^2}}\sigma^{-k}e^{-\frac{(\mathbf{b}-\mathbf{b}_0)'\boldsymbol{\Lambda}_0(\mathbf{b}-\mathbf{b}_0)}{2\sigma^2}}e^{-\frac{b_0}{2\sigma^2}} \\ &\propto \sigma^{-I-k-2(1+a_0)}\exp\left(-\frac{\left(\mathbf{y}-\mathbf{X}\hat{\mathbf{b}}\right)'\left(\mathbf{y}-\mathbf{X}\hat{\mathbf{b}}\right)+(\mathbf{b}-\hat{\mathbf{b}})'\mathbf{X}'\mathbf{X}(\mathbf{b}-\hat{\mathbf{b}})+(\mathbf{b}-\mathbf{b}_0)'\boldsymbol{\Lambda}_0(\mathbf{b}-\mathbf{b}_0)+b_0}{2\sigma^2}\right) \end{aligned}$$

The above expression is simply a quadratic form in $\mathbf{b}$ and it can be rewritten after burdensome algebra in a much more compact manner:

$$p(\mathbf{b}|\mathbf{y},\mathbf{X},\sigma) \propto \left[\sigma^{-k}e^{-\frac{(\mathbf{b}-\mathbf{b}_*)'\boldsymbol{\Lambda}_*(\mathbf{b}-\mathbf{b}_*)}{2\sigma^2}}\right] \times \left[(\sigma^2)^{-1-a_*}e^{-\frac{b_*}{2\sigma^2}}\right], \qquad (10.8)$$

where

$$\begin{aligned} \boldsymbol{\Lambda}_* &= \mathbf{X}'\mathbf{X}+\boldsymbol{\Lambda}_0 \\ \mathbf{b}_* &= \boldsymbol{\Lambda}_*^{-1}(\boldsymbol{\Lambda}_0\mathbf{b}_0+\mathbf{X}'\mathbf{X}\hat{\mathbf{b}}) \\ a_* &= a_0+I/2 \\ b_* &= b_0+\frac{1}{2}\left(\mathbf{y}'\mathbf{y}+\mathbf{b}_0'\boldsymbol{\Lambda}_0\mathbf{b}_0+\mathbf{b}_*'\boldsymbol{\Lambda}_*\mathbf{b}_*\right) \end{aligned}$$

This expression has two parts: the Gaussian component which relates mostly to $\mathbf{b}$, and the inverse gamma component, entirely dedicated to σ. The mix between the prior and the data is clear. The posterior covariance matrix of the Gaussian part ($\boldsymbol{\Lambda}_*$) is the sum between the prior and a quadratic form from the data. The posterior mean $\mathbf{b}_*$ is a weighted average of the prior $\mathbf{b}_0$ and the sample estimator $\hat{\mathbf{b}}$. Such blends of quantities estimated from data and a user-supplied version are often called *shrinkages*. The original covariance matrix $\mathbf{X}'\mathbf{X}$ is shrunk towards the prior $\boldsymbol{\Lambda}_0$. This can be viewed as a regularization procedure: the pure fit originating from the data is mixed with some 'external' ingredient to give some structure to the final extimation.

The interested reader can also have a look at section 16.3 of Greene (2018) (the case of conjugate priors is treated in section 16.3.2).

The formulae above can be long and risky to implement. Luckily, there is a package that performs Bayesian inference for linear regression using the conjugate priors. Below, we provide one example of how it works. To simplify the code and curtail computation times, we consider two predictors. In statistics, the precision matrix is the inverse of the covariance matrix. In the parameters, the first two priors relate to the Gaussian law and the last two to the inverse gamma distribution:

$$f_{\text{invgamma}}(x,\alpha,\beta) = \frac{\beta^\alpha}{\Gamma(\alpha)}x^{-1-\alpha}e^{-\frac{\beta}{x}},$$

where α is the shape and β is the scale.

```
prior_mean <- c(0.01,0.1,0.1)                     # Average value of parameters (prior)
precision_mat <- diag(prior_mean^2) %>% solve()  # Inverse covariance matrix of parameters (prior)
fit_lmBayes <- bayesLMConjugate(
    R1M_Usd ~ Mkt_Cap_3M_Usd + Pb,           # Model: size and value
    data = testing_sample,                   # Data source, here, the test sample
    n.samples = 2000,                        # Number of samples used
    beta.prior.mean = prior_mean,            # Avg prior: size & value rewarded & unit beta
    beta.prior.precision = precision_mat,    # Precision matrix
    prior.shape = 0.5,                       # Shape for prior distribution of sigma
    prior.rate = 0.5)                        # Scale for prior distribution fo sigma
```

In the above specification, we must also provide a prior for the constant. By default, we set its average value to 0.01, which corresponds to a 1% average monthly return.

```
fit_lmBayes$p.beta.tauSq.samples[,1:3] %>% as_tibble() %>%
    `colnames<-`(c("Intercept", "Size", "Value")) %>%
    gather(key = coefficient, value = value) %>%
    ggplot(aes(x = value, fill = coefficient)) + geom_histogram(alpha = 0.5)
```

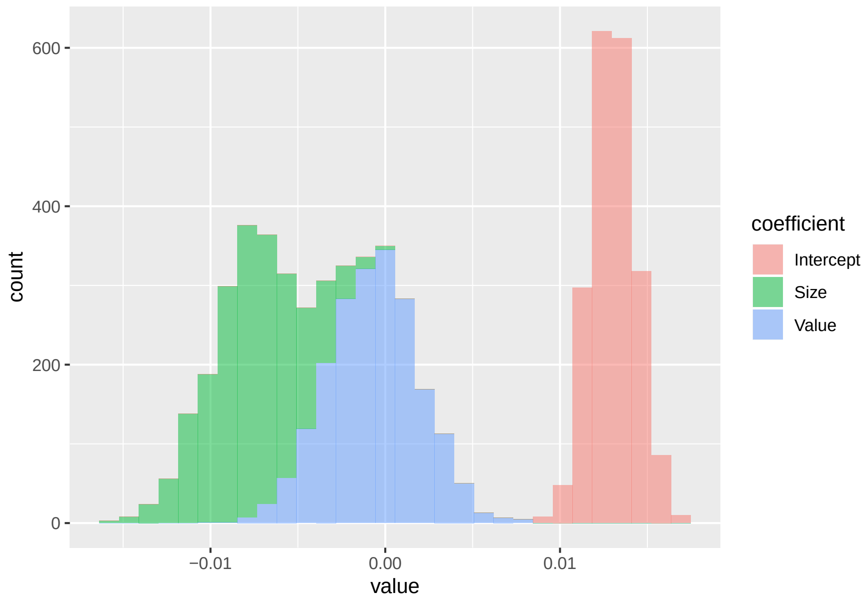


FIGURE 10.1: Distribution of linear regression coefficients.

The distribution of the constant is firmly to the right, hence soldily positive. For the size coefficient it is the opposite: it is negative (small firms are more profitable). With regard to value, it is hard to conclude, the distribution is balanced around zero.

10.4 Naive Bayes classifier

Bayes' theorem can also be easily applied to classification. We formulate it with respect to the label and features and write

$$P[\mathbf{y}|\mathbf{X}] = \frac{P[\mathbf{X}|\mathbf{y}]P[\mathbf{y}]}{P[\mathbf{X}]} \propto P[\mathbf{X}|\mathbf{y}]P[\mathbf{y}], \tag{10.9}$$

and then split the input matrix into its column vectors $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_K)$. This yields

$$P[\mathbf{y}|\mathbf{x}_1, \dots, \mathbf{x}_K] \propto P[\mathbf{x}_1, \dots, \mathbf{x}_K|\mathbf{y}]P\mathbf{y}]. \tag{10.10}$$

The 'naive' qualification of the method comes from a simplifying assumption on the features.[1] If they are all mutually independent, then the likelihood in the above expression can be expanded into

$$P[\mathbf{y}|\mathbf{x}_1, \dots, \mathbf{x}_K] \propto P[\mathbf{y}] \prod_{k=1}^{K} P[\mathbf{x}_k|\mathbf{y}]. \tag{10.11}$$

The next step is to be more specific about the likelihood. This can be done non-parametrically (via kernel estimation) or with common distributions (Gaussian for continuous data, Bernoulli for binary data). In factor investing, the features are continuous, thus the Gaussian law is more adequate:

$$P[x_{i,k} = z|\mathbf{y}_i = c] = \frac{e^{-\frac{(z-m_c)^2}{2\sigma_c^2}}}{\sigma_c\sqrt{2\pi}},$$

where c is the value of the classes taken by y and σ_c and m_c are the standard error and mean of $x_{i,k}$, conditional on y_i being equal to c. In practice, each class is spanned, the training set is filtered accordingly and σ_c and m_c are taken to be the sample statistics. This Gaussian parametrization is probably ill-suited to our dataset because the features are uniformly distributed. Even after conditioning, it is unlikely that the distribution will be even remotely close to Gaussian. Technically, this can be overcome via a double transformation method. Given a vector of features $\mathbf{x}_k$ with empirical cdf $F_{\mathbf{x}_k}$, the variable

$$\tilde{\mathbf{x}}_k = \Phi^{-1}\left(F_{\mathbf{x}_k}(\mathbf{x}_k)\right), \tag{10.12}$$

will have a standard normal law whenever $F_{\mathbf{x}_k}$ is not pathological (lies in (0,1) for instance). If all features are independent, the transformation should not have any impact on the correlation structure. Otherwise, we refer to the literature on the NORmal-To-Anything (NORTA) method (see, e.g., Chen (2001), Coqueret (2017)).

Lastly, the prior $P[\mathbf{y}]$ in Equation (10.11) is often either taken to be uniform across the classes ($1/K$ for all k) or equal to the sample distribution.

We illustrate the naive Bayes classification tool with a simple example. While the package *e1071* embeds such a classifier, the *naivebayes* library offers more options (Gaussian, Bernoulli, multinomial and nonparametric likelihoods). Below, since the features are uniformly distributed, thus the transformation in (10.12) amounts to apply the inverse Gaussian cdf.

[1]This assumption can be relaxed, but the algorithms then become more complex and are out of the scope of the current book. One such example that generalizes the naive Bayes approach is Friedman et al. (1997).

```
library(naivebayes)
gauss_features_train <- training_sample %>%
    select(features_short) %>%
    as.matrix() %>%
    `*`(0.999) %>%
    + (0.0001) %>%
    qnorm() %>%
    `colnames<-`(features_short)
fit_NB_gauss <- naive_bayes(x = gauss_features_train,      # Transformed features
                            y = training_sample$R1M_Usd_C) # Label
layout(matrix(c(1,1,2,3,4,5,6,7), 4, 2, byrow = TRUE), widths=c(0.9,0.45))
par(mar=c(1, 1, 1, 1))
plot(fit_NB_gauss, prob = "conditional")
```

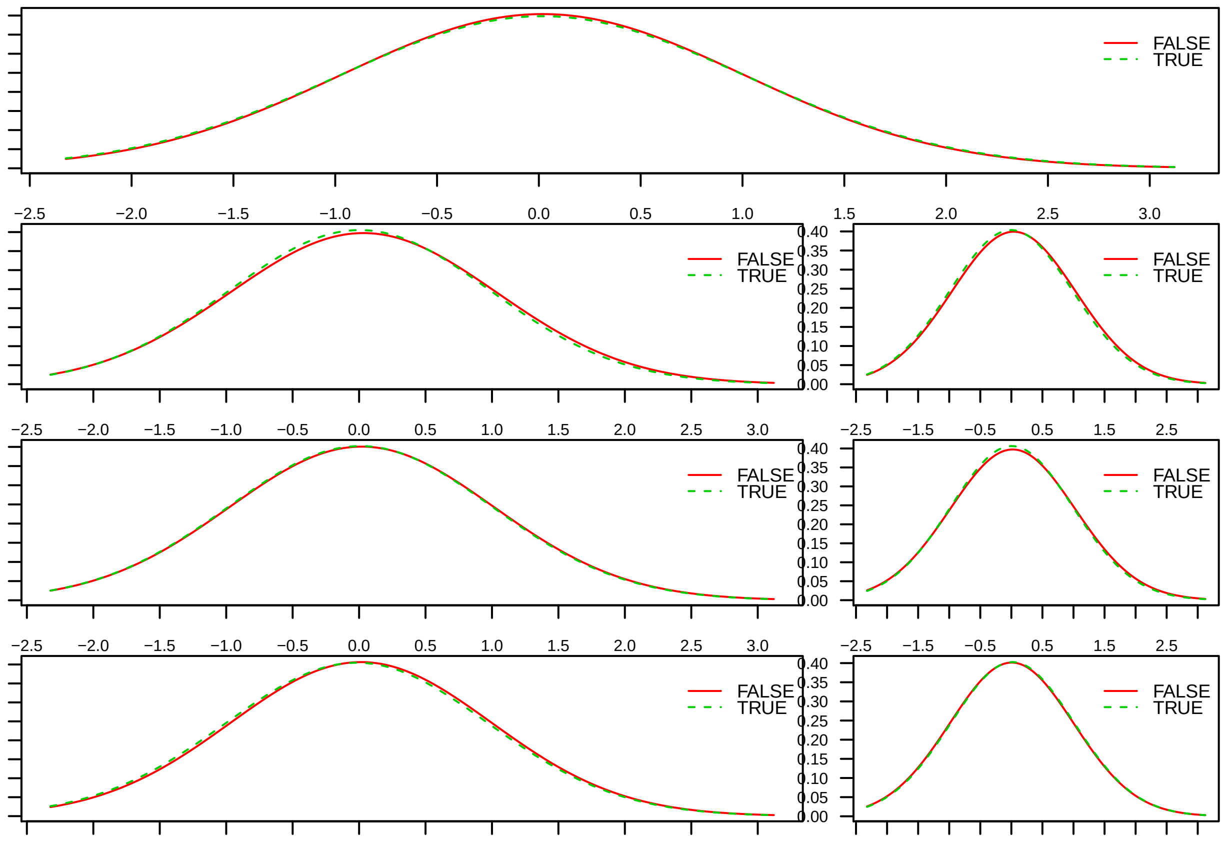


FIGURE 10.2: Distributions of predictor variables, conditional on the class of the label. TRUE is when the instance corresponds to an above median return and FALSE to a below median return.

The plots show the distributions of the features, conditionally on each value of the label. Essentially, those are the densities $P[\mathbf{x}_k|\mathbf{y}]$. For each feature, both distributions are very similar.

As usual, when the model has been trained, the accuracy of predictions can be evaluated.

```
gauss_features_test <- testing_sample %>%
    select(features_short) %>%
    as.matrix() %>%
    `*`(0.999) %>%
    + (0.0001) %>%
    qnorm() %>%
    `colnames<-`(features_short)
mean(predict(fit_NB_gauss, gauss_features_test) == testing_sample$R1M_Usd_C) # Hit ratio
```

```
## [1] 0.4956985
```

The performance of the classifier is not satisfactory as it underperforms a random guess.

10.5 Bayesian additive trees

10.5.1 General formulation

Bayesian additive regression trees (BARTs) are an ensemble technique that mixes Bayesian thinking and regression trees. In spirit, they are close to boosted trees, but they differ greatly in their implementation. In BARTs like in Bayesian regressions, the regularization comes from the prior. The original article is Chipman et al. (2010) and the implementation (in R) follows Sparapani et al. (2019).

Formally, the model is an aggregation of M models, which we write as

$$y = \sum_{m=1}^{M} \mathcal{T}_m(q_m, \mathbf{w}_m, \mathbf{x}) + \epsilon, \tag{10.13}$$

where ϵ is a Gaussian noise with variance σ^2, and the $\mathcal{T}_m = \mathcal{T}_m(q_m, \mathbf{w}_m, \mathbf{x})$ are decision trees with structure q_m and weights vectors $\mathbf{w}_m$. This decomposition of the tree is the one we used for boosted trees and is illustrated in Figure 7.5. q_m codes all splits (variables chosen for the splits and levels of the splits) and the vectors $\mathbf{w}_m$ correspond to the leaf values (at the terminal nodes).

At the macro level, BARTs can be viewed as traditional Bayesian objects, where the parameters $\boldsymbol{\theta}$ are all of the unknowns coded through q_m, $\mathbf{w}_m$ and σ^2 and where the focus is set on determining the posterior

$$\left(q_m, \mathbf{w}_m, \sigma^2\right) |(\mathbf{X}, \mathbf{Y}). \tag{10.14}$$

Given particular forms of priors for $\left(q_m, \mathbf{w}_m, \sigma^2\right)$, the algorithm draws the parameters using a combination of Metropolis-Hastings *and* Gibbs samplers.

10.5.2 Priors

The definition of prior in tree models is delicate and intricate. The first important assumption is independence: independence between σ^2 and all other parameters and independence between trees, that is, between couples $(q_m, \mathbf{w}_m)$ and $(q_n, \mathbf{w}_n)$ for $m \neq n$. This entails

$$P((q_1,\mathbf{w}_1),\dots,(q_M,\mathbf{w}_M),\sigma^2)=P(\sigma^2)\prod_{m=1}^{M}P(q_m,\mathbf{w}_m).$$

Moreover, it is customary (for simplicity) to separate the structure of the tree (q_m) and the terminal weights ($\mathbf{w}_m$), so that

$$P((q_1,\mathbf{w}_1),\dots,(q_M,\mathbf{w}_M),\sigma^2)=\underbrace{P(\sigma^2)}_{\text{noise term}}\prod_{m=1}^{M}\underbrace{P(\mathbf{w}_m|q_m)}_{\text{tree weights}}\underbrace{P(q_m)}_{\text{tree struct.}} \tag{10.15}$$

It remains to formulate the assumptions for each of the three parts.

We start with the trees' structures, q_m. Trees are defined by their splits (at nodes) and these splits are characterized by the splitting variable and the splitting level. First, the size of trees is parametrized such that a node at depth d is nonterminal with probability given by

$$\alpha(1+d)^{-\beta},\quad \alpha\in(0,1),\quad \beta>0. \tag{10.16}$$

The authors recommend to set $\alpha=0.95$ and $\beta=2$. This gives a probability of 5% to have 1 node, 55% to have 2 nodes, 28% to have 3 nodes, 9% to have 4 nodes and 3% to have 5 nodes. Thus, the aim is to force shallow structures.

Second, the choice of splitting variables is driven by a generalized Bernoulli (categorical) distribution which defines the odds of picking one particular feature. In the original paper by Chipman et al. (2010), the vector of probabilities was uniform (each predictor has the same odds of being chosen for the split). This vector can also be random and sampled from a Dirichlet distribution. The level of the split is drawn uniformly on the set of possible values for the chosen predictor.

Having determined the prior of structure of the tree q_m, it remains to fix the terminal values at the leaves ($\mathbf{w}_m|q_m$). The weights at all leaves are assumed to follow a Gaussian distribution $\mathcal{N}(\mu_\mu,\sigma_\mu^2)$, where $\mu_\mu=(y_{\min}+y_{\max})/2$ is the center of the range of the label values. The variance σ_μ^2 is chosen such that μ_μ plus or minus two times σ_μ^2 covers 95% of the range observed in the training dataset. Those are default values and can be altered by the user.

Lastly, for computational purposes similar to those of linear regressions, the parameter σ^2 (the variance of ϵ in (10.13)) is assumed to follow and inverse Gamma law $\text{IG}(\nu/2,\lambda\nu/2)$. The parameters are by default computed from the data so that the distribution of σ^2 is realistic and prevents overfitting. We refer to the original article, section 2.2.4, for more details on this topic.

In sum, in addition to M (number of trees), the prior depends on a small number of parameters: α and β (for the tree structure), μ_μ and σ_μ^2 (for the tree weights) and ν and λ (for the noise term).

10.5.3 Sampling and predictions

The posterior distribution in (10.14) cannot be obtained analytically but simulations are an efficient shortcut to the model (10.13). Just as in Gibbs and Metropolis-Hastings sampling, the distribution of simulations is expected to converge to the sought posterior. After some burn-in sample, a prediction for a newly observed set $\mathbf{x}_*$ will simply be the average (or median) of the predictions from the simulations. If we assume S simulations after burn-in, then the average is equal to

$$\tilde{y}(\mathbf{x}_*) := \frac{1}{S}\sum_{s=1}^{S}\sum_{m=1}^{M}\mathcal{T}_m\left(q_m^{(s)}, \mathbf{w}_m^{(s)}, \mathbf{x}_*\right).$$

The complex part is naturally to generate the simulations. Each tree is sampled using the Metropolis-Hastings method: a tree is proposed, but it replaces the existing one only under some (possibly random) criterion. This procedure is then repeated in a Gibbs-like fashion.

Let us start with the WH building block. We seek to simulate the conditional distribution

$$(q_m, \mathbf{w}_m) \mid (q_{-m}, \mathbf{w}_{-m}, \sigma^2, \mathbf{y}, \mathbf{x}),$$

where q_{-m} and $\mathbf{w}_{-m}$ collect the structures and weights of all trees except for tree number m. One tour de force in BART is to simplify the above Gibbs draws to

$$(q_m, \mathbf{w}_m) \mid (\mathbf{R}_m, \sigma^2),$$

where $\mathbf{R}_m = \mathbf{y} - \sum_{l\neq m}\mathcal{T}_l(q_l, \mathbf{w}_l, \mathbf{x})$ is the partial residual on a prediction that excludes the m^{th} tree.

The new MH proposition for q_m is based on the previous tree and there are four possible (and random) alterations to the tree:
- growing a terminal node (increase the complexity of the tree by adding a supplementary leaf);
- pruning a pair of terminal nodes (the opposite operation);
- changing splitting rules.

For simplicity, the third option is often excluded. Once the tree structure is defined (i.e., sampled), the terminal weights are independently drawn according to a Gaussian distribution $\mathcal{N}(\mu_\mu, \sigma_\mu^2)$.

After the tree is sampled, the WH principle requires that it be accepted or rejected based on some probability. This probability increases with the odds that the new tree increases the likelihood of the model. Its detailed computation is cumbersome and we refer to Section 2.2 in Sparapani et al. (2019) for details on the matter.

Now, we must outline the overarching Gibbs procedure. First, the algorithm starts with trees that are simple nodes. Then, a speficied number of loops include the following *sequential* steps:

1. sample $(q_1, \mathbf{w}_1) \mid (\mathbf{R}_1, \sigma^2)$;

2. sample $(q_2, \mathbf{w}_2) \mid (\mathbf{R}_2, \sigma^2)$;
   $\dots$;
   M. sample $(q_M, \mathbf{w}_M) \mid (\mathbf{R}_M, \sigma^2)$;
   M+1. sample σ^2 given the full residual $\mathbf{R} = \mathbf{y} - \sum_{l=1}^{M}\mathcal{T}_l(q_l, \mathbf{w}_l, \mathbf{x})$

At each step m, the residual $\mathbf{R}_m$ is updated with the values from step $m-1$. We illustrate this process in Figure 10.3 in which $M = 3$. At step 1, a partition is proposed for the first tree, which is a simple node. In this particular case, the tree is accepted. In this scheme, the terminal weights are omitted for simplicity. At step two, another partition is proposed for the tree, but it is rejected. In the third step, the proposition for the third is accepted. After the third step, a new value for σ^2 is drawn and a new round of Gibbs sampling can commence.

10.5.4 Code

There are several R packages that implement BART methods: *BART*, *bartMachine* and an older one (the original), *BayesTree*. The first one is highly efficient, hence we work with it. We resort to only a few parameters, like the power and base, which are the β and α defined in (10.16). The program is a bit verbose and delivers a few parametric details.

```r
fit_bart <- gbart(
    x.train = select(training_sample, features_short) %>% data.frame(), # Training features
    y.train = select(training_sample, R1M_Usd) %>% as.matrix() ,        # Training label
    x.test = select(testing_sample, features_short)  %>% data.frame(),  # Testing features
    type = "wbart",                                                     # Option: label is continuous
    ntree = 20,                                                         # Number of trees in the model
    nskip = 100,                                                        # Size of burn-in sample
    ndpost = 200,                                                       # Number of posteriors drawn
    power = 2,                                                          # beta in the tree structure prior
    base = 0.95)                                                        # alpha in the tree structure prior
```

```
## *****Calling gbart: type=1
## *****Data:
## data:n,p,np: 198128, 7, 70208
## y1,yn: -0.049921, 0.024079
## x1,x[n*p]: 0.010000, 0.810000
## xp1,xp[np*p]: 0.270000, 0.880000
## *****Number of Trees: 20
## *****Number of Cut Points: 100 ... 100
## *****burn,nd,thin: 100,200,1
```

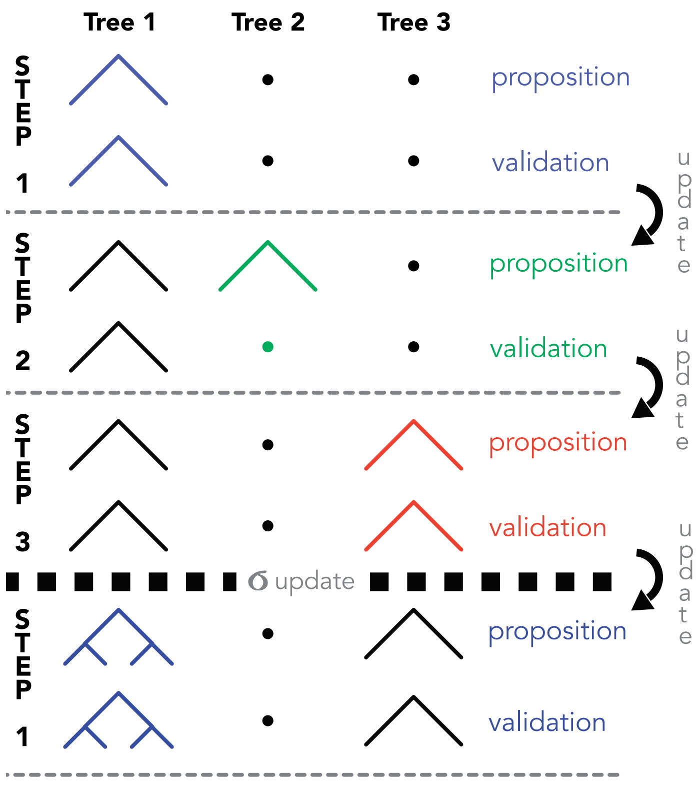


FIGURE 10.3: Diagram of the MH/Gibbs sampling of BARTs.

```
## *****Prior:beta,alpha,tau,nu,lambda,offset: 2,0.95,1.57391,3,2.84908e-31,0.0139209
## *****sigma: 0.000000
## *****w (weights): 1.000000 ... 1.000000
## *****Dirichlet:sparse,theta,omega,a,b,rho,augment: 0,0,1,0.5,1,7,0
## *****printevery: 100
##
## MCMC
## done 0 (out of 300)
## done 100 (out of 300)
## done 200 (out of 300)
## time: 26s
## trcnt,tecnt: 200,200
```

Once the model is trained,[2] we evaluated its performance. We simply compute the hit ratio. The predictions are embedded within the fit variable, under the name '*yhat.test*'.

```
mean(fit_bart$yhat.test * testing_sample$R1M_Usd > 0)
```

```
## [1] 0.542666
```

The performance *seems* reasonable but is by no means not impressive.

[2]In the case of BARTs, the training is consists exactly in the drawing of posterior samples.

11

Validating and tuning

As is shown in chapters 6 to 12, ML models require user-specified choices before they can be trained. These choices encompass parameter values (learning rate, penalization intensity, etc.) or architectural choices (e.g., the structure of a network). Alternative designs in ML engines can lead to different predictions, hence selecting a good one can be critical. We refer to the work of Probst et al. (2018) for a study on the impact of hyperparameter tuning. For some models (neural networks and boosted trees), the number of degrees of freedom is so large that finding the right parameters can become complicated and challenging. This chapter addresses these issues but the reader must be aware that there is no shortcut to building good models. Crafting an effective model is time-consuming and often the result of many iterations.

11.1 Learning metrics

The parameter values that are set before training are called **hyperparameters**. In order to be able to choose good hyperparameters, it is imperative to define metrics that evaluate the performance of ML models. As is often the case in ML, there is a dichotomy between models that seek to predict numbers (regressions) and those that try to forecast categories (classifications).

11.1.1 Regression analysis

Errors in regression analyses are usually evaluated in a straightforward way. The L^1 and L^2 norms are mainstream; they are both easy to interpret and to compute. The second one, the root mean squared error (RMSE) is differentiable everywhere but harder to grasp and gives more weight to outliers. The first one, the mean absolute error gives the average distance to the realized value but is not differentiable at zero. Formally, we define them as

$$\text{MAE}(\mathbf{y}, \tilde{\mathbf{y}}) = \frac{1}{I}\sum_{i=1}^{I}|y_i - \tilde{y}_i|, \tag{11.1}$$

$$\text{MSE}(\mathbf{y}, \tilde{\mathbf{y}}) = \frac{1}{I}\sum_{i=1}^{I}(y_i - \tilde{y}_i)^2, \tag{11.2}$$

and the RMSE is simply the square root of the MSE.

These metrics are widely used outside ML to assess forecasting errors. Below, we present other indicators that are also sometimes used to quantify the quality of a model. In line with the linear regressions, the R^2 can be computed in any predictive exercise.

$$R^2(\mathbf{y},\tilde{\mathbf{y}})=1-\frac{\sum_{i=1}^I(y_i-\tilde{y}_i)^2}{\sum_{i=1}^I(y_i-\bar{y})^2}, \tag{11.3}$$

where $\bar{y}$ is the sample average of the label. One important difference with the classical R^2 is that the above quantity can be computed on the testing sample and not on the training sample. In this case, the R^2 can be negative when the mean squared error in the numerator is larger than the (biased) variance of the testing sample. Sometimes, the average value $\bar{y}$ is omitted in the denominator (as in Gu et al. (2018) for instance). The benefit of removing the average value is that it compares the predictions of the model to a zero prediction. This is particularly relevant with returns because the simplest prediction of all is the constant zero value and the R^2 can then measure if the model beats this naive benchmark. A zero prediction is always preferable to a sample average because the latter can bery much be period dependent.

Beyond the simple indicators detailed above, several exotic extensions exist and they all consist in altering the error before taking the averages. Two notable examples are the Mean Absolute Percentage Error (MAPE) and the Mean Square Percentage Error (MSPE). Instead of looking at the raw error, they compute the error relative to the original value (to be predicted). Hence, the error is expressed in a percentage score and the averages are simply equal to:

$$\text{MAPE}(\mathbf{y},\tilde{\mathbf{y}})=\frac{1}{I}\sum_{i=1}^I\left|\frac{y_i-\tilde{y}_i}{y_i}\right|, \tag{11.4}$$

$$\text{MSPE}(\mathbf{y},\tilde{\mathbf{y}})=\frac{1}{I}\sum_{i=1}^I\left(\frac{y_i-\tilde{y}_i}{y_i}\right)^2, \tag{11.5}$$

where the latter can be scaled by a square root if need be. When the label is positive with possibly large values, it is possible to scale the magnitude of errors, which can be very large. One way to do this is to resort to the Root Mean Squared Logarithmic Error (RMSLE), defined below:

$$\text{RMSLE}(\mathbf{y},\tilde{\mathbf{y}})=\sqrt{\frac{1}{I}\sum_{i=1}^I\log\left(\frac{1+y_i}{1+\tilde{y}_i}\right)}, \tag{11.6}$$

where it is obvious that when $y_i=\tilde{y}_i$, the error metric is equal to zero.

11.1.2 Classification analysis

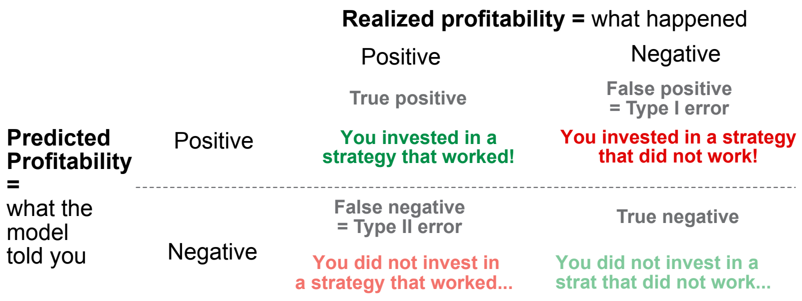
The performance metrics for categorical outcomes are substantially different compared to those of numerical outputs. A large proportion of these metrics are dedicated to binary classes, though some of them can easily be generalized to multiclass models.

We present the concepts pertaining to these metrics in an increasing order of complexity and start with the notion of true/false positive/negative. In binary classification, it is convenient to think in terms of true versus false. In an investment setting, true can be related to a positive return, or a return being above that of a benchmark - false being the opposite.

There are then 4 types of possible results for a prediction. Two when the prediction is right (predict true with true realization or predict false with false outcome) and two when the prediction is wrong (predict true with false realization and the opposite). We define the corresponding aggregate metrics below:

- frequency of true positive: $TP = I^{-1}\sum_{i=1}^I 1_{\{y_i=\tilde{y}_i=1\}}$,
- frequency of true negative: $TN = I^{-1}\sum_{i=1}^I 1_{\{y_i=\tilde{y}_i=0\}}$,
- frequency of false positive: $FP = I^{-1}\sum_{i=1}^I 1_{\{\tilde{y}_i=1,y_i=0\}}$,
- frequency of false negative: $FN = I^{-1}\sum_{i=1}^I 1_{\{\tilde{y}_i=0,y_i=1\}}$,

where true is conventionally encoded into 1 and false into 0. The sum of the four figures is equal to one. These four numbers have very different impacts on out-of-sample results, as is shown in Figure 11.1. In this table (also called a confusion matrix), it is assumed that some proxy for future profitability is forecast by the model. Each row stands for the model's prediction and each column for the realization of the profitability. The most important cases are those in the top row, when the model predicts a positive result because it is likely that assets with positive profitability (possibly relative to some benchmark) will end up in the portfolio. Of course, this is not a problem if the asset does well (left cell), but it becomes penalizing if the model is wrong because the portfolio will suffer.



		Realized profitability = what happened	
		Positive	Negative
Predicted Profitability = what the model told you	Positive	True positive You invested in a strategy that worked!	False positive = Type I error You invested in a strategy that did not work!
	Negative	False negative = Type II error You did not invest in a strategy that worked...	True negative You did not invest in a strat that did not work...

FIGURE 11.1: Confusion matrix: summary of binary outcomes.

Among the two types of errors, the type I is the most daunting for investors because it has a direct effect on the portfolio. The type II error is simply a missed opportunity and is somewhat less impactful. Finally, true negatives are those assets which are correctly excluded from the portfolio.

From the four baseline rates, it is possible to derive other interesting metrics:

- Accuracy $= TP + TN$ is the percentage of correct forecasts;
- Recall $= \frac{TP}{TP+FN}$ measures the ability to detect a winning strategy/asset (left column analysis). Also known as sensitivity or true positive rate (TPR);
- Precision $= \frac{TP}{TP+FP}$ computes the probability of good investments (top row analysis);
- Specificity $= \frac{TN}{FP+TN}$ measures the proportion of actual negatives that are correctly identified as such (right column analysis);

- Fallout $= \frac{FP}{FP+TN} = 1-$Specificity is the probability of false alarm (or false positive rate), i.e., the frequence at which the algorithm detects falsely performing assets;

- F-score, $\mathbf{F}_1 = 2\frac{\text{recall}\times\text{precision}}{\text{recall}+\text{precision}}$ is the harmonic average of recall and precision.

All of these items lie in the unit interval and a model is deemed to perform better when they increase (except for fallout for which it is the opposite). Many other indicators also exist, like the false discovery rate or false omission rate, but they are not as mainstream and less cited. Moreover, they are often simple functions of the ones mentioned above.

A metric that is popular but more complex is the area under the (ROC) curve, often referred to as AUC. The complicated part is the ROC curve where ROC stands for Receiver Operating Characteristic; the name comes from signal theory. We explain how it is built below.

As seen in Chapters 7 and 8, classifiers generate output that are probabilities that one instance belongs to one class. These probabilities are then translates into a class by choosing the class that has the highest value. In binary classification, the class with a score above 0.5 basically wins.

In practice, this 0.5 threshold may not be optimal and the model could very well correctly predict false instances when the probability is below 0.4 and true ones otherwise. Hence, it is a natural idea to test what happens if the decision threshold changes. The ROC curve does just that and plots the recall as a function of the fallout when the threshold increases from zero to one.

When the threshold is equal to 0, true positives are equal to zero because the model never forecasts positive values. Thus, both recall and fallout are equal to zero. When the threshold is equal to one, false negatives shrink to zero and true negatives too, hence recall and fallout are equal to one. The behaviour of their relationship in between these two extremes is called the ROC curve. We provide stylized examples below. A random classifier would fare equally good for recall and fallout and thus the ROC curve would be a linear line from the point (0,0) to (1,1). To prove this, imagine a sample with a $p \in (0,1)$ proportion of true instances and a classifier that predicts true randomly with a probability $p' \in (0,1)$. Then because the sample and predictions are independent, $TP = p'p$, $FP = p'(1-p)$, $TN = (1-p')(1-p)$ and $FN = (1-p')p$. Given the above definition, this yields that both recall and fallout are equal to p'.

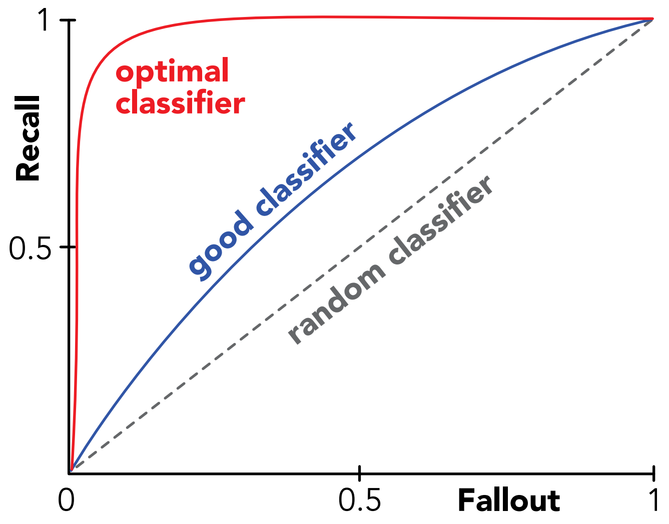


FIGURE 11.2: Stylized ROC curves.

An algorithm with a ROC curve above the 45° angle is performing better than an average classifier. Indeed, the curve can be seen as a tradeoff between benefits (probability of detecting good strategies on the y axis) minus costs (odds of selecting the wrong assets on the x axis). Hence being above the 45° is paramount. The best possible classifier has a ROC curve that goes from point (0,0) to point (0,1) to point (1,1). At point (0,1), fallout is null, hence there are no false positives, and recall is equal to one so that there are also no false negatives: the model is always right. The opposite is true: at point (1,0), the model is always wrong.

Below, we use a particular package (*caTools*) to compute a ROC curve for a given set of predictions on the testing sample.

```
if(!require(caTools)){install.packages("caTools")}
```

```
library(caTools)  # Package for AUC computation
colAUC(X = predict(fit_RF_C, testing_sample, type = "prob"),
       y = testing_sample$R1M_Usd_C,
       plotROC = TRUE)
```

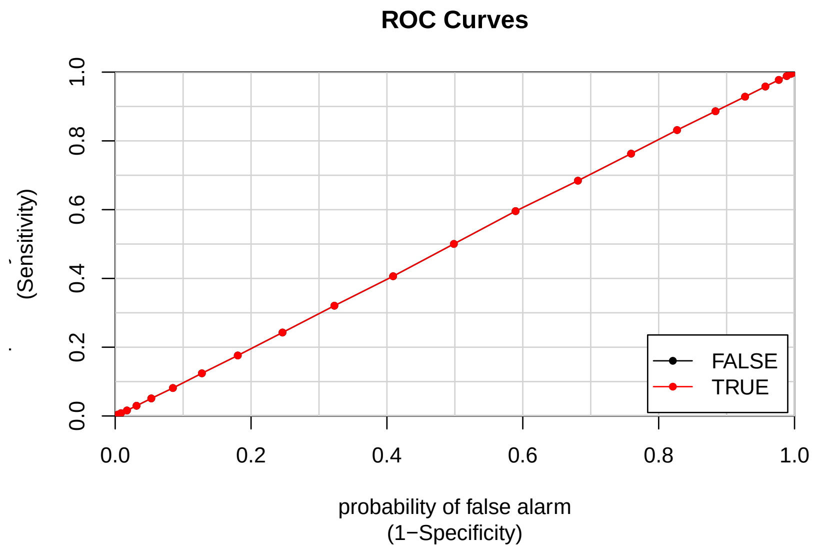


FIGURE 11.3: Example of ROC curve.

```
##                          FALSE      TRUE
## FALSE vs. TRUE 0.5003885 0.5003885
```

In the above figure, the curve is very close to the 45° angle and the model seems as good (or, rather, as bad) as a random classifier.

Finally, having one entire curve is not practical for comparison purposes, hence the information of the whole curve is synthesized into the area below the curve, i.e., the integral of the corresponding function. The 45° angle (quadrant bisector) has an area of 0.5 (it is half the unit square which has a unit area). Thus, any good model is expected to have an area under the curve (AUC) above 0.5. A perfect model has an AUC of one.

We end this subsection with a word on multiclass data. When the output (i.e., the label) has more than two categories, things become more complex. It is still possible to compute a confusion matrix, but the dimension is larger and harder to interpret. The simple indicators like TP, TN, etc., must be generalized in a non standard way. The simplest metric in the case is the cross-entropy defined in Equation ((8.10)).

11.2 Validation

Validation is the stage at which a model is tested and tuned before it starts to be deployed on real or live data (e.g., for trading purposes). Needless to say that it is critical.

11.2.1 The variance-bias tradeoff: theory

The variance-bias tradeoff is one of the core concepts in ML. To explain it, let us assume that the data is generated by the simple model

$$y_i = f(\mathbf{x}_i) + \epsilon_i, \quad \mathbb{E}[\boldsymbol{\epsilon}] = 0, \quad \mathbb{V}[\boldsymbol{\epsilon}] = \sigma^2,$$

but the model that is estimated yields

$$y_i = \hat{f}(\mathbf{x}_i) + \hat{\epsilon}_i.$$

Given an unkown sample $\mathbf{x}$, the decomposition of the average squared error is

$$\begin{aligned}
\mathbb{E}[\hat{\epsilon}^2] &= \mathbb{E}[(y - \hat{f}(\mathbf{x}))^2] = \mathbb{E}[(f(\mathbf{x}) + \epsilon - \hat{f}(\mathbf{x}))^2] \\
&= \underbrace{\mathbb{E}[(f(\mathbf{x}) - \hat{f}(\mathbf{x}))^2]}_{\text{total quadratic error}} + \underbrace{\mathbb{E}[\epsilon^2]}_{\text{irreducible error}} \\
&= \mathbb{E}[\hat{f}(\mathbf{x})^2] + \mathbb{E}[f(\mathbf{x})^2] - 2\mathbb{E}[f(\mathbf{x})\hat{f}(\mathbf{x})] + \sigma^2 \\
&= \mathbb{E}[\hat{f}(\mathbf{x})^2] + f(\mathbf{x})^2 - 2f(\mathbf{x})\mathbb{E}[\hat{f}(\mathbf{x})] + \sigma^2 \\
&= \left[\mathbb{E}[\hat{f}(\mathbf{x})^2] - \mathbb{E}[\hat{f}(\mathbf{x})]^2\right] + \left[\mathbb{E}[\hat{f}(\mathbf{x})]^2 + f(\mathbf{x})^2 - 2f(\mathbf{x})\mathbb{E}[\hat{f}(\mathbf{x})]\right] + \sigma^2 \\
&= \underbrace{\mathbb{V}[\hat{f}(\mathbf{x})]}_{\text{variance of model}} + \underbrace{\mathbb{E}[(f(\mathbf{x}) - \hat{f}(\mathbf{x}))]^2}_{\text{squared bias}} + \sigma^2
\end{aligned} \tag{11.7}$$

In the above derivation, $f(x)$ is not random, but $\hat{f}(x)$ is. Also, in the second line, we assumed $\mathbb{E}[\epsilon(f(x) - \hat{f}(x))] = 0$, which may not always hold (though it is a very common assumption). The average squared error thus has three components:

- one irreducible error (independent from the choice of a particular model);

- the variance of of the model (over its predictions);
- and the squared bias of the model.

The first one is immune to changes in models, so the challenge is to minimize the sum of the other two. This is known as the variance-bias tradeoff because reducing one often leads to increasing the other. The goal is thus to assess when a small increase in either one can lead to a larger decrease in the other.

There are several ways to represent this tradeoff and we display two of them. The first one relates to archery (see Figure 11.4) below. The best case (top left) is when all shots are concentrated in the middle: on average, the archer aims correctly and all the arrows are very close to one another. The worst case (bottom right) is the exact opposite: the average arrow is above the center of the target (the bias is nonzero) and the dispersion of arrows is large.

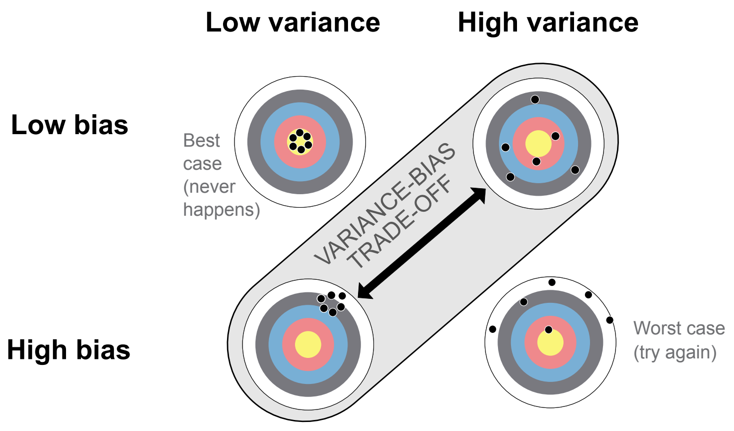


FIGURE 11.4: First representation of the variance-bias tradeoff.

The most often encountered cases in ML are the other two configurations: either the arrows (predictions) are concentrated in a small perimeter, but the perimeter is not the center of the target; or the arrows are one average well distributed around the center, but they are far from it.

The second way the variance bias tradeoff is often depicted is via the notion of model complexity. The most simple model of all is a constant one: the prediction is always the same, for instance equal to the average value of the label in the training set. Of course, this prediction will often be far from the realized values of the testing set (its bias will be large), but at least its variance is zero. On the other side of the spectrum, a decision tree with as many leaves as there are instances has a very complex structure. It will probably have a smaller bias, but undoubtedly it is not obvious that this will compensate the increase in variance incurred by the intricacy of the model.

This facet of the tradeoff is depicted in Figure 11.5 below. To the left of the graph, a simple model has a small variance but a large bias while to the right it is the opposite for a complex model. Good models often lie somewhere in the middle, but the perfect mix is hard to find.

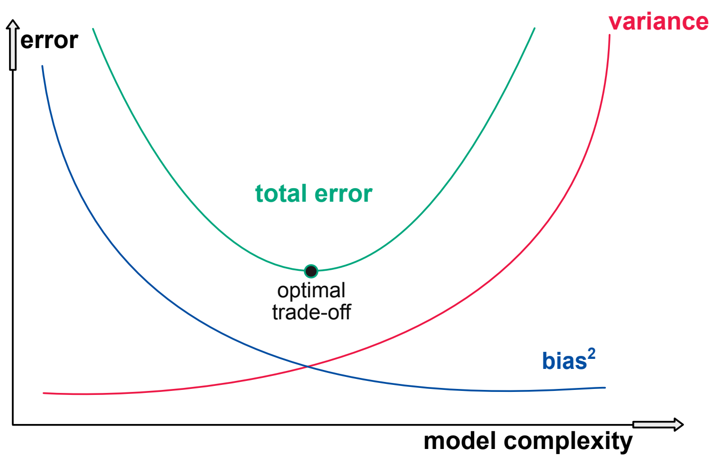


FIGURE 11.5: Second representation of the variance-bias tradeoff.

The most tractable theoretical form of the variance-bias tradeoff is the ridge regression.[1] The coefficient estimates in this type of regression are given by $\hat{\mathbf{b}}_\lambda = (\mathbf{X}'\mathbf{X} + \lambda\mathbf{I}_N)^{-1}\mathbf{X}'\mathbf{Y}$, where λ is the penalization intensity. Assuming a *true* linear form for the data generating process ($\mathbf{y} = \mathbf{X}\mathbf{b} + \boldsymbol{\epsilon}$ where $\mathbf{b}$ is unknown and σ^2 the variance of errors), this yields

$$\mathbb{E}[\hat{\mathbf{b}}_\lambda] = \mathbf{b} - \lambda(\mathbf{X}'\mathbf{X} + \lambda\mathbf{I}_N)^{-1}\mathbf{b}, \tag{11.8}$$

$$\mathbb{V}[\hat{\mathbf{b}}_\lambda] = \sigma^2(\mathbf{X}'\mathbf{X} + \lambda\mathbf{I}_N)^{-1}\mathbf{X}'\mathbf{X}(\mathbf{X}'\mathbf{X} + \lambda\mathbf{I}_N)^{-1}. \tag{11.9}$$

Basically, this means that the bias of the estimator is equal to $-\lambda(\mathbf{X}'\mathbf{X} + \lambda\mathbf{I}_N)^{-1}\mathbf{b}$, which is zero in the absence of penalization (classical regression) and converges to some finite number when $\lambda \to \infty$, i.e., when the model becomes constant. Note that if the estimator has a zero bias, then predictions will too: $\mathbb{E}[\mathbf{X}(\mathbf{b} - \hat{\mathbf{b}})] = \mathbf{0}$.

The variance (of estimates) in the case of an unconstrained regression is equal to $\mathbb{V}[\hat{\mathbf{b}}] = \sigma(\mathbf{X}'\mathbf{X})^{-1}$. In Equation ((11.9)), the λ reduces the magnitude of figures in the inverse matrix. The overall effect is that as λ increases, the variance decreases and in the limit $\lambda \to \infty$, the variance is zero when the model is constant. The variance of predictions is

$$\begin{aligned}\mathbb{V}[\mathbf{X}\hat{\mathbf{b}}] &= \mathbb{E}[(\mathbf{X}\hat{\mathbf{b}} - \mathbb{E}[\mathbf{X}\hat{\mathbf{b}}])(\mathbf{X}\hat{\mathbf{b}} - \mathbb{E}[\mathbf{X}\hat{\mathbf{b}}])'] \\ &= \mathbf{X}\mathbb{E}[(\hat{\mathbf{b}} - \mathbb{E}[\hat{\mathbf{b}}])(\hat{\mathbf{b}} - \mathbb{E}[\hat{\mathbf{b}}])']\mathbf{X}' \\ &= \mathbf{X}\mathbb{V}[\hat{\mathbf{b}}]\mathbf{X}\end{aligned}$$

All in all, ridge regressions are very handy because with a single parameter, they are able to provide a cursor that directly tunes the variance-bias tradeoff.

It's easy to illustrate how easy it is to display the tradeoff with the ridge regression. In the example below we recycle the ridge model trained in Chapter 6.

```r
ridge_errors <- predict(fit_ridge, x_penalized_test) -                      # Errors from all models
    (rep(testing_sample$R1M_Usd, 100) %>%
    matrix(ncol = 100, byrow = FALSE))
ridge_bias <- ridge_errors %>% apply(2, mean)                               # Biases
ridge_var <- predict(fit_ridge, x_penalized_test) %>% apply(2, var)         # Variance
tibble(lambda, ridge_bias^2, ridge_var, total = ridge_bias^2+ridge_var) %>%  # Plot
    gather(key = Error_Component, value = Value, -lambda) %>%
    ggplot(aes(x = lambda, y = Value, color = Error_Component)) + geom_line()
```

[1]Another angle, critical of neural networks is provided in Geman et al. (1992).

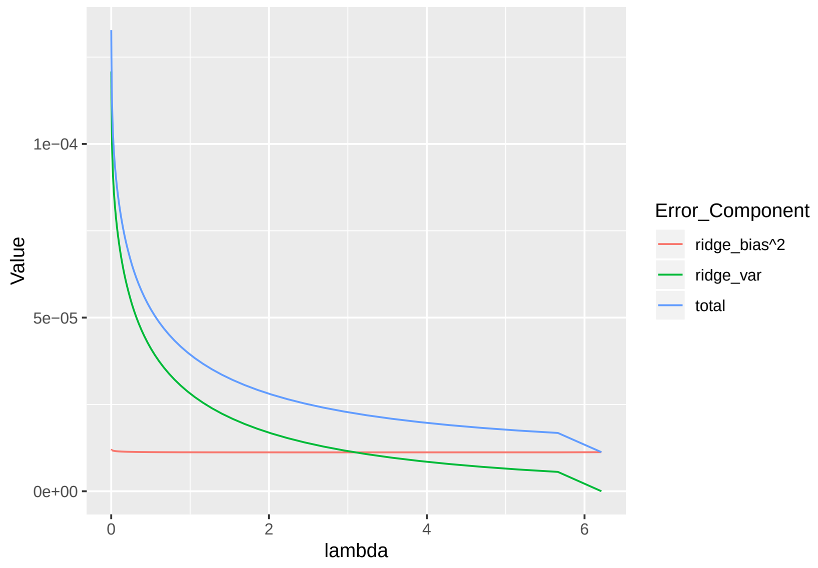


FIGURE 11.6: Error decomposition for a ridge regression

In Figure 11.6, the pattern is different from the one depicted in Figure 11.5. In the graph, when the intensity lambda increases, the magnitude of parameters shrinks and the model becomes simpler. Hence, the most simple model seems like the best choice: adding complexity increases variance but does not improve the bias! One possible reason for that is that features don't actually carry much predictive value and hence a constant model is just as good as more sophisticated ones based on unrelevant variables.

11.2.2 The variance-bias tradeoff: illustration

The variance-bias tradeoff is often presented in theoretical terms that are easy to grasp. It is nonetheless useful to demonstrate how it operates on true algorithmic choices. Below, we take the example of trees because their complexity is easy to evaluate. Basically, a tree with many terminal nodes is more complex than a tree with a handful of clusters.

We start with the parcimonious model, which we train below.

```
fit_tree_simple <- rpart(formula,
             data = training_sample,     # Data source: training sample
             cp = 0.0001,                # Precision: smaller = more leaves
             maxdepth = 2                # Maximum depth (i.e. tree levels)
             )
rpart.plot(fit_tree_simple)
```

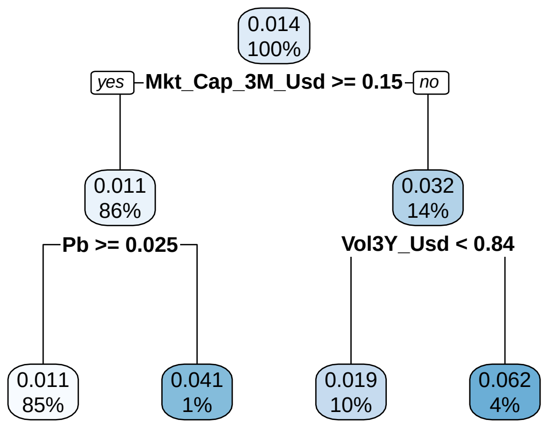


FIGURE 11.7: Simple tree.

The model only has 4 clusters, which means that the predictions can only take four values. The smallest one is 0.011 and encompasses a large portion of the sample (85%) and the largest one is 0.062 and corresponds to only 4% of the training sample.
We are then able to compute the bias and the variance of the predictions on the *testing* set.

```r
mean(predict(fit_tree_simple, testing_sample) - testing_sample$R1M_Usd) # Bias
```

```
## [1] 0.004973917
```

```r
var(predict(fit_tree_simple, testing_sample))                           # Variance
```

```
## [1] 0.0001398003
```

On average, the error is slightly positive, with an overall overestimation of 0.005 . As expected, the variance is very small (10^{-4}).

For the complex model, we take the boosted tree that was obtained in Section 7.4.6 (fit_xgb). The model aggregates 40 trees with a maximum depth of 4, it is thus undoubtedly more complex.

```r
mean(predict(fit_xgb, xgb_test) - testing_sample$R1M_Usd) # Bias
```

```
## [1] 0.00324996
```

```r
var(predict(fit_xgb, xgb_test))                           # Variance
```

```
## [1] 0.002206797
```

The bias is indeed smaller compared to that of the simple model, but in exchange, the variance increases sustantially. The net effect (via the *squared bias*) is in favor of the simpler model.

11.2.3 The risk of overfitting: principle

The notion of overfitting is one of the most important in machine learning. When a model overfits, the accuracy of its predictions will be disappointing, thus it is one major reason why *some* strategies fail out-of-sample. Therefore, it is important to understand not only what overfitting is, but also how to mitigate its effects.

One recent reference on this topic and its impact on portfolio strategies is Hsu et al. (2018), which builds on the work of White (2000). Both these reference do not deal with ML models, but the principle is the same. When given a dataset, a sufficiently intense level of analysis (by a human or a machine) will always be able to detect some patterns. Whether these patterns are spurious are not is the key question.

In Figure 11.8, we illustrate this idea with a simple visual example. We try to find a model that maps x into y. The data points are the small black circles. The simplest model is the constant one (only one parameter), but with two parameters (level and slope), the fit is already quite good. This is shown with the blue line. With a sufficient number of parameters, it is possible to build a model that flows through all the points. One example would be a high dimensional polynomial. One such models is represented with the red line. Now there seems to be a strange point in the dataset and the complex model fits closely to match this point.

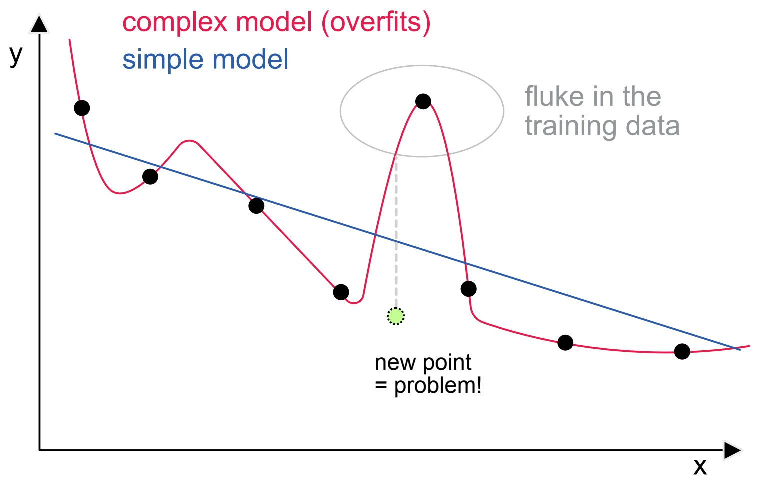


FIGURE 11.8: Confusion matrix: summary of binary outcomes.

A new point is added in light green. It is fair to say that is follows the general pattern of the other points. The simple model is not perfect and the error is non-negligible. Nevertheless, the error stemming from the complex model (shown with the dotted gray line) is approximately twice as large. This simplified example shows that models that are too close to the training data will catch idiosyncracies that will not occur in other datasets. A good model would overlook these idiosyncracies and stick to the enduring structure of the data.

11.2.4 The risk of overfitting: some solutions

Obviously, the easiest way to avoid overfitting is to resist to the temptation of complicated models (e.g., high dimensional neural networks or tree ensembles).

The complexity of models is often proxied via two measures: the number of parameters of the model and their magnitude (often synthesized through their norm). These proxies are not perfect because some *complex* models may only require a small number of parameters (or even small parameter values), but at least they are straightforward and easy to handle. There is no universal way of handling overfitting. Below, we detail a few tricks for some families of ML tools.

For **regressions**, there are two simple ways to deal with overfitting. The first is the number of parameters, that is, the number of predictors. Sometimes, it can be better to only select a subsample of features, especially if some of them are highly correlated (often, a threshold of 70% is considered as too high for absolute correlations between features). The second solution is penalization, which helps reduce the magnitude of estimates and thus of the variance of predictions.

For tree-based methods, there are a variety of ways to reduce the risk of overfitting. When dealing with **simple trees**, the only way to proceed is to limit the number of leaves. This can be done in many ways. First, by imposing a maximum depth. If it is equal to d, then the tree can have at most 2^d terminal nodes. It is often advised not to go beyond $d = 6$. The complexity parameter in *rpart* is another way the shrink the size of trees. When it is large, any new split must lead to a substantial reduction in loss. If not, the split is not deemed useful and is thus not performed. The last two parameters are the minimum number of instances required in each leaf and the minimum number of instance per cluster requested in order to continue the splitting process. The higher (i.e., the more coercive) these figures are, the harder it is to grow complex trees.

In addition to these options, **random forests** allow to control for the number of trees in the forest. Theoretically (see Breiman (2001)), this parameter is not supposed to impact the variance-bias tradeoff. In practice, and for the sake of computation times, it is not recommended to go beyong 1,000 trees. Two other hyperparameters are the subsample size (on which each learner is trained) and the number of features retained for learning. They do not have a straightforward impact of bias and tradeoff, but rather on raw performace. For instance, if subsamples are too small, the trees will not learn enough. Same problem if the number of features is too low. On the other hand, choosing a large number of predictors (i.e., close to the total number) may lead to high correlations between each learner's prediction because the overlap in information contained in the training samples may be high.

Boosted trees have other options that can help alleviate the risk of overfitting. The most obvious one is the learning rate, which discounts the impact of each new tree by a large factor. When the learning rate is high, the algorithm learns too fast and is prone to sticking close to the training data. When it's low, the model learns very progressively, which can be efficient if there are sufficiently many trees in the ensemble. Indeed, the learning rate and the number of trees must be chosen synchronously: if both are low, the ensemble will learn nothing and if both are large, it will overfit. The arsenal of boosted trees does not stop there. The penalizations, both of score values and of the number of leaves, are naturally a tool to prevent the model from going to deep in the particularities of the trainig sample. Finally, constrainsts of monotonicity like those mentioned in Section 7.4.5 are also an efficient way to impose some structure on the model and force it to detect particular patterns.

Lastly **neural networks** also have many options aimed at protecting them against overfitting. Just like for boosted trees, one of them is the penalization of weights and biases (via their norm). Constraints, like nonnegative constraints can also help when the modeltheoretically requires positive inputs. Finally, dropout is always a direct way to reduce the dimension (number of parameters) of a network.

11.3 The search for good hyperparameters

11.3.1 Methods

Let us assume that there are p parameters to be defined before a model is run. The simplest way to proceed is to test different values of these parameters and choose the one that yields the best results. There are mainly two ways to perform these tests: independently and sequentially.

Independent tests are easy and come in two families: grid (deterministic) search and random exploration. The advantage of a deterministic approach is that is covers the space uniformly and makes sure that no corners are omitted. The drawback is the computation time. Indeed, for each parameter, it seems reasonable to test at least five values, which makes 5^p combinations. If p is small (smaller than 3), this is manageable when the backtests are not too lengthy. When p is large, the number of combinations may become prohibitive. This is when random exploration can be useful because in this case, the user speciifies the number of tests upfront and the parameters are drawn randomly (usually uniformly). The flaw in random search is that some areas in the parameter space may not be covered, which can be problematic if the best choice is located there. It is nonetheless shown in Bergstra and Bengio (2012) that random exploration is preferable to grid search.

Both grid and random searches are suboptimal because they are likely to spend time in zones of the parameter space that are irrelevant, thereby wasting computation time. Given a number of parameter points that have been tested, it is preferable to focus the search in areas where the best points are the most likely. This is possible via an interative process that adapts the search after each new point has been tested.

One other popular approach in this direction is Bayesian optimization (BO). The central object is the objective function of the learning process. We call this function O and it can be widely seen as a loss function possibly combined with penalization and constraints. For simplicity here, we will not mention the training/testing samples and they are considered to be fixed. The variable of interest is the vector $\mathbf{p} = (p_1, \dots, p_l)$ which synthesizes the hyperparameters (learning rate, penalization intensities, number of models, etc.) that have an impact on O. The program we are interested in is

$$\mathbf{p}_* = \underset{\mathbf{p}}{\text{argmin}}\ O(\mathbf{p}). \tag{11.10}$$

The main problem with this optimization is that the computation of $O(\mathbf{p})$ is very costly. Therefore, it is critical to choose each trial for $\mathbf{b}$ wisely. One key assumption of BO is that the distribution of O is Gaussian and that O can be proxied by a linear combination of the p_l. Said differently, the aim is to build a Bayesian linear regression between the input $\mathbf{p}$ and the output (dependent variable) O. Once a model has been estimated the information that is concentrated in the posterior density of O is used to make an educated guess at where to look at for new values of $\mathbf{p}$.

This educated guess is made based on a so-called *acquisition function*. Suppose we have tested m values for $\mathbf{p}$, which we write $\mathbf{p}^{(m)}$. The current best parameter is written $\mathbf{p}_m^* = \underset{1\le k\le m}{\text{argmin}}\ O(\mathbf{p}^{(k)})$. If we test a new point $\mathbf{p}$, then it will lead to an improvement only if

$O(\mathbf{p}) < O(\mathbf{p}_m^*)$, that is if the new objective improves the minimum value that we already know. The average value of this improvement is

$$\mathbf{EI}_m(\mathbf{p}) = \mathbb{E}_m[[O(\mathbf{p}_m^*) - O(\mathbf{p})]_+], \tag{11.11}$$

where the positive part $[\cdot]_+$ emphasizes that when $O(\mathbf{p}) \geq O(\mathbf{p}_m^*)$, the gain is zero. The expectation is indexed by m because it is computed with respect to the posterior distribution of $O(\mathbf{p})$ based on the m samples $\mathbf{p}^{(m)}$. The best choice for the next sample $\mathbf{p}^{m+1}$ is then

$$\mathbf{p}^{m+1} = \underset{\mathbf{p}}{\text{argmax}}\ \mathbf{EI}_m(\mathbf{p}), \tag{11.12}$$

which corresponds to the maximum location of the expected improvement. Instead of the EI, the optimization can be performed on other measures, like the probability of improvement, which is $\mathbb{P}_m[O(\mathbf{p}) < O(\mathbf{p}_m^*)]$.

In compact form, the iterative process can be outlined as follows:

- **step 1**: compute $O(\mathbf{p}^{(m)})$ for $m = 1, \dots, M_0$ values of parameters.
- **step 2a**: compute sequentially the posterior density of O on all available points.
- **step 2b**: compute the optimal new point to test $\mathbf{p}^{m+1}$ given in Equation (11.12).
- **step 2c**: compute the new objective value $O(\mathbf{p}^{m+1})$.
- **step 3**: repeat steps 2a to 2c as much as deemed reasonable and return the $\mathbf{p}^m$ that yields the smallest objective value.

The interested reader can have a look at Snoek et al. (2012) and Frazier (2018) for more details on the numerical facets of this method.

Finally, for the sake of completeness, we mention a last way to tune hyperparameters. Since the optimization scheme is $\underset{\mathbf{p}}{\text{argmin}}\ O(\mathbf{p})$, a natural way to proceed would be to use the sensitivity of O with respect to $\mathbf{p}$. Indeed, if the gradient $\frac{\partial O}{\partial p_l}$ is known, then a gradient descent will always improve the objective value. The problem is that it is hard to compute a reliable gradient (finite differences can become costly). Nonetheless, some methods (e.g., Maclaurin et al. (2015)) have been applied successfully to optimize over large dimensional parameter spaces.

11.3.2 Example: grid search

In order to illustrate the process of grid search, we will try to find the best parameters for a boosted tree. We seek to quantify the impact of three parameters:

- **eta**, the learning rate,
- **nrounds**, the number of trees that are grown,
- **lambda**, the weight regulariser which penalises the objective function through the total sum of squared weights/scores.

Below, we create a grid with the values we want to test for these parameters.

```r
eta <- c(0.1, 0.3, 0.5, 0.7, 0.9)         # Values for eta
nrounds <- c(10, 50, 100)                 # Values for nrounds
lambda <- c(0.01, 0.1, 1, 10, 100)        # Values for lambda
pars <- expand.grid(eta, nrounds, lambda) # Exploring all combinations!
eta <- pars[,1]
nrounds <- pars[,2]
lambda <- pars[,3]
```

Given the computational cost of grid search, we perform the exploration on the dataset with the small number of features (which we recycle from Chapter 7). In order to avoid the burden of loops, we resort to the functional programming capabilities of R, via the *purrr* package. This allows us to define a function that will lighten and simplify the code. This function, coded below, takes data and parameter inputs and returns an error metric for the algorithm. We choose the mean squared error to evaluate the impact of hyperparameter values.

```r
grid_par <- function(train_matrix, test_features, test_label, eta, nrounds, lambda){
    fit <- train_matrix %>%
        xgb.train(data = .,                           # Data source (pipe input)
                  eta = eta,                          # Learning rate
                  objective = "reg:linear",           # Objective function
                  max_depth = 5,                      # Maximum depth of trees
                  lambda = lambda,                    # Penalisation of leaf values
                  gamma = 0.1,                        # Penalisation of number of leaves
                  nrounds = nrounds,                  # Number of trees used
                  verbose = 0                         # No comment from algo
        )

    pred <- predict(fit, test_features)               # Preditions based on fitted model & test values
    return(mean((pred-test_label)^2))                 # Mean squared error
}
```

The grid_par function can then be processed by the functional programming tool **pmap** that is going to perform the loop on parameter values automatically.

```r
# grid_par(train_matrix_xgb, xgb_test, testing_sample$R1M_Usd, 0.1, 3, 0.1) # Test of the function
grd <- pmap(list(eta, nrounds, lambda),             # Parameters for the grid search
            grid_par,                               # Function on which to apply the grid search
            train_matrix = train_matrix_xgb,        # Input for function: training data
            test_features = xgb_test,               # Input for function: test features
            test_label = testing_sample$R1M_Usd     # Input for function: test labels (returns)
)
grd <- data.frame(eta, nrounds, lambda, error = unlist(grd)) # Dataframe with all results
```

Once the squared mean errors have been gathered, it is possible to plot them. We chose to work with 3 parameters on purpose because their influence can be simultaneuously plotted on one graph.

```r
grd$eta <- as.factor(eta)                                 # Parameters as categories (for plotting)
grd %>% ggplot(aes(x = eta, y = error, fill = eta)) +     # Plot!
    geom_bar(stat = "identity") +
    facet_grid(rows = vars(nrounds), cols = vars(lambda))
```

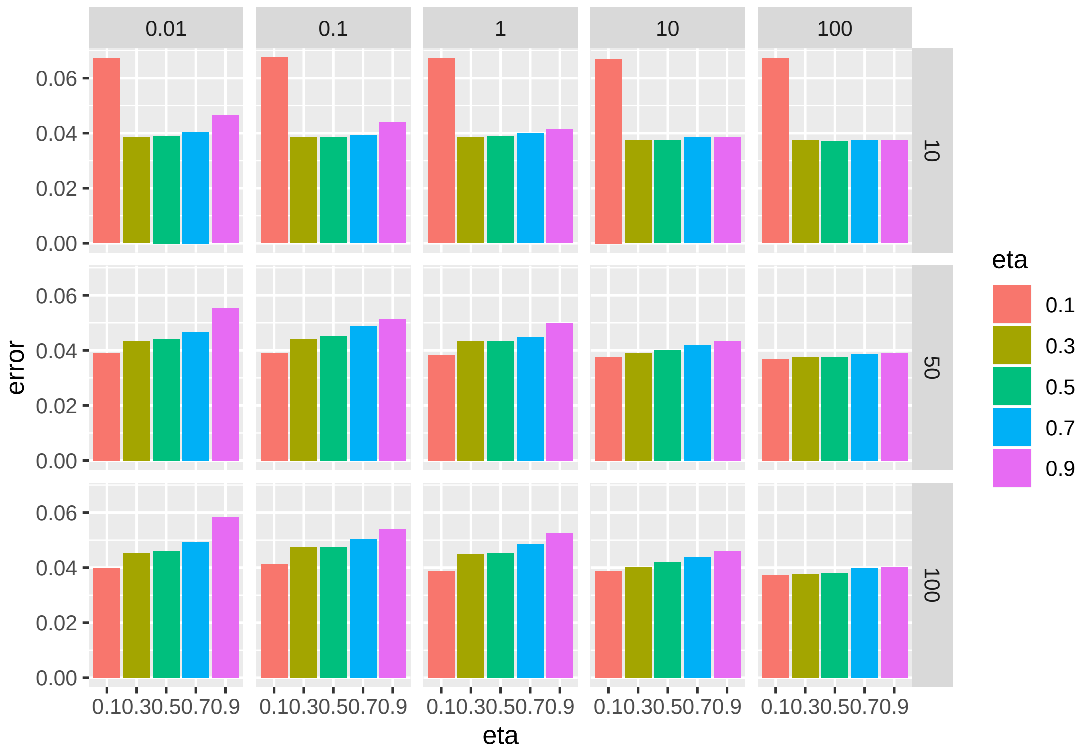


FIGURE 11.9: Plot of error metrics (SMEs) for many parameter values. Each row of graph corresponds to nrounds and each column to lambda.

In Figure 11.9, the main information is that a small learning rate ($\eta = 0.1$) is detrimental to the quality of the forecasts. This remains true even when the number of trees is large (nrounds=100), which means that the algorithm does not learn enough.

Grid search can be performed in two stages: the first stage helps locate the zones that are of interest (with the lowest loss/objective values) and then zoom in on these zones with refined values for the parameter on the grid. With the results above, this would means shrinking the support of eta to $[0.3, 0.7]$ and also probably avoid the lowest values for lambda.

11.3.3 Example: Bayesian optimization

There are several packages in R that relate to Bayesian optimization. We work with *rBayesianOptimization*, which is general purpose but also needs more coding involvment.

Just as for the grid search, we need to code the objective function on which the hyperparameters will be optimized. Under *rBayesianOptimization*, the output has to have a particular form, with a score and a prediction variable. The function will *maximize* the score, hence we will define it as minus the mean squared error.

```
bayes_par_opt <- function(train_matrix = train_matrix_xgb,        # Input for function: training data
            test_features = xgb_test,                             # Input for function: test features
            test_label = testing_sample$R1M_Usd,                  # Input for function: test label
            eta, nrounds, lambda){                                # Input for function: parameters
    fit <- train_matrix %>%
        xgb.train(data = .,                         # Data source (pipe input)
                  eta = eta,                        # Learning rate
                  objective = "reg:linear",         # Objective function
                  max_depth = 5,                    # Maximum depth of trees
                  lambda = lambda,                  # Penalisation of leaf values
                  gamma = 0.1,                      # Penalisation of number of leaves
                  nrounds = round(nrounds),         # Number of trees used
                  verbose = 0                       # No comment from algo
        )

    pred <- predict(fit, test_features)             # Preditions based on fitted model & test values
    list(Score = -mean((pred-test_label)^2),        # Minus RMSE
         Pred = pred)                               # Predictions on test set
}
```

Once the objective function is defined, it can be fed to the Bayesian optimizer.

```
library(rBayesianOptimization)
bayes_opt <- BayesianOptimization(bayes_par_opt,            # Function to maximize
                     bounds = list(eta = c(0.2, 0.8),       # Bounds for eta
                                   lambda = c(0.5, 15),     # Bounds for lambda
                                   nrounds = c(10, 100)),   # Bounds for nrounds
                     init_points = 10,             # Nb initial points for first estimation
                     n_iter = 24,                  # Nb optimization steps/trials
                     acq = "ei",                   # Acquisition function = expected improvement
                     verbose = FALSE)
```

```
##
##  Best Parameters Found:
## Round = 26   eta = 0.3954    lambda = 11.8355    nrounds = 12.0519   Value = -0.0372
```

```
bayes_opt$Best_Par
```

```
##        eta     lambda    nrounds
##  0.3953682 11.8354694 12.0519044
```

The final parameters indicate that it is advised to resist overfitting: small number of learners and large penalization seem to be the best choices.

As a confirmation of these results, we plot the relationship between the loss (up to the sign) and two hyperparameters. Each point corresponds to a value tested in the optimization. The best values are clearly to the left of the left graph and to the right of the right graph and the pattern is reliably pronounced.

```
library("ggpubr") # Package for combining plots
plot_rounds <- bayes_opt$History %>%
    ggplot(aes(x = nrounds, y = Value)) + geom_point() + geom_smooth(method = "lm")
plot_lambda <- bayes_opt$History %>%
    ggplot(aes(x = lambda, y = Value)) + geom_point() + geom_smooth(method = "lm")
par(mar = c(1,1,1,1))
ggarrange(plot_rounds, plot_lambda, ncol = 2)
```

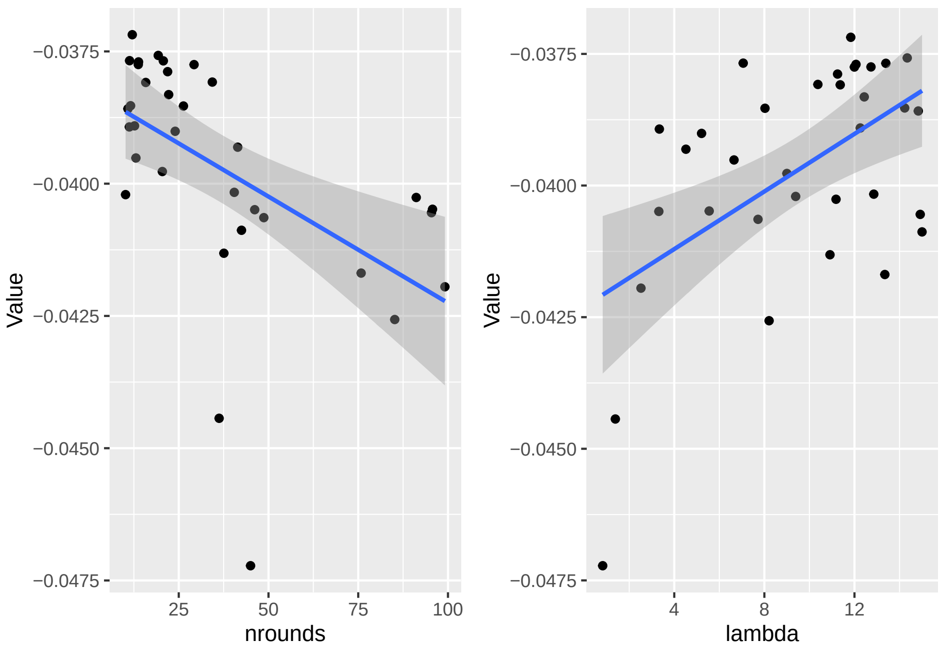


FIGURE 11.10: Relationship between (minus) loss and hyperparameter values.

11.4 Short discussion on validation in backtests

The topic of validation in backtest is more complex than it can seem. There are in fact two scales at which it can operate, depending on whether the forecasting model is dynamic (updated at each rebalancing) or fixed.

Let us start with the first option. In this case, the aim is to build a unique model and to test it on different time periods. There is an ongoing debate on the methods that are suitable to validate a model in that case. Usually, it make sense to test the model on successive dates, moving forward posterior to the training. This is what makes more sense, as it replicates what would happen in a live situation.

In machine learning, a popular approach is to split the data into K partitions and to test K different models: each one is tested one on of the partitions but trained on the $K-1$ others. This so-called cross-validation (CV) is proscribed by most experts (and common sense) for a simple reason: most of the time, the training set encompasses data from future dates and tests on past values. Nonetheless, some advocate one particular form of CV that aims at making sure that there is no informational overlap between the training and testing set (Sections 7.4 and 12.4 in De Prado (2018)). The premise is that if the structure of the cross-section of returns is constant through time, then training on future points and testing on past data is not problematic as long as there is no overlap.

One example cited in De Prado (2018) is the reaction to a model to a unseen crisis. Following the market crash of 2008, at least 11 years have followed without any major financial shake.

One option to test the reaction of a recent model to a crash would be to train it on recent years (say 2015-2019) and test it on various points (e.g., months) in 2008.

The advantage of a fixed model is that validation is easy: for one set of hyperparameters, test the model on a set of dates, and evaluate the performance of the model. Repeat the process for other parameters and choose the best alternative (or use Bayesian optimization).

The second major option is when the model is updated (retrained) at each rebalancing. The underlying idea here is that the structure of returns evolves through time and a dynamic model will capture the most recent trends. The drawback is that validation can then take place at each rebalancing date.

Let us recall the dimensions of backtests:
- number of **strategies**: possibly dozens or hundreds, or even more;
- number of trading **dates**: hundreds for monthly rebalancing;
- number of **assets**: hundreds or thousands;
- number of **features**: dozens or hundreds.

Even with a lot of computational power (GPUs, etc.), training many models over many dates is time-consuming, especially when it comes to hyper-parameter tuning when the parameter space is large. Thus, validating models at each trading date of the out-of-sample period is not realistic.

One solution is to keep an early portion of the training data and to perform a smaller scale validation on this subsample. Hyperparameters are tested on a limited number of dates and most of the time, they exhibit stability: satisfactory parameters for one date are usual acceptable for the next one and the following one as well. Thus, the full backtest can be carried out with these values when updating the models at each period. The backtest nonetheless remains compute-intensive because the model has to be re-trained with the most recent data for each rebalancing moment.

12

Ensemble models

Let us be honest. When facing a prediction task, it is not obvious to determine the best choice between ML tools: penalized regressions, tree methods, neural networks, SVMs, etc. A natural and tempting alternative is to combine several algorithms (or the predictions that result from them) to try to extract value out of each engine (or learner). This intention is not new and contributions towards this goal go back at least to Bates and Granger (1969) (for the prupose of passenger flow forecasting).

Below, we outline a few books on the topic of ensembles. The first four are monographs while the last two are compilations of contributions:

- Zhou (2012): a very didactic book that covers the main ideas of ensembles;
- Schapire and Freund (2012): the main reference for boosting (and hence, ensembling) with many theoretical results and thus strong mathematical groundings;
- Seni and Elder (2010): an introduction dedicated to tree methods mainly;
- Claeskens and Hjort (2008): an overview of model selection techniques with a few chapter focused on model averaging;
- Zhang and Ma (2012): a collection of thematic chapters on ensemble learning;
- Okun et al. (2011): examples of applications of ensembles.

In this chapter, we cover the basic ideas and concepts behind the notion of ensembles. We refer to the above books for deeper treatments on the topic. We underline that several ensemble methods have already been mentioned and covered earlier, notably in Chapter 7. Indeed, random forests and boosted trees are examples of ensembles. Hence, other early articles on the combination of learners are Schapire (1990), Jacobs et al. (1991) (for neural networks particularly), and Freund and Schapire (1997).

12.1 Linear ensembles

12.1.1 Principles

In this chapter we adopt the following notations. We work with M models where $\tilde{y}_{i,m}$ is the prediction of model m for instance i and errors $\epsilon_{i,m} = y_i - \tilde{y}_{i,m}$ are stacked into a $(I \times M)$ matrix $\mathbf{E}$. A linear combination of models has sample errors equal to $\mathbf{Ew}$, where $\mathbf{w} = w_m$ are the weights assigned to each model and we assume $\mathbf{w}'\mathbf{1}_M = 1$. Minimizing the total (squared) error is thus a simple quadratic program with unique constraint. The Lagrange function is $L(\mathbf{w})=\mathbf{w}'\mathbf{E}'\mathbf{E}\mathbf{w}-\lambda(\mathbf{w}'\mathbf{1}_M-1)$ and hence

$$\frac{\partial}{\partial \mathbf{w}}L(\mathbf{w})=\mathbf{E}'\mathbf{E}\mathbf{w}-\lambda\mathbf{1}_M=0 \quad \Leftrightarrow \quad \mathbf{w}=\lambda(\mathbf{E}'\mathbf{E})^{-1}\mathbf{1}_M,$$

and the constraint imposes $\mathbf{w}^*=\frac{(\mathbf{E}'\mathbf{E})^{-1}\mathbf{1}_M}{(\mathbf{1}_M'\mathbf{E}'\mathbf{E})^{-1}\mathbf{1}_M}$. This form is similar to that of minimum variance portfolios. If errors are unbiased ($\mathbf{1}_I'\mathbf{E}=\mathbf{0}_M'$), then $\mathbf{E}'\mathbf{E}$ is the covariance matrix of errors.

This expression shows an important feature of optimized linear ensembles: they can only add value if the models tell different stories. If two models are redundant, $\mathbf{E}'\mathbf{E}$ will be close to singular and $\mathbf{w}^*$ will arbitrage one against the other in a spurious fashion. This is the exact same problem as when mean-variance portfolios are constituted with highly correlated assets: in this case, diversification fails because when things go wrong, all assets go down. Another problem arises when the number of observations is too small compared to the number of assets so that the covariance matrix of returns is singular. This is not an issue for ensembles because the number of observations will usually be much larger than the number of models ($I>>M$).

In the limit when correlations increase to one, the above formulation becomes highly unstable and ensembles cannot be trusted. One heuristic way to see this is when $M=2$ and

$$\mathbf{E}'\mathbf{E}=\left[\begin{array}{cc}\sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2\end{array}\right] \quad \Leftrightarrow \quad (\mathbf{E}'\mathbf{E})^{-1}=\frac{1}{1-\rho^2}\left[\begin{array}{cc}\sigma_1^{-2} & -\rho(\sigma_1\sigma_2)^{-1} \\ -\rho(\sigma_1\sigma_2)^{-1} & \sigma_2^{-2}\end{array}\right]$$

so that when $\rho\rightarrow 1$, the model with the smallest errors (minimum σ_i^2) will see its weight increasing towards infinity while the other model will have a similarly large negative weight.

One improvement proposed to circumvent this trouble, advocated in a seminal publication (Breiman (1996)), is to enforce positivity constraints on the weights and solve

$$\underset{\mathbf{w}}{\text{argmin}}\ \mathbf{w}'\mathbf{E}'\mathbf{E}\mathbf{w}, \quad \text{s.t.} \quad \left\{\begin{array}{l}\mathbf{w}'\mathbf{1}_M=1 \\ w_m\ge 0 \quad \forall m\end{array}\right. .$$

Mechanically, if several models are highly correlated, the constraint will impose that only one of them will have a nonzero weight. If there are many models, then just a few of them will be selected by the minimization program. In a very different context, Jagannathan and Ma (2003) have shown the benefits of constraint in the construction mean-variance portfolios. In our setting, the constraint will similarly help discriminate wisely among the 'best' models.

In the literature, forecast combination and model averaging (which are synonyms of ensembles) have been tested on stock markets as early as in Von Holstein (1972). Surprisingly, the articles were not published in Finance journals but rather in fields such as Management (Virtanen and Yli-Olli (1987), Wang et al. (2012)), Economics and Econometrics (Donaldson and Kamstra (1996), Clark and McCracken (2009)), Operations Reasearch (Huang et al. (2005), Leung et al. (2001), and, recently, Bonaccolto and Paterlini (2019)), and Computer Science (Harrald and Kamstra (1997), Hassan et al. (2007)).

In the general forecasting literature, many alternative (refined) methods for combining forecasts have been studied. Trimmed opinion pools (Grushka-Cockayne et al. (2016)) compute averages over the predictions that are not too extreme. We refer to Gaba et al. (2017) for a more exhaustive list of combinations as well as for an empirical study of their respective efficiency. Overall, findings are mixed and the heuristic simple average is, as usual, hard to beat (see, e.g., Genre et al. (2013))

12.1.2 Example

In order to build an ensemble, we must gather the predictions and the corresponding errors into the **E** matrix. We will work with 5 models that were trained in the previous chapters: penalized regression, simple tree, random forest, xgboost and feedforward neural network. The training errors have zero means, hence $\mathbf{E}'\mathbf{E}$ is the covariance matrix of errors between models.

```
err_pen_train <- predict(fit_pen_pred, x_penalized_train) - training_sample$R1M_Usd  # Regression
err_tree_train <- predict(fit_tree, training_sample) - training_sample$R1M_Usd       # Tree
err_RF_train <- predict(fit_RF, training_sample) - training_sample$R1M_Usd           # Random Forest
err_XGB_train <- predict(fit_xgb, train_matrix_xgb) - training_sample$R1M_Usd        # XGBoost
err_NN_train <- predict(model, NN_train_features) - training_sample$R1M_Usd          # Neural Network
E <- cbind(err_pen_train, err_tree_train, err_RF_train, err_XGB_train, err_NN_train) # The E matrix
colnames(E) <- c("Pen_reg", "Tree", "RF", "XGB", "NN")                               # Column names
cor(E)                                                                               # Corr. matrix
```

```
##           Pen_reg      Tree        RF       XGB        NN
## Pen_reg 1.0000000 0.9982507 0.9968224 0.9475378 0.9959097
## Tree    0.9982507 1.0000000 0.9973481 0.9459733 0.9964547
## RF      0.9968224 0.9973481 1.0000000 0.9446178 0.9971627
## XGB     0.9475378 0.9459733 0.9446178 1.0000000 0.9437547
## NN      0.9959097 0.9964547 0.9971627 0.9437547 1.0000000
```

As is shown by the correlation matrix, the models fail to generate heterogeneity in their predictions. The minimum correlation (though above 95%!) is obtained by the boosted tree models. Below, we compare the training accuracy of models by computing the average absolute value of errors.

```
apply(abs(E), 2, mean) # Mean absolute error or columns of E
```

```
##    Pen_reg       Tree         RF        XGB         NN
## 0.08345916 0.08366795 0.08327121 0.08916813 0.08375389
```

The best performing ML engine is the random forest. The boosted tree model is the worst, by far. Below, we compute the optimal (non constrained) weights for the combination of models.

```
w_ensemble <- solve(t(E) %*% E) %*% rep(1,5)                                         # Optimal weights
w_ensemble <- w_ensemble / sum(w_ensemble)
w_ensemble
```

```
##                 [,1]
## Pen_reg -0.608296947
## Tree     0.017714031
## RF       1.334108376
## XGB     -0.002460814
## NN       0.258935354
```

Because of the high correlations, the optimal weights are not balanced and diversified: they load heavily on the random forest learner (best in sample model) and ‘short’ a few models in order to compensate.

Note that the weights are of course computed with **training errors**. The optimal combination is then tested on the testing sample. Below, we compute out-of-sample (testing) errors and their average absolute value.

```
err_pen_test <- predict(fit_pen_pred, x_penalized_test) - testing_sample$R1M_Usd     # Regression
err_tree_test <- predict(fit_tree, testing_sample) - testing_sample$R1M_Usd          # Tree
err_RF_test <- predict(fit_RF, testing_sample) - testing_sample$R1M_Usd              # Random Forest
err_XGB_test <- predict(fit_xgb, xgb_test) - testing_sample$R1M_Usd                  # XGBoost
err_NN_test <- predict(model, NN_test_features) - testing_sample$R1M_Usd             # Neural Network
E_test <- cbind(err_pen_test, err_tree_test, err_RF_test, err_XGB_test, err_NN_test) # The E matrix
colnames(E_test) <- c("Pen_reg", "Tree", "RF", "XGB", "NN")
apply(abs(E_test), 2, mean) # Mean absolute error or columns of E
```

```
##    Pen_reg       Tree         RF        XGB         NN
## 0.06618181 0.06650492 0.06710349 0.07149006 0.06766225
```

The boosted tree model is still the worst performing algorithm while the simple models (regression and simple tree) are the ones that fare the best. The most naive combination is the simple average of model and predictions.

```
err_EW_test <- apply(E_test, 1, mean)  # Equally weighted combination
mean(abs(err_EW_test))
```

```
## [1] 0.06695977
```

Because the errors are very correlated, the equally-weighted combination of forecasts yields an average error which lies 'in the middle' of individual errors. The diversification benefits are too small. Let us now test the 'optimal' combination $\mathbf{w}^*=\frac{(\mathbf{E}'\mathbf{E})^{-1}\mathbf{1}_M}{(\mathbf{1}_M'\mathbf{E}'\mathbf{E})^{-1}\mathbf{1}_M}$.

```
err_opt_test <- E_test %*% w_ensemble   # Optimal unconstrained combination
mean(abs(err_opt_test))
```

```
## [1] 0.06838265
```

Again, the result is disappointing because of the lack of diversification across models. The correlations are high not only on the training sample, but also on the testing sample, as shown below.

```
cor(E_test)
```

```
##           Pen_reg      Tree        RF       XGB        NN
## Pen_reg 1.0000000 0.9985518 0.9968882 0.9706767 0.9958360
## Tree    0.9985518 1.0000000 0.9977644 0.9752078 0.9968229
## RF      0.9968882 0.9977644 1.0000000 0.9776554 0.9974873
## XGB     0.9706767 0.9752078 0.9776554 1.0000000 0.9776927
## NN      0.9958360 0.9968229 0.9974873 0.9776927 1.0000000
```

The leverage from the optimal solution only exacerbates the problem and underperforms the heuristic uniform combination. We end this section with the constrained formulation of Breiman (1996) using the *quadprog* package. If we write $\mathbf{\Sigma}$ for the covariance matrix of errors, we seek

$$\mathbf{w}^*=\underset{\mathbf{w}}{\text{argmin}} \ \mathbf{w}'\mathbf{\Sigma}\mathbf{w}, \quad \mathbf{1}'\mathbf{w}=1, \quad w_i\ge 0,$$

The constraints will be handled as:

$$\mathbf{Aw} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{w} \qquad \text{compared to} \qquad \mathbf{b} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

where the first line will be an equality and the last three will be inequalities.

```
library(quadprog)                      # Package for quadratic programming
Sigma <- t(E) %*% E                    # Unscaled covariance matrix
nb_mods <- nrow(Sigma)                 # Number of models
w_const <- solve.QP(Dmat = Sigma,      # D matrix =  Sigma
              dvec = rep(0, nb_mods),  # Zero vector
              Amat = rbind(rep(1, nb_mods), diag(nb_mods)) %>% t(), # A matrix for constraints
              bvec = c(1,rep(0, nb_mods)),                          # b vector for constraints
              meq = 1                  # One line of constraints is equality, others are inequalities
              )
w_const$solution %>% round(3)          # Solution
```

```
## [1] 0.000 0.000 0.929 0.000 0.071
```

Compared to the unconstrained solution, the weights ar sparse a concentrated on one or two models, usually those with small training sample errors.

12.2 Stacked ensembles

12.2.1 Two stage training

Stacked ensembles are a natural generalization of linear ensembles. The idea of generalizing linear ensembles goes back at least to Wolpert (1992). In the general case, the training is performed in two stages. The first stage is the simple one, whereby the M models are trained independently, yielding the predictions $\tilde{y}_{i,m}$ for instance i and model$m$. The second step is to consider the output of the trained models as input for a new level of machine learning optimization. The second level predictions are $\breve{y}_i = h(\tilde{y}_{i,1}, \dots, \tilde{y}_{i,M})$, where h is a new learner (see Figure 12.1). Linear ensembles are of course stacked ensembles in which the second layer is a linear regression.

The same techniques are then applied to minimize the error between the true values y_i and the predicted ones $\breve{y}_i$.

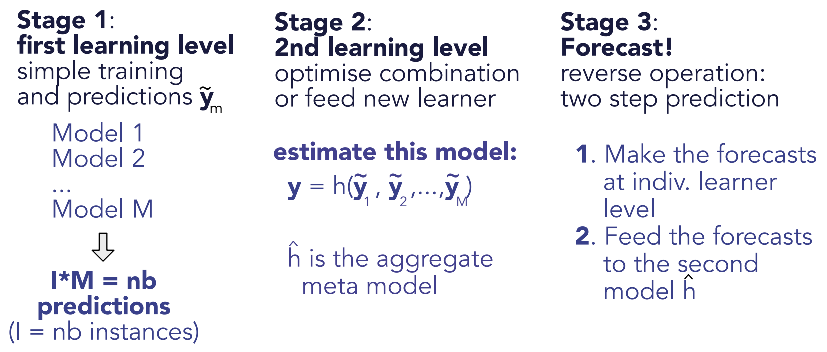

Stage 1: first learning level
simple training and predictions $\tilde{\mathbf{y}}_m$
Model 1
Model 2
...
Model M
I*M = nb predictions
(I = nb instances)
Stage 2: 2nd learning level
optimise combination or feed new learner
estimate this model:
$\mathbf{y} = h(\tilde{\mathbf{y}}_1, \tilde{\mathbf{y}}_2, \dots, \tilde{\mathbf{y}}_M)$
$\hat{h}$ is the aggregate meta model
Stage 3: Forecast!
reverse operation: two step prediction
1. Make the forecasts at indiv. learner level
2. Feed the forecasts to the second model $\hat{h}$

FIGURE 12.1: Scheme of Stacked Ensembles.

12.2.2 Code and results

Below, we create a low-dimensional neural network which takes in the individual predictions of each model and compiles them into a synthetic forecast.

```
model_stack <- keras_model_sequential()
model_stack %>%   # This defines the structure of the network, i.e. how layers are organized
    layer_dense(units = 8, activation = 'relu', input_shape = nb_mods) %>%
    layer_dense(units = 4, activation = 'tanh') %>%
    layer_dense(units = 1)
```

The configuration is very simple. We do not include any optional arguments and hence the model is likely to overfit. As we seek to predict returns, the loss function is the standard L^2 norm.

```
model_stack %>% compile(                        # Model specification
    loss = 'mean_squared_error',                # Loss function
    optimizer = optimizer_rmsprop(),            # Optimisation method (weight updating)
    metrics = c('mean_absolute_error')          # Output metric
)
summary(model_stack)                            # Model architecture
```

```
## ___________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## ===========================================================================
## dense_8 (Dense)                  (None, 8)                     48
## ___________________________________________________________________________
## dense_9 (Dense)                  (None, 4)                     36
## ___________________________________________________________________________
## dense_10 (Dense)                 (None, 1)                     5
## ===========================================================================
## Total params: 89
## Trainable params: 89
## Non-trainable params: 0
## ___________________________________________________________________________
```

```
y_tilde <- E + matrix(rep(training_sample$R1M_Usd, nb_mods), ncol = nb_mods)    # Training predictions
y_test <- E_test + matrix(rep(testing_sample$R1M_Usd, nb_mods), ncol = nb_mods) # Testing
fit_NN_stack <- model_stack %>% fit(y_tilde,                                  # Training features
                     training_sample$R1M_Usd,                                 # Training labels
```

```
                      epochs = 6, batch_size = 512,                          # Training parameters
                      validation_data = list(y_test,                         # Test data (features)
                                              testing_sample$R1M_Usd)        # Test data (labels)
)
plot(fit_NN_stack)                                                           # Plot, evidently!
```

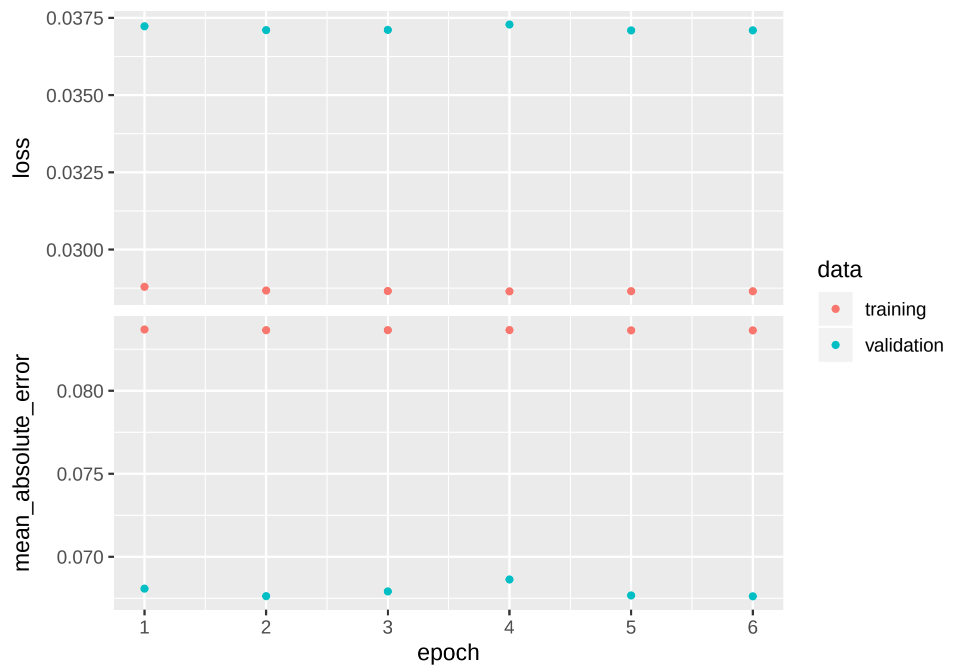


FIGURE 12.2: Training metrics for the ensemble model.

The performance of the ensemble is again disappointing. The training adds little value which means that the new overarching layer of ML does not enhance the original predictions. Again, this is because all ML engines seem to be capturing the same patterns and their both linear and non-linear combinations fail to improve their performance.

12.3 Extensions

12.3.1 Exogenous variables

In a financial context, macro-economic indicators could add value to the process. It is possible that some models perform better under certain conditions and exogenous predictors can help introduce a flavor of economic-driven conditionality in the predictions.

Adding macro variables to the set of predictors (here, predictions) $\tilde{y}_{i,m}$ could seem like one way to achieve this. However, this would amount to mix predicted values with (possibly scaled) economoc indicators and that would not make much sense.

One alternative outside the perimeter of ensembles is to train simple trees on a set of macroeconomic indicators. If the labels are the (possibly absolute) errors stemming from the original predictions, then the trees will create clusters of homogeneous error values. This will hint towards which conditions lead to the best and worst forecasts. We test this idea below, using aggregate data from the Federal Reserve of Saint Louis. A simple downloading function is available in the *quantmod* package. We download and format the data in the next chunk. CPIAUCSL is a code for consumer price index and T10Y2YM is a code for the term spread (10Y minus 2Y).

```
library(quantmod)                                      # Package that extracts the data
library(lubridate)                                     # Package for date management
getSymbols("CPIAUCSL", src = "FRED")                   # FRED is the Fed of St Louis
```

```
## [1] "CPIAUCSL"
```

```
getSymbols("T10Y2YM", src = "FRED")
```

```
## [1] "T10Y2YM"
```

```
cpi <- fortify(CPIAUCSL) %>%
    mutate (inflation = CPIAUCSL / lag(CPIAUCSL) - 1) # Inflation via Consumer Price Index
ts <- fortify(T10Y2YM)                                 # Term spread (10Y minus 2Y rates)
colnames(ts)[2] <- "termspread"                        # To make things clear
ens_data <- testing_sample %>%
    select(date) %>%
    cbind(err_NN_test) %>%
    mutate(Index = make_date(year = lubridate::year(date),       # Change date to first day of month
                             month = lubridate::month(date),
                             day = 1)) %>%
    left_join(cpi) %>%                                 # Add CPI to the dataset
    left_join(ts)                                      # Add termspread
head(ens_data)                                         # Show first lines
```

```
##         date  err_NN_test      Index CPIAUCSL   inflation termspread
## 1 2014-01-31 -0.150481847 2014-01-01  235.288 0.002424175       2.47
## 2 2014-02-28  0.077492399 2014-02-01  235.547 0.001100779       2.38
## 3 2014-03-31 -0.005263589 2014-03-01  236.028 0.002042055       2.32
## 4 2014-04-30 -0.067815196 2014-04-01  236.468 0.001864186       2.29
## 5 2014-05-31 -0.082144422 2014-05-01  236.918 0.001903006       2.17
## 6 2014-06-30  0.047112189 2014-06-01  237.231 0.001321132       2.15
```

We can now build a tree that tries to explain the accuracy of models as a function of macro variables.

```
library(rpart.plot)     # Load package for tree plotting
fit_ens <- rpart(abs(err_NN_test) ~ inflation + termspread, # Tree model
                 data = ens_data,
                 cp = 0.001)                                # Complexity parameter (size of tree)
rpart.plot(fit_ens)                                         # Plot tree
```

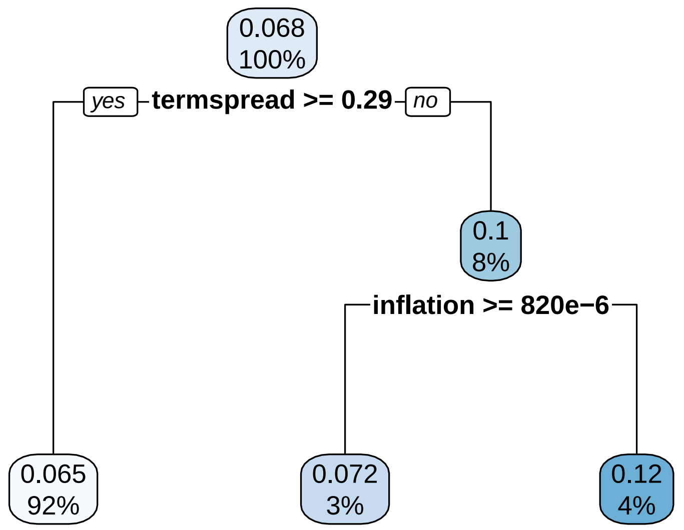


FIGURE 12.3: Conditional performance of an ML engine.
(#fig:r ensfred)

The tree creates clusters which have homogeneous values of absolute errors. One big cluster gathers 92% of predictions (the left one) and is the one with the smallest average. It corresponds to the periods when the term spread is above 0.29 (in percentage points). The other two groups (when the term spread is below 0.29%) are determined according to the level of inflation. If the latter is positive, then the average absolute error is 7%, if not, it is 12%. This last number, the highest of the three clusters, indicates that when the term spread is low and the inflation negative, the model's predictions are not trustworthy because their errors have a magnitude twice as large as in other periods. Under these circumstances (which seem to be linked to a dire economic environment), it may be wiser not to use ML-based forecasts.

12.3.2 Shrinking inter-model correlations

As shown earlier in this chapter, one major problem with ensembles arises when the first layer of predictions is highly correlated. In this case, ensemble are pretty much useless. Their are several tricks that can help reduce this correlation but the simplest and best is probably to alter training samples. If algorithms do not see the same data, they will probably infer different patterns.

There are several ways to split the training data so as to build different subsets of training samples. The first dichotomy is between random versus deterministic splits. Random splits are easy and require only the target sample size to be fixed. Note that the training samples can be overlapping as long as the overlap is not too large. Hence if the original training sample has I instance and the ensemble requires M models, then a subsample size of $\lfloor I/M \rfloor$ may be too conservative especially if the training sample is not very large. In this case $\lfloor I/\sqrt{M} \rfloor$ may be a better alternative. Random forests are one example of ensembles built in random training samples.

One advantage of deterministic splits is that they are easy to reproduce and their outcome does not depend on the random seed. By the nature of factor-based training samples, the second splitting dichotomy is between time and assets. A split within assets is straightforward: each model is trained on a different set of stocks. Note that the choices of sets can be random, or dictacted by some factor-based criterion: size, momentum, book-to-market ratio, etc.

A split in dates requires other decisions: is the data split in large blocks (like years) and each model gets a block, which may stand for one particular kind of market condition? Or are the training dates divided more regularly? For instance, if there are 12 models in the ensemble, each model can be trained on data from a given month (e.g., January for the first models, February for the second, etc.).

Below, we train four models on four different years to see if this help reduce the inter-model correlations. This process is a bit lengthy because the samples and models need to be all redefined. We start by creating the four training samples. The third model works on the small subset of features, hence the sample is smaller.

```
training_sample_2007 <- training_sample %>%
    filter(date > "2006-12-31", date < "2008-01-01")
training_sample_2009 <- training_sample %>%
    filter(date > "2008-12-31", date < "2010-01-01")
training_sample_2011 <- training_sample %>% select(c("date",features_short, "R1M_Usd")) %>%
    filter(date > "2010-12-31", date < "2012-01-01")
training_sample_2013 <- training_sample %>%
    filter(date > "2012-12-31", date < "2014-01-01")
```

Then, we proceed to the training of the models. The syntaxes are those used in the previous chapters, nothing new here. We start with a penalized regression. In all predictions below, the original testing sample is used.

```
y_ens_2007 <- training_sample_2007$R1M_Usd                                      # Dep. variable
x_ens_2007 <- training_sample_2007 %>%                                          # Predictors
    select(features) %>% as.matrix()
fit_ens_2007 <- glmnet(x_ens_2007, y_ens_2007, alpha = 0.1, lambda = 0.1)       # Model
err_ens_2007 <- predict(fit_ens_2007, x_penalized_test) - testing_sample$R1M_Usd # Prediction errors
```

We continue with a random forest.

```
fit_ens_2009 <- randomForest(formula,            # Same formula as for simple trees!
                 data = training_sample_2009,    # Data source: 2011 training sample
                 sampsize = 4000,                # Size of (random) sample for each tree
                 replace = FALSE,                # Is the sampling done with replacement?
                 nodesize = 100,                 # Minimum size of terminal cluster
                 ntree = 40,                     # Nb of random trees
                 mtry = 30                       # Nb of predictive variables for each tree
    )
err_ens_2009 <- predict(fit_ens_2009, testing_sample) - testing_sample$R1M_Usd # Prediction errors
```

The third model is a boosted tree.

```
train_features_2011 <- training_sample_2011 %>%
    select(features_short) %>% as.matrix()                         # Independent variable
train_label_2011 <- training_sample_2011 %>%
    select(R1M_Usd) %>% as.matrix()                          # Dependent variable
train_matrix_2011 <- xgb.DMatrix(data = train_features_2011,
                                 label = train_label_2011)        # XGB format!
fit_ens_2011 <- xgb.train(data = train_matrix_2011,          # Data source
              eta = 0.4,                                     # Learning rate
              objective = "reg:linear",                      # Objective function
```

```
             max_depth = 4,                                  # Maximum depth of trees
             nrounds = 18                                    # Number of trees used
    )
err_ens_2011 <- predict(fit_ens_2011, xgb_test) -  testing_sample$R1M_Usd # Prediction errors
```

Finally, the last model is a simple neural network.

```
NN_features_2013 <- select(training_sample_2013, features) %>% as.matrix() # Matrix format is important
NN_labels_2013 <- training_sample_2013$R1M_Usd
model_ens_2013 <- keras_model_sequential()
model_ens_2013 %>%   # This defines the structure of the network, i.e. how layers are organized
    layer_dense(units = 16, activation = 'relu', input_shape = ncol(NN_features_2013)) %>%
    layer_dense(units = 8, activation = 'tanh') %>%
    layer_dense(units = 1)
model_ens_2013 %>% compile(                    # Model specification
    loss = 'mean_squared_error',               # Loss function
    optimizer = optimizer_rmsprop(),           # Optimisation method (weight updating)
    metrics = c('mean_absolute_error')         # Output metric
)
model_ens_2013 %>% fit(NN_features_2013,                         # Training features
                       NN_labels_2013,                           # Training labels
                       epochs = 9, batch_size = 128              # Training parameters
)
err_ens_2013 <- predict(model_ens_2013, NN_test_features) - testing_sample$R1M_Usd
```

Endowed with the errors of the four models, we can compute their correlation matrix.

```
E_subtraining <- tibble(err_ens_2007,
                        err_ens_2009,
                        err_ens_2011,
                        err_ens_2013)
cor(E_subtraining)
```

```
##              err_ens_2007 err_ens_2009 err_ens_2011 err_ens_2013
## err_ens_2007    1.0000000    0.9399344    0.6415089    0.9988782
## err_ens_2009    0.9399344    1.0000000    0.6191353    0.9455117
## err_ens_2011    0.6415089    0.6191353    1.0000000    0.6422570
## err_ens_2013    0.9988782    0.9455117    0.6422570    1.0000000
```

The results are overall disappointing. Only one model manages to extract patterns that are somewhat different from the other ones, resulting in a 70% correlation across the board. One possible explanation could be that the models capture mainly noise and little signal. Working with long term labels like annual returns could help improve diversification across models.

12.4 Exercise

Create an ensemble with XXXXXX models trained on XXXXXX data.

13

Portfolio backtesting

In this section, we introduce the notations and framework that will be used when analysing and comparing investment strategies. Portfolio backtesting is often conceived and perceived as a quest to find the best strategy - or at least a solidly profitable one. When carried out thoroughly, this possibly long endeavour may entice the layman to confuse a fluke for a robust policy. Two papers published back-to-back warn against the perils of data snooping.

Fabozzi and de Prado (2018) acknowledge that only strategies that work make it to the public, while thousands (at least) have been tested. Picking the pleasing outlier (the only strategy that seemed to work) is likely to generate disappointment when switching to real life trading. In a similar vein, Arnott et al. (2019b) provide a list of principles and safeguards that any analyst should follow to avoid any type of error when backtesting strategy. The worst type is arguably false positives whereby strategies are found (often by cherrypicking) to outperform in one very particular setting, but will likely fail in live trading.

In addition to these recommendations on portfolio constructions, Arnott et al. (2019a) also warn against the perils of blindly investing in smart beta products related to academic factors. Plainly, expectations should not be set too high or face the risk of being disappointed. Another takeaway from their article is that economic cycles have a strong impact on factor returns: correlations change quickly and drawdowns can be magnified in times of major downturns.

Backtesting is more complicated than it seems and it is easy to make small mistakes that lead to *apparently* good portfolio policies. This chapter lays out a rigorous approach to this exercise, discusses a few caveats, and proposes a lengthy example.

13.1 Setting the protocol

We consider a dataset with three dimensions: time $t = 1, \dots, T$, assets $n = 1, \dots, N$ and characteristics $k = 1, \dots, K$. One of these attributes must be the price of asset n at time t, which we will denote $p_{t,n}$. From that, the computation of the arithmetic return is straightforward ($r_{t,n} = p_{t,n}/p_{t-1,n} - 1$) and so is any heuristic measure of profitability. For simplicity, we assume that time points are equidistant or uniform, i.e., that t is the index of a trading day or of a month for example. If each point in time t has data available for all assets, then this makes a dataset with $I = T \times N$ rows.

The dataset is first split in two: the out-of-sample period and the initial buffer period. The buffer period is required to train the models for the first portfolio composition. This period is determined by the size of the training sample. There are two options for this size: fixed (usually equal to 2 to 5 years) and expanding. In the first case, the training sample will roll over time, taking into account only the most recent data. In the second case, models are built on all of the available data. This last option can create problems because the first dates of the backtest are based on much smaller amounts of information compared to the last dates. Moreover, there is an ongoing debate on whether including the full history of returns and characteristics is advantageous or not. Proponents argue that this allows models to see many different market conditions. Opponents make the case that old data is by definition outdated and thus useless and possibly misleading.

Henceforth, we choose the rolling period option for the training sample, as depicted in Figure 13.1.

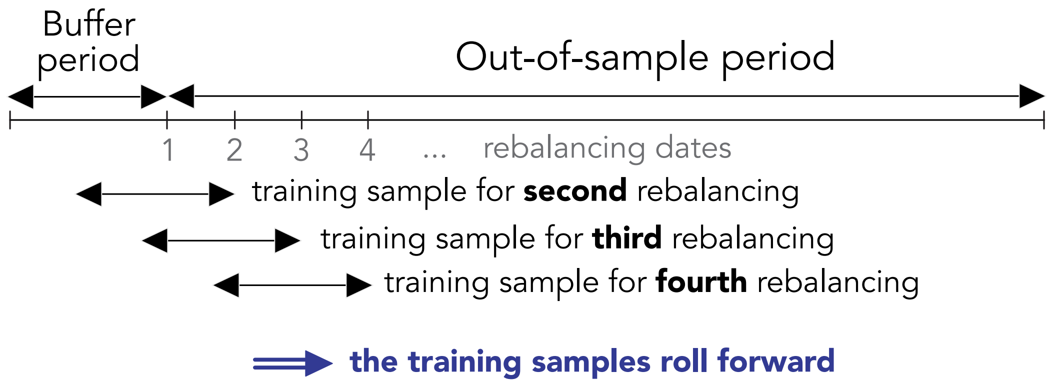

Buffer period
Out-of-sample period
1
2
3
4
...
rebalancing dates
training sample for **second** rebalancing
training sample for **third** rebalancing
training sample for **fourth** rebalancing
the training samples roll forward

FIGURE 13.1: Backtesting with rolling windows. The training set of the first period is simply the buffer period.

Two crucial design choices are the **rebalancing frequency** and the **horizon** at which the label is computed. It is not obvious that they should be equal but their choice should make sense. It can seem right to train on a 12 month forward label (which capture longer trends) and invest monthly or quarterly. However, it seems odd to do the opposite and train on short term movements (monthly) and invest at a long horizon.

These choices have a direct impact on how the backtest is carried out. If we note:

- Δ_h for the holding period between 2 rebalancing dates (in days or months);
- Δ_s for the size of the desired training sample (in days or months - not taking the number of assets into consideration);
- Δ_l for the horizon at which the label is computed (in days or months),

then the total length of the training sample should be $\Delta_s + \Delta_l$. Indeed, at any moment t, the training sample should stop at $t - \Delta_l$ so that the last point corresponds to a label that is calculated until time t. This is highlighted in Figure 13.2 in the form of the red danger zone. We call it the red zone because any feature which has a time index s inside the interval $(t-\Delta_l, t]$ will engender a forward looking bias. Indeed if a feature is indexed by $s \in (t-\Delta_l, t]$, then by definition, the label covers the period $[s, s+\Delta_l]$ with $s + \Delta_l > t$. At time t, this requires knowledge of the future and is naturally not realistic.

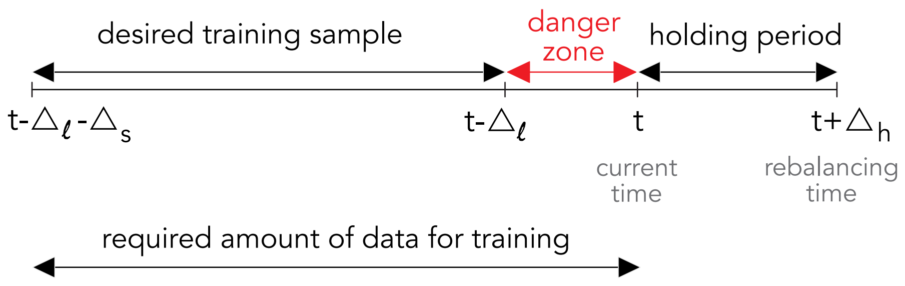


FIGURE 13.2: The tricks in rolling training samples.

13.2 Turning signals into portfolio weights

The predictive tools outlined in Chapters 6 to XXX are only meant to provide a signal that is expected to provide some information on the future profitability of assets. There are many ways that this signal can be integrated in an investment decision.

First and foremost, there are at least two steps in the portfolio construction process and the signal can be used at any of these stages. Relying on the signal for both steps puts a lot of emphasis on the predictions and should only be considered when the level of confidence in the forecasts is high.

The first step is **selection**. While a forecasting exercise can be carried out on a large number of assets, it is not compulsory to invest in all of these assets. In fact, it would make sense to take advantage of the signal to exclude those assets that are presumably likely to underperform in the future. Often, portfolio policies have fixed sizes that impose a constant number of assets. One heuristic way to exploit the signal is to select the assets that have the most favorable predictions and to discard the others. This naive idea is often used in the asset pricing literature: portfolios are formed according to underlying characteristics and this characteristic is deemed interesting if the portfolios exhibit very different profitabilities.

This is for instance an efficient way to test the relevance of the signal. If Q portfolios $q = 1, \dots, Q$ are formed according to the rankings of the assets with respect to the signal, then one would expect that the out-of-sample performance of the portfolios be monotonic with q. While a rigorous test of monotonicity would require to account for all portfolios (see, e.g., Romano and Wolf (2013)), it is often only assumed that the extreme portfolios suffice. If the difference between portfolio number 1 and portfolio number Q is substantial, then the signal is valuable. Whenever the investor is able to short assets, this amounts to a dollar neutral strategy.

The second step is **weighting**. If the selection process relied on the signal, then a simple weighting scheme is often a good idea. Equally-weighted portfolios are known to be hard to beat (see DeMiguel et al. (2007)), especially compared to their cap-weighted alternative, as is shown in Plyakha et al. (2014). More advanced schemes include equal risk contributions (Maillard et al. (2010)) and constrained minimum variance (Coqueret (2015)). Both only rely on the covariance matrix of the assets and not on any proxy for the vector of expected returns.

For the sake of completeness, we explicit a generalization of Coqueret (2015) which is a generic constrained quadratic program:

$$\underset{\mathbf{w}}{\min}\ \frac{\lambda}{2}\mathbf{w}'\mathbf{\Sigma}\mathbf{w}-\mathbf{w}'\boldsymbol{\mu}, \quad \text{s.t.} \quad \begin{array}{l} \mathbf{w}'\mathbf{1}=1, \\ (\mathbf{w}-\mathbf{w}_-)'\mathbf{\Lambda}(\mathbf{w}-\mathbf{w}_-)\le \delta_R, \\ \mathbf{w}'\mathbf{w}\le \delta_D, \end{array} \tag{13.1}$$

where it is easy to recognize the usual mean-variance optimization in the left-hand side. We impose three constraints on the right-hand side. The first one is the budget constraint (weights sum to one). The second one penalizes variations in weights (compared to the current allocation) via a diagonal matrix $\mathbf{\Lambda}$. This is a crucial point. Portfolios are rarely constructed from scratch and are most of the time adjustments from existing positions. In order to reduce the orders and the corresponding transaction costs, it is possible to penalize large variations from the existing portfolio. In the above program, the current weights are written $\mathbf{w}_-$ and the desired ones $\mathbf{w}$ so that $\mathbf{w}-\mathbf{w}_-$ is the vector of deviations from the current positions. The term $(\mathbf{w}-\mathbf{w}_-)\mathbf{\Lambda}(\mathbf{w}-\mathbf{w}_-)$ is an expression that characterizes the sum of squared deviations, weighted by the diagonal coefficients $\Lambda_{n,n}$. This can be helpful because some assets may be more costly to trade due to liquidity (large cap stocks are more liquid and their trading costs are lower). When δ_R decreases, the rotation is reduce because weights are not allowed to deviate to much from $\mathbf{w}_-$. The last constraint enforces diversification via the Herfindhal index of the portfolio: the smaller δ_D, the more diversified the portfolio.

Recalling that there are N assets in the universe, the Lagrange form of (13.1) is:

$$L(\mathbf{w})=\frac{\lambda}{2}\mathbf{w}'\mathbf{\Sigma}\mathbf{w}-\mathbf{w}'\boldsymbol{\mu}-\eta(\mathbf{w}'\mathbf{1}_N-1)+\kappa_R((\mathbf{w}-\mathbf{w}_-)\mathbf{\Lambda}(\mathbf{w}-\mathbf{w}_-)-\delta_R)+\kappa_D(\mathbf{w}'\mathbf{w}-\delta_D), \tag{13.2}$$

and the first order condition

$$\frac{\partial}{\partial \mathbf{w}}L(\mathbf{w})=\lambda\mathbf{\Sigma}\mathbf{w}-\boldsymbol{\mu}-\eta\mathbf{1}_N+2\kappa_R\mathbf{\Lambda}(\mathbf{w}-\mathbf{w}_-)+2\kappa_D\mathbf{w}=0,$$

yields

$$\mathbf{w}^*_\kappa=(\lambda\mathbf{\Sigma}+2\kappa_R\mathbf{\Lambda}+2\kappa_D\mathbf{I}_N)^{-1}\left(\boldsymbol{\mu}+\eta_{\lambda,\kappa_R,\kappa_D}\mathbf{1}_N+2\kappa_R\mathbf{\Lambda}\mathbf{w}_-\right), \tag{13.3}$$

with

$$\eta_{\lambda,\kappa_R,\kappa_D}=\frac{1-\mathbf{1}_N'(\lambda\mathbf{\Sigma}+2\kappa_R\mathbf{\Lambda}+2\kappa_D\mathbf{I}_N)^{-1}(\boldsymbol{\mu}+2\kappa_R\mathbf{\Lambda}\mathbf{w}_-)}{\mathbf{1}_N'(\lambda\mathbf{\Sigma}+2\kappa_R\mathbf{\Lambda}+2\kappa_D\mathbf{I}_N)^{-1}\mathbf{1}_N}.$$

This parameter ensures that the budget constraint is satisfied. The optimal weights in (13.3) depend on three tuning parameters: λ, κ_R and κ_D. - When λ is large, the focus is set more on risk reduction than one profit maximization (which is often a good idea given that risk is easier to predict);
- When κ_R is large, the importance of transaction costs in (13.2) is high and thus, in the limit when $\kappa_R\to\infty$, the optimal weights are equal to the old ones $\mathbf{w}_-$ (for finite values of the other parameters).
- When κ_D is large, the portfolio is more diversified and (all other things equal) when $\kappa_D\to\infty$, the weights are all equal (to $1/N$).
- When $\kappa_R=\kappa_D=0$, we recover the classical mean-variance weights that are proportional to $(\mathbf{\Sigma})^{-1}\boldsymbol{\mu}$.

This seemingly complex formula is in fact very flexible and tractable. It requires some tests and adjustments before finding realistic values for λ, κ_R and κ_D.

13.3 Performance metrics

The evaluation of performance is a key stage in a backtest. This section, while not exhaustive, is intended to cover the most important facets of portfolio assessment.

13.3.1 Discussion

While the evaluation of the accuracy of ML tools (See Section 11.1) is of course valuable, the portfolio returns are the ultimate yardstick during a backtest. One essential element in such an exercise is a **benchmark** because raw and absolute metrics don't mean much one their own.

This is not only true at the portfolio level, but also at the ML engine level. In most of the trials of the previous chapters, the MSE of the models on the testing set revolves around 0.037. An interesting figure is the variance of one month returns on this set, which corresponds to the error made by a constant prediction of 0 all the time. This figure is equal to 0.037, which means that the sophisticated algorithm don't really improve on a naive heuristic. This benchmark is the one used in the out-of-sample R^2 of Gu et al. (2018).

In portfolio choice, the most elementary allocation is the uniform one, whereby each asset receive the same weight. This seemingly simplistic solution is in fact an incredible benchmark, one that is hard to beat consistently (see DeMiguel et al. (2007) and Plyakha et al. (2014)). Below, we will pick this equally-weighted portfolio as the benchmark.

13.3.2 Pure performance and risk indicators

We then turn to the definition of the usual metrics used both by practitioners and academics alike. Henceforth, we write $r^P=(r_t^P)_{1\le t\le T}$ and $r^B=(r_t^B)_{1\le t\le T}$ for the returns of the portfolio and those of the benchmark, respectively. When refering to some generic returns, we simply write r_t. There are many ways to analyze them and most of them rely on their distribution.

The simplest indicator is the average return:

$$\mu_P=\mathbb{E}[r^P]\approx\frac{1}{T}\sum_{t=1}^T r_t^P, \quad \mu_B=\mathbb{E}[r^B]\approx\frac{1}{T}\sum_{t=1}^T r_t^B,$$

where, obviously, the portfolio is noteworthy if $\mathbb{E}[r^P]>\mathbb{E}[r^B]$. Note that we use the arithmetic average above but the geometric one is also an option, in which case:

$$\tilde{\mu}_P\approx\left(\prod_{t=1}^T(1+r_t^P)\right)^{1/T}-1, \quad \tilde{\mu}_B=\approx\left(\prod_{t=1}^T(1+r_t^B)\right)^{1/T}-1.$$

The benefit of this second definition is that it takes the compounding of returns into account and hence compensates for volatility pumping. To see this, consider a very simple two period model with returns $-r$ and $+r$. The arithmetic average is zero, but the geometric one $\sqrt{(1-r^2)}-1<0$ is negative.

Hit ratios are often referred to when the portfolio is built on a *good guess*. This can be evaluated at the asset level (the proportion of positions in the correct direction[1]) or a the portfolio level. In all cases, the computation can be performed on raw returns or on relative returns (e.g., compared to a benchmark). A meaningful hit ratio is the proportion of times that a strategy beats its benchmark. This is of course not sufficient, as many small wins can be offset by a few large losses.

Pure performance measures are almost always accompanied by risk measures. The second moment of returns is usually used to quantify the magnitude of fluctuations of the portfolio. A large variance implies sizable movements in returns, and hence in portfolio values. This is why the standard deviation is called the volatility of the portfolio.

$$\sigma^2_P=\mathbb{V}[r^P]\approx \frac{1}{T-1}\sum_{t=1}^T(r_t^P-\mu_P)^2, \quad \sigma^2_B=\mathbb{V}[r^B]\approx \frac{1}{T-1}\sum_{t=1}^T(r_t^B-\mu_B)^2.$$

In this case, the portfolio can be preferred if it is less risky compared to the benchmark, i.e., when $\sigma_P^2<\sigma_B^2$.

Higher order moments of returns are sometimes used (skewness and kurtosis), but they are far less common. We refer for instance to Harvey et al. (2010) for one method that takes them into account in the portfolio construction process.

For some people, the volatility is an incomplete measure of risk. It can be argued that it should be decomposed into 'good' volatility (when prices go up) versus 'bad' volatility when they go down. The downward semi-variance is computed as the variance taken over the negative returns:

$$\sigma^2_-=\mathbb{V}[r]\approx \frac{1}{T-1}\sum_{t=1}^T(r_t-\mu_P)^2 1_{\{r_t<0\}}.$$

The average return and the volatility are the typical moment-based metrics used by practitioners. Other indicators rely on different aspects of the distribution of returns with a focus on tails and extreme events. The Value-at-Risk (VaR) is one such example. If F_r is the empirical cdf of returns, the VaR at a level of confidence α (often taken to be 95%) is

$$\text{VaR}_\alpha(\mathbf{r}_t)=F_r(1-\alpha).$$

It is equal to the worst case scenario if we omit the worst $1-\alpha\%$ returns. An even more conservative measure is the so-called Conditional Value at Risk (CVaR), also known as expected shortfall, which computes the average loss in the worst $\alpha\%$ scenarios. Its empirical evaluation is

$$\frac{1}{\text{Card}(r_t<\text{VaR}_\alpha(\text{r}_t))}\sum_{r_t<\text{VaR}_\alpha(\text{r}_t)}r_t.$$

Going crescendo in the severity of risk measures, the ultimate evaluation of loss is the maximum drawdown. It is equal to the maximum loss suffered from the peak value of the strategy If we write P_t for the time-t value of a portfolio, the drawdown is

$$D_T^P=\underset{0\le t\le T}{\max} P_t-P_T,$$

[1]A long position in an asset with positive return or a short position in an asset with negative return.

and the maximum drawdown is

$$MD_T^P = \max_{0\le s\le T}\left(\max_{0\le t\le s} P_t - P_s, 0\right).$$

This quantity evaluates the greatest loss over the time frame $[0, T]$ and is thus the most conservative risk measure of all.

13.3.3 Factor-based evaluation

In the spirit of factor models, performance can also be assessed through the lens of exposures. If we recall the original formulation from Equation (4.1):

$$r_{t,n} = \alpha_n + \sum_{k=1}^{K} \beta_{t,k,n} f_{t,k} + \epsilon_{t,n},$$

then the estimated $\hat{\alpha}_n$ is the performance that cannot be explained by the other factors. When returns are *excess* returns (over the risk-free rate) and when there is only one factor, the market factor, then this quantity is called Jensen's alpha (Jensen (1968)). Often, it is simply referred to as *alpha*. The other estimate, $\hat{\beta}_{t,M,n}$ (M for market), is the market beta.

Because of the rise of factor investing, it has become customary to also report the alpha of more exhaustive regressions. Adding the size and value premium (as in Fama and French (1993)) and even momentum (Carhart (1997)) helps understand if a strategy generates value beyond that that can be obtained through the usual factors.

13.3.4 Risk-adjusted measures

Now, the tradeoff between the average return and the volatility is a cornerstone in modern finance, since Markowitz (1952). The simplest way to synthesize both metrics is via the **information ratio**:

$$IR(P, B) = \frac{\mu_{P-B}}{\sigma_{P-B}},$$

where the index $P - B$ implies that the mean and standard deviations are computed on the long-short portfolio with returns $r_t^P - r_t^B$. The denominator σ_{P-B} is sometimes called the **tracking error**.

The most widespread information ratio is the Sharpe ratio (Sharpe (1966)) for which the benchmark is some riskless asset. Instead of directly computing the information ratio between two portfolios or strategies, it is often customary to compare their Sharpe ratios. Simple comparisons can benefit from statistical tests (see e.g., Ledoit and Wolf (2008)).

More extreme risk measures can serve as denominator in risk-adjusted indicators. The Managed Account Report ratio (MAR ratio) is for example computed as

$$MAR^P = \frac{\tilde{\mu}_P}{MD^P},$$

while the Treynor ratio is equal to

$$\text{Treynor} = \frac{\mu_P}{\hat{\beta}_M},$$

i.e., the (excess) return divided by the market beta (see Treynor (1965)). This definition was generalized to multifactor expositions by Hübner (2005) into the generalized Treynor ratio:

$$\text{GT} = \mu_P \frac{\sum_{k=1}^K \bar{f}_k}{\sum_{k=1}^K \hat{\beta}_k \bar{f}_k},$$

where the $\bar{f}_k$ are the sample average of the factors $f_{t,k}$. We refer to the original article for a detailed account of the analytical properties of this ratio.

13.3.5 Transaction costs and turnover

Updating portfolio composition is not free. In all generality, the total cost of one rebalancing at time t is proportional to $C_t = \sum_{n=1}^N |\Delta w_{t,n}| c_{t,n}$, where $\Delta w_{t,n}$ is the change in position for asset n and $c_{t,n}$ the corresponding fee. This last quantity is often hard to predict, thus it is customary to use a proxy that depends for instance on market capitalization (large stocks have more liquid shares and thus require smaller fees) or bid-ask spreads (smaller spreads mean smaller fees).

As a first order approximation, it is often useful to compute the average turnover:

$$\text{Turnover} = \frac{1}{T-1} \sum_{t=2}^T \sum_{n=1}^N |w_{t,n} - w_{t-,n}|,$$

where $w_{t,n}$ are the desired t-time weights in the portfolio and $w_{t-,n}$ are the weights just before the rebalancing. The positions of the first period (lauching weights) are exluded from the computation by convention. Transaction costs can then be proxied as a multiple of turnover (times some average or median cost in the cross-section of firms). This is a first order estimate of realized costs that does not take into consideration the evolution of the scale of the portfolio. Nonetheless, a rough figure is much better than none at all.

Once transaction costs (TC) have been annualized, they can be deducted from average returns to yield a more realistic picture of profitability. In the same vein, the transaction cost-adjusted Sharpe ratio of a portfolio P is given by

$$SR_{TC} = \frac{\mu_P - TC}{\sigma_P}. \tag{13.4}$$

Transaction costs are often overlooked in academic articles but can have a sizable impact in real life trading (see e.g., Novy-Marx and Velikov (2015)). Martin Utrera et al. (2018) show how to use factor investing (and exposures) to combine and offset positions and reduce overall fees.

13.4 Common errors and issues

13.4.1 Forward looking data

One of the most common mistakes in portfolio backtesting is to use forward looking data. It is for instance easy to fall in the trap of the danger zone depicted in Figure 13.2. In this case, the labels used at time t are computed with knowledge of what happens at times $t+1$, $t+2$, etc. It is worth triple checking every step in the code to make sure that strategies are not built on prescient data.

13.4.2 Backtest overfitting

The second major problem is backtest overfitting. The analogy with training set overfitting is easy to grasp. It is a well known issue and was formalized for instance in White (2000) and Romano and Wolf (2005). In portfolio choice, we refer to Bajgrowicz and Scaillet (2012) and Bailey and de Prado (2014) and the references therein.

At any given moment, a backtest depends on one particular dataset. Often, the result of the first backtest will not be satisfactory - for many possible reasons. Hence, it is tempting to have another try, when altering some parameters that were probably not optimal. This second test may be better, but not quite good enough - yet. Thus, a in a third trial, a new weighting scheme can be tested, along with a new forecasting engine (more sophisticated). Iteratively, the backtester can only end up with a strategy that performs well enough, it is just a matter of time and trials.

One consequence of backtest overfitting is that it is illusory to hope for the same Sharpe ratios in live trading as those obtained in the backtest. Reasonable professionals divide the Sharpe ratio by two at least (Harvey and Liu (2015), Suhonen et al. (2017)). In Bailey and de Prado (2014), the authors even propose a statistical test for Sharpe ratios, provided that all metrics of all tested strategies are stored in memory. The formula is:

$$t=\phi\left((SR-SR^*)\sqrt{\frac{T-1}{1-\gamma_3SR+\frac{\gamma_4-1}{4}SR^2}}\right), \tag{13.5}$$

where SR is the Sharpe Ratio obtained by the best strategy, and

$$SR^*=\mathbb{E}[SR]+\sqrt{\mathbb{V}[SR]}\left((1-\gamma)\phi^{-1}\left(1-\frac{1}{N}\right)+\gamma\phi^{-1}\left(1-\frac{1}{Ne}\right)\right),$$

is the theoretical average maximum SR. Moreover,

- T is the number of trading dates;
- γ_3 and γ_4 are the *skewness* and *kurtosis* of the returns of the chosen strategy;
- ϕ is the cdf of the standard Gaussian law and γ is the Euler-Mascheroni constant;
- N refers to the number of strategy trials.

If t defined above is below a certain threshold (e.g., 0.95), then the SR cannot be deemed significant: **the best strategy is not outstanding**. Most of the time, sadly, that is the case. In Equation (13.5), the realized SR must be above the theoretical maximum SR^* and the scaling factor must be sufficiently large to push the argument inside ϕ close enough to two.

In the scientific community, test overfitting is also known as p-hacking. It is rather common in financial economics and the reading of Harvey (2017) is strongly advised to grasp the magnitude of the phenomenon. p-hacking is also present in most fields that use statistical tests (see, e.g., Head et al. (2015) to cite but one reference). There are several ways to cope with p-hacking:

1. don't rely on p-values (Amrhein et al. (2019));
2. use detection tools (Elliott et al. (2019));
3. or, finally, use advanced methods that process arrays of statistics (e.g., the Bayesianized versions of p-values to include some prior assessment from Harvey (2017), or other tests such as those proposed in Romano and Wolf (2005) and Simonsohn et al. (2014)).

The first option is wise, but the drawback is that the decision process is then left to another arbitrary yardstick.

13.4.3 Simple saveguards

As is mentioned at the beginning of the chapter, two common sense references for backtesting are Fabozzi and de Prado (2018) and Arnott et al. (2019b). The pieces of advice provide in these two articles are often judicious and thoughtful.

One additional comment pertains to the output of the backtest. One simple, intuitive and widespread metric is the transaction cost-adjusted Sharpe ratio defined in Equation (13.4). In the backtest, let us call SR_{TC}^B the corresponding value for the benchmark, which we like to define as the equally-weighted portfolio of all assets in the trading universe (in our dataset, roughly one thousand US equities). If the SR_{TC}^P of the best strategy is above $2 \times SR_{TC}^B$, then there is probably a glitch somewhere in the backtest.

This criterion holds under two assumptions: 1) a sufficiently long enough out-of-sample period and 2) long-only portfolios. It is unlikely that any realistic strategy can outperform a solid benchmark by a very wide margin over the long term. Being able to improve the benchmark's annualized return by 150 basis points (with comparable volatility) is already a great achievement. Backtests that deliver returns 5% above those of the benchmark are dubious.

13.5 Implication of non-stationarity: forecasting is hard

This subsection is split into two parts: in the first, we discuss the reason that make forecasting such a difficult task and in the second we present an important theoretical result originally developed towards machine learning but that sheds light on any discipline confronted with out-of-sample tests.

13.5.1 General comments

The careful reader must have noticed that throughout Chapters 6 to 12, the performance of ML engines is underwhelming. These disappointing results are there on purpose and highlight the crucial truth that machine learning is no panacea, no magic wand, no philosopher's stone that can transform data into golden predictions. Most ML-based forecasts fail. This is in fact not only true for very enhanced and sophisticated techniques, but also for simpler econometric approaches (Dichtl et al. (2019)), which again underlines the need to replicate results to challenge their validity.

One reason for that is that datasets are full of noise and extracting the slightest amount of signal is a tough challenge (we recommend a careful reading of the introduction of Timmermann (2018) for more details on this topic). One rationale for that is the ever time-varying nature of factor analysis in the equity space. Some factors can perform very well during one year and then poorly the next year and these reversals can be costly in the context of fully automated data-based allocation processes.

In fact, this is one major difference with many fields for which ML has made huge advances. In image recognition, numbers will always have the same shape, and so will cats, buses, etc. Likewise, a verb will always be a verb and syntaxes in languages do not change. This invariance, though sometimes hard to grasp[2] is nonetheless key to the great improvement both in computer vision and natural language processing.

In factor investing, there does not seem to be such invariance. There is no factor and no (possibly nonlinear) combination of factors that can explain and accurately forecast returns over decades long periods.[3] The academic literature has yet to find such a model; but even if it did, a simple arbitrage reasoning would invalidate its conclusions in future datasets.

One key ingredient in the backtest of a dynamic strategy is the depth of the sample size, i.e., how far back in time do each training sets go. There should be some coherence between: - the trading horizon determined by the rebalancing frequency;
- the training sample depth (number of years or months in the training set);
- the time window on which the label is computed.

There is probably no perfect combination, but there are some odd choices. Short-term labels and long trading horizons will plausibly not match very well.

Online learning & Active learning XXXXXXXXXXX Settles (2009)

13.5.2 The no free lunch theorem

13.6 Example

We finally propose a full detailed example of one implementation of a ML-based strategy run on a careful backtest. What follows is a generalization of the content of Section 6.2.2. In the same spirit, we split the backtest in four parts:

1. the creation/initialization of variables;
2. the definition of the strategies in one main functions;
3. the backtesting loop itself;
4. the performance indicators.

[2]We invite the reader to have a look at the thoughtful albeit theoretical paper by Arjovsky et al. (2019).
[3]In the thread `https://twitter.com/fchollet/status/1177633367472259072`, François Chollet, the creator of Keras argues that ML predictions based on price data cannot be profitable on the long term. Given the wide access to financial data, it is likely that the statement holds for predictions stemming from factor-related data as well.

Accordingly, we start with initializations.

```
sep_oos <- as.Date("2007-01-01")                        # Starting point for backtest
ticks <- data_ml$stock_id %>%                           # List of all asset ids
    as.factor() %>%
    levels()
N <- length(ticks)                                      # Max number of assets
t_oos <- returns$date[returns$date > sep_oos] %>%       # Out-of-sample dates
    unique() %>%                                        # Remove duplicates
    as.Date(origin = "1970-01-01")                      # Transform in date format
Tt <- length(t_oos)                                     # Nb of dates, avoid T = TRUE
nb_port <- 2                                            # Nb of portfolios/stragegies
portf_weights <- array(0, dim = c(Tt, nb_port, N))      # Initialize portfolio weights
portf_returns <- matrix(0, nrow = Tt, ncol = nb_port)   # Initialize portfolio returns
```

This first step is crucial, it lays the groundwork for the core of the backtest. We consider only two strategies: one ML-based and the EW (1/N) benchmark. The main (weighting) function will consist of these two components, but we define the sophisticated one in a dedicated wrapper. The ML-based weights are derived from XGBoost predictions with 80 trees, a learning rate of 0.3 and a maximum tree depth of 4. This makes the model complex but not exceedingly so. Once the predictions are obtained, the weighting scheme is simple: it is an EW portfolio over the best half of the stocks (those with above median prediction).

In the function below, all parameters (e.g., the learning rate) are hard-coded. They can easily be passed in arguments next to the data inputs. One very important detail is that in contrast to the rest of the book, the label is the 12 month future return. The main reason for this is rooted in the discussion from Section 5.6.

```
weights_xgb <- function(train_data, test_data, features){
    train_features <- train_data %>% select(features) %>% as.matrix()       # Indep. variable
    train_label <- train_data$R12M_Usd / exp(train_data$Vol1Y_Usd)          # Dependent variable
    train_matrix <- xgb.DMatrix(data = train_features, label = train_label)  # XGB format
    fit <- train_matrix %>%
        xgb.train(data = .,                         # Data source (pipe input)
                  eta = 0.3,                        # Learning rate
                  objective = "reg:linear",         # Number of random trees
                  max_depth = 4,                    # Maximum depth of trees
                  nrounds = 80                      # Number of trees used
        )
    xgb_test <- test_data %>%                       # Test sample => XGB format
        select(features) %>%
        as.matrix() %>%
        xgb.DMatrix()

    pred <- predict(fit, xgb_test)                  # Single prediction
    w <- pred > median(pred)                        # Keep only the 50% best predictions
    w$weights <- w / sum(w)
    w$names <- unique(test_data$stock_id)
    return(w)                                       # Best predictions, equally-weighted
}
```

Compared to the structure proposed in Section 7.4.6, the differences are that the label is not only based on *long-term* returns, but it also relies on a volatility component. Even though the denominator in the label is the exponential a quantile of the volatility, it seems fair to say that it is inspired by the Sharpe ratio and that the model seeks to explain and forecast a risk-adjusted return instead of a *raw* return. A stock with very low volatility will have its return unchanged in the label, while a stock with very high volatility will see its return divided by a factor close to three (exp(1)=2.718).

This function is then embedded in the global weighting function which only wraps two schemes: the EW benchmark and the ML-based policy.

```
portf_compo <- function(train_data, test_data, features, j){
    if(j == 1){                                 # This is the benchmark
        N <- test_data$stock_id %>%             # Test data dictates allocation
            factor() %>% nlevels()
        w <- 1/N                                # EW portfolio
        w$weights <- rep(w,N)
        w$names <- unique(test_data$stock_id)   # Asset names
        return(w)
    }
    if(j == 2){                                 # This is the ML strategy.
        return(weights_xgb(train_data, test_data, features))
    }
}
```

Equipped with this function, we can turn to the main backtesting loop. Given the fact that we use a large scale model, the computation time for the loop is large (possibly a few hours on a slow machine).

```
m_offset <- 12                                          # Offset in months for buffer period
train_size <- 5                                         # Size of training set in years
for(t in 1:(length(t_oos)-1)){                          # Stop before the last date: no fwd return!
    if(t%%12==0){print(t_oos[t])}                       # Just checking the date status
    train_data <- data_ml %>% filter(date < t_oos[t] - m_offset * 30,   # Rolling window with buffer
                                    date > t_oos[t] - m_offset * 30 - 365 * train_size)
    test_data <- data_ml %>% filter(date == t_oos[t],                   # Current values
                                    stock_id %in% train_data$stock_id)  # Common with past info!
    realized_returns <- test_data %>%                   # Computing returns via:
        select(R1M_Usd)                                 # 1M holding holding period!
    realized_names <- test_data %>%                     # Names of assets with holding period returns
        select(stock_id) %>%
        as.matrix()
    for(j in 1:nb_port){
        temp_weights <- portf_compo(train_data, test_data, features, j) # Hard-coded params, beware!
        ind1 <- match(temp_weights$names, realized_names) %>% na.omit() # Index between train & test
        ind2 <- match(ticks, realized_names) %>% na.omit()              # Index between all & test
        portf_weights[t,j,ind2] <- temp_weights$weights[ind1]           # Locating the weights correctly
        portf_returns[t,j] <- sum(temp_weights$weights[ind1] * realized_returns) # Compute returns
    }
}
```

```
## [1] "2007-12-31"
## [1] "2008-12-31"
## [1] "2009-12-31"
## [1] "2010-12-31"
## [1] "2011-12-31"
## [1] "2012-12-31"
## [1] "2013-12-31"
## [1] "2014-12-31"
## [1] "2015-12-31"
## [1] "2016-12-31"
## [1] "2017-12-31"
```

There are two important comments to be made on the above code. The first comment pertains to the two parameters that are defined in the first lines. They refer to the size of the training sample (5 years) and the length of the buffer period shown in Figure 13.2. This **buffer period is imperative** because the label is based on a long-term (12 month) return. This lag is compulsory to avoid any forward looking bias in the backtest.

Below, we create a function that compute the turnover (variation in weights). It requires both the weight values as well as the returns of all assets because the weights just before a rebalancing depend on the weights assigned in the previous period as well as on the returns of the assets that have altered these original weights.

```
turnover <- function(weights, asset_returns, t_oos){
    turn <- 0
    for(t in 2:length(t_oos)){
        realised_returns <- returns %>% filter(date == t_oos[t]) %>% select(-date)
        prior_weights <- weights[t-1,] * (1 + realised_returns) # Before rebalancing
        turn <- turn + apply(abs(weights[t,] - prior_weights/sum(prior_weights)),1,sum)
    }
    return(turn/(length(t_oos)-1))
}
```

Once turnover is defined, we embed it into a function that computes several key indicators.

```
perf_met <- function(portf_returns, weights, asset_returns, t_oos){
    avg_ret <- mean(portf_returns, na.rm = T)                     # Arithmetic mean
    vol <- sd(portf_returns, na.rm = T)                           # Volatility
    Sharpe_ratio <- avg_ret / vol                                 # Sharpe ratio
    VaR_5 <- quantile(portf_returns, 0.05)                        # Value-at-risk
    turn <- 0                                                     # Initialisation of turnover
    for(t in 2:dim(weights)[1]){
        realized_returns <- asset_returns %>% filter(date == t_oos[t]) %>% select(-date)
        prior_weights <- weights[t-1,] * (1 + realized_returns)
        turn <- turn + apply(abs(weights[t,] - prior_weights/sum(prior_weights)),1,sum)
    }
    turn <- turn/(length(t_oos)-1)                                # Average over time
    met <- data.frame(avg_ret, vol, Sharpe_ratio, VaR_5, turn)    # Aggregation of all of this
    rownames(met) <- "metrics"
    return(met)
}
```

Lastly, we build a function that loops on the various strategies.

```
perf_met_multi <- function(portf_returns, weights, asset_returns, t_oos, strat_name){
    J <- dim(weights)[2]              # Number of strategies
    met <- c()                        # Initialization of metrics
    for(j in 1:J){                    # One very ugly loop
        temp_met <- perf_met(portf_returns[, j], weights[, j, ], asset_returns, t_oos)
        met <- rbind(met, temp_met)
    }
    row.names(met) <- strat_name      # Stores the name of the strat
    return(met)
}
```

Given the weights and returns of the portfolios, it remains to compute the returns of the assets to plug them in the aggregate metrics function.

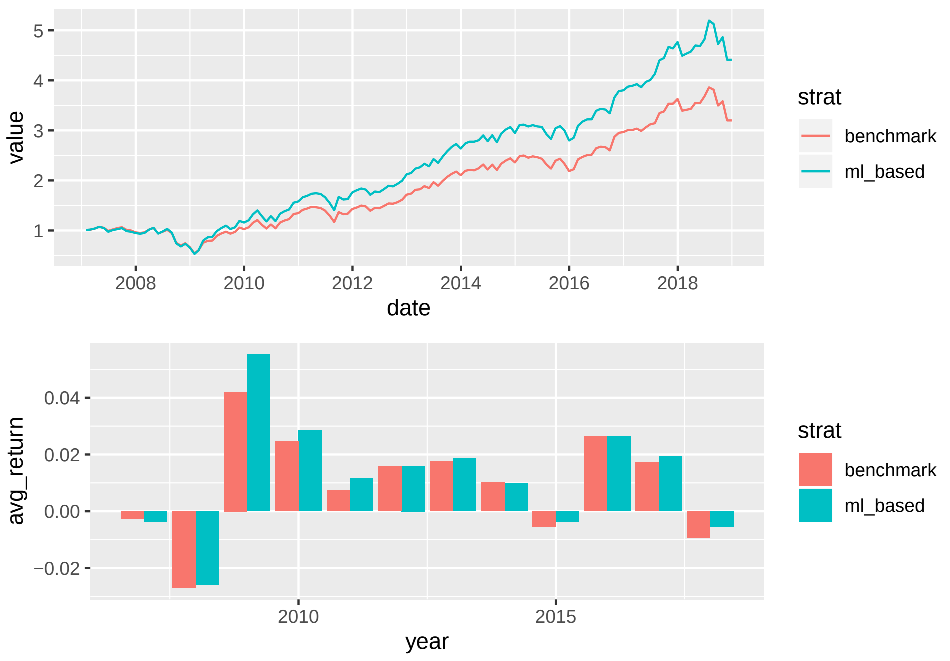
```
asset_returns <- data_ml %>%                          # Compute return matrix: start from original data
    select(date, stock_id, R1M_Usd) %>%               # Keep 3 attributes
    spread(key = stock_id, value = R1M_Usd)           # Shape in matrix format
asset_returns[is.na(asset_returns)] <- 0              # Zero returns for missing points

met <- perf_met_multi(portf_returns = portf_returns,  # Computes performance metrics
                      weights = portf_weights,
                      asset_returns = asset_returns,
                      t_oos = t_oos,
                    strat_name = c("EW", "XGB_SR"))
met                                                       # Displays perf metrics
```

```
##             avg_ret        vol Sharpe_ratio       VaR_5      turn
## EW      0.009699314 0.05643112    0.1718788 -0.07712509 0.0711028
## XGB_SR 0.012271508 0.06232806    0.1968858 -0.08236267 0.6589811
```

The ML-based strategy performs finally well! The gain is mostly obtained by the average return, while the volatility is higher than that of the benchmark. The net effect is that the Sharpe ratio is improved compared to the benchmark. The augmentation is not breathtaking, but (hence?) it seems reasonable. It is noteworthy to underline that turnover is substantially higher for the sophisticated strategy. Removing costs in the numerator (say, 0.005 times the turnover as in Goto and Xu (2015)) only mildly reduces the superiority in Sharpe ratio of the ML-based strategy.

Finally, it is always tempting to plot the corresponding portfolio values.

```
library(lubridate) # Date management
library(cowplot)   # Plot grid management
g1 <- tibble(date = t_oos,
      benchmark = cumprod(1+portf_returns[,1]),
      ml_based = cumprod(1+portf_returns[,2])) %>%
    gather(key = strat, value = value, -date) %>%
    ggplot(aes(x = date, y = value, color = strat)) + geom_line() +theme_grey()
g2 <- tibble(year = lubridate::year(t_oos),
             benchmark = portf_returns[,1],
             ml_based = portf_returns[,2]) %>%
    gather(key = strat, value = value, -year) %>%
    group_by(year, strat) %>%
    summarise(avg_return = mean(value)) %>%
    ggplot(aes(x = year, y = avg_return, fill = strat)) + geom_col(position = "dodge") + theme_grey()
plot_grid(g1,g2, nrow = 2)
```



Out of the 12 years of the backtest, the advanced strategy outperforms the benchmark during

10 years. It is less hurtful in all three years of aggregate losses (2008, 2015 and 2018). This is a satisfactory improvement because the EW benchmark is tough to beat!

13.7 Coding exercises

1. Code the advanced weighting function defined in Equation (13.3).
2. Test it in a small backtest and check its sensitivity to the parameters.
3. Using the functional programming package *purrr*,

14 Interpretability

This chapter is dedicated to the techniques that help understand the way models process inputs into outputs. A recent book (Molnar (2019) available at `https://christophm.github.io/interpretable-ml-book/`) is entirely devoted to this topic and we highly recommend to have a look at it. Another more introductory and less technical reference is Hall and Gill (2019). Obviously, in this chapter, we will adopt an tone which is factor-investing orientated and discuss examples related to ML models trained on a financial dataset.

Quantitative tools that aim for interpretability of ML models are required to satisfy two simple conditions:

1. That they provide information about the model.
2. That they are highly comprehensible.

Often, these tools generate graphical outputs which are easy to read and yield immediate conclusions.

In attempts to white-box complex machine learning models, one dichotomy stands out:

- **Global models** seek to determine the relative role of features in the construction of the prediction once the model has been trained. This is done at the global level, so that the output hold *on average* over the whole training set.
- **Local models** aim to characterise how the model behaves around one particular instance by considering small variations around this instance. The way these variations are processed by the original model allows to simplify it, i.e., approximate it, e.g., in a linear fashion. This approximation will determine the sign and magnitude of the impact of each relevant feature in the vicinity of the original instance.

To be fair, Molnar (2019) proposes another partition by splitting interpretations that depend on one particular model (e.g., linear regression or decision tree) versus the interpretations that can be obtained for any kind of model. In the sequel, we present the methods according to the global versus local dichotomy.

14.1 Global interpretations

Let us start with the simplest example of all. In a linear model,

$$y_i = \alpha + \sum_{k=1}^{K} \beta_k x_i^k + \epsilon_i,$$

the following elements are usually extracted from the estimation of the β_k:

- the R^2, which appreciates the global fit of the model (possibly penalized to prevent overfitting with many regressors);

- the sign of the estimates $\hat{\beta}_k$, which indicates the direction of the impact of each feature x^k on y;
- the t-statistics $t_{\hat{\beta}_k}$, which evaluate the magnitude of this impact: regardless of its direction, large statistics in absolute value reveal prominent variables. Often, the t-statistics are translated into p-values which are computed under some suitable distributional assumptions.

The last two indicators are useful because they inform the user on which features matter the most and on the sign of the effect of each predictor. This gives a simplified view of how the model processes the features into the output. Most tools that aim to explain black boxes follow the same principles.

14.1.1 Variable importance (tree-based)

One incredibly favorable feature of simple decision trees is their interpretability. Their visual representation is simple and straightforward. Just like regressions (which are another building block in ML), simple trees are simple to comprehend and do not suffer from the black-box rebuke that is often associated to more sophisticated tools.

Indeed, both random forests and boosted trees fail to provide perfectly accurate accounts of what is happening inside the engine. Nonetheless, it is possible to compute the aggregate share (or importance) of each feature in the determination of the structure of the tree once it has been trained.

After training, it is possible to compute, at each node n the gain $G(n)$ obtained by the subsequent split (if there are any, i.e., if the node is not a terminal leaf). It is also easy to determine which variable is chosen to perform the split to write $\mathcal{N}_k$ the set of nodes related to feature k. Then, the global importance of each feature is given by

$$I(k)=\sum_{n\in\mathcal{N}_k}G(n),$$

and it is often rescaled so that the sum of $I(k)$ across all k is equal to one. In this case, $I(k)$ measures the relative contribution of feature k in the reduction of loss during the training. A variable with high importance will have a greater impact on predictions. Generally, these variables are those that are located close to the root of the tree.

Below, we take a look at the results obtained from the tree-based models trained in Chapter 7. We start by recylcling the output from the three regression models we used.

```
tree_VI <- fit_tree$variable.importance  %>%                            # VI from tree model
    as_tibble(rownames = NA) %>%                                        # Transform in tibble
    rownames_to_column("Feature")                                       # Add feature column
RF_VI <- fit_RF$importance  %>%                                         # VI from random forest
    as_tibble(rownames = NA) %>%                                        # Transform in tibble
    rownames_to_column("Feature")                                       # Add feature column
XGB_VI <- xgb.importance(model = fit_xgb)[,1:2]                        # VI from boosted trees
VI_trees <- tree_VI %>% left_join(RF_VI) %>% left_join(XGB_VI)          # Aggregate the VIs
colnames(VI_trees)[2:4] <- c("Tree", "RF", "XGB")                       # New column names
norm_1 <- function(x){return(x / sum(x))}                               # Normalizing function
VI_trees %>% na.omit %>% mutate_if(is.numeric,  norm_1) %>%             # Plotting sequence
```

```
gather(key = model, value = value, -Feature) %>%
ggplot(aes(x = Feature, y = value, fill = model)) + geom_col(position = "dodge") +
theme(axis.text.x = element_text(angle = 35, hjust = 1))
```

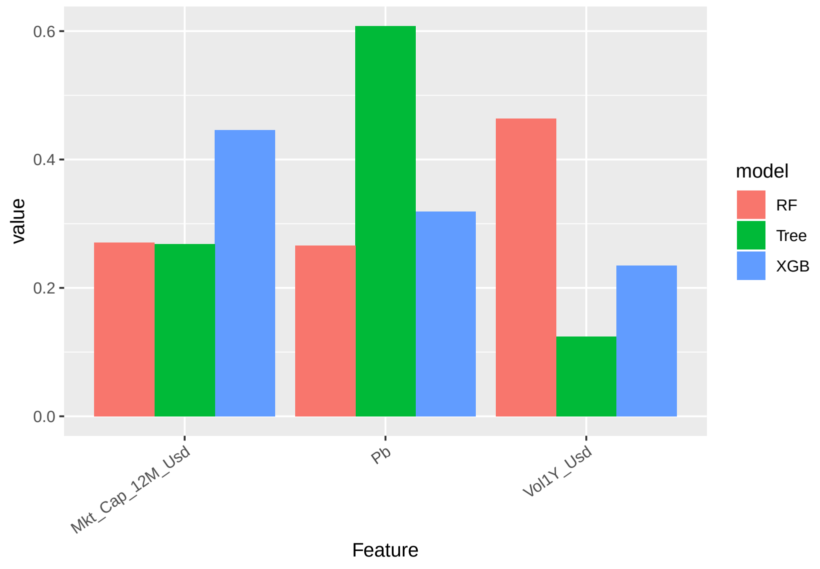


FIGURE 14.1: Variable importance for tree-based models

In the above code, tibbles are like dataframes (they are the v2 of dataframes so to speak). Given the way the graph is coded, Figure 14.1 is in fact misleading. Indeed, by construction, the tree model only has a small number of features with nonzero importance. The graph shows these variables (with the intersection of nonzero importances of the other two models). For scale reasons, the normalization is performed *after* the subset of features is chosen, hence the bars do not exactly correspond to the true values. Nonetheless, we prefered to limit the number of features shown on the graph for obvious readability concerns.

There are differences in the way the models rely on the features. For instance, the boosted tree model gives the most importance to Ebit_Ta, while the other two models overlook this variable. Across all models, market capitalization is pregnant as it appears via three definitions of the variable. The price-to-book ratio and volatility measures are the other two type of features that seem to matter in the training of the models.

One defining property of random forests is that they give a chance to all features. Indeed, by randomizing the choice of predictors, each individual exogenous variable has a shot at explaining the label. The scores of the features for RF seem more balanced and the spread between the most important and least important feature is the smallest among the three models.

14.1.2 Variable importance (agnostic)

The idea of quantifying the importance of each feature in the learning process can be extended to non tree-based models. We refer to the paper mentioned in the study Fisher et al. (2018) for more information on this stream of the literature. The premise is the same as above: the aim is to quantify to what extent one feature contributes to the learning process.

One way to track the added value of one particular feature is to look at what happens if its values inside the training set are entirely shuffled. If the original feature plays an important role in the explanation of the dependent variable, then the shuffled version of the feature will lead to a much higher loss.

The baseline method to assess feature importance in the general case is the following.

1. Train the model on the original data and compute the associated loss l^*.
2. For each feature k, create a new training dataset in which the features' values and randomly permuted. Then, train the model on this altered sample and record the corresponding loss l_k.
3. Rank the variable importance of each feature, computed as a difference $\text{VI}_k = l_k - l^*$ or a ratio $\text{VI}_k = l_k/l^*$.

The above procedure is of course random and can be repeated so that the importances are averages to improve the stability of the results. This algorithm is implemented in the FeatureImp() function of the iml package developed by the author of Molnar (2019). Below, we implement this algorithm for the features appearing on Figure 14.1. We test this approach on ridge regressions and recycle the variables used in Chapter 6. We start by the first step: computing the loss on the original training sample.

```
fit_ridge_0 <- glmnet(x_penalized_train, y_penalized_train, alpha = 0, lambda = 0.01) # Trained model
l_star <- mean((y_penalized_train-predict(fit_ridge_0, x_penalized_train))^2)         # Loss
```

Next, we evaluate the loss when each of the predictors have been sequentially shuffled. To reduce computation time, we only make one round of shuffling.

```
l <- c()
for(i in 1:nrow(VI_trees)){
    temp_data <- training_sample %>% select(features)                             # Temp feature matrix
    temp_data[, which(colnames(temp_data) == as.character(VI_trees[i,1]))] <-     # This line shuffles the values
        sample(temp_data[, which(colnames(temp_data) == as.character(VI_trees[i,1]))]
               %>% pull(1), replace = FALSE)
    x_penalized_temp <- temp_data %>% as.matrix()                                 # Turns predictors into matrix
    fit_ridge_temp <- glmnet(x_penalized_temp, y_penalized_train,                 # Fits the model
                             alpha = 0, lambda = 0.01)
    l[i] <- mean((y_penalized_train-predict(fit_ridge_temp, x_penalized_temp))^2) # Computes loss
}
```

Finally, we plot the results.

```
data.frame(Feature = VI_trees[,1], loss = l - l_star) %>%
    ggplot(aes(x = Feature, y = loss)) + geom_col() +
    theme(axis.text.x = element_text(angle = 35, hjust = 1))
```

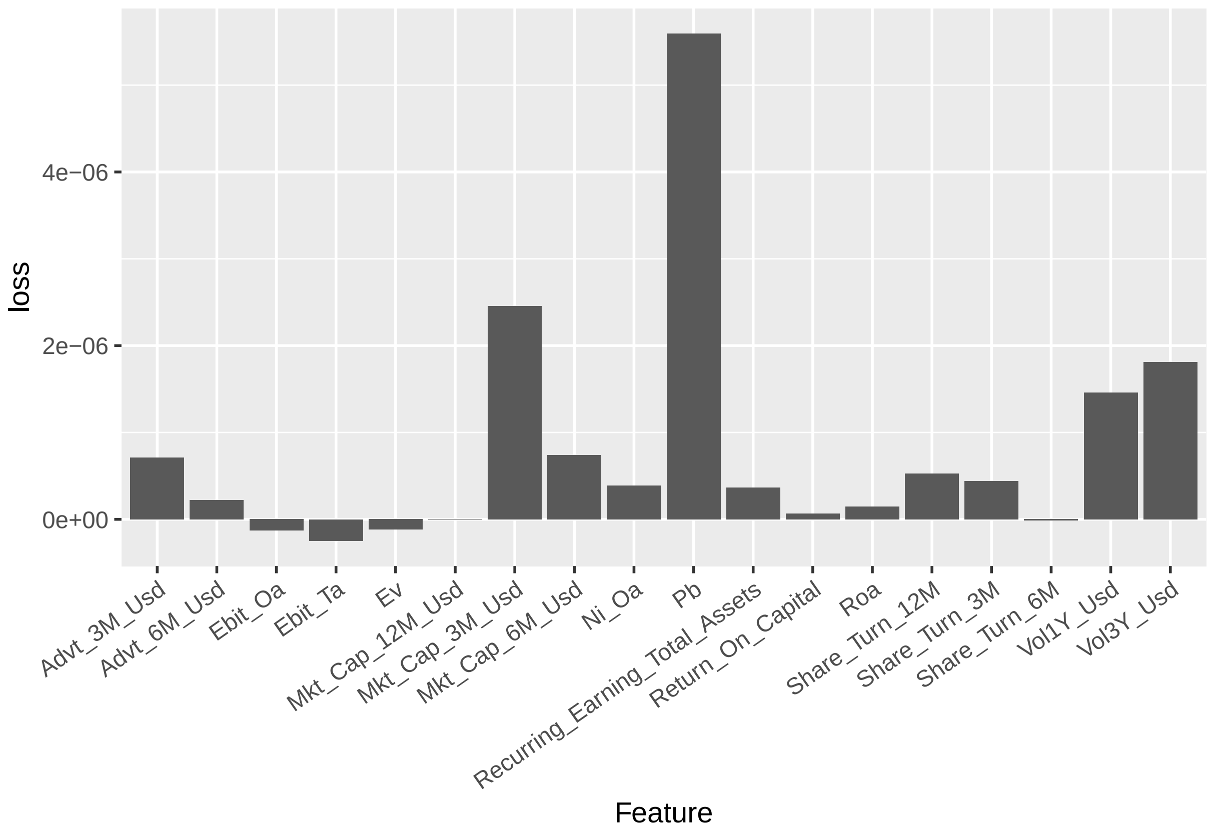


FIGURE 14.2: Variable importance for a ridge regression model.

The resulting importances are in line with thoses of thre tree-based models: the most prominent variables are volatility-based, market capitalization-based, and the price-to-book ratio. Note that in some cases (e.g., the ROA), the score can even be negative, which means that the predictions are more accurate than the baseline model when the values of the predictor are shuffled!

14.1.3 Partial dependence plot

Partial dependence plots (PDPs) aim at showing the relationship between the output of a model and the value of a feature (we refer to section 8.2 of Friedman (2001) for an early treatment of this subject).

Let us fix a feature k. We want to understand the average impact of k on the predictions of the trained model $\hat{f}$. In order to do so, we assume that the feature space is random and we split it in two: k versus $-k$, which stands for all features except for k. The partial dependence plot is defined as

$$\bar{f}_k(x_k) = \mathbb{E}[\hat{f}(\mathbf{x}_{-k}, x_k)] = \int \hat{f}(\mathbf{x}_{-k}, x_k) d\mathbb{P}_{-k}(\mathbf{x}_{-k}), \tag{14.1}$$

where $d\mathbb{P}_{-k}(\cdot)$ is the (multivariate) distribution of the non-k features $\mathbf{x}_{-k}$. In practice, this average is evaluated using Monte-Carlo simulations:

$$\bar{f}_k(x_k) \approx \frac{1}{M}\sum_{m=1}^{M} \hat{f}\left(x_k, \mathbf{x}_{-k}^{(m)}\right), \tag{14.2}$$

where $\mathbf{x}_{-k}^{(m)}$ are independent samples of the non-k features.

Theoretically, PDPs could be computed for more than one feature at a time. In practice, this is only possible for two features (yielding a 3D surface) and is more computationally intense.

Finally, we refer to Zhao and Hastie (2019) for a theoretical discussion on the *causality* property of PDPs. Indeed, a deep look at the construction of the PDPs suggests that they could be interpreted as a causal representation of the feature on the model's output.

We illustrate this concept below, using the dedicated package iml (interpretable machine learning). The model we seek to explain is the random forest built in Section 7.2. We recycle some variables used therein. We choose to test the impact of the price-to-book ratio on the outcome of the model.

```
library(iml)                                         # One package for interpretability
mod_iml <- Predictor$new(fit_RF,                     # This line encapsulates the objects
                         data = training_sample %>% select(features))
pdp_PB = FeatureEffect$new(mod_iml, feature = "Pb")  # This line computes the PDP for price-to-book ratio
plot(pdp_PB)                                         # Plot the partial dependence.
```

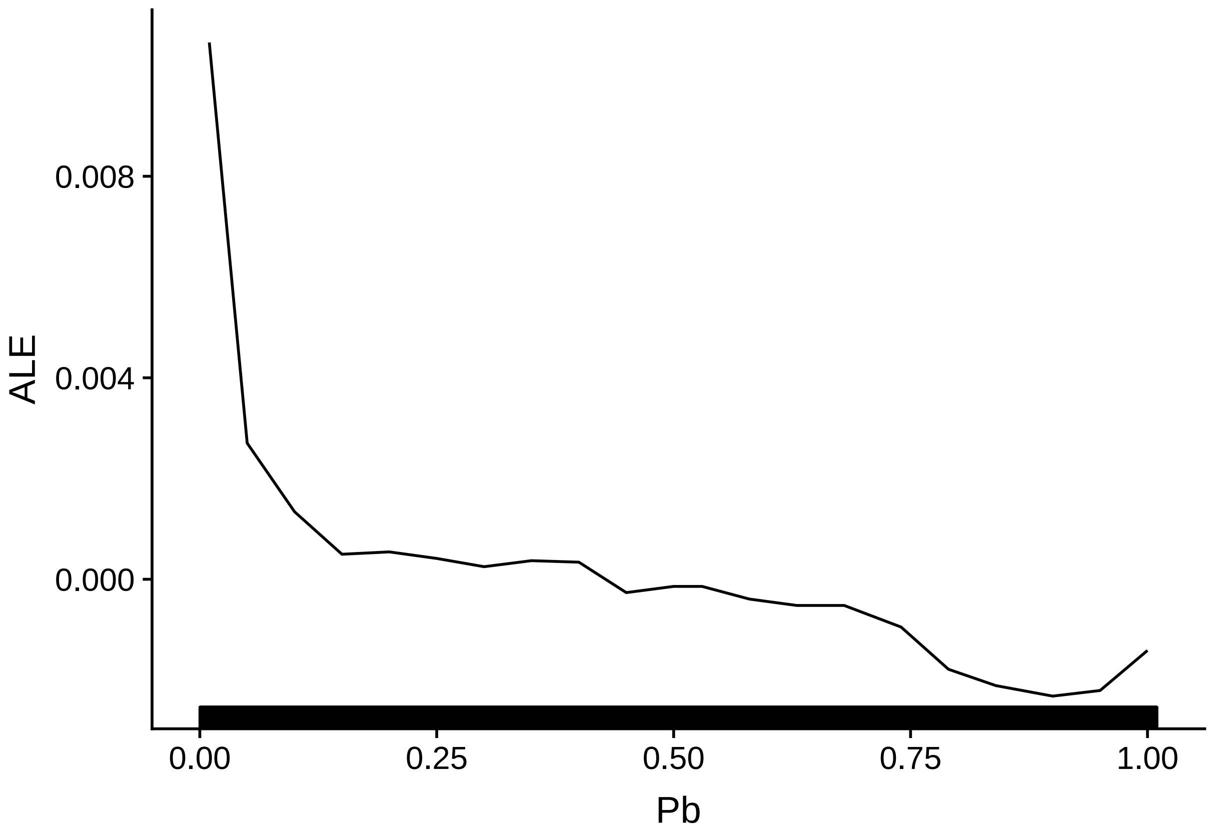


FIGURE 14.3: Partial dependence plot for the price-to-book ratio on the random forest model

The average impact of the price-to-book ratio on the predictions is decreasing. This was somewhat expected, given the conditional average of the dependent variable given the price-to-book ratio. This latter function is depicted in Figure 7.3 and shows a behavior comparable to the above curve: strongly decreasing for small value of P/B and then relatively flat.

14.2 Local interpretations

Whereas global interpretations seek to assess the impact of features on the output *overall*, local methods try to quantify the behavior of the model on particular instances or the neighborhood thereof. Local interpretability has recently gained traction and many papers have been published on this topic. Below, we outline the most widespread methods.[1]

[1]For instance, we do not mention the work of Horel and Giesecke (2019) but the interested reader can have a look at their work on neural networks (and also at the references cited in the paper).

14.2.1 LIME

LIME (Local Interpretable Model-Agnostic Explanations) is a methodology originally proposed by Ribeiro et al. (2016). Their aim is to provide a faithfull account of the model under two constraints:

- **simple interpretability**, which implies a limited number of variables with visual or textual representation. This is to make sure any human can easily understand the outcome of the tool;
- **local faithfulness**: the explanation holds for the vicinity of the instance.

The original (black-box) model is f and we assume we want to approximate its behavior around instance x with the interpretable model g.[2] The simple function g belongs to a larger class G. The vicinity of x is denoted π_x and the complexity of g is written $\Omega(g)$. LIME seeks an interpretation of the form

$$\xi(x)=\underset{g\in G}{\text{argmin}}\,\mathcal{L}(f,g,\pi_x)+\Omega(g),$$

where $\mathcal{L}(f,g,\pi_x)$ is the loss function (error/imprecision) induced by g in the vicinity π_x of x. The penalisation $\Omega(g)$ is for instance the number of leaves or depth of a tree, or the number of predictors in a linear regression.

It now remains to define some of the above terms. The vicinity of x is defined by $\pi_x(z)=e^{-D(x,z)^2/\sigma^2}$, where D is some distance measure. We underline that this function decreases when z shifts away from x.

The tricky part is the loss function. In order to minimise it, LIME generates artificial samples close to x and averages/sums the error on the label that the simple representation makes. For simplicity, we assume a scalar output for f, hence the formulation is the following:

$$\mathcal{L}(f,g,\pi_x)=\sum_z \pi_x(z)(f(z)-g(z))^2$$

and the errors are weighted according to their distance from the initial instance x: the closest points get the largest weights. In its most basic implementation, the set of models G consists of all linear models.

In Figure 14.4, we provide a simplified diagram of how LIME works.

For expositional clarity, we work with only one dependent variable. The original training sample is shown with the black points. The fitted (trained) model is represented with the blue line and we want to approximate how the model works around one particular instance which is highlighted by the red square around it. In order to build the approximation, we sample 5 new points around the instance (the 5 red triangles). Each triangle lies on the blue line (they are model predictions) and has a weight proportional to its size: the triangle closest to the instance has a bigger weight. Using weighted least-squares, we build a linear model that fits to these 5 points. This is the outcome of the approximation. It gives the two parameters of the model: the intercept and the slope. Both can be evaluated with standard statistical tests.

[2]In the original paper, the authors dig deeper into the notion of interpretable representations. In complex machine learning settings (image recognition or natural language processing), the original features given to the model can be hard to interpret. Hence, this requires an additional translation layer because the outcome of LIME must be expressed in terms of easily understood quantities. In factor investing, the features are elementary, hence we do not need to deal with this issue).

The sign of the slope is important. It is fairly clear that if the instance had been taken closer to $x = 0$, the slope would have probably been almost flat and hence the predictor could be locally discarded. Another important detail is the number of sample points. In our explanation, we take only five, but in practice, a robust estimation usually requires around one thousand points or more. Indeed, when too few neighbors are sampled, the estimation risk is high and the approximation may be rough.

We proceed with an example of implementation. There are several steps:

1. Fit a model on some training data.

2. Wrap everything using the lime() function.

3. Focus on a few predictors and see their impact over a few particular instances (via the explain() function).

We start with the first step. This time, we work with a boosted tree model.

```
library(lime)                                  # Package for LIME interpretation
params_xgb <- list(                            # Parameters of the boosted tree
    max_depth = 5,                             # Max depth of each tree
    eta = 0.5,                                 # Learning rate
    gamma = 0.1,                               # Penalization
    colsample_bytree = 1,                      # Proportion of predictors to be sampled (1 = all)
    min_child_weight = 10,                     # Min number of instances in each node
    subsample = 1)                             # Proportion of instance to be sampled (1 = all)
xgb_model <- xgb.train(params_xgb,             # Training of the model
                       train_matrix_xgb,       # Training data
                       nrounds = 10)           # Number of trees
```

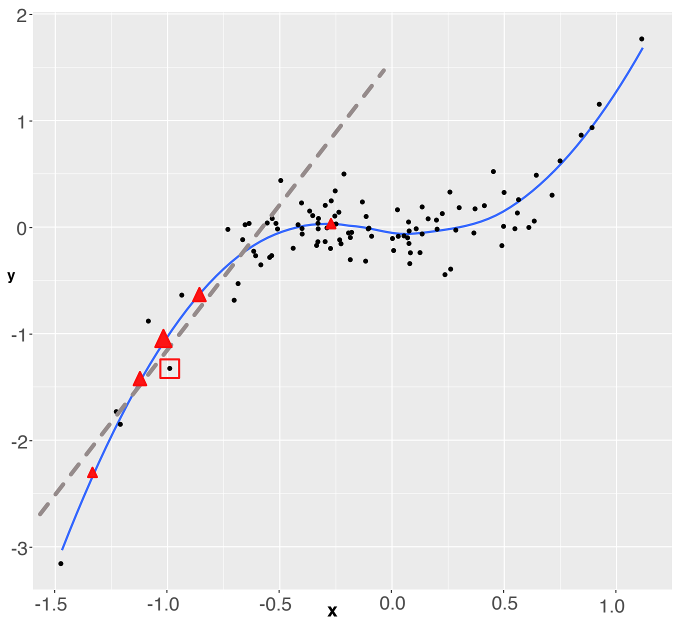
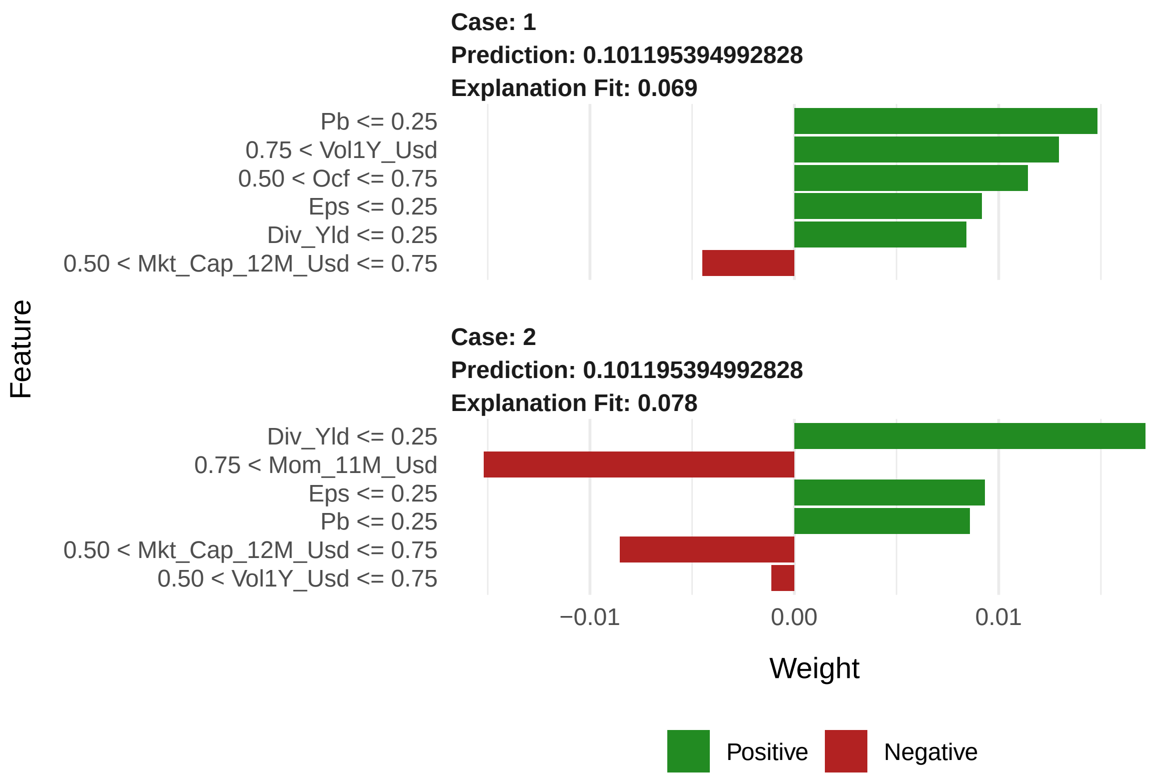


FIGURE 14.4: Simplistic explanation of LIME.

Then, we head on to steps two and three. As underlined above, we resort to the lime() and explain() functions.

```
explainer <- lime(training_sample %>% select(features_short), xgb_model) # Step 2.
explanation <- explain(x = training_sample %>%           # Step 3.
                           select(features_short) %>%
                           dplyr::slice(1:2),            # We look at the first two instances in train_sample
                       explainer = explainer,            # Input: the explainer variable created above
                       n_permutations = 900,             # Nb samples used to compute the loss function
                       dist_fun = "euclidean",           # Distance function. "gower" is one alternative
                       n_features = 6                    # Number of features displayed (most importance ones)
)
plot_features(explanation, ncol = 1)                     # Visual display
```



In each graph (one graph corresponds to the explanation around one instance), there are two types of information: the sign of the impact and the magnitude of the impact. The sign is revealed with the color (positive in green, negative in red) and the magnitude is shown with the size of the rectangles. For the first occurence, the Ebit_Oa variable is associated with a strong negative effect. In the second example, the Earning per Share has a strong positive effect on the future return.

The values on the left of the graphs show the ranges of the features with which the local approximations were computed.

Lastly, we briefly discuss the choice of distance function chosen in the code. It is used to evaluate the distance between the true instance and a simulated one to weight the prediction of the sampled instance. Our dataset comprises only numerical data, hence the Euclidean distance is a natural choice:

$$\text{Euclidean}(\mathbf{x},\mathbf{y})=\sqrt{\sum_{n=1}^N(x_i-y_i)^2}.$$

Another possible choice would be the Manhattan distance:

$$\text{Manhattan}(\mathbf{x},\mathbf{y})=\sum_{n=1}^N|x_i-y_i|.$$

The problem with these two distances is that they fail to handle categorical variables. This is where the Gower distance steps in (Gower (1971)). The distance imposes a different treatment on features of different types (classes versus numbers essentially, but it can also handle missing data!). For categorical features, the Gower distance applies a binary treatment: the value is equal to 1 if the features are equal and to zero if not (i.e., $1_{\{x_n=y_n\}}$). For numerical features, the spread is quantified as $1-\frac{|x_n-y_n|}{R_n}$, where R_n is the maximum absolute value the feature can take. All similarity measurements are then aggregated to yield the final score. Note that in this case, the logic is reversed: $\mathbf{x}$ and $\mathbf{y}$ are very close if the Gower distance is close to one and they are far away if the distance is close to zero.

14.2.2 Shapley values

The approach of Shapley values is somewhat different compared to LIME and closer in spirit to PDPs. It originates from cooperative game theory (Shapley (1953)). The rationale is the following. One way to assess the impact (or usefulness) of a variable is to look at what happens if we remove this variable from the dataset. If this is very detrimental to the quality to the model (i.e., to its predictions), then it means that the variable is substantially valuable.

The simplest way to proceed is to take all variables and remove one to evaluate its predictive ability. Shapley values are computed on a larger scale because they consider all possible combinations of variables to which they add the target predictor. Formally, this gives:

$$\phi_k=\sum_{S\subseteq\{x_1,\dots,x_K\}\backslash x_k}\underbrace{\frac{\text{Card}(S)!(K-\text{Card}(S)-1)!}{K!}}_{\text{weight of coalition}}\underbrace{\left(\hat{f}_{S\cup\{x_k\}}(S\cup\{x_k\})-\hat{f}_S(S)\right)}_{\text{gain when adding } x_k} \qquad (14.3)$$

S is any subset of the **coalition** that doesn't include feature k and its size is Card(S). \ In the equation above, the model f must be altered because it's impossible to evaluate f when features are missing. In this case, several possible options:

- set the missing value to its average or median value (in the whole sample) so that its effect is some 'average' effect
- directly compute an average value $\int_{\mathbb{R}f(x_1,\dots,x_k,\dots,x_K)d\mathbb{P}_{x_k}}$, where $d\mathbb{P}_{x_k}$ is the empirical distribution of x_k in the sample.

Obviously, Shapley values can take a lot of time to compute if the number of predictors is large. We refer to Chen et al. (2018) for a discussion on a simplifying method that reduces computation times in this case. Extensions of Shapley values for interpretability are studied in Lundberg and Lee (2017).

The implementation of Shapley values is permitted via the *iml* package. There are two restrictions compared to LIME. First, the features must be filtered upfront because all features are shown on the graph (which becomes illegible beyond 20 features). This is why in the code below, we use the short list of predictors (from Section 2.2). Second, instances are analyzed one at a time.

We start by fitting a random forest model.

```r
fit_RF_short <- randomForest(R1M_Usd ~.,    # Same formula as for simple trees!
                 data = training_sample %>% select(c(features_short), "R1M_Usd"),  # Data source
                 sampsize = 10000,          # Size of (random) sample for each tree
                 replace = FALSE,           # Is the sampling done with replacement?
                 nodesize = 250,            # Minimum size of terminal cluster
                 ntree = 40,                # Nb of random trees
                 mtry = 4                   # Nb of predictive variables for each tree
    )
```

We can then analyze the behavior of the model around the first instance of the training sample.

```r
predictor <- Predictor$new(fit_RF_short,    # This wraps the model & data
                          data = training_sample %>% select(features_short),
                          y = training_sample$R1M_Usd)
shapley <- Shapley$new(predictor,                          # Compute the Shapley values...
                       x.interest = training_sample %>%
                           select(features_short) %>%
                           dplyr::slice(1))                # On the first instance
plot(shapley) + coord_fixed(1500) +                        # Plot
    theme(axis.text.x = element_text(angle = 35, hjust = 1)) + coord_flip()
```

In the output, we XXXXXXX

14.2.3 Breakdown

Breakdown (see e.g., Staniak and Biecek (2018)) is a mixture of ideas from PDPs and Shapley values. The core of breakdown is the relaxed model prediction, which is close in spirit to Equation (14.1). The difference is that we are working at the local level, i.e., on one particular observation, say x^*. We want to measure the impact of a set of predictors on the prediction associated to x^*, hence we fix two sets $\mathbf{k}$ (fixed) and $-\mathbf{k}$ (free) and evaluate a proxy for the average prediction of the estimated model $\hat{f}$ when the set $\mathbf{k}$ of predictors is fixed at the values of x^*:

$$\tilde{f}_{\mathbf{k}}(x^*) = \frac{1}{M}\sum_{m=1}^{M} \hat{f}\left(x^{(m)}_{-\mathbf{k}}, x^*_{\mathbf{k}}\right).$$

The $x^{(m)}$ in the above expression are either simulated values of instances or simply sampled values from the dataset. The notation implies that the instance has some values replaced by those of x^*, namely those that correspond to the indices $\mathbf{k}$. When $\mathbf{k}$ consists of all features, then $\tilde{f}_{\mathbf{k}}(x^*)$ is equal to the raw model prediction $\hat{f}(x^*)$ and when $\mathbf{k}$ is empty, it is equal to the average sample value of the label (constant prediction).

The quantity of interest is the so-called contribution of feature $j \notin \mathbf{k}$ with respect to data point x^* and set $\mathbf{k}$:

$$\phi^j_{\mathbf{k}}(x^*) = \tilde{f}_{\mathbf{k}\cup j}(x^*) - \tilde{f}_{\mathbf{k}}(x^*).$$

Just as for Shapley values, the above indicator computes an average impact when augmenting the set of predictors with feature j. By definition, it depends on the set $\mathbf{k}$, so this is one notable difference with Shapley values. In Staniak and Biecek (2018), the authors devise a procedure that incrementally increases or decreases the set $\mathbf{k}$. This greedy idea helps alleviate the burden of computing all possible combinations of features. Moreover, a very convenient property of their algorithm is that the sum of all contributions is equal to the predicted value.

In order to illustrate one implementation of breakdown, we train a random forest on a limited number of features, as shown below. This will increase the readability of the output of the breakdown.

```
formula_short <- paste("R1M_Usd ~", paste(features_short, collapse = " + ")) #  Model
formula_short <- as.formula(formula_short)                                  # Forcing formula
fit_RF_short <- randomForest(formula_short, # Same formula as before
                 data = select(training_sample, c(features_short, "R1M_Usd")), # Features + label
                 sampsize = 10000,          # Size of (random) sample for each tree
                 replace = FALSE,           # Is the sampling done with replacement?
                 nodesize = 250,            # Minimum size of terminal cluster
                 ntree = 12,                # Nb of random trees
```

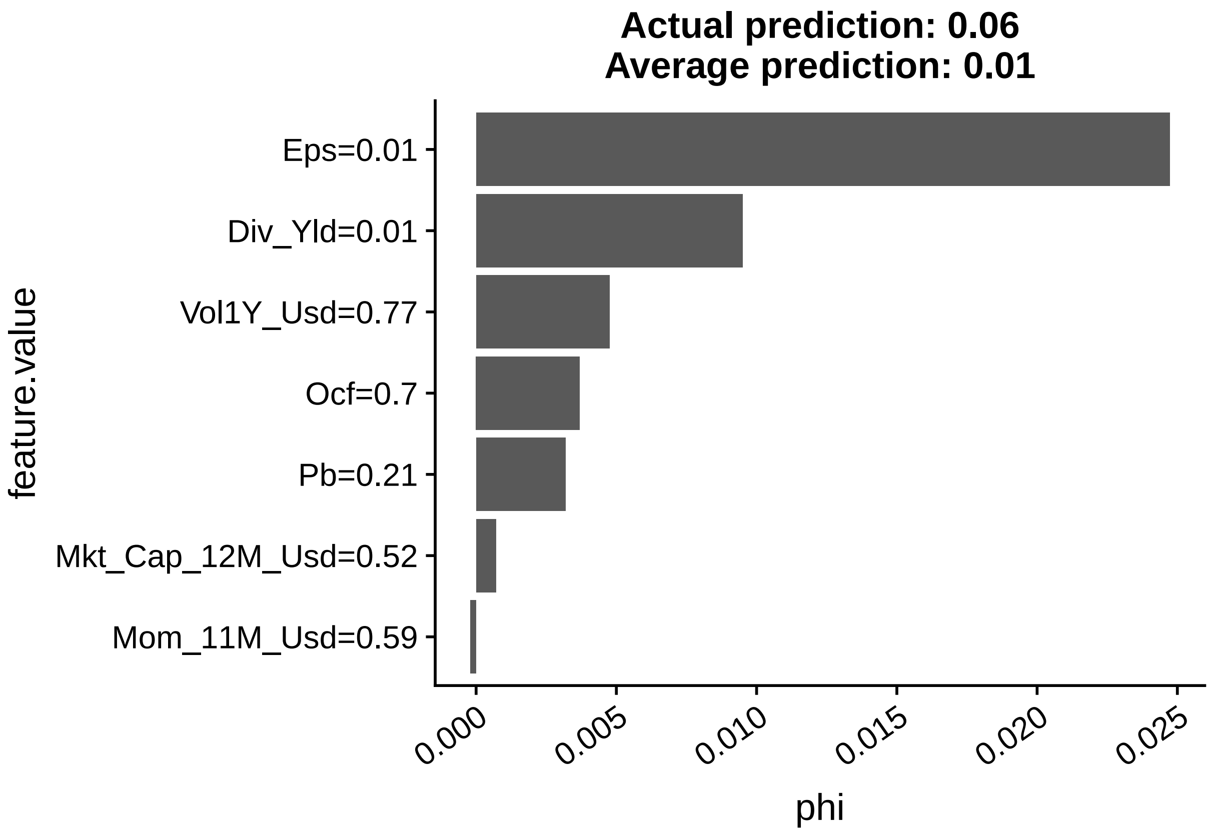


FIGURE 14.5: Illustration of the Shapley method.

```
                mtry = 5                    # Nb of predictive variables for each tree
    )
```

Once the model is trained, the syntax for the breakdown of predictions is very simple.

```
library(breakDown)
explain_break <- broken(fit_RF_short,
                        data_ml[6,] %>% select(features_short),
                        data = data_ml %>% select(features_short))
plot(explain_break)
```

The graphical output is intuitively interpreted. The grey bar is the prediction of the model at the chosen instance. Green bars signal a positive contribution and the yellowish rectangles show the variables with negative impact. The relative sizes indicate the importance of each feature.

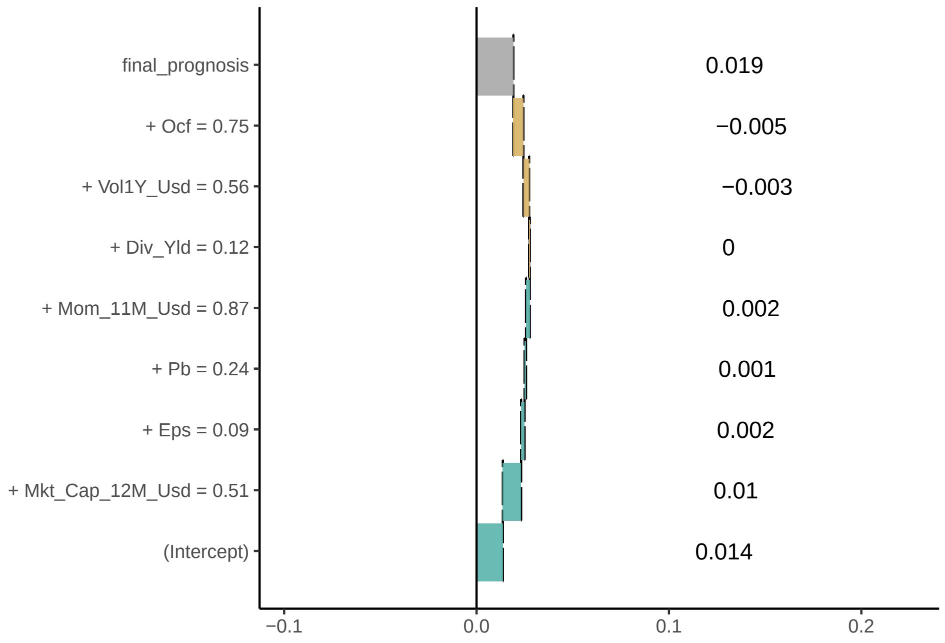


FIGURE 14.6: Example of a breakdown output.

15

Two key concepts: causality and non-stationarity

A prominent point of criticism faced by ML tools is their inability to uncover causality relationship between features and labels because they are mostly focused (by design) to capture correlations. Correlations are much weaker than causality because they characterize a two way relationship ($\mathbf{X} \leftrightarrow \mathbf{y}$) while causality specifies a direction $\mathbf{X} \rightarrow \mathbf{y}$ or $\mathbf{X} \leftarrow \mathbf{y}$. One fashionable example is sentiment. Many academic articles seem to find that sentiment (irrespectively of its definition) is a significant driver of future returns. A high sentiment for a particular stock may increase the demand for this stock and push its price up (though contrarian reasonings may also apply: if sentiment is high, it is a sign that mean-reversion is possibly about to happen). The reverse causation is also plausible: returns may well cause sentiment. If a stock experiences a long period of market growth, people become bullish about this stock and sentiment increases (this notably comes from extrapolation, see Barberis et al. (2015) for a theoretical model). In Coqueret (2018), it is found (in opposition to most findings in this field), that the latter relationship (returns → sentiment) is more likely. This result is backed by causality driven tests (see Section 15.1.1).

Statistical causality is a large field and we refer to Pearl (2009) for a deep dive into this topic. Recently, researchers have sought to link causality with ML approaches (see, e.g., Peters et al. (2017), Heinze-Deml et al. (2018), Arjovsky et al. (2019)). The key notion in their work is **invariance**.

Often, data is collected not at once, but from difference sources at different moments. Some relationships found in these different sources will changes while others may remain the same. The relationships that are invariant to changing environments are likely to stem from (and signal) causality. One counter-example is the following (related in Beery et al. (2018)): training a computer vision algorithm to discriminate between cows and camels will lead the algorithm to focus on grass versus sand! Thus, a picture of a camel on grass will be classified as cow while a cow on sand would be labelled "camel". It is only with pictures of these two animals in different contexts (environments) that the learner will end up truly finding what makes a cow and a camel. A camel will remain a camel no matter where it is pictured: it should be recognized as such by the learner. If so, the representation of the camel becomes invariant over all datasets and the learner has discovered causality, i.e., the true attributes that make a camel a camel (overall silhouette, shape of the back, face, color (possibly misleading!), etc.).

This search for invariance makes sense for many disciplines like computer vision or natural language processing (languages don't change much). In finance, it is not obvious that invariance may exist. Market conditions are known to be time-varying and the relationships between firm characteristics also change from year to year. One solution to this issue may simply be to embrace non-stationarity. In Chapter 13, we advocate to do that by updating models as frequently as possible with rolling training sets: this allows the predictions to be based on the most recent trends. In Section 15.2 below, we introduce other theoretical and practical options.

15.1 Causality

Traditional machine learning models aim to uncover relationships between variables but do not usually specify *directions* for these relationships. One typical example is the linear regression. If we write $y = a + bx + \epsilon$, then it is also true that $x = b^{-1}(y - a - \epsilon)$, which is of course also a linear relationship (with respect to y). These equations do not define causation whereby x would be a clear determinant of y ($x \rightarrow y$, but the opposite could be false).

15.1.1 Granger causality

The most notable tool first proposed by Granger (1969) is probably the simplest. For simplicity, we consider only two stationary processes, X_t and Y_t. A strict definition of causality could be the following. X can be said to cause Y, whenever, for some integer k,

$$(Y_{t+1}, \dots, Y_{t+k})|(\mathcal{F}_{Y,t} \cup \mathcal{F}_{X,t}) \quad \overset{d}{\neq} \quad (Y_{t+1}, \dots, Y_{t+k})|\mathcal{F}_{Y,t},$$

that is, when the distribution of future values of Y_t, conditionally on the knowledge of both processes is not the same as the distribution with the sole knowledge of the filtration $\mathcal{F}_{Y,t}$. Hence X does have an impact on Y because its trajectory alters that of Y.

Now, this formulation is too vague and impossible to handle numerically, thus we simplify the setting via a linear formulation. We keep the same notations as Section 5 of the original paper Granger (1969). The tests consists in two regressions:

$$X_t = \sum_{j=1}^{m} a_j X_{t-j} + \sum_{j=1}^{m} b_j Y_{t-j} + \epsilon_t$$
$$Y_y = \sum_{j=1}^{m} c_j X_{t-j} + \sum_{j=1}^{m} d_j Y_{t-j} + \nu_t$$

where for simplicity, it is assumed that both processes have zero mean. The usual assumptions apply: the Gaussian noises ϵ_t and ν_t are uncorrelated in every possible way (mutually and through time). The test is the following: if one b_j is nonzero, then it is said that Y Granger-causes X and if one c_j is nonzero, X Granger-causes Y. The two are not mutually exclusive and it is widely accepted that feedback loops can very well occur.

Statistically, under the null hypothesis, $b_1 = \dots = b_m = 0$ (*resp.* $c_1 = \dots = c_m = 0$), which can be tested using the usual Fischer distribution. Obviously, the linear restriction can be dismissed but the tests are then much more complex. The main financial article in this direction is Hiemstra and Jones (1994).

There are many R packages that embed Granger causality functionalities. One of the most widespread is *lmtest* so we work with it below. The syntax is incredibly simple. The *order* is the maximum lag m in the above equation. We test if market capitalization averaged over the past 6 months Granger-causes 1 month ahead returns for one particular stock (the first in the sample).

```
library(lmtest)
x_granger <- training_sample %>%                        # X variable =...
                                filter(stock_id ==1) %>%      # ... stock nb 1
                                pull(Mkt_Cap_6M_Usd)          # ... & Market cap
y_granger <- training_sample %>%                        # Y variable = ...
                                filter(stock_id ==1) %>%      # ... stock nb 1
                                pull(R1M_Usd)                 # ... & 1M return
fit_granger <- grangertest(x_granger,                   # X variable
                            y_granger,                  # Y variable
                            order = 6,                  # Maximmum lag
                            na.action = na.omit)        # What to do with missing data
fit_granger
```

```
## Granger causality test
##
## Model 1: y_granger ~ Lags(y_granger, 1:6) + Lags(x_granger, 1:6)
## Model 2: y_granger ~ Lags(y_granger, 1:6)
##   Res.Df Df     F    Pr(>F)
## 1    149
## 2    155 -6 4.111 0.0007554 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The test is directional and only test if X Granger-causes Y. In order to test the reverse effect, it is required to inverse the arguments in the function. In the output above, the p-value is very low, hence the probability of observing similar samples as ours knowing that H_0 holds is negligible. Thus is seems that market capitalization does Granger cause one month returns.

15.1.2 Causal additive models

The zoo of causal model encompasses a variety of beasts. The interested reader can have a peek at Pearl (2009), Peters et al. (2017) and Maathuis et al. (2018) and the references therein. It is always hard to single out one type of model in particular so we choose one that can be explained with simple mathematical tools.

We start with the simplest definition of a structural causal model (SCM), where we follow here Chapter 3 of Peters et al. (2017). The idea behind these models is to introduce some hierarchy (i.e., some additional structure) in the model. Formally, this gives

$$\begin{aligned} X &= \epsilon_X \\ Y &= f(X, \epsilon_Y), \end{aligned}$$

where the ϵ_X and ϵ_Y are independent noise variables. Plainly, a realization of X is drawn randomly and has then an impact on the realization of Y via f. Now this scheme could be more complex if the number of observed variable was larger. Imagine a third variable comes in so that

$$\begin{aligned} X &= \epsilon_X \\ Y &= f(X, \epsilon_Y), \\ Z &= g(Y, \epsilon_Z) \end{aligned}$$

In this case, X has a causation effect on Y and then Y has a causation effect on Z. We thus have the following connexions:

$$\begin{tikzcd}
X \arrow[dr] & & \\
& Y \arrow[r] & Z. \\
\epsilon_Y \arrow[ur] & \epsilon_Z \arrow[ur] &
\end{tikzcd}$$

The above representation is called a graph and graph theory has its own nomenclature, which we very briefly summarize. The variables are often referred to as *vertices* (or *nodes*) and the arrows as *edges*. Because arrows have a direction, they are called *directed* edges. When two vertices are connected via an edge, they are called *adjacent*. A sequence of adjacent vertices is called a *path* and it is directed if all edges are arrows. Within a directed path, a vertex that comes first is a parent node and the one just after is a child node.

Graphs can be summarized by adjacency matrices. An adjacency matrix $\mathbf{A} = A_{ij}$ is a matrix filled with zeros and ones. $A_{ij} = 1$ whenever there is an edge from vertex i to vertex j. Usually, self-loops ($X \to X$) are prohibited so that adjacency matrices have zeros on the diagonal. If we consider a simplified version of the above graph like $X \to Y \to Z$, the corresponding adjacency matrix is

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}.$$

where letters X, Y, and Z are naturally ordered alphabetically. There are only two arrows: from X to Y (first row, second column) and from Y to Z (second row, third column).

A *cycle* is a particular type of path that creates a loop, i.e., when the first vertice is also the last. The sequence $X \to Y \to Z \to X$ is a cycle. Technically, cycles pose problems. To illustrate this, consider the simple sequence $X \to Y \to X$. This would imply that a realization of X causes Y which in turn would cause the realization of Y. While Granger causality can be viewed as allowing this kind of connexion, general causal models usually avoid cycles and work with direct acyclic graphs (DAGs).

Equipped with these tools, we can explicit a very general form of models:

$$X_j = f_j\left(\mathbf{X}_{\mathrm{pa}_D(j)}, \epsilon_j\right), \tag{15.1}$$

where the noise variables are mutually independent. The notation $\mathrm{pa}_D(j)$ refers to the set of parent nodes of vertex j within the graph structure D. Hence, X_j is a function of all of its parents and some noise term ϵ_j. An additive causal model is a mild simplification of the above specification:

$$X_j = \sum_{k \in \mathrm{pa}_D(j)} f_{j,k}\left(\mathbf{X}_k\right) + \epsilon_j, \tag{15.2}$$

where the nonlinear effect of each variable is cumulative, hence the term '*additive*'. Note that there is no time index there. In contrast to Granger causality, there is no natural ordering. Such models are very complex and hard to estimate. The details can be found in Bühlmann et al. (2014). Fortunately, the authors have developed an R package that determines the DAG D.

Below, we build the adjacency matrix pertaining to the small set of predictor variables plus the 1 month ahead return (on the training sample). We use the *CAM* package which has a very simple syntax.

```
library(CAM)
fit_cam <- CAM(training_sample %>% select(c("R1M_Usd", features_short)))
fit_cam$Adj
```

```
## 8 x 8 sparse Matrix of class "dgCMatrix"
##
## [1,] . 1 1 1 1 1 1 1
## [2,] . . . 1 . . 1 .
## [3,] . 1 . 1 . . 1 1
## [4,] . . . . . . . .
## [5,] . 1 1 1 . 1 1 1
## [6,] . 1 1 1 . . 1 1
## [7,] . . . 1 . . . .
## [8,] . 1 . 1 . . 1 .
```

The matrix is not too sparse, which means that a the model has uncovered many relationships between the variables within the sample. Sadly, none are in the direction that are of interest for the prediction task that we seek. Indeed, the first variable is the one we want to predict and its column is empty. However, its row is full, which indicates the reverse effect: future returns cause the predictors values, which may seem rather counter-intuitive.

15.2 Dealing with changing environments

The most common assumption in machine learning contributions is that the samples that are studied are i.i.d. realizations of a phenomenon that we are trying to characterize. This constraint is natural because if the relationship between X and y always changes, then it is very hard to infer anything from observations. One major problem in Finance is that this is often the case: markets, behaviors, policies, etc., evolve all the time. This can be seen as a consequence of absence of arbitrage: if a trading strategy worked all the time, all agents would eventually adopt it via herding, which would annihilate the corresponding gains.[1] If the strategy is kept private, its holder would become infinitely rich, which obviously has never happened.

There are several ways to define changes in environments. If we denote with $\mathbb{P}_{XY}$ the multivariate distribution of all variables, with $\mathbb{P}_{XY}=\mathbb{P}_X\mathbb{P}_{Y|X}$, then two simple changes are possible:

- **covariate shift**: $\mathbb{P}_X$ changes but $\mathbb{P}_{Y|X}$ does not: the features have a fluctuating distribution, but their relationship with Y holds still;
- **concept drift**: $\mathbb{P}_{Y|X}$ changes but $\mathbb{P}_X$ does not: features are stable, but their relation to Y is altered.

Obviously, we omit the case when both items change is too complex to handle. In factor investing, the feature engineering process (see Section 5.4) is parlty designed to bypass

[1]See for instance the papers on herding in factor investing: Krkoska and Schenk-Hoppé (2019) and Santi and Zwinkels (2018).

the risk of covariate shift. Uniformization guarantees that the marginals stay the same but correlations between features may of course change. The main issue is probably concept drift when the way features explain the label changes through time. In Cornuejols et al. (2018), the authors distinguish four types of drifts, which we reproduce in Figure 15.1. In factor models, changes are presumably a combination of all four types: they can be abrupt during crashes, but most of the time they are progressive (gradual or incremental) and never ending (continuously recurring).

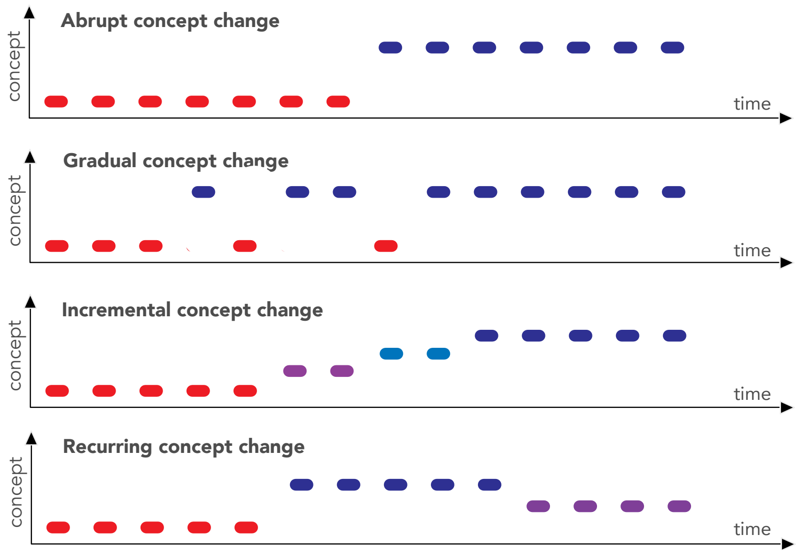


FIGURE 15.1: Different flavours of concept change.

Naturally, if we aknowledge that the environment changes, it appears logical to adapt models accordingly, i.e., dynamically. This gives rise to the so-called **stability-plasticity dilemma**. This dilemma is a trade-off between model **reactiveness** (new instances have an important impact on updates) versus **stability** (these instances may not be representative of a slower trend and thus shift the model in a suboptimal direction).

Practically, there are two ways to shift the cursor with respect to this dilemma: alter the chronological depth of the training sample (e.g., go further back in time) or, when it's possible, allocate more weight to recent instances. We discuss the first option in Section 13.1 and the second is mentioned in Section 7.3 (though the purpose in Adaboost is precisely to let the algorithm handle the weights). In neural networks, it is possible, in all generality to introduce instance-based weights in the computation of the loss function, though this option is not (yet) available in keras (to the best of our knowledge). For simple regressions, this idea is known as **weighted least squares** wherein errors are weighted inside the loss:

$$L=\sum_{i=1}^I w_i(y_i-\mathbf{x}_i\mathbf{b})^2.$$

In matrix terms, $L=(\mathbf{y}-\mathbf{X}\mathbf{b})'\mathbf{W}(\mathbf{y}-\mathbf{X}\mathbf{b})$, where $\mathbf{W}$ is a diagonal matrix. The gradient with respect to $\mathbf{b}$ is equal to $2\mathbf{X}'\mathbf{W}\mathbf{X}\mathbf{b}-2\mathbf{X}'\mathbf{W}\mathbf{y}$ so that the loss is minimized for $\mathbf{b}^*=(\mathbf{X}'\mathbf{W}\mathbf{X})^{-1}\mathbf{X}'\mathbf{W}\mathbf{y}$. The standard least-square solution is recovered for $\mathbf{W}=\mathbf{I}$. In order to fine-tune the reactiveness of the model, the weights must be a function that decreases as instances become older in the sample.

There is of course no perfect solution to changing financial environements. Below, we mention two routes that are taken in the ML literature to overcome the problem of non-stationarity in the data generating process. But first, we propose a clear verification that markets do experience time-varying distributions.

15.2.1 Non-stationarity: an obvious illustration

One of the most basic practices in (financial) econometrics is to work with returns (relative price changes). The simple reason is that returns seem to behave consistently through time (monthly returns are bounded, they usually lie between -1 and +1). Prices on the other hand shift and often, some prices never come back to past values. This makes prices harder to study.

Stationarity is a key notion in financial econometrics: it is much easier to characterize a phenomenon that remains the same through time. Sadly, the distribution of returns is not stationary: both the mean and the variance of returns change along cycles.

Below, we illustrate this fact by computing the average monthly return for all calendar years in the whole dataset.

```r
data_ml %>%
    mutate(year = year(date)) %>%          # Create a year variable
    group_by(year) %>%                     # Group by year
    summarize(avg_ret = mean(R1M_Usd)) %>% # Compute average return
    ggplot(aes(x = year, y = avg_ret)) + geom_col() + theme_grey()
```

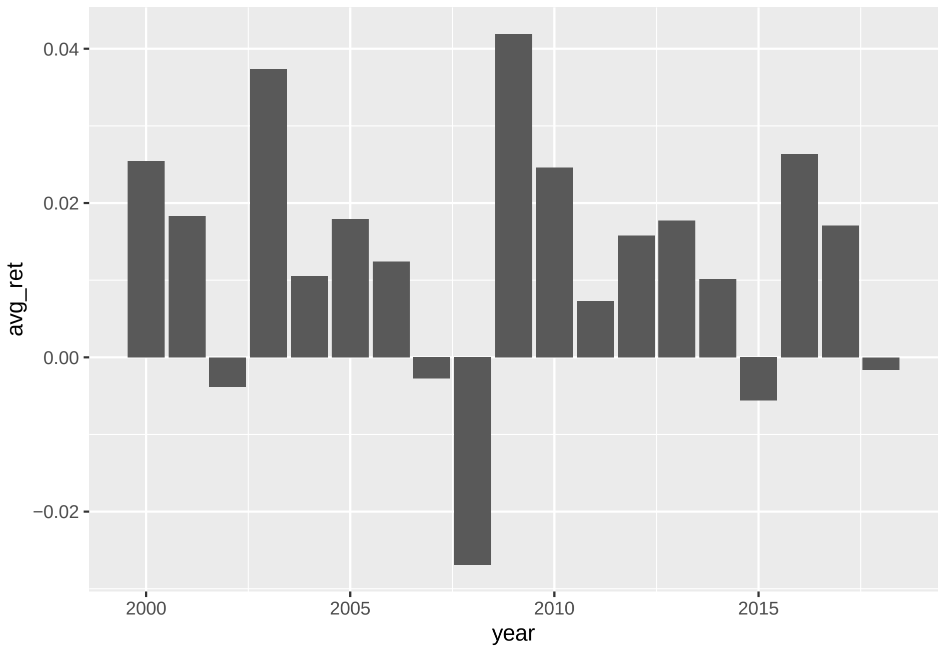


FIGURE 15.2: Average monthly return on a yearly basis.

These changes in the mean are also accompanied by variations in the second moment (variance/volatility). This effect, known as volatility clustering has been widely documented ever since the theoretical breakthrough of Engle (1982) (and even well before). We refer for instance to Cont (2007) for more details on this topic. For the computation of realized volatility in R, we strongly recommend Chapter 4 in Regenstein (2018).

In terms of machine learning models, this is also true. Below, we estimate a pure characteristic regression with one predictor, the market capitalization averaged over the past 6 months. The label is the 6 month forward return and the estimation is performed over every calendar year.

```
data_ml %>%
    mutate(year = year(date)) %>%                       # Create a year variable
    group_by(year) %>%                                  # Group by year
    summarize(beta_cap = lm(R6M_Usd ~ Mkt_Cap_6M_Usd) %>%   # Perform regression
                  coef() %>%                            # Extract coefs
                  t() %>%                               # Transpose
                  data.frame() %>%                      # Format into df
                  pull(Mkt_Cap_6M_Usd)) %>%             # Pull coef (remove intercept)
    ggplot(aes(x = year, y = beta_cap)) + geom_col() +  # Plot
    theme_grey()
```

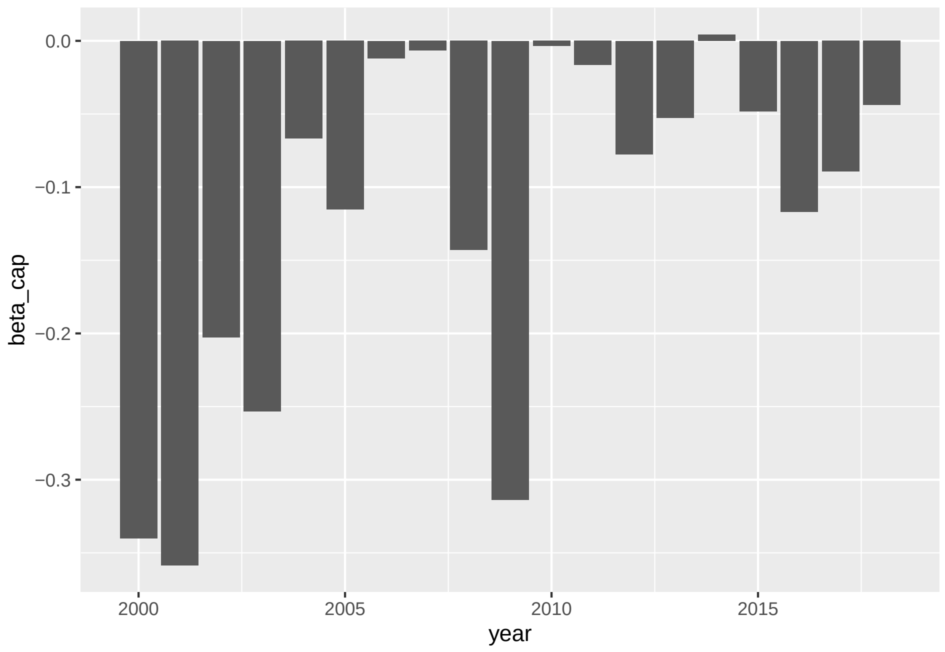


FIGURE 15.3: Variations in betas with respect to 6 month market capitalization

The figure highlights the concept drift: overall, the relationship between capitalization and returns is negative (the **size effect** again). Sometimes it is markedly negative, sometimes, not so much. The ability of capitalization to explain returns is time-varying and models must adapt accordingly.

15.2.2 Online learning

Online learning refers to a subset of machine learning in which new information arrives progressively and the integration of this flow is performed iteratively (the term '*online*' is not linked to internet). In order to take the latest data updates into account, it is imperative to update the model (stating the obvious). This is clearly the case in finance and this topic is closely related to the discussion on learning windows in Section 13.1.

The problem is that if a 2019 model is trained on data from 2010 to 2019, the (dynamic) 2020 model will have to be re-trained with the whole dataset including the latest points from 2020. This can be heavy and including just the latest points in the learning process would substantially decrease its computational cost. In neural networks, the sequential batch updating of weights can allow a progressive change in the model. Nonetheless, this is typically impossible for decision trees because the splits are decided once and for all. One notable exception is Basak (2004) but in that case, the construction of the trees differs strongly from the original algorithm (and spirit).

The simplest example of online learning is the Widrow-Hodd algorithm (originally from Widrow and Hoff (1960)). Originally, the idea comes from the so-called ADALINE (ADAptive LInear NEuron) model which is a neural network with one hidden layer with linear activation function (i.e., like a perceptron, but with a different activation).

Suppose the model is linear, that is $\mathbf{y} = \mathbf{X}\mathbf{b} + \mathbf{e}$ (a constant can be added to the list of predictors) and that the amount of data is both massive and coming in at a high frequency so that updating the model on the full sample is proscribed because technically intractable. A simple and heuristic way to update the values of $\mathbf{b}$ is to compute

$$\mathbf{b}_{t+1} \longleftarrow \mathbf{b}_t - \eta(\mathbf{x}_t\mathbf{b} - y_t)\mathbf{x}_t,$$

where $\mathbf{x}_t$ is the row vector of instance t. The justification is simple. The quadratic error $(\mathbf{x}_t\mathbf{b} - y_t)^2$ has a gradient with respect to $\mathbf{b}$ equal to $2(\mathbf{x}_t\mathbf{b} - y_t)\mathbf{x}_t$ therefore the above update is a simple example of gradient descent. η must of course be quite small: if not, each new point will considerably alter $\mathbf{b}$ resulting in a volatile model.

An exhaustive review of techniques pertaining to online learning is presented in Hoi et al. (2018) (section 4.11 is even dedicated to portfolio selection). The book Hazan et al. (2016) covers online convex optimization which is a very close domain with a large overlap with online learning. The presentation below is adapted from the second and third parts of the first survey.

Datasets are indexed by time: we write $\mathbf{X}_t$ and $\mathbf{y}_t$ for features and labels (the usual column index (k) and row index (i) will not be used in this section). Time has a bounded horizon T. The machine learning model depends on some parameters $\boldsymbol{\theta}$ and we denote it with $f_{\boldsymbol{\theta}}$. At time t (when dataset ($\mathbf{X}_t$, $\mathbf{y}_t$) is gathered), the loss function L fo the trained model naturally depends on the data ($\mathbf{X}_t$, $\mathbf{y}_t$) and on the model via $\boldsymbol{\theta}_t$ which are the parameter values fitted to the time-t data. For notational simplicity, we henceforth write $L_t(\boldsymbol{\theta}_t) = L(\mathbf{X}_t, \mathbf{y}_t, \boldsymbol{\theta}_t)$. The key quantity in online learning is the regret over the whole time sequence:

$$R_T = \sum_{t=1}^T L_t(\boldsymbol{\theta}_t) - \inf_{\boldsymbol{\theta}^* \in \boldsymbol{\Theta}} \sum_{t=1}^T L_t(\boldsymbol{\theta}^*). \tag{15.3}$$

The regret is the total loss incurred by the models $\boldsymbol{\theta}_t$ minus the minimal loss that could have obtained with full knowledge of the data sequence (hence computed in hindsight). The basic methods in online learning are in fact quite similar to the batch-training of neural networks. The updating of the parameter is based on

$$\mathbf{z}_{t+1} = \boldsymbol{\theta}_t - \eta_t \nabla L_t(\boldsymbol{\theta}_t), \tag{15.4}$$

where $\nabla L_t(\boldsymbol{\theta}_t)$ denotes the gradient of the current loss L_t. One problem that can arise is when $\mathbf{z}_{t+1}$ falls out of the bounds that are prescribed for $\boldsymbol{\theta}_t$. Thus, the candidate vector for the new parameters, $\mathbf{z}_{t+1}$, is projected onto the feasible domain which we call S here:

$$\boldsymbol{\theta}_{t+1} = \Pi_S(\mathbf{z}_{t+1}), \quad \text{with} \quad \Pi_S(\mathbf{u}) = \underset{\boldsymbol{\theta} \in S}{\text{argmin}}\, ||\boldsymbol{\theta} - \mathbf{u}||_2. \tag{15.5}$$

Hence $\boldsymbol{\theta}_{t+1}$ is as close as possible to the intermediate choice $\mathbf{z}_{t+1}$. In Hazan et al. (2007), it is shown that under suitable assumptions (e.g., L_t being strictly convex with bounded gradient $\left|\left|\sup_{\boldsymbol{\theta}} \nabla L_t(\boldsymbol{\theta})\right|\right| \le G$), the regret R_T satisfies

$$R_T \le \frac{G^2}{2H}(1 + \log(T)),$$

where H is a scaling factor for the learning rate (also called step sizes): $\eta_t = (Ht)^{-1}$.

More sophisticated online algorithms generalize (15.4) and (15.5) by integrating the Hessian matrix $\nabla^2 L_t(\boldsymbol{\theta}) := [\nabla^2 L_t]_{i,j} = \frac{\partial}{\partial \boldsymbol{\theta}_i \partial \boldsymbol{\theta}_j} L_t(\partial \boldsymbol{\theta})$ and/or by including penalizations to reduce instability in $\boldsymbol{\theta}_t$. We refer to Section 2 in Hoi et al. (2018) for more details on these extensions.

An interesting stream of parameter updating is that of the passive-aggressive algorithms (PAAs) formalized in Crammer et al. (2006). The base case involves classification tasks, but we stick to the regression setting below (section 5 in Crammer et al. (2006)). One strong limitation with PAAs is that they rely on the set of parameters where the loss is either zero or negligible: $\boldsymbol{\Theta}^*_\epsilon = \{\boldsymbol{\theta}, L_t(\boldsymbol{\theta}) < \epsilon\}$. For general loss functions and learner f, this set is largely inaccessible. Thus, the algorithms in Crammer et al. (2006) are restricted to a particular case, namely linear f and ϵ-insensitive hinge loss:

$$L_\epsilon(\boldsymbol{\theta}) = \begin{cases} 0 & \text{if} \quad |\boldsymbol{\theta}'\mathbf{x} - y| \le \epsilon \quad \text{(close enough prediction)} \\ |\boldsymbol{\theta}'\mathbf{x} - y| - \epsilon & \text{if} \quad |\boldsymbol{\theta}'\mathbf{x} - y| > \epsilon \quad \text{(prediction too far)} \end{cases},$$

for some parameter $\epsilon > 0$. If the weight $\boldsymbol{\theta}$ is such that the model is close enough to the true value, then the loss is zero, if not, it is equal to the absolute value of the error minus ϵ. In PAA, the update of the parameter is given by

$$\boldsymbol{\theta}_{t+1} = \underset{\boldsymbol{\theta}}{\text{argmin}} ||\boldsymbol{\theta} - \boldsymbol{\theta}_t||_2^2, \quad \text{subject to} \quad L_\epsilon(\boldsymbol{\theta}) = 0,$$

hence the new parameter values are chosen such that two conditions are satisfied:
- the loss is zero (by the definition of the loss, this means that the model is close enough to the true value);
- and, the parameter is as close a possible to the previous parameter values.

Hence, if the model is good enough, the model does not move (passive phase), but if not, it is rapidly shifted towards values that yield satisfactory results (aggressive phase).

We end this section with a historical note. Some of the ideas from online learning stem from the financial literature and from the concept of universal portfolios from Cover (1991) in particular. The setting is the following. The function f is assumed to be linear $f(\mathbf{x}_t) = \boldsymbol{\theta}'\mathbf{x}_t$ and the data $\mathbf{x}_t$ consists of asset returns, thus, the values are portfolio returns as long as $\boldsymbol{\theta}'\mathbf{1}_N = 1$ (the budget constraint). The loss functions L_t correspond to concave utility functions and the regret is reversed:

$$R_T = \sup_{\boldsymbol{\theta}^* \in \boldsymbol{\Theta}} \sum_{t=1}^T L_t(\mathbf{r}_t'\boldsymbol{\theta}^*) - \sum_{t=1}^T L_t(\mathbf{r}_t'\boldsymbol{\theta}_t),$$

where $\mathbf{r}'_t$ are the returns. Thus, the program is transformed to maximize a concave function. Several articles (often from the Computer Science or ML communities) have proposed solutions to this type of problems: Blum and Kalai (1999), Agarwal et al. (2006) and Hazan et al. (2007).

15.2.3 Homogeneous transfer learning

The next two subsections are mostly conceptual and will not be illustrated by coded applications. The ideas behind transfer learning and active learning can be valuable in that they can foster novel ideas.

Transfer learning has been surveyed numerous times. One classical reference is Pan and Yang (2009), but Weiss et al. (2016) is more recent and more exhaustive. Suppose we are given two datasets D_S (source) and D_T (target). Each dataset has its own features $\mathbf{X}^S$ and $\mathbf{X}^T$ and labels $\mathbf{y}^S$ and $\mathbf{y}^T$. In classical supervised learning, the patterns of the target set are learned only through $\mathbf{X}^T$ and $\mathbf{y}^T$. Transfer learning proposes to improve the function f^T (obtained by minimizing the fit $y_i^T = f^T(\mathbf{x}_i^T) + \epsilon_i^T$ on the target data) via the function f^S (from $y_i^S = f^S(\mathbf{x}_i^S) + \varepsilon_i^S$ on the source data). Homogeneous transfer learning is when the feature space does not change, which is the case in our setting. In asset management, this may not always be the case if for instance new predictors are included (e.g., based on alternative data like sentiment, satelitte imagery, credit card logs, etc.).

There are many subcategories in transfer learning depending on what changes between the source S and the target T: is it the feature space, the distribution of the labels, and/or the relationship between the two? This latter case is of interest in finance because the link with non-stationarity is evident: it is when the model f in $\mathbf{y} = f(\mathbf{X})$ changes through time. In transfer learning jargon, it is written as $P[\mathbf{y}^S|\mathbf{X}^S] \neq P[\mathbf{y}^T|\mathbf{X}^T]$: the conditional law of the label knowing the features is not the same when switching from the source to the target. Often, the term 'domain adaptation' is used as synonym to transfert learning. Because of a data shift, we must adapt the model to increase is accuracy. These topics are reviewed in a series of chapters in the collection by Quionero-Candela et al. (2009).

An important and elegant result in the theory was proven by Ben-David et al. (2010) in the case of binary classification. We state it below. We consider f and h two classifiers with values in $\{0,1\}$. The average error between the two over the domain S is defined by

$$\epsilon_S(f,h) = \mathbb{E}_S[|f(\mathbf{x}) - h(\mathbf{x})|].$$

Then,

$$\epsilon_T(f_T,h) \le \epsilon_S(f_S,h) + \underbrace{2\sup_B |P_S(B) - P_T(B)|}_{\text{difference between domains}} + \underbrace{\min\left(\mathbb{E}_S[|f_S(\mathbf{x}) - f_T(\mathbf{x})|], \mathbb{E}_T[|f_S(\mathbf{x}) - f_T(\mathbf{x})|]\right)}_{\text{difference between the two learning tasks}}.$$

The above inequality is a bound on the generalization performance of h. If we take f_S to be the best possible classifier for S and f_T the best for T, then the error generated by h in T is smaller than the sum of three components:
- the error in the S space;
- the distance between the two domains (by how much the data space has shifted);
- the ditance between the two best models (generators).

One solution that is often mentioned in transfer learning is instance weighting. We present it here in a general setting. In machine learning, we seek to minimize

$$\epsilon_T(f) = \mathbb{E}_T\left[L(\text{y}, f(\mathbf{X}))\right],$$

where L is some loss function that depends on the taks (regression versus classification). This can be arranged

$$\begin{aligned}\epsilon_T(f) &= \mathbb{E}_T\left[\frac{P_S(\mathbf{y},\mathbf{X})}{P_S(\mathbf{y},\mathbf{X})}L(\text{y}, f(\mathbf{X}))\right] \\ &= \sum_{\mathbf{y},\mathbf{X}} P_T(\mathbf{y},\mathbf{X})\frac{P_S(\mathbf{y},\mathbf{X})}{P_S(\mathbf{y},\mathbf{X})}L(\text{y}, f(\mathbf{X})) \\ &= \mathbb{E}_S\left[\frac{P_T(\mathbf{y},\mathbf{X})}{P_S(\mathbf{y},\mathbf{X})}L(\text{y}, f(\mathbf{X}))\right]\end{aligned}$$

The key quantity is thus the transition ratio $\frac{P_T(\mathbf{y},\mathbf{X})}{P_S(\mathbf{y},\mathbf{X})}$ (Radon–Nikodym derivative under some assumptions). Of course this ratio is largely inaccessible in practice, but it is possible to find a weighting scheme (over the instances) that yields improvements over the error in the target space. The weighting scheme, just as in Coqueret and Guida (2019) can be binary, thereby simply excluding some observations in the computation of the error.

More generally, the above expression can be viewed as a theoretical invitation for user-specified instance weighting (as in 7.4.7). In the asset allocation parlance, this can be viewed as introducing views so as to which observations are the most interesting, e.g., value stocks can be allowed to have a larger weight in the computation of the loss. Naturally, it remains to minimize this loss.

15.2.4 Active learning

We end this section with the notion of active learning. To the best of our knowledge, it is not widely used in quantitative investment, but the underlying concept is enlightening, hence we dedicate a few paragraphs to this notion for the sake of completeness.[2]

In general supervised learning, there is sometimes an asymmetry in the ability to gather features versus labels. For instance, it is free to have access to images, but the labelling of the content of the image (e.g., “a dog”, “a truck”, “a pizza”, etc.) is costly because it requires human annotation. In formal terms, $\mathbf{X}$ is cheap but the corresponding $\mathbf{y}$ is expensive.

As is often the case when facing cost constraints, a evident solution is greed. Ahead of the usual learning process, a filter (often called *query*) is used to decide which data to label and train on (possibly in relationship with the ML algorithm). The labelling is performed by a so-called *oracle* (which/who knows the truth) - usually human. This technique that focuses on the most informative instances is referred to as **active learning**. We refer to the surveys Settles (2009) and Settles (2012) for a detailed account of this field (which we briefly summarize below). The term **active** comes from the fact that the learner does not passively accept data samples but actively participates to the choices of items it learns from.

One major dichotomy in active learning pertains to the data source $\mathbf{X}$ on which the query is based. One obvious case is when the original sample $\mathbf{X}$ is very large and not labelled and

[2]As a matter of fact, active learning is not known to be particularly useful for learning from non-stationary environments.

the learner asks for particular instances within this sample to be labelled. The second case is when the learner has the ability to simulate/generate its own values $\mathbf{x}_i$. This can sometimes be problematic if the oracle does not recognize the data that is generated by the machine. For instance, if the purpose is to label images of characters and numbers, the learner mat generate shapes that do not correspond to any letter or digit: the oracle cannot label it.

In active learning, one key question is: how does the learner choose the instances to be labelled? Heuristically, the answer is: by picking those obsrevations that maximize learning efficiency. In binary classification, a simple criterion is the probability of belonging to one particular class. If this probability is far from 0.5, then the algorithm will have no difficulty of picking one class (even though it can be wrong). The interesting case is when the probability is close to 0.5: the machine may hesitate. Thus, having the oracle label is useful in this case because it helps the learner in a configuration it which it is undecided.

Other methods seek to estimate the fit that can be obtained when including particular (new) instances in the training set - and then to optimize this fit. Recalling Section 3.1 in Geman et al. (1992) on the variance-bias tradeoff, we have, for a training dataset D and one instance x (we omit bold font for simplicity),

$$\mathbb{E}\left[(y-\hat{f}(x;D))^2\middle|\{D,x\}\right]=\mathbb{E}\left[\underbrace{(y-\mathbb{E}[y|x])^2}_{\text{indep. from } D \text{ and } \hat{f}}\middle|\{D,x\}\right]+(\hat{f}(x;D)-\mathbb{E}[y|x])^2,$$

where the notation $f(x;D)$ is used to highlight the dependence between the model $\hat{f}$ and the dataset D: the model has been trained on D. The first term is irreducible, as it does not depend on $\hat{f}$. Thus, only the second term is of interest. If we take the average of this quantity, taken over all possible values of D:

$$\mathbb{E}_D\left[(\hat{f}(x;D)-\mathbb{E}[y|x])^2\right]=\underbrace{\left(\mathbb{E}_D\left[\hat{f}(x;D)-\mathbb{E}[y|x]\right]\right)^2}_{\text{squared bias}}+\underbrace{\mathbb{E}_D\left[(\hat{f}(x,D)-\mathbb{E}_D[\hat{f}(x;D)])^2\right]}_{\text{variance}}$$

If this expression is not too complicated to compute, the learner can query the x that minimizes the tradeoff. Thus, on average, this new instance will be the one that yields the best learning angle (as measured by the L^2 error). Beyond this approach (which is limited because it requires the oracle to label a possibly irrelevant instance), many other criteria exist for querying and we refer to Section 3 from Settles (2009) for an exhaustive list.

One final question: is active learning applicable to factor investing? One straightfoward answer is that data cannot be annotated by human intervention. Thus, the learner cannot simulate its own instances and ask for corresponding labels. One possible option is to provide the learner with $\mathbf{X}$ but not $\mathbf{y}$ and keep only a queried subset of observations with the corresponding labels. In spirit, this is close to what is done in Coqueret and Guida (2019) except that the query is not performed by a machine but by the human user. Indeed, it is shown in this paper that not all observations carry the same amount of signal. Instances with 'average' label values seem less informative compared to those with extreme label values.

16
Unsupervised learning

All algorithms presented in Chapters 6 to 10 belong to the larger class of supervised learning tools. Such tools seek to unveil a mapping between predictors $\mathbf{X}$ and a label $\mathbf{y}$. The supervision comes from the fact that it is asked that the data tries to explain this particular variable $\mathbf{y}$. Another important part of machine learning consists of unsupervised tasks, that is, when $\mathbf{y}$ is not specified and the algortithm tries to make sense of $\mathbf{X}$ on its own. Often, relationships between the components of $\mathbf{X}$ are identified. This field is much to vast to be summarized in one book, let alone one chapter. The purpose here is to briefly explain in what ways unsupervised learning can be used, especially in the data pre-processing phase.

16.1 The problem with correlated predictors

Often, it is tempting to supply all predictors to a ML-fueled predictive engine. That may not be a good idea when some predictors are highly correlated. To illustrate this, the simplest example is a regression on two variables with zero mean and covariance and precisions matrices:

$$\mathbf{\Sigma}=\mathbf{X}'\mathbf{X}=\begin{bmatrix}1 & \rho \\ \rho & 1\end{bmatrix}, \quad \mathbf{\Sigma}^{-1}=\frac{1}{1-\rho^2}\begin{bmatrix}1 & -\rho \\ -\rho & 1\end{bmatrix}.$$

When the covariance/correlation ρ increase towards 1, the scaling denominator in $\mathbf{\Sigma}$ goes to zero and the formula $\hat{\boldsymbol{\beta}}=\mathbf{\Sigma}^{-1}\mathbf{X}'\mathbf{y}$ implies that one coefficient will be highly positive and one highly negative. The regression creates a spurious arbitrage between the two variables. Of course, this is very inefficient a yields disastrous results out-of-sample.

We illustrate what happens when many variables are used in the regression below (Table 16.1). One elucidation of the aforementioned phenomenon comes from the variables Mkt_Cap_12M_Usd and Mkt_Cap_6M_Usd, which have a correlation of 99.6% in the training sample. Both are singled out as highly significant but their signs are contradictory. Moreover, the magnitude of their coefficients are very close (0.21 versus 0.18) so that their net effect cancels out. Naturally, providing the regression with only one of these two input would have been wiser.

```
library(broom)                              # Package for clean regression output
training_sample %>%
    select(c(features,  "R1M_Usd")) %>%     # List of variables
    lm(R1M_Usd ~ . , data = .) %>%          # Model: predict R1M_Usd
    tidy() %>%                              # Put output in clean format
    filter(abs(statistic) > 3)  %>%         # Keep highly significant predictors only
    knitr::kable(booktabs = TRUE,
                 caption = "Significant predictors in the training sample.")
```

TABLE 16.1: Significant predictors in the training sample.

term	estimate	std.error	statistic	p.value
(Intercept)	0.0405741	0.0053427	7.594323	0.0000000
Ebitda_Margin	0.0132374	0.0034927	3.789999	0.0001507
Ev_Ebitda	0.0068144	0.0022563	3.020213	0.0025263
Fa_Ci	0.0072308	0.0023465	3.081471	0.0020601
Fcf_Bv	0.0250538	0.0051314	4.882465	0.0000010
Fcf_Yld	-0.0158930	0.0037359	-4.254126	0.0000210
Mkt_Cap_12M_Usd	0.2047383	0.0274320	7.463476	0.0000000
Mkt_Cap_6M_Usd	-0.1797795	0.0459390	-3.913443	0.0000910
Mom_5M_Usd	-0.0186690	0.0044313	-4.212972	0.0000252
Mom_Sharp_11M_Usd	0.0178174	0.0046948	3.795131	0.0001476
Ni	0.0154609	0.0044966	3.438361	0.0005854
Ni_Avail_Margin	0.0118135	0.0038614	3.059359	0.0022184
Ocf_Bv	-0.0198113	0.0052939	-3.742277	0.0001824
Pb	-0.0178971	0.0031285	-5.720637	0.0000000
Pe	-0.0089908	0.0023539	-3.819565	0.0001337
Sales_Ps	-0.0157856	0.0046278	-3.411062	0.0006472
Vol1Y_Usd	0.0114250	0.0027923	4.091628	0.0000429
Vol3Y_Usd	0.0084587	0.0027952	3.026169	0.0024771

In fact, since there are several indicators for the market capitalization and maybe only one would suffice, but it is not obvious to tell which one is the best choice.

In the remainder of the chapter, we present two approaches that help reduce the number of predictors:
- the first one aims at creating new variables that are uncorrelated with each other. Low correlation is favorable from an algorithmic point of view, but the new variables lack interpretability;
- the second one gathers predictors into homogeneous clusters and only one feature should be chosen out of this cluster. Here the rationale is reversed: interpretability is favored over statistical properties because the resulting set of features may still include high correlations, albeit to a lesser point compared to the original one.

16.2 Principal component analysis and autoencoders

The first method is a cornerstone in dimensionality reduction. It seeks to determine a smaller number of factors ($K' < K$) such that:
i) the level of explanatory power remains as high as possible;
ii) the resulting factors are linear combinations of the original variables;
iii) the resulting factors are orthogonal.

16.2.1 A bit of algebra

In this short subsection, we define some key concepts that are required to fully understand the derivation of PCA. Henceforth, we work with matrices (in bold fonts). An $I \times K$ matrix $\mathbf{X}$ is orthonormal if $I > K$ and $\mathbf{X}'\mathbf{X} = \mathbf{I}_K$. When $I = K$, the (square) matrix is called orthogonal and $\mathbf{X}'\mathbf{X} = \mathbf{X}\mathbf{X}' = \mathbf{I}_K$, i.e., $\mathbf{X}^{-1} = \mathbf{X}'$.

One foundational result in matrix theory is the Singular Value Decomposition (SVD, see, e.g., Chapter 5 in Meyer (2000)). The SVD is formulated as follows: any $I \times K$ matrix $\mathbf{X}$ can be decomposed into

$$\mathbf{X} = \mathbf{U}\boldsymbol{\Delta}\mathbf{V}', \tag{16.1}$$

where $\mathbf{U}$ $(I \times I)$ and $\mathbf{V}$ $(K \times K)$ are orthogonal and $\boldsymbol{\Delta}$ $(I \times K)$ is diagonal, i.e. $\Delta i, k = 0$ whenever $i \neq k$. In addition, $\Delta i, i \geq 0$: the diagonal terms of $\boldsymbol{\Delta}$ are nonnegative.

For simplicity, we assume below that $\mathbf{1}_I'\mathbf{X} = \mathbf{0}_K'$, i.e., that all columns have zero sum (and hence zero mean).[1] This allows to write that the covariance matrix is equal to its sample estimate $\boldsymbol{\Sigma}_X = \frac{1}{I-1}\mathbf{X}'\mathbf{X}$.

One crucial feature of covariance matrices is their symmetry. Indeed, real-valued symmetric (square) matrices enjoy a SVD which is much more powerful: when $\mathbf{X}$ is symmetric, there exist an orthogonal matrix $\mathbf{Q}$ and a diagonal matrix $\mathbf{D}$ such that

$$\mathbf{X} = \mathbf{Q}\mathbf{D}\mathbf{Q}'. \tag{16.2}$$

This process is called diagonalization (see Chapter 7 in Meyer (2000)) and conveniently applies to covariance matrices.

16.2.2 PCA

The goal of PCA is to build a dataset $\tilde{\mathbf{X}}$ that has fewer columns but that keeps as much information compared to the original one, $\mathbf{X}$. The key notion is the change of base, which is a linear transformation of $\mathbf{X}$ into $\mathbf{Y}$, a matrix with identical dimension, via

$$\mathbf{Y} = \mathbf{X}\mathbf{P},$$

where P is a $K \times K$ matrix. There are of course an infinite number of ways to transform $\mathbf{X}$ into $\mathbf{Y}$, but two fundamental constraints help reduce the possibilities. The first constraint is that the columns of $\mathbf{Y}$ be uncorrelated. Having uncorrelated features is desirable because they then all tell different stories and have zero redundancy. The second constraint is that the variance of the columns of $\mathbf{Y}$ is highly concentrated. This means that a few factors (columns) will capture most of the explanatory power (signal) while most (the others) will consist predominantly of noise. All of this is coded in the covariance matrix of Y:
- the first condition imposes that the covariance matrix be diagonal;
- the second condition imposes that the diagonal elements, when ranked in decreasing magnitude, see their value decline (sharply if possible).

The covariance matrix of $\mathbf{Y}$ is

$$\boldsymbol{\Sigma}_Y = \frac{1}{I-1}\mathbf{Y}'\mathbf{Y} = \frac{1}{I-1}\mathbf{P}'\mathbf{X}'\mathbf{X}\mathbf{P} = \frac{1}{I-1}\mathbf{P}'\boldsymbol{\Sigma}_X\mathbf{P} \tag{16.3}$$

[1] In practice, this is not a major problem: since we work with features that are uniformly distributed, de-meaning amounts to remove 0.5 to all feature values.

In this expression, we plug the decomposition (16.2) of $\mathbf{\Sigma}_X$:

$$\mathbf{\Sigma}_Y = \frac{1}{I-1}\mathbf{P}'\mathbf{Q}\mathbf{D}\mathbf{Q}'\mathbf{P},$$

thus picking $\mathbf{P}=\mathbf{Q}$, we get $\mathbf{\Sigma}_Y = \frac{1}{I-1}\mathbf{D}$, that is, a diagonal covariance matrix for $\mathbf{Y}$. The columns of $\mathbf{Y}$ can then be re-shuffled in decreasing order of variance so that the diagonal elements of $\mathbf{\Sigma}_Y$ progressively shrink. This is useful because it helps locate the factors with most informational content (the first factors). In the limit, a constant vector (with zero variance) carries no signal.

The matrix $\mathbf{Y}$ is a linear transformation of $\mathbf{X}$, thus, it is expected to carry the same information, even though this information is coded differently. Since the columns are ordered according to their relative importance, it is simple to omit some of them. The new set of features $\tilde{\mathbf{X}}$ consists in the first K' (with $K'<K$) columns of $\mathbf{Y}$.

Below, we show how to perform PCA and visualize the output with the *factoextra* package. To ease readability, we use the smaller sample with few predictors.

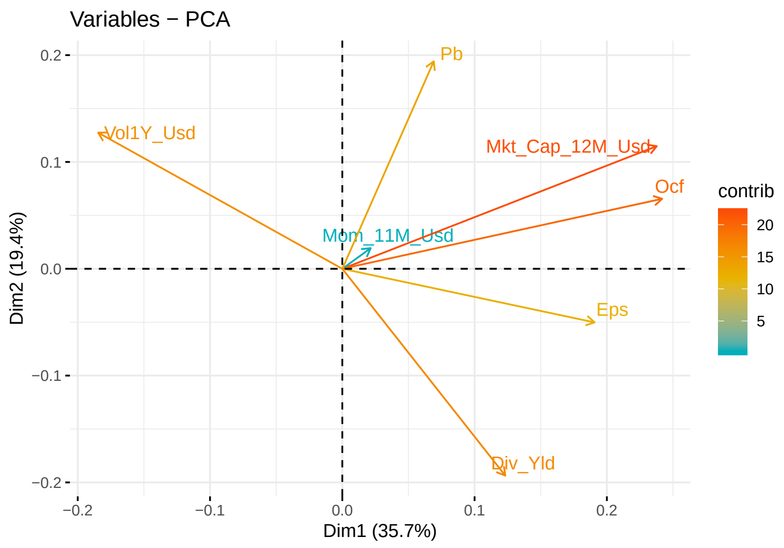
```
pca <- training_sample %>%
    select(features_short) %>%     # Smaller number of predictors
    prcomp()                       # Performs PCA
pca                                # Show the result
```

```
## Standard deviations (1, .., p=7):
## [1] 0.4536601 0.3344080 0.2994393 0.2452000 0.2352087 0.2010782 0.1140988
##
## Rotation (n x k) = (7 x 7):
##                        PC1         PC2         PC3         PC4
## Div_Yld         0.27159946 -0.57909866  0.04572501 -0.52895604
## Eps             0.42040708 -0.15008243 -0.02476659  0.33737265
## Mkt_Cap_12M_Usd  0.52386846  0.34323935  0.17228893  0.06249528
## Mom_11M_Usd     0.04723846  0.05771359 -0.89715955  0.24101481
## Ocf             0.53294744  0.19588990  0.18503939  0.23437100
## Pb              0.15241340  0.58080620 -0.22104807 -0.68213576
## Vol1Y_Usd      -0.40688963  0.38113933  0.28216181  0.15541056
##                        PC5          PC6          PC7
## Div_Yld        -0.22662581 -0.506566090  0.032011635
## Eps             0.77137719 -0.301883295  0.011965041
## Mkt_Cap_12M_Usd -0.25278113 -0.002987057  0.714319417
## Mom_11M_Usd    -0.25055884 -0.258476580  0.043178747
## Ocf            -0.35759553 -0.049015486 -0.676866120
## Pb              0.30866476 -0.038674594 -0.168799297
## Vol1Y_Usd      -0.06157461 -0.762587677  0.008632062
```

The rotation gives the matrix $\mathbf{P}$: it's the tool that changes the base. The first row of the output indicates the standard deviation of each new factor (column). Each factor is indicated via a PC index (principal component). Often, the first PC loads positively on all initial columns: a convex weighted average of all features is expected to carry a lot of information. In the above example, it is almost the case, with the exception of volatility, which has a negative coefficient in the first PC.

Sometimes, it can be useful to visualize the way the principal components are builts. We show one popular representation that is used for two factors (usually the first two).

```
library(factoextra)                      # Package for PCA visualization
fviz_pca_var(pca,                        # Source of PCA decomposition
             col.var="contrib",
             gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07"),
             repel = TRUE                # Avoid text overlapping
)
```



The plot shows that no initial factor has negative signs for the first two principal components. Volatility is negative for the first one and earnings per share and dividend yield are negative for the second. The numbers indicated along the axes are the proportion of explained variance of each PC. Compared to the figures in the first line of the output, the numbers are squared and then divided by the total sum of squares.

Once the rotation is known, it is possible to select a subsample of the transformed data. From the original 7 features, it is easy to pick just 4.

```
training_sample %>%                                    # Start from large sample
    dplyr::select(features_short) %>%                  # Keep only 7 features
    as.matrix() %>%                                    # Transform in matrix
    multiply_by_matrix(pca$rotation[,1:4]) %>%         # Rotate via PCA (first 4 columns of P)
    `colnames<-`(c("PC1", "PC2", "PC3", "PC4")) %>%    # Change column names
    head()                                             # Show first 6 lines
```

```
##            PC1       PC2         PC3       PC4
## [1,] 0.3989674 0.7578132 -0.13915223 0.3132578
## [2,] 0.4284697 0.7587274 -0.40164338 0.3745255
## [3,] 0.5215295 0.5679119 -0.10533870 0.2574949
## [4,] 0.5445359 0.5335619 -0.08833864 0.2281793
## [5,] 0.5672644 0.5339749 -0.06092424 0.2320938
## [6,] 0.5871306 0.6420126 -0.44566482 0.3075399
```

These 4 factors can then be used as orthogonal features in any ML engine. The fact that the features are uncorrelated is undoubtedly an asset. But the price of this convenience is high: the features are no longer immediately interpretable. De-correlating the predictors adds yet another layer of "*blackbox-ing*" in the algorithm.

16.2.3 Autoencoders

In a PCA, the coding from $\mathbf{X}$ to $\mathbf{Y}$ is straightfoward, linear and works both ways:

$$\mathbf{Y} = \mathbf{XP} \quad \text{and} \quad \mathbf{X} = \mathbf{YP}',$$

so that we recover $\mathbf{X}$ from $\mathbf{Y}$. This can be writen differently:

$$\mathbf{X} \quad \overset{\text{encode via } \mathbf{P}}{\longrightarrow} \quad \mathbf{Y} \quad \overset{\text{decode via } \mathbf{P}'}{\longrightarrow} \quad \mathbf{X} \tag{16.4}$$

If we take the truncated version and seek a smaller output (with only K' columns), this gives:

$$\mathbf{X}, \ (I \times K) \quad \overset{\text{encode via } \mathbf{P}_{K'}}{\longrightarrow} \quad \tilde{\mathbf{X}}, \ (I \times K') \quad \overset{\text{decode via } \mathbf{P}'_{K'}}{\longrightarrow} \quad \breve{\mathbf{X}}, \ (I \times K), \tag{16.5}$$

where $\mathbf{P}_{K'}$ is the restriction of $\mathbf{P}$ to the K' columns that correspond to the factors with the largest variances. The dimensions of matrices are indicated inside the brackets. In this case, the recoding cannot recover $\mathbf{P}$ exactly but only an approximation, which we write $\breve{\mathbf{X}}$. This approximation is coded with less information, hence this new data $\breve{\mathbf{X}}$ is compressed and provides a parcimonious representation of the original sample $\mathbf{X}$.

An autoencodeur generalizes this concept to nonlinear coding functions. Simple linear autoencoders are linked to latent factor models (see Proposition 1 in Gu et al. (2019) for the case of single layer autoencoders.) The scheme is the following

$$\mathbf{X}, \ (I \times K) \quad \overset{\text{encode via } N}{\longrightarrow} \quad \tilde{\mathbf{X}} = N(\mathbf{X}), \ (I \times K') \quad \overset{\text{decode via } N'}{\longrightarrow} \quad \breve{\mathbf{X}} = N'(\tilde{\mathbf{X}}), \ (I \times K), \tag{16.6}$$

where the encoding and decoding functions N and N' are often taken to be neural networks. The term **auto-encoder** comes from the fact that the target output, which we often write $\mathbf{Y}$ is the original sample $\mathbf{X}$. Thus, the algorithm seek to determine the function N that minimizes the distance (to be defined) between $\mathbf{X}$ and the output value $\breve{\mathbf{X}}$. The encoder generates an alternative representation of $\mathbf{X}$ while the decoder tries to recode it back to its original values. Naturally, the intermediate (coded) version $\tilde{\mathbf{X}}$ is targeted to have a smaller dimension compared to $\mathbf{X}$.

16.2.4 Application

Auto-encoders are easy to code in Keras. To underline the power of the framework, we resort to another way of coding a NN: the so-called functional API. For simplicity, we work with the small number of predictors (7). The structure of the network consists of two symmetric networks with only one intermediate layer containing 32 units. The activation function is sigmoid: this makes sense since the input have values in the unit interval.

```
input_layer <- layer_input(shape = c(7))    # features_short has 7 columns

encoder <- input_layer %>%       # First, encode
    layer_dense(units = 32, activation = "sigmoid") %>%
    layer_dense(units = 4)       # 4 dimensions for the output layer (same as PCA example)

decoder <- encoder %>%           # Then, from encoder, decode
    layer_dense(units = 32, activation = "sigmoid") %>%
    layer_dense(units = 7)       # the original sample has 7 features
```

In the training part, we optimize the MSE and use an Adam update of the weights (see Section 8.2.3).

```
ae_model <- keras_model(inputs = input_layer, outputs = decoder) # Builds the model

ae_model %>% compile(                # Learning parameters
    loss = 'mean_squared_error',
    optimizer = 'adam',
    metrics = c('mean_absolute_error')
)
```

Finally, we are ready to train the data onto itself!

```
fit_ae <- ae_model %>%
    fit(training_sample %>% select(features_short) %>% as.matrix(),  # Input
        training_sample %>% select(features_short) %>% as.matrix(),  # Output
        epochs = 15, batch_size = 512,
        validation_data = list(testing_sample %>% select(features_short) %>% as.matrix(),
                               testing_sample %>% select(features_short) %>% as.matrix())
    )
plot(fit_ae) + theme_grey()
```

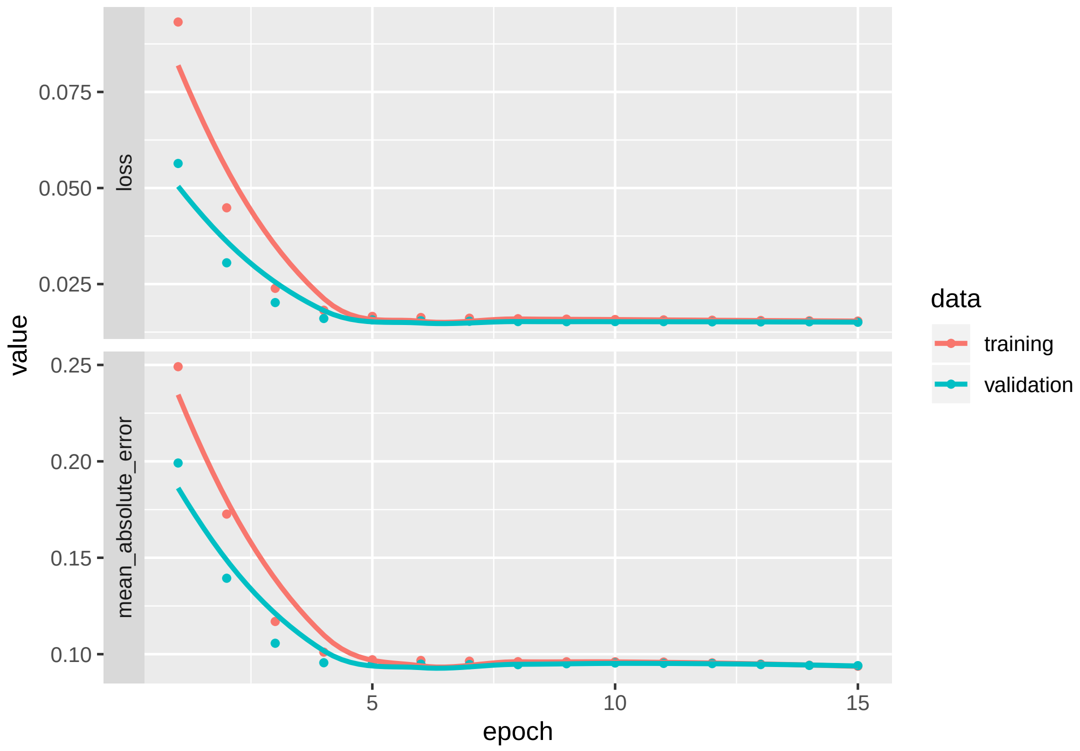


FIGURE 16.1: Output from the training of an auto-encoder.

In order to get the details of all weights and biases, the syntax is the following.

```
ae_weights <- ae_model %>% get_weights()
```

Retrieving the encoder and processing the data into the compressed format is just a matter of matrix manipulation (see exercise below).

16.3 Clustering via k-means

The second family of unsupervised tools pertains to clustering. Features are grouped into homogeneous families of predictors. It is then possible to single out one among the group (or to create a synthetic average of all of them). Mechanically, the number of predictors is reduced.

The principle is simple: among a group of variables (the reasoning would be the same for observations in the other dimension) $\mathbf{x}_{\{1\le j\le J\}}$, find the combination of $k<J$ groups that minimize

$$\sum_{i=1}^{k}\sum_{\mathbf{x}\in S_i}||\mathbf{x}-\mathbf{m}_i||^2, \tag{16.7}$$

where $||\cdot||$ is some norm which is usually taken to be the Euclidean l^2-norm. The S_i are the groups and the minimization is run on the whole set of groups $\mathbf{S}$. The $\mathbf{m}_i$ are the group means (also called centroids or barycenters): $\mathbf{m}_i=(\text{card}(S_i))^{-1}\sum_{\mathbf{x}\in S_i}\mathbf{x}$.

In order to ensure optimality, all possible arrangements must be tested, which is prohibitively long when k and J are large. Therefore, the problem is usually solved with greedy algorithms that seek (and find) solutions that are not optimal but 'good enough'.

One heuristic way to proceed is the following:

0. Start with a (possibly random) partition of k clusters.
1. For each cluster, compute the optimal mean values $\mathbf{m}_i^*$ that minimizes expression (16.7). This is a simple quadratic program.
2. Given the optimal centers $\mathbf{m}_i^*$, reassign the points $\mathbf{x}_i$ so that they are all the closest to their center.
3. Repeat steps 1. and 2. until the points do not change cluster at step 2.

Below, we illustrate this process with an example. From all 93 features, we build 10 clusters.

```
set.seed(42)                              # Setting the random seed (the optimization is random)
k_means <- training_sample %>%            # Performs the k-means clustering
    select(features) %>%
    as.matrix() %>%
    t() %>%
    kmeans(10)
clusters <- tibble(factor = names(k_means$cluster),    # Organize the cluster data
                   cluster = k_means$cluster) %>%
    arrange(cluster)
clusters %>% filter(cluster == 4)                      # Shows one particular group
```

```
## # A tibble: 20 x 2
##    factor                         cluster
##    <chr>                              <int>
##  1 Asset_Turnover                         4
##  2 Bb_Yld                                 4
##  3 Ebit_Bv                                4
##  4 Ebit_Noa                               4
##  5 Ebit_Oa                                4
##  6 Ebit_Ta                                4
##  7 Eps                                    4
##  8 Eps_Basic                              4
##  9 Eps_Basic_Gr                           4
## 10 Eps_Contin_Oper                        4
## 11 Eps_Dil                                4
## 12 Ni_Oa                                  4
## 13 Op_Prt_Margin                          4
## 14 Pb                                     4
## 15 Recurring_Earning_Total_Assets         4
## 16 Return_On_Capital                      4
## 17 Roa                                    4
## 18 Roc                                    4
## 19 Roce                                   4
## 20 Roe                                    4
```

We single out the fourth cluster which is composed mainly of accounting ratios related to the profitability of firms. Given these 10 clusters, we can build a much smaller group of features that can then be fed to the predictive engines described in Chapters 6 to 10. The representative of a cluster can be the member that is closest to the center, or simply the center itself. This pre-processing step can nonetheless cause problems in the forecasting phase. Typically, it requires that the training data be also clustered.

16.4 Nearest neighbors

To the best of our knowledge, nearest neighbors are not used in large scale portfolio choice applications. The reason is simple: computational cost. Nonetheless, the concept of neighbors is widespread in unsupervised learning and can be used locally in complement to interpretability tools.

In what follows, we seek to find neighbors of one particular instance $\mathbf{x}_i$ (a K-dimensional row vector). Note that there is a major difference with the previous section: the clustering is intended at the observation level (row) and not at the predictor level (column).

Given a dataset with the same (corresponding) columns $\mathbf{X}_{i,k}$, the neighbors are defined via a similarity measure (or distance)

$$D(\mathbf{x}_j,\mathbf{x}_i)=\sum_{k=1}^K c_k d_k(x_{j,k},x_{i,k}), \tag{16.8}$$

where the distances functions d_k can operate on various data types (numerical, categorical, etc.). For numerical values, $d_k(x_{j,k},x_{i,k})=(x_{j,k}-x_{i,k})^2$ or $d_k(x_{j,k},x_{i,k})=|x_{j,k}-x_{i,k}|$. For categorical values, we refer to the exhaustive survey by Boriah et al. (2008) which lists 14 possible measures. Finally the c_k in Equation (16.8) allow some flexbility by weighting features. This is useful because both raw values ($x_{i,k}$ versus $x_{i,k'}$) or measure outputs (d_k versus $d_{k'}$) can have different scales.

Once the distances are computed over the whole sample, they are ranked using indices $l_1^i, \dots, l_I^i$:

$$D\left(\mathbf{x}_{l_1^i}, \mathbf{x}_i\right) \le D\left(\mathbf{x}_{l_2^i}, \mathbf{x}_i\right) \le \dots, \le D\left(\mathbf{x}_{l_I^i}, \mathbf{x}_i\right)$$

The nearest neighbors are those indexed by l_m^i for $m = 1, \dots, k$. We leave out the case when there are problematic equalities of the type $D\left(\mathbf{x}_{l_m^i}, \mathbf{x}_i\right) = D\left(\mathbf{x}_{l_{m+1}^i}, \mathbf{x}_i\right)$ for the sake of simplicity and because they rarely occur in practice as long as there are sufficiently many numerical predictors.

Given these neighbors, it is now possible to build a prediction for the label side y_i. The rationale is straightforward: if $\mathbf{x}_i$ is close to other instances $\mathbf{x}_j$ then the label value y_i should also be close to y_j (under the assumption that the features carry some predictive information over the label y).

An intuitive prediction for y_i is the following weighted average:

$$\hat{y}_i = \frac{\sum_{j \neq i} h(D(\mathbf{x}_j, \mathbf{x}_i)) y_j}{\sum_{j \neq i} h(D(\mathbf{x}_j, \mathbf{x}_i))},$$

where h is a decreasing function. Thus, the further $\mathbf{x}_j$ is from $\mathbf{x}_i$, the smaller the weight in the average. A typical choice for h is $h(z) = e^{-az}$ for some parameter $a > 0$ that determines how penalizing the distance $D(\mathbf{x}_j, \mathbf{x}_i)$ is. Of course, the average can be taken in the set of k nearest neighbors, in which case the h is equal to zero beyond a particular distance threshold:

$$\hat{y}_i = \frac{\sum_{j \text{ neighbor}} h(D(\mathbf{x}_j, \mathbf{x}_i)) y_j}{\sum_{j \text{ neighbor}} h(D(\mathbf{x}_j, \mathbf{x}_i))}.$$

A more agnostic rule is to take $h := 1$ over the set of neighbors and in this case, all neighbors have the same weight (see the old discussion by Bailey and Jain (1978) in the case of classification). For classification tasks, the procedure involves a voting rule whereby the class with the most votes wins the contest, with possible tie-breaking methods. The interested reader can have a look at the short survey Bhatia et al. (2010).

For the choice of optimal k, several complicated techniques and criteria exists (see, e.g., Ghosh (2006) and Hall et al. (2008)). Heuristic values often do the job pretty well. A rule of thumb is that $k = \sqrt{I}$ (I being the total number of instances) is not too far from the optimal value, unless I is exceedingly large.

Below, we illustrate this concept. We pick one date (31th of December 2006) and single out one asset (with stock_id equal to 13). We then seek to find the $k = 30$ stocks that are the closest to this asset at this particular date. We resort to the *FNN* package that propose an efficient computation of Euclidean distances (and their ordering).

```
library(FNN)     # Package for Fast Nearest Neighbors detection
knn_data <- filter(data_ml, date == "2006-12-31")     # Dataset for k-NN exercise
knn_target <- filter(knn_data, stock_id == 13) %>%   # Target observation
              select(features)
knn_sample <- filter(knn_data, stock_id != 13) %>%   # All other observations
              select(features)
neighbors <- get.knnx(data = knn_sample, query = knn_target, k = 30)
neighbors$nn.index                                   # Indices of the k nearest neighbors
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
## [1,]  905  876  730  548 1036  501  335  117  789    54   618   130   342
##      [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24]
## [1,]   360   673   153   265   858   830   286  1150   166   946   192
##      [,25] [,26] [,27] [,28] [,29] [,30]
## [1,]   340   162   951   376   785     2
```

Once the neighbors and distances are known, we can compute a prediction for the return of the target stock. We use the function $h(z) = e^{-z}$ for the weighting of instances (via the distances).

```
knn_labels <- knn_data[neighbors$nn.index,] %>% select(R1M_Usd)          # y values for the neighbors
sum(knn_labels * exp(-neighbors$nn.dist) / sum(exp(-neighbors$nn.dist)))  # Prediction for k(z)=exp(-z)
```

```
## [1] 0.003042282
```

```
filter(knn_data, stock_id == 13) %>%                                      # True y (to check the error)
              select(R1M_Usd)
```

```
## # A tibble: 1 x 1
##   R1M_Usd
##     <dbl>
## 1   0.089
```

The prediction is neither very good, nor very bad (the sign is correct!). However, note that this example cannot be used for predictive purposes because we use data from 2006-12-31 to predict a return at the same date. In order to avoid the forward looking bias, the knn_sample variable should be chosen from a prior point in time.

The above computations are fast (a handful of seconds at most), but hold for only one asset. In a k-NN exercise each stock gets a customed prediction and the set of neighbors must be re-assessed each time. This is particularly costly in a backtest, especially when several parameters can be tested (k, a in the weighting function $h(z) = e^{-az}$).

16.5 Coding exercise

1. Code the compressed version of the data via the autoencoder.

17

Reinforcement learning

Due to its increasing popularity within the Machine Learing community, we dedicate a chapter to reinforcement learning (RL). In 2019 only, more than 25 papers dedicated to RL have been submitted to (or updated on) arXiv under the **q:fin** (quantitative finance) classification. Moreover, an early survey of RL-based portfolios is compiled in Sato (2019) and general financial applications are discussed in Kolm and Ritter (2019b) and Meng and Khushi (2019). This shows that RL has recently gained traction among the quantitative finance community.[1]

XXXXXXXXX

While RL is a framework more than a particular algorithm, its efficient application in portfolio management is not straightforward, as we will show.

17.1 Theoretical layout

17.1.1 General framework

In this section, we introduce the core concepts of RL and follow relatively closely the notations (and layout) of Sutton and Barto (2018), which is widely considered as a solid reference in the field, along with Bertsekas (2017). One central tool in the field is called the Markov Decision Process (MDP, see Chapter 3 in Sutton and Barto (2018)).

MDPs involve the interaction between an agent (e.g., a trader or portfolio manager) and an environment (e.g., a financial market). The agent performs actions that may alter the state of environment and gets a reward (possibly negative) for each action. This short sequence can be repeated an arbitrary number of times, as is shown in Figure 17.1.

Given initialized values for the state of the environement (S_0) and reward (usually $R_0 = 0$), the agent performs an action (e.g., invests in some assets). This generates future rewards R_1 (e.g., returns, profits, Sharpe ratio) and also a future state of the environment (S_1). Based on that, the agent performs and new action and the sequence continues. When the sets of states, actions and rewards are finite, the MDP is logically called *finite*. In a financial framework, this is somewhat unrealistic and we discuss this issue later on. It nevertheless is not hard to think of simplified and discretized financial problems. For instance, the reward can be binary: win money versus lose money. In the case of only one asset, the action can also be dual: investing versus not investing. When the number of assets is sufficiently small,

[1]Like neural networks, reinforcement learning methods have also been recently developed for derivatives pricing and hedging, see for instance Kolm and Ritter (2019a).

it is possible to set fixed proportions that lead to a reasonable number of combinations of portfolio choices, etc.

We pursue our exposé with finite MDPs (they are the most common in the literature and their formal treatment is simpler). As is often the case with markovian objects, the key notion is that of transition probability:

$$p(s',r|s,a)=\mathbb{P}\left[S_t=s',R_t=r|S_{t-1}=s,A_{t-1}=a\right], \tag{17.1}$$

which is the probability of reaching state s' and reward r at time t, conditionally on being in state s and performing action a at time $t-1$. The finite sets of states and actions will be denoted with $\mathcal{S}$ and $\mathcal{A}$ henceforth. Sometimes, this probability is averaged over the set of rewards which gives the following decomposition:

$$\sum_r rp(s',r|s,a)=\mathcal{P}^a_{ss'}\mathcal{R}^a_{ss'}, \quad \text{where} \tag{17.2}$$

$$\begin{aligned}\mathcal{P}^a_{ss'}&=\mathbb{P}\left[S_t=s'|S_{t-1}=s,A_{t-1}=a\right], \quad \text{and}\\ \mathcal{R}^a_{ss'}&=\mathbb{E}\left[R_t|S_{t-1}=s,S_t=s',A_{t-1}=a\right].\end{aligned}$$

The goal of the agent is to maximize some function of the stream of rewards. This gain is usually defined as

$$\begin{aligned}G_t&=\sum_{k=0}^T\gamma^kR_{t+k+1}\\ &=R_{t+1}+\gamma G_{t+1},\end{aligned} \tag{17.3}$$

i.e., it is a discounted version of the reward, where the discount factor is $\gamma\in(0,1]$. The horizon T may be infinite, which is why γ was originally introduced. Assuming the rewards are bounded, the infinite sum diverges for $\gamma=1$, but may converge when $\gamma<1$. When T is finite, the task is called *episodic* and otherwise, it is said to be *continuous*.

In RL, the focal unknown is the *policy* π, which drives the actions of the agent. More precisely, $\pi(a,s)=\mathbb{P}[A_t=a|S_t=s]$, that is, π equals the probability of taking action a if the state of the environment is s. This means that actions are subject to randomness, just like for mixed strategies in game theory. While this may seem disappointing because an investor would want to be sure to take *the* best action, but it is also a good reminder that the best way to face random outcomes may well be to randomize actions as well.

Finally, in order to try to determine the *best* policy, one key indicator is the so-called value function:

$$v_\pi(s)=\mathbb{E}_\pi\left[G_t|S_t=s\right], \tag{17.4}$$

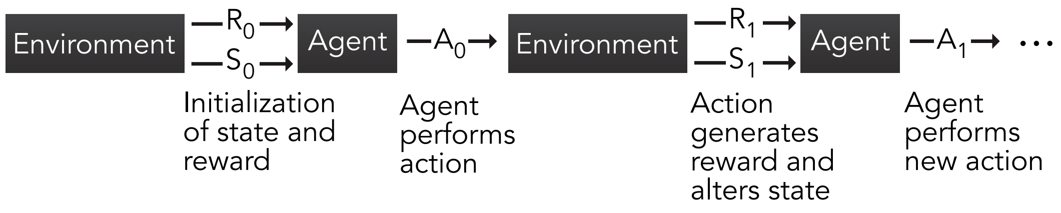

Environment
R0
S0
Agent
A0
Environment
R1
S1
Agent
A1
...
Initialization of state and reward
Agent performs action
Action generates reward and alters state
Agent performs new action

FIGURE 17.1: Scheme of Markov Decision Process. R, S and A stand for reward, state and action, respectively.

where the time index t is not very relevant and omitted in the notation of the function. The index π under the expectation operator $\mathbb{E}[\cdot]$ simply indicates that the average is taken when the policy π is enforced. The value function is simply equal to the average gain conditionally on the state being equal to s. In financial terms, this is equivalent to the average profit if the agents takes action a when the market environment is s. More generally, it is also possible to condition not only on the state, but also on the action taken. We thus introduce the q_π action-value function:

$$q_\pi(s,a) = \mathbb{E}_\pi\left[G_t|S_t = s,\ A_t = a\right]. \tag{17.5}$$

The q_π function is highly important because it gives the average gain when the state and action are fixed. Hence, if the current state is known, then one obvious choice is to select the action for which $q_\pi(s,\cdot)$ is the highest. Of course, this is the best solution if the optimal value of q_π is known, which is not always the case in practice. The value function can easily be accessed via q_π: $v_\pi(s) = \sum_a \pi(a,s)q_\pi(s,a)$.

The optimal v_π and q_π are straightforwardly defined as

$$v_*(s) = \max_\pi v_\pi(s),\ \forall s \in \mathcal{S}, \quad \text{and} \quad q_*(s,a) = \max_\pi q_\pi(s,a),\ \forall (s,a) \in \mathcal{S}\times\mathcal{A}.$$

If only $v_*(s)$ is known, then the agent must span the set of actions and find those that yield the maximum value for any given state s.

Finding these optimal values is a very complicated task and many articles are dedicated to solving this challenge. One reason why finding the best $q_\pi(s,a)$ is difficult is because it depends on two elements (s and a) on one side and π on the other. Usually, for a fixed policy π, it can be time consuming to evaluate $q_\pi(s,a)$ for a given stream of actions, states and rewards. Once $q_\pi(s,a)$ is estimated, then a new policy π' must be tested and evaluated to determine if it is better than the original one. Thus, this iterative search for a good policy can take long. For more details on policy improvement and value function updating, we recommend Chapter 4 of Sutton and Barto (2018) which is dedicated to dynamic programming.

17.1.2 Q-learning

An interesting shortcut to the problem of finding $v_*(s)$ and $q_*(s,a)$ is to remove the dependence on the policy. Consequently, there is then of course no need to iteratively improve the policy. The central relationship that is required to do this is the so-called Bellman equation that is satisfies by $q_\pi(s,a)$. We detail its derivation below. First of all, we recall that

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a], \end{aligned}$$

where the second equality stems from (17.3). The expression $\mathbb{E}_\pi[R_{t+1}|S_t = s, A_t = a]$ can be further decomposed. Since the expectation runs over π, we need to sum over all possible actions a' and states s' and resort to $\pi(a',s')$. In addition, the sum on the s' and r arguments of the probability $p(s',r|s,a) = \mathbb{P}\left[S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\right]$ gives access to the distribution of the random couple (S_{t+1}, R_{t+1}) so that in the end $\mathbb{E}_\pi[R_{t+1}|S_t = s, A_t = a] = \sum_{a',r,s'} \pi(a',s')p(s',r|s,a)r$. A similar reasoning applies to the second portion of q_π and:

$$\begin{aligned} q_\pi(s,a) &= \sum_{a',r,s'} \pi(a',s')p(s',r|s,a)\left[r + \gamma\mathbb{E}_\pi[G_{t+1}|S_t = s', A_t = a']\right] \\ &= \sum_{a',r,s'} \pi(a',s')p(s',r|s,a)\left[r + \gamma q_\pi(s',a')\right]. \end{aligned} \tag{17.6}$$

This equation links $q_\pi(s,a)$ to the future $q_\pi(s',a')$ from the states and actions (s',a') that are accessible from (s,a).

Notably, Equation (17.6) is also true for the optimal action-value function $q_* = \max_\pi q_\pi(s,a)$:

$$\begin{aligned} q_*(s,a) &= \max_{a'} \sum_{r,s'} p(s',r|s,a)\left[r + \gamma q_*(s',a')\right], \\ &= \mathbb{E}_{\pi^*}[r|s,a] + \gamma \sum_{r,s'} p(s',r|s,a)\left(\max_{a'} q_*(s',a')\right) \end{aligned} \tag{17.7}$$

because one optimal policy is one that maximizes $q_\pi(s,a)$, for a given state s and over all possible actions a. This expression is central to a cornerstone algorithm in reinforcement learning called Q-learning (the formal proof of convergence is outlined in Watkins and Dayan (1992)). In Q-learning, the state-action function does no longer depend on policy and is written with capital Q. The process is the following:

Initialize values $Q(s,a)$ for all states s and actions a. For each episode:

0. Initialize state S_0 and for each iteration i until the end of the episode:
1. observe state s_i;
2. perform action a_i (depending on Q);
3. receive reward r_{i+1} and observe state s_{i+1}
4. Update Q as follows:

$$Q_{i+1}(s_i,a_i) \longleftarrow Q_i(s_i,a_i) + \eta\left(\underbrace{r_{i+1} + \gamma \max_a Q_i(s_{i+1},a)}_{\text{echo of (17.7)}} - Q_i(s_i,a_i)\right) \tag{17.8}$$

The underlying reason why this update rule works can be linked to fixed point theorems of contraction mappings. If a function f satisfies $|f(x)-f(y)| < \delta|x-y|$ (Lipshitz continuity), then a fixed point z satisfying $f(z)=z$ can be iteratively obtained via $z \leftarrow f(z)$. This updating rule converges to the fixed point. Equation (17.7) can be solved using a similar principle, except that a learning rate η slows the learning process (but also technically ensures convergence under technical assumptions).

More generally, (17.8) has a form that is widespread in reinforcement learning that is summarized in Equation (2.4) of Sutton and Barto (2018):

$$\text{New estimate} \leftarrow \text{Old estimate} + \text{Step size } (\textit{i.e.}, \text{ learning rate}) \times (\text{Target - Old estimate}), \tag{17.9}$$

where the last part can be viewed as an error term. Starting from the old estimate, the new estimate therefore goes in the 'right' (or sought) direction, modulo a discount term that makes sure that the magnitude of this direction is not too large. The update rule in (17.8) is often referred to as '*temporal difference*' learning because it is driven by the improvement yielded by estimates that are known at time $t+1$ (target) versus those known at time t.

One important step of the Q-learning sequence is the second one where the action a_i is picked. In RL, the best algorithms combine two features: exploitation and exploration. Exploitation is when the machine uses the currrent information at its disposal to choose the next action. In this case, for a given state s_i, it chooses the action a_i that maximizes the expected reward

$Q_i(s_i, a_i)$. While obvious, this choice is not optimal if the current function Q_i is relatively far from the *true* Q. Indeed: decisions (actions) are bound to be suboptimal and repeated many times.

In order to gather new information stemming from actions that have not been tested much (but that can potentially generate higher rewards), exploration is needed. This is when an action a_i is chosen randomly. The most common way to combine these two concepts is called ϵ-greedy exploration. The action a_i is assigned according to:

$$a_i = \begin{cases} \underset{a}{\text{argmax}}\, Q_i(s_i, a) & \text{with probability } 1-\epsilon \\ \text{randomly (uniformly) over } \mathcal{A} & \text{with probability } \epsilon \end{cases}. \tag{17.10}$$

Thus, with probability ϵ, the algorithm explores and with probability $1-\epsilon$, it exploits the current knowledge of the expected reward and picks the best action. Because all actions have a non-zero probability of being chosen, the policy is called "soft". Indeed, then best action has a probability of selection equal to $1-\epsilon(1-\text{card}(\mathcal{A})^{-1})$ while all other actions are picked with probability $\epsilon/\text{card}(\mathcal{A})$.

17.1.3 SARSA

In Q-learning, the algorithm seeks to find the action-value function of the optimal policy. Thus, the policy that is followed to pick actions is different from the one that is learned (via Q). Such algorithms are called *off-policy. On-policy* algorithms seek to improve the estmation of the action-value function q_π by continuously acting according to the policy π. One canonical example of on-policy learning is the SARSA method which requires two consecutive states and actions **SARSA**. The way the quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ is processed is presented below.

The main difference between Q learning and SARSA is the update rule. In SARSA, it is given by

$$Q_{i+1}(s_i, a_i) \longleftarrow Q_i(s_i, a_i) + \eta \left(r_{i+1} + \gamma\, Q_i(s_{i+1}, a_{i+1}) - Q_i(s_i, a_i)\right) \tag{17.11}$$

The improvement comes only from the local point $Q_i(s_{i+1}, a_{i+1})$ that is based on the new states and actions (s_{i+1}, a_{i+1}), whereas in Q-learning, it comes from all possible actions of which only the best is retained $\max_a Q_i(s_{i+1}, a)$.

A more robust (but computationally demanding) version of SARSA is *expected* SARSA in which the target Q function is averaged over all actions:

$$Q_{i+1}(s_i, a_i) \longleftarrow Q_i(s_i, a_i) + \eta \left(r_{i+1} + \gamma \sum_a \pi(a, s_{i+1}) Q_i(s_{i+1}, a) - Q_i(s_i, a_i)\right) \tag{17.12}$$

Expected SARSA is less volatile than SARSA because the latter is strongly impacted by the random choice of a_{i+1}. In expected SARSA, the average smoothes the learning process.

17.2 Issues and potential solutions

17.2.1 The curse of dimensionality

Reinforcement learning is a framework and not a particular algorithm. In fact, different tools can very well co-exist in a RL task (AlphaGo combined both tree methods and neural networks, see Silver et al. (2016)). Nonetheless, any RL attempt will always rely on the three key concepts that are: the states, the actions and the rewards. In factor investing, they fairly easy to identify, though there is always room for interpretation. Actions are evidently defined by portfolio compositions. The states can be viewed as the current values that describe the economy: as a first order approximation, it can be assumed that the feature levels fulfill this role (possibly conditioned or complemented with macro-economic data). The rewards are even more straightforward. Returns or any relevant performance metric (e.g., Sharpe ratio which is for instance used in Moody et al. (1998), Bertoluzzo and Corazza (2012) and Aboussalah and Lee (2020) or drawdown-based ratios, as in Almahdi and Yang (2017)) can account for rewards.

A major problem lies in the dimensionality of both states and actions. Assuming an absence of leverage (no negative weights), the actions take values on the simplex

$$\mathbb{S}_N = \left\{ \mathbf{x} \in \mathbb{R}^N \left| \sum_{n=1}^N x_n = 1, \ x_n \ge 0, \ \forall n = 1, \dots, N \right. \right\} \tag{17.13}$$

and assuming that all features have been uniformized, their space is $[0,1]^{NK}$. Needless to say, the dimensions of both spaces are numerically impractical.

A simple solution to this problem is discretization: each space is divided into a small number of categories. Some authors do take this route. In Yang et al. (2018), the state space is discretized into three values depending on volatility, and actions are also split into three categories. Bertoluzzo and Corazza (2012) and Xiong et al. (2018) also choose three possible actions (buy, hold, sell). In Almahdi and Yang (2019), the learner is expected to yield binary signals for buying or shorting. García-Galicia et al. (2019) consider a larger state space (8 elements) but restrict the action set to 3 options.[2] In terms of the state space, all articles assume that the state of the economy is determined by prices (or returns).

One strong limitation of these approaches is the marked simplification they imply. Realistic discretizations are numerically intractable when investing in multiple assets. Indeed, splitting the unit interval in h points yields h^{NK} possibilities for feature values. The number of options for weight combinations is exponentially increasing N. As an example: just 10 possible values for 10 features of 10 stocks yield 10^{100} permutations.

The problems mentioned above are of course not restricted to portfolio construction. Many solutions have been proposed to solve Markov Decision Processes in continuous spaces. We refer for instance to Section 4 in Powell and Ma (2011) for a review of early methods (outside finance).

This curse of dimensionality is accompanied by fundamental question of training data. Two options are conceivable: market data or simulations. Under a given controlled generator of

[2]Some recent papers consider arbitrary weights (e.g., Jiang et al. (2017) and Yu et al. (2019)) for a limited number of assets.

samples, it is hard to imagine that the algorithm will beat the solution that maximizes a given utility function (if anything, it should converge towards the static optimal solution). This leaves market data as a preferred solution but even with large datasets, there is little chance to cover all the (actions, states) combinations mentioned above. Datasets have depths that run through a few decades of monthly data, which means several hundreds of time-stamps at most. This is by far too limited to allow for a reliable learning process. It is always possible to generate synthetic data (as in Yu et al. (2019)), but it is unclear that this will solidly improve the performance of the algorithm.

17.2.2 Policy gradient

Beyond the discretization of action and state spaces, a powerful trick is **parametrization**. When a and s can take discrete values, action-value functions must be computed for all pairs (a,s), which can be prohibitively cumbersome. An elegant way to circumvent this problem is to assume that the policy is driven by a relatively modest number of parameters. The learning process is then focused on optimizing this set of parameters $\boldsymbol{\theta}$. We then write $\pi_{\boldsymbol{\theta}}(a,s)$ for the probability of choosing action a in state s. One intuitive way to define $\pi_{\boldsymbol{\theta}}(a,s)$ is to resort to a soft-max form:

$$\pi_{\boldsymbol{\theta}}(a,s) = \frac{e^{\boldsymbol{\theta}'\mathbf{h}(a,s)}}{\sum_b e^{\boldsymbol{\theta}'\mathbf{h}(b,s)}}, \tag{17.14}$$

where the output of function $\mathbf{h}(a,s)$, which has the same dimension as $\boldsymbol{\theta}$ is called a feature vector representing the pair (a,s). Typically, $\mathbf{h}$ can very well be a simple neural network with two input units and an output dimension equal to the length of $\boldsymbol{\theta}$.

One desired property for $\pi_{\boldsymbol{\theta}}$ is that it be differentiable with respect to $\boldsymbol{\theta}$ so that $\boldsymbol{\theta}$ can be improved via some gradient method. The most simple and intuitive results about policy gradients are known in the case of episodic tasks (finite horizon) for which it is sought to maximize the average gain $\mathbb{E}_{\boldsymbol{\theta}}[G_t]$ where the gain is defined in Equation (17.3). The expectation is computed according to a particular policy that depends on $\boldsymbol{\theta}$, this is why we use a simple subscript. One central result is the so-called policy gradient theorem which states that

$$\nabla \mathbb{E}_{\boldsymbol{\theta}}[G_t] = \mathbb{E}_{\boldsymbol{\theta}}\left[G_t \frac{\nabla \pi_{\boldsymbol{\theta}}}{\pi_{\boldsymbol{\theta}}}\right]. \tag{17.15}$$

This result can then be used for gradient ascent: when seeking to maximize a quantity, the parameter change must go in the upward direction:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \nabla \mathbb{E}_{\boldsymbol{\theta}}[G_t]. \tag{17.16}$$

This simple update rule is known as the REINFORCE algorithm. Many extensions of this simple idea exist (adding baseline, actor-critic methods, etc) and we refer to Chapter 13 of Sutton and Barto (2018) for a detailed account on this topic.

One interesting application of parametric policies is outlined in Aboussalah and Lee (2020). In their article, the authors define a trading policy that is based on a recurrent neural network. Thus, the parameter $\boldsymbol{\theta}$ in this case encompasses all weights and biases in the network.

Another favorable feature of parametric policies is that they are compatible with continuous sets of actions. Beyond the form (17.14), there are other ways to shape $\pi_{\boldsymbol{\theta}}$. If $\mathcal{A}$ is a subset of $\mathbb{R}$, and $f_{\boldsymbol{\Omega}}$ is a density function with parameters $\boldsymbol{\Omega}$, then a candidate form for $\pi_{\boldsymbol{\theta}}$ is

$$\pi_{\boldsymbol{\theta}} = f_{\boldsymbol{\Omega}(s,\boldsymbol{\theta})}(a), \tag{17.17}$$

in which the parameters $\boldsymbol{\Omega}$ are in turn functions of the states and of the underlying (second order) parameters $\boldsymbol{\theta}$.

While the Gaussian distribution (see Section 13.7 in Sutton and Barto (2018)) are often a preferred choice, they would require some processing to lie inside the unit interval. One easy way to obtain such values is to apply the normal cumulative distribution function to the output. In Wang and Zhou (2019), the multivariate Gaussian policy is theoretically explored, but it assumes no constraint on weights.

Some natural parametric distributions emerge as alternatives. If only one asset is traded, then the Bernoulli distribution can be used to determine whether or not to buy the asset. If a riskless asset is available, the beta distribution offers more flexibility because the values for the proportion invested in the risky asset span the whole interval; the remainder can be invested into the safe asset. When many asset are traded, things become more complicated because of the budget constraint. One ideal candidate is the Dirichlet distribution because it is defined on a simplex (see Equation ((17.13))):

$$f_{\boldsymbol{\alpha}}(x_1,\dots,x_n) = \frac{1}{B(\boldsymbol{\alpha})}\prod_{i=1}^n x_i^{\alpha_i-1},$$

where $B(\boldsymbol{\alpha})$ is a normalizing constant depending on n strictly positive parameters $\alpha_1,\dots,\alpha_n$.

17.3 Simple examples

17.3.1 Q-learning with simulations

To illustrate the gist of the problems mentioned above, we propose two implementations of Q-learning. For simplicity, the first one is based on simulations. This helps understand the learning process in a simplified framework. We consider two assets: one risky and one riskless, with return equal to zero. The returns for the risky process follow an autoregressive model of order one (AR(1)): $r_{t+1} = a + \rho r_t + \epsilon_{t+1}$ with $|\rho| < 1$ and ϵ following a standard white noise with variance σ^2.

The environment consists only in observing the past return r_t. Since we seek to estimate the Q function, we need to discretize this state variable. The simplest choice is to resort to a binary variable: equal to -1 (negative) if $r_t < 0$ and to +1 (positive) if $r_t \geq 0$. The actions are summarized by the quantity invested in the risky asset. It can take 5 values: 0 (risk-free portfolio), 0.25, 0.5, 0.75 and 1 (fully invested in the risky asset). This is for instance the same choice as in Pendharkar and Cusatis (2018).

The landscape of R libraries for RL is surprisingly sparse. We resort to the package *ReinforcementLearning* which has an intuitive implementation of Q-learning (another option would be the *reinforcelearn* package). It requires a dataset with the usual inputs: state, action, reward and subsequent state. We start by simulating the returns: they drive the states and the rewards (portfolio returns). The actions are sampled randomly. Technically, the main function of the package requires that states and actions be of character type. The data is built in the chunk below.

```r
library(ReinforcementLearning)                            # Package for RL
set.seed(42)                                              # Fixing the random seed
n_sample <- 10^5                                          # Number of samples to be generated
rho <- 0.3                                                # Autoregressive parameter
sd <- 0.4                                                 # Std. dev. of noise
a <- 0.06 * rho                                           # Scaled mean of returns
data_RL <- tibble(returns = a/rho + arima.sim(n = n_sample, # Returns via AR(1) simulation
                                      list(ar = rho),
                                      sd = sd),
                  action = round(runif(n_sample)*4)/4) %>%  # Random action (portfolio)
    mutate(state = if_else(returns < 0, "neg", "pos"),      # Coding of state
           reward = returns * action,                       # Reward = portfolio return
           new_state = lead(state),                         # Next state
           action = as.character(action)) %>%
    na.omit()                                             # Remove one missing state
data_RL %>% head()                                        # Show first lines
```

```
## # A tibble: 6 x 5
##   returns action state reward new_state
##     <dbl> <chr>  <chr>  <dbl> <chr>
## 1   0.674 0      pos   0      pos
## 2   0.206 0.25   pos   0.0516 pos
## 3   0.911 0.25   pos   0.228  pos
## 4   0.290 0      pos   0      pos
## 5   0.651 0.25   pos   0.163  pos
## 6   1.15  0.25   pos   0.288  neg
```

There are 3 parameters in the implementation of the Q-learning algorithm:

- γ, which is the learning rate in the updating Equation (17.8). In *ReinforcementLearning*, this is coded as *alpha*;

- γ, the discounting rate for the rewards (also shown in Equation (17.8));
- and ϵ, which controls the rate of exploration versus exploitation (see Equation (17.10)).

```r
control <- list(alpha = 0.1,                          # Learning rate
                gamma = 0.7,                          # Discount factor for rewards
                epsilon = 0.1)                        # Exploration rate

fit_RL <- ReinforcementLearning(data_RL,              # Main RL function
                               s = "state",
                               a = "action",
                               r = "reward",
                               s_new = "new_state",
                               control = control)
print(fit_RL)   # Show the output
```

```
## State-Action function Q
##          0.25         0         1      0.75       0.5
## neg 0.4704760 0.4945049 0.2073319 0.3247837 0.3099006
## pos 0.6814258 0.5432345 1.0100722 0.9102886 0.7287897
##
## Policy
## neg pos
## "0" "1"
##
## Reward (last iteration)
## [1] 2877.216
```

The output shows the Q function, which depends naturally both on states and actions. When the state is negative, large risky positions (action equal to 0.75 or 1.00) are associated with negative average rewards whereas small positions yield positive average rewards. When the state is positive, the average rewards are the highest for the largest allocations. Thus, the recommendation of the algorithm (i.e., the policy) is to be fully invested in a postive state and to refrain from investing in a negative state. Given the positive autocorrelation of the underlying process, this does make sense.

Basically, the algorithm has simply learned that positive (*resp.* negative) returns are more likely to follow positive (*resp.* negative) returns. While this is somewhat reassuring, it is by no means impressive and much simpler tools would yield similar conclusions and guidance.

17.3.2 Q-learning with market data

The second application is based on the financial dataset. To reduce the dimensionality of the problem, we will assume:
- that only one feature (price-to-book ratio) captures the state of the environment;
- that actions take values over a discrete set consisting of three positions: +1 (buy the market), -1 (sell the market) and 0 (hold no risky positions);
- that only two assets are traded: those with stock_id equal to 1 and 3.

The construction of the dataset is unelegantly coded below.

```
return_1 <- data_ml %>% filter(stock_id ==1) %>% pull(R1M_Usd)      # Return of asset 1
return_3 <- data_ml %>% filter(stock_id ==3) %>% pull(R1M_Usd)      # Return of asset 3
pb_1 <- data_ml %>% filter(stock_id ==1) %>% pull(Pb)               # P/B ratio of asset 1
pb_3 <- data_ml %>% filter(stock_id ==3) %>% pull(Pb)               # P/B ratio of asset 3
action_1 <- floor(runif(length(pb_1))*3) - 1                        # Action for asset 1 (random)
action_3 <- floor(runif(length(pb_1))*3) - 1                        # Action for asset 3 (random)
RL_data <- tibble(return_1, return_3,                               # Building the dataset
                  pb_1, pb_3,
                  action_1, action_3) %>%
    mutate(action = paste(action_1, action_3),                      # Uniting actions
           pb_1 = round(5 * pb_1),                                  # Simplifying states
           pb_3 = round(5 * pb_3),                                  # Simplifying states
           state = paste(pb_1, pb_3),                               # Uniting states
           reward = action_1*return_1 + action_3*return_3,          # Computing rewards
           new_state = lead(state)) %>%                             # Infer new state
    select(-pb_1, -pb_3, -action_1, -action_3, -return_1, -return_3) # Remove superfluous vars.
head(RL_data)                                                       # Showing the result
```

```
## # A tibble: 6 x 4
##   action state reward new_state
##   <chr>  <chr>  <dbl> <chr>
## 1 0 -1   1 1   -0.077 1 1
## 2 -1 -1  1 1   -0.239 1 1
## 3 0 0    1 1    0     1 1
## 4 0 -1   1 1   -0.027 1 1
## 5 -1 -1  1 1    0.047 1 1
## 6 1 1    1 1   -0.149 1 1
```

Actions and states have to be merged to yield all possible combinations. To simplify the states, we round 5 times the price-to-book ratios.

We keep the same hyperparameters as in the previous example. Columns below stand for actions: the first (*resp.* second) number notes the position in the first (*resp.* second) asset.

The rows correspond to states. The scaled P/R ratios are separated by a point (e.g., “X2.3” means that the first (*resp.* second) asset has a scale P/B of 2 (*resp.* 3).

```r
fit_RL2 <- ReinforcementLearning(RL_data,           # Main RL function
                                 s = "state",
                                 a = "action",
                                 r = "reward",
                                 s_new = "new_state",
                                 control = control)
fit_RL2$Q <- round(fit_RL2$Q, 3) # Round the Q-matrix
print(fit_RL2)                     # Show the output
```

```
## State-Action function Q
##         0 0    0 1   0 -1  -1 -1   -1 0   -1 1   1 -1    1 0    1 1
## X0.0 0.000  0.000  0.000 -0.026  0.000  0.000  0.017  0.000  0.000
## X0.1 0.000  0.000 -0.058  0.000  0.009  0.000  0.000  0.000  0.000
## X4.1 0.000  0.002 -0.005  0.008  0.000  0.000  0.000 -0.006 -0.008
## X3.1 0.021 -0.005  0.017  0.017  0.003  0.049 -0.013  0.030 -0.007
## X3.2 0.001  0.020  0.013  0.010  0.041  0.015 -0.002  0.000  0.000
## X2.0 0.000  0.000 -0.017  0.000  0.000 -0.013  0.000  0.000  0.000
## X2.1 0.004 -0.019 -0.002 -0.037 -0.017 -0.024  0.020  0.002 -0.011
## X2.2 0.002 -0.002  0.005  0.034  0.010 -0.006  0.020  0.000 -0.020
## X2.3 0.000  0.000  0.005  0.000  0.000  0.000 -0.007  0.013  0.000
## X1.1 0.004 -0.003 -0.012 -0.033  0.015 -0.006  0.017  0.037  0.014
## X1.2 0.000  0.009 -0.008  0.000 -0.016  0.000  0.021  0.015  0.021
##
## Policy
##    X0.0    X0.1    X4.1    X3.1    X3.2    X2.0    X2.1    X2.2    X2.3
##  "1 -1"  "-1 0" "-1 -1"  "-1 1"  "-1 0"   "0 0"  "1 -1" "-1 -1"   "1 0"
##    X1.1    X1.2
##   "1 0"  "1 -1"
##
## Reward (last iteration)
## [1] -1.362
```

The output show that there are many combinations of states and actions that are not spanned by the data. Some states seem to be more often represented (“X1.1”, “X2.1” and X3.1“). It is hard to make any sense of the recommendations. The first two states (”X0.0" and “X0.1”) are close but the decisions related to them are very different (buy and short versus short and hold). Moreover, there is no coherence and no monotonicity in actions with respect to individual state values: low values of states can be associated to very different actions. The only seemingly robust pattern is the advice to sell the first asset when its scaled P/B ratio is high (values of 3 or 4).

One reason why these conclusion do not appear trustworthy pertains to the data size. With only 200+ time points and 99 state-action pairs (11 times 9), this yields on average only two data points to compute the Q function. This could be improved by testing more random actions, but the limits of the sample size would eventually (rapidly) be reached anyway.

17.4 Concluding remarks

Reinforcement learning has been applied to financial problems for a long time. Early contributions in the late 1990s include Neuneier (1996), Moody and Wu (1997), Moody et al. (1998) and Neuneier (1998). Since then, many researchers in the computer science field have sought to apply RL techniques to portfolio problems. The advent of massive datasets and the increase in dimensionality make it hard for RL tools to adapt well to very rich environments that are encountered in factor investing.

Recently, some approaches seek to adapt RL to continuous action spaces (Wang and Zhou (2019), Aboussalah and Lee (2020)) but not to high-dimensional state spaces. These spaces are those required in factor investing because all firms yields hundreds of data points characterizing their economic situation. In addition, applications of RL in financial frameworks have a particularity compared to many typical RL tasks: in financial markets, actions of agents have no impact on the environment (unless the agent is able to perform massive trades, which is rare and ill-advised because it pushes prices in the wrong direction). This lack of impact of actions may possibly mitigate the efficiency of traditional RL approaches.

Those are challenges that will need to be solved in order for RL to become competitive with alternative (supervised) methods. We end this chapter by underlining that reinforcement learning has also been used to estimate complex theoretical models (Halperin and Feldshteyn (2018), García-Galicia et al. (2019)). The research in the field is incredibly diversified and is orientated towards many directions. It is likely that captivating work will be published in the near future.

17.5 Exercises

1. Test what happens if the process for generating returns has a negative autocorrelation. What is the impact on the Q function and the policy?
2. Increase the size of RL_data by testing all possible action combination for each original data point. Re-run the Q-learning function and see what happens.

18

Natural Language Processing

Loughran and McDonald (2016)

Gentzkow et al. (2019)

Cong et al. (2019a)

Cong et al. (2019b)

19

Conclusion

A

Data Description

Column Name	Short Description
stock_id	security id
date	date of the data
Advt_12M_Usd	average daily volume in amount in USD over 12 months
Advt_3M_Usd	average daily volume in amount in USD over 3 months
Advt_6M_Usd	average daily volume in amount in USD over 6 months
Asset_Turnover	total sales on average assets
Bb_Yld	buyback yield
Bv	book value
Capex_Ps_Cf	capital expenditure on price to sale cash flow
Capex_Sales	capital expenditure on sales
Cash_Div_Cf	cash dividends cash flow
Cash_Per_Share	cash per share
Cf_Sales	cash flow per share
Debtequity	debt to equity
Div_Yld	dividend yield
Dps	dividend per share
Ebit_Bv	EBIT on book value
Ebit_Noa	EBIT on non operating asset
Ebit_Oa	EBIT on operating asset
Ebit_Ta	EBIT on total asset
Ebitda_Margin	EBITDA margin
Eps	earnings per share
Eps_Basic	earnings per share basic
Eps_Basic_Gr	earnings per share growth
Eps_Contin_Oper	earnings per share continuing operations
Eps_Dil	earnings per share diluted
Ev	enterprise value
Ev_Ebitda	enterprise value on EBITDA
Fa_Ci	fixed assets on common equity
Fcf	free cash flow
Fcf_Bv	free cash flow on book value
Fcf_Ce	free cash flow on capital employed

TABLE A.1: Description of features in the dataset.

Column Name	**Short Description**
Fcf_Margin	free cash flow margin
Fcf_Noa	free cash flow on net operating assets
Fcf_Oa	free cash flow on operating assets
Fcf_Ta	free cash flow on total assets
Fcf_Tbv	free cash flow on tangible book value
Fcf_Toa	free cash flow on total operating assets
Fcf_Yld	free cash flow yield
Free_Ps_Cf	free cash flow on price sales
Int_Rev	intangibles on revenues
Interest_Expense	interest expense coverage
Mkt_Cap_12M_Usd	average market capitalisation over 12 months in USD
Mkt_Cap_3M_Usd	average market capitalisation over 3 months in USD
Mkt_Cap_6M_Usd	average market capitalisation over 6 months in USD
Mom_11M_Usd	price momentum 12 - 1 months in USD
Mom_5M_Usd	price momentum 6 - 1 months in USD
Mom_Sharp_11M_Usd	price momentum 12 - 1 months in USD / divided by volatility
Mom_Sharp_5M_Usd	price momentum 6 - 1 months in USD / divided by volatility
Nd_Ebitda	net debt on EBITDA
Net_Debt	net debt
Net_Debt_Cf	net debt on cash flow
Net_Margin	net margin
Netdebtyield	net debt yield
Ni	net income
Ni_Avail_Margin	net income available margin
Ni_Oa	net income on operating asset
Ni_Toa	net income on total operating asset
Noa	net operating asset
Oa	operating asset

TABLE A.2: Description of features in the dataset (continued).

Column Name	**Short Description**
Ocf	operating cash flow
Ocf_Bv	operating cash flow on book value
Ocf_Ce	operating cash flow on capital employed
Ocf_Margin	operating cash flow margin
Ocf_Noa	operating cash flow on net operating assets
Ocf_Oa	operating cash flow on operating assets
Ocf_Ta	operating cash flow on total assets
Ocf_Tbv	operating cash flow on tangible book value
Ocf_Toa	operating cash flow on total operating assets
Op_Margin	operating margin
Op_Prt_Margin	net marging 1Y growth
Oper_Ps_Net_Cf	cash flow from operations per share net
Pb	price to book
Pe	price earnings
Ptx_Mgn	margin pretax
Recurring_Earning_Total_Assets	reccuring earnings on total assets
Return_On_Capital	return on capital
Rev	revenue
Roa	return on assets
Roc	return on capital
Roce	return on capital employed
Roe	return on equity
Sales_Ps	price to sales
Share_Turn_12M	average share turnover 12 months
Share_Turn_3M	average share turnover 3 months
Share_Turn_6M	average share turnover 6 months
Ta	total assets
Tev_Less_Mktcap	total enterprise value less market capitalisation
Tot_Debt_Rev	total debt on revenue
Total_Capital	total capital
Total_Debt	total debt on revenue
Total_Debt_Capital	total debt on capital
Total_Liabilities_Total_Assets	total liabilities on total assets
Vol1Y_Usd	volatility of returns one year
Vol3Y_Usd	volatility of returns 3 years
R1M_Usd	return forward 1 month
R3M_Usd	return forward 3 months
R6M_Usd	return forward 6 months
R12M_Usd	return forward 12 months

TABLE A.3: Description of features and labels in the dataset.

B

Solution to exercises

B.1 Chapter 4

For annual values:

```r
data_ml %>%
    group_by(date) %>%
    mutate(growth = Pb > median(Pb)) %>%             # Creates the sort
    ungroup() %>%                                    # Ungroup
    mutate(year = lubridate::year(date)) %>%         # Creates a year variable
    group_by(year, growth) %>%                       # Analyze by year & sort
    summarize(ret = mean(R1M_Usd)) %>%               # Compute average return
    ggplot(aes(x = year, y = ret, fill = growth)) + geom_col(position = "dodge") + # Plot!
    theme(legend.position = c(0.7, 0.8))
```

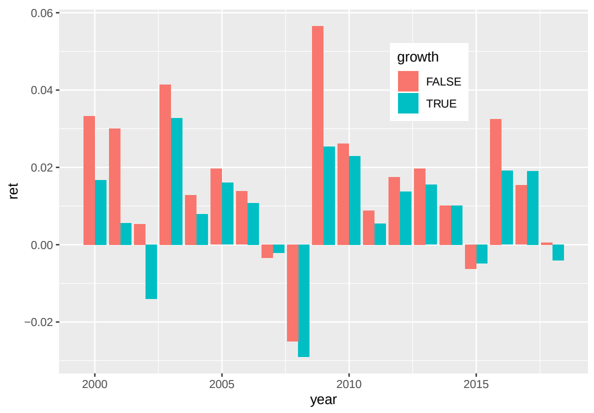


FIGURE B.1: The value factor: annual returns

For monthly values:

```r
returns <- data_ml %>%
    group_by(date) %>%
    mutate(growth = Pb > median(Pb)) %>%                          # Creates the sort
    group_by(date, growth) %>%                                    # Analyze by date & sort
    summarize(ret = mean(R1M_Usd)) %>%                            # Compute average return
    spread(key = growth, value = ret) %>%                         # Pivot to wide matrix format
    ungroup()
colnames(returns)[2:3] <- c("value", "growth")                    # Changing column names
returns %>%
    mutate(value = cumprod(1 + value),                            # From returns to portf. values
           growth = cumprod(1 + growth)) %>%
    gather(key = portfolio, value = value, -date) %>%              # Back in tidy format
    ggplot(aes(x = date, y = value, color = portfolio)) + geom_line() +      # Plot!
    theme(legend.position = c(0.7, 0.8))
```

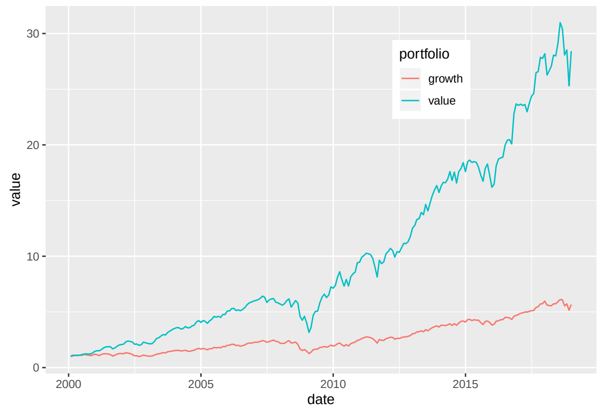


FIGURE B.2: The value factor: portfolio values.

B.2 Chapter 5

B.3 Chapter 6

B.4 Chapter 7

B.5 Chapter 8

B.6 Chapter 9

B.7 Chapter 12

B.8 Chapter 13

B.8.1 Functional programming in the backtest

B.8.2 Advanced weighting function

C

References

Bibliography

Abbasi, A., Albrecht, C., Vance, A., and Hansen, J. (2012). Metafraud: a meta-learning framework for detecting financial fraud. *MIS Quarterly*, pages 1293–1327.

Aboussalah, A. M. and Lee, C.-G. (2020). Continuous control with stacked deep dynamic recurrent reinforcement learning for portfolio optimization. *Expert Systems with Applications*, 140:112891.

Agarwal, A., Hazan, E., Kale, S., and Schapire, R. E. (2006). Algorithms for portfolio management based on the newton method. In *Proceedings of the 23rd international conference on Machine learning*, pages 9–16. ACM.

Aggarwal, C. C. (2013). *Outlier analysis*. Springer.

Allison, P. D. (2001). *Missing data*, volume 136. Sage publications.

Almahdi, S. and Yang, S. Y. (2017). An adaptive portfolio trading system: A risk-return portfolio optimization using recurrent reinforcement learning with expected maximum drawdown. *Expert Systems with Applications*, 87:267–279.

Almahdi, S. and Yang, S. Y. (2019). A constrained portfolio trading system using particle swarm algorithm and recurrent reinforcement learning. *Expert Systems with Applications*, 130:145–156.

Ammann, M., Coqueret, G., and Schade, J.-P. (2016). Characteristics-based portfolio choice with leverage constraints. *Journal of Banking & Finance*, 70:23–37.

Amrhein, V., Greenland, S., and McShane, B. (2019). Scientists rise up against statistical significance. *Nature*, 567:305–307.

Anderson, J. A. and Rosenfeld, E. (2000). *Talking nets: An oral history of neural networks*. MIT Press.

Ang, A. (2014). *Asset management: A systematic approach to factor investing*. Oxford University Press.

Ang, A., Hodrick, R. J., Xing, Y., and Zhang, X. (2006). The cross-section of volatility and expected returns. *Journal of Finance*, 61(1):259–299.

Ang, A. and Kristensen, D. (2012). Testing conditional factor models. *Journal of Financial Economics*, 106(1):132–156.

Ang, A., Liu, J., and Schwarz, K. (2018). Using individual stocks or portfolios in tests of factor models. *SSRN Working Paper*, 1106463.

Arik, S. O. and Pfister, T. (2019). Tabnet: Attentive interpretable tabular learning. *arXiv Preprint*, (1908.07442).

Arjovsky, M., Bottou, L., Gulrajani, I., and Lopez-Paz, D. (2019). Invariant risk minimization. *arXiv Preprint*, (1907.02893).

Arnott, R., Harvey, C. R., Kalesnik, V., and Linnainmaa, J. (2019a). Alice's adventures in factorland: Three blunders that plague factor investing. *The Journal of Portfolio Management*, 45(4):18–36.

Arnott, R., Harvey, C. R., and Markowitz, H. (2019b). A backtesting protocol in the era of machine learning. *Journal of Financial Data Science*, 1(1):64–74.

Asness, C., Frazzini, A., Israel, R., Moskowitz, T. J., and Pedersen, L. H. (2018). Size matters, if you control your junk. *Journal of Financial Economics*, 129(3):479–509.

Asness, C. S., Moskowitz, T. J., and Pedersen, L. H. (2013). Value and momentum everywhere. *Journal of Finance*, 68(3):929–985.

Astakhov, A., Havranek, T., and Novak, J. (2019). Firm size and stock returns: A quantitative survey. *Journal of Economic Surveys*, XXX:XX–XX.

Back, K. (2010). *Asset pricing and portfolio choice theory.* Oxford University Press.

Baesens, B., Van Vlasselaer, V., and Verbeke, W. (2015). *Fraud analytics using descriptive, predictive, and social network techniques: a guide to data science for fraud detection.* John Wiley & Sons.

Bailey, D. H. and de Prado, M. L. (2014). The deflated sharpe ratio: correcting for selection bias, backtest overfitting, and non-normality. *Journal of Portfolio Management*, 40(5):94–107.

Bailey, T. and Jain, A. (1978). A note on distance-weighted k-nearest neighbor rules. *IEEE Trans. on Systems, Man, Cybernetics*, 8(4):311–313.

Bajgrowicz, P. and Scaillet, O. (2012). Technical trading revisited: False discoveries, persistence tests, and transaction costs. *Journal of Financial Economics*, 106(3):473–491.

Baker, M., Bradley, B., and Wurgler, J. (2011). Benchmarks as limits to arbitrage: Understanding the low-volatility anomaly. *Financial Analysts Journal*, 67(1):40–54.

Baker, M., Luo, P., and Taliaferro, R. (2017). Detecting anomalies: The relevance and power of standard asset pricing tests.

Bali, T. G., Engle, R. F., and Murray, S. (2016). *Empirical asset pricing: the cross section of stock returns.* John Wiley & Sons.

Ballings, M., Van den Poel, D., Hespeels, N., and Gryp, R. (2015). Evaluating multiple classifiers for stock price direction prediction. *Expert Systems with Applications*, 42(20):7046–7056.

Ban, G.-Y., El Karoui, N., and Lim, A. E. (2016). Machine learning and portfolio optimization. *Management Science*, 64(3):1136–1154.

Bansal, R., Hsieh, D. A., and Viswanathan, S. (1993). A new approach to international arbitrage pricing. *Journal of Finance*, 48(5):1719–1747.

Bansal, R. and Viswanathan, S. (1993). No arbitrage and arbitrage pricing: A new approach. *Journal of Finance*, 48(4):1231–1262.

Banz, R. W. (1981). The relationship between return and market value of common stocks. *Journal of Financial Economics*, 9(1):3–18.

Barberis, N. (2018). Psychology-based models of asset prices and trading volume. In *Handbook of Behavioral Economics-Foundations and Applications.*

Barberis, N., Greenwood, R., Jin, L., and Shleifer, A. (2015). X-capm: An extrapolative capital asset pricing model. *Journal of Financial Economics*, 115(1):1–24.

Barberis, N., Jin, L. J., and Wang, B. (2019). Prospect theory and stock market anomalies. *SSRN Working Paper*, 3477463.

Barberis, N., Mukherjee, A., and Wang, B. (2016). Prospect theory and stock returns: An empirical test. *Review of Financial Studies*, 29(11):3068–3107.

Barberis, N. and Shleifer, A. (2003). Style investing. *Journal of Financial Economics*, 68(2):161–199.

Barillas, F. and Shanken, J. (2018). Comparing asset pricing models. *Journal of Finance*, 73(2):715–754.

Barron, A. R. (1993). Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information Theory*, 39(3):930–945.

Barron, A. R. (1994). Approximation and estimation bounds for artificial neural networks. *Machine learning*, 14(1):115–133.

Basak, J. (2004). Online adaptive decision trees. *Neural computation*, 16(9):1959–1981.

Bates, J. M. and Granger, C. W. (1969). The combination of forecasts. *Journal of the Operational Research Society*, 20(4):451–468.

Baz, J., Granger, N., Harvey, C. R., Le Roux, N., and Rattray, S. (2015). Dissecting investment strategies in the cross section and time series. *SSRN Working Paper*, 2695101.

Beery, S., Van Horn, G., and Perona, P. (2018). Recognition in terra incognita. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 456–473.

Belsley, D. A., Kuh, E., and Welsch, R. E. (2005). *Regression diagnostics: Identifying influential data and sources of collinearity*, volume 571. John Wiley & Sons.

Ben-David, S., Blitzer, J., Crammer, K., Kulesza, A., Pereira, F., and Vaughan, J. W. (2010). A theory of learning from different domains. *Machine learning*, 79(1-2):151–175.

Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305.

Berk, J. B., Green, R. C., and Naik, V. (1999). Optimal investment, growth options, and security returns. *Journal of Finance*, 54(5):1553–1607.

Bertoluzzo, F. and Corazza, M. (2012). Testing different reinforcement learning configurations for financial trading: Introduction and applications. *Procedia Economics and Finance*, 3:68–77.

Bertsekas, D. P. (2017). *Dynamic programming and optimal control - Volume II, Fourth Edition.* Athena Scientific.

Betermier, S., Calvet, L. E., and Jo, E. (2019). A supply and demand approach to equity pricing. *SSRN Working Paper*, 3440147.

Betermier, S., Calvet, L. E., and Sodini, P. (2017). Who are the value and growth investors? *Journal of Finance*, 72(1):5–46.

Bhamra, H. S. and Uppal, R. (2019). Does household finance matter? small financial errors with large social costs. *American Economic Review*, 109(3):1116–54.

Bhatia, N. et al. (2010). Survey of nearest neighbor techniques. *arXiv Preprint*, (1007.0085).

Bhattacharyya, S., Jha, S., Tharakunnel, K., and Westland, J. C. (2011). Data mining for credit card fraud: A comparative study. *Decision Support Systems*, 50(3):602–613.

Biau, G. (2012). Analysis of a random forests model. *Journal of Machine Learning Research*, 13(Apr):1063–1095.

Biau, G., Devroye, L., and Lugosi, G. (2008). Consistency of random forests and other averaging classifiers. *Journal of Machine Learning Research*, 9(Sep):2015–2033.

Black, F. and Litterman, R. (1992). Global portfolio optimization. *Financial Analysts Journal*, 48(5):28–43.

Blank, H., Davis, R., and Greene, S. (2019). Using alternative research data in real-world portfolios. *Journal of Investing*, 28(4):95–103.

Blum, A. and Kalai, A. (1999). Universal portfolios with and without transaction costs. *Machine Learning*, 35(3):193–205.

Boehmke, B. and Greenwell, B. (2019). *Hands-on Machine Learning with R*. Chapman and Hall.

Boloorforoosh, A., Christoffersen, P., Gourieroux, C., and Fournier, M. (2019). Beta risk in the cross-section of equities. *Review of Financial Studies*, Forthcoming.

Bonaccolto, G. and Paterlini, S. (2019). Developing new portfolio strategies by aggregation. *Annals of Operations Research*, pages 1–39.

Boriah, S., Chandola, V., and Kumar, V. (2008). Similarity measures for categorical data: A comparative evaluation. In *Proceedings of the 2008 SIAM international conference on data mining*, pages 243–254.

Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM.

Bouchaud, J.-p., Krueger, P., Landier, A., and Thesmar, D. (2019). Sticky expectations and the profitability anomaly. *The Journal of Finance*, 74(2):639–674.

Brandt, M. W., Santa-Clara, P., and Valkanov, R. (2009). Parametric portfolio policies: Exploiting characteristics in the cross-section of equity returns. *Review of Financial Studies*, 22(9):3411–3447.

Braun, H. and Chandler, J. S. (1987). Predicting stock market behavior through rule induction: an application of the learning-from-example approach. *Decision Sciences*, 18(3):415–429.

Breiman, L. (1996). Stacked regressions. *Machine Learning*, 24(1):49–64.

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

Breiman, L. et al. (2004). Population theory for boosting ensembles. *Annals of Statistics*, 32(1):1–11.

Breiman, L., Friedman, J., Stone, C. J., and Olshen, R. (1984). *Classification And Regression Trees*. Chapman and Hall.

Brodie, J., Daubechies, I., De Mol, C., Giannone, D., and Loris, I. (2009). Sparse and stable markowitz portfolios. *Proceedings of the National Academy of Sciences*, 106(30):12267–12272.

Brown, I. and Mues, C. (2012). An experimental comparison of classification algorithms for imbalanced credit scoring data sets. *Expert Systems with Applications*, 39(3):3446–3453.

Bryzgalova, S. (2019). Spurious factors in linear asset pricing models.

Bryzgalova, S., Huang, J., and Julliard, C. (2019a). Bayesian solutions for the factor zoo: We just ran two quadrillion models. *SSRN Working Paper*, 3481736.

Bryzgalova, S., Pelger, M., and Zhu, J. (2019b). Forest through the trees: Building cross-sections of stock returns. *SSRN Working Paper*, 3493458.

Buehler, H., Gonon, L., Teichmann, J., and Wood, B. (2019). Deep hedging. *Quantitative Finance*, pages 1–21.

Bühlmann, P., Peters, J., Ernest, J., et al. (2014). Cam: Causal additive models, high-dimensional order search and penalized regression. *Annals of Statistics*, 42(6):2526–2556.

Burrell, P. R. and Folarin, B. O. (1997). The impact of neural networks in finance. *Neural Computing & Applications*, 6(4):193–200.

Cao, L.-J. and Tay, F. E. H. (2003). Support vector machine with adaptive parameters in financial time series forecasting. *IEEE Transactions on Neural Networks*, 14(6):1506–1518.

Carhart, M. M. (1997). On persistence in mutual fund performance. *Journal of Finance*, 52(1):57–82.

Carlson, M., Fisher, A., and Giammarino, R. (2004). Corporate investment and asset price dynamics: Implications for the cross-section of returns. *Journal of Finance*, 59(6):2577–2603.

Cattaneo, M. D., Crump, R. K., Farrell, M., and Schaumburg, E. (2019). Characteristic-sorted portfolios: Estimation and inference. Forthcoming.

Cazalet, Z. and Roncalli, T. (2014). Facts and fantasies about factor investing. *SSRN Working Paper*, 2524547.

Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(3):15.

Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: A library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27.

Che, Z., Purushotham, S., Cho, K., Sontag, D., and Liu, Y. (2018). Recurrent neural networks for multivariate time series with missing values. *Scientific reports*, 8(1):6085.

Chen, A. Y. and Zimmermann, T. (2019). Publication bias and the cross-section of stock returns. *Review of Asset Pricing Studies*, Forthcoming.

Chen, H. (2001). Initialization for norta: Generation of random vectors with specified marginals and correlations. *INFORMS Journal on Computing*, 13(4):312–331.

Chen, J., Song, L., Wainwright, M. J., and Jordan, M. I. (2018). L-shapley and c-shapley: Efficient model interpretation for structured data. *arXiv Preprint*, (1808.02610).

Chen, J.-F., Chen, W.-L., Huang, C.-P., Huang, S.-H., and Chen, A.-P. (2016). Financial time-series data analysis using deep convolutional neural networks. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pages 87–92. IEEE.

Chen, L., Da, Z., and Priestley, R. (2012). Dividend smoothing and predictability. *Management Science*, 58(10):1834–1853.

Chen, L., Pelger, M., and Zhu, J. (2019). Deep learning in asset pricing. *SSRN Working Paper*, 3350138.

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International conference on knowledge discovery and data mining*, pages 785–794. ACM.

Chib, S., Zeng, X., and Zhao, L. (2019). On comparing asset pricing models. *Journal of Finance*, Forthcoming.

Chinco, A., Clark-Joseph, A. D., and Ye, M. (2019a). Sparse signals in the cross-section of returns. *Journal of Finance*, 74(1):449–492.

Chinco, A., Neuhierl, A., and Weber, M. (2019b). Estimating the anomaly baserate. *SSRN Working Paper*, 3344499.

Chipman, H. A., George, E. I., and McCulloch, R. E. (2010). Bart: Bayesian additive regression trees. *Annals of Applied Statistics*, 4(1):266–298.

Choi, S. M. and Kim, H. (2014). Momentum effect as part of a market equilibrium. *Journal of Financial and Quantitative Analysis*, 49(1):107–130.

Chollet, F. (2017). *Deep learning with Python*. Manning Publications Company.

Chordia, T., Goyal, A., and Shanken, J. (2015). Cross-sectional asset pricing with individual stocks: betas versus characteristics. *SSRN Working Paper*, 2549578.

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2015). Gated feedback recurrent neural networks. In *International Conference on Machine Learning*, pages 2067–2075.

Claeskens, G. and Hjort, N. L. (2008). *Model selection and model averaging*. Cambridge University Press.

Clark, T. E. and McCracken, M. W. (2009). Improving forecast accuracy by combining recursive and rolling forecasts. *International Economic Review*, 50(2):363–395.

Cocco, J. F., Gomes, F., and Lopes, P. (2019). Evidence on expectations of household finances. *SSRN Working Paper*, 3362495.

Cochrane, J. H. (2009). *Asset pricing: Revised edition*. Princeton university press.

Cong, L. W., Liang, T., and Zhang, X. (2019a). Analyzing textual information at scale. *SSRN Working Paper*, 3449822.

Cong, L. W., Liang, T., and Zhang, X. (2019b). Textual factors: A scalable, interpretable, and data-driven approach to analyzing unstructured information. *SSRN Working Paper*, 3307057.

Cong, L. W. and Xu, D. (2019). Rise of factor investing: asset prices, informational efficiency, and security design. *SSRN Working Paper*, 2800590.

Cont, R. (2007). Volatility clustering in financial markets: empirical facts and agent-based models. In *Long memory in economics*, pages 289–309. Springer.

Cooper, I. and Maio, P. F. (2019). New evidence on conditional factor models. *Journal of Financial and Quantitative Analysis*, 54(5):1975–2016.

Coqueret, G. (2015). Diversified minimum-variance portfolios. *Annals of Finance*, 11(2):221–241.

Coqueret, G. (2017). Approximate norta simulations for virtual sample generation. *Expert Systems with Applications*, 73:69–81.

Coqueret, G. (2018). The economic value of firm-specific news sentiment. *SSRN Working Paper*, 3248925.

Coqueret, G. and Guida, T. (2019). Training trees on tails with applications to portfolio choice. *SSRN Working Paper*, 3403009.

Cornuejols, A., Miclet, L., and Barra, V. (2018). *Apprentissage artificiel: Deep learning, concepts et algorithmes*. Eyrolles.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3):273–297.

Costarelli, D., Spigler, R., and Vinti, G. (2016). A survey on approximation by means of neural network operators. *Journal of NeuroTechnology*, 1(1).

Cover, T. M. (1991). Universal portfolios. *Mathematical Finance*, 1(1):1–29.

Crammer, K., Dekel, O., Keshet, J., Shalev-Shwartz, S., and Singer, Y. (2006). Online passive-aggressive algorithms. *Journal of Machine Learning Research*, 7(Mar):551–585.

Cronqvist, H., Previtero, A., Siegel, S., and White, R. E. (2015a). The fetal origins hypothesis in finance: Prenatal environment, the gender gap, and investor behavior. *Review of Financial Studies*, 29(3):739–786.

Cronqvist, H., Siegel, S., and Yu, F. (2015b). Value versus growth investing: Why do different investors have different styles? *Journal of Financial Economics*, 117(2):333–349.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314.

Dangl, T. and Halling, M. (2012). Predictive regressions with time-varying coefficients. *Journal of Financial Economics*, 106(1):157–181.

Daniel, K., Hirshleifer, D., and Sun, L. (2019). Short and long horizon behavioral factors. *Review of Financial Studies*, XXX(XXX).

Daniel, K. and Titman, S. (1997). Evidence on the characteristics of cross sectional variation in stock returns. *Journal of Finance*, 52(1):1–33.

Daniel, K. and Titman, S. (2012). Testing factor-model explanations of market anomalies. *Critical Finance Review*, 1(1):103–139.

Daniel, K., Titman, S., and Wei, K. J. (2001a). Explaining the cross-section of stock returns in japan: Factors or characteristics? *Journal of Finance*, 56(2):743–766.

Daniel, K. D., Hirshleifer, D., and Subrahmanyam, A. (2001b). Overconfidence, arbitrage, and equilibrium asset pricing. *Journal of Finance*, 56(3):921–965.

d'Aspremont, A. (2011). Identifying small mean-reverting portfolios. *Quantitative Finance*, 11(3):351–364.

De Moor, L., Dhaene, G., and Sercu, P. (2015). On comparing zero-alpha tests across multifactor asset pricing models. *Journal of Banking & Finance*, 61:S235–S240.

De Prado, M. L. (2018). *Advances in Financial Machine Learning*. John Wiley & Sons.

DeMiguel, V., Garlappi, L., and Uppal, R. (2007). Optimal versus naive diversification: How inefficient is the 1/n portfolio strategy? *Review of Financial Studies*, 22(5):1915–1953.

Denil, M., Matheson, D., and De Freitas, N. (2014). Narrowing the gap: Random forests in theory and in practice. In *International Conference on Machine Learning*, pages 665–673.

Dichtl, H., Drobetz, W., Neuhierl, A., and Wendt, V.-S. (2019). Data snooping in equity premium prediction. *SSRN Working Paper*, 2972011.

Dingli, A. and Fournier, K. S. (2017). Financial time series forecasting–a deep learning approach. *International Journal of Machine Learning and Computing*, 7(5):118–122.

Donaldson, R. G. and Kamstra, M. (1996). Forecast combining with neural networks. *Journal of Forecasting*, 15(1):49–61.

Drucker, H. (1997). Improving regressors using boosting techniques. In *International Conference on Machine Learning*, volume 97, pages 107–115.

Drucker, H., Burges, C. J., Kaufman, L., Smola, A. J., and Vapnik, V. (1997). Support vector regression machines. In *Advances in Neural Information Processing Systems*, pages 155–161.

Du, K.-L. and Swamy, M. N. (2013). *Neural networks and statistical learning*. Springer Science & Business Media.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

Dunis, C. L., Likothanassis, S. D., Karathanasopoulos, A. S., Sermpinis, G. S., and Theofilatos, K. A. (2013). A hybrid genetic algorithm–support vector machine approach in the task of forecasting and trading. *Journal of Asset Management*, 14(1):52–71.

Eakins, S. G., Stansell, S. R., and Buck, J. F. (1998). Analyzing the nature of institutional demand for common stocks. *Quarterly Journal of Business and Economics*, pages 33–48.

Elliott, G., Kudrin, N., and Wuthrich, K. (2019). Detecting p-hacking. *arXiv Preprint*, (1906.06711).

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14(2):179–211.

Enders, C. K. (2001). A primer on maximum likelihood algorithms available for use with missing data. *Structural Equation Modeling*, 8(1):128–141.

Enders, C. K. (2010). *Applied missing data analysis*. Guilford press.

Engle, R. F. (1982). Autoregressive conditional heteroscedasticity with estimates of the variance of united kingdom inflation. *Econometrica*, pages 987–1007.

Enke, D. and Thawornwong, S. (2005). The use of data mining and neural networks for forecasting stock market returns. *Expert Systems with Applications*, 29(4):927–940.

Fabozzi, F. J. and de Prado, M. L. (2018). Being honest in backtest reporting: A template for disclosing multiple tests. *Journal of Portfolio Management*, 45(1):141–147.

Fama, E. F. and French, K. R. (1992). The cross-section of expected stock returns. *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 French, K. R. (2015). A five-factor asset pricing model. *Journal of Financial Economics*, 116(1):1–22.

Fama, E. F. and French, K. R. (2018). Choosing factors. *Journal of Financial Economics*, 128(2):234–252.

Fama, E. F. and MacBeth, J. D. (1973). Risk, return, and equilibrium: Empirical tests. *Journal of Political Economy*, 81(3):607–636.

Farmer, L., Schmidt, L., and Timmermann, A. (2019). Pockets of predictability. *SSRN Working Paper*, 3152386.

Fastrich, B., Paterlini, S., and Winker, P. (2015). Constructing optimal sparse portfolios using regularization methods. *Computational Management Science*, 12(3):417–434.

Feng, G., Giglio, S., and Xiu, D. (2019a). Taming the factor zoo: A test of new factors. *Journal of Finance*, Forthcoming.

Feng, G., Polson, N. G., and Xu, J. (2019b). Deep learning in characteristics-sorted factor models. *SSRN Working Paper*, 3243683.

Fischer, T. and Krauss, C. (2018). Deep learning with long short-term memory networks for financial market predictions. *European Journal of Operational Research*, 270(2):654–669.

Fisher, A., Rudin, C., and Dominici, F. (2018). All models are wrong but many are useful: Variable importance for black-box, proprietary, or misspecified prediction models, using model class reliance. *arXiv Preprint*, (1801.01489).

Frazier, P. I. (2018). A tutorial on bayesian optimization. *arXiv Preprint*, (1807.02811).

Frazzini, A. and Pedersen, L. H. (2014). Betting against beta. *Journal of Financial Economics*, 111(1):1–25.

Freeman, R. N. and Tse, S. Y. (1992). A nonlinear model of security price responses to unexpected earnings. *Journal of Accounting Research*, pages 185–209.

Freund, Y. and Schapire, R. E. (1996). Experiments with a new boosting algorithm. In *Machine Learning: Proceedings of the Thirteenth International Conference*, volume 96, pages 148–156.

Freund, Y. and Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139.

Friedman, J., Hastie, T., and Tibshirani, R. (2008). Sparse inverse covariance estimation with the graphical lasso. *Biostatistics*, 9(3):432–441.

Friedman, J., Hastie, T., Tibshirani, R., et al. (2000). Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *Annals of Statistics*, 28(2):337–407.

Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, pages 1189–1232.

Friedman, J. H. (2002). Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378.

Friedman, N., Geiger, D., and Goldszmidt, M. (1997). Bayesian network classifiers. *Machine Learning*, 29(2-3):131–163.

Fu, X., Du, J., Guo, Y., Liu, M., Dong, T., and Duan, X. (2018). A machine learning framework for stock selection. *arXiv Preprint*, (1806.01743).

Gaba, A., Tsetlin, I., and Winkler, R. L. (2017). Combining interval forecasts. *Decision Analysis*, 14(1):1–20.

Gagliardini, P., Ossola, E., and Scaillet, O. (2016). Time-varying risk premium in large cross-sectional equity data sets. *Econometrica*, 84(3):985–1046.

Gagliardini, P., Ossola, E., and Scaillet, O. (2019). Estimation of large dimensional conditional factor models in finance. *SSRN Working Paper*, 3443426.

Galili, T. and Meilijson, I. (2016). Splitting matters: how monotone transformation of predictor variables may improve the predictions of decision tree models. *arXiv Preprint*, (1611.04561).

García-Galicia, M., Carsteanu, A. A., and Clempner, J. B. (2019). Continuous-time reinforcement learning approach for portfolio management with time penalization. *Expert Systems with Applications*, 129:27–36.

García-Laencina, P. J., Sancho-Gómez, J.-L., Figueiras-Vidal, A. R., and Verleysen, M. (2009). K nearest neighbours with mutual information for simultaneous classification and missing data imputation. *Neurocomputing*, 72(7-9):1483–1493.

Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., and Rubin, D. B. (2013). *Bayesian Data Analysis, 3rd Edition.* Chapman and Hall/CRC.

Geman, S., Bienenstock, E., and Doursat, R. (1992). Neural networks and the bias/variance dilemma. *Neural computation*, 4(1):1–58.

Genre, V., Kenny, G., Meyler, A., and Timmermann, A. (2013). Combining expert forecasts: Can anything beat the simple average? *International Journal of Forecasting*, 29(1):108–121.

Gentzkow, M., Kelly, B., and Taddy, M. (2019). Text as data. *Journal of Economic Literature*, 57(3):535–74.

Ghosh, A. K. (2006). On optimum choice of k in nearest neighbor classification. *Computational Statistics & Data Analysis*, 50(11):3113–3123.

Giglio, S. and Xiu, D. (2018). Asset pricing with omitted factors. *SSRN Working Paper*, 2865922.

Gomes, J., Kogan, L., and Zhang, L. (2003). Equilibrium cross section of returns. *Journal of Political Economy*, 111(4):693–732.

Gonzalo, J. and Pitarakis, J.-Y. (2018). Predictive regressions. In *Oxford Research Encyclopedia of Economics and Finance.*

Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. (2016). *Deep learning*, volume 1. MIT press Cambridge.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pages 2672–2680.

Gospodinov, N., Kan, R., and Robotti, C. (2019). Too good to be true? Fallacies in evaluating risk factor models. *Journal of Financial Economics*, 132(2):451–471.

Goto, S. and Xu, Y. (2015). Improving mean variance optimization through sparse hedging restrictions. *Journal of Financial and Quantitative Analysis*, 50(6):1415–1441.

Gower, J. C. (1971). A general coefficient of similarity and some of its properties. *Biometrics*, pages 857–871.

Goyal, A. (2012). Empirical cross-sectional asset pricing: a survey. *Financial Markets and Portfolio Management*, 26(1):3–38.

Granger, C. W. (1969). Investigating causal relations by econometric models and cross-spectral methods. *Econometrica*, pages 424–438.

Green, J., Hand, J. R., and Zhang, X. F. (2013). The supraview of return predictive signals. *Review of Accounting Studies*, 18(3):692–730.

Greene, W. H. (2018). *Econometric analysis, Eighth Edition.* Pearson Education.

Grinblatt, M. and Han, B. (2005). Prospect theory, mental accounting, and momentum. *Journal of Financial Economics*, 78(2):311–339.

Grushka-Cockayne, Y., Jose, V. R. R., and Lichtendahl Jr, K. C. (2016). Ensembles of overfit and overconfident forecasts. *Management Science*, 63(4):1110–1130.

Gu, S., Kelly, B. T., and Xiu, D. (2018). Empirical asset pricing via machine learning. *SSRN Working Paper*, 3159577.

Gu, S., Kelly, B. T., and Xiu, D. (2019). Autoencoder asset pricing models. *SSRN Working Paper*, 3335536.

Guida, T. and Coqueret, G. (2018a). Ensemble learning applied to quant equity: gradient boosting in a multifactor framework. In *Big Data and Machine Learning in Quantitative Investment*, pages 129–148. Wiley.

Guida, T. and Coqueret, G. (2018b). Machine learning in systematic equity allocation: A model comparison. *Wilmott*, 2018(98):24–33.

Guliyev, N. J. and Ismailov, V. E. (2018). On the approximation by single hidden layer feedforward neural networks with fixed weights. *Neural Networks*, 98:296–304.

Gupta, M., Gao, J., Aggarwal, C., and Han, J. (2014). Outlier detection for temporal data. *IEEE Transactions on Knowledge and Data Engineering*, 26(9):2250 − 2267.

Guresen, E., Kayakutlu, G., and Daim, T. U. (2011). Using artificial neural network models in stock market index prediction. *Expert Systems with Applications*, 38(8):10389–10397.

Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *Journal of Lachine Learning Research*, 3(Mar):1157–1182.

Hall, P. and Gill, N. (2019). *An Introduction to Machine Learning Interpretability-Second Edition.* O'Reilly.

Hall, P., Park, B. U., Samworth, R. J., et al. (2008). Choice of neighbor order in nearest-neighbor classification. *Annals of Statistics*, 36(5):2135–2152.

Halperin, I. and Feldshteyn, I. (2018). Market self-learning of signals, impact and optimal trading: Invisible hand inference with free energy. *arXiv Preprint*, (1805.06126).

Han, Y., He, A., Rapach, D., and Zhou, G. (2019). Firm characteristics and expected stock returns. *SSRN Working Paper*, 3185335.

Hansen, L. P. (1982). Large sample properties of generalized method of moments estimators. *Econometrica*, pages 1029–1054.

Harrald, P. G. and Kamstra, M. (1997). Evolving artificial neural networks to combine financial forecasts. *IEEE Transactions on Evolutionary Computation*, 1(1):40–52.

Harvey, C. and Liu, Y. (2017). Lucky factors. *SSRN Working Paper*, 2528780.

Harvey, C. R. (2017). Presidential address: the scientific outlook in financial economics. *Journal of Finance*, 72(4):1399–1440.

Harvey, C. R., Liechty, J. C., Liechty, M. W., and Müller, P. (2010). Portfolio selection with higher moments. *Quantitative Finance*, 10(5):469–485.

Harvey, C. R. and Liu, Y. (2015). Backtesting. *Journal of Portfolio Management*, 42(1):13–28.

Harvey, C. R. and Liu, Y. (2019). A census of the factor zoo. *SSRN Working Paper*, 3341728.

Harvey, C. R., Liu, Y., and Zhu, H. (2016). ... and the cross-section of expected returns. *Review of Financial Studies*, 29(1):5–68.

Hassan, M. R., Nath, B., and Kirley, M. (2007). A fusion model of hmm, ann and ga for stock market forecasting. *Expert Systems with Applications*, 33(1):171–180.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning*. Springer.

Haykin, S. S. (2009). *Neural networks and learning machines*. Prentice Hall.

Hazan, E., Agarwal, A., and Kale, S. (2007). Logarithmic regret algorithms for online convex optimization. *Machine Learning*, 69(2-3):169–192.

Hazan, E. et al. (2016). Introduction to online convex optimization. *Foundations and Trends® in Optimization*, 2(3-4):157–325.

Head, M. L., Holman, L., Lanfear, R., Kahn, A. T., and Jennions, M. D. (2015). The extent and consequences of p-hacking in science. *PLoS biology*, 13(3):e1002106.

Heinze-Deml, C., Peters, J., and Meinshausen, N. (2018). Invariant causal prediction for nonlinear models. *Journal of Causal Inference*, 6(2).

Henkel, S. J., Martin, J. S., and Nardari, F. (2011). Time-varying short-horizon predictability. *Journal of Financial Economics*, 99(3):560–580.

Henrique, B. M., Sobreiro, V. A., and Kimura, H. (2019). Literature review: Machine learning techniques applied to financial market prediction. *Expert Systems with Applications*, XXX(XXX).

Hiemstra, C. and Jones, J. D. (1994). Testing for linear and nonlinear granger causality in the stock price-volume relation. *Journal of Finance*, 49(5):1639–1664.

Ho, T. K. (1995). Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282. IEEE.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Hodge, V. and Austin, J. (2004). A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126.

Hoechle, D., Schmid, M., and Zimmermann, H. (2018). Correcting alpha misattribution in portfolio sorts. *SSRN Working Paper*, 3190310.

Hoi, S. C., Sahoo, D., Lu, J., and Zhao, P. (2018). Online learning: A comprehensive survey. *arXiv Preprint*, (1802.02871).

Honaker, J. and King, G. (2010). What to do about missing values in time-series cross-section data. *American Journal of Political Science*, 54(2):561–581.

Horel, E. and Giesecke, K. (2019). Towards explainable AI: Significance tests for neural networks. *arXiv Preprint*, (1902.06021).

Hoseinzade, E. and Haratizadeh, S. (2019). Cnnpred: CNN-based stock market prediction using a diverse set of variables. *Expert Systems with Applications*, XXX.

Hou, K., Xue, C., and Zhang, L. (2015). Digesting anomalies: An investment approach. *Review of Financial Studies*, 28(3):650–705.

Hou, K., Xue, C., and Zhang, L. (2019). Replicating anomalies. *Review of Financial Studies*, XX(XX):XXX–XX.

Hsu, P.-H., Han, Q., Wu, W., and Cao, Z. (2018). Asset allocation strategies, data snooping, and the 1/n rule. *Journal of Banking & Finance*, 97:257–269.

Huang, W., Nakamori, Y., and Wang, S.-Y. (2005). Forecasting stock market movement direction with support vector machine. *Computers & Operations Research*, 32(10):2513–2522.

Hübner, G. (2005). The generalized treynor ratio. *Review of Finance*, 9(3):415–435.

Huck, N. (2019). Large data sets and machine learning: Applications to statistical arbitrage. *European Journal of Operational Research*, 278(1):330–342.

Ilmanen, A. (2011). *Expected returns: An investor's guide to harvesting market rewards*. John Wiley & Sons.

Ilmanen, A., Israel, R., Moskowitz, T. J., Thapar, A. K., and Wang, F. (2019). Factor premia and factor timing: A century of evidence. *SSRN Working Paper*, 3400998.

Jacobs, H. and Müller, S. (2019). Anomalies across the globe: Once public, no longer existent? *Journal of Financial Economics*, XXX(XXX):XXX–XXX.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., Hinton, G. E., et al. (1991). Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87.

Jagannathan, R. and Ma, T. (2003). Risk reduction in large portfolios: Why imposing the wrong constraints helps. *Journal of Finance*, 58(4):1651–1683.

Jagannathan, R. and Wang, Z. (1998). An asymptotic theory for estimating beta-pricing models using cross-sectional regression. *Journal of Finance*, 53(4):1285–1309.

Jegadeesh, N., Noh, J., Pukthuanthong, K., Roll, R., and Wang, J. L. (2019). Empirical tests of asset pricing models with individual assets: Resolving the errors-in-variables bias in risk premium estimation. *Journal of Financial Economics*, Forthcoming(XXXX).

Jegadeesh, N. and Titman, S. (1993). Returns to buying winners and selling losers: Implications for stock market efficiency. *Journal of Finance*, 48(1):65–91.

Jensen, M. C. (1968). The performance of mutual funds in the period 1945–1964. *Journal of Finance*, 23(2):389–416.

Jha, V. (2019). Implementing alternative data in an investment process. In *Big Data and Machine Learning in Quantitative Investment*, pages 51–74. Wiley.

Jiang, Z., Xu, D., and Liang, J. (2017). A deep reinforcement learning framework for the financial portfolio management problem. *arXiv Preprint*, (1706.10059).

Johnson, T. C. (2002). Rational momentum effects. *Journal of Finance*, 57(2):585–608.

Jordan, M. I. (1997). Serial order: A parallel distributed processing approach. In *Advances in psychology*, volume 121, pages 471–495. Elsevier.

Jurczenko, E. (2017). *Factor Investing: From Traditional to Alternative Risk Premia.* Elsevier.

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. (2017). Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3146–3154.

Ke, Z. T., Kelly, B. T., and Xiu, D. (2019). Predicting returns with text data. *SSRN Working Paper*, 3388293.

Kelly, B., Pruitt, S., and Su, Y. (2019). Characteristics are covariances: A unified model of risk and return. *Journal of Financial Economics*, Forthcoming(XXX):XXX–XXX.

Kim, K.-j. (2003). Financial time series forecasting using support vector machines. *Neurocomputing*, 55(1-2):307–319.

Kim, S., Korajczyk, R. A., and Neuhierl, A. (2019). Arbitrage portfolios. *SSRN Working Paper*, 3263001.

Kimoto, T., Asakawa, K., Yoda, M., and Takeoka, M. (1990). Stock market prediction system with modular neural networks. In *1990 IJCNN international joint conference on neural networks*, pages 1–6. IEEE.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv Preprint*, (1412.6980).

Koijen, R. S., Richmond, R. J., and Yogo, M. (2019). Which investors matter for global equity valuations and expected returns? *SSRN Working Paper*, 3378340.

Koijen, R. S. and Yogo, M. (2019). A demand system approach to asset pricing. *Journal of Political Economy*, 127(4):1475–1515.

Kolm, P. N. and Ritter, G. (2019a). Dynamic replication and hedging: A reinforcement learning approach. *The Journal of Financial Data Science*, 1(1):159–171.

Kolm, P. N. and Ritter, G. (2019b). Modern perspectives on reinforcement learning in finance. *Journal of Machine Learning in Finance*, 1(1).

Kozak, S., Nagel, S., and Santosh, S. (2018). Interpreting factor models. *Journal of Finance*, 73(3):1183–1223.

Krauss, C., Do, X. A., and Huck, N. (2017). Deep neural networks, gradient-boosted trees, random forests: Statistical arbitrage on the s&p 500. *European Journal of Operational Research*, 259(2):689–702.

Kremer, P. J., Lee, S., Bogdan, M., and Paterlini, S. (2019). Sparse portfolio selection via the sorted l1-norm. *Journal of Banking & Finance*, page 105687.

Krkoska, E. and Schenk-Hoppé, K. R. (2019). Herding in smart-beta investment products. *Journal of Risk and Financial Management*, 12(1):47.

Kruschke, J. (2014). *Doing Bayesian Data Analysis: A tutorial with R, JAGS, and Stan (2nd Ed.).* Academic Press.

Kuhn, M. and Johnson, K. (2019). *Feature Engineering and Selection: A Practical Approach for Predictive Models.* CRC Press.

Lakonishok, J., Shleifer, A., and Vishny, R. W. (1994). Contrarian investment, extrapolation, and risk. *Journal of Finance*, 49(5):1541–1578.

Leary, M. T. and Michaely, R. (2011). Determinants of dividend smoothing: Empirical evidence. *Review of Financial Studies*, 24(10):3197–3249.

Ledoit, O. and Wolf, M. (2004). A well-conditioned estimator for large-dimensional covariance matrices. *Journal of Multivariate Analysis*, 88(2):365–411.

Ledoit, O. and Wolf, M. (2008). Robust performance hypothesis testing with the sharpe ratio. *Journal of Empirical Finance*, 15(5):850–859.

Ledoit, O. and Wolf, M. (2017). Nonlinear shrinkage of the covariance matrix for portfolio selection: Markowitz meets goldilocks. *Review of Financial Studies*, 30(12):4349–4388.

Ledoit, O., Wolf, M., and Zhao, Z. (2018). Efficient weighting: A more powerful test for cross-sectional anomalies. *SSRN Working Paper*, 2881417.

Legendre, A. M. (1805). *Nouvelles méthodes pour la détermination des orbites des comètes*. F. Didot.

Lempérière, Y., Deremble, C., Seager, P., Potters, M., and Bouchaud, J.-P. (2014). Two centuries of trend following. *arXiv Preprint*, (1404.3274).

Lettau, M. and Pelger, M. (2018). Factors that fit the time series and cross-section of stock returns. *SSRN Working Paper*, 3211106.

Lettau, M. and Pelger, M. (201XXX). Estimating latent asset-pricing factors. *Journal of Econometrics*, XXX:XXX–XXX.

Leung, M. T., Daouk, H., and Chen, A.-S. (2001). Using investment portfolio return to combine forecasts: A multiobjective approach. *European Journal of Operational Research*, 134(1):84–102.

Linnainmaa, J. T. and Roberts, M. R. (2018). The history of the cross-section of stock returns. *Review of Financial Studies*, 31(7):2606–2649.

Lintner, J. (1965). The valuation of risk assets and the selection of risky investments in stock portfolios and capital budgets. *Review of Economics and Statistics*, 47(1):13–37.

Little, R. J. and Rubin, D. B. (2014). *Statistical analysis with missing data*, volume 333. John Wiley & Sons.

Loreggia, A., Malitsky, Y., Samulowitz, H., and Saraswat, V. (2016). Deep learning for algorithm portfolios. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 1280–1286. AAAI Press.

Loughran, T. and McDonald, B. (2016). Textual analysis in accounting and finance: A survey. *Journal of Accounting Research*, 54(4):1187–1230.

Lundberg, S. M. and Lee, S.-I. (2017). A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems*, pages 4765–4774.

Ma, S., Lan, W., Su, L., and Tsai, C.-L. (2018). Testing alphas in conditional time-varying factor models with high dimensional assets. *Journal of Business & Economic Statistics*, XXX(XXX):1–34.

Maathuis, M., Drton, M., Lauritzen, S., and Wainwright, M. (2018). *Handbook of Graphical Models*. CRC Press.

Maclaurin, D., Duvenaud, D., and Adams, R. (2015). Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, pages 2113–2122.

Maillard, S., Roncalli, T., and Teiletche, J. (2010). The properties of equally weighted risk contribution portfolios. *Journal of Portfolio Management*, 36(4):60–70.

Markowitz, H. (1952). Portfolio selection. *Journal of finance*, 7(1):77–91.

Martin, I. and Nagel, S. (2019). Market efficiency in the age of big data. *SSRN Working Paper*, XXXXXXXXXXXXX.

Martin Utrera, A., DeMiguel, V., Uppal, R., and Nogales, F. J. (2018). A transaction-cost perspective on the multitude of firm characteristics. *SSRN Working Paper*, 2912819.

Mason, L., Baxter, J., Bartlett, P. L., and Frean, M. R. (2000). Boosting algorithms as gradient descent. In *Advances in neural information processing systems*, pages 512–518.

Masters, T. (1993). *Practical neural network recipes in C++*. Morgan Kaufmann.

Matías, J. M. and Reboredo, J. C. (2012). Forecasting performance of nonlinear models for intraday stock returns. *Journal of Forecasting*, 31(2):172–188.

McLean, R. D. and Pontiff, J. (2016). Does academic research destroy stock return predictability? *Journal of Finance*, 71(1):5–32.

Meng, T. L. and Khushi, M. (2019). Reinforcement learning in financial markets. *Data*, 4(3):110.

Metropolis, N. and Ulam, S. (1949). The monte carlo method. *Journal of the American statistical association*, 44(247):335–341.

Meyer, C. D. (2000). *Matrix analysis and applied linear algebra*, volume 71. SIAM.

Molnar, C. (2019). Interpretable machine learning: A guide for making black box models explainable.

Moody, J. and Wu, L. (1997). Optimization of trading systems and portfolios. In *Proceedings of the IEEE/IAFE 1997 Computational Intelligence for Financial Engineering (CIFEr)*, pages 300–307. IEEE.

Moody, J., Wu, L., Liao, Y., and Saffell, M. (1998). Performance functions and reinforcement learning for trading systems and portfolios. *Journal of Forecasting*, 17(5-6):441–470.

Moritz, B. and Zimmermann, T. (2016). Tree-based conditional portfolio sorts: The relation between past and future stock returns. *SSRN Working Paper*, 2740751.

Moskowitz, T. J., Ooi, Y. H., and Pedersen, L. H. (2012). Time series momentum. *Journal of financial economics*, 104(2):228–250.

Mossin, J. (1966). Equilibrium in a capital asset market. *Econometrica: Journal of the econometric society*, 34(4):768–783.

Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2). In *Doklady AN USSR*, volume 269, pages 543–547.

Neuneier, R. (1996). Optimal asset allocation using adaptive dynamic programming. In *Advances in Neural Information Processing Systems*, pages 952–958.

Neuneier, R. (1998). Enhancing q-learning for optimal asset allocation. In *Advances in neural information processing systems*, pages 936–942.

Ngai, E. W., Hu, Y., Wong, Y., Chen, Y., and Sun, X. (2011). The application of data mining techniques in financial fraud detection: A classification framework and an academic review of literature. *Decision support systems*, 50(3):559–569.

Novy-Marx, R. and Velikov, M. (2015). A taxonomy of anomalies and their trading costs. *Review of Financial Studies*, 29(1):104–147.

Okun, O., Valentini, G., and Re, M. (2011). *Ensembles in machine learning applications*, volume 373. Springer Science & Business Media.

Olazaran, M. (1996). A sociological study of the official history of the perceptrons controversy. *Social Studies of Science*, 26(3):611–659.

Orimoloye, L. O., Sung, M.-C., Ma, T., and Johnson, J. E. (2019). Comparing the effectiveness of deep feedforward neural networks and shallow architectures for predicting stock price indices. *Expert Systems with Applications*, page 112828.

Pan, S. J. and Yang, Q. (2009). A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359.

Patel, J., Shah, S., Thakkar, P., and Kotecha, K. (2015a). Predicting stock and stock price index movement using trend deterministic data preparation and machine learning techniques. *Expert Systems with Applications*, 42(1):259–268.

Patel, J., Shah, S., Thakkar, P., and Kotecha, K. (2015b). Predicting stock market index using fusion of machine learning techniques. *Expert Systems with Applications*, 42(4):2162–2172.

Patton, A. J. and Timmermann, A. (2010). Monotonicity in asset returns: New tests with applications to the term structure, the CAPM, and portfolio sorts. *Journal of Financial Economics*, 98(3):605–625.

Pearl, J. (2009). *Causality: Models, Reasoning and Inference. Second Edition*, volume 29. Cambridge University Press.

Penasse, J. (2018). Understanding alpha decay. *SSRN Working Paper*, 2953614.

Pendharkar, P. C. and Cusatis, P. (2018). Trading financial indices with reinforcement learning agents. *Expert Systems with Applications*, 103:1–13.

Perrin, S. and Roncalli, T. (2019). Machine learning optimization algorithms & portfolio allocation. *SSRN Working Paper*, 3425827.

Peters, J., Janzing, D., and Schölkopf, B. (2017). *Elements of causal inference: foundations and learning algorithms*. MIT press.

Petersen, M. A. (2009). Estimating standard errors in finance panel data sets: Comparing approaches. *Review of Financial Studies*, 22(1):435–480.

Plyakha, Y., Uppal, R., and Vilkov, G. (2014). Equal or value weighting? implications for asset-pricing tests. *SSRN Working Paper*, 1787045.

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17.

Popov, S., Morozov, S., and Babenko, A. (2019). Neural oblivious decision ensembles for deep learning on tabular data. *arXiv Preprint*, (1909.06312).

Powell, W. B. and Ma, J. (2011). A review of stochastic algorithms with continuous value function approximation and some new approximate policy iteration algorithms for multidimensional continuous applications. *Journal of Control Theory and Applications*, 9(3):336–352.

Probst, P., Bischl, B., and Boulesteix, A.-L. (2018). Tunability: Importance of hyperparameters of machine learning algorithms. *arXiv Preprint*, (1802.09596).

Pukthuanthong, K., Roll, R., and Subrahmanyam, A. (2018). A protocol for factor identification. *Review of Financial Studies*, 32(4):1573–1607.

Quionero-Candela, J., Sugiyama, M., Schwaighofer, A., and Lawrence, N. D. (2009). *Dataset shift in machine learning*. The MIT Press.

Rapach, D. and Zhou, G. (2019). Time-series and cross-sectional stock return forecasting: New machine learning methods. *SSRN Working Paper*, 3428095.

Rapach, D. E., Strauss, J. K., and Zhou, G. (2013). International stock return predictability: what is the role of the united states? *Journal of Finance*, 68(4):1633–1662.

Rashmi, K. V. and Gilad-Bachrach, R. (2015). Dart: Dropouts meet multiple additive regression trees. In *AISTATS*, pages 489–497.

Ravisankar, P., Ravi, V., Rao, G. R., and Bose, I. (2011). Detection of financial statement fraud and feature selection using data mining techniques. *Decision Support Systems*, 50(2):491–500.

Reboredo, J. C., Matías, J. M., and Garcia-Rubio, R. (2012). Nonlinearity in forecasting of high-frequency stock returns. *Computational Economics*, 40(3):245–264.

Regenstein, J. K. (2018). *Reproducible Finance with R: Code Flows and Shiny Apps for Portfolio Analysis*. Chapman and Hall/CRC.

Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). Why should I trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144. ACM.

Ridgeway, G., Madigan, D., and Richardson, T. (1999). Boosting methodology for regression problems. In *AISTATS*.

Roberts, G. O. and Smith, A. F. (1994). Simple conditions for the convergence of the gibbs sampler and metropolis-hastings algorithms. *Stochastic Processes and their Applications*, 49(2):207–216.

Romano, J. P. and Wolf, M. (2005). Stepwise multiple testing as formalized data snooping. *Econometrica*, 73(4):1237–1282.

Romano, J. P. and Wolf, M. (2013). Testing for monotonicity in expected asset returns. *Journal of Empirical Finance*, 23:93–116.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386.

Ross, S. A. (1976). The arbitrage theory of capital asset pricing. *Journal of Economic Theory*, 13(3):341–60.

Rousseeuw, P. J. and Leroy, A. M. (2005). *Robust regression and outlier detection*, volume 589. Wiley.

Ruf, J. and Wang, W. (2019). Neural networks for option pricing and hedging: a literature review. *arXiv Preprint*, (1911.05620).

Santi, C. and Zwinkels, R. C. (2018). Exploring style herding by mutual funds. *SSRN Working Paper*, 2986059.

Sato, Y. (2019). Model-free reinforcement learning for financial portfolios: A brief survey. *arXiv Preprint*, (1904.04973).

Schafer, J. L. (1999). Multiple imputation: a primer. *Statistical methods in medical research*, 8(1):3–15.

Schapire, R. E. (1990). The strength of weak learnability. *Machine learning*, 5(2):197–227.

Schapire, R. E. (2003). The boosting approach to machine learning: An overview. In *Nonlinear estimation and classification*, pages 149–171. Springer.

Schapire, R. E. and Freund, Y. (2012). *Boosting: Foundations and algorithms*. MIT press.

Scornet, E., Biau, G., Vert, J.-P., et al. (2015). Consistency of random forests. *Annals of Statistics*, 43(4):1716–1741.

Seni, G. and Elder, J. F. (2010). Ensemble methods in data mining: improving accuracy through combining predictions. *Synthesis lectures on data mining and knowledge discovery*, 2(1):1–126.

Settles, B. (2009). Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences.

Settles, B. (2012). Active learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(1):1–114.

Shah, A. D., Bartlett, J. W., Carpenter, J., Nicholas, O., and Hemingway, H. (2014). Comparison of random forest and parametric imputation models for imputing missing data using mice: a caliber study. *American Journal of Epidemiology*, 179(6):764–774.

Shanken, J. (1992). On the estimation of beta-pricing models. *Review of Financial Studies*, 5(1):1–33.

Shapley, L. S. (1953). A value for n-person games. *Contributions to the Theory of Games*, 2(28):307–317.

Sharpe, W. F. (1964). Capital asset prices: A theory of market equilibrium under conditions of risk. *Journal of Finance*, 19(3):425–442.

Sharpe, W. F. (1966). Mutual fund performance. *Journal of Business*, 39(1):119–138.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., and Lanctot, M. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529:484–489.

Simonian, J., Wu, C., Itano, D., and Narayanam, V. (2019). A machine learning approach to risk factors: A case study using the Fama-French-Carhart model. *Journal of Financial Data Science*, 1(1):32–44.

Simonsohn, U., Nelson, L. D., and Simmons, J. P. (2014). P-curve: a key to the file-drawer. *Journal of Experimental Psychology: General*, 143(2):534.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, pages 2951–2959.

Sparapani, R., Spanbauer, C., and McCulloch, R. (2019). The bart r package. Technical report, Comprehensive R Archive Network.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958.

Stambaugh, R. F. (1999). Predictive regressions. *Journal of Financial Economics*, 54(3):375–421.

Staniak, M. and Biecek, P. (2018). Explanations of model predictions with live and breakdown packages. *arXiv Preprint*, (1804.01955).

Stekhoven, D. J. and Bühlmann, P. (2011). Missforest—non-parametric missing value imputation for mixed-type data. *Bioinformatics*, 28(1):112–118.

Stevens, G. V. (1998). On the inverse of the covariance matrix in portfolio analysis. *Journal of Finance*, 53(5):1821–1827.

Suhonen, A., Lennkh, M., and Perez, F. (2017). Quantifying backtest overfitting in alternative beta strategies. *Journal of Portfolio Management*, 43(2):90–104.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction (2nd Edition)*. MIT press.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288.

Tierney, L. (1994). Markov chains for exploring posterior distributions. *Annals of Statistics*, pages 1701–1728.

Timmermann, A. (2018). Forecasting methods in finance. *Annual Review of Financial Economics*, 10:449–479.

Ting, K. M. (2002). An instance-weighting method to induce cost-sensitive trees. *IEEE Transactions on Knowledge & Data Engineering*, (3):659–665.

Treynor, J. L. (1965). How to rate management of investment funds. *Harvard Business Review*, 43(1):63–75.

Tsantekidis, A., Passalis, N., Tefas, A., Kanniainen, J., Gabbouj, M., and Iosifidis, A. (2017). Forecasting stock prices from the limit order book using convolutional neural networks. In *2017 IEEE 19th Conference on Business Informatics (CBI)*, volume 1, pages 7–12.

Uematsu, Y. and Tanaka, S. (2019). High-dimensional macroeconomic forecasting and variable selection via penalized regression. *Econometrics Journal*, 22(1):34–56.

Van Buuren, S. (2018). *Flexible imputation of missing data.* Chapman and Hall/CRC.

Van Dijk, M. A. (2011). Is size dead? A review of the size effect in equity returns. *Journal of Banking & Finance*, 35(12):3263–3274.

Vapnik, V. and Lerner, A. (1963). Pattern recognition using generalized portrait method. *Automation and Remote Control*, 24:774–780.

Vayanos, D. and Woolley, P. (2013). An institutional theory of momentum and reversal. *Review of Financial Studies*, 26(5):1087–1145.

Virtanen, I. and Yli-Olli, P. (1987). Forecasting stock market prices in a thin security market. *Omega*, 15(2):145–155.

Von Holstein, C.-A. S. S. (1972). Probabilistic forecasting: An experiment related to the stock market. *Organizational Behavior and Human Performance*, 8(1):139–158.

Wang, G., Hao, J., Ma, J., and Jiang, H. (2011). A comparative assessment of ensemble learning for credit scoring. *Expert Systems with Applications*, 38(1):223–230.

Wang, H. and Zhou, X. Y. (2019). Continuous-time mean-variance portfolio selection: A reinforcement learning framework. *SSRN Working Paper*, 3382932.

Wang, J.-J., Wang, J.-Z., Zhang, Z.-G., and Guo, S.-P. (2012). Stock index forecasting based on a hybrid model. *Omega*, 40(6):758–766.

Wang, W., Li, W., Zhang, N., and Liu, K. (2019). Portfolio formation with preselection using deep learning from long-term financial data. *Expert Systems with Applications*, Forthcoming:XXX–XXX.

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292.

Weiss, K., Khoshgoftaar, T. M., and Wang, D. (2016). A survey of transfer learning. *Journal of Big Data*, 3(1):9.

White, H. (1988). Economic prediction using neural networks: The case of ibm daily stock returns. In *IEEE 1988 International Conference on Neural Networks*.

White, H. (2000). A reality check for data snooping. *Econometrica*, 68(5):1097–1126.

Widrow, B. and Hoff, M. E. (1960). Adaptive switching circuits. In *IRE WESCON Convention Record*, volume 4, pages 96–104.

Wolpert, D. H. (1992). Stacked generalization. *Neural networks*, 5(2):241–259.

Xiong, Z., Liu, X.-Y., Zhong, S., Yang, H., and Walid, A. (2018). Practical deep reinforcement learning approach for stock trading. *arXiv Preprint*, (1811.07522).

Yang, S. Y., Yu, Y., and Almahdi, S. (2018). An investor sentiment reward-based trading system using gaussian inverse reinforcement learning algorithm. *Expert Systems with Applications*, 114:388–401.

Yu, P., Lee, J. S., Kulyatin, I., Shi, Z., and Dasgupta, S. (2019). Model-based deep reinforcement learning for dynamic portfolio optimization. *arXiv Preprint*, (1901.08740).

Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv Preprint*, (1212.5701).

Zhang, C. and Ma, Y. (2012). *Ensemble machine learning: methods and applications*. Springer.

Zhang, Y. and Wu, L. (2009). Stock market prediction of s&p 500 via combination of improved bco approach and bp neural network. *Expert Systems with Applications*, 36(5):8849–8854.

Zhao, Q. and Hastie, T. (2019). Causal interpretations of black-box models. *Journal of Business & Economic Statistics*, (just-accepted):1–19.

Zhou, Z.-H. (2012). *Ensemble methods: foundations and algorithms*. Chapman and Hall/CRC.

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.