Diploma Thesis

Höhere Technische Bundeslehranstalt Leonding
Abteilung für Informatics

Analyzing the Stock Market Using Machine Learning

Submitted by: **Leon Schlömmer, 5BHIF**
Lukas Stransky, 5BHIF
Date: **April 6, 2020**
Supervisor: **Peter Bauer**

Declaration of Academic Honesty

Hereby, I declare that I have composed the presented paper independently on my own and without any other resources than the ones indicated. All thoughts taken directly or indirectly from external sources are properly denoted as such.

This paper has neither been previously submitted to another authority nor has it been published yet.

Leonding, April 6, 2020

Leon Schlömmer, Lukas Stransky

Abstract

Since the opening of the first stock exchange people have been interested in special ways of analyzing the stock market. But only when the internet started to establish, exchanging stocks got available to truly everyone, and with that the possibility came up to perform *shorter* term stock exchanging, buying a stock only as an investment opportunity, and not only to fund the company behind the stock. Automated trading bots and servers which make millions of hardcoded decisions every minute have been available to professionals and big institutions since the beginning of trading, individuals however do not have access to such services, which is why they have to learn stock analysis, especially chart analysis, manually, in many cases losing money in the process.

The goal of this project is to offer a tool to individuals, which does *not* trade automatically, but instead performs analysis on financial markets and presenting these insights to the user, whilst also making predictions into the future based on past financial data.

The goal was not to hardcode a system which calculates certain indicators of past charts, or make hardcoded decisions about predicting future financial development, but to realize a system which learns from historical data instead, a system which autonomously develops rules and predicts future movements in price based on the rules it established.

The deep learning library used is Tensorflow, a widely adopted, open source project by Google, which integrates the Keras API to develop neural networks. Since financial data is a kind of sequential data, in later parts of the thesis recurrent neural networks are the chosen type of neural networks.

Zusammenfassung

Seit der Eröffnung der ersten Börse interessierten sich Menschen für die Analyse des Aktienmarkts. Erst als sich das Internet etablierte wurde das Handeln mit Aktien für jedermann möglich, und einher kam auch die Möglichkeit Aktien über *kürzere* Zeitspannen zu kaufen, das Kaufen von Aktien rein als Investment-Möglichkeit, und nicht mehr um die dahinterstehenden Gesellschaften zu finanzieren. Automatisierte Trading-Bots sowie Server, welche jede Sekunde Millionen von hartkodierten Entscheidungen treffen wurden für Institutionen verfügbar, individuelle Personen allerdings haben keinerlei Zugriff auf solche Möglichkeiten. Deshalb müssen Individualpersonen die Aktienanalyse per Hand erlernen und verlieren in dem Prozess meist viel Geld.

Das Ziel dieser Diplomarbeit liegt darin, den genannten Individualpersonen einen Service zu bieten, welcher *nicht* automatisiert Trades ausführt, sondern stattdessen Analysen im Aktienmarkt durchführt und dem Nutzer diese Einsichten in die Märkte zur Verfügung stellt.

Das Ziel lag darin ein System zu erstellen, welches aus vergangenen Marktentwicklungen lernt, und selbstständig lernt Entscheidungen zu treffen und Prognosen zu erstellen.

Tensorflow und Keras wurden als Deep Learning Bibliotheken verwendet. Tensorflow ist ein Open Source Projekt von Google, welches die Keras API einbindet. Da Preisentwicklungsdaten sequentielle Daten sind, wurden in späteren Teilen der Arbeit rekurrente neuronale Netze verwendet.

Acknowledgments

At this point we would like to thank those individuals, who assisted, aided and encouraged us in the creation of this thesis.

Firstly, we would like to express our sincere gratitude to our supervisor Peter Bauer for his continuous support and guidance throughout the implementation of the project and writing the thesis. At any time he provided insightful comments and was available for questions.

Besides our advisor we would like to thank Dr. Ulrich Bodenhofer for providing us insights into stock market analysis with deep learning.

Leon Schlömmer

I would like to thank my family, in particular my father DI(FH) Christian Schlömmer who always provided critical insights, suggestions and supported me through this time.

Contents

Chapter 1

Introduction

1.1 Initial Situation

In general, there are two types of participants in the financial markets: institutions and private investors (traders). Institutions employ full time analysts and have access to all available data as well as access to automated trading systems and server farms performing calculations and analysis with the available data. Private investors do not have access to such systems and to all the data. Private investors and traders perform their trades manually and do the analysis themselves.

APIs that offer historical data are hard to find and mostly do not provide enough data. There is one API that offers live data, however the number of requests is highly limited.

Some services which are available to individuals offer so called *social trading*. It allows individuals to copy the trades (execute the same trades) other people make. An example for an existing platform offering social trading is eToro (see [18]).

1.2 Problem

Being profitable with buying and selling financial instruments is not an easy task. Most people make losses when trading. Manually analyzing charts of multiple instruments and trying to see patterns is almost impossible for humans, many try themselves with try-and-error approaches.

Trading signal services are one-sided, they do not offer analysis at all, they just provide calls to action. There are next to no services that offer an analysis of charts, stating probabilities of certain predictions and that offer continuous predictions.

1.3 Definition of Task

The primary focus in this project is to develop a backend which continuously scans live market data and trains new machine learning models with the newly available data to

learn the evolution of patterns in charts. Before creating the backend, the primary task is researching which machine learning models work best, and what the machine learning models should learn from the available data, and what they should predict. In this research there are two planned approaches to examine. The first approach is to try to group similar looking segments of charts and to predict in which group a future chart will belong. In the corresponding user interface the user would be presented all the so called clusters with all the parts of charts inside them, and the user would be shown the probabilities of a future development belonging in one of the clusters.

Another approach to examine is predicting whether or not the price of an instrument will rise on the basis of recent charts.

Once the investigations of which machine learning models to use and which predictions to make is finished, a backend will be developed which provides live analysis. Clients for mobile and web-based platforms will then display the live analyzations to the user.

1.4 Goal

By giving individuals the opportunity to gain big scale insights into the markets, the goal is to decrease the losses made by unexperienced individuals, and to support their decision-making process before entering a trade or buying a stock.

1.5 Structure of the Thesis

This thesis is structured in three parts. In the first parts the fundamentals and theory behind the machine- and deep learning models, which are used in this thesis, are provided to the user. It covers the theory behind the shallow learning techniques: linear regression, logistic regression and K-means clustering. Furthermore this part covers the very basics of deep learning and the mathematics behind neural networks. In addition to that the reader is given a brief introduction into the world of stocks and financial markets.

The second part contains the core part of the thesis, which is a thorough explanation of the examined approaches of learning from financial data. The reader is presented with the theory, the implementation and the results of the different approaches.

The third and final parts concludes the results and reviews the course of events of this thesis.

Part I

Theoretical Foundation

Chapter 2

Introduction to Stocks and Trading

This chapter addresses basic domain specific insights on the stock market and stock trading. It will be explained just enough so that later chapters can be understood. For a deeper treatment of this topic we refer to [24], from which also the main part of this chapter has been taken. Any further resources that were used are mentioned in the respective sections.

2.1 Basic Terms

Financial Trading deals with the buying and selling of financial instruments with one core principle: to predict whether a financial instrument will go up in price, or down. These instruments can take many forms, but some of the main categories are:

Share or also called stock is a small unit of ownership in a company. By allowing investors to buy part of the company, the management is able to raise capital to put back into the business in order to invest. To buy shares on a stock exchange, the company must carry out an initial public offering, or IPO. To purchase stocks from a private company, it is necessary to contact them directly, but they are not obliged to sell them. The dividend is an amount of money paid to shareholders, representing a portion of the companies profit. When a company makes profit, the management decides how much to reinvest into the business and how much to distribute as dividends to the shareholders.

Stock Index refers to a measure of the value within a certain section of the stock market. This certain section can either be an exchange, a region or a sector. Stock indices give traders and investors an indication of how an exchange, region or sector is performing. Some major indices are:

- FTSE 100 - UK

- DAX - Germany
- CAC 40 - France
- Dow Jones - USA
- TecDAX - Germany

Commodities includes physical assets, raw materials and agricultural products. To be officially tradable, a commodity must be entirely interchangeable with another commodity of the same type, no matter where it was produced, mined or farmed.

Cryptocurrency is an exchange medium based on the internet that applies cryptographic techniques to perform financial transactions. They only exist on computers and there are no coins and currencies in circulation. Cryptocurrency uses block chain technology to achieve decentralization, transparency and immutability. One of the most important feature of a cryptocurrency is the fact that it is not under the control of a central authority. Cryptocurrencies are transmitted between individuals online using private and public keys for identification. Users do not interact with each other through banks, PayPal or Facebook. Instead they are directly dealing with each other. A further description will not be given at this point, because otherwise it would go beyond the scope of the thesis. A much more detailed explanation of this topic can be found here [10].

Forex also known as foreign exchange, FX or the currency market is the way individuals and businesses convert one currency to another. It is the largest financial market in the world. The amount of transactions that take place everyday are around 100 times more than the worlds biggest stock exchange. The forex market is also important for financial institutions, central banks, and governments. Forex prices are always quoted in currency pairs, because in forex trading one currency is bought while another is sold. The stronger the economy of a country, the stronger its currency will be compared to other currencies. Some major pairs are:

- EUR/USD - Euro/US dollar
- USD/JPY - US dollar/Japanese yen
- GBP/USD - Sterling/US dollar

Financial markets enable traders to exchange financial instruments either electronically or physically. Some of the major financial markets are:

- NASDAQ
- London Stock Exchange
- Chicago Mercantile Exchange
- Forex market

- NYSE

Markets tend to have very strict rules and regulations which help to reduce fraud and illegal activity. Trading with these assets is all about balancing the risk with the potential reward.

There are basically two different approaches in dealing with financial instruments. The first approach is essentially a long-term form of financial trading which involves buying and holding financial assets over a number of months or years. The other one is called trading, also known as speculation which tends to focus on the short-term market movements.

In order to trade on the market, there must be participants involved in trading and the most important ones are listed.

Retail traders are private individuals who buys and sells financial instruments using a personal account, not on behalf of an organization.

Brokers are mandatory to buy or sell in the stock market and other financial markets. They act on behalf of the client and are generally an authorized intermediary. A broker can either be a firm or an individual and might offer different services:

- **Full service** - actively managing investments and providing personal advice
- **Advisory management** - providing recommendations but leaving the final decision to the client
- **Execution-only dealing** – simply carrying out instructions to trade, on demand

Institutional traders are organizations that deal in the financial markets, generally on a much larger scale than retail traders. These organizations manage large pools of money so sometimes they make trades of a magnitude such that they influence market price movements, particularly in shares. Banks are an example of institutional traders.

High-frequency trading (HFT) uses computers to automatically apply trading strategies and algorithms, finding and exploiting patterns, trends and tiny fluctuations in the market. Instead of employing teams of analysts and traders, HFT companies are relying on technology that can make multiple trading decisions in a fraction of the time it takes for a human brain.

To establish a market, two sides are required:

Buyers who usually believe an assets value is likely to rise - known as bulls

Sellers who generally think an assets value is set to fall - known as bears

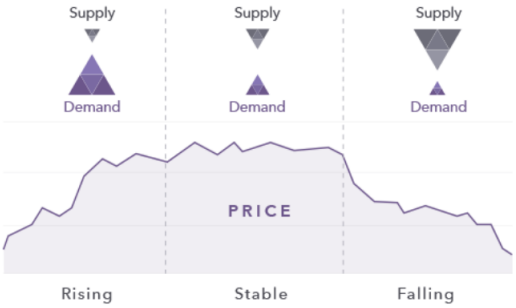


Figure 2.1: Supply - Demand

The relationship between them powers the movements in market prices. The balance of demand from buyers and supply from sellers influences prices in financial markets, as to be seen in figure 2.1. Basically, if more people want to buy a financial instrument rather than to sell it, the price will rise because the instrument is more sought-after (the demand outstrips the supply). Conversely, if supply is greater than demand then the price will fall. Supply and demand can be influenced by many factors, but the two most important, specifically for stocks, are:

Earnings are the profits a business makes. If the earnings are better than expected, the share price generally rises. If the earnings disappoint, the share price is likely to fall. Companies tend to release earnings announcements for a specific time period, usually a quarter, half or full year. The firms share price can be particularly volatile immediately before and after the announcement, especially if the figures are significantly better or worse than anticipated.

Sentiment is perhaps the most complex and important factor in a share price. Share prices tend to react strongly to expectations of the companies future performance. These expectations are built on any number of factors, such as upcoming industry legislation, public faith in the companies management team, or the general health of the economy.

An offer for a financial instrument contains not one but two prices and that are:

Bid price is called the price a seller receives.

Ask price is what the buyer has to pay, also called offer price.

Spread is the gap between the bid and ask prices of a financial instrument. The tighter the bid-ask spread, the quicker it is possible to profit when the market moves in that direction. In general, the higher the current volume of trades on an asset, the narrower the spread tends to be. The spread may also incorporate a brokers fee for handling the trade. The broker quotes their clients a price slightly lower

(a) Long (b) Short

Figure 2.2: Long and Short Trading

than the fundamental bid price or slightly higher than the ask price, keeping the difference to cover its costs.

Two expressions are used frequently in the context of trading, the long and short trading. When going long, the trade is opened by buying an asset whose value is expected to rise and close it by selling for a higher price. To go short, the opposite is done what means to sell when the price is expected to go down in the hope of buying the asset later back at a lower price. The two types can be seen in figure 2.2. Taking a short position is a common strategy to offset (or hedge) the risk of adverse price movements when holding a long position, which is called hedging. In short, the idea is that if the long position makes a loss, the short position compensates with a profit.

When deciding to open or close a trade, the broker has to be informed in order to sell or buy which is called placing an order. An order is simply an instruction to buy or sell an asset. The different order types are:

Market order is used to deal immediately at the best price available. An order that has been executed is called a filled order.

Limit order is an instruction to trade if a markets price reaches a particular level that is more favourable than the current price. As well as using a limit order to open a new trade, it can also be used to close an existing position – protecting the profit if the market might change direction.

Stop order also know as stop-loss, is an instruction to trade when the market hits a level less favourable than the current price. At the point where the loss would be unacceptable, the position will be closed.

For an order to be filled, there must be sufficient buyers or sellers in the market to take the other side of their trade. For assets that are traded on exchange, prices are sourced from an order book, which is the list of buyers and sellers currently placing orders with the price and size of each order.

Slippage occurs when markets are moving rapidly and the desired price is no longer available until the execution of the order, because the market prices can change in a matter of milliseconds.

Tick refers to a measure of the minimum upward or downward movement in the price of a financial instrument. It can also refer to the change in the price from one trade to the next trade. This definition has been taken from [39].

Volatility is a statistical measure of the amount an assets price changes during a given period of time. It has become a popular way of assessing how risky an asset is – the higher the level of volatility, the more risk is associated with the asset. Volatile markets are characterized by extremely fast-paced price changes and high trading volumes, which is seen as increasing the likelihood that the market will make major, unforeseen price movements. On the other hand, markets that exhibit lower volatility tend to remain stable, and have less-dramatic price fluctuations.

When buying an asset in the traditional way, generally the full purchase price has to be paid but there is another way to trade financial instruments.

Leverage means that only a fraction of the value of a position needs to be raised. The amount that needs to be raised is called margin or deposit requirement. Effectively, the provider lends the remaining amount to the client. Nevertheless any profit, which is still based on the full value of the position, appears magnified in comparison to the outlay. The flip side of this is that any losses are magnified in the same way. With leverage, both the profit and any loss can actually exceed the initial outlay. Leveraged trading can be regarded as riskier than traditional trading and the most important aspect is to be aware of the potential loss.

2.2 Analysis

Market analysis is an essential part of any trading strategy. It involves using the information that is available now, to make educated predictions as to how a market will behave in the future. There are two main types of market analysis which are explained in the following two sections.

2.2.1 Technical Analysis

Technical analysis focuses exclusively on the price movements of a market and it is based on the premise that what happens in the past can be used to predict what might happen in the future. Of course this can never be guaranteed, which means technical analysis should not be used in isolation.

In the technical analysis the traders study the historical price movements of markets in order to examine the trends and patterns. Technical analysts can interpret the behaviour of buyers and sellers to give an indication of where the market could move next.

(a) Candlestick (b) Candlestick chart

Figure 2.3: Candlestick Chart

Since there are certain types of behavioural patterns that have occurred repeatedly in the past, they can be identified as they emerge and predict the most possible future movement of the market. The best way to analyse the market is to use charts. The three main types of chart are:

Line charts just display the closing price of an asset over time. They are essentially just a number of data points joined by a line.

Bar charts or also called HLOC show the high, low, open and close prices for each period.

Candlesticks are generally the most popular chart type among traders, as they are able to convey a large amount of data quickly. For candlestick charts, the body of the candle signifies the range between the open and close, the wick is the high/low range, and the colour displays whether the price has gone up or down, as it can be seen in figure 2.3. A green (or white) body shows the price has increased while a red (or black) body shows it has dropped. Long candle bodies and short wicks signify there has been strong buying or selling pressure in one direction. Short bodies and long wicks indicate there was significant buying or selling pressure in one direction, but that pressure then reversed.

Support is the ceiling that the price struggles to break through.

Resistance is the floor where the price tends to stop declining.

Support and resistance are the building blocks of technical analysis and many effective trading strategies can be based solely around them. The reason why these levels appear is the balance between buyers and sellers, or demand and supply. In figure 2.4 the first line is resistance and the second one is the support.

Areas of support and resistance are relatively straightforward to plot on a chart, the only thing that needs to be done is to find a level that the market has reached but not

Figure 2.4: Support - Resistance

broken through. The more times the market reaches that level, but fails to go through it, the stronger that level of support or resistance is said to be. It is unusual for a market to hit exactly the same price time after time before reversing, so it is probably more useful to think of them as support or resistance zones. When a market touches or jumps through support or resistance briefly before reversing, it is said to be testing the level.

Breakout refers to a situation in which a price pushes through a support or resistance level aggressively.

Fakeout implies that the price returns shortly after it passes through the support or resistance.

Market prices do not generally rise or fall in straight lines over a period of time, but rather in a series of zigzags. The price will rise to a *peak* or a high, then drop to a *trough* or low. Despite this, the market will usually move in one overall direction or *trend*, and it is the relative positioning of the peaks and troughs that define this trend. There are three types of market trends:

Uptrend occurs in the market when each successive peak is higher than the last, and each trough is also higher than the preceding one.

Downtrend refers to a series of successively lower peaks and lower troughs.

Sideway trend has no clear pattern to the peaks and troughs, with the price generally oscillating in a fairly narrow range between support and resistance levels.

These trends are visualized in figure 2.5. Trend lines are only confirmed if the price touches them at least three times. Because if the price touches the trend line only twice, it only tells whether it was a rise or a drop and not necessarily a trend.

Chart patterns tend to repeat themselves, so they can be used to predict where a market might move next. When learning to recognize these patterns and when they start to form, then it is often possible to gather clues about what the price is likely to do next. Note that these patterns are not reliable in any case and should just be used

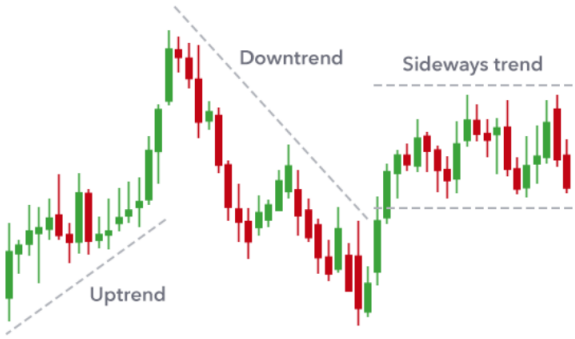


Figure 2.5: Trends

Figure 2.6: Simple moving average

as guides as to the future direction of a market. These patterns are not explained in detail, only a few well-known patterns are mentioned.

- Double tops and bottoms
- Triple tops and bottoms
- (Reverse) Head and shoulders

To get automatically detectable factors from these charts a whole area of different measures, so called technical indicators have been developed.

Moving average is one of the most popular indicator. The simple moving average (MA or SMA) calculates the average price of a market over a specified period of time. In figure 2.6 three MA's can be observed. One with a period of 10 days, the second with 20 and the last with 50 days. Fast moving averages stick close to the market price, while slow moving averages are smoother and lag further behind.

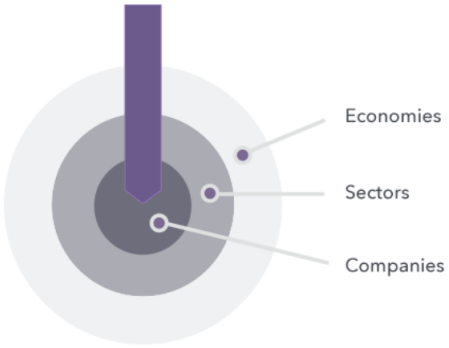


Figure 2.7: Top down approach

In an uptrend, faster moving averages tend to stay above the slower ones. In a downtrend the reverse is true. Moving averages can also be used as dynamic support and resistance levels. Additionally there exists the exponential moving average (EMA) that is is similar to MA but gives more weight to recent periods.

2.2.2 Fundamental Analysis

While both technical and fundamental analysis involve using information that is available now in order to predict how a market might perform in future, they do this in different ways. When conducting technical analysis, the focus is exclusively on price data and movements, seeking trends and patterns that indicate the likely future direction of a market.

In fundamental analysis, on the other hand, a more holistic view is taken and all the circumstances surrounding the market are considered. This could include assessing all kinds of factors, from a whole country/region employment rates or manufacturing output right down to an individual companies cash flow and expenditure. Fundamental analysis is often said to be more rigorous and comprehensive than technical analysis because everything that can affect the assets value is studied and taken into consideration, not just price data and chart patterns. Fundamental analysts generally use a top-down approach, as shown in figure 2.7. The steps are the following:

- **Consider the overall economic performance -** If an economy is in expansion, it is reasonable to suppose that the industries and companies within the country/region will benefit and see growth in turn. Conversely, if an economy is declining then even successful businesses might suffer.
- **Assess which sectors will be affected -** When an economy as a whole is expanding, certain industries may benefit more than others. There can also be times when economic expansion has a universally beneficial effect that is felt by most sectors and companies.

- **Select a sector to focus on -** Having identified some industry sectors of interest, the next step is to consider levels of supply and demand for the products or services in each, and the other factors affecting their future prospects.

- **Look at companies within the sector -** The last step is to take a closer look at it is constituent companies and decide which ones to explore in more depth.

When performing fundamental analysis on an individual company, it is advisable to take into account a number of qualitative factors. First of all it is important to understand the business model of the company, what means how they earn money.

The next step is to consider the competitive landscape and compare the company to others that might threaten its position. Important is maybe that the company offers something that its competitors do not or the product is slightly more suitable in terms of a few characteristics.

The last step is to ask whether the company has the capacity to evolve successfully in a changing environment to stay ahead of the competition. Also the quality of the management team is a key factor affecting a companies prospects so the the strengths, weaknesses and experience of each executive is considered. For example, it could be examined whether managers increase or reduce their shares in the company. If a series of senior executives have left in quick succession this could point to weaknesses in the companies culture or troubled relationships in the management team.

The final step is to carry out a detailed review of the companies finances. Key areas are listed below.

Balance sheet is a statement of the companies assets, liabilities and capital at the end of a particular reporting period. The best way to interpret a balance sheet is to compare it with previous releases, looking to see which way the figures have been changing over time. It is also a good idea to check them alongside the balance sheets of other similar companies within the industry. From a balance sheet it is possible to derive a number of useful ratios, such as the popular debt/equity ratio.

Cash flow is a record of the cash generated and used by a company. Unlike the balance sheet, which is a snapshot of a particular point in time, the cash flow statement covers a certain period specified by the company. In principle, the cash flow statement indicates whether a company consistently generates more money than it consumes. If cash from operations exceeds net income, the company is operating efficiently so it will potentially be able to increase its dividend payments, pay off existing debts or repurchase shares.

Income statement measures a companies financial performance over a set timeframe that illustrates the future profitability of the business, comparing the value of sales made with the cost of making them. It is based on the equation: income = revenue – expenses

Fundamental analysis has several advantages:

- **Identify long-term trends -** Whereas technical analysis generally only identifies short-term patterns and opportunities, a fundamental approach will highlight companies worthy of longer-term investment.
- **Develop a sense of business -** By carrying out comprehensive analyses of a number of companies, participants build up a thorough understanding of the way these businesses work, getting to know their industry sector and what drives their revenue and profit.

It also has certain drawbacks:

- **It could take a long time to pay off -** Unlike technical analysis, where the moment to buy or sell can often be predicted on a chart, there is no way of knowing how long it might take to secure a profit.
- **No two sectors are the same -** The way how the value of a company is assessed depends on the industry to which it belongs. This means that every time a new company in an unfamiliar sector is considered, it is likely that the approach will have to be completely different.
- **Companies spin their own news -** Most of the information fundamental analysts use to assess a company comes from the company it self and they try to reflect as positively on their performance as possible.

Chapter 3

The Data

Generally, it is very expensive to gain access to financial data, such as stock market chart data, cryptocurrency chart data, forex chart data, etc. Whilst there are numerous APIs which offer historic data, only a few of them offer some of that data very limited for free. Finding usable data is most critical part in the course of this project, because the data is at the very heart of the project. If no data is available the whole project fails. This chapter goes over the available options that were used in this thesis.

3.1 Alpha Vantage

Alpha Vantage is a service that purely focuses on providing a REST API for realtime as well as historic data on stocks, foreign exchange and cryptocurrencies. They also provide technical indicators.

3.1.1 Advantages

All of the data is available for free. The API offers daily prices, weekly prices and monthly prices, all including the corresponding volumes. It offers prices for stocks, forex and cryptocurrencies. Moreover it offers a clear documentation for all its features.

3.1.2 Disadvantages

Intraday prices which are in a 1-minute interval are only available 1 day into the past. This is too little data in this high resolution for a machine learning model to learn meaningful patterns from. Furthermore the free request rate is limited to 5 requests per minute. Another disadvantage is, that it does not include any stock indices like the DAX or the Dow Jones.

3.1.3 What was used

The data used from this API are historic stock prices in a one-day resolution. All the price points used sum to around 10000.

3.2 IG Labs

IG is actually mainly a broker for stocks, contracts for differences (CFDs) and forex. However, they also offer a REST API that allows their users to execute trades and view historic such as live data on all the instruments they offer.

3.2.1 Advantages

They offer historic data in a one-minute resolution up to 15 years into the past, on all their instrument. Moreover these instruments include stock indices like the DAX and the Dow Jones. The API also provides live data on all instruments.

3.2.2 Disadvantages

To be able to access the API, one must be a registered customer of their brokerage. Even then, the allowed weekly requests are highly limited. Although the website of IG Labs states that there is a possibility to increase the number of requests possible, they declined our request. In addition to that, when accessing the historic data only one price point can be accessed with one request, making gathering a lot of historic data from their API a tedious task.

3.2.3 What was used

The data used from this API are historic stock index prices of the DAX in 5 minute and 15 minute resolutions. All the price points used sum to around 10000 as well.

3.3 Cryptocurrency Data

After doing some research, a sight was found which offers a dataset for cryptocurrency data (see [21]). This dataset contains data in one-minute resolution for the instruments Bitcoin, Litecoin and Ethereum. The price data is from the time of June 2018 to August 2018. For each instrument there are around 100000 individual price points included.

Chapter 4

Introduction to Machine Learning

In the following chapter the topic Machine Learning is explained in a fundamental way. The rough structure of the chapter is taken from [20]. Additionally used resources are listed in the respective sections.

4.1 Machine Learning Definition

> *"Machine Learning is the science of getting computers to learn and act like humans do, and improve their learning over time in autonomous fashion, by feeding them data and information in the form of observations and real-world interactions."* Daniel Fagella in [16]

If someone searches the internet for the ultimate definition of Machine Learning, one gets overwhelmed because everyone explains it differently. The above quote describes it fairly accurately.

Basically, and in this point all sources agree, it is a system which is able to learn from examples by self-improvement and without being explicitly coded by programmer. Machine Learning combines data together with statistical tools to predict an outcome which is then used to produce usable insights. In simple terms, the machine is receiving data as input and applies an algorithm to produce resulting answers.

4.2 Artificial Intelligence vs. Machine Learning vs. Deep Learning

Artificial Intelligence (AI), Machine Learning (ML) and Deep Learning (DL) are three terms that are often used interchangeably for describing programs which act in an "intelligent manner". However, it is useful to understand the main differences between them in order to be able to distinguish between these concepts. Following the definitions given in [35].

Artificial Intelligence refers to a broader concept than Machine Learning which involves the use of computers to mimic human cognitive functions. Whenever computers

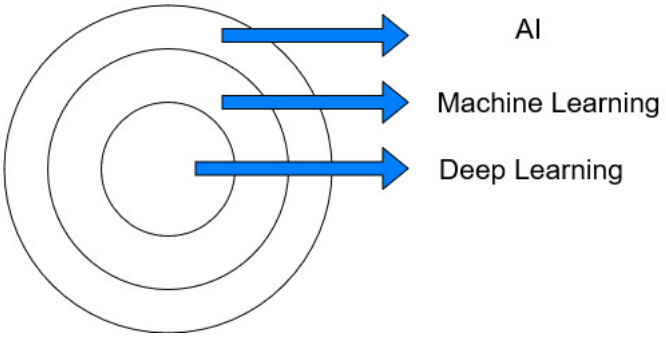


Figure 4.1: AI vs. Machine Learning vs. Deep Learning

height	weight	age
1.75	81	25
1.61	49	14
1.93	91	45
1.82	73	19
1.79	65	22
1.61	45	47
1.45	30	12
1.98	89	67

(a) Table

height
1.75
1.61
1.93
1.82
1.79
1.61
1.45
1.98

(b) Instance

1.75	81	25

(c) Feature

Figure 4.2: Table, Instance and Feature

perform tasks that are based on algorithms in an "intelligent" way, that is AI. Machine Learning is a subgroup of AI and is focused on the capability of machines being able to receive a set of data and learn on their own, as described in section 4.1.

Teaching computers on how to think like humans is accomplished by the use of Deep Learning and this is just a phrase that refers to complex Neural Networks. Neural networks are a number of algorithms that are modelled in a simplified way based upon the human brain model. We will talk about Neural Networks more accurately in section 7.

Figure 4.1 is intended to illustrate the distinction between the three concepts. There it can be observed that Machine Learning is a subset of Artificial Intelligence and Deep Learning is a subset of Machine Learning.

4.3 Terminology

In order to understand the following sections and chapters, some expressions should be known. The information has been taken from [36] and [25].

We start with the assumption that we have given data from a specific domain in a

tabular form. An example of such a table is given in figure 4.2 where an extract of the data from a medical examination is given, consisting of height, body weight and age of each patient.

First of all, the individual parts of the table are defined.

Definition 1. A single row of data is called an *Instance*, as shown in figure 4.2, and it is an observation from the domain.

Definition 2. A collection of multiple instances is referred to as a *Dataset*. The table in figure 4.2 is called a dataset in its entirety.

Definition 3. A single column of data in a dataset is called a *Feature*. It is a component or also known as an attribute of an instance. An example can be observed in figure 4.2.

Definition 4. The aggregation of n features into an n-tuple $(F_1, F_2, \ldots, F_n)$ is called a *Feature Vector*.

Now we come to the definitions of the central terms for Machine Learning.

Definition 5. A Machine Learning *Model* is simply said a representation of a mathematical function that is used to learn correlations between the given data in order to perform some specific tasks such as prediction.

Definition 6. The *Error* is defined as the deviation between the value predicted by the model and the actual value.

Definition 7. A dataset that is fed into the Machine Learning model in order to train it, is called *Training Dataset*.

Definition 8. *Model Parameters* are internal variables of the model being adjusted to make the model predict outputs to given inputs as good as possible.

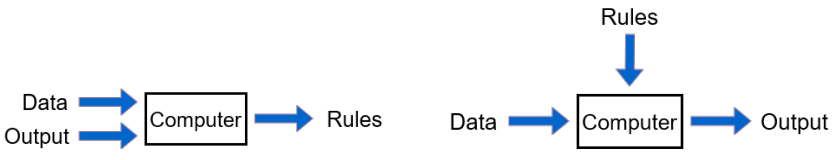
Remark 1. An example of such parameters are the weights in a neural network. This is explained in more detail in chapter 7.

Definition 9. *Model Hyperparameters* are adjustable parameters that must be tuned and set manually in order to obtain a model with optimal performance.

Remark 2. Here an example would be the number of clusters in the k-means algorithm as described in section 6.3.1.

Definition 10. Given a training dataset according to definition 2 and a Machine Learning model as given in definition 5 with n parameters $P_1, P_2, \ldots, P_n$, the adaption of the model parameters $P_1, P_2, \ldots, P_n$, as defined in definition 8 to minimize the overall Error E is called the *Training Procedure*.

Remark 3. There are basically three different types of training procedures for a Machine Learning model which are unsupervised, supervised and reinforcement learning. This is treated in detail in section 4.6.



(a) Machine Learning (b) Traditional Programming

Figure 4.3: Machine Learning vs. Traditional Programming

Definition 11. To validate the accuracy of the model a *Testing Dataset* or also called *Validation Dataset* is used that is not applied in the training process.

Definition 12. What the Machine Learning model needs to predict is referred as the *Label* or *Target*.

Definition 13. *Noisy Data* is data with a large amount of additional meaningless information in it called noise. This includes corrupt data and any data that the system cannot understand and interpret correctly. [6]

4.4 Machine Learning vs. Traditional Programming

Traditional programming is significantly different compared to Machine Learning. With traditional programming, a programmer codes all of the rules, usually in collaboration with an established expert in the domain for which the software is being developed. Every rule is based on a logical foundation. Following these logical statements, the machine produces some output. More rules are needed to be defined as the system increases in complexity. Whereas this is a perfect approach for many situations like website programming it becomes very difficult to completely impossible for some domains like image recognition or product recommendations.

The aim of Machine Learning is to overcome this problem. The Machine Learning model will learn for example the correlation between input and output data. In response to new input and experience, the model adapts its parameters to improve performance and accuracy over time. This way the programmers no longer have to write new rules each time new data is received.

So the main difference is that in traditional programming one must formulate/code all the rules by hand, whereas in Machine Learning the algorithm will automatically formulate the rules from data. Figure 4.3 represents these two fundamentally different approaches.

4.5 The Machine Learning Procedure

Here is a brief overview of the necessary steps towards a complete Machine Learning model as given in [41] and [37].

- **Define the Problem**: The first and one of the most critical tasks is to find out what the main goal is supposed to be. There are several other aspects to consider, such as whether Machine Learning is really necessary or if the normal approach would also lead to the desired results.

- **Data collection**: The first step in the Machine Learning model development cycle is to collect relevant and comprehensive data. This amount depends on many factors, like the complexity of the problem and learning algorithm. This step is very important because the quantity and quality of the data determines how accurate the model is. There are various techniques to collect the data but they are beyond the scope of this thesis. Often pre-collected datasets are used. The outcome of this step is generally a representation of the data in form of a table.

- **Data preparation**: Before starting to train a model, it is important to transform the data in such a way that it can be fed into a Machine Learning model. This preparation includes, but is not limited to the following steps:

  - **Dealing with missing data**: It happens quite often that some values of the data samples are not recorded. This may be caused by errors in the data collection, blanks in surveys, incorrect measurements, etc. The problem is that most algorithms are not capable of handling such missing values which means that they must be considered before data is fed into the models. For example the samples or features with missing values can be eliminated or replaced with the mean value of the other samples.
  - **Feature scaling**: This is an essential step in the pre-processing stage, since most Machine Learning algorithms perform much better when dealing with features that are on the same scale. A common technique for example is normalization, where the features are rescaled to a range between 0 and 1.
  - **Splitting data**: The dataset has to be split into a training set and a validation set as defined in the definitions 7 and 11. The majority of the data is going into the training set.

- **Choose a model**: The next step is to select a model. There are a variety of models that have been developed over the years. Depending on the task, the most suitable model should be used. Such models are explained in more detail in the chapters 6, 7 and 8.

- **Train the model**: In short, the training process involves the initialization of some random values for the model parameters. Then it predicts the output of the input data by using the initial random values. At first the error will be high, but by comparing the prediction of the model with the correct output, the model is able to adjust the parameters until a good prediction model is obtained.

- **Evaluate the model**: Once training is complete, the model is evaluated against the validation set. This should be representative of how the model may perform in the real world.
- **Parameter tuning**: Hyperparameters as given in definition 9 may be tuned, to increase the accuracy of the model. Adjusting or tuning these hyperparameters remains a little bit of an art and is rather an experimental process that depends strongly on the specifics of the dataset, model and training process. An example is the number of times the training dataset is passed through during the training. This refers to the fact that the complete dataset can be fed to the model not only once, but several times. Often this results in higher accuracies.
- **Make predictions**: After all these steps the model can finally be used to make predictions with new data.

4.6 Types of Machine Learning Systems

In the following section the different Machine Learning approaches will be examined in more detail. Details covering this can be taken from [23].

4.6.1 Supervised Learning

Given is a model M with $P_1, P_2, \ldots, P_n$ model parameters. The training set T consists of a feature vector $X(x_1, x_2, \ldots, x_n)$ and a feature with the desired output $Y(y_1, y_2, \ldots, y_n)$ for each instance of X. The error function E is dependent on the desired output y and the prediction M made for the corresponding instance x from T. The aim of M is to predict the label for each instance of T and then compare it with the desired output of Y in order to adjust its model parameters $P_1, P_2, \ldots, P_n$ in such a way that E is minimal.

Once the supervised learning algorithm is completely trained, it will be able to observe a new, unknown example and predict a proper label for it.

The most important supervised learning algorithms are:

- Linear Regression (see section 6.1)
- Logistic Regression (see section 6.2)
- Artificial neural network (see chapter 7)
- Support Vector Machines (SVMs)
- Decision tree
- k-nearest neighbours
- Naive Bayes classifier

4.6.2 Unsupervised Learning

In unsupervised learning the instances are not labelled with the desired output. Unsupervised learning works by analysing the data without its labels for the hidden structures within it, and through determining the correlations, and for features that actually correlate two data items. It can then be trained to group, cluster and/or organize the data in a manner that allows a human (or other intelligent algorithm) to access and make sense of the newly formed data. The vast majority of the data is not labelled therefore unsupervised algorithms are highly important.

Some of the most important unsupervised learning algorithms are:

- K-Means-Clustering (see section 6.3.1)
- Autoencoders
- Principal Component Analysis (PCA)
- Isolation Forest

4.6.3 Reinforcement Learning

Reinforcement learning is quite different from supervised and unsupervised learning. Whereas the relationship between supervised and unsupervised (the presence or absence of labels) is easy to identify, the relationship is somewhat more obscure for reinforcement learning.

In simple terms, Reinforcement Learning can be seen as learning from mistakes. It is about taking suitable action to maximize reward in a particular situation in order to find the best possible behaviour or path the machine should take in a specific situation. Place a reinforcement learning algorithm into any environment and the algorithm will make a lot of mistakes in the first place. As long as the algorithm is being given a kind of signal that associates good behaviours with a positive signal and bad behaviours with a negative signal, it is possible to reinforce the algorithm so that it prefers good behaviours rather than bad ones.

Over time, the learning algorithm is learning to make fewer mistakes than before. Reinforcement learning is very behaviour-oriented. In order to understand reinforcement learning better, a example is now used. Lets look at how to teach an agent how to play Mario. For each reinforcement learning problem it is necessary to have an agent and an environment and a way to connect the two via a feedback loop. In order to connect the agent with the environment, it provides the agent with a number of actions that it can take and that have an impact on the environment. To connect the environment to the agent, let the environment constantly send two signals to the agent: an updated state and a reward (our reinforcing signal for the behaviour). In the Mario game, the agent is the learning algorithm and the environment is the game (most likely a certain level). The agent has a number of actions and these are the button states. The updated state will be each game frame as time progresses, and the reward signal will be the change in the score.

4.7 Applications of Machine Learning

A few areas of applications for Machine Learning are listed in this section. A deeper treatment of this topic can be found here [15].

- **Virtual Personal Assistants**: Siri or Alexa are just a few of the most popular examples of virtual personal assistants. As the name implies, they help to gather information when asked by voice. They only need to be activated and asked for something. In order to answer them, for example, the personal assistant searches the internet for the desired information. It is also possible to instruct the assistant to perform certain tasks, like setting an alarm.

- **Online Customer Support**: Nowadays, a variety of websites offer the possibility to chat with customer support representative while navigating through the website. However, not every website has a live agent to answer all of the questions. In most cases, a chatbot is used to communicate. These bots usually extract information from the website and present it to customers. With each customer interaction, the chatbot improves over time. They will tend to better interpret users requests and provide them more accurate answers which is possible through Machine Learning models.

- **Product Recommendations**: Often when a product is purchased online, similar products are suggested to the customer afterwards. This is happening because based on the behaviour with the site/application, previous purchases, items liked or placed in the shopping cart, brand preferences, etc., the product recommendations are created using Machine Learning.

- **Social Media Services**: From the personalization of news feeds to improved targeting of advertising, social media platforms use Machine Learning for their own and users benefits. For example Facebook constantly keeps track of friends with whom a user connects, the profiles that this user visits very often, the interests, the workplace, etc. On the basis of this information, a list of Facebook users with whom a user can establish a friendship is suggested.

- **Search Engine Result Refining**: Search engines like Google and others use Machine Learning to improve the search results. Every time a search is performed, the algorithms in the backend monitors how the results are responded to. When the top results are opened and a long time is spent on this web pages, the search engine assumes that the results displayed matched the query. If the second or third page of search results is reached but none of the results are opened, the search engine assumes that the results displayed did not match the query. In this way, the algorithms operating in the backend will improve the search results.

4.8 Main Challenges of Machine Learning

Within this section, the major problems that usually occur during Machine Learning are outlined. The following resources were used [30] and [14].

4.8.1 Understand the Limits of Machine Learning Technology

In general, people tend to overestimate the present capabilities in relation to Machine Learning. Due to the hype and media fuss, people began to expect that Machine Learning algorithms will quickly solve all complex problems.

4.8.2 Black Box Problem

In the early stages of Machine Learning, simple, shallow methods were commonly used. Nowadays, Neural Networks are used quite often. These models can get very complex and as a result, Machine Learning engineers no longer understand them fully. This phenomenon is known as a black box. This is a major drawback when developing applications in the area of medicine or driverless cars. Because then it is difficult to justify, for example, if the diagnosis of an algorithm is wrong or the autopilot has caused a fatal accident.

4.8.3 It Takes Time to Achieve Results

The traditional software development is quite straightforward. Having defined goals and functionalities, choosing the technology to build them, and assuming that it will take a few months to publish a working version. Machine Learning has more layers of development, as described in section 4.5 and in most cases it takes more time. While traditional website or application development can be done by an experienced team with a good estimate of time, a Machine Learning project is much more difficult to estimate in relation to its duration. One reason for this is that it can get quite difficult to find errors within the Machine Learning model, because of the Black Box problem, described here 4.8.2.

4.8.4 Data

The biggest problem in the area of Machine Learning is about the data. Some problems that deal with data are listed here.

Data is Not Free

In order to train a Machine Learning model, data is essential. Nowadays storage may be cheap but it takes time to gather a sufficient amount of data. Furthermore, the purchase of ready-to-use data sets is quite expensive.

Trustable Data

Many data scientists are often using data from an external source. But usually there is no quality control or quality assurance procedure for how this data was obtained. Therefore, the question often arises whether to trust these data.

Preparing Data

The preparation of data for training is a complex and complicated process, however one of the most essential one. It also demands a large amount of time. The required steps are listed in section 4.5.

Insufficient Quantity of Training Data

Most Machine Learning algorithms typically require huge amounts of data before they are able to deliver useful results. As the architecture expands, more data is needed to achieve useful results.

Nonrepresentative Training Data

To be able to properly generalize, it is essential that the training data is fully representative for the new scenario which need to be generalized onto.

Poor Quality Data

If the training data is filled with errors, outliers and noise, it is much more harder for the system to recognize underlying patterns, making it less likely to work properly. Therefore, the process of cleaning the data is extremely important, and the vast majority of data scientists spend a large part of their time doing just that.

4.9 Tensorflow and Keras

Whilst it is possible to develop every kind of neural network from scratch using pure Python and Numpy, this was not a feasible option for this thesis. Deep learning libraries such as Tensorflow have many benefits. Perhaps one of the biggest advantages of deep learning libraries is that they enable accelerated training. This means, that the graphical processing unit (GPU) is used to do perform calculations. GPUs are usually multiple times faster than CPUs for training neural networks, because they are designed to perform matrix operations efficiently.

Moreover, deep learning libraries handle mathematical operations such as calculating derivatives of cost functions with respect to weights, updating the weights, calculate loss, etc. In addition to that, Tensorflow integrates Keras, which accelerates the development of neural networks with a simple API to create complex models.

This section will briefly cover Tensorflow as well as Keras, and will show how to implement the models used in this thesis.

4.9.1 Tensorflow

At its core, Tensorflow is a open source deep learning library maintained by Google. It does automatic differentiation, creates calculation graphs in the background and provides hardware accelerated training. Moreover it provides a wide ecosystem to deploy the deep learning models in production system, with data preparation pipelines, client libraries for many different systems, etc.

4.9.2 Keras

Keras is a high-level neural networks API. This means that it provides an API to combine common neural network layers with little effort. Since version 2 it is tightly integrated in Tensorflow. It can also be used with other deep learning libraries like PyTorch, CNTK, Theano, or PlaidML. In this thesis, the models that was used is the *sequential* neural network model, which contained either a densely connected layer or a recurrent layer such as the LSTM or GRU layers.

Chapter 5

Introduction to the Python Environment

This chapter aims to explain some basics about the Python environment.

5.1 Python 2 vs. Python 3

In the past, there was a bit of a debate in the coding community about which Python version was the best one to learn: Python 2 vs Python 3. Nowadays Python 3 is the clear winner for new learners or those wanting to update their skills because the changes have actually made it easier to understand Python. Therefore a beginner should always learn the latest version. That said, there are still some situations where picking up Python 2 might be advantageous. Some companies are still using Python 2 for legacy reasons or maybe a project depends on certain third-party libraries that can not be ported to Python 3. But more and more companies are using Python 3, or beginning to make the switch from version 2 to 3. For a deeper treatment of this comparison we refer to [29].

5.1.1 History

Python 2.0 was first released in 2000. Its latest version, 2.7, was released in 2010. Python 3.0 was released in 2008 and the newest version, 3.7, was released in the year 2018. Although Python 2.7 is still widely used, the adoption of Python 3 is growing very quickly. In 2016, 71,9% of projects used Python 2.7, but by 2019, it had fallen to 13% according to [26]. Python 2 is dwindling rapidly as 9 out of 10 developers claim to be using Python 3 in 2019. Last year, a quarter were still using Python 2. The reason why Python 2 is declining is that the version is not actively developed anymore which means it does not get new features and its maintenance has already stopped in the beginning of 2020.

5.1.2 Main Differences

The following list shows just a few among the many differences that exist between the two versions.

Python 2 is Legacy

Since Python 2 has been the most popular version for over a decade and a half, it is still entrenched in the software at certain companies. However, since more companies are moving from Python 2 to 3, someone who wants to learn Python should avoid spending time on a version that is becoming obsolete.

Incompatible Libraries

Many of todays developers create libraries exclusively for Python 3. Similarly, many older libraries built for Python 2 are not forward-compatible.

Unicode Support

In Python 3, text strings are Unicode by default. In Python 2, strings are stored as ASCII by default. Unicode is more versatile than ASCII because strings can store foreign language letters, Roman letters and numerals, symbols, emojis, etc., offering more choices.

5.2 Installing Python Packages

In this section, the basics for installing Python packages are explained. For a more in-depth coverage of this topic, we refer [7] and [8], where the main part of the chapter is also taken from.

A package is simply a collection of several Python files with pre-written code, such as objects or functions that can be added to an application. This allows developers to accomplish a certain functionality rather than having to program it from scratch each time. Because Python is a popular open source development project, it is supported by an active community of contributors. Python developers offer their software to other users under open source licensing terms. As a result, Python users can benefit from the solutions that others have already developed. Developers can also contribute their own solution for all kinds of tasks to the collaborative pool of packages.

In order to install, upgrade or remove such packages, the program *pip* is required. Since Python 3.4 it is part of the Python binary installers by default. It is a command line program and the pip command is added to the system which can then be executed from the command prompt as follows.

```
100    $ pip <pip arguments>
```

The most frequent use of pip is the installation of packages from the Python Package Index, called PyPI, which is a public repository of all contributed open source licensed Python packages. With the following command, the latest version of a package and its associated dependencies are installed from the Python Package Index.

```
100  $ pip install package
```

Often it is not always desirable to install the latest version of a package, due to incompatibility problems with other packages for example. Therefore it is possible to specify e.g. the exact or minimum version for a package directly on the command line using a *requirement specifier*. In general, a requirement specifier consists of a package name that is followed by an optional version specifier. To install the specific version 2.3 the following command needs to be executed.

```
100  $ pip install "package==2.3"
```

Further reading for all specifier can be done here [32]. Once a suitable package has been installed, a new attempt to install it will usually have no further effect. An upgrade for existing packages must be explicitly requested. It is also possible to specify all packages with their respective versions in a *requirements.txt* file in order to install them with a single pip command. This way, time is saved and the required pip command does not need to be called for each individual package.

5.3 Virtual Environment

Often Python applications use packages that are not part of the standard library. Many applications require different versions of a specific library. It is important to note that by default every application on the system uses the same directories for storing and retrieving third-party packages. This may not seem like a big deal at first glance and it really is not for system packages (packages that are part of the standard Python library) but for third-party packages it matters. Therefore, a single Python environment where all third-party packages are stored, can not always meet the requirements for each application.

Consider the following scenario of having two applications: application A and application B, both dependent on the same package. The issue becomes obvious when both applications need different versions of the required package. For example, application A might need version 1.0 while application B requires the newer version 2.0. The packages are only saved by name, so there is no differentiation between the versions.

To overcome this problem, the solution is to create *virtual environments*. The primary goal of virtual Python environments is to provide an isolated environment for each Python application. That means every application can have its own dependencies, regardless of what dependencies any other environment has. For solving the earlier example of conflicting demands, application A may have installed its own virtual environment with version 1.0, while application B has a different virtual environment with version 2.0. When application B requires a library to be upgraded to version 3.0, this

does not have an impact on the environment of application A. There are no limits to the numbers of environments that can be created. When developing Python applications, it is always strongly advised to use a virtual environment.

In order to create virtual environments, the command line tools *venv* (for Python 3) and *virtualenv* (for Python 2) are used. The difference between these two tools is that venv is a package included in the Python Standard Library, starting with Python 3.3 and virtualenv needs to be installed separately with pip, but supports Python 2.7+ and Python 3.3+. Virtualenv provides more functionality compared to venv, by supporting Python 2.7+ for example as already mentioned and many others. Furthermore virtualenv continues to be more popular among Python developers.

The creation of such environments with these two tools is fairly simple, but will not be explained in this thesis. Worth mentioning is the pip *freeze* command, which will generate a list of all installed packages within an environment. This list can be dumped into a requirements.txt file and can then be shipped as part of an application to reconstruct the "frozen" environment on a different system. More information about this topic can be found here [1] and [9].

5.4 Conda

Conda is an open source package and environment management system. Both conda and pip are often regarded as being almost identical. Even though the functionality of these two tools partly overlap, they still differ from each other slightly. The information for this section is taken from [27].

Pip is the tool recommended by the Python Packaging Authority for installing packages from the Python Package Index, PyPI. Conda on the other hand, is a cross-platform package manager for installing and managing conda packages from either the Anaconda repository or the Anaconda cloud. Furthermore, conda packages are not restricted to Python software. They can also include C or C++ libraries, R packages, or any other software. This illustrates an essential difference between conda and pip. Pip installs Python packages, while conda installs packages that can contain software written in any language.

Another important difference between the two tools is the fact that conda has the additional ability to create isolated environments that can contain different versions of Python and/or the packages installed in them. While Pip has no built-in support for environments, it relies on other tools such as virtualenv or venv to create isolated environments, but both try to solve the same problem. Tools such as pipenv for example wrap pip and virtualenv together to offer a unified method for working with environments and installing packages.

Another difference between pip and conda is how dependency relationships are fulfilled inside an environment. If packages are installed with pip, no attention is paid to ensure that the dependencies of all packages are fulfilled simultaneously. This can result in environments that are subtly disrupted when packages installed earlier in the order have incompatible dependency versions relative to packages installed later in the

order. On the contrary, conda checks that all requirements of all packages installed in an environment are fulfilled. This verification can take additional time, but helps to avoid creating broken environments.

With regard to the similarities between conda and pip, not surprisingly, some are trying to combine these tools. A main reason for the combination of pip and conda is when one or several packages can only be installed via pip. There are over 1,500 packages available in the Anaconda repository, including the most popular data science, Machine Learning and AI libraries. Despite this large collection of packages, it remains still small in comparison to the over 150,000 packages available on PyPI. Sometimes a package is required that is not available as a conda package, but is available on PyPI and can be installed with pip. In such cases, it is useful to consider using both conda and pip.

5.5 Python for Machine Learning

This section points out the main reasons why Python is such a popular choice for Machine Learning as given in [13].

5.5.1 Library Ecosystem

One of the major reasons or rather the major reason why Python has become one of the most popular programming languages for Machine Learning is because of its wide range of libraries. Additionally the variety of useful libraries and tools are completely free. The most popular and frequently used libraries for Machine Learning are:

- *Numpy* is the fundamental package for scientific computing with Python.
- For handling basic Machine Learning algorithms such as linear or logistic regression, clustering and so on, *Scikit-learn* is used.
- *Keras* and *Tensorflow* for deep learning.
- *Matplotlib* for creating charts, histograms and other different types of visualization.
- For high-level data structures and analysis, *Pandas* is used.

5.5.2 Low Entry Barrier

Work in the Machine Learning industry involves dealing with a lot of data that needs to be processed in the most efficient and convenient manner. The low entry barrier allows more data scientists to quickly get their hands on Python and use it for Machine Learning development instead of spending too much time and effort on learning the language. Even if they have never programmed before, Python is best suited as a first programming language since it is quite easy to understand because the syntax is very simple and easy to read, compared to others.

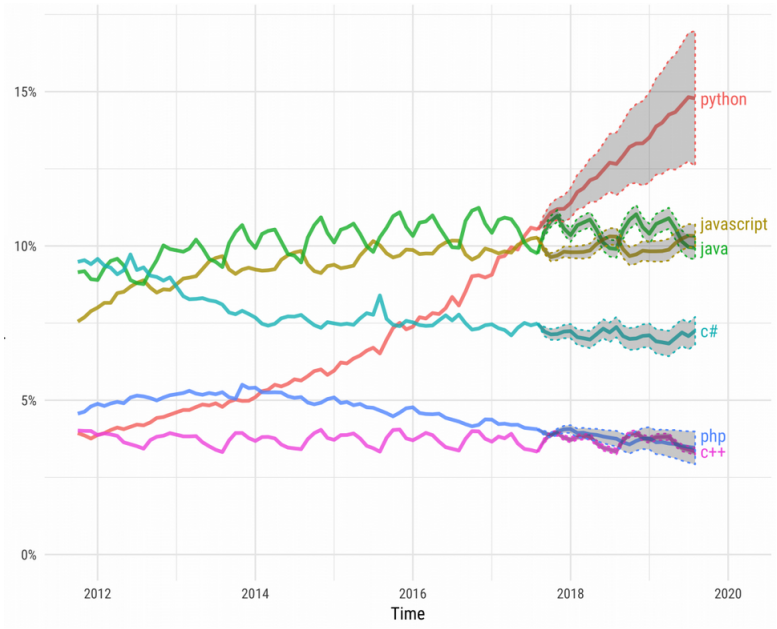


Figure 5.1: Projection of future traffic for major programming languages [17]

5.5.3 Community Support

Having a strong community support built around a programming language like Python is always very helpful, because that way there is a lot of documentation available as well as Python communities and forums. Furthermore, Python is becoming more and more popular amongst data scientists. The popularity of Python is expected to increase until 2020 at least, according to StackOverflow [17], which can be observed in figure 5.1.

5.5.4 Alternatives

Of course other programming languages can also be used for Machine Learning purposes and a few of the most important ones are listed here. More detailed information about this topic can be found here [22].

Java

The second most common language commonly used by data scientists and Machine Learning developers is Java. 15% of experts use Java for network security/cyber attacks and fraud detection where Python is not really preferred for such tasks.

R

R is a graphic-based language that is used for statistical calculations, analysis and visualization in Machine Learning. It is the perfect platform for those wanting to use graphics to explore statistical data. It is very popular, especially amongst statisticians and it is mainly used for statistical purposes.

Scala

Especially when dealing with an extremely large amount of data, also called Big Data, the programming language Scala is used often.

5.6 Interpreter

A common question is, if Python is compiled or interpreted. The answer is both. In simple terms compiling means to convert a program from a high-level language like C or Java into a binary executable file which is full of machine code (CPU instructions). The reason for such a step is that a machine can not understand these high-level programming languages because by default computers work only with machine language which is binary format. Compiling does not always mean to convert a high-level language into machine language. It is also possible to take a program in one language and convert it into another language. Usually the source form is a higher-level language than the destination form, for example when converting from JavaScript 8 to JavaScript 5.

When a Python file gets executed, Python automatically compiles it into byte code behind the scenes. Note that the Python byte code is not binary machine code. Bytecode is an intermediate code generated from the compiling process which can be executed by a virtual machine. The file where the byte code is stored has a .pyc extension. The next time the same program is executed, Python loads the .pyc file and will skip the compilation step if it has not been changed. It automatically checks the timestamps of source and bytecode files in order to know when it should recompile. If the source code is saved again, the bytecode is automatically recreated the next time the program is executed.

After that the byte code gets transferred to a specific virtual machine for Python called PVM (Python virtual machine) for execution. The compiled byte code will run on that virtual machine which converts it into native code. That simply means it does not matter which machine or rather which CPU architecture is used. The PVM itself is not a separated program. It is not necessary to install the PVM on its own. In fact, the PVM just iterates through our bytecode statement, one by one, to perform its operations. The PVM is the runtime engine for Python. The PVM is available as part of the Python system at any time. Actually, this is the component that executes the scripts. Technically, it is just the final step of what is called the Python interpreter.

The reason for this more or less complicated approach is to achieve portability for platform independence. Meaning that the code is written once and can be run on various platforms. Consider the following example, when a code is compiled to a binary

file on one specific machine, it can not be executed on another machine because the CPU architecture might differ. With the help of the PVM it can be executed on every machine.

In summary it can be said, when a Python program is executed, Python reads the .py file into memory and compiles it to get bytecode. It first checks to see whether there is a precompiled bytecode version, in a .pyc, that has a timestamp which corresponds to its .py file. If not, it needs to be compiled. After the program is compiled into bytecode, it is delivered to the Python Virtual Machine (PVM) for execution. Details covering this can be taken from [34] and [40].

Chapter 6

Shallow Learning and its Statistical Foundation

This chapter will cover prerequisites for Deep Learning, including Shallow Learning techniques and their statistical foundations. The resources for the information presented in the sections 6.1 and 6.2 are [5] and [3].

6.1 Linear Regression

In contrast to most textbooks about statistical methods and Machine Learning basics, linear regression is explained in a very detailed form on the following pages, even very small mathematical steps are given explicitly, to allow a mathematically not so well trained reader to follow along.

Linear Regression is the most basic form of Machine Learning. Like all Machine Learning approaches, it tries to predict an output y given an input x. Linear Regression does this by trying to find the line, plane or hyperplane of best fit, depending on the input X.

X is a matrix of the dimensions $N \times D$.

- N is the number of samples - the number of points in the training sample
- D is the number of features, meaning the number of inputs needed for an output y to be calculated
- For every sample in N samples exist D inputs

Linear regression produces a *line* of best fit for one-dimensional data, a *plane* of best fit for 2-dimensional data and a *hyperplane* of best fit for multidimensional data. A multidimensional example of X could look like:

$$\begin{pmatrix} 1 & 2 & 1 \\ 3 & 1 & 2 \\ 2 & 3 & 3 \\ 1 & 1 & 1 \end{pmatrix}$$

In this example $N = 4$ and $D = 3$. Therefore it is 3-dimensional, and has 4 samples. Every row represents one sample.

If $D = 1$ the matrix only has one column, and can also be referred to as a column vector. An example for a column vector can look like:

$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

A visual example of one-dimensional data is given in figure 6.1. Every point in the plot represents a sample consisting of $\{(x, y)\}$

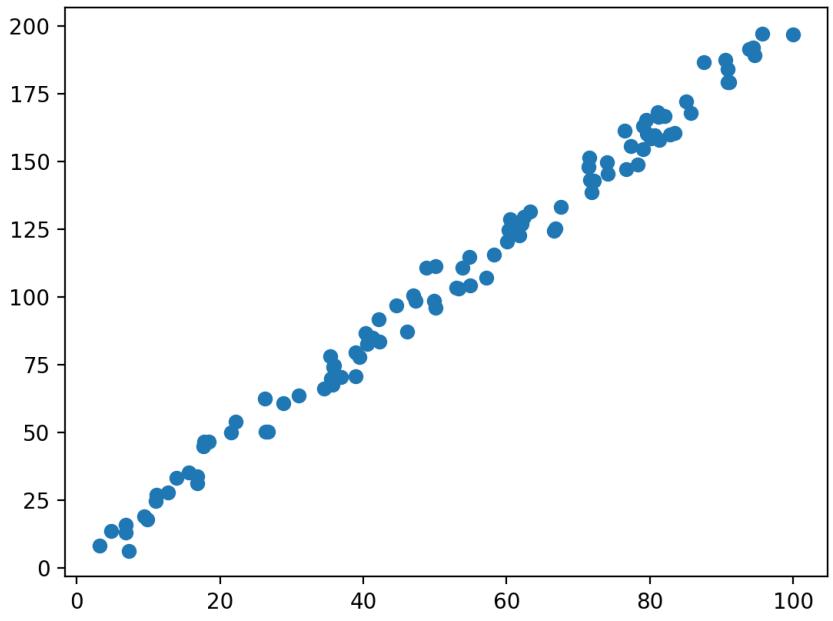


Figure 6.1: Example one-dimensional data. Note, that this plot also includes the corresponding targets, which is why the plot is two-dimensional. The x-axis is the input, it is one-dimensional.

From now on, the line, plane or hyperplane of best fit will be referred to as the function of best fit, or prediction function. The prediction function is denoted as:

$$\hat{y} : \mathbb{R} \to \mathbb{R}$$

As the name suggests, the prediction function is used to predict / calculate an output given an input, that the model has never seen before.

6.1.1 The Error Function

The quality of the prediction function is measured by comparing actual target values of the training set to the predicted values of the prediction function for input values of the training dataset. For finding the function of best fit, another function defining the quality of the prediction function is needed. This function will be called the *error* function E and is defined as:

$$E = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{6.1}$$

where N again is the number of points, y_i is the target (the output from the training dataset) given an x and $\hat{y}_i$ is the predicted output of the linear regression function given the same x.

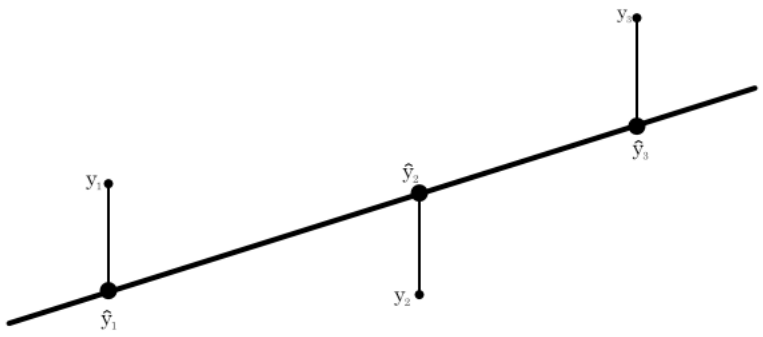


Figure 6.2: The error function measures the distance between target values and predicted values

The output of the error function measures how far away predicted values are from actual values from the training data set, given the same input. Another variant of the error function could be:

$$E = \sum_{i=1}^{N} (y_i - \hat{y}_i)$$

The problem in this case however is obvious. If $\hat{y}_i$ is greater than y_i, the error would actually become smaller. There is no unity in *penalization*. This could be fixed by using the absolute difference, like:

$$E = \sum_{i=1}^{N} |y_i - \hat{y}_i|$$

This version too, however, brings inconveniences. As section 6.1.2 shows, the derivative of the error function has to be calculated - which generally is possible, however the derivation is not continuous which makes calculating the global minimum by gradient

descent 6.1.6 impossible. Therefore, the function 6.1 is optimal for linear regression and regression in general. Squaring the differences has an additional benefit, as the penalization grows exponentially with the difference from prediction to target, and for a difference smaller than 1 the penalty becomes even smaller.

For every dataset $\{X, Y\}$, E is actually a function of $\hat{y}(X)$.

$$E(\hat{y}) : f \to \mathbb{R}$$

This of course implies, that for every possible prediction function exists an error value for the given training dataset. The error function can actually be plotted for one-dimensional linear regression, as the prediction function only has 2 tunable parameters a and b (see 6.1.2). An example for a plot of the error function is given in figure 6.3. The horizontal axes are the parameters a and b where as the vertical axis is the error. This will be further explained in 6.1.2.

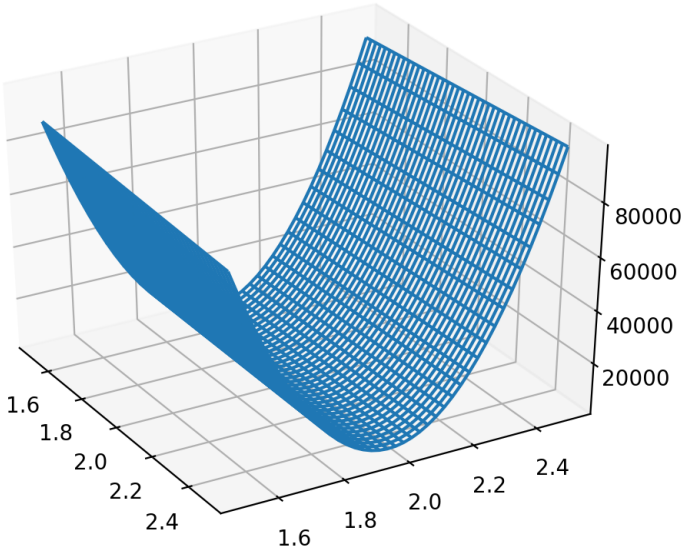


Figure 6.3: Example plot of an error function. The vertical axis is the error.

In order to find the best possible prediction, the parameters where the output of the error function is minimal have to be found. Therefore, the error function is the function that will be minimized, this means that the parameters of E will be tuned, so that E is as small as possible. The first technique of minimizing E will be discussed in the following section.

6.1.2 Linear Regression for One-Dimensional Data

This section will solve (minimize E) the most basic form of linear regression - linear regression for one-dimensional data. This means, that our function will generate a one-dimensional output (only 1 value) $\hat{y}$ for any given input x in X, where x is also one-dimensional. Now the output will be a basic linear function:

$$\hat{Y}(X) = a \cdot X + b$$

To minimize E, the above function will be substituted for $\hat{y}$ in E. Now E takes following form:

$$E(a, b) = \sum_{i=1}^{N} (y_i - (a \cdot x_i + b))^2$$

In order to find the global minimum of any function, the derivative of the function has to be set equal to 0 and solved for its parameter. E has two parameters a and b, making matters a bit more complicated. When a and b are now calculated to minimize the error E for a given dataset X the term

$$\sum_{i=1}^{N}$$

will be used frequently. To simplify the notation, a simpler term

$$\sum x_i = \sum_{i=1}^{N} x_i$$

is introduced. First, the partial derivatives have to be calculated, using the chain rule:

$$\frac{\partial E}{\partial a} = \sum 2 \cdot (y_i - (a \cdot x_i + b)) \cdot (-x_i) \tag{6.2}$$

$$\frac{\partial E}{\partial b} = \sum (-2) \cdot (y_i - (a \cdot x_i + b)) \tag{6.3}$$

To find the parameters a and b which minimize E, both derivatives have to be set to 0. In the last step, the now evolved system of equations has to be solved for a and b.

$$\frac{\partial E}{\partial a} = \sum 2 \cdot (y_i - (a \cdot x_i + b)) \cdot (-x_i) = 0$$

$$\sum (-y_i \cdot x_i) + a \cdot \sum (x_i^2) + b \cdot \sum x_i = 0$$

$$a \cdot \sum (x_i^2) + b \cdot \sum (x_i) = \sum (y_i \cdot x_i) \tag{6.4}$$

Notice how the sums are being split up, this will turn out to be helpful later.

$$\frac{\partial E}{\partial b} = \sum(-2) \cdot (y_i - (a \cdot x_i + b)) = 0$$
$$\sum(y_i) - a \cdot \sum x_i - b \cdot \sum 1 = 0$$
$$\sum y_i = a \cdot \sum x_i + b \cdot N \tag{6.5}$$

Now the system of equations becomes clearer. One way to solve it, is by solving one of the two equations for a and by substituting what is a into the other equation.

$$\sum(y_i) = a \cdot \sum(x_i) + b \cdot N$$
$$a = \frac{b \cdot N - \sum(y_i)}{-\sum(x_i)} = \frac{\sum(y_i) - b \cdot N}{\sum(x_i)}$$

This term will now be substituted in the transformed equation 6.3:

$$\sum(x_i \cdot y_i) = \frac{\sum(y_i) - b \cdot N}{\sum(x_i)} \cdot \sum(x_i^2) + b \cdot \sum(x_i) =$$
$$\left(\frac{\sum y_i}{\sum x_i} - \frac{b \cdot N}{\sum x_i}\right) \cdot \sum(x_i^2) + b \cdot \sum(x_i) =$$
$$\frac{\sum(y_i) \cdot \sum(x_i^2)}{\sum(x_i)} + b \cdot \left(\sum(x_i) - \frac{N \cdot \sum(x_i^2)}{\sum(x_i)}\right)$$

From this point it is easier to solve this equation for b

$$\sum(x_i \cdot y_i) = \frac{\sum(y_i) \cdot \sum(x_i^2)}{\sum(x_i)} + b \cdot \left(\sum(x_i) - \frac{N \cdot \sum(x_i^2)}{\sum(x_i)}\right)$$
$$b = \frac{\sum(x_i \cdot y_i) - \frac{\sum(y_i) \cdot \sum(x_i^2)}{\sum(x_i)}}{\sum(x_i) - \frac{N \cdot \sum(x_i^2)}{\sum(x_i)}}$$

After simplifying this equation finally b becomes clear:

$$b = \frac{\sum(x_i \cdot y_i) \cdot \sum(x_i) - \sum(y_i) \cdot \sum(x_i^2)}{(\sum(x_i))^2 - N \cdot \sum(x_i^2)} \tag{6.6}$$

a can actually be solved using elimination of the equations 6.4 and 6.5 :

$$a \cdot \sum(x_i^2) + b \cdot \sum(x_i) = \sum(x_i \cdot y_i)$$
$$-a \cdot \sum(x_i) - b \cdot N = -\sum(y_i)$$

Adding these equations yields:

$$a \cdot \left(N \cdot \sum(x_i^2) - \left(\sum(x_i) \right)^2 \right) = N \cdot \sum(x_i \cdot y_i) - \sum(y_i) \cdot \sum(x_i)$$

And finally a too becomes clear:

$$a = \frac{N \cdot \sum(x_i \cdot y_i) - \sum(y_i) \cdot \sum(x_i)}{N \cdot \sum(x_i^2) - \left(\sum(x_i) \right)^2} \tag{6.7}$$

This concludes the mathematical part of solving one-dimensional linear regression. Whilst it is in its simplest mathematical form, this however can still be simplified for implementation with Numpy.

Implementation

Numpy does matrix and array operations with high efficiency and is therefore quicker than, for example, using for-loops. Furthermore Numpy offers a number of matrix operations that the built-in python library does not include, like calculating the mean of an array. This can be applied to calculate 6.4 and 6.5. Unfortunately the mean is nowhere to be found in neither a nor b. This is not a problem however, because by expanding the equations the mean of X, Y, X^2 and $X \cdot Y$ can be found. The mean of a given sample X is defined as:

$$\bar{x} = \frac{1}{N} \cdot \sum x_i$$

Transforming all the summations over X, Y, X^2 and $X \cdot Y$ can be done as follows.

$$a = \frac{N \cdot \sum(x_i \cdot y_i) - \sum(y_i) \cdot \sum(x_i)}{N \cdot \sum(x_i^2) - \sum(x_i) \cdot \sum(x_i)} \cdot \frac{\frac{1}{N^2}}{\frac{1}{N^2}} =$$

$$\frac{\frac{\sum(x_i \cdot y_i)}{N} - \frac{\sum(y_i)}{N} \cdot \frac{\sum(x_i)}{N}}{\frac{\sum(x_i^2)}{N} - \frac{\sum(x_i)}{N} \cdot \frac{\sum(x_i)}{N}} = \frac{\bar{xy} - \bar{x}\bar{y}}{\bar{x^2} - \bar{x}^2} \tag{6.8}$$

This same logic can also be used to simplify b.

$$b = \frac{\frac{\sum(x_i^2)}{N} \cdot \frac{\sum(y_i)}{N} - \frac{\sum(x_i \cdot y_i)}{N} \cdot \frac{\sum(x_i)}{N}}{\frac{N \cdot \sum(x_i^2)}{N^2} - \frac{\left(\sum(x_i)\right)^2}{N^2}} = \frac{\bar{x^2}\bar{y} - \bar{xy} \cdot \bar{x}}{\bar{x^2} - \bar{x}^2} \tag{6.9}$$

Given, that a the dataset consisting of X and Y is already loaded, implementing linear regression only consists of a few lines of code.

```
100 denominator = (X * X).mean() - X.mean()**2
101 a = ((X*Y).mean() - X.mean() * Y.mean()) / denominator
102 b = ((X*X).mean() * Y.mean() - (X * Y).mean() * X.mean()) / denominator
103 Yhat = a*X + b
```

Executing this for the samples in figure 6.1 yields $a = 1.972$ and $b = 2.864$. A plot of this function over the sample data can be seen in figure 6.4.

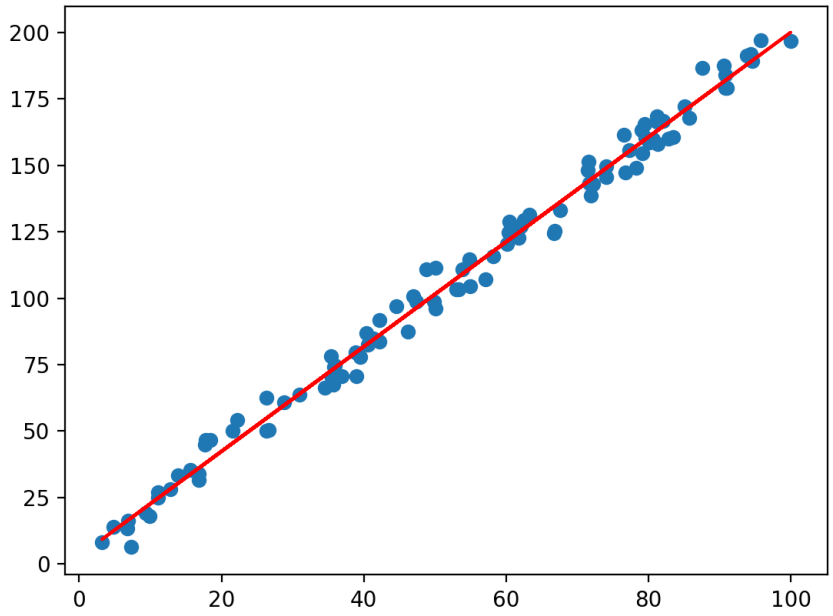


Figure 6.4: Red line: prediction function, blue dots: training data

6.1.3 Measuring the Model

One method to measure how good a model performs, in the above case the model of minimizing squared error, is by comparing it to simply predicting the average of all the targets in the training data. A possible formula for this is R^2:

$$R^2 = r^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y}_i)^2}$$

If R^2 is close to 1, that means that the model is performing good, because the squared error is close to zero, meaning that the entire fraction becomes close to zero. If R^2 is close to 0, that means that the model is performing only as good as simply predicting the mean of Y. In the rare case of R^2 being smaller than 0, that means that the model is performing worse than predicting the average of Y.

6.1.4 Multiple Dimension Linear Regression

This section covers the most general case of linear regression, multiple dimension linear regression. x can now contain more than 1 value, it can be multidimensional and has D dimensions. Now the formula of for the prediction function is as follows:

$$\hat{y} = w^T x + b$$

Both w and x are column vectors, they need to have the same dimensionality, therefore w also has D dimensions. However, one cannot take the dot product of the two vectors when they have the same dimensions[1]. For that reason w is transposed, so that it has as many columns as x has rows.

b can actually be absorbed into X by adding a column of 1's to X:

$$x_0 \dots 1$$
$$w_0 \dots b$$

The full multiplication written out now looks like the following:

$$\hat{y} = w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 + \dots$$

Working with matrices

X is a matrix with N rows and D dimensions, it is of size $N \times D$. The prediction function 6.10 predicts an output for only one set of inputs, one sample in X. One sample of X is one row of X and is of shape (has the dimensions) 1xD. By definition it is a *feature vector* and not a column vector. In linear algebra however *column vectors* are the convention. Transposing the feature vector yields a column vector of size Dx1. w inherently has one row and D dimensions.

Predicting the output for one sample of X is done the following way:

$$\hat{y}_i = w^T x_i \tag{6.10}$$

whilst predicting N samples works like:

$$y_{N\text{x}1} = X_{N\text{x}D} w_{D\text{x}1} \tag{6.11}$$

In equation 6.11 the subscripts indicate the dimensions of the matrices.

Minimizing Squared Error

Finding the hyperplane of best fit by minimizing the squared error also holds good for multiple dimension linear regression. Substituting the prediction function 6.10 into the squared error function 6.1 yields following equation:

$$E = \sum (y_i - w^T x_i) \tag{6.12}$$

Just like in one-dimensional linear regression, finding the global minimum of $E(\hat{y})$ is done by deriving it, setting it to 0 and solving for its parameter. In the case of multiple dimension linear regression E has to be minimized for every component of w. In the

[1] The dot product of two column vectors is the same as the matrix product.

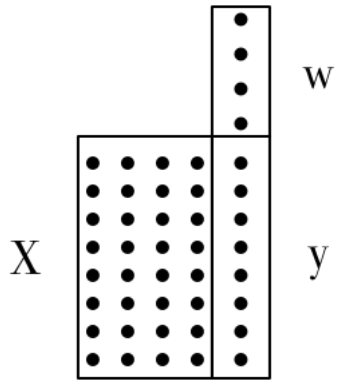


Figure 6.5: A visual example of predicting N y's

following equation all the components of w are indexed by $j \dots D$. w has as many components as X has features.

$$\begin{aligned}\frac{\partial E}{\partial w_j} &= \sum_{i=1}^{N} 2 \cdot (y_i - w^T x_i) \cdot (\frac{dw^T x_i}{dw_j}) \cdot (-1) \\ &= \sum_{i=1}^{N} 2 \cdot (y_i - w^T x_i) \cdot (-x_{ij})\end{aligned}$$

This equation now has to be set equal to 0 and solved for w:

$$\begin{aligned}\sum_{i=1}^{N} (-2) \cdot (y_i - w^T x_i) \cdot x_{ij} &= 0 \\ \sum_{i=1}^{N} y_i \cdot x_{ij} - w^T x_i \cdot x_{ij} &= 0 \\ \sum_{i=1}^{N} w^T x_i \cdot x_{ij} &= \sum_{i=1}^{N} y_i \cdot x_{ij} \\ w^T \cdot \sum_{i=1}^{N} x_i \cdot x_{ij} &= \sum_{i=1}^{N} y_i \cdot x_{ij} \qquad (6.13)\end{aligned}$$

Since w is not indexed by i it can be removed from the summation. Furthermore, this equation actually represents D different equations.

$$
\begin{aligned}
w^T \cdot \sum_{i=1}^{N} x_i \cdot x_{i1} &= \sum_{i=1}^{N} y_i \cdot x_{i1} \\
w^T \cdot \sum_{i=1}^{N} x_i \cdot x_{i2} &= \sum_{i=1}^{N} y_i \cdot x_{i2} \\
&\dots \\
w^T \cdot \sum_{i=1}^{N} x_i \cdot x_{ij} &= \sum_{i=1}^{N} y_i \cdot x_{ij}
\end{aligned}
$$

It helps to think of all the summations as matrices and vectors again. In more general terms, the summation over the products of two different vectors equates to the multiplication of the vectors.

$$a^T b = \sum a_i \cdot b_i$$

Now the above terms can be simplified.

$$
\begin{aligned}
\sum_{i=1}^{N} x_i \cdot x_{ij} &= X^T X \\
\sum_{i=1}^{N} y_i \cdot x_{ij} &= y^T X
\end{aligned}
$$

Finally, equation 6.13 can be solved for w using matrices.

$$
\begin{aligned}
w^T (X^T X) &= X^T y \\
(X^T X) w &= X^T y \\
w &= (X^T X)^{-1} X^T y
\end{aligned}
\tag{6.14}
$$

Implementation

Implementing this with numpy is as simple as a few lines of code.

```
100 w = numpy.linalg.solve(X.transpose().dot(X),
101     X.transpose().dot(Y))
```

6.1.5 A Word on Notation

Targets

The y's from the dataset were previously referred to as targets and have been denoted as y_i for a specific target of the dataset or Y for the whole dataset. They can however also be referred to as t_i and T respectively. This is especially then the case, when the output of the prediction function is denoted as y_i instead of $\hat{y}_i$.

The Objective Function

Previously the sum of squared errors was used as the error function. The error function is sometimes also referred to as the *cost function*.

The error function, or cost function, in more general terms is an *objective function*. In the case of the sum of squared errors, the *objective* was to minimize the sum of squared errors. As will be discussed in later chapters, there are also objective functions, like the log-likelihood, where the objective is to *maximize* the objective function.

Generally, the objective function can be denoted as J, the cost function is mostly denoted as C.

6.1.6 Final Notes on Linear Regression

As this and the next chapter build the fundament for deep learning, some more generally useful concepts will be discussed in this subsection.

Minimizing Error

In theory a sum of squared errors of 0 could be achieved by making a Taylor-function that fits the training data perfectly. The goal of linear regression however is not to find the function that best fits some data, but instead to best predict outputs that the model has never seen before. Therefore, it is actually the goal to leave in some so called *generalization error*. This is especially important in processes where models are trained iteratively, which are discussed later. With every iteration the error sinks on the dataset that a model is trained on. Achieving a good amount of generalization error is done by testing the model on a *validation dataset*. There are multiple techniques for splitting data into training data and validation data. The challenge lies in choosing the

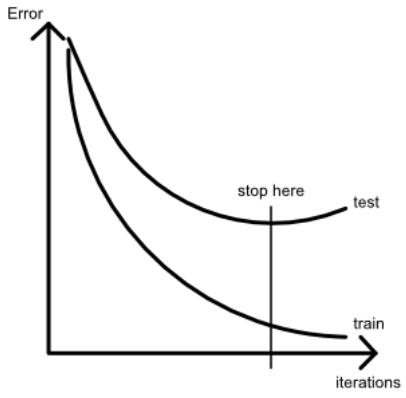


Figure 6.6: The error on training and validation datasets per iteration

right amount of iterations. In figure 6.6 a possible plot is shown. The right amount of iterations is, when the error on the validation dataset is at its minimum.

Using categorical Inputs

Some inputs do not have a numerical value, but instead represent a category. An example could be colors, where the inputs can range from red, to green, to blue. There are two main techniques for dealing with such inputs, *K-1 encoding* and *one-hot encoding*. They are fairly simple and similar techniques. By using one-hot encoding every value in a category in X is represented by either a 0 or a 1, depending on which value the category of the sample has. In the previously introduced color example a representation of the colors could look like:

- Red: $[1, 0, 0]$
- Blue: $[0, 1, 0]$
- Green: $[0, 0, 1]$

K-1 encoding is similar to one-hot encoding, the difference is that one value is absorbed into the bias term. This means, that one value is represented by all 0s.

- Red: $[1, 0]$
- Blue: $[0, 1]$
- Green: $[0, 0]$

These variables are called *dummy variables*. By including dummy variable in a regression model however, one should be careful of the Dummy Variable Trap. The Dummy Variable trap is a scenario in which the independent variables are multicollinear - a scenario in which two or more variables are highly correlated; in simple terms one variable can be predicted from the others. In the above model, the sum of all category dummy variable for each row is equal to the intercept value of that row - in other words there is perfect multi-collinearity (one value can be predicted from the other values) (See [19]).

The implication of this is, that the equation 6.14 can not be solved anymore, because $X^T X$ cannot be inverted. A solution to this, which is widely used in machine and deep learning, is called *gradient descent*, and will be discussed in the next chapter.

Gradient Descent

Gradient descent is a procedure, in which a function can be minimized without solving the equation of setting the first derivative to zero. Gradient descent is one of the most versatile techniques in Machine Learning, because its principles can be applied to many different problems. Gradient descent can be used to solve linear regression with multicollinear variables as well as advanced recurrent neural networks.

In the example of the Dummy Variable Trap, an equation is impossible to solve. In other cases the cost function could be minimized without using gradient descent, but instead by solving equations. The complexity of these equations however rises with the complexity of the models and with the number of parameters. In addition to that, it is

not always desirable to minimize the error on train-datasets, as this might lead to worse predictions on data that the models have not seen before (see 6.1.6).

The basic procedure of gradient descent is updating the parameters, the weights, of the cost function iteratively in the direction of the slope.

Definition 14. Given a weight w and a gradient $\frac{dJ}{dw}$ as well as a *learning rate* η, the weights are updated like shown in equation 6.15.

$$w = w - \eta \cdot \frac{dJ}{dw} \tag{6.15}$$

Remark 4. In equation 6.15 a new parameter η was introduced. η stands for the learning rate, and it is a *hyperparameter*. As apparent from the equation, η determines how much the parameters are updated in an iteration.

Finding η is very difficult however, as there is no scientific way of determining how big it should be. One can only find the best working size of η empirically. Most commonly it ranges from 0.1 to 10^{-7}. If η is too big, the weights will bounce back and forth between their optimal values, and if η is too small, the training process becomes slow, as more iterations are necessary to approach the optimum.

6.2 Logistic Regression

6.2.1 Basics

Despite its name, logistic regression does actually not solve regression problems, but solves classification problems instead. This means, that it tries to predict which class a given set of inputs belongs to. Logistic regression plays an important role in machine

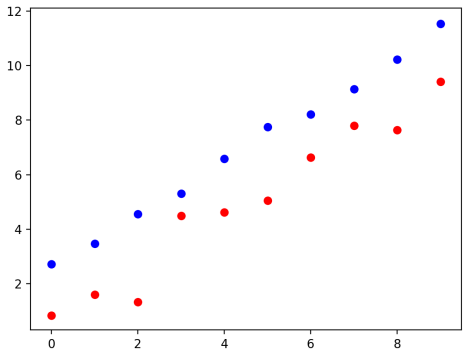


Figure 6.7: Example data, blue dots represent one class and red dots another

and deep learning, because it is the most basic building block of a neural network - a logistic regression unit is a *neuron*. This is explained in detail in chapter 7.

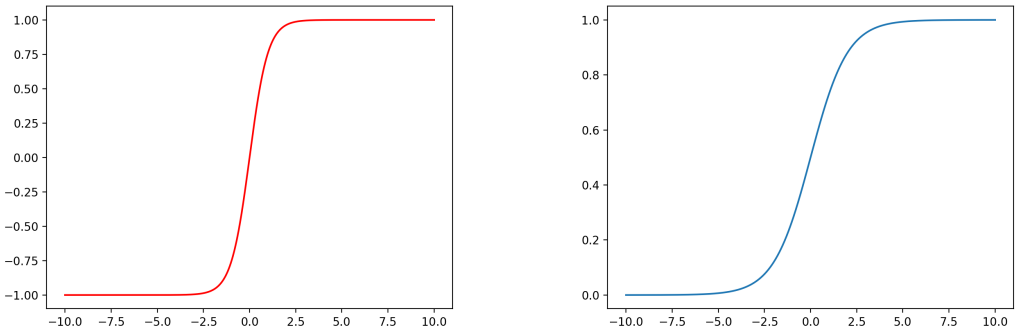
The output of logistic regression discloses how likely it is, that some input belongs to some class. Logistic regression is fairly similar to linear regression, in the sense that it wraps the output of linear regression in another function. From now on, this output will be denoted as:

$$a = w^T X$$

Logistic regression transforms a so that it can be interpreted as a probability. This is done by "squashing" it into some range, which can then be mapped to probabilities.

In logistic regression *sigmoid* functions are used for that purpose. Sigmoid functions approach their minimum and maximum asymptotically. Examples for sigmoid functions are:

- tanh (see 6.8a)
- $\sigma(a) = \frac{1}{1+e^{-a}}$ (see 6.8b)



(a) Plot of the hyperbolic tangens

(b) Plot of the sigma function

Figure 6.8: Plots of different activation functions

With the tanh function the output ranges from -1 to 1 and intercepts the y-axis at 0. The σ function ranges from 0 to 1 and intercepts the y-axis at 0.5. A benefit of σ is, that the output can directly be interpreted as a probability.

Definition 15. With logistic regression binary classification problems can be solved. In binary classification one output is enough to disclose the two probabilities, of some input belonging to the two classes. This is thanks to the properties of probabilities in binary cases, see 6.16.

$$p(y = 0|x) = 1 - p(y = 1|x) \tag{6.16}$$

6.2.2 The Objective Function

One of the main reasons, why the sum of squared errors (equation 6.1) cannot be used as an objective function for logistic regression, is because the targets are only 0 and 1. Therefore we define a different cost function L as follows.

Definition 16. Given N targets t_n and N predictions y_n, the cost function L is defined as:

$$L = \prod_{n=1}^{N} y_n^{t_n} \cdot (1 - y_n)^{1-t_n} \tag{6.17}$$

Remark 5. This is the *Likelihood function* L, which is an objective function that shall be maximized. This functions works good for both cases of the target. If the target is 0, the first term becomes irrelevant, anything to the power of zero is 1. For the same reason the second term cancels out, if the target is 1. If the model works good, the likelihood of the model predicting correctly is high, which is why this objective function has to be maximized.

To maximize it however, $\log(L)$ is used, as it has the same maximum as L but is easier to calculate.

$$\begin{aligned} &\log(\prod_{n=1}^{N} y_n^{t_n} \cdot (1 - y_n)^{1-t_n}) \\ &= \sum_{n=1}^{N} \log(y_n^{t_n}) + \log((1 - y_n)^{1-t_n}) \end{aligned}$$

$$l = \sum_{n=1}^{N} t_n \cdot \log(y_n) + (1 - t_n) \cdot \log(1 - y_n) \tag{6.18}$$

6.2.3 Optimizing the Weights

Just like in linear regression, $\frac{\partial J}{\partial w} = 0$ does not always work. Therefore gradient descent will also be used for logistic regression.

The easiest way to take the derivative of J is, again, to use the chain rule.

$$\frac{\partial J}{\partial w_i} = \sum_{n=1}^{N} \frac{\partial J}{\partial y_n} \frac{\partial y_n}{\partial a_n} \frac{\partial a_n}{\partial w_i}$$

The sigma function will be used as the sigmoid function. The following are the functions that have to be derived.

$$\begin{aligned} a &= w^T X \\ y &= \frac{1}{1 + e^{-a}} \\ J &= l \end{aligned}$$

The derivative of a is the same as in linear regression.

$$\frac{\partial a_n}{\partial w_i} = x_{ni}$$

$$\begin{aligned}
\frac{\partial y_n}{\partial a_n} &= (-1) \cdot (1 + e^{-a_n})^{-2} \cdot e^{-a_n} \cdot (-1) \\
&= \frac{1}{1 + e^{-a_n}} \cdot \frac{e^{-a_n}}{1 + e^{-a_n}} \\
&= y_n \cdot (\frac{1 + e^{-a_n}}{1 + e^{-a_n}} - \frac{1}{1 + e^{-a_n}}) \\
&= y_n \cdot (1 - y_n)
\end{aligned}$$

$$\frac{\partial J}{\partial y_n} = \frac{t_n}{y_n} - \frac{1 - t_n}{1 - y_n}$$

Now all the derivatives of the chain rule are substituted by the above terms and finally simplified.

$$\begin{aligned}
\frac{\partial J}{\partial w_i} &= \sum_{n=1}^{N} (\frac{t_n}{y_n} - \frac{1 - t_n}{1 - y_n}) \cdot y_n \cdot (1 - y_n) \cdot x_{ni} \\
&= \sum_{n=1}^{N} t_n \cdot (1 - y_n) \cdot x_{ni} - (1 - t_n) \cdot y_n \cdot x_{ni} \\
&= \sum_{n=1}^{N} x_{ni} \cdot (t_n - t_n \cdot y_n - y_n + t_n \cdot y_n)
\end{aligned}$$

$$\frac{\partial J}{\partial w_i} = \sum_{n=1}^{N} (t_n - y_n) \cdot x_{ni} \tag{6.19}$$

For easier implementation in Numpy this will now be put into vectorized form.

$$\sum_{n=1}^{N} (t_n - y_n) \cdot x_{ni} = X^T (T - Y) \tag{6.20}$$

6.2.4 Implementation

Logistic regression, too, can be implemented using just Python and Numpy.

Making Predictions

The input, again, is a matrix of the dimensions $N \times D$. And like in linear regression, a column of ones is added to X to absorb the bias term into the weights.

```
100 N, D = X.shape
101 ones = numpy.ones((N, 1))
102 Xb = numpy.concatenate((ones, X), axis=1)
```

w has to have the same dimensions as X with the column of ones appended to it. Before training the model, it is good practice to initialize the weights randomly.

```
103 w = numpy.random.randn(D + 1)
```

a is calculated just like in linear regression.

```
104 a = Xb.dot(w)
```

And finally, making predictions is done like in the following listing:

```
105 prediction = 1 / (1 + numpy.exp(-a))
```

Training the Model

For training the model, the *log-likelihood* is chosen as the objective function. The following hyperparameters are used:

- Number of iterations: 1000
- Learning rate η: 0.001

In order to see the improvement of the model per iteration, the log-likelihood is used. If the log-likelihood increases in an iteration, this means that the model has improved. To calculate it, a function will be defined:

```
106 def log_likelihood(T, Y):
107     likelihood = 0
108     for i in range(len(T)):
109         if T[i] == 1:
110             likelihood = likelihood + numpy.log(Y[i])
111         if T[i] == 0:
112             likelihood = likelihood + numpy.log(1 - Y[i])
113     return likelihood
```

Now training the model using gradient descent is just a few lines of code.

```
114 costs = []
115 learning_rate = 0.001
116 for t in range(1000):
117     Yhat = Xb.dot(w)
```

```
118     delta = Y - Yhat
119     w = w + learning_rate * Xb.transpose().dot(delta)
120     l = log_likelihood(Y, Yhat)
121     costs.append(l)
```

When this code has finished running, the model is trained. The development of the model can be visualized using matplotlib, the costs (log-likelihood) per iteration is plotted (see figure 6.9).

```
122 import matplotlib.pyplot as plt
123
124 plt.plot(costs)
125 plt.show()
```

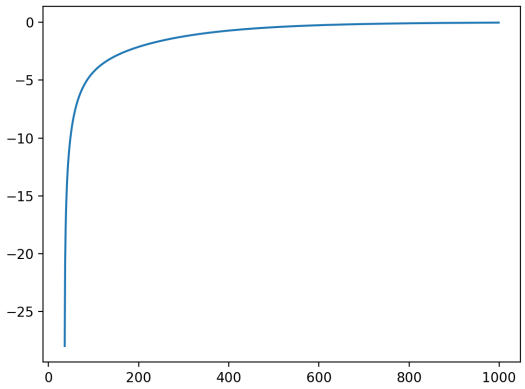


Figure 6.9: An exemplary plot of the costs per iteration.

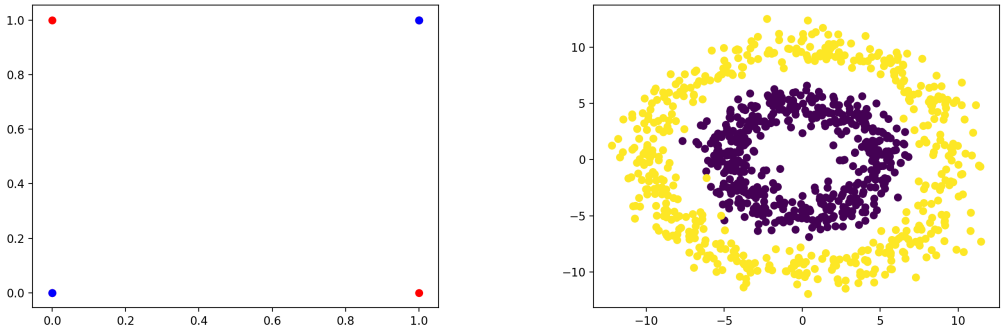
6.2.5 Final Notes on Logistic Regression

Limits of Logistic Regression

Logistic regression is limited by the fact, that it can only classify data that is *linearly separable*. This means, that it is only good at classifying data, that can be separated by a line, a plane, or a hyperplane.

There are two demonstrative problems to showcase the implications of the limits of logistic regression. The *donut problem* (figure 6.10b) and the *XOR problem* (figure 6.10a).

In both cases, there is no line that can separate the classes. In order for this to work, one would need to feature engineer another dimension into the problems, so that in the third dimension, there is a plane that can separate the groups.



(a) The XOR problem

(b) The donut problem

Figure 6.10: Unsolvable problems with Logistic Regression

6.3 Cluster Analysis - Unsupervised Learning

The information for this section has been taken from [33]. Extra used resources are listed. In most cases, especially when problems occur in the real world, data is not provided with predefined labels. Therefore, Machine Learning models that can correctly classify this data by identifying features in common have been developed.

The most popular unsupervised learning technique is clustering, where data is grouped into so-called clusters based on the similarity of the data. At this point we do not go into detail concerning the notion of "similarity". We only assume that there exists a measure to determine the similarity between two elements.

The final aim of clustering is to find underlying patterns or hidden structures in the data and also to isolate different elements from each other. In other words data points in the respective clusters should have several common properties, which makes them distinguishable from data points in other clusters.

According to Amirsina Torfi [11] at first it may sound like classification, but one should keep in mind that clustering and classification try to solve two very different problems. To determine new, potential groups in a data set, clustering is used while classification is used to assign a new input to an already existing cluster.

Suppose a bank wants to better understand the structure of their client base. Therefore, they analyze each customer concerning their debts and income situation. In figure 6.11 the X-axis represents the income and the Y-axis the amount of debt of each customer. From figure 6.12 it is easily visible that the customers fall apart into 4 clearly separated customer types. Based on this knowledge the bank could take actions to, e.g., reduce the risk of granting a loan to an uniquid customer.

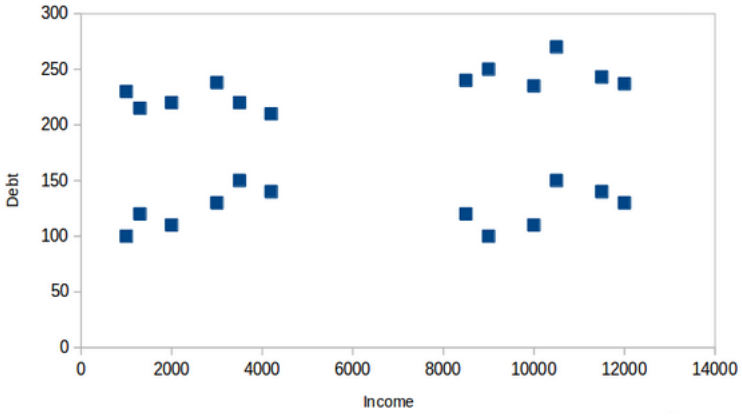


Figure 6.11: Customer data [33]

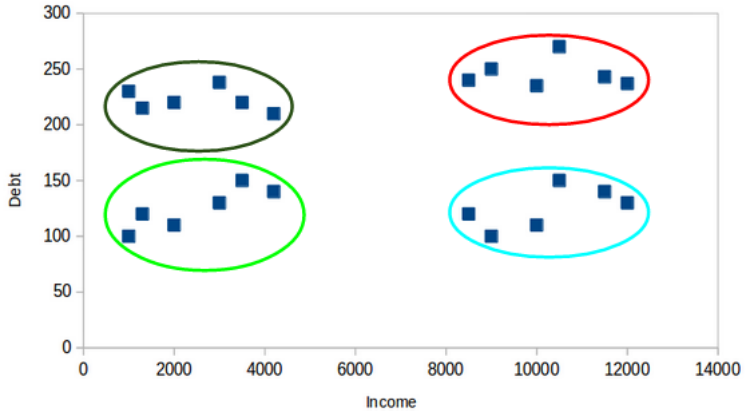


Figure 6.12: Clustered customer data [33]

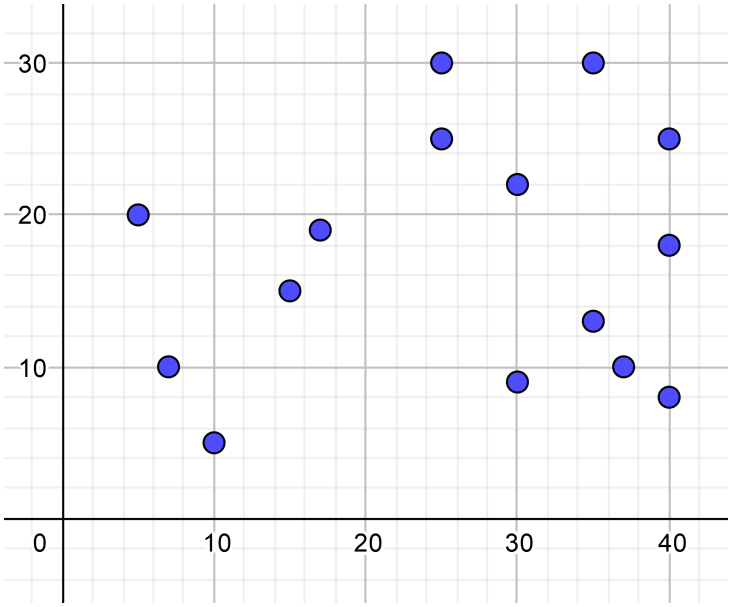


Figure 6.13: Plotted data

6.3.1 K-Means-Clustering

K-Means is one of the most commonly used clustering algorithms for grouping objects. The goal of the algorithm is to find the data points with high similarity and to group them into k clusters. The variable k is only the number of clusters used.

It is a centroid-based algorithm, or a distance-based algorithm, where the distances between points are calculated to assign a point to a cluster. In K-Means, each cluster is associated with the most representative point within the group, called a centroid.

The algorithm aims to minimize the sum of distances between the points and their associated cluster centroid. Resulting of the algorithm are the centroids of each cluster, which can be used to assign new data points to a cluster and the training data used are now labelled, which means that they are all assigned to an associated cluster.

Functionality of the Algorithm

The functionality of K-Means is additionally taken from [12] and is now explained using an example because it is easier to follow.

Assume the dataset looks like as given in figure 6.13. With regard to the goal of clustering, there are a number of unmarked data points that need to be grouped together. Typically, these clusters are marked with numbers by the algorithm itself. What is implicitly needed is a decision boundary that separates the different groups.

Firstly, the number of clusters k has to be selected. In this example three clusters are used. There are some ways to find the optimal number of required clusters, explained

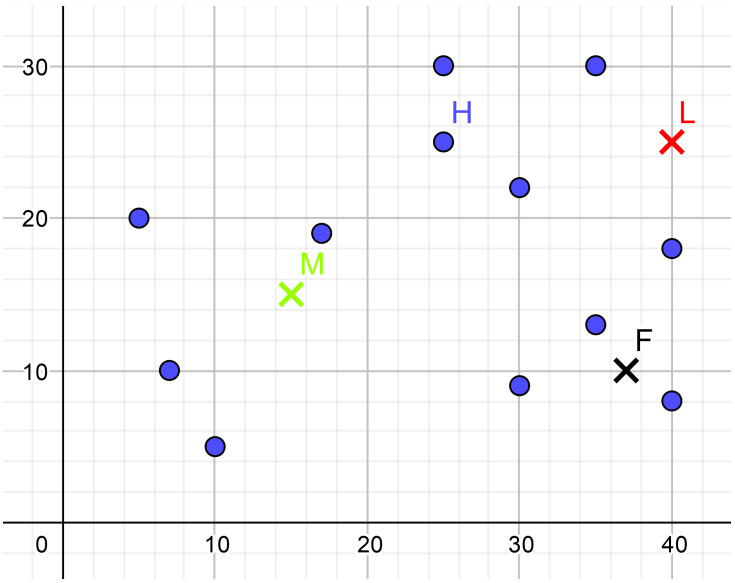


Figure 6.14: Plotted data with cluster centroids

here 6.3.1. Now a value for the starting centroid is required for each cluster. There are several ways to select the midpoints. For example, the values can be chosen randomly. In the beginning, this may be a good and quick fix. In some cases, however, this can lead to faulty clusters if the random initialization is not suitable.

To work around this problem, there is the K-Means ++ algorithm. This algorithm makes it more likely to find a solution that competes with the optimal K-Means solution. Another possibility would be to select the cluster centroids according to randomly selected points of the existing data points, which is done in this example for simplicity.

In figure 6.14 the red, green and black cross represents the centroids of the clusters. As the next step, the algorithm calculates the distance to all cluster centroids for each data point. Then each data point is grouped to the nearest centroid (smallest distance).

To calculate the distance between two points in plane or in multidimensional space, the *Euclidian distance* is used. Further reading about the Euclidean distance can be done here [38]. For representing the distance between two points, this is the most common method. If the points p and q are given by the coordinates $p = \{p_1,...,p_n\}$ and $q = \{q_1,...,q_n\}$ in a Cartesian coordinate system, the distance value d from p to q or from q to p is calculated using the Pythagoras formula. In general, the distance for an n-dimensional space is defined as the following:

$$d(p,q)=d(q,p)=\sqrt{(p_1-q_1)^2+(p_2-q_2)^2+...+(p_i-q_i)^2}=\sqrt{\sum_{i=1}^{n}(p_i-q_i)^2} \qquad (6.21)$$

To demonstrate this formula with an example, the Euclidean distance to each cluster

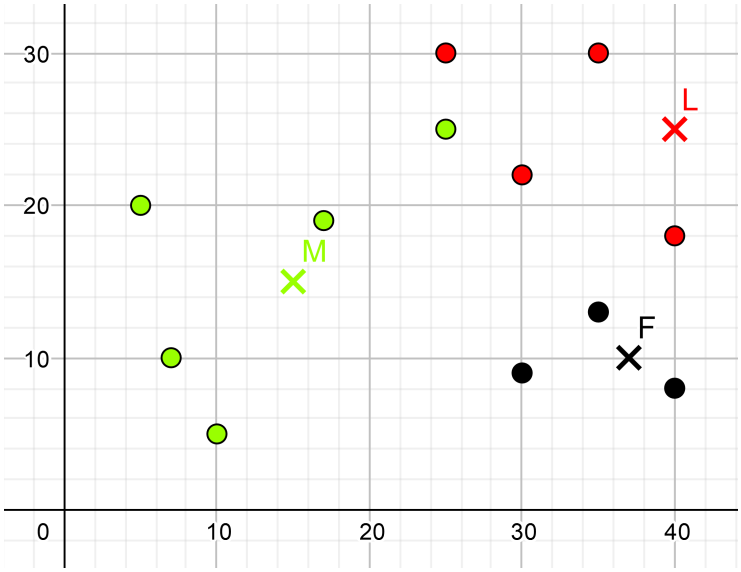


Figure 6.15: Assigning each data point to its nearest cluster

Centroid is now calculated for the point H. The exact coordinates for the point H and the cluster Centroids M, L, F are:

$$H{=}(25,25), M{=}(15,15), L{=}(40,25), F{=}(37,10)$$

$$d(H,M){=}\sqrt{(25-15)^2+(25-15)^2}{=}14.142$$
$$d(H,L){=}\sqrt{(25-40)^2+(25-25)^2}{=}15$$
$$d(H,F){=}\sqrt{(25-37)^2+(25-10)^2}{=}3\sqrt{41} \cong 19.21$$

The algorithm now uses the three results to determine the shortest distance. The point H is then assigned to the respective cluster, in this case M. At first glance, one might think that the cluster centroids L and M are relatively equidistant from each other or that the centroid L is perhaps closer to H than M, which is unfortunately only an optical illusion.

After each data point has been matched to its nearest cluster centroid, what can be observed in 6.15, the mean values are recalculated, i.e. the values of the centroids. A cluster centroid C is defined as the following:

$$C(x_c,y_c){=}(P_1,P_2,...,P_N){=}((x_1,y_1),(x_2,y_2),...,(x_N,y_N))$$

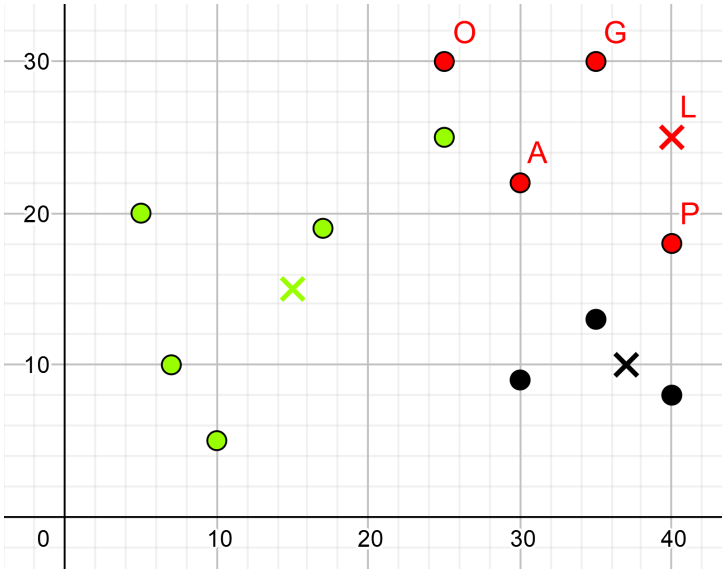


Figure 6.16: Calculate the new mean value for the red cluster

P_N is just representing a point in a cartesian coordinate-system belonging to the cluster, consisting of a x_N and y_N value. The new value x_c or y_c of a centroid is calculated by summing all x_c respectively y_c values of all points in the according cluster and then divide the sum by the number of points N, described with the following formula:

$$x_c = \frac{\sum_{i=1}^{N} x_i}{N}$$
$$y_c = \frac{\sum_{i=1}^{N} y_i}{N}$$

To illustrate this, the new mean value for the red cluster centroid is calculated. The following five coordinate points are required for the calculation, as can be seen in 6.16.

$$O=(25,30), A=(30,22), G=(35,30), P=(40,18), L=(40,25)$$

Note that the point L is required for the calculation, because the cluster centroids were selected randomly from the given points at the beginning. If the point L is simply omitted in the calculation, a point is just ignored and forgotten. After that the calculation would be faulty. Now the new x and y values are determined for the red centroid:

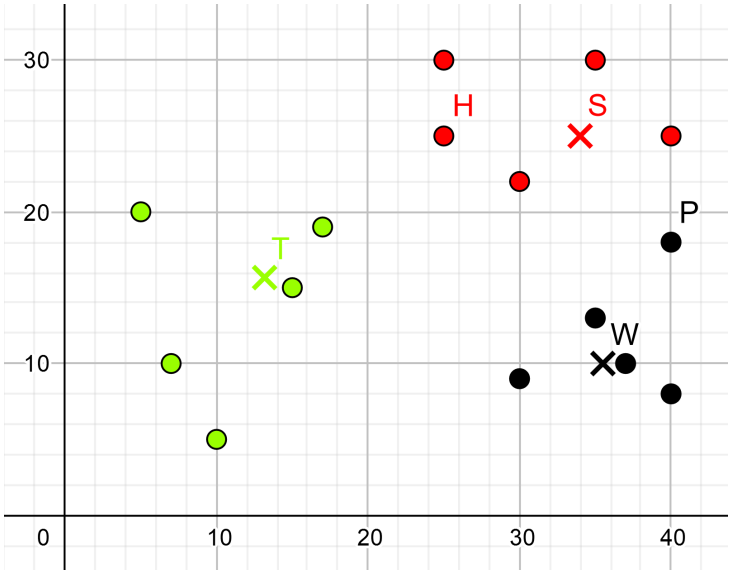


Figure 6.17: Recalculation and reassignment after first iteration

$$x = \frac{25 + 30 + 35 + 40 + 40}{5} = 34$$
$$y = \frac{30 + 22 + 30 + 18 + 25}{5} = 25$$

After calculating the new mean values, the previous step is repeated, which is the assignment of each point to its closest cluster centroid. As the centroids change, some data points will also change clusters. After all points are assigned, the mean values for the centroids are recalculated. As it can be seen, these two steps are being repeated until certain conditions are achieved. These certain criteria are explained here 6.3.1. After a further iteration of the two steps the data points look like in 6.17.

Two points have changed clusters. The point H was in the green cluster after the first iteration, but is now in the red cluster. Point P has changed from the red to the black one. It also shows how all cluster centroids have changed compared to the graph 6.15. The final result is shown figure 6.18.

Stopping Criteria for K-Means Clustering

Basically, there are three cancellation conditions that should be applied to stop the K-Means algorithm:

- The same number of points remain in the same cluster, also after further iterations.

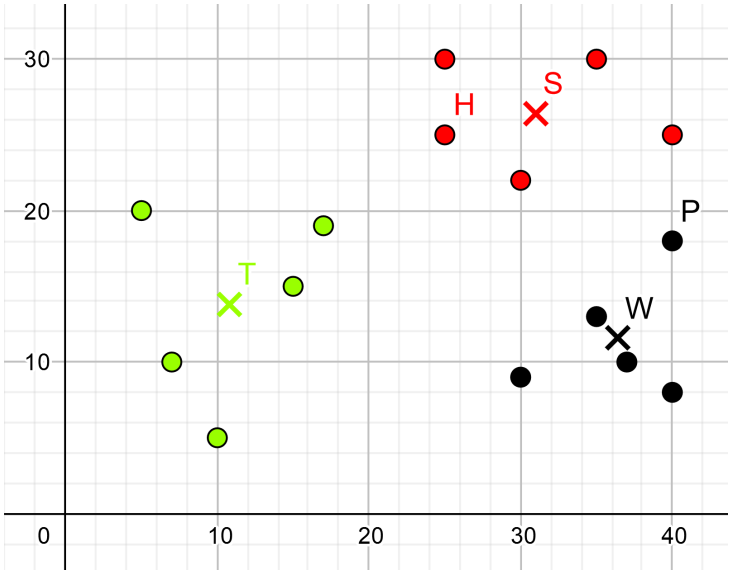


Figure 6.18: Final result

- The centroids of newly created clusters do not differ from the previous one. Even after multiple iterations, if the centroids are identical for all clusters, the algorithm is not learning a new pattern and it is a signal to end the process.
- Lastly, once the maximum amount of iterations has been reached, the training can be stopped. Suppose the number of iterations is 50. The process is then repeated for 50 iterations before it is aborted.

Choosing the Optimal K

Probably the most common challenge people have when working with K-Means is choosing the right number of clusters. There is unfortunately no clear answer for this problem. Determining the optimal number of clusters is kind of subjective and is dependent on the method which is used to measure similarities. However, there are a few methods to roughly determine this number and now one of them is explained in more detail.

Elbow Method It is probably the best known method to determine the amount of required clusters, where the sum of the distances is computed for each number of cluster, and visualized graphically. To calculate the distances, the formula which is declared here 6.21, is used. The K-Means clustering models are built by increasing the number of clusters iteratively from a starting point to an endpoint. With an increasing value of k, there will be less elements in the cluster therefore the average distortion is decreased. The smaller amount of elements means a greater proximity to the centroids. The user then looks for a change of slope from steep to flat (one elbow) to identify the most

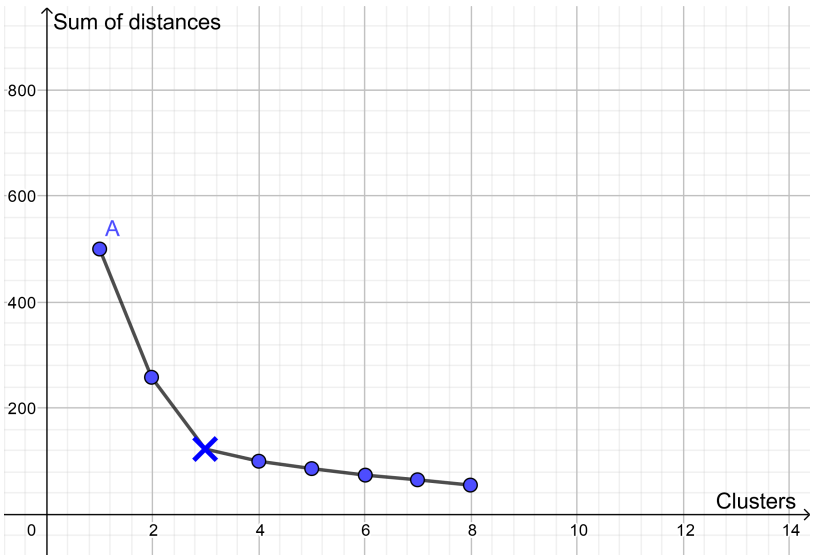


Figure 6.19: Elbow method

appropriate number of clusters and that is the point at which the distortion decreases most. Although this method is inaccurate, it is still potentially useful.

As shown in figure 6.19 the Y-Axis is the sum of the distances and the X-Axis is the amount of clusters. Clearly, the elbow is formed at k=3, marked with a cross, therefore the optimal value is 3 for performing K-Means in this example.

Chapter 7

Deep Learning

This chapter addresses the concepts and fundamentals of Deep Learning, and builds on the concepts of chapter 6. The resource for the information presented in this chapter is [2].

7.1 Feedforward Neural Networks with 1 Hidden Layer

Neural networks are made to solve *nonlinear* problems. As discussed in 6.2.5, both linear regression and logistic regression work under the assumption, that the data is linear and linearly separable, respectively. For classification problems, neural networks can find nonlinear boundaries separating the data. They achieve *nonlinearity* by combining logistic regression units. Furthermore, neural networks can do multiclass classification and multiple regression, meaning that they can have more than just one output.

7.1.1 The Basic Concept

Neural networks consist of *nodes*. An input unit, meaning a dimension of the input, is a node, a logistic regression unit is a node and a output unit is a node as well. The nodes are grouped into layers, in figure 7.1 there are three layers. An input layer consisting of three input nodes (X is therefore 3-dimensional), one hidden layer consisting of 4 logistic regression units and an output layer consisting of two softmax units. Softmax units are explained in detail in 7.1.2. In traditional feedforward neural networks it is essential that *every* node of a layer is connected to *every* node of the next layer.

The Architecture

A node in a neural network is something, that yields a value. The output of a node in layer is an input for every node in the next layer. In both figures 7.1 and 7.2 the lines represent the weights for the inputs of each node. To get a better understanding of how getting an output y of a neural network works, figure 7.2 explains how each layer generates an output. The input layer is denoted as X, the hidden layer is denoted as Z

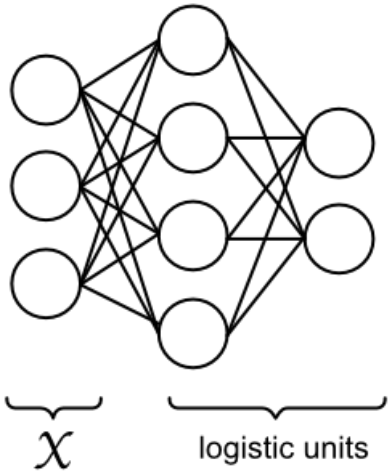


Figure 7.1: The concept of Neural Networks - the circles represent nodes

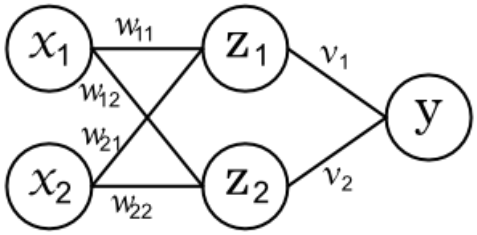


Figure 7.2: Example architecture with denotations for clarity

and the output layer is conventionally denoted as Y.

$$Z = sigmoid(w^T X + b)$$
$$Y = sigmoid(v^T Z + c)$$

In scalar form these equations would look like:

$$z_1 = \sigma(x_1 * w_{11} + x_2 * w_{12})$$
$$z_2 = \sigma(x_1 * w_{21} + x_2 * w_{22})$$
$$y = \sigma(z_1 * v_1 + z_2 * v_2)$$

Note, that for ease of notation the weights between the hidden layer and the output layer are denoted as v.

A further important element is, that nonlinear functions, sigmoid functions, are inputs to other nonlinear functions. This is what enables neural networks to distinguish nonlinear classes, and what enables them to predict nonlinear coherences.

A big benefit of neural networks is that they are highly modular. The number of hidden layers, such as the number of units in the hidden layers, are freely selectable. The complexity of calculations and the time it takes to calculate outputs and train the networks however are directly correlated with the number of layers and number of units in the layers. But for simplicity's sake, this section will only cover neural networks with one hidden layer.

Denotation

With neural networks come new matrices and layers that have to be named and denoted. For better readability, the number of units in the hidden layer such as the weights from the hidden layer to the output layer will be named v, even though this does not conform to the convention. The convention of course generalizes the naming, so the weights would be named w regardless of the layer they are in. The convention suggests the weights to be named:

$$w^{(l)}_{m^l m^{l+1}}$$

The weights are mainly indexed l.

- The superscript l indexes the layer the weight is in
- m^l indexes the which node of layer l the weight is connected to
- m^{l+1} indexes the which node of layer $l+1$ the weight is connected to

In this section the weights are named w_{ij} for the input to hidden layer, with i indexing the node of the input layer and j indexing the node of the hidden layer. Weights connecting the hidden layer to the output layer are named z_{ij}, with i indexing the node of the hidden layer and j indexing the output node.

The number of units in the hidden layer is denoted by M, the number of units in the output layer is denoted by K The weight matrix Z is therefore of the dimensions $M \times K$. The output matrix Y now has the dimensions $N \times K$.

Compared to logistic regression and linear regression, the input is now an input to M hidden units in the hidden layer, therefore the weight matrix W is not 1-dimensional anymore, but instead it is of shape $D \times M$.

Interpreting the Weights of Neural Networks

In linear and logistic regression, interpreting the weights is a straight forward process. The weights are indicators for how much an input dimension affects the output. With neural networks this process is not so easy anymore. The problem is, that the weights of neurons past the input layer must not have a real world meaning. With the purpose of neural networks being solving nonlinear problems, their weights resemble abstract geometry and boundaries that make nonlinear classification and regression possible. Because of the sigmoid functions causing nonlinearities, the weights beyond the first

layer represent features that the neural network extracts. The deeper the layer, the more abstract are the features being extracted.

With neural networks, not much effort should be put into trying to interpret the weights. More interesting and a bigger area of concern is trying to see what the decision boundaries of a network look like.

7.1.2 Multiclass Classification

A limitation of logistic regression is, that it can only perform binary classification. Neural networks extend this approach, by using a *softmax* layer as the output layer, instead of using multiple sigmoid units. If multiple sigmoid units would be used, the sum of all the probabilities of the input belonging to the K different classes would not be 1. The probabilities however should always sum to one, as the probability of the input belonging to a class is 100%. Calculating the probability of an input belonging to an output is done like equation 7.1.2 shows.

$$P(y = k|x) = \frac{e^{a_k}}{\sum_{i=1}^{K} e^{a_i}} \tag{7.1}$$

$$a = v^T Z$$

This approach works for any K, even for $K = 2$. In that case, it even is the same as using one sigmoid unit, as equation 7.2 shows.

$$\frac{e^{v_1^T Z}}{e^{v_1^T Z} + e^{v_2^T Z}} * \frac{\frac{1}{e^{v_1^T Z}}}{\frac{1}{e^{v_1^T Z}}}$$

$$= \frac{1}{1 + \frac{e^{v_2^T Z}}{e^{v_1^T Z}}}$$

$$= \frac{1}{1 + e^{(v_2 - v_1)^T Z}} \tag{7.2}$$

Furthermore this equation shows that by using the softmax layer for binary classification some weights are redundant. From a development standpoint however it is safer and less work to always use the softmax approach.

This concludes how predictions are made in feedforward neural networks.

7.1.3 Implementation of Making Predictions

7.2 Training Neural Networks with 1 Hidden Layer

In chapter 6, *gradient descent* was introduced. Gradient descent is a technique of minimizing a function without finding its global minimum using calculus. Instead, the

parameters of the function are tuned in an iterative fashion, where each iteration the parameters are tweaked in the direction of the slope of the function.

Gradient descent can also be used to train neural networks, the process however is slightly more complicated than it was in previous chapters of this thesis, as neural networks are highly complicated. Just like in chapter 6 the weights, of in this case the neural network, have to be tweaked for some error function to be minimal. Weights of layers at the top of the networks however affect the weights of deeper layers, which is why the derivative of the objective function is more complicated to calculate.

7.2.1 The Objective Function for Multiclass Classification

At first, a new objective function has to be found, as previously predictions were either made for regression or *binary* classification problems. This chapter however introduces *multiclass* classification - the binary cross entropy function is not applicable anymore. This is why the *categorical cross entropy* function will now be introduced. The categorical cross entropy is based on the categorical distribution, or multinoulli distribution, where the probability of predicting the correct category of running a random experiment once is calculated as follows.

$$\prod_{k=1}^{K} w_k^{t_k}$$

K is the number of categories, w_k is the probability of the random variable being of category k and t_k is the state of the random variable after the random experiment being run once. The categorical cross entropy is, as now apparent, yet another way of measuring the likelihood of predicting correctly. The categorical cross entropy extends the above formula by multiplying the likelihoods of all samples, which is allowed, as the samples are independent (in probability theory this is the equivalent of two random events being independent, where the probability of both events occurring can therefore be multiplied).

$$L = \prod_{n=1}^{N} \prod_{k=1}^{K} y_{nk}^{t_{nk}} \tag{7.3}$$

Like with previously introduced objective functions, it is easier to take the log of 7.3, as it has the same minima and maxima but is easier to derive.

$$J = \log(L) = \sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} * \log(y_{nk}) \tag{7.4}$$

The in 7.4 defined objective function J is one that shall be maximized, as it is desirable to have a high likelihood of predicting correctly. If the target t_{nk} is 1 and the prediction y_{nk} is 0 (in the real world a value close to 0, as 0 would mean that the model is absolutely sure that the input n does not belong to category k - which almost never is the case), the likelihood is lower than if the prediction were 1 (or a value close to 1).

7.2.2 A Short Return to Logistic Regression

The avid reader will notice, that the output layer of a neural network (as seen in figure 7.1) is basically just multiple logistic regression units combined by a softmax layer.

This subsection will show how to derive the objective function J from equation 7.4 with respect to the weights of logistic regression for multiclass classification. This will prove to be helpful in later sections. Once more, the objective J is a function of a function of a function ... To gain a comprehensive view of the functions involved, they will be split into their individual parts:

$$\begin{aligned} a &= W^T x \\ y &= softmax(a) \\ J &= \sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} * \log(y_{nk}) \end{aligned}$$

The gradient of J has to be found for each individual weight.

$$\frac{\partial J}{\partial w_{ik}} = \sum_{n=1}^{N} \sum_{k'=1}^{K} \frac{\partial J}{\partial y_{nk'}} \frac{\partial y_{nk'}}{\partial a_{nk}} \frac{\partial a_{nk}}{\partial w_{ik}}$$

The weights are indexed by i and k. i indexes the input the weight belongs to and k indexes the output which the weight connects to the input. The most interesting and confusing part of this equation, is when the index k switches from k' to k. This is confusing at first, however it becomes clear when taking a closer look at y.

$$y_{nk} = softmax(a) = \frac{e^{a_{nk}}}{\sum_{i=1}^{K} e^{a_{ni}}}$$

y_{nk} does not only depend on a_k, instead it depends on all a's!

Getting back to J, this means, that y_{nk} not only depends on the weights of its specific output nodes, but on all weights. The implication of this circumstance is that the derivation of J is not as straightforward as it seems, because there are 2 cases of the derivative $\frac{\partial y_{nk'}}{\partial a_{nk}}$. a_{nk} is either in the numerator or it is not. The derivatives look different for the 2 cases for obvious reasons. This derivation will be looked into after the other 2, less complex, derivatives.

$$\begin{aligned} \frac{\partial J}{\partial y_{nk'}} &= \frac{t_{nk'}}{y_{nk'}} \\ \frac{\partial a_{nk}}{\partial w_{ik}} &= x_{ni} \end{aligned}$$

Now to the complex derivative where k switches to k'. In both cases it is easer to look at the derivative the following way:

$$\frac{\partial y_{nk'}}{\partial a_{nk}} = \frac{\partial}{\partial a_{nk}} e^{a_{nk'}} * \left(\sum_{i=1}^{K} e^{a_{ni}} \right)^{-1}$$

In the case of $[k' \neq k]$:

$$\begin{aligned}\frac{\partial y_{nk'}}{\partial a_{nk}} &= e^{a_{nk'}} * \left(\sum_{i=1}^{K} e^{a_{ni}}\right)^{-2} * e^{a_{nk}} * (-1) \\ &= -\frac{e^{a_{nk'}}}{\sum_{i=1}^{K} e^{a_{ni}}} \frac{e^{a_{nk}}}{\sum_{i=1}^{K} e^{a_{ni}}}\end{aligned}$$

This term can even be simplified further, it can be expressed in terms of y.

$$\frac{\partial y_{nk'}}{\partial a_{nk}} = -y_{nk'} * y_{nk}$$

The derivative in the case of $[k' = k]$ is more complicated as it involves the chain rule of calculus:

$$\begin{aligned}\frac{\partial y_{nk'}}{\partial a_{nk}} &= e^{a_{nk'}} * \left(\sum_{i=1}^{K} e^{a_{ni}}\right)^{-1} - e^{a_{nk'}} * \left(\sum_{i=1}^{K} e^{a_{ni}}\right)^{-2} * e^{a_{nk'}} \\ &= \frac{e^{a_{nk}}}{\sum_{i=1}^{K} e^{a_{ni}}} - \left(\frac{e^{a_{nk}}}{\sum_{i=1}^{K} e^{a_{ni}}}\right)^{2} \\ &= \frac{e^{a_{nk}}}{\sum_{i=1}^{K} e^{a_{ni}}} * \left(1 - \frac{e^{a_{nk}}}{\sum_{i=1}^{K} e^{a_{ni}}}\right)\end{aligned}$$

This too can be expressed in terms of y.

$$\frac{\partial y_{nk'}}{\partial a_{nk}} = y_{nk} * (1 - y_{nk})$$

To summarize, here are both cases next to one another:

$$[k' \neq k] : y_{nk'} * (0 - y_{nk}) \tag{7.5}$$

$$[k' = k] : y_{nk'} * (1 - y_{nk}) \tag{7.6}$$

In the gradient of J this unfortunately does not fit in well as it complicates all the operations done in later steps. Keeping this logic in the gradient would result in tremendous performance drops. Therefore it is a good idea to simplify this logic, possibly into one term. The kind of function necessary to unite the two cases of k would be a function

that yields 0 if k' and k are not equal, and that yields 1 if k' and k are equal. A possible function for this use case is the *Kronecker Delta-Function*. It does exactly that.

$$\delta_{kk'} = 0 : k' \neq k$$
$$\delta_{kk'} = 1 : k' = k$$

Now the derivative of y with respect to a can be simplified in one line.

$$\frac{\partial y_{nk'}}{\partial a_{nk}} = y_{nk'} * (\delta_{kk'} - y_{nk})$$

Finally the derivative of the objective function J with respect to a particular weight w can be pieced together with all the derivations found above.

$$\begin{aligned}\frac{\partial J}{\partial w_{ik}} &= \sum_{n=1}^{N}\sum_{k'=1}^{K}\frac{t_{nk'}}{y_{nk'}} * y_{nk'} * (\delta_{kk'} - y_{nk}) * x_{ni} \\ &= \sum_{n=1}^{N}\sum_{k'=1}^{K} t_{nk'} * (\delta_{kk'} - y_{nk}) * x_{ni}\end{aligned}$$

Unfortunately the kronecker delta is still a bête noire. Because of it the first degree derivative of the objective function with respect to the weights is not *smooth* anymore. By using a bit of logic one can however remove it from the equation, as the following steps show. These steps only show parts important for the logic of removing the delta function.

$$\begin{aligned}&\sum_{k'=1}^{K} t_{nk'} * (\delta_{kk'} - y_{nk}) \\ &= \sum_{k'=1}^{K} t_{nk'} * \delta_{kk'} - t_{nk'} * y_{nk} \\ &= \sum_{k'=1}^{K} t_{nk'} * \delta_{kk'} - \sum_{k'=1}^{K} t_{nk'} * y_{nk}\end{aligned}$$

The attentive reader will at this point notice to parts which can be simplified.

Firstly, the kronecker delta function always yields 0 if k and k' are unequal. The delta function yields 1 only once. In all other cases, this summation sums up $0 * t_{nk'}$ with one single occurrence of $1 * t_{nk'}$. This whole term is therefore equal to $t_{nk'}$.

Secondly, the latter term contains a summation over all the targets. In the specific logic of multiclass classification, a certain input into the model can only contain to one class, therefore only one target is 1, whereas all the other targets are 0. Therefore the only relevant part of the second summation is $t_{nk'} * y_{nk} = y_{nk}$.

The term now is as simple as:

$$\sum_{k'=1}^{K} t_{nk'} * \delta_{kk'} - \sum_{k'=1}^{K} t_{nk'} * y_{nk} =$$
$$t_{nk} - y_{nk}$$

making the gradient of the objective function as simple as:

$$\frac{\partial J}{\partial w_{ik}} = \sum_{n=1}^{N} (t_{nk} - y_{nk}) * x_{ni} \tag{7.7}$$

For easier implementation this term is also given in matrix format:

$$\nabla J = X^T (T - Y)$$

7.2.3 Backpropagation Principals

This subsection introduces the concept of backpropagation. *Backpropagation* is the goto technique for training all kinds of neural networks. At its core it is a simple concept, as it basically just determines the order in which the weights of a neural network are tuned in gradient descent. Usually in gradient descent all weights are trained at the same time. In code, this is done in just a few lines, as matrices are being worked with. The weights of different layers have different sizes however, in neural networks with one hidden layer the weights W are of size $D \times M$ and the weights V are of size $M \times K$, therefore they cannot be trained at the same time. At this point the question arises, in which order the gradient descent for W and V has to be executed. The obvious approach would be to simply train W first, and then train V.

There is however a problem in this approach, which is that the gradient of W *depends* on V. When tuning W before tuning V, possibly the error of the network might not have improved at all, because the updates to V caused the gradient of W to be reversed.

This is why training neural networks happens *backwards*. The deeper layers are trained before the shallower layers. The gradients of deeper weights are not affected by the values of weights of shallower layers. The answer to why, in the specific case of neural networks with 1 hidden layer, the weights W depend on the weights V lies in the derivatives of the cost function J, which will become clear in the next subsection.

This concludes the explanation of the core principals of *Backpropagation*, and after reading the above the naming of this concept becomes intuitive.

7.2.4 Applying Backpropagation to Neural Networks with 1 Hidden Layer

A good first step is to visualize the dimensions of the different weight-matrices and node-matrices involved.

- $X \ldots N \times D$
- $W \ldots D \times M$
- $b \ldots M$
- $Z \ldots N \times M$

- $V \ldots M \times K$
- $c \ldots K$
- $Y \ldots N \times K$

Bias terms are introduced again, notice how the bias that connects the hidden layer to the output layer is denoted by c.

As explained above, the first step is to update the weights and biases connecting the hidden layer to the output layer. Therefore the derivative of J with respect to V and c has to be calculated. At this point it is apparent why the return to logistic regression above was helpful, as the gradient of J with respect to the weights V is calculated exactly the same way as the gradient of J with respect to the weights of multiple logistic regression. All that changes are the names of the weights. It is once more also helpful to write out all the functions involved, as this makes finding the derivative easier.

$$\begin{aligned} a_{nk} &= V_{:,k}^T z_n + c_k \\ y_{nk} &= softmax(a_{nk}) \\ J &= \sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} * \log(y_{nk}) \end{aligned}$$

Now the only new circumstance is the different notation, the calculation remains the same as with multiple logistic regression.

$$\begin{aligned} \frac{\partial J}{\partial V_{mk}} &= \sum_{n=1}^{N} \sum_{k'=1}^{K} \frac{\partial J_{nk'}}{\partial y_{nk'}} \frac{\partial y_{nk'}}{\partial a_{nk}} \frac{\partial a_{nk}}{\partial V_{mk}} \\ &= Z^T (T - Y) \end{aligned}$$

The bias c also has to be optimized, so the derivative of J with respect to c also has to be calculated.

$$\frac{\partial J}{\partial c_k} = \sum_{n=1}^{N} \sum_{k'=1}^{K} \frac{\partial J_{nk'}}{\partial y_{nk'}} \frac{\partial y_{nk'}}{\partial a_{nk}} \frac{\partial a_{nk}}{\partial c_k}$$

Fortunately a lot of the derivations are already known, so there is only one more derivation to do:

$$\frac{\partial a_{nk}}{\partial c_k} = 1$$

Making the derivative of J with respect to c:

$$\frac{\partial J}{\partial c_k} = \sum_{n=1}^{N} (t_{nk} - y_{nk})$$

Unfortunately there is no convenient matrix notation for this sum, so the numpy implementation will be used.

```
100 grad_c = numpy.sum(T-Y, axis=0)
```

Gradients with Respect to the Weights W and the Bias b

Calculating the gradients of J with respect to the weights and bias from the input layer to the hidden layer is the conceptually most challenging part of section, as there are 5 layers of functions, and because there is a dependency between the weights W and the weights V. The effects of this become clear in the following steps.

Again it is helpful to write out all the functions involved in J at this point.

$$\begin{aligned}
\alpha &= W^T x + b \\
z &= \sigma(\alpha) \\
a &= V^T z + c \\
y &= softmax(a) \\
J &= \sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} * \log(y_{nk})
\end{aligned}$$

Notice, that in this section the activation function for the sigmoid functions of the hidden layer is denoted as α. Furthermore, σ is a placeholder for all sigmoid functions, however in this example the sigma function will be used. Nevertheless an alternative for plugging in another sigmoid function will be presented.

Once again, the chain rule of calculus is used to calculate the two important gradients.

$$\begin{aligned}
\frac{\partial J}{\partial W_{dm}} &= \sum_{k=1}^{K} \sum_{n=1}^{N} \sum_{k'=1}^{K} \frac{\partial J_{nk'}}{\partial y_{nk'}} \frac{\partial y_{nk'}}{\partial a_{nk}} \frac{\partial a_{nk}}{\partial z_{nm}} \frac{\partial z_{nm}}{\partial \alpha_{nm}} \frac{\partial \alpha_{nm}}{\partial W_{dm}} \\
\frac{\partial J}{\partial b_{m}} &= \sum_{k=1}^{K} \sum_{n=1}^{N} \sum_{k'=1}^{K} \frac{\partial J_{nk'}}{\partial y_{nk'}} \frac{\partial y_{nk'}}{\partial a_{nk}} \frac{\partial a_{nk}}{\partial z_{nm}} \frac{\partial z_{nm}}{\partial \alpha_{nm}} \frac{\partial \alpha_{nm}}{\partial b_{m}}
\end{aligned}$$

The most interesting part about these equations is, that K is being summed over *twice*. Previously the dummy variable k which indexes a existed outside of the summation, in these new derivatives it does not anymore. Why that implies that K must be summed over again, can be explained with the law of total derivatives.

$$\begin{aligned}
&f \dots f(x(t), y(t)) \\
\frac{df}{dt} &= \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}
\end{aligned}$$

Analogously J is a function *all* a's, and all a's are functions of the specific W_{dm} and b_m, respectively. By the law of total derivatives this means, every a has to be derived by W_{dm} and b_m. Except for the derivative of sigmoid functions, all other derivations have already been covered in some form or another in this chapter. And the derivative of the sigma function, which is the sigmoid function of choice in this section, was already

calculated in 6.

$$\frac{\partial a_{nk}}{\partial z_{nm}} = V_{mk}$$
$$\frac{\partial z_{nm}}{\partial \alpha_{nm}} = z_{nm} * (1 - z_{nm})$$
$$\frac{\partial \alpha_{nm}}{\partial W_{dm}} = x_{nd}$$
$$\frac{\partial \alpha_{nm}}{\partial b_m} = 1$$

Finally the gradients of J can be calculated.

$$\frac{\partial J}{\partial W_{dm}} = \sum_{k=1}^{K} \sum_{n=1}^{N} (t_{nk} - y_{nk}) * V_{mk} * z_{nm} * (1 - z_{nm}) * x_{nd} \tag{7.8}$$
$$\frac{\partial J}{\partial b_m} = \sum_{k=1}^{K} \sum_{n=1}^{N} (t_{nk} - y_{nk}) * V_{mk} * z_{nm} * (1 - z_{nm}) \tag{7.9}$$

For convenience reasons these terms can be put into vectorized form.

$$X^T (T - Y) V^T \odot Z \odot (1 - Z) \tag{7.10}$$

The $\odot$ sign represents element-wise multiplication of matrices.

And a more general form where other sigmoid functions can be used is the following:

$$X^T (T - Y) V^T \odot Z' \tag{7.11}$$

7.3 Neural Networks with an Arbitrary Number of Hidden Layers

As mention in previous parts of this chapter, neural networks can contain an arbitrary number of hidden layers with arbitrary numbers of hidden units. The fundamental principals however always remain the same. Every node, of every layer, is connected with every node of the previous layer. In other words, the output of any node is an input to every node of the respective next layer. For making predictions this means, that any node of any layer *depends* on *every* node of *every previous layer*. In terms of training neural networks this means, that the gradient of any weight in any layer depends on *every* weight of *every subsequent layer*. This in turn makes backpropagation more complicated for every new hidden layer. This section will enlarge upon backpropagation and finalize its formula into a form that is conventional. The previous section was a preparation for what is to come in this section.

7.3.1 Denotation

The previous section already introduced the naming and denotation convention for the weights of a neural network with an arbitrary number of hidden layers. This convention will be completed in the following paragraphs.

- The input is still denoted by X
- The activation functions are denoted as $a^{(l)}$ where l is the number of the hidden layer
- The output of the logistic units (the hidden nodes) is denoted by $z^{(l)}$
- The weights are denoted as $W^{(l)}$
- The biases are denoted as $b^{(l)}$
- The number of units in a hidden layer is denoted by $M^{(l)}$

Note the following:

- $M^{(0)}$ is equal to D, the dimensionality of the input
- $M^{(L+1)}$, where L is the last hidden layer, is equal to K, the number of outputs of the neural network

Activation functions and outputs of nodes are indexed by n and $m^{(l)}$, where n is the number of the sample of the input and $m^{(l)}$ is the unit m of the layer l. Weights are indexed by $m^{(l)}$ and $m^{(l+1)}$, where $m^{(l)}$ is the unit of the hidden layer l and $m^{(l+1)}$ is the unit of the layer which is right after l. Biases are indexed by $m^{(l)}$, where once again m is the node the bias is for and l is the layer the node is in.

7.3.2 Training Neural Networks with 2 Hidden Layers

After the above sections the next logical step is to find the gradients of the objective function with respect to the weights of a neural network with 2 hidden layers. This is a necessary step in the progression of explaining backpropagation, as in 2 hidden layer neural networks for the first time the patterns, which backpropagation follows, show themselves clearly. The manner in which backpropagation was explained previously continues in this section, and once again it is a good first step to write out all the functions that are nested within J.

$$
\begin{aligned}
a^{(1)} &= (W^{(1)})^T X + b^{(1)} \\
Z^{(1)} &= \sigma(a^{(1)}) \\
a^{(2)} &= (W^{(2)})^T Z^{(1)} + b^{(2)} \\
Z^{(2)} &= \sigma(a^{(2)}) \\
a^{(3)} &= (W^{(3)})^T Z^{(2)} + b^{(3)} \\
y &= softmax\left(a^{(3)}\right)
\end{aligned}
$$

Now the gradient of the objective function has to be calculated with respect to all the different weights and biases, again using the chain rule.

$$
\begin{aligned}
\frac{\partial J}{\partial W^{(3)}_{m^{(2)}k}} &= \sum_{n=1}^{N}\sum_{k'=1}^{K} \frac{\partial J_{nk'}}{\partial y_{nk'}} \frac{\partial y_{nk'}}{\partial a^{(3)}_{nk}} \frac{\partial a^{(3)}_{nk}}{\partial W^{(3)}_{m^{(2)}k}} \\
\frac{\partial J}{\partial b^{(3)}_{k}} &= \sum_{n=1}^{N}\sum_{k'=1}^{K} \frac{\partial J_{nk'}}{\partial y_{nk'}} \frac{\partial y_{nk'}}{\partial a^{(3)}_{nk}} \frac{\partial a^{(3)}_{nk}}{\partial b^{(3)}_{k}}
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial J}{\partial W^{(2)}_{m^{(1)}m^{(2)}}} &= \sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{k'=1}^{K} \frac{\partial J_{nk'}}{\partial y_{nk'}} \frac{\partial y_{nk'}}{\partial a^{(3)}_{nk}} \frac{\partial a^{(3)}_{nk}}{\partial Z^{(2)}_{nm^{(2)}}} \frac{\partial Z^{(2)}_{nm^{(2)}}}{\partial a^{(2)}_{nm^{(2)}}} \frac{\partial a^{(2)}_{nm^{(2)}}}{\partial W^{(2)}_{m^{(1)}m^{(2)}}} \\
\frac{\partial J}{\partial b^{(2)}_{m^{(2)}}} &= \sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{k'=1}^{K} \frac{\partial J_{nk'}}{\partial y_{nk'}} \frac{\partial y_{nk'}}{\partial a^{(3)}_{nk}} \frac{\partial a^{(3)}_{nk}}{\partial Z^{(2)}_{nm^{(2)}}} \frac{\partial Z^{(2)}_{nm^{(2)}}}{\partial a^{(2)}_{nm^{(2)}}} \frac{\partial a^{(2)}_{nm^{(2)}}}{\partial b^{(2)}_{m^{(2)}}}
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial J}{\partial W^{(1)}_{dm^{(1)}}} &= \sum_{m^{(2)}=1}^{M^{(2)}}\sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{k'=1}^{K} \frac{\partial J_{nk'}}{\partial y_{nk'}} \frac{\partial y_{nk'}}{\partial a^{(3)}_{nk}} \frac{\partial a^{(3)}_{nk}}{\partial Z^{(2)}_{nm^{(2)}}} \frac{\partial Z^{(2)}_{nm^{(2)}}}{\partial a^{(2)}_{nm^{(2)}}} \frac{\partial a^{(2)}_{nm^{(2)}}}{\partial Z^{(1)}_{nm^{(1)}}} \frac{\partial Z^{(1)}_{nm^{(1)}}}{\partial a^{(1)}_{nm^{(1)}}} \frac{\partial a^{(1)}_{nm^{(1)}}}{\partial W^{(1)}_{dm^{(1)}}} \\
\frac{\partial J}{\partial b^{(1)}_{m^{(1)}}} &= \sum_{m^{(2)}=1}^{M^{(2)}}\sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{k'=1}^{K} \frac{\partial J_{nk'}}{\partial y_{nk'}} \frac{\partial y_{nk'}}{\partial a^{(3)}_{nk}} \frac{\partial a^{(3)}_{nk}}{\partial Z^{(2)}_{nm^{(2)}}} \frac{\partial Z^{(2)}_{nm^{(2)}}}{\partial a^{(2)}_{nm^{(2)}}} \frac{\partial a^{(2)}_{nm^{(2)}}}{\partial Z^{(1)}_{nm^{(1)}}} \frac{\partial Z^{(1)}_{nm^{(1)}}}{\partial a^{(1)}_{nm^{(1)}}} \frac{\partial a^{(1)}_{nm^{(1)}}}{\partial b^{(1)}_{m^{(1)}}}
\end{aligned}
$$

There are several things to notice in the above equations.

1. The law of total derivatives applies for each layer individually - for weights in the first layer the term inside the summation is enumerated over by 4 summations. A possible rule of thumb could be: every index that does not exist outside of the summations should be summed over.

2. Even though the resulting derivatives are not explicitly given anymore, all of these derivatives were already covered in some form or another, no new derivatives appear anymore

3. Erstwhile terms of gradients of weights in layers with lower indexes appear in all following terms - the gradient of the weights in the first layer could be expressed in terms of the gradients of the weights of the second layer, which in turn could be expressed in terms of the gradients of the weights of the third layer, etc.

To follow up on the last point of the above enumeration, the gradient $\frac{\partial J}{\partial W^{(3)}_{m^{(2)}k}}$ contains the term $\frac{\partial J_{nk'}}{\partial y_{nk'}} \frac{\partial y_{nk'}}{\partial a^{(3)}_{nk}}$, which is also contained in the gradients $\frac{\partial J}{\partial W^{(2)}_{m^{(1)}m^{(2)}}}$ and $\frac{\partial J}{\partial W^{(1)}_{dm^{(1)}}}$.

The gradient $\frac{\partial J}{\partial W^{(2)}_{m^{(1)}m^{(2)}}}$ contains the new term $\frac{\partial a^{(3)}_{nk}}{\partial Z^{(2)}_{nm^{(2)}}} \frac{\partial Z^{(2)}_{nm^{(2)}}}{\partial a^{(2)}_{nm^{(2)}}}$, which is also contained in the gradient $\frac{\partial J}{\partial W^{(1)}_{dm^{(1)}}}$.

The next subsection will factorize the pattern and will present backpropagation in its most general form.

7.3.3 Backpropagation in its Commonly Known Form

Factorizing the Pattern

For even deeper networks this pattern would continue. For every layer the shared terms expand by a new term which contains one multiplication of the partial derivative of a function a by a function z of a layer with the index of a minus one, and the partial derivative of this function z by a function a of the same layer as z.

The functions involved are now going to be expressed in their most general form.

$$a^{(L)} = (W^{(L)})^T Z^{(L-1)} + b^{(L)}$$
$$Z^{(L)} = \sigma(a^{(L)})$$

For every gradient there is a term that is not contained in any other gradient, which is the final partial derivative of a function a of the layer where the weight, which the derivative is for, is in.

$$\frac{\partial a^{(l)}_{nm^{(l)}}}{\partial W^{(l)}_{m^{(l=1)}m^{(l)}}} = Z^{(l-1)}_{nm^{(l-1)}}$$

This derivative is always going to be the input of the function a. The following list sums up the patterns the chain rule of the gradient of J follows.

1. Every gradient $\frac{\partial J}{\partial W^{(l)}_{m^{(l-1)}m^{(l)}}}$ expands the gradient $\frac{\partial J}{\partial W^{(l+1)}_{m^{(l)}m^{(l+1)}}}$ by the term $\frac{\partial a^{(l+1)}{}_{nm^{(l+1)}}}{Z^{(l)}_{nm^{(l)}}} \frac{Z^{(l)}_{nm^{(l)}}}{a^{(l)}_{nm^{(l)}}}$
2. For the gradient $\frac{\partial a^{(l)}_{nm^{(l)}}}{\partial W^{(l)}_{m^{(l-1)}m^{(l)}}}$ the distinguishing final derivative is always going to be the input of the function a

3. For the gradient $\frac{\partial a^{(l)}_{nm^{(l)}}}{\partial b^{(l)}_{m^{(l)}}}$ the distinguishing final derivative is always going to be 1.

Finally, all of the factorizations can be put into one simple recursive formula for the gradient of the objective function J.

$$\nabla_{W^{(l)}} J = (Z^{(l-1)})^T \delta^{(l)} \tag{7.12}$$

The equation 7.12 introduces the *delta* function $\delta^{(l)}$. The recursive definition of this function is:

$$\delta^{(l)}_{nm^{(l)}} = \sum_{m^{(l+1)}=1}^{M^{(l+1)}} \delta^{(l+1)}_{nm^{l+1}} W^{(l+1)}_{m^{(l)}m^{(l+1)}} (Z^{(l)}_{nm^{(l)}})' \tag{7.13}$$

where the $'$ in $(Z^{(l)}_{nm^{(l)}})'$ indicates the derivation of $Z^{(l)}_{nm^{(l)}}$, and

$$W^{(l+1)}_{m^{(l)}m^{(l+1)}} (Z^{(l)}_{nm^{(l)}})'$$

is the derivative of

$$\frac{\partial a^{(l+1)}{}_{nm^{(l+1)}}}{Z^{(l)}_{nm^{(l)}}} \frac{Z^{(l)}_{nm^{(l)}}}{a^{(l)}_{nm^{(l)}}}$$

The only missing part of this equation is the base case of the recursive function δ. The base case is $\delta^{(L)}$, where L is the the number of layers, so $\delta^{(L)}$ is the output layer. If the objective function is the categorical cross entropy function, then the base case of δ is:

$$\delta^{(L)} = \sum_{k'=1}^{K} \frac{\partial J_{nk'}}{\partial y_{nk'}} \frac{\partial y_{nk'}}{\partial a_{nk}} = t_{nk} - y_{nk} \tag{7.14}$$

To conclude this section the formula of the general gradient of J with respect to the bias terms is shown in vectorized form.

$$\nabla_{b^{(l)}} J = (\delta^{(L)})^T 1_N \tag{7.15}$$

where 1_N is a vector with N rows.

7.3.4 Implementation with Keras

The code to implement a densely connected neural network with 2 hidden layers is in listing 7.1. To add more hidden layers, dense layers are simply added to the constructor array of the sequential model.

Listing 7.1: Code to implement a neural network in Keras

```
100 import tensorflow.keras as keras
101
102 model = keras.models.Sequential([
```

```
103     keras.layers.Input(D),
104     keras.layers.Dense(M, activation=keras.activations.sigmoid),
105     keras.layers.Dense(M_2, activation=keras.activations.tanh),
106     keras.layers.Dense(K, activation=keras.activations.softmax)
107 ])
108
109 model.compile(optimizer=keras.optimizers.Adam(
110                 learning_rate=learning_rate),
111                 loss="categorical_crossentropy"
112 )
113 model.fit(X_train, Y_train, epochs=epochs,
114           validation_data=(X_test, Y_test))
```

The above listing creates a classification model. To let the model do, the activation parameter has to be removed from the last layer in the constructor of the sequential model.

7.4 Conclusion

This chapter covers the two most important aspects of any machine learning algorithm, which are:

1. How predictions are made
2. How the training process works

It introduced the cross-entropy-error objective function and explained its statistical foundations, which is necessary for making predictions in an environment where more than one class for classification is needed.

Backpropagation was explained in high detail, it is a conceptually challenging approach of training feedforward neural networks. All the steps leading up to factorizing the δ function were necessary, because the concepts are easier to grasp when approaching the topic by slowly making the neural networks more complex. Unlike many other resources backpropagation was explained in an inverted manner, as often explanations start with the delta function and then explain where the delta function comes from and how it works.

The approach that was used to explain backpropagation in this chapter, is to give the reader the possibility to see the patterns for themselves and to allow them to slowly build a fundamental understanding of the concepts themselves.

Chapter 8

Sequence Analysis

Traditional feedforward neural networks can accomplish almost any task, however there are types of neural networks that excel for certain tasks, in terms of performance and results. These special types include *convolutional* neural networks (CNNs, networks designed for working with images), and *recurrent* neural networks (RNNs). This chapter focuses on recurrent neural networks, which are neural networks built to work with sequential data. In general, traditional feedforward networks can accomplish this task as well, however training lasts a multitude longer to achieve the same results as recurrent neural networks. This chapter explains the general principles of the three most important types of RNNs and explains how making predictions works for each type. Unlike previous theoretical chapters, this chapter will not explain in detail how finding the gradients with respect to some error functions works, because the same basic principles of backpropagation (explained in chapter 7) also apply to RNNs. The resource for the information presented in this chapter is [4].

8.1 Recurrent Neural Networks in General

Previously in this thesis the data fed to feedforward neural networks had the dimensions $N \times D$. This chapter introduces a new kind of data, *sequential data*. Sequences add another dimension to the input tensors, they have the dimensionality $N \times T \times D$, with the newly introduced sequence length T. In many cases T refers to a time frame, and all t's in T are time steps that depend on each other. For each time step there can be D multiple captured values, which explains why the data is now of shape $N \times T \times D$.

In many cases traditional feedforward neural networks cannot achieve the same results as RNNs. One of the key concepts of RNNs is *parameter sharing*. FFNNs would have to flatten the data so that it would be of size $N \times T \cdot D$, where each input has its own weights to each hidden unit. RNNs also have weights for each input d in D, however they share a weight-matrix W_h for each time step t in T. This not only allows them to generalize for sequences of variable length T, but also enables them to extract *information* that can occur at any step in a sequence. To illustrate this power, the reader is encouraged to think about language processing - grammar rules for example

apply regardless of the position in the sentence. FFNNs could not generalize rules like grammar when they are not explicitly given training samples of the rules appearing in all possible steps of sequences.

8.2 The Simple Recurrent Unit

As the name suggests, the simple recurrent unit is the building block of the simplest recurrent neural network. A recurrent unit is part of a recurrent layer.

8.2.1 Architecture

(a) Traditional Feedforward Neural Networks

(b) Simple Recurrent Unit

Figure 8.1: A comparison of the architectures of feedforward neural networks to recurrent neural networks

In feedforward neural networks, the input is passed to the hidden layer, and the output of the hidden layer is passed to the output layer. This is visually explained in figure 8.1a. The simple recurrent layer extends this with a connection from the hidden layer to the hidden layer. The output of the hidden layer at the time step t ($h(t)$) depends on the output of the recurrent layer from $t-1$. The hidden layer is *fully connected* to itself, this means that every unit of the hidden layer is connected to every other hidden unit. If the hidden layer has M hidden units, then the size of the weight matrix W_h is $M \times M$.

Definition 17. The mathematical definition for making predictions (conceptually explained in 8.1b) is:

$$h(t) = f(W_x \cdot x(t) + W_h \cdot h(t-1) + b_h) \tag{8.1}$$

with f being a nonlinear activation function like the sigmoid function, tanh or relu.

Remark 6. The initial value for h, $h_0 = h(0)$ can either be a trainable parameter or can be set to a fixed value (often 0). The output a hidden recurrent layer actually varies whether it outputs to another recurrent layer or to a dense layer. h outputs values in a tensor of the shape $N \times T \times M$ to other recurrent layers, for every t in T the hidden recurrent layer outputs $h(t)$. To non-recurrent layers the output is of shape $N \times M$. The output is $h(t)$ where t is the last value of the sequence.

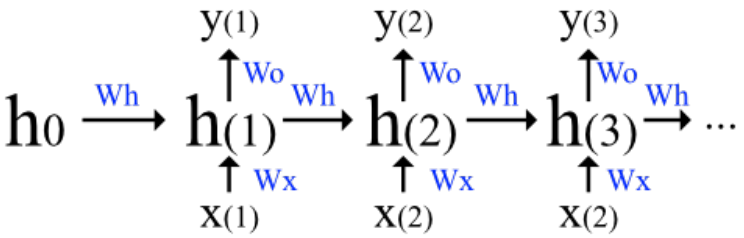


Figure 8.2: A figure illustrating the concept of parameter sharing by unfolding a recurrent unit over T

8.2.2 Conceptual Backpropagation

Although the calculation of the gradients with respect to the shared weights will not be shown, the concept of backpropagation *through time* will be explained.

What makes backpropagation with shared parameters more complicated is that the same weights are in every layer of the gradient. This implies, that due to the chain rule, the same values are multiplied with themselves repeatedly. This is problematic - for long sequences this means that the gradient either asymptotically approaches 0 or exponentially nears infinity. These problems are known as the *vanishing gradient problem* and *exploding gradient problem* respectively.

The solution to this problem are more complex recurrent units, as discussed in the following sections.

8.3 GRU - The Gated Recurrent Unit

Even though the GRU was invented after the LSTM unit, in this thesis it is listed before the LSTM unit, because it is a less complex version of the LSTM unit with the same concept. The GRU has less parameters than the LSTM unit, training is therefore faster.

The GRU tries to counter the vanishing gradient problem with a *gate* that *acts* on small values. This gate modifies the output of h to itself if h is small.

8.3.1 Architecture

In the GRU this gate is called the *reset gate* containing the activation functions r and Z. The architecture is best explained by its mathematical description.

Definition 18. The output of the hidden unit to itself in the GRU is:

$$h(t) = h_{t-1} \odot (1 - z_t) + \hat{h_t} \odot z_t \tag{8.2}$$

Definition 19. The modified output $\hat{h}$ such as the reset gate:

$$r_t = \sigma(x_t \cdot W_{xr} + h_{t-1} \cdot W_{hr} + b_r) \tag{8.3}$$

$$\hat{h}_t = g(x_t \cdot W_{hx} + (r_t \odot h_{t-1}) \cdot W_{hh} + b_h) \tag{8.4}$$

$$z_t = \sigma(x_t \cdot W_{xz} + h_{t-1} \cdot W_{hz} + b_z) \tag{8.5}$$

Remark 7. A closer look at the reset gate explains why it is called the *reset* gate. Both r and z contain the new input x_t such as the old output h_{t-1}. If the old output of h is very small, this then is as if a new sequence would be starting.

8.4 LSTM - Long Short Term Memory Neural Networks

The LSTM unit has a similar concept to the GRU in trying to counter the vanishing gradient problem. Instead of the reset gate it has three different gates instead:

- Forget Gate f: determines how much the previous cell value affects the current memory cell
- Output Gate o
- Input Gate i: determines how much the new h affects the memory cell

And instead of $\hat{h}$ it uses a memory cell c.

For the sake of completeness the functions responsible for making predictions in a LSTM unit will be listed.

Definition 20. Like in the GRU, the architecture is best described by its mathematical form.

$$i_t = \sigma(x_t \cdot W_{xi} + h_{t-1} \cdot W_{hi} + c_{t-1} \cdot W_{ci} + b_i) \tag{8.6}$$
$$f_t = \sigma(x_t \cdot W_{xf} + h_{t-1} \cdot W_{hf} + c_{t-1} \cdot W_{cf} + b_f) \tag{8.7}$$
$$c_t = f_t \cdot c_{t-1} + i_t \cdot \tanh(x_t \cdot W_{xc} + h_{t-1} \cdot W_{hc} + b_c) \tag{8.8}$$
$$o_t = \sigma(x_t \cdot W_{xo} + h_{t-1} \cdot W_{ho} + c_t \cdot W_{co} + b_o) \tag{8.9}$$
$$h_t = o_t \cdot \tanh(c_t) \tag{8.10}$$

In more pragmatic terms, the increased amount of parameters in the LSTM units and GRU make the models more *expressive*. This gives the models a better ability to adapt to learned data and to learn abstract and complex patterns and coherences.

8.5 Implementation with Keras

The code to implement a recurrent neural network with 1 hidden layer is shown in listing 8.1.

Listing 8.1: Code to implement a recurrent neural network in Keras

```
100 import tensorflow.keras as keras
101
102 model = keras.models.Sequential([
103     keras.layers.Input(shape=(T, D)),
104     keras.layers.LSTM(M, activation=keras.activations.sigmoid),
```

```
    keras.layers.Dense(K, activation=keras.activations.softmax)
])

model.compile(optimizer=keras.optimizers.Adam(
                learning_rate=learning_rate),
                loss="categorical_crossentropy"
)
model.fit(X_train, Y_train, epochs=epochs,
            validation_data=(X_test, Y_test))
```

In order to add multiple recurrent layers in a model, the respective previous recurrent layer has to return a sequence of predictions for each time step. (See listing 8.2)

Listing 8.2: Code to implement a recurrent neural network with multiple hidden layers in Keras

```
model = keras.models.Sequential([
        keras.layers.Input(shape=(T, D)),
        keras.layers.LSTM(M, return_sequences=True),
        keras.layers.GRU(M_2, return_sequences=True),
        keras.layers.SimpleRNN(M_3),
        keras.layers.Dense(K, activation='sigmoid')
    ])
```

Part II

Implementation

Chapter 9

Implementing a K-Means Clustering Algorithm and a Neural Network

This chapter will examine and explain the implementation of a k-Means clustering algorithm in more detail. The overall functionality of this algorithm has already been described in section 6.3.1. Also the implementation of a neural network with the *keras* library is explained further. The functionality of a simple neural network is treated in more detail here 7.

9.1 General Idea

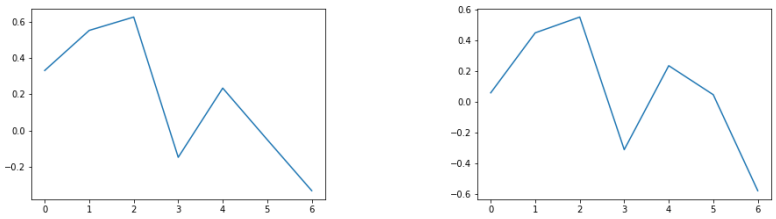
The first, initial idea was to use an unsupervised machine learning algorithm, in this case the k-Means algorithm, to find similar patterns from given data. This data included currency pairs, indices, stocks and crypto currencies. These financial instruments just mentioned have already been introduced in chapter 2.

A pattern is simply a sequence of prices with a certain length that needs to be defined. In the course of the project this length is defined with *patternSize*. For example, if the size just mentioned is equal to five, each pattern will consist of five consecutive prices.

The goal was to end up with a number of clusters where each cluster would contain similar patterns.

The next step was to feed these found patterns with their corresponding labels for the clusters to a neural network for training purposes. The neural network is a supervised process, because it has all the prices of the respective patterns and the cluster in which each pattern belongs, so that it is able to learn from this correlation. The neural network should be able to determine which pattern is most likely to be formed and not which pattern is currently present, since this would be of no use to an analyst, when the neural network receives current market data.

Therefore we tried to give the neural network only a percentage of the prices for each



(a) First pattern (b) Second pattern

Figure 9.1: Similar patterns

pattern for training. In other words, if the *patternSize* is ten, this would mean that each pattern consists of ten prices. The neural network, on the other hand, would not be given all these ten prices, but only a certain percentage, for example six prices. This way the neural network should be able to decide, when it retrieves current market data for the last six prices, that the pattern of cluster X is most likely to be formed with a percentage of Y. Not only the cluster with the highest probability, but also those with the following higher probabilities could also be listed.

9.2 Clustering

In the first cluster approach, the shares *AAPl, GOOGL, MSFT* and the currency pairs *EUR-USD, GBP-USD, USD-CHF* were used as data. The goal, as mentioned above, was to end up with n clusters with similar patterns in it. Such similar patterns can be seen in figure 9.1. The list below includes the required steps to accomplish the cluster process.

1. Read data from csv files
2. Create patterns
3. Calculation of differences between consecutive prices
4. Create k-Means model
5. Delete directory with the patterns
6. Plot and save the patterns as images
7. Add cluster labels to each pattern
8. Save dataset as csv file

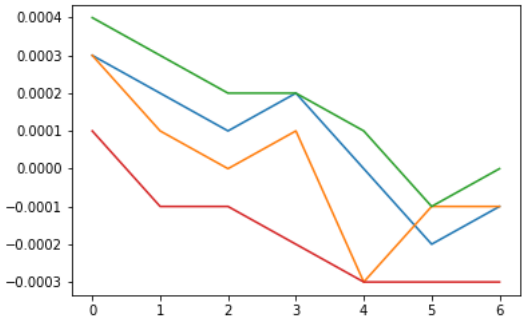


Figure 9.2: Pattern with open, close, high, low price

9.2.1 Code Explanation

Read Csv Files

To read a csv file, the library *pandas* offers the *read_csv* method where the path of the file must be passed as an argument and as the result a dataframe is returned. Because the data is not stored in an overall csv file, the individual files have to be read in separately. In order to avoid having an arbitrary number of dataframes at the end, the *append* method can be used to append one dataframe to another. It simply appends the rows from the dataframe that was passed in as an argument to the dataframe that called the method. It is important to mention that in the *append* method the *ignore_index* value must be set to *true* in order to prevent the index from starting at 1 for each dataframe that was appended.

Since the data was read in the wrong chronological order, it must be reversed. The reason for this is that the most recent market price in the csv file is located at the beginning, implying that it is indexed with 0 at the time of reading the file. However the latest price should be at index 0 in order not to counterfeit the original market graph. The function simply reverses the series of prices.

The reason why only the *open* price is considered is the following. If all four prices are used for the cluster process, each pattern would consist of four different graphs, i.e. one for the *open*, *high*, *low* and *close* price as can be seen in figure 9.2. The probability to find similar patterns with four individual graphs is very low, therefore this approach to take all prices makes no sense. It makes no real difference what price is taken from the four.

```
100     def reverse_prices(series):
101         index = 0
102         reversed_series = pd.Series()
103         for value in reversed(series):
104             reversed_series.at[index] = value
105             index += 1
```

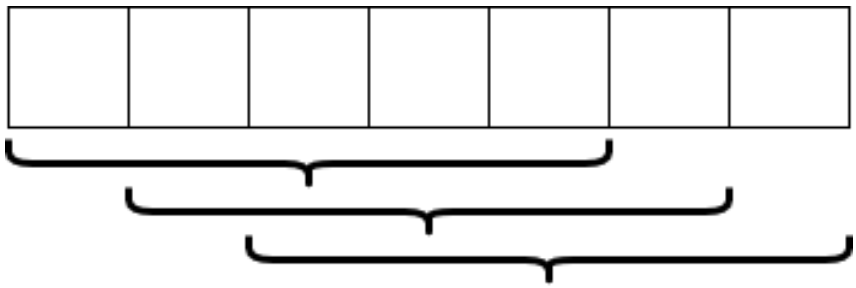


Figure 9.3: Patterns

```
106         return reversed_series
```

```
100     def read_csv_files():
101         data = pd.read_csv('csvData/EUR_USD.csv')
102         data = data.append(pd.read_csv('csvData/GBP_USD.csv'),
    ↪ ignore_index=True)
103         data = data.append(pd.read_csv('csvData/USD_CHF.csv'),
    ↪ ignore_index=True)
104         data = data.append(pd.read_csv('csvData/AAPL.csv'), ignore_index
    ↪ =True)
105         data = data.append(pd.read_csv('csvData/GOOGL.csv'),
    ↪ ignore_index=True)
106         data = data.append(pd.read_csv('csvData/MSFT.csv'), ignore_index
    ↪ =True)
107         openPricesSeries = reverse_prices(data['open'])
108         return openPricesSeries
```

Create Patterns

The purpose of this function is to create patterns. A pattern is basically a series of prices with a specified length. For a better comprehension, the intention of this method is illustrated in figure 9.3. In this illustration it can be observed that for each pattern a particular amount of prices is taken, depending on how large the *patternSize* is defined, in this case it is 5.

To accomplish this task it requires a total of two loops, an outer one that iterates over the entire prices series and a inner one that is responsible for the creation of each pattern. At the end a list with all the patterns that have been created is returned.

```
100     def create_patterns(openPricesSeries, patternSize):
101         resultList = []
102         for i in range(0, len(openPricesSeries) - patternSize):
103             pattern = []
```

```
104             for j in range(i, i + patternSize):
105                     pattern.append(openPricesSeries[j])
106             resultList.append(pattern)
107         return resultList
```

Calculate Differences

Here, for each price of the pattern the difference to the following price is calculated in order to have one less feature for the k-Means model. Each *prices* list in the list with all patterns that was passed as an argument is sent to the following method which is responsible for the delta calculation.

```
100     def calc_difference_between(pattern):
101         deltaPattern = []
102         for i in range(0, len(pattern) - 1):
103             delta = pattern[i + 1] - pattern[i]
104             deltaPattern.append(delta)
105         return deltaPattern
```

Then the new differences are appended to the result.

```
100     def calc_differences(patternList):
101         resultList = []
102         for pattern in patternList:
103             deltaPrices = calc_difference_between(pattern)
104             resultList.append(deltaPrices)
105         return resultList
```

Normalize

In order to bring the data to the same value scale, the data must be normalized and that is accomplished with the *normalize* function. It just takes the data which has to be normalized as an argument.

```
100     deltaPatternListNormalized = normalize(deltaPatternList)
```

Create k-Means Model

To implement a k-Means clustering algorithm in Python, only very few instructions are needed. The method responsible for this task receives the data that should be clustered as a parameter, in this case the price differences. The second parameter is the amount of desired clusters.

Now to create a k-Means model, the method *KMeans* from the *sklearn* library needs to be called with the amount of clusters as an argument. Furthermore, the *fit* method

must be invoked on the model that was just created and the data which should be clustered must be passed in. The model is now complete and can be used. With the method *labels_* the assigned clusters can be queried for each pattern and with *cluster_centres_* the cluster centres can be retrieved. In the end the model is returned.

```
100     def createAndFitKMeans(data, clusters):
101         model = KMeans(n_clusters=clusters)
102         model = model.fit(data)
103         return model
```

Delete Directory With the Patterns

Since the clusters with the corresponding patterns will be saved as images for verification, the directory must be deleted first and this is the purpose of this method. It simply takes the directory name which should be deleted as an argument. It will then use the *os.path.exists* function to check whether the passed path even exists. If that is the case the directory is deleted with *shutil.rmtree*.

```
100     def delete_directory(directory_to_delete):
101         if os.path.exists(directory_to_delete):
102             shutil.rmtree(directory_to_delete)
```

Plot and Save the Patterns as Images

In order to visualize the clusters with the corresponding patterns, they are stored as images in the subfolder that was passed in as an argument in order to be able to make a general verification of how well the clustering has worked.

With the *matplotlib* library it is possible to plot a graph using the *plot* method. As arguments it is necessary firstly to pass the values for the x-axis, in this case simply the length of the *patternSize* starting at 0 and secondly the values for the y-axis, here the prices.

To save the graph as image, the library offers the *savefig* method, which receives the path where the image should be saved as a parameter. The function *clf* just clears the current figure in order to plot a new one.

```
100     def plotAndSavePatterns(labels, data, subfolder):
101         for i in range(len(labels)):
102             prices = data[i]
103             plt.plot(range(len(deltas)), deltas)
104             try:
105                 os.makedirs('Patterns/' + subfolder + '/cluster' + str(
    ↪ labels[i]))
106             except FileExistsError:
107                 pass
```

```
108             plt.savefig('Patterns/' + subfolder + '/cluster'
109                         + str(labels[i]) + '/pattern_'
110                         + str(i) + '.png')
111             plt.clf()
```

Add Cluster Labels to Each Pattern

To enable the neural network to use the data, each pattern record must contain the corresponding cluster and that is accomplished within this method.

To combine the array of prices and the cluster, both are converted into lists. To convert an array into a list, the method *tolist* is invoked for the desired array. To convert a single value to a list, just enclose the value in two square brackets. These lists can then be combined with a plus sign.

```
100     def add_labels_to_data(data, labels):
101         result = []
102         for i in range(0, len(data)):
103             prices = data[i].tolist()
104             prices_labeled = [labels[i]] + prices
105             result.append(prices_labeled)
106         return result
```

Save Dataset as Csv File

Finally, the data must be saved as csv file. Labelled data and the name of the file are passed as parameters. With the use of the *csv* library the *csv.writer* method returns a writer object which converts the data into delimited strings. The *writerow* method writes all given rows into the specified file.

```
100     def save_data_as_csv(data, name):
101         with open(name, 'w', newline='') as csvfile:
102             writer = csv.writer(csvfile)
103             writer.writerows(data)
```

9.2.2 Results

The problem is that clustering worked quite well under the condition that the *patternSize* was rather low, more precisely if it was higher than about 6, clustering tended to not work as it should.

The cause for this is that with for example 9 prices per pattern there are simply too many different pattern types that can occur. Figure 9.4 shows a segment of a cluster where the *patternSize* is 6. It can be seen that the result is quite good in general. In contrast to figure 9.5, where a pattern size of 9 has been used, the results are rather bad.

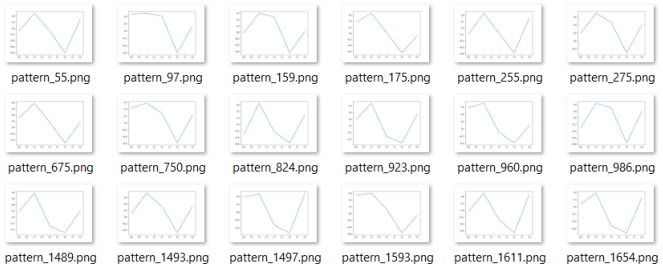


Figure 9.4: Good cluster

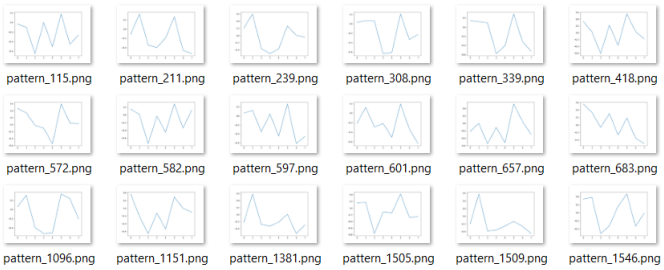


Figure 9.5: Bad cluster

The reason why it is a problem if the *patternSize* is too small is the following. As already mentioned in section 9.1, not all prices per pattern are used to train the neural network, only a certain percentage in order to accomplish a predictions and not a classification. The neural network should not decide to which cluster a pattern belongs, it should rather predict which cluster is most likely to form from the given pattern.

If the *patternSize* is to small, meaning that each pattern consists of only a few prices, almost all prices have to be fed to the neural network for training in order to achieve a fairly good accuracy. Because the neural network is not able to learn a proper correlation between patterns with very few provided prices.

So the major problem is now, if the *patternSize* is too small, not enough prices can be provided to the neural network in order to train it properly. On the contrary, if the *patternSize* is to large, clustering will not work as it should, as shown in figure 9.5.

Because of this problem we had the idea of the *2 stage clustering* which is explained in the next section.

9.3 2 Stage Clustering

In the second attempt the DAX index was used as data. Two different csv files were used, one with a time interval of 5 minutes and the other one with a time interval of 15 minutes.

startTimestamp	endTimestamp
2019/08/15 09:15:00	2019/08/15 10:15:00

(a) 15min timestamp

startTimestamp	endTimestamp
2019/08/14 10:15:00	2019/08/14 10:35:00

(b) 5min timestamp

Figure 9.6: Matching timestamps

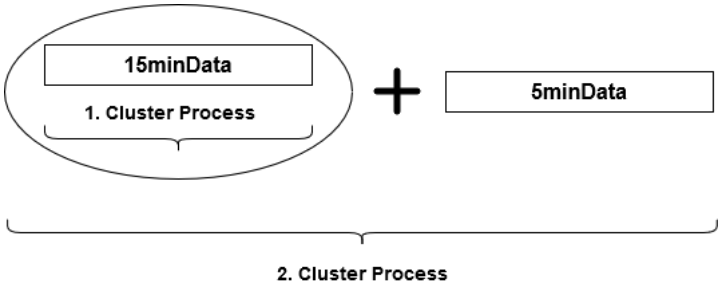


Figure 9.7: 2 stage clustering

The idea with the *2 stage clustering* was not only to cluster the data once, but also a second time to get a larger time span. First the patterns with the 15 minute intervals are clustered. Then for each 15 minute pattern the subsequent 5 minute pattern is searched. In this context, subsequent means that the end time of the pattern with the 15 minute interval is equal to the start time of the 5 minute interval. Such a match can be viewed in figure 9.6. The cluster of the 15 minute pattern is then combined with the prices of the following 5 minute pattern. This combined dataset is clustered again afterwards. An overview of this procedure is shown in figure 9.7.

The following list contains the additional steps that needs to be done compared with 9.2 to achieve the *2 stage clustering*.

1. Remove NaN values from the dataset
2. Determining the time gaps in the dataset
3. Create patterns with correct timestamps
4. Establishing the average error of the model
5. Create combined dataset

9.3.1 Code Explanation

Read Data from Csv Files

As a result of reading the csv files, two dataframes are returned, each for the 5 and 15 minute intervals as shown in Figure 9.8. Furthermore only the *open* price and the

timestamp	open	high	low	close
2019/08/14 10:15:00	11725	11725.2	11710.7	11716.7
2019/08/14 10:20:00	11717	11726.7	11713.2	11722.7
2019/08/14 10:25:00	11723	11736.5	11710.7	11733.7
2019/08/14 10:30:00	11734	11743.7	11723.7	11723.7
2019/08/14 10:35:00	11723.5	11726	11707.5	11708.2
2019/08/14 10:40:00	11707.7	11713	11685.2	11685.2

(a) 5min Interval

timestamp	open	high	low	close
2019/06/26 10:30:00	12216.2	12234	12213	12231.2
2019/06/26 10:45:00	12231	12239.5	12230.7	12234.2
2019/06/26 11:00:00	12234.5	12291.5	12229.2	12287.2
2019/06/26 11:15:00	12287.5	12312.2	12283.2	12302.7
2019/06/26 11:30:00	12303.2	12309	12293.5	12304.2
2019/06/26 11:45:00	12304.5	12309.2	12294.5	12298

(b) 15min Interval

Figure 9.8: Data

timestamp are needed, therefore these columns are extracted from the dataframe.

```
100     gap15minData = pd.read_csv('csvData\dax_15_4999.csv')
101     gap5minData = pd.read_csv('csvData\dax_5_4999.csv')
102     gap15minData = gap15minData[['timestamp', 'open']]
103     gap5minData = gap5minData[['timestamp', 'open']]
```

Remove NaN Values from the Dataset

For unknown reasons, a few open prices contain the value NaN, and therefore these values must be removed from the dataframe. *NaN* means "not a number" and in pandas missing values are marked with it.

To remove these values the following method is defined. It takes a dataframe as an argument and then iterates over every price of it and compares if the value is NaN, using the *isnan* method from the *math* library. If that is the case the value is removed. To actually edit the original data structure that was passed in as an argument, the *inplace* parameter is set to True. Because elements are removed, the index is no longer correct as it contains gaps which should be avoided. To counteract this, the method *reset_index* is called which simply recreates the indexes. The *drop* parameter is also worth mentioning, because it makes sure that the old index column with the gaps is deleted from the dataframe.

```
100     def removeNanFormDataFrame(df):
101         for i in range(len(df)):
102             if math.isnan(df['open'][i]):
103                 df.drop(index=i, inplace=True)
104         df.reset_index(inplace=True, drop=True)
```

Determining the Time Gaps in the Dataset

Sometimes the interval between two timestamps is too long, so these timestamps must be located to avoid creating patterns with incorrect time intervals afterwards. This is the task of this method.

2019/06/28 22:45:00	12445.8
2019/07/01 00:00:00	12526.9

Figure 9.9: Time gap

Such a incorrect interval can be observed in figure 9.9. As parameters a series of timestamps and the time interval in seconds is passed, either 900 seconds for the 15 minute or 300 seconds for the 5 minute intervals. In case of incorrect time intervals the index of this timestamp is stored in the *gapIndexes* array which is returned at the end.

The timestamp series is iterated and the respective one with the following timestamp is split in order to just retrieve the time and not the date. To be able to calculate with this time the string has to be converted into a datetime object, using the *datetime.datetime.strptime* function. After that the delta seconds between these two timestamps is calculated.

There is a problem that can occur with the calculation of this delta seconds, which is when the second timestamp is midnight. Because then the calculation results in a high negative amount, although the timestamp interval would be correct, and the following if statement prevents the calculation from being performed when the second time is midnight.

```
100     if not ((date_time_str1 == '23:55:00' or date_time_str1 == '23:45:00
    ↪ ') and (date_time_str2 == '00:00:00')):
```

If the delta seconds do not match the correct time interval that was passed in, the index of the incorrect timestamp is added to the array that is returned at the end.

```
100     def getGapIndexes(timestamps, seconds):
101         gapIndexes = []
102         for i in range(len(timestamps) - 1):
103             split = timestamps[i].split()
104             date_time_str1 = split[1]
105             date_time_obj1 = datetime.datetime.strptime(date_time_str1,
    ↪ '%H:%M:%S')
106             split = timestamps[i + 1].split()
107             date_time_str2 = split[1]
108             date_time_obj2 = datetime.datetime.strptime(date_time_str2,
    ↪ '%H:%M:%S')
109             if not ((date_time_str1 == '23:55:00' or date_time_str1 == '
    ↪ 23:45:00') and (date_time_str2 == '00:00:00')):
110                 delta = date_time_obj2 - date_time_obj1
111                 delta_seconds = delta.total_seconds()
112                 if delta_seconds != seconds:
113                     gapIndexes.append(i)
```

```
114         return gapIndexes
```

Create Patterns with Correct Timestamps

The process of creating the patterns was already covered here 9.2.1 but there are a few extra steps that need to be added.

First, the start and end time of each pattern is additionally needed. To avoid creating a pattern with a wrong time interval, it is necessary to check whether the current index is included in the *gaps* array which contains all indexes of the incorrect timestamps. If that is the case, the bool variable *indexInGap* is set to true and the loop is left afterwards. After the inner loop each pattern is added to the result, under the condition that the bool variable is false to prevent wrong patterns with incorrect time intervals from being added. In order to create a dataframe object from a list the *pd.Dataframe* method is used where simply the list and the column names are passed. At the end the dataframe with the patterns is returned.

```
100     def create_patterns(df, patternSize, gaps):
101         patterns = []
102         for i in range(len(df) - patternSize):
103             startTimestamp = df['timestamp'][i]
104             endTimestamp = df['timestamp'][i + patternSize - 1]
105             prices = []
106             indexInGap = False
107             for j in range(patternSize):
108                 if (i + j) in gaps:
109                     indexInGap = True; break
110                 else:
111                     prices.append(df['open'][i + j])
112             if indexInGap == False:
113                 patterns.append([startTimestamp, endTimestamp, prices])
114         return pd.DataFrame(patterns, columns=['startTimestamp', '
    ↪ endTimestamp', 'prices'])
```

Calculation of Differences Between Consecutive Prices

The only difference to 9.2.1 is that here not only the prices but also the end and start time are needed.

After the calculation the new differences are appended to the result, including the start and end timestamp for the given prices. Again, a dataframe containing the differences and the start and end time is returned.

```
100     def createDataFrameWithDeltaPatterns(df):
101         setsWithDeltaPrices = []
102         for i in range(len(df)):
```

```
            deltaPrices = calculateDeltaForPattern(df['prices'][i])
            setsWithDeltaPrices.append([df['startTimestamp'][i],
                                        df['endTimestamp'][i],
                                        deltaPrices])
        return pd.DataFrame(setsWithDeltaPrices, columns=['
    ↪ startTimestamp', 'endTimestamp', 'prices'])
```

Establishing the Average Error of the Model

This method calculates the average error. Error in this context refers to the deviation from each pattern to the respective cluster center.

For each label of a pattern the corresponding cluster center is determined and then the difference between the pattern prices and the center is calculated. The second part of the method iterates over all error lists and sums each list to a total number. In the third part, each of these numbers is summed to a single number and then it is divided by the number of patterns to get an average error.

The purpose of this method is to be able to compare different models and decide on the basis of the error which one is better or worse.

```
    def calculateAverageError(dataSet, cluster_labels, cluster_centres):
        errors = []
        for i in range(len(dataSet)):
            data = dataSet[i]
            cluster_label = cluster_labels[i]
            cluster_center = []
            for j in range(len(cluster_centres)):
                if cluster_label == j:
                    cluster_center = cluster_centres[j]
            error = abs(cluster_center - data)
            errors.append(error)

        summedErrors = []
        for i in range(len(errors)):
            data = errors[i]
            errorSum = 0
            for j in range(len(data)):
                errorSum += data[j]
            summedErrors.append(errorSum)

        sum = 0
        for i in range(len(summedErrors)):
            sum += summedErrors[i]
        averageError = sum / len(summedErrors)
        return averageError
```

Create Combined Dataset

Before this function can be used, a k-Means model must be created with the 15 minute interval data, as described in section 9.2.1. This method is responsible for combining the 15 minutes with the 5 minutes pattern as described in the beginning of section 9.3.

The parameters are the k-Means model for the 15 minute patterns, as well as the data frames for the 5 and 15 minute intervals patterns. The method used for finding the matching 5 minute pattern for the 15 minute pattern receives as parameter the end time of the 15 minutes pattern and the whole data frame of the 5 minute patterns. Thanks to the library *bisect* it is easy to perform a binary search with just a few lines of code which will then return the pattern according to the previously mentioned criteria.

```
100     def getCorresponding5Pattern(endTimestamp15, data5):
101         i = bisect_left(deltaData5.startTimestamp, endTimestamp15)
102         if i != len(deltaData5.startTimestamp) and deltaData5.
    ↪ startTimestamp[i] == endTimestamp15:
103             return deltaData5.prices[i]
```

If a matching 5 minute pattern has been found, the cluster of the 15 minute pattern is also required in order to merge these two elements into one data record.

Such a record now consists of the 15 minute pattern cluster and the prices of the following 5 minute pattern. It is added to the result data structure which will be returned at the end. With such records the second k-Means model has to be created.

```
100     def createCombinedData(delta15, delta5, model15):
101         data = []
102         for i in range(len(delta15)):
103             prices = delta15.prices[i]
104             corresponding5Pattern = getCorresponding5Pattern(
105                     delta15.endTimestamp[i], delta5)
106             if corresponding5Pattern is not None:
107                 cluster15 = model15.predict([prices])[0]
108                 cluster15WithCorresponding5Pattern = [cluster15] +
    ↪ corresponding5Pattern
109                 data.append(cluster15WithCorresponding5Pattern)
110         return data
```

After this process, the following functions that have already been defined must be called.

1. 9.2.1 Delete directory with the patterns
2. 9.2.1 Plot and save the patterns as images

3. 9.2.1 Add cluster labels to each pattern
4. 9.2.1 Save dataset as csv file

9.3.2 Problems That Occurred

Data

A big problem was the data, because not enough was available. It is very unlikely to encounter for example DAX market data on the internet with a considerable time in history for free.

Therefore we tried to get such data via an API on platforms like *Alphavantage* or *IG*. Unfortunately, these platforms only allowed limited access to this data without having to pay for it. We also tried to write a script that generates a new API key each time to retrieve data, but that did not work either. We even contacted *IG* with an email to see if we could get more access to the data, since it was a school project, we argued. This request was ignored several times and finally rejected.

Functionality

Another problem that appeared was that clustering as described above did not really work as we had expected. Clustering with the purely 15 minute patterns did work quite good and looked like figure 9.4, but when combined with the 5 minute patterns, the results were rather disappointing like figure 9.5. The reason for this was that several clusters contained many different patterns.

9.4 2 Stage Clustering with New Data

In the later stages of the project, we randomly found data of crypto currencies with about 100 000 records. Compared to the quantities used previously, this is an enormous amount.

An attempt was also made to implement the *2 stage clustering* using this data, although not very successfully. However, the implementation was much easier, because the data was stored in a single file and not in two different files, as was previously the case with the DAX. The implementation was basically a mixture of the first cluster attempt and the initial *2 stage clustering* approach. Which is why no new code has really been added, so the implementation is not covered. But in the end, the result did not differ that much from the initial *2 stage clustering* approach.

9.5 Implementation of a Neural Network

This section deals with the implementation of a neural network using the *keras* library. The following list presents the essential steps for building a simple neural network.

1. Pre-processing data

2. Build classifier
3. Fitting neural network to training-set
4. Predicting
5. Find best cluster for each prediction
6. Calculate how many predictions were false and right
7. Delete directory
8. Plot and save predicted patterns
9. Parameter tuning (optional)

9.5.1 Code Explanation

Preprocessing Data

First the data provided by the k-Means algorithm must be read in. This data included the cluster for each pattern and the respective prices.

The *pandas* library provides a method called *iloc* to retrieve specified rows and columns from a dataframe. The dataframe contains the cluster of each pattern in the first column and the prices in the remaining ones. The first value in the square brackets refers to the rows, the second to the columns. To get all prices, first a needs to be passed, which means that all rows should be returned and as a second parameter a *1* is passed. That is a slicing operator and indicates that all columns starting from the first column should be returned. To get all labels, the first parameter is a to get all rows and the second parameter is a *0* to get only the first column.

The library *sklearn* offers the method *train_test_split*, which splits the transferred data into training and validation datasets. With *test_size* it is possible to specify how many percent of the data should be used as training data. As a result four arrays are returned, two for training, each containing once the prices and the labels, and the other two for validation.

```
100     def preprocess_data(testSize, clusters):
101         data = pd.read_csv('labeled_data.csv', header=None)
102         prices = data.iloc[:, 1:].values
103         labels = data.iloc[:, 0].values
104         prices_train, prices_test, labels_train, labels_test =
    ↪ train_test_split(prices, labels, test_size = testSize)
105         y_train = np_utils.to_categorical(y_train, num_classes =
    ↪ clusters)
106         return x_train, y_train, x_test, y_test
```

Build Classifier

This method is responsible for initializing the neural network. First of all an instance of the *Sequential* class has to be created in order to add layers to it. This class represents the neural network model.

After that the input layer and the first hidden layer is added with the *add* method. Inside this method a *Dense* layer is created and this is just a regular layer of neurons in a neural network. Here it is possible to specify a number of parameters. With *input_dim* the number of input neurons is defined which in this case has to be the number of prices for a pattern. Obviously, the input neurons are only specified for the first layer. The *init* function is used to randomly initialize the weights at the beginning. The activation function for the neurons can also be set. *output_dim* specifies the number of neurons in the hidden layer.

In total tow hidden layers are added and one output layer in the end, which has the number of clusters as neurons and the activation function here is *softmax* because this function is for categorical outcome. The reason for a categorical result is because the neural network gives a certain percentage for a pattern to each cluster, depending on the probability that this pattern is formed within the cluster. Now that the model is defined, it can be compiled and returned.

```
100     def build_classifier(optimizer):
101         classifier = Sequential()
102         classifier.add(Dense(output_dim = output_dim, init = 'uniform',
    ↪ activation = 'relu', input_dim = input_dim))
103         classifier.add(Dense(output_dim = output_dim, init = 'uniform',
    ↪ activation = 'relu'))
104         classifier.add(Dense(output_dim = clusters, init = 'uniform',
    ↪ activation = 'softmax'))
105         classifier.compile(optimizer = optimizer, loss = '
    ↪ categorical_crossentropy', metrics = ['accuracy'])
106         return classifier
```

Fitting Neural Network to Validation Set

In order to train the model, the *fit* method needs to be invoked with the following parameters. The data, which represents the prices of the samples, followed by the cluster labels then the *batch_size*, which is the number of observations after the weights are updated. Lastly, the amount of *epochs* is required. An epoch refers to one cycle through the full training dataset

```
100     def fitting_ann_to_train_set(classifier, x_train, y_train, batchSize
    ↪ , epochs):
101         classifier.fit(x_train, y_train, batch_size = batchSize, epochs
    ↪ = epochs)
```

Predicting

This method is responsible for predicting the clusters for each pattern of the validation data. As parameters the trained neural network and the validation data is passed. In order to make predictions with the model, the *predict* method is called with the test data passed as an argument.

The result is an array with n columns, depending on how many clusters have been defined previously. As already mentioned in section 9.5.1, each cluster is given a certain percentage representing the probability of this pattern being formed within that cluster.

```
100     def predict_test_results(classifier, x_validation):
101         y_pred = classifier.predict(x_validation)
102         return y_pred
```

Find Best Cluster For Each Prediction

This method is responsible for finding the cluster with the highest probability for each pattern. To do so, the array with the probabilities must be iterated over and each line has to be converted to a list in order to be able to call the *max* method, which returns the highest value of the list. However, it is not the value itself that is actually required, but rather the index of the value, since the index corresponds to the cluster. To retrieve that index, the *index* method is called with the max value as an argument. A list with the predicted clusters for each pattern is returned at the end.

```
100     def find_best_cluster_for_predictions(y_pred):
101         pred_clusters = []
102         for i in range(0, len(y_pred)):
103             data = y_pred[i].tolist()
104             max_value = max(data)
105             max_index = data.index(max_value)
106             pred_clusters.append(max_index)
107         return pred_clusters
```

Calculate How Many Predictions Were False and Right

This is where the number of correct or incorrect predictions is calculated. Each predicted cluster is simply compared with the actual cluster and based on that, the counters for right or false are incremented.

```
100     def calculate_right_false(pred_clusters, y_test):
101         right_predictions = 0
102         false_predictions = 0
103         right_false_predictions = []
104         for i in range(0, len(pred_clusters)):
105             if pred_clusters[i] == y_test[i]:
```

```
106                 right_predictions += 1
107                 right_false_predictions.append('right')
108             else:
109                 false_predictions += 1
110                 right_false_predictions.append('false')
111         print("Right predictions: " + str(right_predictions) + "\nFalse
    ↪ Predictions: " + str(false_predictions))
112         return right_false_predictions
```

Afterwards the predicted patterns are saved as images, as it was done already approximately in section 9.2.1, so this will not be explained here.

Parameter Tuning

To find the perfect number for the *batch_size* or *epochs* and the best *optimizer* method, *keras* offers a *GridSearchCV* method. The parameters that should be tested must be specified first. The neural network is then trained once with each parameter and afterwards the parameters are returned with the best results. It is important to mention that if this is done on a conventional computer, it will take an extremely long time to complete.

```
100     def parameter_tuning(x_train, y_train, cv):
101         classifier = KerasClassifier(build_fn = build_classifier)
102         parameters = {'batch_size': [15, 25, 32, 40],
103                       'epochs': [100, 200, 300, 500],
104                       'optimizer': ['adam', 'rmsprop']}
105         grid_search = GridSearchCV(estimator = classifier,
106                                    param_grid = parameters,
107                                    scoring = 'accuracy',
108                                    cv = cv)
109         grid_search = grid_search.fit(x_train, y_train)
110         best_parameters = grid_search.best_params_
111         best_accuracy = grid_search.best_score_
112         return best_parameters, best_accuracy
```

9.5.2 Results

Our initial intention to give the neural network only a certain percentage of the prices of each pattern for training turned out to be unfeasible because the resulting accuracy of the model was too low, usually between 30 and 50 percent. The main reason for the inaccuracy of the model lies in the problems encountered with the *2 stage clustering*, which have already been mentioned here 9.3.2. Another reason is that the neural network had problems to find correlations between two patterns if only a certain percentage of the prices is used, for example 6 out of 10 prices.

Chapter 10

Data Preparation

As part of this project, different algorithms were developed to prepare the data for supervised machine learning models. One of the core concepts of machine learning is: *All data is the same*. Different models do not need differently processed data, the only distinction that has to be made is in preparing data for classification problems and for regression problems. Even then, the data processing algorithms share a big part of the code. This simplifies the process of using different models with rising complexity, as the simplest models can learn from the same data as the most complex ones. There is however one exception to this rule, which is working with sequence data, as explained in chapter 8.

At the start of this project some of the questions to be answered were:

- How much of the past has relevant effects on future price movements?
- How dense do charts of the past need to be in order to capture meaningful price movements?
- How far can one predict into the future given some recent chart?

These questions were examined in a trial-and-error-fashion. Of course the above questions had to be taken into consideration when implementing the methods for data preparation.

10.1 Available Data

In later stages of the project different data was available than in the beginning. The data however always shared a similar format. Mostly continuous time series data was present in *csv* files with the following columns (In this context continuous means that the data is available as one series with no gaps, the time difference between the prices is equal in all steps):

- Some sort of timestamp

- The *open* price
- The *closing* price
- A *high* price
- A *low* price

In this project only the opening or closing prices were used as input to the models. The reasoning behind this is, that the *low* and *high* columns contain prices that are chronologically not equidistant, because they can be placed anywhere between the *open* and *close* prices. The models would have a harder time learning patterns.

timestamp	open	high	low	close
2019/06/26 10:30:00	12216.2	12234.0	12213.0	12231.2
2019/06/26 10:45:00	12231.0	12239.5	12230.7	12234.2
2019/06/26 11:00:00	12234.5	12291.5	12229.2	12287.2
2019/06/26 11:15:00	12287.5	12312.2	12283.2	12302.7
2019/06/26 11:30:00	12303.2	12309.0	12293.5	12304.2

Figure 10.1: Exemplary data of the DAX instrument with the tick prices being 15 minutes apart

timestamp	low	high	open	close	volume
1528968720	485.98999	486.5	486.019989	486.01001	26.019083
1528968780	486.00000	486.0	486.000000	486.00000	8.449400
1528968840	485.75000	486.0	486.000000	485.75000	26.994646
1528968900	485.75000	486.0	485.750000	486.00000	77.355759
1528968960	485.98999	486.0	486.000000	486.00000	7.503300

Figure 10.2: Exemplary data of the Ethereum instrument with the tick prices being 1 minute apart - notice that the time stamp is in the format of time in seconds since the *epoch*

10.2 Course of Action

The basic goal of data preparation is to transform the available sequence of price data into N short sequences of length T - an input matrix of size $N \times T$. Conceptually this can be visualized by imagining a window, which only looks at a part of the sequence, rolling over the entire sequence and this way chopping the main sequence up into smaller, overlapping bits. In the processes of preparing the input data, the rolling sequences have to contain $T + 1$ prices, because the price at position $T + 1$ is the target, which the model has to learn to predict. Therefore, when chopping the main sequence up into rolling sequences, the rolling sequences have to be of length $T + 1$.

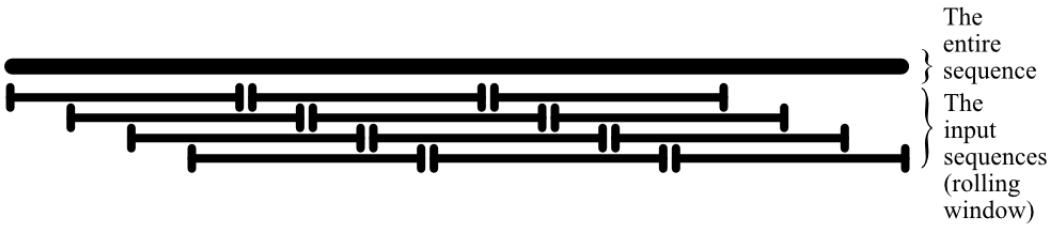


Figure 10.3: A visual explanation of the *rolling window*-concept

The next step is to split the obtained rolling sequences into input sequences and targets. To do that, a method is needed that takes in the rolling sequences and returns the input sequences and the corresponding target prices in separate arrays.

The third step is part of the normalization process, and is therefore potentially optional. At this point in the procedure, the rolling sequences contain absolute, unaltered prices. This is suboptimal for the models, as they would need to learn to abstract the patterns from the absolute prices. Especially for less complex models this is potentially impossible. Therefore a certain *baseline* (which is different for each rolling sequence) will be subtracted from each value of the rolling sequence. Note that the term *rolling sequence* is used to intentionally refer to labels as well as input sequences. In this section the baseline is defined as either the first or the second to last price of the rolling sequence. The new rolling sequence values are calculated by subtracting the baseline from each original value. This now means, that in every input sequence the first (or second to last) value is always 0. It can be dropped because it contains no information.

Finally the targets can be transformed to either be regression targets or classification targets. If the model is a regression model, nothing has to be done. When the model is a classification model, the only work that has to be done is to check whether the target value is higher or lower than the value of the input sequence at position T, and then set the new target to be a label of either the value 1, or 0. In this project the value 1 was used to indicate that the price to predict is higher than the last price of the input sequence, whereas 0 was used that the target label is either equal to or lower than the last value of the input sequence.

One last distinction has to be made for shallow techniques and neural networks. Because the general architecture of neural networks enables them to predict more than one output, that output has a different dimensionality than the output of logistic regression models (without softmax). When a neural network with K output classes is fed N samples for prediction, the output of the neural network is of size $N \times K$, whereas when a logistic regression model is fed N samples, its output is just N, because by default it only does binary classification. If a neural network does binary classification, its output

is of shape $N \times 1$ (K would be 2). Since the targets of the above procedure are a vector with length N, they have to be reshaped and altered to be of size $N \times 1$ to act as targets for neural networks.

The following is a compact list that contains the steps necessary to process the available data, which is in the form of one long sequence, to be usable input data.

1. Load the data from the CSV file(s)
2. Generate the rolling sequences from the main sequence
3. For each rolling sequence calculate the deltas from all values to the baseline and set them to be the new values of the rolling sequences (`get_deltas`)
4. Separate targets and input sequences
5. Optional: If a classification model is used turn the targets into classification labels, and if neural networks are used reshape the target array
6. Split the data into train data and test data

10.2.1 Train-Test-Split

To validate the models, the data has to be split into train data and validation data. Usually the samples are rearranged randomly and validation data is selected at random. In the case of time series prediction this however is not useful, because it leads to overfitting, since the model is able to peek the validation data. The results would be falsified.

This is only a problem if if the rolling sequences are overlapping. When mixing them and randomly selecting the validation samples, the probability of validation sequences being contained in training sequences is high.

Therefore another approach had to be used in this project. The approach that was used, was to select the last 20% of all rolling sequences (arranged chronologically) as validation data. This way there is next to no overlap between train and test sequences.

10.3 Non-Sequential Data Preparation

The strategy for finding models that deliver good results was to try models with rising complexity. The less complex models need data in rows and columns - data that is prepared non-sequentially. More complex models used for sequence analysis need data with an added dimension, the *sequence length* T.

This section explains how continuous time series data is prepared for non-sequential models. This means that the data can only have 2 dimensions, therefore each step of a sample sequence is a feature. In the scope of this project only one time series was given as an input for non-sequential models, because there was no hope in the models performing better if even more complex data would be given to them.

10.3.1 Code Explanation

The following will explain the code that accomplishes the steps described in section 10.2

Loading the Data from a CSV File

The first step is to load the data from a csv file using the library *pandas*.

```
100 def load_from_csv(filename, header, column_headers):
101     df = pd.read_csv(filename, header=header,
102                      names=column_headers)
103     return df
```

This method acts as a wrapper for the pandas `read_csv` method. The pandas method takes in many different arguments, and because only some arguments are needed, they cannot be given as positional arguments but instead have to be named arguments. The wrapper method above simply enables the caller to use positional arguments again.

The arguments `header` and `column_headers` are used in an either-or-fashion. If the header argument is provided, it has to be an integer, which indicates the row-number where the names of the column headers are (usually at row 0). One can however still specify the `column_headers` argument to name the column headers differently than given in the file. When the file has no column headers, the `header` should be set to `None`.

The data frame is not yet indexed by the timestamp, this is done in the next step.

Generating the Rolling Sequences

Multiple ways of accomplishing this task have been implemented. The basic idea is to loop over every single value of the main sequence. Every price in the main sequence is then the start to a rolling sequence with length `sequence_length_T`

This function was called hundreds of times in order to evaluate the predictability of many combinations of instruments and interval resolutions of the consecutive prices. Therefore an implementation was needed that used Numpy for fast array operations.

The implementation is split up into multiple functions, including one that generates rolling sequences with the interval the data natively has (the inner function), and another that then takes the rolling sequences and picks the prices in the wanted interval (the outer function).

The Rolling Window Method

The method for generating the rolling sequences with the rolling window method uses a special numpy function:

```
100 numpy.lib.stride_tricks.as_strided
```

This function creates a view into an array given a wanted shape. This means it manipulates the internal data structure of the array. Arrays created with this function contain

self overlapping memory - this is how the rolling sequences are created. The complete function is in listing 10.1

Listing 10.1: A function for efficiently creating rolling sequences with Numpy

```
100 def _rolling_window(a: np.ndarray, window_size: int)
101         -> np.ndarray:
102     shape = (a.shape[-1] - window_size + 1, window_size)
103     strides = a.strides + (a.strides[-1],)
104     return np.array(np.lib.stride_tricks.as_strided(
105             a, shape=shape, strides=strides
106         ))
```

The Rolling Sequences Method - The Outer Function

This function is responsible for providing rolling sequences of a certain interval and input sequence length T. It takes the following parameters:

- The complete time series: `time_series`
- The wanted input sequence length T: `sequence_length_T`
- The wanted interval: `interval`
- A boolean to indicate whether volatilities should also be calculated: `include_volatilities`

Because the rolling window method only provides rolling sequences of native interval, it needs to return sequences of the wanted input sequence length multiplied by the wanted interval. The rolling sequences method then takes only the elements in the wanted interval and drops all other elements. This is best explained by the code in listing 10.2.

Listing 10.2: The function for creating rolling sequences

```
100 def calculate_indices_to_take(sequence_length_T, interval):
101     indices_to_take = []
102     for i in range(sequence_length_T):
103         index = i * interval
104         if index != 0:
105             index = index - 1
106         indices_to_take.append(index)
107     return indices_to_take
108 
109 def get_rolling_sequences(time_series: np.ndarray,
110     sequence_length_T: int, interval: int = 1,
111     include_volatilities=True) \
112         -> (np.ndarray, np.ndarray, np.ndarray):
```

```
    window_size: int = sequence_length_T * interval
    sequences: np.array \
        = _rolling_window(time_series, window_size)

    indices_to_take = calculate_indices_to_take(
                        sequence_length_T, interval
                      )

    volatilities = np.array([])
    if include_volatilities:
    volatilities = volatility(sequences,
                              np.array(indices_to_take))

    shortened_sequences \
        = sequences.take(indices_to_take, axis=1)
    targets = sequences.take([window_size - 1], axis=1)
    return shortened_sequences, targets, volatilities
```

10.3.2 The Method to Get the Deltas to the Baseline

This is the method responsible for calculating the deltas to the baseline of each rolling sequence. The method then returns new rolling sequences with the newly calculated values. It creates a new array with the same number of rows as the original rolling sequences and with one less column. The column of the baseline values is dropped, because like explained above, this column would contain only 0's. It then simply loops over every single value (except the first one as it is the *baseline*) and subtracts the baseline from them.

```
def calculate_deltas_to_first(sequences):
    n, t = sequences.shape

    new_sequences = np.array(sequences)
    for i in range(t):
        new_sequences[:, i] \
            = new_sequences[:, i] - new_sequences[:, 0]

    # delete the first column as it is always 0
    return np.delete(new_sequences, 0, 1)
```

In later parts of the thesis the placement of the baseline has been evaluated. It was tested, whether the results were better when setting the baseline as the last value of the input sequences. For this another method was implemented (using Numpy). Note, that this function also already separates the targets from the input sequences.

```
def get_deltas_to_last(sequences: np.ndarray,
    targets: np.ndarray) -> (np.ndarray, np.ndarray):
    n, t = sequences.shape

    new_targets = targets - sequences[:, (t-1)]

    new_sequences = np.array(sequences)
    for i in range(t):
        new_sequences[:, i] \
            = new_sequences[:, i] - new_sequences[:, (t-1)]

    # delete the last column as it is always 0
    return np.delete(new_sequences, t-1, 1), new_targets
```

10.3.3 Separating Targets and Input Sequences when the Baseline is at t=0

This is the only part of the data processing where the procedure for regression models and classification models differ. Two separate methods are needed for the respective approaches. Generating classification targets is more complex than generating regression targets, because the regression targets are simply the last column of the rolling sequences. For generating classification targets there is the additional logic of checking whether the value of the last column is greater than the value of the second to last column.

Both approaches however share generating the input sequences, as well as the initial part of generating the output sequences, which is as simple as one line of code, using a handy Numpy array operation.

```
X = sequences[:, :-1]
target_values = sequences[:, -1:]
```

The function for generating the classification targets when the baseline is at $t = 0$, first flattens the target values so that they are not 2-dimensional and contain just 1 column.

```
def get_X_Y_sequence_binary_classification(sequences):
    target_values = sequences[:, -1:]
    target_values = np.array(target_values)
        .flatten()X = sequences[:, :-1]
    X = sequences[:, :-1]
    N, D = X.shape
    Y = np.zeros(N)
    for i in range(N):
        if X[i, D-1] < target_values[i]:
            Y[i] = 1
    return X, Y
```

When the baseline is the first value of the input sequence, the input sequence is needed to evaluate whether the target is greater or lower than the last value of the input sequence. When the baseline is at the last value of the input sequence however, the target is either greater or smaller than 0. This alone is already an indicator whether the target should be 0 or 1, which is why the input sequence is not needed for the function of getting calculating the targets when the baseline is at $t = T - 1$. This function uses Pandas for efficient handling of the logic.

```
100 def get_targets_as_classification(targets: np.ndarray)
101         -> np.ndarray:
102     df = pd.DataFrame({'a': targets})
103     df['a'][df['a'] >= 0] = 1
104     df['a'][df['a'] < 0] = 0
105 
106     return np.array(df.a)
```

10.3.4 Splitting Train and Test Data

The function responsible for splitting the data into train data and test data is rather simple. The test data is simply the sequences that appear at the end of the main sequences.

```
100 def split_train_test(X, Y, percentage_train):
101     X_train = X[:(round(X.shape[0]*percentage_train)), :]
102     Y_train = Y[:(round(Y.shape[0]*percentage_train))]
103 
104     X_test = X[(round(X.shape[0]*percentage_train)):, :]
105     Y_test = Y[(round(Y.shape[0]*percentage_train)):]
106 
107     return X_train, X_test, Y_train, Y_test
```

10.3.5 The Mirror Function to Counter the Long / Short Bias

The machine learning models suffer from the fact, that the dataset is imbalanced. Instead of learning patterns from the charts, they simply learn that one class is more likely to occur than another. To prevent that, one approach was to *mirror* the train sequences after the baseline has already been subtracted. All the values in the rolling sequences simply have to be multiplied by -1. This also applies to regression targets. Categorical targets have to be set to 0 if they are one and vice versa. When the training data is mirrored, the dataset is balanced. However, this method also distorts the training data, so there is a possibility for the validation results to be worse than without mirroring.

The function to mirror rolling sequences is really simple, it is just one Numpy operation. It does require however that the rolling sequences are already in their *delta-to-baseline* form. This function works for the input sequences and regression targets.

```
100 def mirror(a: np.ndarray) -> np.ndarray:
101     return np.array(a * (-1))
```

For mirroring the targets for classification another function is needed, as this operation is not as simple as simply flipping signs. 0s have to become 1s and vice versa. This can be done efficiently with Pandas.

```
100 def mirror_binary_targets(targets: np.ndarray)
101         -> np.ndarray:
102     new_targets = np.array(targets - 1)
103     df = pd.DataFrame({'a': new_targets})
104     df['a'][df['a'] < 0] = 1
105     return np.array(df.a)
```

Chapter 11

First Implementation Of Supervised Techniques

Because the process of trying to cluster similar looking parts of the chart seemed not to work as expected, the next logical step was to let the machine learning algorithms do more of the abstraction. Previously the assumption was made, that parts of the chart of *some* length $T_{previous}$ influenced the future movement of the instrument. The choice of how big $T_{previous}$ and $T_{predict}$ are, was made by humans in fixed code, and not left for the algorithms to learn.

The following sections describe different approaches of making the algorithms learn patterns and coherences completely by themselves, without humans making decisions, like choosing the size of $T_{previous}$ and $T_{predict}$. Both regression approaches and classification approaches were used. Regression was used to predict future instrument prices, whereas classification was used to predict whether an instrument would rise or fall in value.

The procedure which was used to determine which algorithms to try out was to increase the complexity of the models until reasonably good results were achieved.

11.1 Logistic Regression

Although logistic regression is a conceptually more complex model than linear regression, classifying whether the next value of a time series is greater or lower than a previous value, is a simpler task than predicting the next value itself. This is because to some extent regression is an abstract form of continuous classification. If one were to look at regression from a classification standpoint, there would be an infinite amount of classes. Predicting whether the next value is greater or lower than the previous one is a classification problem with only two classes. From this perspective it is easy to see why classification is a simpler problem to solve than regression, at least in the space of time series prediction.

11.1.1 Implementation

Logistic regression was implemented without any deep learning libraries, because the effort of coding the model by hand is about equal to researching how to implement logistic regression with deep learning libraries. The only steps necessary for implementing logistic regression are to:

- Initialize the weights
- Implement a *predict* function
- Implement a *fit* function using gradient descent
- Implement the metric *classification rate*

The classification rate is the ratio of correct predictions to incorrect predictions.

An object oriented approach was chosen to implement the logistic regression model, so a class `LogisticRegression` was implemented. This class defines the methods:

- `predict`
- `fit`
- `classification_rate`

The only constructor argument is the input dimensionality, this way the weights can be initialized randomly in it.

The code for this task is only 3 lines.

```
100 class LogisticRegression:
101     def __init__(self, D):
102         self.W = np.random.randn(D + 1)
```

Note, that by initializing the weights with one extra column, the bias term is absorbed into the weights.

The sigmoid function was used as the activation function, because its output can be interpreted as a probability. Making a prediction is as simple as rounding the output of the sigmoid function.

```
103     def predict(self, X):
104         return np.round(sigmoid(X.dot(self.W)))
```

Note, that the function *sigmoid* is defined in the same file where the class is defined.

To fit the model to the inputs using gradient descent, the *fit* function needs the parameters:

- X (input)
- T (targets)

- The learning rate
- The validation inputs and the validation targets

The cross entropy error is used as the cost function. To calculate it the following function has been implemented.

```
100 def cross_entropy_error(T, Y):
101     J = 0
102     for i in range(len(T)):
103         if T[i] == 1:
104             J -= np.log(Y[i])
105         else:
106             J -= np.log(1 - Y[i])
107     return J
```

This is unfortunately the simplest way of doing this operation.

The fit function stores the progression of the costs in python lists and returns them in the end, this way they can be plotted later. The most important line in the fit function is where the weights are updated in the direction of the gradient. Using the gradient calculated in chapter 6 the weights are updated with the following line:

```
100 self.W -= learning_rate * X.transpose().dot(Y - T)
```

The complete *fit* function is part of the `LogisticRegression` class and contained in the following listing.

```
100     def fit(self, X, T, epochs, learning_rate,
101         print_cost_rate, X_test, T_test):
102         costs = []
103         costs_test = []
104         classification_rate = []
105         classification_rate_test = []
106         Y = sigmoid(X.dot(self.W))
107 
108         for i in range(epochs):
109             if i % print_cost_rate == 0:
110                 c = cross_entropy_error(T, Y)
111                 Y_test = sigmoid(X_test.dot(self.W))
112                 c_t = cross_entropy_error(T_test, Y_test)
113                 costs.append(c)
114                 costs_test.append(c_t)
115                 clr = self.classification_rate(X, T)
116                 clr_test =
117                     self.classification_rate(X_test, T_test)
118                 classification_rate.append(clr)
119                 classification_rate_test.append(clr_test)
```

```
120                 print("cross␣entropy␣error", c,
121                     "␣classification_rate", clr)
122             self.W -=
123                 learning_rate * X.transpose().dot(Y - T)
124             Y = sigmoid(X.dot(self.W))
125         return costs, costs_test,
126             classification_rate, classification_rate_test
```

The *classification_rate* function is given an input and the corresponding targets. First, it calculates the predictions. Next, it subtracts each prediction from the corresponding target and takes the absolute value from this output. If the prediction is equal to the target, the output of this calculation is 0, else it is 1. Then it sums up all the outputs of the calculations. This is now the number of incorrectly predicted samples. To get the number of correctly predicted samples the number of incorrectly predicted samples is subtracted from the overall number of samples.

```
100     def classification_rate(self, X, T):
101         Y = self.predict(X)
102         wrong = np.abs(T - Y)
103         n_incorrect = wrong.sum()
104         n_correct = len(Y) - n_incorrect
105         return n_correct / len(Y)
```

Putting it Together

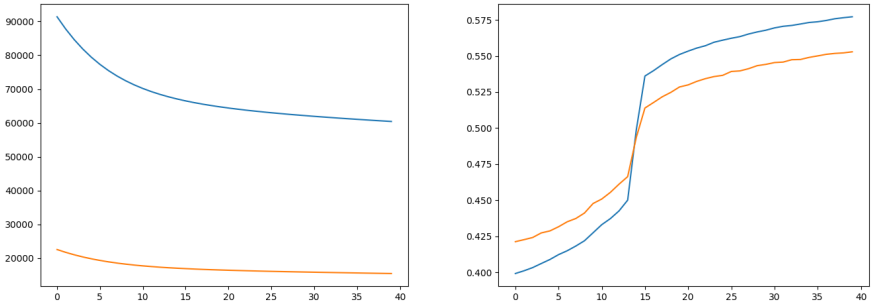
Here is the code which prepares the data and then trains the logistic regression model.

```
100 df = load_from_csv('ETH-USD.csv', [
101     'timestamp',
102     'low',
103     'high',
104     'open',
105     'close',
106     'volume'
107 ])
108 sequences = get_sequences(df, 'timestamp', 'close', 5, 60)
109 X, Y = get_X_Y_sequence_binary_classification(sequences)
110 X = calculate_deltas(X)
111 X = append_ones_to_X(X)
112 X_train, X_test, Y_train, Y_test =
113     split_train_test(X, Y, 0.8)
114
115 lr = LogisticRegression(X.shape[1] - 1)
116 costs, costs_test, cr, cr_test \
```

```
    = lr.fit(X_train, Y_train, 200, 1e-6, 5, X_test, Y_test)
plt.plot(costs)
plt.plot(costs_test)
plt.show()
```

After the training has finished running the costs and the classification rates are plotted. The training lasts 200 epochs, the learning rate is initially set to 10^{-6} and the costs are printed and saved every fifth epoch.

11.1.2 Results



(a) Costs (b) Classification Rate

Figure 11.1: The training metrics of the logistic regression training process

Despite all expectations the model actually learns from the data. The costs sink for both training samples and test samples, while the ratio of correct predictions to incorrect predictions rises. The results have to be further examined.

One interesting aspect to look at, is how often the model predict that the price will rise (in the validation data), how often the price really rises, and how often the model was correct with predicting that the price would go up. This can be accomplished with the following few lines of code, which are executed after training is finished.

```
Y_hat = lr.predict(X_test)
n_rise_hat = Y_hat.sum()
n_rise = Y_test.sum()
n_samples = len(Y_test)
print("Number␣of␣times␣price␣was␣to␣predicted␣to␣rise:␣",
    n_rise_hat)
print("Number␣of␣times␣the␣price␣actually␣did␣rise:␣",
    n_rise)
print('Number␣of␣overall␣samples:␣', n_samples)

indexes_predicted_to_rise = []
for i in range(n_samples):
```

```
112     if round(Y_hat[i]) == 1:
113         indexes_predicted_to_rise.append(i)
114
115 n_correct = 0
116
117 for i in range(len(indexes_predicted_to_rise)):
118     index = indexes_predicted_to_rise[i]
119     if round(Y_hat[index]) == round(Y_test[index]):
120         n_correct += 1
121
122 print("Number␣of␣times␣the␣rise␣prediction␣was␣correct:␣",
123     n_correct)
124 print("Ratio␣of␣correct␣risen␣vs␣incorrect␣risen:␣",
125     n_correct / len(indexes_predicted_to_rise))
```

```
Number of times price was to predicted to rise:  4127.0
Number of times the price actually did rise:  8342.0
Number of overall samples:  20102
Number of times the rise prediction was correct:  1744
Ratio of correct risen vs incorrect risen:  0.4225829900654228
```

Figure 11.2: The output of the code to inspect the results of the logistic regression model

11.1.3 Explanation

The reason for such a simple model to be able to predict correctly 60% of the time is actually rather simple. Over the whole dataset the price is falling. This means that in general (given the available dataset) the probability of the price falling is higher than the probability of the price rising. The model learned this, and learned that it is correct more often if it simply predicts that the price will fall. In cases where the model predicted the price to rise, it was actually more often wrong than it was right, it performs even worse than if one were to simply guess. The model can still improve, this means that the model has to learn over more epochs. To see whether it could still be improved, the number of epochs was increased to 1000.

Improved Results

The overall classification rate actually does not improve when training the model longer. What is interesting to note however, is that eventually the model stops predicting that the price rises, as seen in figure 11.3.

```
Number of times price was to predicted to rise:  1.0
Number of times the price actually did rise:  8342.0
Number of overall samples:  20102
Number of times the rise prediction was correct:  0
Ratio of correct risen vs incorrect risen:  0.0
```

Figure 11.3: The output after training the logistic regression model for 5000 epochs

11.1.4 Conclusion

Whilst the results are actually better than guessing, the model failed to learn meaningful patterns from the chart. This was to be expected, as logistic regression is only good at making predictions on data which is linearly separable.

11.2 A Neural Network with 1 Hidden Layer

The next more complex model is a neural network with 1 hidden layer. Neural networks are able to learn nonlinear coherences in the data, so they are better suited for such complex data.

11.2.1 Implementation

From this point on, Tensorflow was used to implement the models. With Tensorflow the implementation of common models less time-consuming and more flexible. Although implementing neural networks manually is possible, another benefit of Tensorflow is that the training can be accelerated using Graphical Processing Units.

The code for generating the datasets can be taken from the previous section, as it remains the same as generating the code for a logistic regression model. The only new code is the code for generating the model. The necessary imports are in the following listing.

```
100 from tensorflow.keras.layers import Input, Dense
101 from tensorflow.keras.models import Sequential
102 from tensorflow.keras.activations import sigmoid
103 from tensorflow.keras.optimizers import Adam
104 from tensorflow.keras.metrics import Recall,\
105     BinaryAccuracy, AUC, SpecificityAtSensitivity
```

Monitoring different metrics throughout the training is relatively simple in Tensorflow. Including the metrics, the code to build and train the model is just a few lines. First, the Sequential model has to be initialized with *all* its layers. This initialization also contains the specification for the number of units in each layer.

```
106 model = Sequential([
107     Input(X.shape[1]),
```

```
    Dense(24, sigmoid),
    Dense(1, activation=sigmoid)
])
model.compile(optimizer=Adam(),
                loss="binary_crossentropy",
                metrics=[
                    BinaryAccuracy(),
                    AUC(),
                    Recall(),
                    SpecificityAtSensitivity(0.5)]
                )
history = model.fit(X_train, Y_train, epochs=200,
                    validation_data=(X_test, Y_test))
```

This model has 24 units in the hidden layer. Moreover, training is monitored with the following metrics:

- Binary Accuracy: The proportion of correct predictions to incorrect predictions
- AUC (*Area Under Curve*): Measures the model's capability of distinguishing between the classes
- Recall: The proportion of correctly predicted positives versus all positives
- Specificity: The proportion of correctly predicted negatives versus all negatives

Note, that positives are samples where the label is 1, meaning that the price of the instrument will rise, and negatives are samples where the label is 0, meaning that the price of the instrument will fall.

Area Under Curve compared to Binary Accuracy

The *Area under Curve* will be explained only briefly. It is a good way of measuring the performance of models in cases, where the data is in-balanced, meaning that the target values are not uniformly distributed. The dataset at hand is in-balanced. This means, that when measuring the binary accuracy, the model performs well if it predicts the most common class. When measuring the model using AUC however, it would turn out that the model cannot separate the classes. The avid reader might wonder of which curve the area is taken. Most commonly it is the *Receiver Operating Characteristic-Curve*. This curve takes in the *true positive rate* such as the *false positive rate*. The true positive rate is also named the *recall* - the proportion of correctly predicted positives versus all positives. The false positive rate is the proportion of false positives to all negatives. This is the probability of the model falsely labelling the negatives (labelling the samples that are actually negatives as positives). By taking the area under the receiver operating characteristic curve, the model is not only measured for separability of the classes, but also for how sure a model is with its predictions (if a model predicts values that are

close to 0.5 the model is very unsure, when the predictions are close to 0 or 1 the model is very sure). The true positive rate and the false positive rate are calculated with the following *sub*metrics:

- True Positives: The number of samples where the model predicted the value to rise, where the model was correct
- False Positives: The number of samples where the model predicted the value to rise, where the model was incorrect
- True Negatives: The number of samples where the model predicted the price to fall, where the model was correct
- False Negatives: The number of samples where the model predicted the value to fall, where the model was incorrect

So even when one class is outnumbered by another, the area under curve still reflects that the model cannot distinguish between the classes (if, for example, the model suffers from long or short bias). The information for explaining the AUC metric has been taken from [31] and [28].

11.2.2 Results

Evaluating the results is simply done by visualizing the training history. This is done with the following code:

Listing 11.1: The code for plotting a metric

```
100 plt.plot(history.history['auc'], label='Area␣under␣Curve')
101 plt.plot(history.history['val_auc'],\
102     label='validation␣Area␣under␣Curve')
103 plt.legend()
104 plt.show()
```

This code in listing 11.1 is repeated for all metrics.

Plots

See 11.4

Loss

The development of the loss over the epochs can be seen in figure 11.4b. Interestingly, it seems like the model starts overfitting within the first 20-30 epochs, because the loss for the validation dataset (validation loss) increases, while the loss for the training dataset (train loss) decreases. In all following epochs however, the validation loss decreases - the model improves and learns universal patterns of the data. However it is noteworthy, that through all epochs the validation loss never comes below the loss at which training started (this is because of the initialization of the weights of the model).

Binary Accuracy and AUC

The binary accuracy is depicted in figure 11.4c and the area under curve is plotted in figure 11.4a. The evolution of this model during the training process is a perfect example of how the area under curve sometimes is a more expressive metric than simple binary accuracy. Whilst the binary accuracy did not improve over the training process, neither for training data nor for validation data, the area under the receiver operating characteristic curve improved for both training data and validation data. This is somewhat odd, because the validation loss did not improve in the training. This means that the model improved its ability to separate between the classes, whilst its overall accuracy did not improve. The reasoning for this can be explained by looking at the specificity and recall evolution.

Specificity and Recall

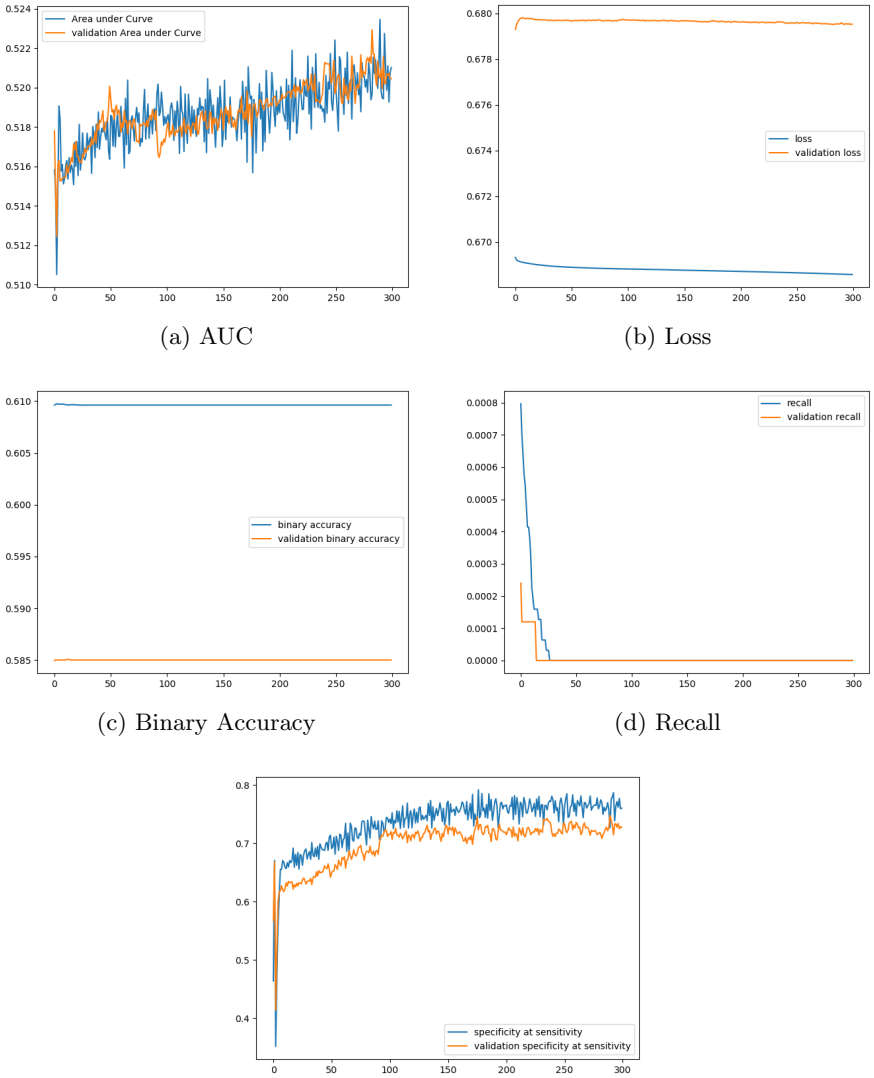
As seen in the recall chart, the model quickly fitted to the *long bias*. After just around 40 training iterations the recall rate sank to 0. This means the model either stopped predicting that the price of the instrument will rise (the positive class), and therefore has no true positives, or it only predicts false positives.

The specificity graph gives more insight on the model's behavior. In the first 30 epochs the specificity drastically increased, which means that the model started predicting more correct negatives.

Combining the insights of both graphs leads to the conclusion, that after the training process has finished it still sometimes predicts that the price will rise, however because the recall is at 0, this means that all of its positive predictions are wrong. Moreover, the specificity graph has another meaning. Since the false positive rate, the probability of the model wrongly predicting that the price will rise, is 1 - the specificity, the specificity graph also tells, that the false positive rate at the end of the training is around 25%.

11.2.3 Conclusion

Unfortunately, just like logistic regression, the neural network also suffers from the long bias problem, because it fails to learn meaningful patterns in the data. Therefore the model simply learned that the probability of a price falling is higher than the probability of the price rising.



(a) AUC

(b) Loss

(c) Binary Accuracy

(d) Recall

(e) Specificity

Figure 11.4: The training metrics of the neural network plotted over 300 epochs

Chapter 12

A Big Evaluation of Recurrent Neural Networks and Hyperparameters

This chapter explains the procedure of finding the best working combination of

- Hyperparameters
- Instruments
- Input Sequence Lengths
- Intervals
- Whether or not to include mirrored training sequences to prevent long / short bias

In the scope of this thesis, *best working* means predictable, measured with the *Area Under the ROC Curve*. All previous models seemed to have trouble learning from the data, and testing the models with different Hyperparameters and input sequences and combination of such was a time intensive, tedious process.

The following is a list of explanations of possible variations.

Hyperparameters: They mainly include the learning rate, the number of hidden units and the number of hidden layers.

Instruments: The available instruments include Bitcoin, Ethereum and Litecoin. It is possible that there are coherences between the price developments of the instruments, which is why every combination of these instruments had to be tested. To test the coherences, rolling sequences of all instruments in *a* combination were used as an input to predict one of the futures in the combination. All the possible combinations include:

- Bitcoin, Ethereum, Litecoin

- Bitcoin, Ethereum
- Bitcoin, Litecoin
- Litecoin, Ethereum
- Bitcoin
- Ethereum
- Litecoin

Input Sequence Lengths: It is unclear how much of the past affects future price developments of the instruments. The rolling sequence length is how many prices of the past are given as inputs to the models.

Intervals: This is the resolution of the history the model is given. The intervals were tested in a range of 5 minutes up to 16 hours. Together with the rolling sequence length, this parameter determines how much of the past is given the model at *which* resolution. The models always predict one step of the interval into the future. If, for example the interval is 1 hour and the input sequence length is 10, the model is given 10 prices of the past, each 1 hour apart, and predicts 1 hour into the future.

12.1 Course of Action

This section only explains the course of action without showing or explaining code. This is because most of the necessary code was already used in previous attempts and has already been shown and explained in previous parts of this thesis.

Unfortunately, not every possible combination could be tested, because this would have meant that thousands of different models would have to be trained. Training so meany models would have taken more time than available.

The basic strategy to try out as many as feasible combinations was to run tests on combinations in different stages, in order to rule out combinations and models that did not deliver promising results. In the first stage models were tested on the different combinations of instruments in order to see which instrument-combinations lead to the best results.

In the next stage these instrument combinations were tested in more detail, and more combinations of input sequence lengths and intervals were tested.

This means that for each new iteration where a new combination is tested out the dataset has to be newly created. A method was created which prepared the available data to be a dataset conforming to the combination of each iteration. Moreover a method was created which created and then trained the different models for different combinations. These methods are called in loops, where the loops iterate over every option for every selected combination to test out at each stage.

12.1.1 Creating the Datasets

The `create_dataset` method works in 10 major steps. First, the complete time series' of all relevant instruments are read from the csv files and subsequently joined on their timestamps to create one data frame. Second, the rolling sequences are created for each instrument, and for the predicted instrument the targets are calculated. At this point the volatilities are optionally also calculated and added to the data frame. Next, the targets are reshaped to a vector. After that the deltas to the baselines of the rolling sequences are calculated. The targets are then optionally converted to classification targets. At this point the data is split into train and test data. Afterwards the targets and all input sequences (price sequences as well as volatility sequences) of the training data, are mirrored to prevent a long or short bias. And finally the sequences are optionally scaled for a possibly better effectiveness of the models.

The following is a comprehensive list of the steps:

1. Read and join time series
2. Create rolling sequences
3. Separate input sequences and targets, keep the targets for the instrument to predict
4. Optional: Calculate volatilities
5. Reshape the targets
6. Calculate the deltas to the baselines
7. Optional: Convert targets for classification
8. Split data into train data and test data
9. Scale train data and apply the scaler to the test data
10. Mirror the input sequences

12.1.2 Creating and Training the Models

The models that were used are neural networks with LSTM hidden layers. In the first stages only one hidden layer was used. Later multiple hidden layers were used to see whether deeper networks would perform better. The training lasted lasted between 100 and 300 epochs, with a mechanism in place that would stop the training if the models did not improve on the test data. This is a measure to save time. After the training is finished the development of the metrics over the epochs are plotted and saved to files.

12.2 The First Training Round

In the first training round the goal was to find the most predictable combinations of instruments. All the combinations used mirroring of the input sequences, as this is the only way of preventing the models to learn the long bias, and scalers were applied. The used resolution was 30 minutes (the prices are 30 minutes apart) and the input sequence length was 10 (10 prices, each 30 minutes apart were the inputs to the models). The performance of the models was judged using the Area under Curve, because it is the most expressive metric.

12.2.1 Results

The overall results were mixed, the best models reached an AUC of up to 57%, whilst the worse models all hovered around 51-52%. However, some patterns formed. All in all, the combinations with the best predictability were those, where the predicted instrument was the Bitcoin, or where the Bitcoin was an input. Combinations where the Litecoin was predicted did not perform as well as BTC-predicting combinations, however they still performed better than the worst combinations.

An area under curve of 57% is an improvement over all previously achieved results, however it is still extremely low, at just 7% better than guessing. A model that is right only 57% percent of the time cannot be used as a foundation for trading, because costs of trading would about equal the profits gained in the few times the model is correct.

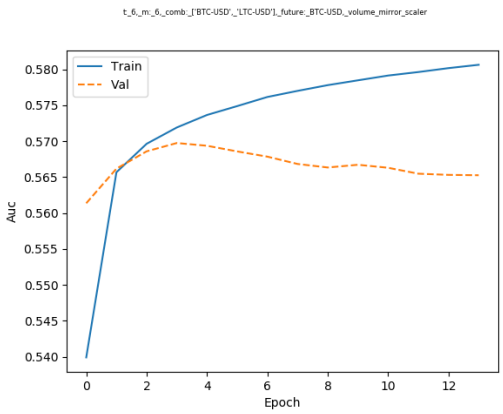
The plots of the development of the AUC over the epochs can be seen in figure 12.1.

The bad performing models mostly predict Ethereum, however some combinations with Litecoin as a target also did not perform well. The models seem to have problems learning patterns from the training data. Within just a few epochs the models start overfitting. The bad performing combinations are in figure 12.2

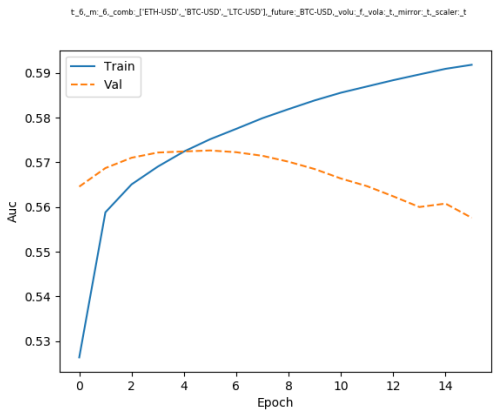
It is noteworthy that all combinations have an AUC of more than 50%, which should not be the case. The weights of the models are initialized randomly, which should mean that 50% of the trainings should start with an AUC below 50%. This might be an indicator for a too large learning rate, this would therefore mean that some minima in the objective function might not be found.

12.3 The Second Training Round

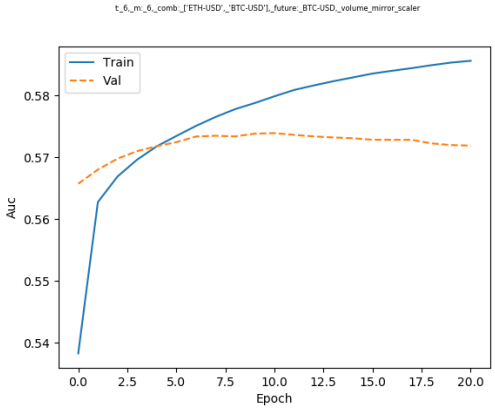
In the second round of training the goal was to test the hypothesis of the learning rate being too big. A learning rate which is too big can lead to deteriorated results, whilst a too small learning read leads to longer trainings and the danger of the models being stuck in local minima of the objective function. In the first round of training a learning rate of 10^{-4} was used. In this round learning rates of 10^{-4}, 10^{-5} and 10^{-6} will be tested.



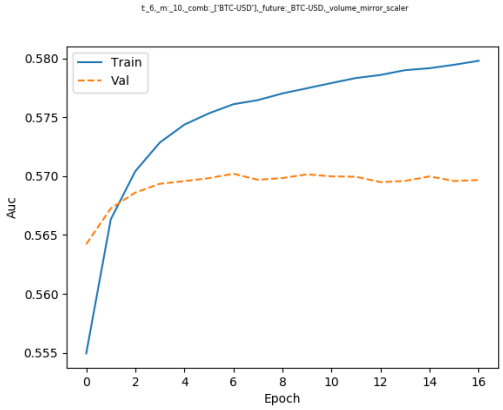
(a) In: BTC and LTC, Predict: BTC



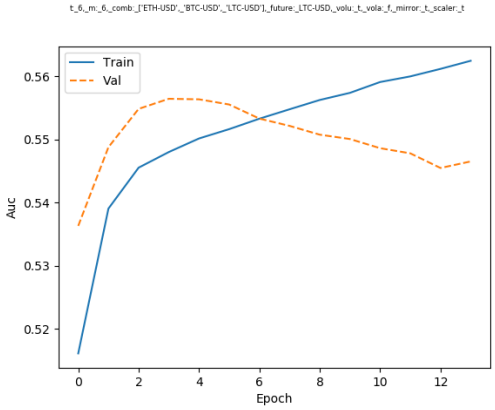
(b) In: All, Predict: BTC



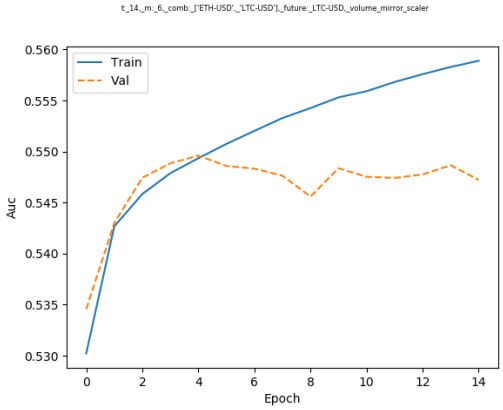
(c) In: BTC and ETC, Predict: BTC



(d) In: BTC

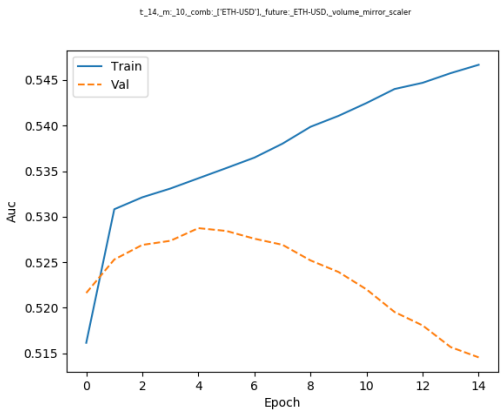


(e) In: All, Predict: LTC

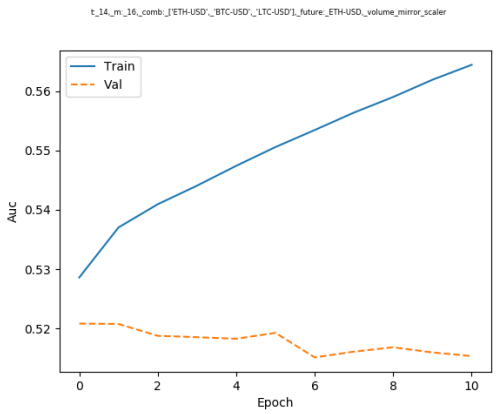


(f) In: ETC and LTC, Predict: LTC

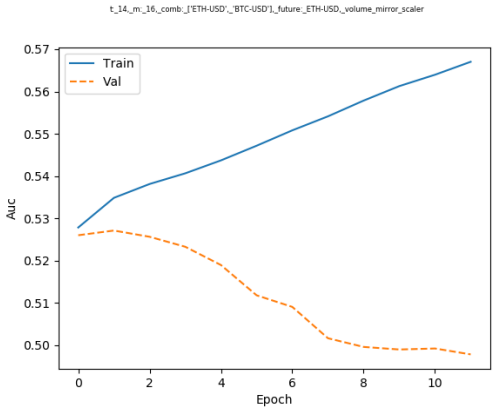
Figure 12.1: The better predictable combinations



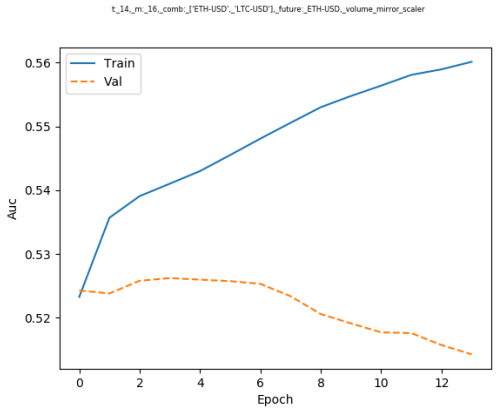
(a) In: ETC



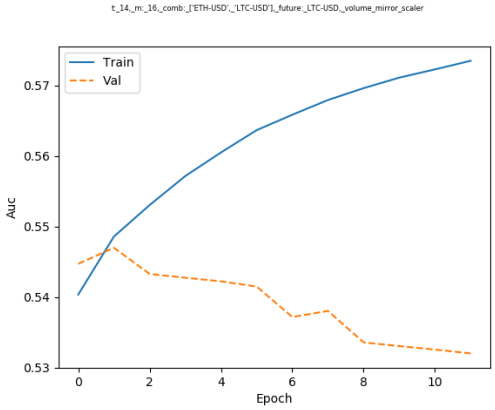
(b) In: All, Predict: ETC



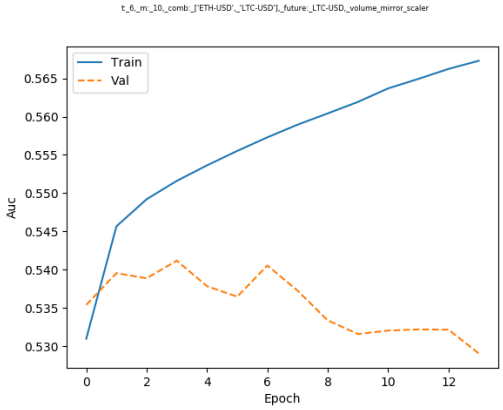
(c) In: BTC and ETC, Predict: ETC



(d) In: ETC and LTC, Predict: ETC

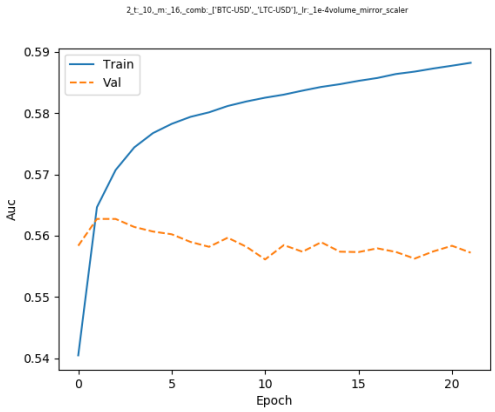


(e) In: ETC and LTC, Predict: LTC

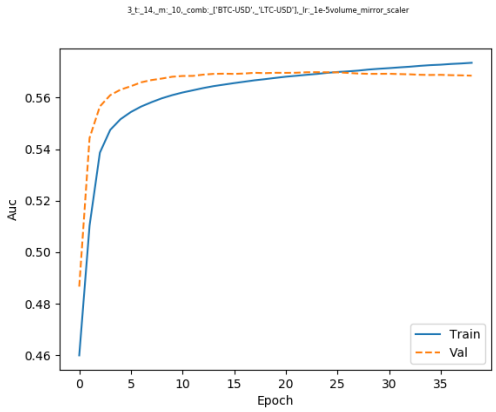


(f) In: ETC and LTC, Predict: LTC

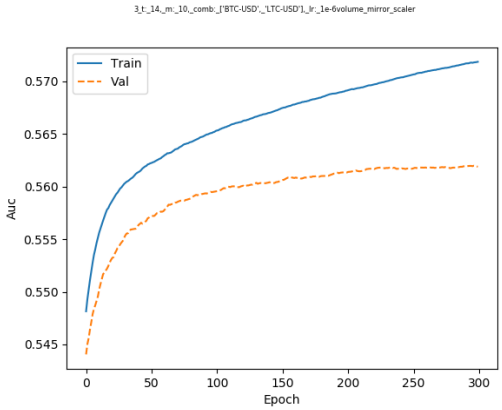
Figure 12.2: The worse predictable combinations



(a) Learning Rate: 10^{-4}



(b) Learning Rate: 10^{-5}



(c) Learning Rate: 10^{-6}

Figure 12.3: All the learning rates tested on the same combination of instruments

12.3.1 Results

The comparison of the results is visible in figure 12.3. The learning tests were tested on many combinations, the results in figure 12.3 are representative of the overall results. The hypothesis, that when the learning rate is smaller, the AUC would be below 50% for a few models proved to be true, as seen in figure 12.3a. Whilst the highest achieved AUC was around the same value of 56% for all learning rates, it is clear that the improvement is much smoother when the learning rate is lower than 10^{-4}.

After this stage, a learning rate of 10^{-5} proved to be the best choice, because the improvement over the training is just as smooth as with a learning rate of 10^{-6}, the training however is faster by a magnitude of 10, as apparent from the scale of the x-axis in figures 12.3b and 12.3c.

12.4 The Third Round of Training

At this stage the goal was to find predictable combinations of intervals and input sequence lengths. The learning rate was set to 10^{-5}, and the most predictable instrument combinations found in section 12.2 were used.

The tested intervals include:

- 30 minutes
- 1 hour
- 2 hours
- 4 hours
- 8 hours
- 16 hours
- 1 day

To achieve a feasible training duration the selection of input sequence lengths was limited to:

- 10
- 14

For the same reason the tested numbers of hidden units in the LSTM layer were:

- 10
- 16

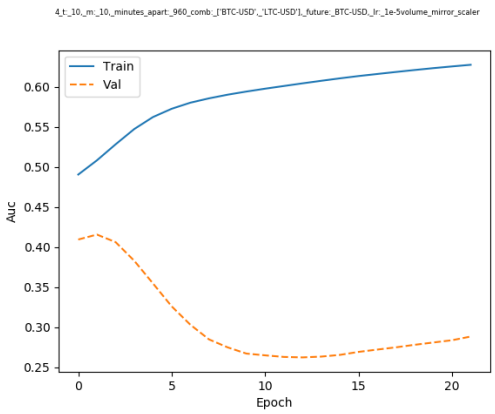
12.4.1 Results

Of the around 300 tested combinations, the combinations are going to be put into the following groups, grouped by best AUC of the validation dataset (over training):

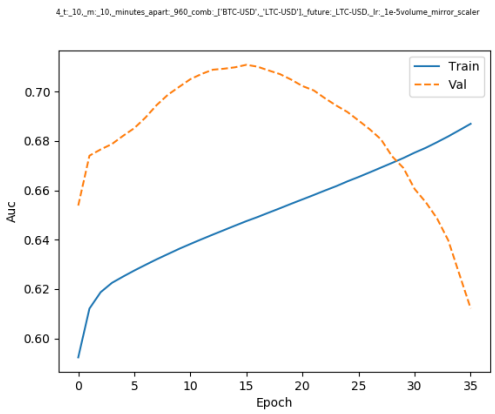
- AUC lower than 60%
- AUC between 60% and 70%
- AUC above 70%

In this round of training the results started to be good with a lot of combinations achieving an AUC of more than 70% (figure 12.4). Again patterns started to form: Generally, the more time is between the prices (the lower the interval) the better the models seem to be able to learn the patterns.

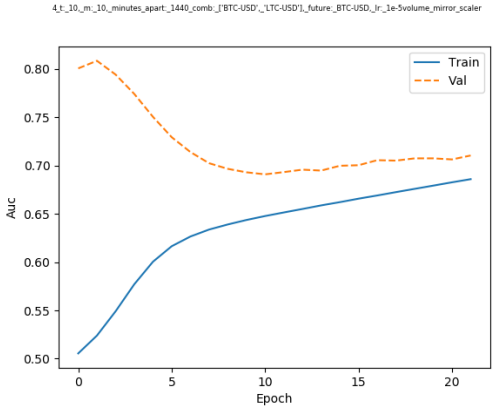
The avid reader will notice, that figure 12.4a is in the group of figures where the AUC is over 70%, although the validation AUC is less than 30%. This however means, that the model is wrong more often than coincidental. This model has learned patterns, however flips positives with negatives - which means the model still learned patterns, and if its outputs were flipped, the AUC would be more than 70%.



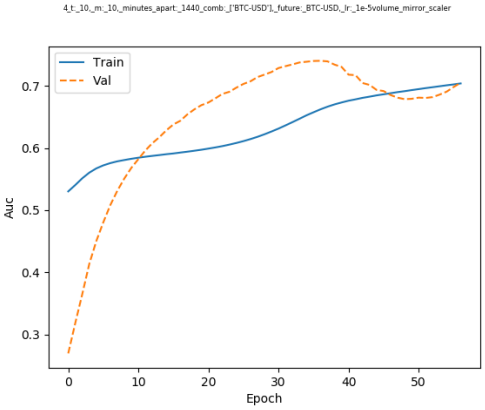
(a) Interval: 16h, Input Length: 10



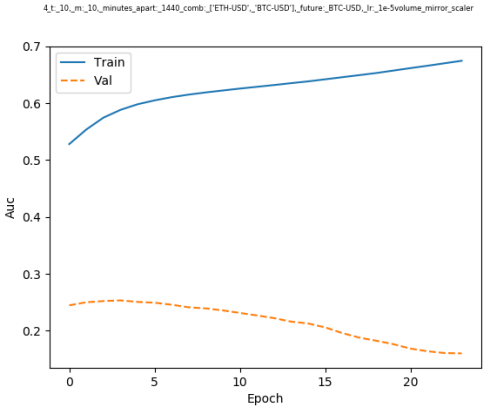
(b) Interval: 16h, Input Length: 10



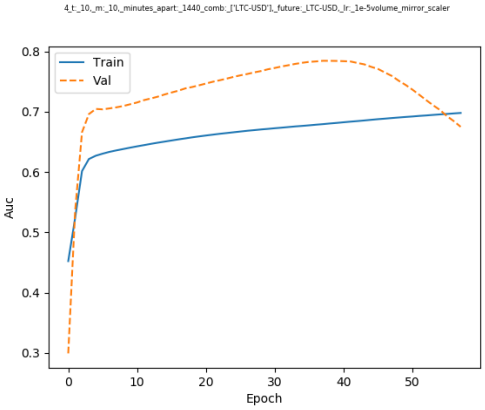
(c) Interval 24h, Input Length: 10



(d) Interval 24h, Input Length: 10

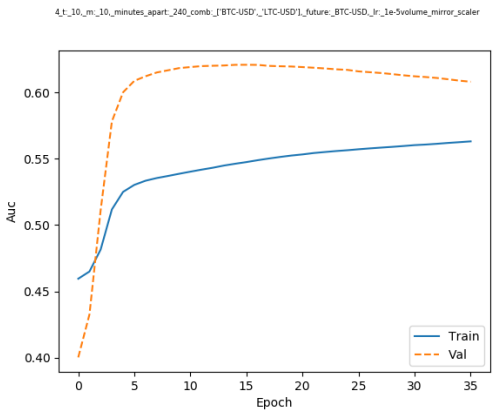


(e) Interval 24h, Input Length: 10

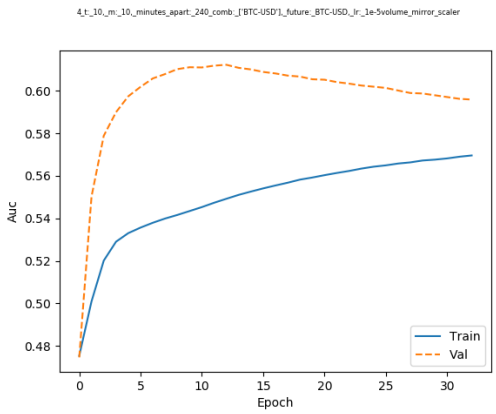


(f) Interval 24h, Input Length: 10

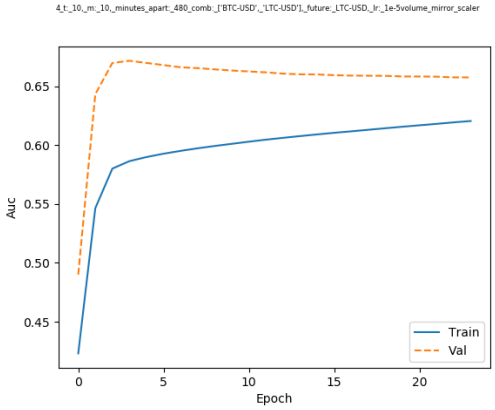
Figure 12.4: Validation AUC above 70%



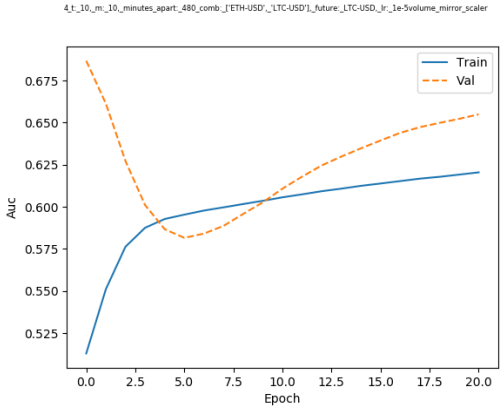
(a) Interval 4h, Input Length: 10



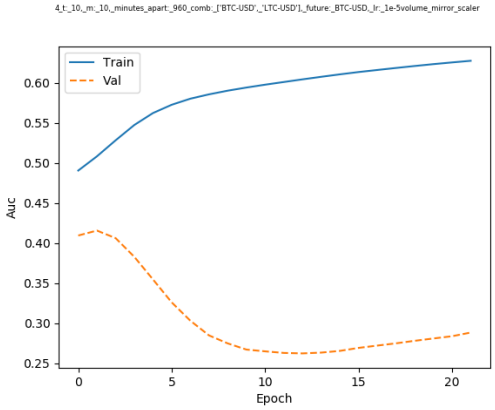
(b) Interval 4h, Input Length: 10



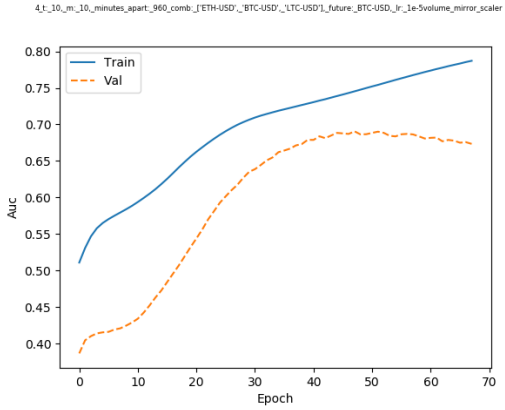
(c) Interval 8h, Input Length: 10



(d) Interval 8h, Input Length: 10

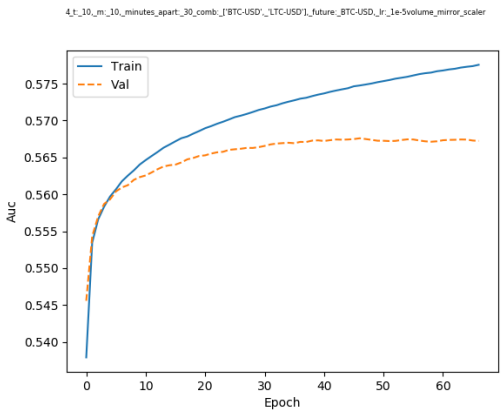


(e) Interval 16h, Input Length: 10

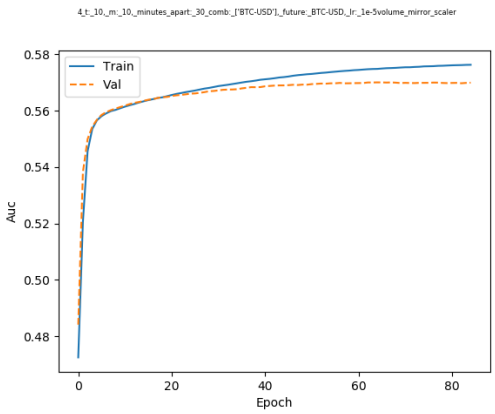


(f) Interval 16h, Input Length: 10

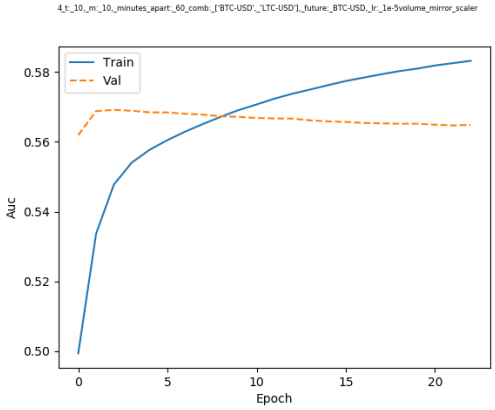
Figure 12.5: Validation AUC between 60% and 70%



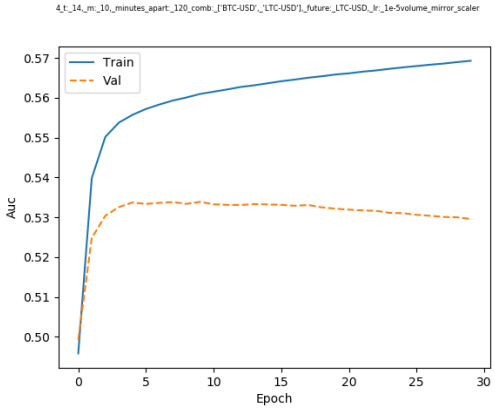
(a) Interval 30min, Input Length: 10



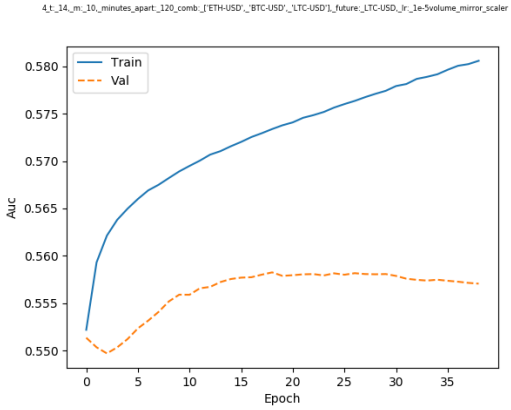
(b) Interval 30min, Input Length: 10



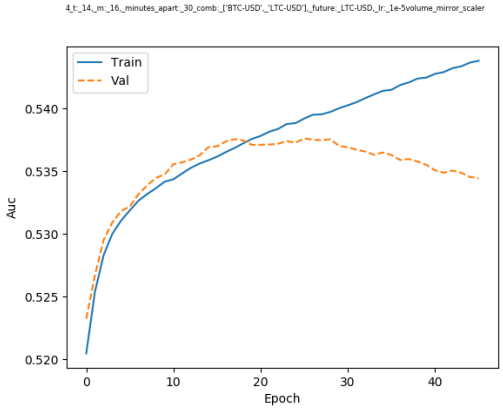
(c) Interval 1h, Input Length: 10



(d) Interval 2h, Input Length: 14



(e) Interval 2h, Input Length: 14



(f) Interval 30min, Input Length: 14

Figure 12.6: Validation AUC below 60%

Part III

Conclusion

Chapter 13

Summary

Amongst other problems, time was the main limited resource. A finished tool as a service could not be implemented in the scope of this thesis. Whilst there were many setbacks in earlier parts of this thesis (for example the hypothesis that segments of financial data could be gathered to groups of similar patterns turned out to be false), final approaches of making predictions based on past financial data are able to make predictions with accuracies of up to 70%. The final result of the research done in this thesis is that financial data of bigger time frames is more expressive and therefore more predictable than short term data.

Different Implementations for Preparing Data Throughout this thesis a lot of effort went into preparing the data for machine learning models, and in the process of implementing different algorithms most of the time went into making these methods perform better and execute quicker. With each iteration new Numpy functions were used to make executions faster. In the earliest version Numpy was not used at all for matrix operations, which turned out to be a great mistake, because this means that these operations had to be done in Python for loops, which are slow. The final version of the data preparation is around 20 times faster than early versions.

Time spent on Learning Deep Learning At the beginning of this thesis, machine learning was a completely new ground for both authors. The width and depth of the topic as well as its sheer complexity were underestimated. A great deal of time went into truly understand how neural networks work and how to properly research and analyze data to get to great results. Although this was not planned, the scope turned out to be *just* finding ways of properly predicting financial instruments, and unfortunately the outcome of this research was not as good as expected.

Insufficient Data Next to time, another big problem in the course of this thesis was getting enough data for deep learning to properly work. Even though a dataset with 2 months of one-minute data was found, this data was outdated, because at the time of writing this thesis the data is already one year and a half old. Initially the plan was not

to work with cryptocurrencies but instead the focus was planned to be on stock indices, like the DAX, and on the foreign exchange market.

13.1 Further Work

The work of this thesis can be extended and built on in numerous ways. This section aims to show a few of the ways on how the work could be improved and taken in further directions.

Collecting Live Data The problem of insufficient data could be solved by using the live data API of, for example Alpha Vantage. Live data for financial instruments could be queried periodically and subsequently saved to a database. If this was done long enough, eventually this data could be used to train the models.

Further Research Further testing with recurrent neural networks could be done. The most interesting aspect to look at is how deeper (recurrent networks with more than one hidden layer) would perform.

Deploying the models in Web Services Once models are found that predict with reasonable accuracy, a backend service would be created. Tensorflow offers libraries to easily deploy the models and use them in web services. For this to efficiently work, one of the libraries is Tensorflow Transform, with this library the data preparation algorithms can be implemented to work in a data pipeline which automatically transforms the data if a web service to make a prediction is called.

Continuous Predictions The models could make live predictions with the available live data, which users can always use as a tool to support the decisions of whether or not to enter a trade or buy an instrument.

User Interface Originally the plan was to offer a mobile application for iOS and Android, as well as a web client. The apps could include functionality to let the user configure to be notified when certain predictions are made. The main use of the interfaces would however be showing the user the live predictions.

Bibliography

[1] 12. Virtual Environments and Packages. URL: https://docs.python.org/3/tutorial/venv.html.

[2] Data Science: Deep Learning in Python. URL: https://www.udemy.com/course/data-science-deep-learning-in-python/.

[3] Deep Learning Prerequisites: Logistic Regression in Python. URL: https://www.udemy.com/course/data-science-logistic-regression-in-python/.

[4] Deep Learning: Recurrent Neural Networks in Python. URL: https://www.udemy.com/course/deep-learning-recurrent-neural-networks-in-python/.

[5] Learn Linear Regression in Python: Deep Learning Basics. URL: https://www.udemy.com/course/data-science-linear-regression-in-python/.

[6] Noisy data. URL: https://en.wikipedia.org/wiki/Noisy_data#cite_note-sba-1.

[7] pip - The Python Package Installer¶. URL: https://pip.pypa.io/en/stable/.

[8] Python 3.8.2 documentation. URL: https://docs.python.org/3/.

[9] Python Virtual Environments: A Primer. URL: https://realpython.com/python-virtual-environments-a-primer/#why-the-need-for-virtual-environments.

[10] Ameer Rosic. What is Cryptocurrency? [Everything You Need To Know!]. URL: https://blockgeeks.com/guides/what-is-cryptocurrency/.

[11] Amirsina Torfi. Python Machine Learning. URL: https://machine-learning-course.readthedocs.io/en/latest/content/unsupervised/clustering.html.

[12] Anas Al-Masri. How Does k-Means Clustering in Machine Learning Work? URL: https://towardsdatascience.com/how-does-k-means-clustering-in-machine-learning-work-fdaaaf5acfa0.

[13] Andrew Luashchuk. Why I Think Python is Perfect for Machine Learning and Artificial Intelligence. URL: https://towardsdatascience.com/8-reasons-why-python-is-good-for-artificial-intelligence-and-machine-learning-4a23f6bed2e6.

[14] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly, 2019.

[15] Daffodil Software. 9 Applications of Machine Learning from Day-to-Day Life. URL: https://medium.com/app-affairs/9-applications-of-machine-learning-from-day-to-day-life-112a47a429d0.

[16] Daniel Faggella. What is Machine Learning? URL: https://emerj.com/ai-glossary-terms/what-is-machine-learning/.

[17] David Robinson. The Incredible Growth of Python. URL: https://stackoverflow.blog/2017/09/06/incredible-growth-python/.

[18] eToro. eToro Social Trading. Library Catalog: www.etoro.com. URL: https://www.etoro.com/trading/social/.

[19] Greg Cope. Dummy Variable Trap in Regression Models. URL: https://www.algosome.com/articles/dummy-variable-trap-regression.html.

[20] Guru99. Machine Learning Tutorial for Beginners. URL: https://www.guru99.com/machine-learning-tutorial.html#1.

[21] Harrison. Cryptocurrency Training Dataset. URL: https://pythonprogramming.net/cryptocurrency-recurrent-neural-network-deep-learning-python-tensorflow-keras/.

[22] Himani Bansal. Best Languages For Machine Learning in 2020! URL: https://becominghuman.ai/best-languages-for-machine-learning-in-2020-6034732dd242.

[23] Hunter Heidenreich. What are the types of machine learning? URL: https://towardsdatascience.com/what-are-the-types-of-machine-learning-e2b9e5d1756f.

[24] IG. Online trading courses. URL: https://www.ig.com/en/learn-to-trade/ig-academy/courses.

[25] Jason Brownlee. Data, Learning and Modeling. URL: https://machinelearningmastery.com/data-learning-and-modeling/.

[26] Jetbrains. Python. URL: https://www.jetbrains.com/lp/devecosystem-2019/python/.

[27] Jonathan Helmus. Understanding Conda and Pip . URL: https://www.anaconda.com/understanding-conda-and-pip/.

[28] Josh Starmer. ROC and AUC, Clearly Explained. URL: https://www.youtube.com/watch?v=4jRBRDbJemM.

[29] Laurence Bradford. What Should I Learn as a Beginner: Python 2 or Python 3? URL: https://learntocodewith.me/programming/python/python-2-vs-python-3/.

[30] Matthew Opala. Major Challenges for Machine Learning Projects. URL: https://www.topbots.com/major-challenges-machine-learning-projects/.

[31] Sarang Narkhede. Understanding AUC - ROC Curve, May 2019. URL: https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5.

[32] Nick Coghlan. PEP 440 – Version Identification and Dependency Specification. URL: https://www.python.org/dev/peps/pep-0440/#version-specifiers.

[33] Pulkit Sharma. The Most Comprehensive Guide to K-Means Clustering You'll Ever Need. URL: https://www.analyticsvidhya.com/blog/2019/08/comprehensive-guide-k-means-clustering/.

[34] Telusko. is Python Compiled or Interpreted Language? URL: https://www.youtube.com/watch?v=0BhSWyDEDC4.

[35] Vaishali Advani. Artificial Intelligence vs. Machine Learning vs. Deep Learning. URL: https://www.datasciencecentral.com/profiles/blogs/artificial-intelligence-vs-machine-learning-vs-deep-learning.

[36] Venkatesan M. What is Machine Learning? How Machine Learning Works and the Future of Machine Learning? URL: https://www.mygreatlearning.com/blog/what-is-machine-learning/.

[37] Victor Roman. How To Develop a Machine Learning Model From Scratch. URL: https://towardsdatascience.com/machine-learning-general-process-8f1b510bd8af.

[38] Wikipedia. Euclidean distance. URL: https://en.wikipedia.org/wiki/Euclidean_distance.

[39] Will Kenton. Tick. URL: https://www.investopedia.com/terms/t/tick.asp.

[40] Yogesh Chaurse. Is Python compiled or interpreted or both ? URL: https://www.quora.com/Is-Python-compiled-or-interpreted-or-both.

[41] Yufeng G. The 7 Steps of Machine Learning. URL: https://towardsdatascience.com/the-7-steps-of-machine-learning-2877d7e5548e.

List of Figures

Appendix A

Individual Goals

A.1 Leon Schlömmer

- Research how chart data can be modelled to be an input to neural networks
- Evaluate which deep learning models work best for predicting financial data

A.2 Lukas Stransky

- Determine if correlation can be used to detect when a given pattern is forming
- Evaluate whether machine learning and neural networks are capable of identifying when a particular pattern is formed.

Appendix B

Apportionment of Work in Writing this Thesis

B.1 Leon Schlömmer

- Abstract
- Introduction, 1
- The Data, 3
- Tensorflow and Keras, 4.9
- Linear Regression, 6.1
- Logistic Regression, 6.2
- Deep Learning, 7
- Sequence Analysis, 8
- Data Preparation, 10
- First Implementation of Supervised Techniques, 11
- A Big Evaluation of Recurrent Neural Networks and Hyperparameters, 12
- Summary, 13

B.2 Lukas Stransky

- Introduction to Stocks and Trading, 2
- Introduction to Machine Learning, 4

- Introduction to the Python Environment, 5
- Cluster Analysis - Unsupervised Learning, 6.3
- Implementing a K-Means Clustering Algorithm and a Neural Network, 9