

Trump Tweets vs The Markets

Final Report for CS39440 Major Project

Author: Mateusz Stankiewicz (mas15@aber.ac.uk)
Supervisor: Dr. Neil Mac Parthaláin (ncm@aber.ac.uk)

02th May 2018

Version 1.0 (Release)

This report is submitted as partial fulfilment of a BSc degree in
Computer Science (G401)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Quality and Records Office (AQRO) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work, I understand and agree to abide by the University's regulations governing these issues.

Name Mateusz Stankiewicz

Date 03.05.2018

Consent to share this work

By including my name below, I hereby agree to this dissertation being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name

Date

Acknowledgements

I'd like to thank StackOverflow community for my Degree.

Abstract

"World events often have a great influence over international markets. Political uncertainty can often drive commodities up or down in value depending on where it occurs in the world. " [1]. Politicians of countries with the biggest markets have got a strong impact on the value of currencies and trading commodities. A simple message such as "With Mexico being one of the highest crime Nations in the world, we must have THE WALL. Mexico will pay for it..." [2] sent from the USA president's account can drop down Mexican Peso value. Trump's infamous Twitter account, which is followed by almost 50 million people, can be an effective tool to influence the markets.

The goal of this project is to develop a system which considers the sentiment of tweets and can predict whether a stock index will increase or decrease depending on the current index, words, phrases and the sentiment of the tweet.

Python, SciKit-learn, and NLTK (Natural Language Toolkit) are used to process the data in this project. The web interface is created using Flask framework.

The results show that when using a Naïve Bayes classifier, the accuracy of predicting the USD Index change is 53.7%, whereas the base rate of the three-class problem (up, down, no change) is 41%.

Contents

CONTENTS	5
1. BACKGROUND, ANALYSIS & PROCESS	6
1.1. BACKGROUND	6
1.1.1. <i>Machine Learning and Data Mining</i>	6
1.1.2. <i>Trump Tweets</i>	6
1.1.3. <i>Similar Systems</i>	7
1.2. ANALYSIS	7
1.3. PROPOSED TASKS	8
1.4. PROCESS	9
2. DESIGN	10
2.1. OVERALL ARCHITECTURE	10
2.1.1. <i>Programming Language</i>	10
2.1.2. <i>Libraries</i>	10
2.1.3. <i>Data storage</i>	11
2.1.4. <i>Interface</i>	12
2.2. DETAILED DESIGN	13
2.2.1. <i>Markets package</i>	13
2.2.2. <i>Webpage package</i>	14
2.2.3. <i>Others</i>	15
2.3. TOOLS USED TO DEVELOP THE PROJECT	15
3. IMPLEMENTATION	16
3.1. DATA GATHERING	16
3.1.1. <i>Tweet scraping</i>	16
3.1.2. <i>Stock value data</i>	16
3.2. SENTIMENT ANALYSIS AND PHRASE EXTRACTION	16
3.2.1. <i>Data selection</i>	16
3.2.2. <i>Feature extraction</i>	18
3.3. MARKETS PREDICTING	21
3.3.1. <i>Data Pre-processing</i>	21
3.3.2. <i>Building initial model</i>	22
3.3.3. <i>Removing infrequent features</i>	23
3.3.4. <i>Feature selection</i>	23
3.3.5. <i>Change of Decision Modelling</i>	24
3.3.6. <i>Removing useless instances</i>	25
3.3.7. <i>Remarking features after sifting</i>	25
3.3.8. <i>Adding more stocks</i>	26
3.3.9. <i>Running WEKA feature selection from Python</i>	27
3.3.10. <i>Using two models to predict changes</i>	28
3.4. RULES LEARNING	29
3.4.1. <i>Final workflow</i>	31
3.5. FLASK WEBPAGE	32
3.5.1. <i>Application initialization</i>	32
3.5.2. <i>Views and templates</i>	32
4. TESTING	33
4.1. UNIT TESTING	33
4.2. INTEGRATION/ACCEPTANCE TESTING	34
4.3. USABILITY TESTING	35
5. CRITICAL EVALUATION	36
APPENDICES	38

A. THIRD-PARTY CODE AND LIBRARIES	38
B. ETHICS SUBMISSION	39

1. Background, Analysis & Process

1.1. Background

1.1.1. Machine Learning and Data Mining

Machine learning and Data Mining are becoming very quickly developing fields of computer science nowadays. The Internet is full of data that can be analysed and processed. The process of data mining is mainly focused on discovering patterns in large data sets involving methods such as machine learning and statistics. [3]

“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.” [4]
In other words, it is a practice of feeding data into algorithms that learn patterns from it and are able to predict the output for unseen data.

One branch of Machine Learning is Supervised learning. Starting from the labelled dataset (objects with labelled output) supervised algorithms produce a function to make predictions about the output values. The key part of supervised learning is that outputs of the training data have to be known in advance. [5]
Instances of that data have to have it set up beforehand or be labelled manually.

1.1.2. Trump Tweets

Twitter is a social networking website on which users can post short (max 140 characters length) messages called “tweets”. As of the October 2017, Twitter had 330 million users [6]. Among users, there can be found many celebrities and politicians, who regularly post their thoughts, news and announcements.

US president, Donald J. Trump is known for his controversial, outrageous and sometimes very hateful tweets. He posts about 10 times every day and his account is followed by over 50 million people [7]. His account is regularly used to attack his opponents and stir up controversies.

His tweets have got a massive impact on markets and political scene. Their scope is so wide that he admitted that without Twitter he would not be a president. [8]

His simple language makes them easy to analyse. He never uses a sarcasm or exquisite words. Due to the fact that his vocabulary is fairly small, it is easy to train a machine learning classifier with it. Such a classifier learned using these tweets associated with for example market changes can be able to predict future events when fed with new data.

His tweets are also very emotional, petulant and sometimes aggressive what makes it easy to determine their sentiment by a computer. They also follow some patterns, most of his positive tweets end up with “Make America Great Again” phrase.

1.1.3. Similar Systems

In preparation for this project, there was done some research. There was found an interesting analysis made by David Robinson (Chief Data Scientist at DataCamp), [9] comparing Trump's tweets content with the device they were sent from (part of tweets is sent from iPhone and some from Android). The results were interesting: most negative tweets attacking his rivals were sent from Android whereas iPhone was used more for benign announcements. The analysis concludes that tweets from these devices are written by different people. Almost all the tweets sent with a picture or hashtags come from iPhone and most of "emotionally charged" words were common for Android device. What we can notice from the analysis is that iPhone tweets are probably sent by people involved in planning his schedule because words like "join" or "tomorrow" come from iPhone. Fact that Android tweets are more objective may mean that these ones are sent by his public relations specialists or either by himself. Research focused more on analysing only the metadata so probably the broader analysis can bring out more interesting information.

There have been many researches about predicting stock market indicators through twitter made already. One of them is research made by Xue Zhang [10], that was based on collecting tweets about stock markets and analyse on measured hope and fear on each day. The data was used to find a correlation with markets. It was found that emotional tweets correlated to the biggest markets such as Dow Jones, NASDAQ and S&P 500.

Prior to starting with the project there was watched many tutorials on Youtube and Pluralsight. A similar project was developed by Harrison Kinsley, whose tutorials can be found on Youtube and his webpage: "pytohnprogramming.net" [11] Tutorials show the basics of NLP (Natural Language Processing) and Twitter Sentiment Analysis. There was shown simple approach to create a live graph of tweets sentiment. The project used *NLTK*, *Tweepy* and *Matplotlib*. Despite the fact that the project has some substantive errors, the overall approach and process are sensible.

One of the major parts of the project was phrase extraction. During the research, there were identified some modules providing this functionally. One of them was a python implementation of RAKE (Rapid Automatic Keyword Extraction) [12] algorithm.

It was used served as an outline to write own module performing a feature extraction, what is described in more detail in the paragraph 3.2.2. Feature Extraction.

1.2. Analysis

Most of the Twitter sentiment analysis experiments available on-line use ready-made analysers like the one built-in in the *TextBlob*. This approach is fast and easy but those classifiers are too general. It was assumed that it is better to build own classifier because peoples languages are different and one people words can have different emotions when spoken by someone else. For example, words like "Mexico" are neutral but in Trump's tweets, they usually have a negative attitude. Another example could be "Make America Great Again" which in all of the cases goes in positive tweets.

The initial idea of the program workflow looked as follows:

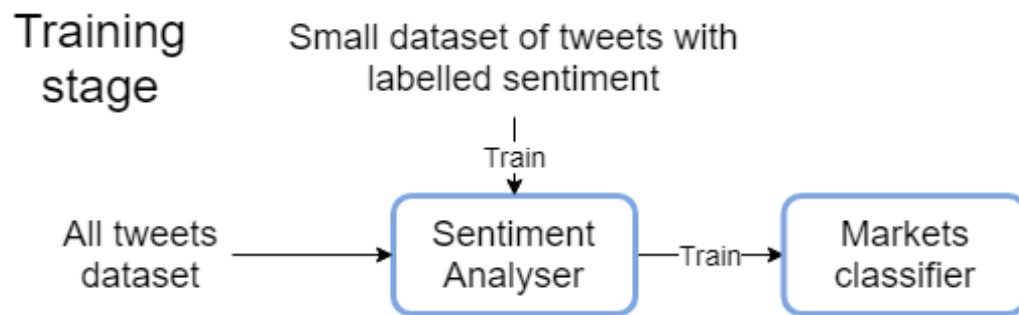


Figure 0-1. Draft of a training process

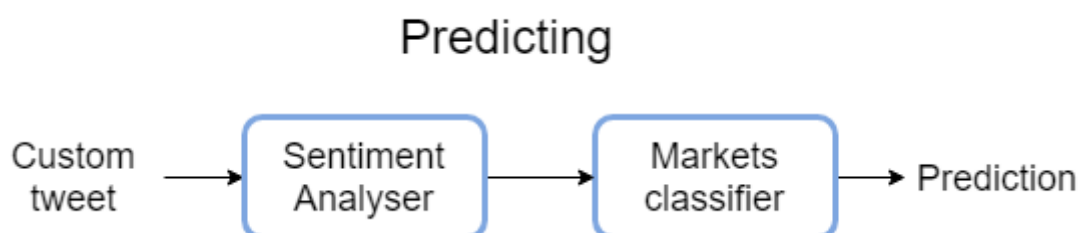


Figure 0-2. Draft of a tweet predicting process

1.3. Proposed tasks

Tasks proposed in OPS are presented as follows:

Setting up git repository - Creating a private git repository, downloading libraries, starting the design of the project.

Tweets sentiment analysis – Investigating which framework or library use to do the sentiment analysis (probably NLTK).

Scraping tweets and U.S. Dollar value and selection – Investigating how to retrieve tweets from Twitter REST API. Twitter official docs provide many libraries made by people such as “*python-twitter*” or “*tweepy*” [13]. Investigating how to store scraped tweets. Selection of tweets and investigating how to scrape currencies values. Investigating how to store tweets and currencies data. One option is to use CSV file format or simple database such as *Sqlite3*, so that data can be presented easily later on.

Window size – Investigating how long should be the window size, for how long do the tweets influence the market.

Building a model that predicts USD changes depending on tweets – Building a model that can predict if currency will drop down or go up depending on a tweet sentiment. Investigating how to do that, for example, use ANNs or Naïve Bayes.

Creating an API – Creating a webpage presenting graphs and charts of currency values, which tweets affect the currency changes, adding an option for a user to provide own tweet contents and presenting what effect would it have. Investigating which framework to use to create an API. To create a webpage there will be used probably *Django* with *Bootstrap* and some JavaScript library to create charts, i.e. *ChartJS*.

Project Meetings and Project Diary – The project will involve half an hour meeting every week with a supervisor on Mondays at 2 pm. A project diary will be kept to remember what was done and all the tasks will be documented.

Tasks to do when there will be spare time:

Adding other currencies – Adding more currencies to be analysed by the program, such as Mexican Peso or Russian Rubel.

Adding other commodities – Adding gold or fuels to be analysed by the program.

Adding option to use any Twitter account to build a classifier – Adding a functionality to choose other twitter accounts than Donald Trump's one. For example, Theresa May can have an influence on the market due to the Brexit.

1.4. Process

Deciding about the development process was obvious. It was decided to use agile methodologies that ensure high product quality, end product satisfaction and reduces the risk of failure with delivering a project.

The project has been split into sprints what allows to deliver new features quickly and frequently. We are sure that if there was too little time then we would still be able to deliver some working software. With waterfall methodology, it would be very likely that there will not be enough time to do tests because implementing all of the features took too much time.

The agile approach that fit the project specification the best is Scrum. Weekly meetings with a supervisor can be treated as client meetings during which a further direction is determined. Each week can be treated as a sprint within which there should be planned adding a new small functionality to the project. Therefore, a student can be seen as a Scrum master and development team whereas a supervisor can be treated as a product owner.

The project was split into sprints and at the end, they looked as follows:

Tasks for each week	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12
Setting up Git, IDE, research												
Data collection												
Sentiment analysis												
Phrase extraction												
Markets model building												
Feature selection												
Creating a webpage												
Adding new currencies												
Integration tests												
Writing a report												

After each completed sprint there was a retrospective with a supervisor where was reviewed a progress and were discussed next task for the next sprint.

Alternative for Scrum that was considered for the project was XP (Extreme Programming). XP was rejected because its aspects such as pair programming or daily stand up meetings would be rather more efficient in teamwork than in single person project. There was also no need to use any Continuous Integration tool, the tests were run manually in the meantime.

2. Design

2.1. Overall Architecture

2.1.1. Programming Language

When choosing a programming language, the main criteria were usability for the project, its libraries, ease of use and experience. The language should provide:

- Machine learning and natural language processing libraries
- Simple Web framework
- Ease of experimenting
- Simple tools to gather the data from web APIs
- Simple tools to do data manipulations and analysis

In this project Python was the best choice. It provides very popular and in-depth machine learning libraries (*NLTK*, *SciKit-learn*, *Textblob*), web frameworks (*Flask*, *Django*, *Pyramid*) and Interpreter which is useful to do quick, ad hoc experimenting. It also provides a *Pandas* library that allows forming data into *DataFrames* that is very handy in data analysis and provides many built-in data processing functions [14]. I also used this language during my Industrial Year. Python is also very popular, has got good documentation and there are great tutorials and resources available on-line. Most of the researchers and projects that were review during background research were done in Python. All of them used *Pandas DataFrames*, *NLTK* and *SciKit-learn*.

Other languages that were taken considered:

- **R** - has good ML and data processing libraries. Language is used mostly for data processing and data analysis, therefore, has not got any libraries providing a way to create a web interface. Choosing this language would require to learn it from scratch.
- **Java** – Provides many web frameworks such as *Spring*, *JSF* or *Vaadin*. If the project was done in Java there could be also used *WEKA* (*Waikato Environment for Knowledge Analysis* [15]) that is a very popular software suite. Writing a code in Java would not be so quick as in Python due to the language syntax. Executing any algorithms in Java would also take more time what would slow down experimenting.
- **Ruby** – provides a good web framework (*Ruby on Rails*) but does not provide any good libraries/gems to do ML. There is *WEKA* for *JRuby* and there also other ways to use Java *WEKA* library in Ruby. Was rejected due to the lack of ML tools and lack of experience in coding in this language.

Moreover, choosing Python is a good opportunity to develop the skills that are used in the industry and meet the needs of the labour market.

In this project, the most recent Python version was used (v3.6).

2.1.2. Libraries

To do natural language processing, firstly the *Textblob* library was used. This is built on top of *NLTK* but unfortunately, due to the poor documentation and lack of its capabilities, a decision was made to move into *NLTK*.

To build a classifier learner which predicts market changes *SciKit-learn* was used. It has good documentation with many useful examples and there are also many topics about it on sites such as *StackOverflow*.

Another possibility was to use *WEKA* but it would require to use either *Jython* (implementation of the Python language for the Java platform [16]) or run *WEKA* library using wrappers around JNI calls such as *jvavrbridge* (a package that allows Python to interact with the JVM [17]). However, both approaches are complicated. Finally, *WEKA* was used anyway to do features selection as a command in a separate subprocess.

The web interface of the program is simple, so I chose *Flask*, which is a micro web framework using the Jinja2 template engine. It allows the creation of simple pages in the very straightforward manner. I did not use *Django* as it was planned while doing OPS because it is better suited for more complex web applications. *Pyramid* framework is also good for creating simple web pages but is much less popular, what makes it harder to find some solutions and helpful articles on-line.

To present the results on there was used common technologies such as HTML, CSS, Javascript. To create neat and decent graphs it was decided to use *ChartJS* library. It allows creating HTML5 charts in a simple way.

To do the data processing and analysing there were used *Pandas* and *NumPy* which are the most popular Python modules. They are used in similar projects and most of the machine learning online tutorials, so it was assumed that it is a good choice. Operations on *Panda's DataFrames* and *NumPy* arrays are much faster than on regular arrays and other language built-in collections. [18]

SciKit-learn does not provide any association rules algorithms so there had to be used another library. It was decided to use *Mlxtend* library [19] and its *Apriori* algorithm implementation.

To scrape tweets from Twitter there was used *Tweepy* library. It is popular, easy to use module that allows accessing Twitter API. Another alternative would be to scrape the Twitter website using HTML scraping tools such as *Beautiful Soup* [20]. However, modules concretely to use with Twitter have been already available.

To write unit test there was used built-in unittest model, and for integration testing it was decided to use *Behave* [21]. The most popular integration testing framework *Cucumber* does not support Python. *Behave* seemed to be the best choice. Other alternatives that were identified are *Lettuce* and *Aloe*. The first has been abandoned a long time ago and does not support Python 3 [22]. The second one is a Python 3 port of *Lettuce* but also has not been developed for 3 years [23] and has got very short and poor documentation.

2.1.3. Data storage

All the stocks historical data available on the Internet is usually in CSV (comma-separated values) format. CSV is also easy-understandable, simple and well supported by Python. Reading and writing to CSV files in Python and using *Pandas* is fairly simple. Therefore, it was decided to store all processed data in that format.

The data about the analysed stocks that is presented on the webpage had to be stored somewhere. It was decided to use small database than JSON or just plain text files because is simpler and quicker to implement. ORM (Object relational mapping) handles saving/reading data, verifying the format and checking if the currency is already analysed, what is more convenient.

For this purpose, was used *Flask-SQLAlchemy* module that is a *Flask* extension adding support for *SQLAlchemy* [24]. (*SQLAlchemy* is a simple object-relational mapper [25]). The database system that has been used is *SQLite3*.

2.1.4. Interface

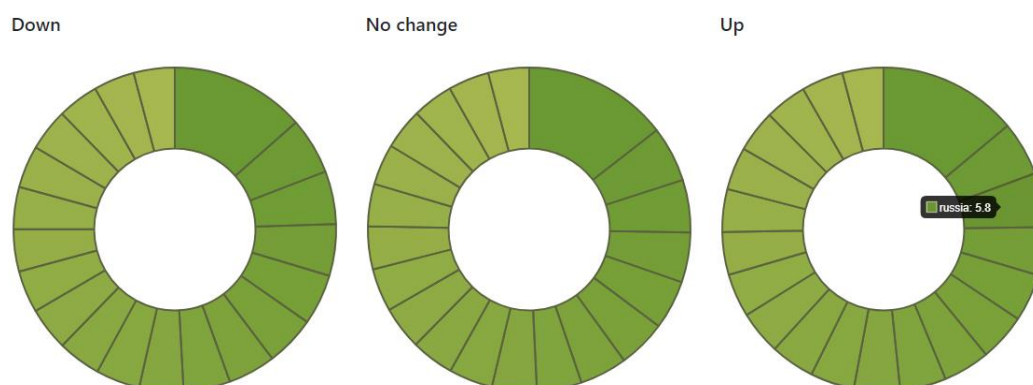
The web interface created in *Flask* presents details about each stock on a separate subpage.



The graph presents the currency price change and when hovering over, the tooltip shows all tweets that were written on a particular day.



The doughnut charts below present 20 the most coefficient features for each class.



The table below presents association rules learned from the dataset used to build a classifier. That means that these are only rules extracted from the sifted dataset and they present only features that appear together (all frequent and infrequent words that have not been included in features vector are omitted). Each row is a set of features that occurred together at least twice and when the row is clicked there are presented rules for each particular rule consisting these words.

Words set	support		confidence		
daca, drug, military	0.00255754		1.0		
	antecedants	consequents	antecedant support	consequent support	lift
	daca, drug	military	0.00255754	0.06479113	15.43421053
	military, drug	daca	0.00255754	0.03239557	30.86842105

2.2. Detailed Design

The project can be split into main two parts: market analysis and web interface. Therefore, it was split into three packages separating project concerns:

- Markets package
- Webpage package
- Tests package

2.2.1. Markets package

All of the code in the “markets” package was split into modules that group together logically related code.

Dataset – a module with a TweetsDataSet class that wraps around a Pandas DataFrame and represents a set of tweets, their features, sentiments, and market effect.

Tweeter Scraping – a module used to scrape tweets from Twitter.

Phrase Extraction – contains a PhrasesExtractor class which builds a vocabulary of phrases and words found in the set of texts and then extracts those features from particular tweets.

Sentiment Analysis – module containing a class responsible for tweets sentiment analysis. It wraps NaiveBayesClassifier from *NLTK* library and uses a PhrasesExtractor to extract features from tweets which are then used to train a model or predict a value of the particular tweet. All of the functionality was wrapped in a class because it is more convenient to load and save the Analyser and perform any tests.

Tweets Feature Extraction – contains all the functions used to extract features from the tweet such as sentiment and phrases/words in the tweet. To gather that information, it uses SentimentAnalyser and PhrasesExtractor instances.

Feature Selection – module containing functions responsible for selecting an informative subset of features, to obtain the best accuracy.

Currency Analysis – the main module that connects all of the functionalities. There is a CurrencyAnalyser class that is used to analyse a CSV file of stock prices and provides results for this analysis, such as association rules, model to predict markets or the most coefficient features. It holds the functionality of reading files and saving the results.

Association – a module holding all of the code that is responsible for reading stock prices from files and merging them with tweets datasets.

Market Predicting – contains all of the code that is responsible for training a classifier that predicts stock changes. Contains 3 classes: Classifier representing a classifier model (MultinomialNB by default) and wrapping all of its functionality; AnalyseResult that represents a result of a single tweet analysis; MarketPredictingModel that contains two Classifier objects and decides which one to use to do a prediction.

Rules – contains functions used to do association rules learning.

The markets module has got also a “data” and “pickled_models” directories. “data” folder stores all data used to do analysis such as a list of stop words, scrapped tweets and CSV files generated by the application. “pickled_models” stores saved sentiment analysis and market predicting models.

2.2.2. Webpage package

Web page module has got a typical structure for *Flask* projects.

It is split into:

- Static folder – for holding static files such as images or CSS styles
- Templates folder – for holding *Jinja2* templates that are filled with content by views.
- Views.py – this is where routes are defined. It defines routes for each currency and gathers data that is sent to the templates and presented.
- models.py – holds the Currency model. This model stores information about currencies such as its name and accuracy of the model in the database.
- __init__.py – Initializes the application, sets up its configuration and database.

- Currencies.db – stores Currencies models that can be loaded when the application runs.

2.2.3. Others

Apart from these 3 packages, there are a few more files typical for a *Flask* project:

- manage.py – a script used to initialize/drop the database, fill the database with some sample data and to run the webpage.
- requirements.txt – a file containing a list of packages that are used in the project and have to be installed
- README – a file explaining how to run the program

2.3. Tools used to develop the project

For my Python IDE, I chose *PyCharm* made by *JetBrains*. I used it during the Industrial Year and I really liked it. It has got all of the code assistance features such as syntax and error highlighting. It supports *Flask* projects and many file extensions such as HTML, CSS, JS, CSV which I used in a project. It also has got integrated debugger which I used a lot.

To keep track on changes and have a backup of the work I set up a *GitHub* repository. Backups of the work were kept on the *GitHub* repository and two machines I worked on. As a Git client for the machine with Windows OS was chosen *GitKraken* and command line git for the machine with Linux.

Keeping the project both on the *GitHub* server and two machines prevents the loss of the project in the event of a problem with a git or even accidental deletion of the repository.

3. Implementation

3.1. Data gathering

3.1.1. Tweet scraping

The first step was to gather all Donald Trump tweets. For this purpose, I used a Tweepy library that allows the retrieval of data from Twitter REST API. To communicate with the API, it was necessary to create a Twitter developers account and obtain a customer key and access tokens.

The first task to complete was a sentiment analysis of the collected tweets. A small number of tweets were initially collected manually (about 120). Writing a scraping script was very useful at this stage because it allowed the retrieval of particular tweet data from the API using the ID of each tweet. The data such as *tweet id*, *creation date*, and text were saved to a CSV file.

The next step was to scrape all of the tweets since the beginning of 2017. It is slightly before Donald Trump became US president so his tweets had already begun to have some influence on markets. Tweepy provides also a special function to get posts from a users' timeline. Unfortunately, it allows the retrieval of only 200 tweets, so I had to do this sequentially. Twitter also allows only to scrape the last 3,240 user tweets and luckily, there were 2,935 tweets created between 01.2017 and 03.2018.

3.1.2. Stock value data

Obtaining the currency indices data was a difficult part of the project because all the web pages that archive historical stock data provide only daily-interval stock prices and changes due to the value of this data. All of the visited websites provide data with smaller intervals (such as hourly changes) in value. In consequence, all markets changes analysed in the project are daily open-close price changes.

3.2. Sentiment Analysis and Phrase Extraction

3.2.1. Data selection

To begin with a sentiment analysis there were 120 tweets scraped manually and their sentiment was also marked manually. It was decided that half of them had to be positive and half should be negative to have balanced classes to do not bias the classifier. We do not want the classifier to favour the majority class because it may lead to misleading results.

Tweets were selected regardless of the date. The most important aspect was to find ones that are clearly positive or very negative to train the model as best as possible.

Tweets were found using Google by searching phrases such as "Most positive Trump tweets", "The worst Trump tweets". Very useful was the "Trump Twitter archive" website [26], where we can see the most popular keywords in his tweets and search for them. The website presents also the most superlative, most disdaining, and the highest rated tweets.

3.2.2 Building a model

When the dataset was selected, the next step was to build a classifier.

To train a classification model tweets had to be split into folds to do a cross-validation what prevents overfitting and gives more reliable results. Due to the fact that SciKit-learn k-fold functions seemed to be complicated, there was a decision to write a folding function manually. The code was splitting a corpus into k chunks with preservation of stratification (each chunk had half of the tweets positive and half negative).

Building a text classification system with Textblob is very trivial:

```
train_data = [
    ('I love this sandwich.', 'pos'),
    ('this is an amazing place!', 'pos'),
    ('I feel very good about these beers.', 'pos'),
    ("I can't deal with this", 'neg'),
    ('he is my sworn enemy!', 'neg'),
    ('my boss is horrible.', 'neg')
]

cl = NaiveBayesClassifier(train_data)
cl.classify("This is an amazing library!")
```

Figure 3-1. Example of training a classifier with TextBlob [27]

The Classifier object just has to be fed with the list of tuples, and each tuple has to consist of tweet text and marked sentiment.

It was decided to use a Naïve Bayes as a classifier. This is a dominant algorithm used for text classification.

Its main advantage is that it treats each feature independently. Texts are full of noise and meaningless words what. It is also much faster to train than SVMs, Artificial Neural Networks or Decision Trees.

Although it is a very simple algorithm, it often performs pretty well. Text classification is not a hard problem so any linear classifier should produce decent results.

Although, it was worth to try out other classifiers as well, to check if any other would cope with the problem better. Algorithms performance highly depends on the data they work with. One technique is better for a particular problem and other ones can be more suitable for different tasks.

Changing a classifier to try out another one using *TextBlob/NLTK* is very simple, mostly it requires to just import another class and set its parameters. There were tried out other alternatives, but Naïve Bayes proved to be the best as expected.

To ensure that results are reliable the training process was run 40 times with 10-fold cross-validation and the resulting accuracy is a mean from those 40 runs. The results seemed to be surprisingly good. Naïve Bayes had 82% of test accuracy.

Unfortunately, after investigating the most informative features it turned out that the most decisive features were words such as “are”, “there” or even punctuation marks

such as brackets. That meant that the model was overfitted and also instead of making decisions based on words such as “good” or “bad” was using the most common words in the language. Even the sentence contained many negative words like for example: “Crooked Hillary Clinton is the worst (and biggest) loser of all time.” [28] was marked as positive because it just had words “is” and “the”.

3.2.2. Feature extraction

Once it had been discovered that *TextBlob* does not extract phrases properly and that it does not provide any option to change this, an alternative was sought. The solution to this was to use the *NLTK* framework classifier.

The next step was to write a custom feature extraction function that splits tweets into words. To do so, very simple *NLTK* functions were used: *sent_tokenize* and *word_tokenize*. These split the text into sentences and then into words. Afterwards, all of the extracted words had been lowercased because for example “Then” and “then” is the same word. Unfortunately, even though these functions are a part of such a popular and reputable library, they had problems with splitting even simple sentences and words such as “doesn’t” were separated into “doesn” and “t”. Moreover, the model was still making decisions basing on words such as “Did”, “of”, “And”. These senseless words are called stop words. These are words that are common in the language and do not tell us anything about the meaning of the sentence.

Once it had been discovered that *NLTK* tokenizing functions cannot handle extracting words and phrases, a few alternatives investigated such as *NLTK ConllExtractor* and *FastExtractor*. Most of these had problems with splitting the sentence properly. They either extract useless stop words or do not extract half of the important phrases. The only one that does it very well is *TextRazor* – a cloud service providing a deep-learning analysis using their web API [29]. Unfortunately, *TextRazor* is not free so a decision was made to write a custom phrase extractor.

After some research, an easy to implement an algorithm called *RAKE* (*Rapid Automatic Keyword Extraction* [12]) was discovered. There are many implementations on the web but it was decided to write a standalone extractor based on the one written by Alyona Medelyan [30]. This one was chosen because of its simplicity. However, the implementation had to be modified because has not got enough configuration options and has too many redundant functions. Originally *RAKE* extracts also adjusted keywords (ones that include a stop word such as “United States of America”). This functionality was also added but was dropped later on due to the lack of these phrases in the corpus and the risk of leading to unnecessary False Positives.

The algorithm basically extracts candidate phrases by splitting the text by stop words. A list of stop words used in the project was downloaded from the Internet. [31] The candidate phrases are then sorted by their length and number of occurrences. Phrases that are not accepted are split into words.

The accuracy of sentiment analyser while using a custom feature extractor increased to 84% and it is more reliable because the most informative words now are more sensible:

Most Informative Features			
economy = True	pos : neg	=	4.3 : 1.0
fake news = True	neg : pos	=	3.7 : 1.0
great = True	pos : neg	=	3.4 : 1.0
wall = True	neg : pos	=	3.0 : 1.0
high = True	pos : neg	=	3.0 : 1.0
president = True	pos : neg	=	3.0 : 1.0
good = True	pos : neg	=	2.6 : 1.0
united states = True	pos : neg	=	2.3 : 1.0
mexico = True	neg : pos	=	1.9 : 1.0
working = True	pos : neg	=	1.7 : 1.0
china = True	neg : pos	=	1.7 : 1.0
fun = True	pos : neg	=	1.7 : 1.0
heard = True	neg : pos	=	1.7 : 1.0
leaving = True	pos : neg	=	1.7 : 1.0
north korea = True	neg : pos	=	1.7 : 1.0
night = True	pos : neg	=	1.7 : 1.0
nice = True	pos : neg	=	1.7 : 1.0
terrible = True	neg : pos	=	1.7 : 1.0

Figure 3-2. Most informative features in sentiment analysis

The next step to improve the accuracy was to lemmatize words. This means that all inflected words were reduced to the root form (for example “playing, plays, played” into “play”). This time NLTKs “lemmatize_word” function was sufficient. When subjected all of the words to lemmatization, a test accuracy has increased to 86%.

The overall process of building a vocabulary is as follows:

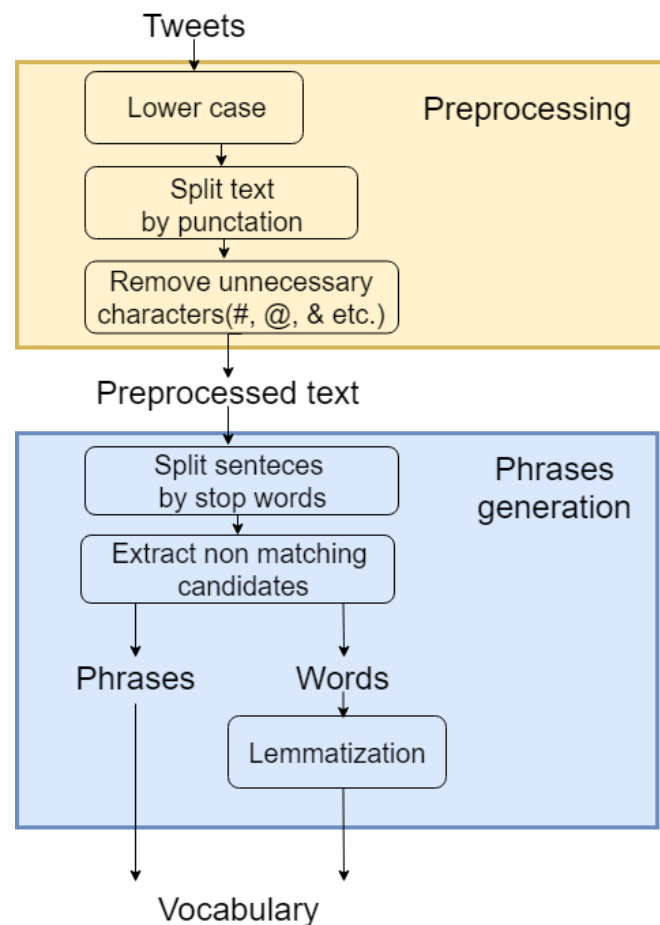


Figure 3-3. The process of building phrase extractor vocabulary

All tweets are firstly pre-processed. All tweets are lowercased and split into sentences. Then from each sentence are removed useless characters such as "@ # - " and each of them is split by stop words:

Very proud of my Executive Order which will allow greatly expanded access and far lower costs for HealthCare. [32]

Is split into:

proud, executive order, greatly expanded access, lower costs, healthcare

This way text is split into phrases and has got useless words removed. Then all of the phrases extracted from the corpus are counted and those that are too rare are split into words and subjected to lemmatization.

The process of training a classifier used to predict a sentiment is as follows:

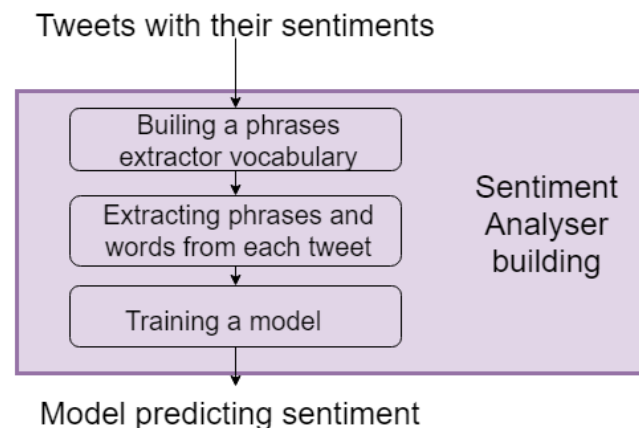


Figure 3-4. The process of building a sentiment analyser

Then the process of analysing a particular tweet is as follows:

- Pre-process tweet (lower case, split by punctuation, remove unnecessary characters)
- Find matching phrases and remove them from a tweet
- Split the rest by stop words to get rid of them
- Lemmatize the words
- Mark words matching with the vocabulary
- Pass the features vector to the model to classify it

Feature extraction from dataset can be run by with "markets/tweets_features_extraction.py" module which executes "build_tweets_features_dataframe" function which reads scraped tweets and saves the result (tweets + features) to a CSV file. All filenames are hardcoded, what would have to be changed if there was added functionality to analyse user-defined Twitter users.

3.3. Markets predicting

Once the sentiment analysis part was done, the next step was to build a market prediction model.

3.3.1. Data Pre-processing

Before training a classifier, the dataset was pre-processed:

- All the tweets containing only video/image and no text were removed.
- Tweets written in languages other than English were also deleted because they would not even be taken into account due to the lack of English words and would make only noise.
- Unicode characters were removed such as ✓✓→
- Some Unicode characters were changed into proper words or characters: & -> "and" because it would make phrase extraction easier. Words like "and" are stop words and are used to split the text into phrases
- All of the links in the tweets were removed

Then pre-processing involved manual removal of all the tweets that could not have any impact such as “Happy birthday”. All short and meaningless tweets such as “Jobs, Jobs, Jobs” were also deleted.

Some of the tweets were “retweets”. It means that Trump posted someone’s tweets on his timeline. They were removed because it is not known what was the purpose of sharing those tweets. It is not known if he endorsed it, wanted to enrage his rivals or maybe shared it to show how abominably the “fake news media” lie.

Another pre-processing step was to merge all tweets that were divided into few separate ones because they were too long (Twitter allows tweets to be at most 140 characters long so when they exceed the limit they are split into few separate ones that start or end with “...”)

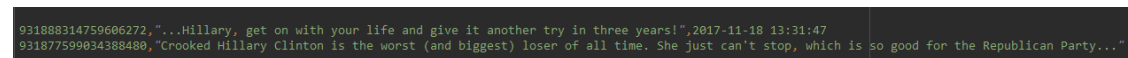


Figure 3-5 Example of a tweet that exceeds 140 characters

Following the removal of irrelevant tweets from the dataset, the number of objects remaining has decreased to 2,026 tweets. All of the tweets got the market change set up using stock prices that were manually collected into CSV file. It was assumed that tweets affect the market within few hours so each tweet was assigned a percent change of the index during the day it was published. It was also decided that all the tweets that are published after 10 pm were an association with the change in value for the next day. Market changes above 0 were marked as positive and the rest as negative thus making this a binary classification.

Each of the tweets was analysed for sentiment and features were extracted.

Tweet	Sentiment	Feature 1	Feat 2	Feat 3	...	Feat n	Change
Tweet content	Positive	1	0	0	...	1	0.05%
Tweet content	Negative	0	1	0	...	0	-0.12%

That prepared dataset could now be used to train a model.

3.3.2. Building initial model

To build a model a *SciKit-learn MultinomialNB* and *LogisticRegressionCV* were used. The first one is a Naïve Bayes classifier implementation that is suitable for classifying discrete values such as “Down”, “No change”, “Up”. Logistic regression classifier used is also known as logit or MaxEnt [33]. It uses a cross-validation to find the best hyperparameters for the data.

The data that is fed to the model has to be in a proper format. Training data has to be split into a 2D array of features and their marks for each instance and a 1D array of results for each instance.

2D array of 2 instances with 5 features:

1	0	1	0	0
0	0	0	1	0
1	1	0	0	0

1D array of targets for 3 instances:

Up	Down	Up
----	------	----

Using a DataFrame to store data was useful because it is easy to process and format it in a such a way.

The model was then built 30 times to take an average of results of different randomized runs. Using a value of 30 gave the same results as 100 so there was no point to run it more times.

Each run trained a model 10 times (10-fold CV) with stratified folds.

The model was built using 2026 objects and over 6000 features and the accuracy that was obtained is as follows:

Classifier	Test accuracy	Train accuracy
Naïve Bayes in SciKit-learn	54.9%	90.6%
Logistic in SciKit-learn	51.6%	100%
J48 in WEKA	51.2%	81.5%

Experiments were performed using a model written in Python and also by exporting a data to the file and using J48 in WEKA.

3.3.3. Removing infrequent features

Feature extraction resulted in 6000 features from the dataset. That was quite a lot of features and the model took a long time to generate during the training phase. The model was also trained using features that occurred only once or twice what just lead to overfitting.

A good set of features describing the data is a key factor to get successful results. Some part of the features was correlated, meaningless and noisy. Therefore, it was necessary to perform a feature selection.

Feature selection is a process of choosing only features that are the most useful or the most relevant to the problem. [34]

It removes redundant attributes what reduces the complexity of the model so it is faster to train and easier to understand (same source as above). Removing unneeded, irrelevant features also increases accuracy.

Running any feature selector on a dataset of this size would result in a very long of processing time, so it was decided to firstly remove features that occur only a few times. After a bit of experimenting it was decided to remove features that occur less than 7 times because it decreased the number of features to more practicable size (1185). It also diminished overfitting and gave a slightly better test accuracy:

Classifier	Test accuracy	Train accuracy
Naïve Bayes in SciKit-learn	56.6%	70.3%
Logistic in SciKit-learn	52.0%	84.0%
J48 in WEKA	51.8%	76.7%

3.3.4. Feature selection

Once the number of features was reduced, it was easier to try different feature selectors. The easiest way was to export the dataset to a CSV file and perform a selection in *WEKA*. Many trials have been carried out with various selectors and

finally the “Wrapper Subset Evaluator” proved to be the best. Wrapper methods try different combinations of features, evaluate them on a model and choose the best set. Their main disadvantage is that they require a significant computation time. However, after all, they give reasonable results.

The “Wrapper Subset Evaluator” selected 116 features that were saved into a file. The file was read during the process of building a model. The resulting accuracy had improved significantly:

Classifier	Test accuracy	Train accuracy
Naïve Bayes in SciKit-learn	67.8%	70.0%
Logistic in SciKit-learn	66.8%	69.2%
J48 in WEKA	62.0%	69.4%

Following that, WEKA was not used to build a classifier due to the fact that uses only one thread, it is extremely slow and it is more convenient to process a data and build a classifier in Python on-the-fly without having to save to file and then opening it in WEKA.

3.3.5. Change of Decision Modelling

Since some of the tweets are completely neutral and have no influence on markets, a third class was added: ‘No-change’. Deciding which tweets did not change the market prices was quite fiddly. We do not want all tweets to be classified as positive or negative because those that are the boundary between positive and negative will spoil prediction through their vagueness.

In the USD Index dataset, there is about 3% of days that the stock did not change at all. To get three classes more balanced there had to be some threshold set up to increase a set of “No change” objects. Following feedback from the project supervisor, it was decided that the threshold should be calculated using a standard deviation. To obtain about 1/3 of the targets as a “no change” the threshold is calculated by 1/3 of the standard deviation distance from the mean:

```
def calculate_thresholds(stock_prices):
    mean = stock_prices.mean()
    sigma = stock_prices.std(ddof=0)
    lower_threshold = (mean - (sigma / 3)).round(2)
    higher_threshold = (mean + (sigma / 3)).round(2)
    return lower_threshold, higher_threshold
```

Figure 3-6. A snippet of function calculating a threshold

It makes all three classes more balanced what seems to be the most reasonable solution in this case. Class distributions got almost balanced and the base rate accuracy was changed to 41%.

Classifier	Test accuracy	Train accuracy
Naïve Bayes	52.7%	59.7%
Logistic regression	49.1%	54.0%

The accuracy looks very good. 52.7% compared to 41% in the 3-class problem is quite satisfying and proves that stock prices are related to Trump's tweets in some degree.

3.3.6. Removing useless instances

After some investigation, it was discovered that if a lot of features is removed then there are many objects left without any feature marked. For example, if tweet "Happy Birthday" had marked "happy" and "birthday" and they were deleted then the instance got useless. Those instances have got only one feature that is a sentiment, what can just spoil training a model.

After removing tweets that do not have any feature the dataset size decreased into 1084 objects. In consequence, the model trained on that dataset gave better accuracy:

Classifier	Test accuracy	Train accuracy
Naïve Bayes	53.6%	60.3%
Logistic regression	51.0%	55.4%

3.3.7. Remarking features after sifting

The next step was to mark features again. Due to the fact that the feature extractor obtains features in less granular way, words are not extracted from found phrases. Therefore, if "Crooked Hillary Clinton" is found in the text, then individual words "Crooked", "Hillary" and "Clinton" will not be marked. This way makes phrases more favoured and increases their chances of not being filtered out during selection and also results in a slightly better accuracy (marking everything that was found (in more granular manner) was dropping the accuracy for about 2%).

If some phrases are found in the text then they are removed and words that this phrase consists of are not marked as presented in the example below:

Text	Crooked	Mexico	Reform	Proud	Crooked Hillary Clinton
Crooked Hillary Clinton is the worst loser of all time.	0	0	0	0	1

After deleting "Proud" and "Crooked Hillary Clinton" it is needed to mark features again to mark words that may have been in removed phrases.

Text	Crooked	Mexico	Reform
Crooked Hillary Clinton is the worst loser of all time.	1	0	0

Marking features again resulted in better accuracy and 5 instances less to delete

Classifier	Test accuracy	Train accuracy
Naïve Bayes	53.8%	60.3%
Logistic regression	52.0%	57.4%

The next change that was performed was making a sentiment to be a continuous value. Before the sentiment was marked as 0/1 (positive/negative). Some tweets could be more neutral and some could be evidently emotional. Therefore, the sentiment values have been changed into continuous values from 0 to 1. Although it did not change the results for Naïve Bayes at all, Logistic regression train accuracy changed only by 0.01%. This is probably due to a large number of features and the fact that Naive Bayes treats features independently so changing one feature, and moreover, in a small extent, had no influence on the result.

3.3.8. Adding more stocks

The main purpose of next sprint was to implement functionality to build a model using another CSV files with stock prices. There was downloaded Euro Index that is a ratio of four major currencies (USD, British Pound, Japanese Yen and Swiss Franc) against the Euro. Due to the fact that a lot of Trump tweets is about the wall on the border with Mexico and hurting their markets with taxes [35], another currency that could be affected is Peso. Unfortunately, currency indices are carried out mostly for the largest currencies and markets, and there is not any for Peso. In consequence, historical data of S&P/BMV was downloaded. It is a ratio of an American stock market index based upon 500 the largest companies to the biggest Mexican stock exchange index.

Adding new currencies involved a lot of refactoring. The whole process of building classification model was changed to work on-the-fly. For each currency, the program retrieves a CSV file that has to be called "currency + _Index.csv", then the file is read and merged with tweets. Features are selected individually for each stock what in consequence selects most efficient features for each data. That is because targets for each stock are different so different instances (and their features) have an impact on the result. Once selected features are saved to the file in a "data" folder to be read next time while building a new model and save time.

The most problematic was to implement feature selection to be done automatically in the program. Until that moment program was using features that had been selected in advance in *WEKA* manually. To automate whole process features had to be selected another way. Three possible ways were identified:

- feature selection using *SciKit-learn*'s selectors
- feature selection using other third-party libraries
- feature selection in *WEKA*

The simplest way seemed to be using *SciKit-learn*. Despite the fact that it is the most popular framework among machine learning community, it includes only a few selectors:

- "VarianceThreshold" - removes features with low variance, what is useless because all the infrequent features are already dropped. (all that occurred less than 7 times) [36]
- Univariate selectors such as *KBest* with *chi2*. These gave very bad results because they rank features and select *K* best from the top of the ranking. [36] That approach is ineffective because features sets are chosen against their helpfulness alone, not when there are together. For example, some features can significantly improve selection when there are used together but they may be useless while being alone. In case of the ranking, they would be near to the bottom of the list and would not be selected. Besides, many of the features from the top of the ranking may be correlated (both would duplicate the information they provide so one of them would be sufficient). Moreover, correlated features make training slower and may change model outputs. Some algorithms such as Naïve Bayes can actually benefit or loss due to the redundant features depending on their correctness. (NB treats features independently so if many correlated features are correct then they can improve the result, but on the other hand, they can spoil it when features are uninformative). That is why the best solution is to use *Wrapper* method for feature selector because it removes features that do not contribute directly to the performance. [37]

- Recursive feature elimination. Eliminator requires being given an external estimator that scores features (e.g. Naïve Bayes that assigns coefficient score to features that were used). Then they are selected by recursively considering smaller and smaller sets of features. [36]

After training on the whole set of features, coefficient marks are taken into account and the less important features are pruned. The process of pruning is repeated until the user-defined number of features is reached.

It was decided to try out RFECV (Recursive feature elimination with cross-validation) which tunes the number of features to prune with cross-validation. [36]. The taken approach was to set a wide range of features number (20-300) to select by RFECV and then choose the best set of features.

RFECV was returning results for each number of features to prune and the results were much better than when using *WEKA*'s wrapper.

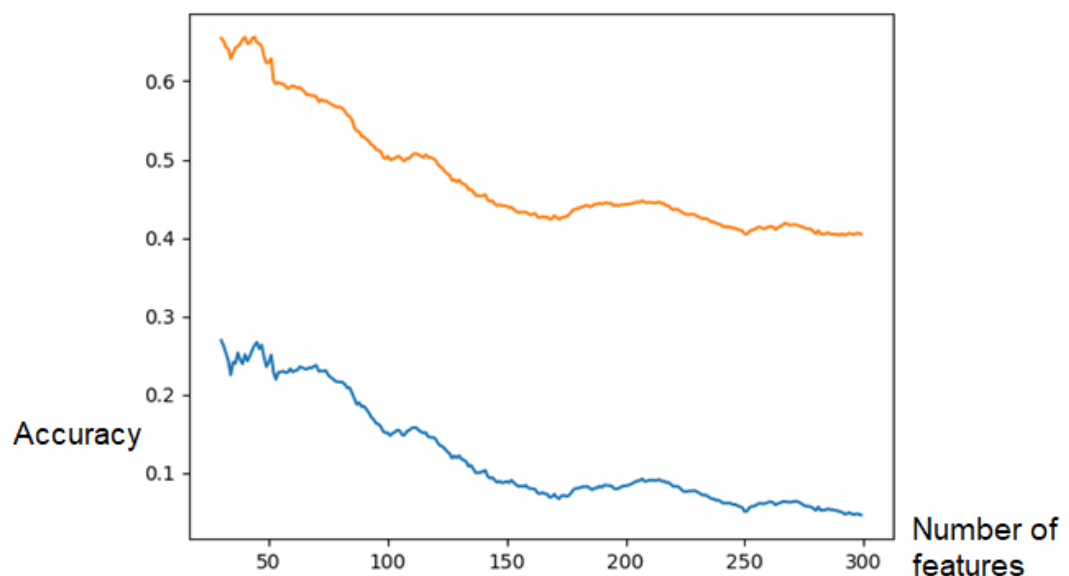


Figure 3-7. The orange line is a test accuracy of the model for a particular number of features. The blue line represents a difference of test accuracy and base accuracy (changing a number of features was altering the size of the majority class).

Unfortunately, what was discovered is that selecting lower and lower number of features was simplifying the problem because tweets that did not include any of the selected features were dropped. It resulted in a smaller number of tweets that were taken into account. Therefore, this way was incorrect because a lower number of features and tweets is extremely simplifying a problem and the model would be able to predict the target only for tweets that have got any of the small set of features. The same number of features chosen by RFECV gave worse results than when chosen in *WEKA*.

3.3.9. Running *WEKA* feature selection from Python

Due to the lack of possibilities to select features with *SciKit-learn*, other options have been considered. Another library called *Mlxtend* was identified. It provides extensions and helper modules for Python's data analysis and machine learning libraries. [19]

There were tested “*ExhaustiveFeatureSelector*” and “*SequentialFeatureSelector*” from this module, but both required a lot of time to process and gave very poor results.

Thus, it was decided to use well-proven *Wrapper Evaluator* from *WEKA*. *WEKA* is a Java library and two options were considered to integrate it with the project. One option was to run *WEKA* using wrappers that allow Python interact with *JVM* such as *JavaBridge*, *Py4j*, *Jpypy*. Another way was to run *WEKA* as a system process and parse the output. This way seemed to be the simplest and was implemented. A *WEKA* jar file was added into a project directory and the package is run as system command using “subprocess” module:

```
def run_weka_with_file(temp_filename):
    command = ['java', '-classpath', WEKA_JAR_FILE,
               'weka.attributeSelection.WrapperSubsetEval',
               '-T', '0.5',
               '-B', 'weka.classifiers.bayes.NaiveBayesMultinomial',
               '-s', 'weka.attributeSelection.BestFirst',
               '-i', temp_filename]
    stdoutdata = subprocess.getoutput(command)
```

Figure 3-8. A snippet of a part of a function running *WEKA* to select features

Then the features have to be parsed from the output. Obviously, this way is not perfect because has got drawbacks such as fact that the exceptions would have to be caught and parsed in the output. It is also harder to debug and test.

The model is then build using a dataset that is sifted with selected features. This way it was selected 103, 99, 103 features and sifted 1173, 1121, 1081 tweets for USD, EUR and MEX respectively, what seems reasonable. Results of running the program with new currencies and feature selection done with *WEKA* is as follows:

Currency	Classifier	Test accuracy	Train accuracy	Baseline
USD	Naïve Bayes	53.8%	60.3%	41.7%
	Logistic regression	52.0%	57.4%	
EUR	Naïve Bayes	52.6%	58.8%	36.0%
	Logistic regression	41.7%	50.3%	
MEX	Naïve Bayes	51.2%	59%	39.7%
	Logistic regression	44.1%	51.9%	

3.3.10. Using two models to predict changes

At this stage when the user analysed a tweet that did not include any of the 10-ish selected features the model would predict the value based on a sentiment. Following feedback from the project supervisor, it was decided that there should be built another model on a dataset without feature selection. That model could be used when analysed tweet features do not match with ones that were selected.

The second model was trained on the dataset that has got dropped infrequent features and instances which only marked feature is sentiment. (807 features and 2022 tweets). It was decided to take the following strategy:

- if the tweet has got any feature from sifted model then use this model
- if has not got any selected features but has any of the features that is the second model built on, then use this one
- if has not got any features matching to the ones that were extracted from the corpus then use both models and return average of results.

Deciding what to do in the case when the tweet does not have any features was hard. The only sensible solution was to just predict based on sentiment. In this case, it was decided to use both modules and return the average value.

Models built on a dataset without selected features have got very poor accuracy:

Currency	Classifier	Test accuracy	Train accuracy	Baseline
USD	Naïve Bayes	38.7%	65.3%	36.2%
	Logistic regression	36.2%	36.2%	
EUR	Naïve Bayes	38.0%	66.2%	39.3%
	Logistic regression	39.3%	39.3%	
MEX	Naïve Bayes	39.0%	67.4%	36.0%
	Logistic regression	36.0%	36.0%	

Naïve Bayes had slightly better accuracy than the baseline but Logistic regression predict exactly as if it was predicting randomly. Besides, NB had always got better accuracy than Logistic Regression, therefore it was chosen to be set as a default classifier.

3.4. Rules learning

Once the model was working properly next thing that was done was association rules learning. It is a "rule-based machine learning method for discovering interesting relations between variables in large databases". [38]

The algorithm that has been used is Apriori. It is the most popular association rule learning algorithm and it was also available in one machine learning modules: Mlxtend.

The outcome of the learning process is a set of rules. Each rule consists of antecedents and consequents. Antecedents are words whose appearance is accompanied by consequents. Basically, rules tell us which words or phrases occur frequently together in a dataset.

For example, this rule tells us that if in any tweet in the dataset has got words "Dossier, Hillary, Trump campaign" it also has word "Clinton":

Antecedents	Conse- quents	Antecedent support	Consequent support	Support	Confidence	Lift
dossier, hillary, trump campaign	clinton	0.00185014	0.03607771	0.00185014	1.0	27.7

- Support is an indication of how frequently the words set appears in the dataset. [38]
If there are 1080 tweets and the words set occurred twice then the support will be $2/1080 = 0.00185014$.
- Antecedent support is an indication how frequently antecedent occurred in the dataset. (no matter what was the consequence)
- Consequent support is an indication how frequently consequent occurred in the dataset.
- Confidence tells us how often the rule has been found to be true [38]. In the example above, every time those antecedents occurred in a tweet, word "clinton" occurred as well.
- Lift is a ratio of the confidence of the rule and the support of the consequents. [39]
Greater lift values indicate stronger associations what can be interpreted as an importance of a rule. [38]

Rules are different for each currency because dataset used to build models are sifted differently. Rules presented on the webpage are filtered by a number of occurrences (at least twice). It was decided that there is no point to present a rule that occurred only once.

Rules presented on the webpage are grouped into words set. For example:

{daca, drug} \rightarrow {military} and {drug, military} \rightarrow {daca} are grouped into one set of words. Both of them have got the same confidence and support because it is common for the whole dataset. That way of presenting results is more readable. By clicking on the words set, rules containing these words are presented.

Words set	support		confidence	
daca, drug, military	0.00255754		1.0	
antecedants	consequents	antecedant support	consequent support	lift
daca, drug	military	0.00255754	0.06479113	15.43421053
military, drug	daca	0.00255754	0.03239557	30.86842105

Figure 3-9. Example of rules presented on the website

3.4.1. Final workflow

After all the process of analysing a currency is as follows:

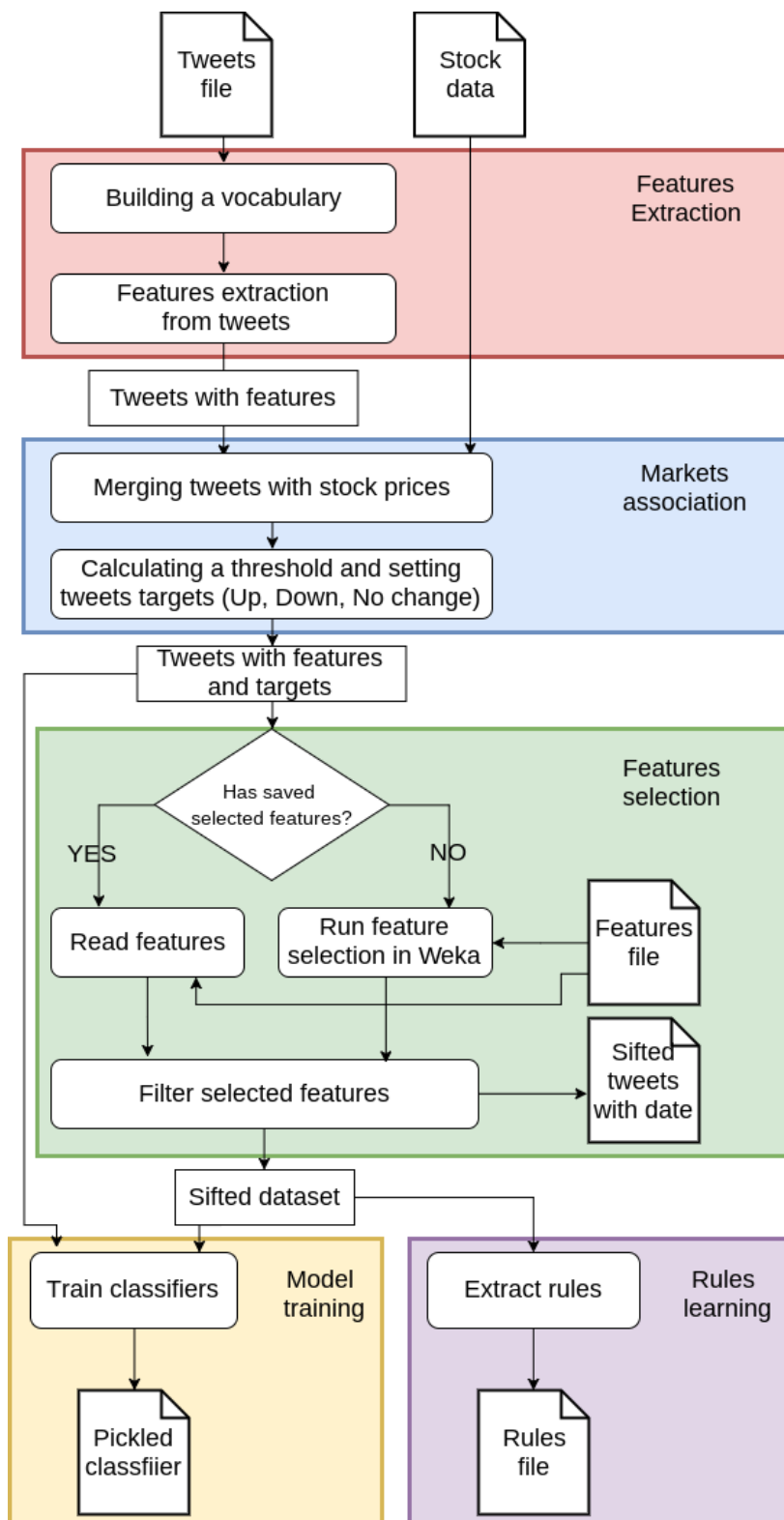


Figure 3-10. The overall process of analysing the association between tweets and markets

3.5. Flask webpage

3.5.1. Application initialization

Once the module analysing association between tweets and markets was done, the next step was to create a *Flask* web interface. The application was organized in a way typical for *Flask* applications.

The whole application is created in “__init__.py” file. It sets up all the configuration and creates a database object that is bind to the application.

The application is loaded from “manage.py” file. It initializes a database and creates a CurrencyAnalyser for each hardcoded currency. Then, analyses the data and creates a classifier. The results are stored in the database and all classifier modules are saved implicitly. Therefore, when the application is run again the data can be retrieved and presented.

When the application is run, all data about analysed markets is loaded from the database and assigned to the application object. For each loaded currency, there is created a CurrencyAnalyser, that can be used to obtain various data about the dataset or to predict custom tweet effect.

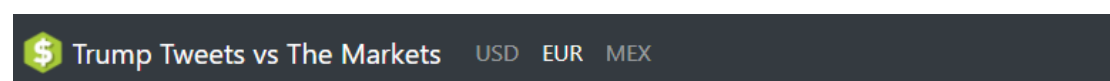
3.5.2. Views and templates

Application loads all routes from views.py file. The web interface presents all the information about the currency on one page so there is only one view specified. All currencies belong to the “/currency” path, so the path for the EUR results will be “/currency/EUR”. The default path “/” was set to redirect to USD webpage. If the user wanted to access currency webpage that is not analysed there is returned 404 status code.

The data is then sent to the *Jinja2* which uses it to populate templates. Templates use common technologies such as HTML, CSS and JavaScript. For example, links to all the analysed stocks are also added to the navbar this way. Therefore, when a new currency is analysed then it is read from the database automatically:

```
<div class="collapse navbar-collapse" id="navbarNav">
  <ul class="navbar-nav">
    {% for c in currencies %}
    <li class="nav-item">
      <a class="nav-link {% if currency_details.name == c.name %}active{% endif %}"
        href="{{ url_for('index', currency=c.name) }}">{{c.name}}</a>
    </li>
    {% endfor %}
  </ul>
</div>
```

Figure 3-11. Example of using Jinja2 to set navigation items.



Predict tweet affect on market

Figure 3-12 Example of a navigation bar presenting 3 different currencies.

The webpage presents data that is fetched from the loaded CurrencyAnalyser object. In the middle of the page, there is a text form that allows the user to input some example tweet to classify its market influence. All the results (such as effect, sentiment, features found in the tweet, probabilities of each class) are presented on the right side of the form.

4. Testing

4.1. Unit testing

Testing was carried out throughout the entire development process. The chosen strategy was TDD (Test driven development), so while adding new functionalities, tests were written first and then was added code to fulfil their requirements and make them pass. Writing unit tests was a basic premise of the project because they prove the quality of the code and that it works at all. Having a set of tests is extremely helpful when modifying the code because we know that while adding one feature we do not break another. To write unit test built-in python “unittest” module was used. Then they are run using “nose” module. Nose finds all the tests in the package and runs them in a more user-friendly way, errors are more readable. “mock” module was also necessary to use in few cases. It allows mocking out and monkey patching some parts of the code that we do not want to be executed.

```
@patch("markets.currency_analysis.DATA_PATH", "data_path_here")
class TestCurrencyAnalyser(unittest.TestCase):
    def setUp(self):
        self.analyser = CurrencyAnalyser("ABC")

    def test_constructor(self):
        self.assertEqual(r"data_path_here\ABC_rules.csv", self.analyser.rules_filename)
```

Figure 4-1. Monkey patching a constant with the data path to make sure that tests will be repeatable on every machine.

Unit tests use parametrized module that is very popular among Python community. It allows running tests fed with different parameters that can be used as arguments and expected results. In consequence, it helps to avoid code duplications.

```
@parameterized.expand([('/currency/abc',), ('/usd',), ('/abc',), ('/currency',)])
def test_404(self, wrong_path):
    response = self.client.get(wrong_path)
    self.assertEqual(404, response.status_code)
```

Figure 4-2. Simple test using the parameterized module

A couple of exceptions was made in unit tests.

Scraping code was not tested at all, because the script was used just to download tweets and there was no point to test it. The code worked properly and tweets were scrapped as assumed. If the program was developed more, there was added a feature to choose any politicians' tweets and Scraping was be done automatically, then tests would be obviously required.

Feature selection module has not been tested because it is mostly executing functions form other modules and testing would require a lot of mocking. However, module functionality is tested in integration tests.

Another code that was not unit tested initially was the code that processes DataFrames. In the beginning, all the experiments were performed just on-the-fly. The data was loaded into a DataFrame and processing was performed in a pipeline. That approach was quick for experimenting but hard to test. It would require some mocking, patching and writing many sample DataFrames, which structure also was changed many times at the begin.

When the tweets processing and classifying part of the code became more complex, the program was split into the more object-oriented way. All the DataFrame code was wrapped into a "DataSet" class encapsulating all the DataFrame operations. It made testing easier because it was possible to test more units without involving DataFrames and creating whole datasets just to test one simple function.

Tests do not check too many exceptions that could occur during input files and data frames analysis. The program assumes that they are in a correct format. If the project was developed a bit more and allow users to add custom stock files then there would be much more tests to write and more corner cases to investigate?

Unit tests cover 72% of the code. Some of the functions were not tested because they were too simple like for example saving to file or using other libraries.

Name	Stmts	Miss	Cover
markets__init__.py	0	0	100%
markets\association.py	46	25	46%
markets\currency_analysis.py	68	30	56%
markets\dataset.py	85	7	92%
markets\feature_selection.py	48	32	33%
markets\market_predicting.py	121	45	63%
markets\phrases_extraction.py	125	5	96%
markets\rules.py	53	13	75%
markets\sentiment.py	75	27	64%
markets\tweets_features_extraction.py	46	6	87%
markets\utils.py	19	4	79%
TOTAL	686	194	72%

Figure 4-3. "Markets" package tests coverage.

The most painful issue with unit tests was that they had been changed while implementing every new functionality. The structure of the program was changing constantly and the code was refactored what lead to the final tests looking completely different than when they were written at the begin.

Most of the functionalities that are not tested separately in unit tests are tested in integration tests.

4.2. Integration/Acceptance Testing

When it was certain that smaller bits of code work then next step was to write integration tests. This is also a very important suite of tests that prove that all the bits of code will work together and the program They test the functionality of every feature.

To write integration tests was used a Behave module. It is an equivalent of Cucumber – popular testing framework among Ruby on Rails' community. All the features are tested by writing scenarios. All the scenarios are written in Gherkin – a simple human-readable language for automated tests.

```
Scenario: Extract words|  
Given we have a vocabulary of: cucumber, apple, banana, tomato  
When we extract phrases from the sentence: I like to eat apples and bananas  
Then we get apple, banana extracted
```

Figure 4-4. Example Gherkin scenario

Those "Steps" written in simple language are converted into code through step definitions:

```
@given('we have vocabulary of: {vocabulary}')  
def step_impl(context, vocabulary):  
    context.e.set_features(vocabulary.split(", "))
```

Figure 4-5. Example step definition

It keeps high-level concerns separated from the code and allows non-technical people to write tests.

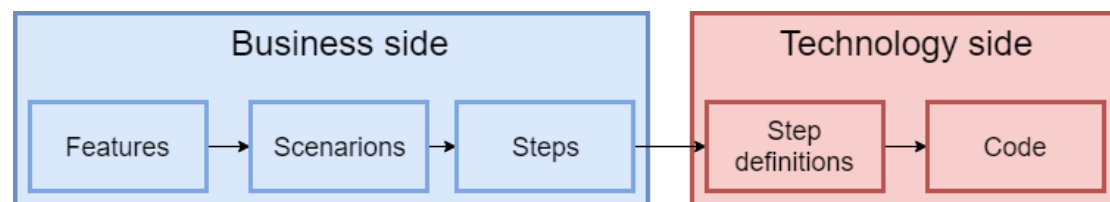


Figure 4-6. Structure of creating integration tests

Functionalities that were tested are:

- Phrase extraction
- Sentiment analysis
- Stock prices prediction

Models use SciKit learn classifiers what would make testing hard and unrepeatable, therefore it was decided to mock out the models for tests. Anyway, the tests should verify the correctness of links among the bits of the project and should not tests third-party code. Mocking out with auto-specing (mock module makes sure that the patched code has got the same interface) should be sufficient in this case.

4.3. Usability testing

Usability testing is a technique used to evaluate how easy is the interface to use. The webpage in this project is only used to present the results and only functionalities are changing currencies pages and analysing tweets what was quickly tested by the student and supervisor.

5. Critical Evaluation

The aim of this project was to determine if there is any connection between Trump tweets and the markets. It was also decided to analyse the tweets dataset and find out what we can learn and what information we can gather.

The goal has been achieved, a relationship between tweets and the markets was found. The classifier is able to predict the currency change much better than if it was doing randomly (54% accuracy to 41% base accuracy).

The most coefficient features while training the model turned out to be quite sensible.

Python was a great choice for this project. It had all the functionality required for the task, all the needed libraries were available, up to date and well-functioning.

Programming in this language was very quick and allowed to do quick experiments.

All the libraries did their job properly. The only thing that has been done different way than specified in OPS was using Flask instead of Django, what was a better choice because allowed to create interface quicker and easier, with the same functionality.

All of the primary tasks proposed in OPS have been accomplished. There was also time to start additional tasks. Other currencies were added and the program allows to add new CSV files with other indices as long as they are in the same format and have got the same columns.

What could be done better/different way?

Writing data processing in the more object-oriented way instead of using DataFrames would make the program simpler. I would just make a DataSet class having a list of tweets and each tweet would store information about its features and sentiment. It would make processing a bit slower but it would help to achieve more modularity and lower cohesion. Therefore, testing would be easier.

I think that if the code was written in a more modular way then it would be easier to test and mocking would not be so much needed in some places.

Some of the tasks that were specified as additional were not accomplished because they required much more work. The program is ready to implement an option to add custom stock prices or even select on the website which ones the user wants to be scraped from the Internet. However, it would require to verify the input, check a lot of corner cases and handle all of the possible exceptions. It was decided to do not even start doing this because it requires a lot of time to do it properly.

Adding option to choose any Twitter account to analyse, would also involve a lot of time. Many corner cases with processing tweets would have to be considered. All the pre-processing that was done manually, such as removing hyperlinks, videos, useless characters would have to be done by the application. It would involve a lot of testing. Moreover, useless Trump's tweets were also filtered out manually. Applying this process to the code is hard. It could be skipped but then the resulting accuracy would be worse.

Lemmatizing could be also slightly fixed because the NLTK library has got the problem with lemmatizing some words, for example, "isis" is treated as a plural and changed into "isi". However, it does not influence the overall results because all these words are reduced to the same root form and are still treated as the same feature.

There is also a problem with displaying long tooltips on the chart. The text is cut when it is too long and does not fit into one line. It is known ChartJS bug that maybe will be fixed in the next version.

Appendices

A. Third-Party Code and Libraries

Image used as a logo (webpage/static/dollar_logo.png) downloaded from <http://www.pngmart.com/image/28615>

TextBlob – a free library for processing textual data. Released using MIT license. [40]
Was used at the begin and then Sci-Kit-learn was used in lieu. Version used 0.15.1

SciKit-Learn – Python machine learning library. It was used to do cross-validation and build classifiers It is free and open source. This library is released using BSD license.

WEKA – is a suite of machine learning software written in Java. It is free software licensed under the GNU General Public License. [15] It was used to for experimenting with the data and to do a feature selection. It is run by the program as subprocess command. WEKA Jar file is included in the project directory. Version 3.8.2 was used.

Mlxtend – The project used this library to do association rule learning with Apriori algorithm. It is released under BSD licence. Version used 0.12. [19]

Pandas – library used to do data manipulation and analysis. The library is released using BSD license. Version used 0.22.0 [41]

Tweepy – Python library used to access the Twitter API. It is released using MIT license. Version used 3.6.0 [42]

Behave – behaviour-driven development testing module. It is used to do integration testing. Version used 1.2.6. [21]

Flask - micro web framework used to create a web interface. It was used with its extensions: flask-sqlalchemy, flask-wtf and flask-scripting to use databases, forms and managing scripts. Released using BSD license. Flask version 1.0.1 was used. (24)

ChartJS – JavaScript library used to create HTML5 charts on the webpage. Released using MIT license. Version used is 2.7.2. [43]

Bootstrap - open source toolkit for developing with HTML, CSS, and JS. Was used as a CSS style. Released using MIT license. Version used 4.1.0. [44]

JQuery - free, open-source JavaScript library under MIT License. Was used to add animations to the webpage. Version used 3.3.1.

NumPy – library adding support for large, multi-dimensional matrices. Pandas uses this module underneath. Was used mostly for testing Pandas related code. Released using BSD license. Version used 1.14.3. [45]
Nose and parametrized

All those libraries were used without modification.

B. Ethics Submission**AU Status**

Undergraduate or PG Taught

Your aber.ac.uk email address

mas15@aber.ac.uk

Full Name

Stankiewicz Mateusz

Please enter the name of the person responsible for reviewing your assessment.

Reyer Zwiggelaar

Please enter the aber.ac.uk email address of the person responsible for reviewing your assessment

rrz@aber.ac.uk

Supervisor or Institute Director of Research Department

cs

Module code (Only enter if you have been asked to do so)

CS39440

Proposed Study Title

MMP Trump Tweets vs The Markets - building a classifier on Trumps tweets to predict stocks changes

Proposed Start Date

29/01/2018

Proposed Completion Date

04/05/2018

Are you conducting a quantitative or qualitative research project?

Mixed Methods

Does your research require external ethical approval under the Health Research Authority?

No

Does your research involve animals?

No

Are you completing this form for your own research?

Yes

Does your research involve human participants?

No

Institute

IMPACS

Please provide a brief summary of your project (150 word max)

Building a machine learning classifier on Trump tweets to predict stock changes. Involved sentiment analysis and text feature extraction.

Where appropriate, do you have consent for the publication, reproduction or use of any unpublished material?

Not applicable

Will appropriate measures be put in place for the secure and confidential storage of data?

Yes

Does the research pose more than minimal and predictable risk to the researcher?

Not applicable

Will you be travelling, as a foreign national, in to any areas that the UK Foreign and Commonwealth Office advise against travel to?

No

Please include any further relevant information for this section here:

If you are to be working alone with vulnerable people or children, you may need a DBS (CRB) check. Tick to confirm that you will ensure you comply with this requirement should you identify that you require one.

Yes

Declaration: Please tick to confirm that you have completed this form to the best of your knowledge and that you will inform your department should the proposal significantly change.

Yes

Please include any further relevant information for this section here:

Annotated Bibliography

1. Neil Mac Parthaláin. MMP: Project descriptions. *MMP website*. [Online] [Accessed: 30 4 2018.] <https://teaching.dcs.aber.ac.uk/mmp>.
2. Donald Trump's tweet. *Twitter*. [Online] 27 2 2017. [Accessed: 30 04 2017.] <https://twitter.com/realdonaldtrump/status/901802524981817344>.
3. Kdd. [Online] 30 04 2018. <http://www.kdd.org/curriculum/index.html>.
4. What is machine learning. *techemergence*. [Online] <https://www.techemergence.com/what-is-machine-learning/>.
5. Machine learning definition. *expertsystem*. [Online] [Accessed: 30 04 2018.] <http://www.expertsystem.com/machine-learning-definition/>.
6. Twitter. *Wikipedia*. [Online] [Accessed: 30 4 2018.] <https://pl.wikipedia.org/wiki/Twitter>.
7. Donald Trump Twitter page. *Twitter*. [Online] [Accessed: 30 4 2018.] <https://twitter.com/realdonaldtrump>.
8. Donald Trump says he would not be President without Twitter. *Independent*. [Online] [Accessed: 30 04 2018.] <https://www.independent.co.uk/news/world/americas/us-politics/donald-trump-tweets-twitter-social-media-facebook-instagram-fox-business-network-would-not-be-a8013491.html>.
9. Text analysis of Trump's tweets confirms he writes only the (angrier) Android half. *varianceexplained*. [Online] [Accessed: 30 04 2018.] <http://varianceexplained.org/r/trump-tweets/>.
10. Xue Zhang, Hauke Fuehres, Peter A. Gloor. *Predicting Stock Market Indicators Through Twitter "I hope it is not as bad as I fear"*. s.l. : Procedia - Social and Behavioral Sciences, 2011. ISSN 1877-0428.
11. Tokenizing words sentences nltk tutorial. *Python programming*. [Online] [Accessed: 2018 4 30.] <https://pythonprogramming.net/tokenizing-words-sentences-nltk-tutorial/>.
12. Rose, Stuart & Engel, Dave & Cramer, Nick & Cowley, Wendy. *Automatic Keyword Extraction from Individual Documents*. s.l. : Text Mining: Applications and Theory, 2010.
13. Twitter libraries. *Twitter docs*. [Online] [Accessed: 30 4 2018.] <https://developer.twitter.com/en/docs/developer-utilities/twitter-libraries>.
14. DataFrames documentation. *Pandas documentation*. [Online] [Accessed: 30 4 2018.] <https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html>.

15. *Official Weka website*. [Online] The University of Waikato. [Accessed: 30 4 2018.] <https://www.cs.waikato.ac.nz/ml/weka/>.
16. What is Jython. *Jython wesite*. [Online] [Accessed: 30 4 2018.] <http://www.jython.org/archive/21/docs/whatis.html>.
17. Description of Javabridge project. *Pypi*. [Online] [Accessed: 30 4 2018.] <https://pypi.org/project/javabridge/>.
18. Alex Rogozhnikov. Benchmarks of speed (Numpy vs all). [Online] 6 1 2015. [Accessed: 30 4 2018.] <http://arogozhnikov.github.io/2015/01/06/benchmarks-of-speed-numpy-vs-all.html>.
19. Sebastian Raschka. *Official Mlxtend website*. [Online] [Accessed: 30 4 2018.] <http://rasbt.github.io/mlxtend/>.
20. *Beautiful Soup Documentation*. [Online] [Accessed: 30 4 2018.] <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>.
21. Benno Rice, Richard Jones and Jens Engel. *Behave documentation*. [Online] 2017. [Accessed: 30 4 2018.] <https://behave.readthedocs.io/en/latest/>.
22. gabrielfalcao. *Lettuce github page*. [Online] [Accessed: 30 4 2018.] <https://github.com/gabrielfalcao/lettuce>.
23. Alexey Kotlyarov. *Aloe github page*. [Online] [Accessed: 30 4 2018.] <https://github.com/alotesting/aloe>.
24. *Flask-sqlalchemy module website*. [Online] [Accessed: 30 4 2018.] <http://flask-sqlalchemy.pocoo.org/2.3/>.
25. *SQLAlchemy website*. [Online] [Accessed: 30 4 2018.] <https://www.sqlalchemy.org/>.
26. *Trump Twitter Archive*. [Online] [Accessed: 30 4 2018.] <http://www.trumptwitterarchive.com/>.
27. Tutorial: Building a Text Classification System. *Textblob documentation*. [Online] [Accessed: 30 4 2018.] <http://textblob.readthedocs.io/en/dev/classifiers.html>.
28. Donald Trump. Donald Trump's tweet. *Twitter*. [Online] 18 11 2017. [Accessed: 30 4 2018.] <https://twitter.com/realDonaldTrump/status/931877599034388480>.
29. *TextRazor website*. [Online] [Accessed: 30 4 2018.] <https://www.textrazor.com/>.
30. Alyona Medelyan. Implementation of RAKE. *Alyona Medelyan's GitHub*. [Online] 2 3 2018. [Accessed: 30 4 2018.] <https://github.com/zelandiya/RAKE-tutorial/blob/master/rake.py>.
31. stop word list from SMART. [Online] 1971. [Accessed: 30 4 2018.] <ftp://ftp.cs.cornell.edu/pub/smart/english.stop>.
32. Donald Trump. Donald Trump's tweet. *Twitter*. [Online] 4 10 2017. [Accessed: 30 4 2018.] <https://twitter.com/realdonaldtrump/status/919162619889704961>.
33. LogisticRegressionCV documentation. *Sci-Kit learn documentation*. [Online] [Accessed: 30 4 2018.] http://SciKit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html.
34. Jason Brownlee. An Introduction to Feature Selection. *Machine Learning Mastery*. [Online] 6 9 2014. [Accessed: 30 4 2018.] <https://machinelearningmastery.com/an-introduction-to-feature-selection/>.
35. Mexico braced for exodus to US as 'Trump effect' hurts the peso. *The Guardian*. [Online] 21 1 2017. [Accessed: 30 4 2018.] <https://www.theguardian.com/business/2017/jan/21/mexico-braces-trump-effect-peso-migration>.
36. Feature selection documentation. *Sci-Kit learn documentation*. [Online] [Accessed: 30 4 2018.] http://scikit-learn.org/stable/modules/feature_selection.html.
37. In supervised learning, why is it bad to have correlated features? *Stackexchange*. [Online] 7 11 2017. [Accessed: 30 4 2018.] <https://datascience.stackexchange.com/questions/24452/in-supervised-learning-why-is-it-bad-to-have-correlated-features>.
38. *Wikipedia*. [Online] [Accessed: 30 4 2018.] https://en.wikipedia.org/wiki/Association_rule_learning.

39. Lift in an association rule. *IBM Knowledge Center*. [Online] [Accessed: 30 4 2018.]
https://www.ibm.com/support/knowledgecenter/en/SSEPGG_9.5.0/com.ibm.im.model.doc/c_lift_in_an_association_rule.html.
40. License. *Textblob documentation*. [Online] [Accessed: 30 4 2018.]
<http://textblob.readthedocs.io/en/dev/license.html>.
41. *Pandas Official website*. [Online] [Accessed: 30 4 2018.]
<https://pandas.pydata.org/>.
42. *Tweepy GitHub*. [Online] [Accessed: 30 4 2018.]
<https://github.com/tweepy/tweepy>.
43. *ChartJS GitHub*. [Online] [Accessed: 30 4 2018.]
<https://github.com/chartjs/Chart.js>.
44. *Bootstrap official website*. [Online] [Accessed: 30 4 2018.]
<https://getbootstrap.com/>.
45. NumPy. *Wikipedia*. [Online] [Accessed: 30 4 2018.]
<https://en.wikipedia.org/wiki/NumPy>.
46. DATA MINING CURRICULUM: A PROPOSAL. *kdd*. [Online] 2018 04 27.
<http://www.kdd.org/curriculum/index.html>.