FDS Group Project - Stock Returns Prediction using Aggregated Sentiment Scores and Traditional Signals

Evan Mc Garry, Gavin Connolly, Anastasia Yanovych

May 31, 2023

UCD Michael Smurfit Graduate Business School

Table 1: Peer Assessment Table

Evaluator	Subject	Peer Effort (/10)	Peer Attitude (/10)	Peer Contribution (/10)	Total Peer Mark\n(/30)
Evan	Evan	10.0	10.0	10.0	30
Gavin	Evan	10.0	10.0	10.0	30
Anastasia	Evan	10.0	10.0	10.0	30
					Evan - Total Average = 10
Evan	Gavin	10.0	10.0	10.0	30
Gavin	Gavin	10.0	10.0	10.0	30
Anastasia	Gavin	10.0	10.0	10.0	30
					Gavin - Total Average = 10
Evan	Anastasia	10.0	10.0	10.0	30
Gavin	Anastasia	10.0	10.0	10.0	30
Anastasia	Anastasia	10.0	10.0	10.0	30
					Anastasia - Total Average = 10

1 Executive Summary

The Finance sector is at the forefront of AI-integration (Forbes, 2019) and this will be confirmed through quantitative analysis of companies annual 10-K reports in the United States (US). The question remains as to whether there is a positive sentiment surrounding the implementation of AI/Machine Learning models in business operations, which this report purports to answer. In the case of a resulting positive sentiment the usefulness of this data in a predictive context for returns is assessed. This data is used in tandem with more generic indicators to see if high predictive accuracy can be achieved using this data thus establishing the relationship between a companies returns and its level of AI maturity.

Each section of this report involves database creation or manipulation, as well as extensive data-cleaning. Significant steps undertaken for database creation or data-cleaning will be contained in Appendix A as well as discussion of the methodology and detailed implementation of each of the key steps listed below:

- **Novel Data Set Collection: SEC Edgar web-scraping & Yahoo Finance Financial Data**
  - The annual 10-K reports for each company within identified sectors are scraped from the SEC Edgar website.
  - The scraped data is stored in a database (with the annual 10-K reports stored as a string). The database is queried to allow for easy access to the vast amount of scraped data for further analysis. It was also necessary to perform data cleaning steps at this stage in order for the database to provide inherent value.
  - Further data is retrieved from Yahoo Finance which relates to individual companies' financial statements and historical stock performance. This data is stored in a separate database and cleaned intensively (including transformations) so it can be input directly into the Machine Learning model.
- **Textual Analysis & Data Visualisation: Sentiment Analysis**
  - The annual 10-K reports (as queried from the database) are examined for occurrences of certain keywords and these occurrences are counted to identify which sectors are most focused on AI/ML integration relative to the others.
  - The average counts per industry is plotted in order to, more accurately, quantify the focus of a particular industry on AI/ML integration.
  - The annual 10-K reports are re-used to perform "Aspected-based Sentiment Analysis" under a *rule-based approach* in order to calculate the aggregated sentiment score for each of the keywords per industry.
  - All of this data is stored in the database.
- **Machine Learning: Predictive Model for Stock Returns & ESG Scores**
  - All necessary data pertaining to predictor and output variables is queried from the databases and cleaned intensively, including transformations, to guarantee seamless fitting of the model.
  - The database queries will provide *returns-related data* and for each company, grouped by industry in order to prevent cross-contamination of the data.
  - The models implemented are "Boosting" and "Random Forest" with the performance of each compared under the same metric, *explained variance* of the data. Instead of backtesting, the performance comparison will be done using cross-validation due to the scope of this project only reaching back to 2017. In these circumstances, cross-validation will undoubtedly be more efficient and will ultimately decide which model is chosen for implementation.
- **Business Analysis: Efficacy of Models for Stock Returns Prediction**
  - The performance of the models as well as the scalability of the code used to gather the data throughout the project is discussed in order to ascertain whether or not implementation of this model would be worthwhile.
  - The specific benefits of implementation are discussed from the perspective of the investor utilising the model.

This report utilised numerous online sources to help with the coding, in particular the web-scraping which was onerous due to the voluminous nature of the annual 10-K reports. All references will be provided in Appendix A.11.

2 Novel Data Set Collection

2.1 SEC Edgar Web-scraping

The primary data source is the SEC Edgar website which is a database of all filings made by public companies in the USA. This process is made easier by the fact that "machine readability" is a focus of the SEC Edgar database (SEC, 2018) which allows for a loop to be written to extract the annual 10-K reports for each company within a specific sector ranging from . This is because the SEC provides a "Central Index Key" (CIK) for each company which are grouped by sector using the "Standard Industrial Classification" (SIC) code. This report groups these SIC codes to form an industry to ensure all potential companies in an industry are being considered. The industries considered, with their corresponding SIC codes, are shown below:

Table 2: SIC Codes Mapped onto Industry

Industry	SIC Codes					
Agriculture	0100	0200	0900			
Financial Services	6172	6199	6200	6211	6221	6282
Aviation	4513	4522	4581			
Banking	6022	6029	6035	6036	6099	

This summarises the scope of the web-scraping, with this in mind the goal of the web-scraping process becomes clear. A loop is constructed, as shown in Appendix A.1, to scrape the annual 10-K reports for each company (as denoted by its unique CIK) within each SIC code for the years 2017-2023. This date range was chosen because 2017 was seen as the start of acceleration in the usage of AI (World Economic Forum, 2017). This is a complex process, so the raw scraped data is inserted into the database first. After this the raw 10-K data is queried and subsequently cleaned through an equally lengthy process, as shown in Appendix A.2, to facilitate further analysis. The cleaned data was then stored in the same database, as a string, in a new table to allow queries to be run against it. A snippet of how the table containing the cleaned data in the database looks is depicted below:

Table 3: Snippet of 10-K Table in SEC Filings DataBase

SIC	CIK	Company	Filing_Date	Filing_Document_Text
6029	0001527383	BankGuam Holding Co	2022-03-28	['false 2021 fy 0001527383 –12-31 true p1y p1y...
6029	0001527383	BankGuam Holding Co	2021-03-22	['10-k 1 bkgm-10k_20201231.htm 10-k bkgm-10k_20...
6029	0001527383	BankGuam Holding Co	2020-03-19	['10-k 1 bkgm-10k_20191231.htm 10-k bkgm-10k_20...
6029	0001527383	BankGuam Holding Co	2019-03-15	['10-k 1 bkgmf-10k_20181231.htm 10-k bkgmf-10k_...
6029	0001527383	BankGuam Holding Co	2018-06-29	['10-k 1 bkgmf-10k_20171231.htm 10-k bkgmf-10k_...
6029	0001527383	BankGuam Holding Co	2017-03-14	['10-k 1 bkgmf-10k_20161231.htm form 10-k bkgmf...
6029	0001531031	Esquire Financial Holdings, Inc.	2023-03-27	['p3y p3y 0.1544 0.08 0.06 0.045 0 p3y 00015310...
6029	0001531031	Esquire Financial Holdings, Inc.	2022-03-11	['p3y p3y 0 p3y 0001531031 2021 fy false http:/...
6029	0001531031	Esquire Financial Holdings, Inc.	2021-03-19	['p3y p3y 0 0 0 p3y 0001531031 –12-31 2020 fy ...
6029	0001531031	Esquire Financial Holdings, Inc.	2020-03-12	['10-k 1 esq-20191231x10k.htm 10-k esq_current_...

2.2 Yahoo Finance API - Financial Data

It is necessary to retrieve further data to be used in the ML predictive model and querying Yahoo Finance API gives access to financial statement data as well as stock-related data. The issue is that Yahoo Finance uses "tickers" whereas we have been identifying companies through their CIK codes and company names up until this point. In line with the SEC's movement towards machine readability, they regularly update a ".json" file that maps CIK codes onto tickers. Therefore it was necessary to request this file from the SEC and run a loop to map tickers onto their respective CIK codes, as shown in Appendix A.3. This was inserted into the database as a separate table as having a mapping of the tickers onto each company's name and CIK code is crucial. A sample of the database is shown below:

Table 4: Mapping Tickers onto CIK Code of each Company

CIK	Company	Ticker
0001531031	Esquire Financial Holdings, Inc.	ESQ
0001050743	PEAPACK GLADSTONE FINANCIAL CORP	PGC
0000034782	1ST SOURCE CORP	SRCE
0000715579	ACNB CORP	ACNB
0000040729	Ally Financial Inc.	ALLY
0001823608	Amalgamated Financial Corp.	AMAL
0000351569	Ameris Bancorp	ABCB
0000007789	ASSOCIATED BANC-CORP	ASB
0000883948	Atlantic Union Bankshares Corp	AUB
0000750574	AUBURN NATIONAL BANCORPORATION, INC	AUBN

With the first step taken care of, the actual financial data is gathered using Yahoo Queries which allows access to the previous 5 years of financial statements. A loop was constructed to extract figures for key terms from the "Balance Sheet" that allow for the calculation of key predictors such as "Book-to-Market", "Gearing", "Cash-to-Assets" and "Market Capital". The same was done for the income statements to obtain figures for the "Earnings-per-Share". In addition to this the stock data for the last 5 years was also retrieved using Yahoo Queries and the log returns calculated using the adjusted close price. This process required lots of cleaning and transformations which are discussed below in Appendix A.4.

3 Textual Analysis & Data Visualisation

3.1 Aggregated Sentiment Scores and Average Mentions

The financial data has been cleaned and transformed into usable data for the ML model, however the raw 10-Ks offer no value as an input to the model as they are. The next step is to calculate the count for the instances in the annual 10-Ks of each company for each of these keywords:

- Artificial Intelligence
- Deep Learning
- Computer Vision
- Bots
- Automation
- Machine Learning
- Algorithm

With the word count for each of these terms ("aspects") calculated for each company, take the average mentions of each keyword per industry for each year. This allows for a greater understanding of which sectors are placing an emphasis on AI/ML each year. The loop used above to calculate the occurrences of each keyword also included

"Aspected-Based Sentiment Analysis" to calculate the aggregated sentiment scores for each aspect per industry. The aggregated sentiment scores are bound between -1 and 1 with positive values indicating positive sentiment towards the keyword and vice versa. This loop both outputs and cleans this data, as discussed in Appendix A.5. The output for this loop is integral to the ML model and is input into the database, with a sample below:

Table 5: Sample of Output from Sentiment Analysis Loop

Industry	Aspect	Year	Count	Aggregated Sentiment	Average Mentions
Agriculture	automation	2023	9	0.284989	0.051724
Aviation	automation	2023	12	0.132717	0.095238
Banking	machine learning	2023	6	0.598433	0.002730
Financial Services	machine learning	2023	101	0.287640	0.071177

3.2 Data Visualisation

The sample of the Aggregated Sentiment Score table above illustrates the format of the data that will act as an input to the ML model, however this data has inherent value in identifying industries that are focused heavily on AI/ML as well as the sentiment of those industries towards AI/ML. This is shown in the bubble chart below, which depicts the aggregated sentiment score of each keyword for each industry as well as the average mentions of the keyword per industry (overall and not by year), the code for which is provided in Appendix A.6:

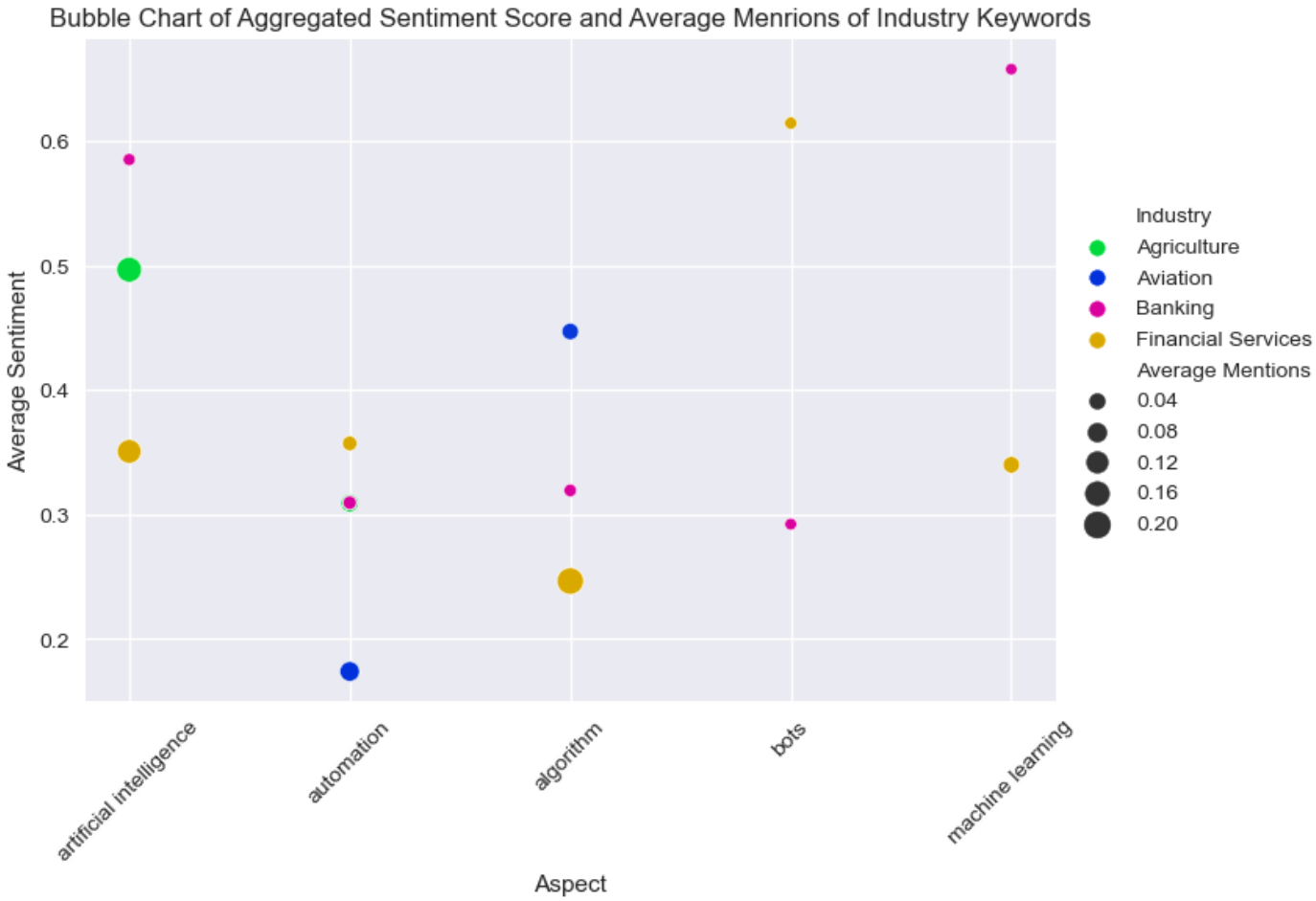

Figure 1: "Bubble Chart" showing Aggregated Sentiment Score and Average Mentions of Keywords per Industry, both averaged across each year.

4 Machine Learning Model

This section will deal with the ML model used to predict stock returns by including a sample of the input into the model which is a pandas DataFrame that undergoes extensive cleaning as discussed in Appendix A.7. For the model choice, two options were considered - "Boosting" and "Random Forest". A brief overview will be given of these models and then the performance of both will be considered after hyperparameter tuning using "Grid-Search Cross-Validation". Normally backtesting is used in this scenario but the limited data necessitated by the recent explosion of AI/ML development makes cross-validation a more viable method to compare performance of these models. The financial data spans back only as far as 2019, so this model will use data from 2019 onwards to predict 2020-2023 returns.

4.1 Data Transformation and Collection

4.1.1 Predictor Variables

Retrieving the aggregated sentiment scores data and financial data from the databases presents us with the raw data necessary for the model. Further transformations are required at this point to make the aggregated sentiment data usable.

Gross Domestic Product (GDP) is retrieved as a macro-economic indicator using the DBnomics API as this can fit directly into the input data. The final inclusion is a momentum indicator, Relative Strength Index is calculated at this point as it can also fit directly into the input data. The transformations and calculations necessary are listed below:

- **Total Sentiment:**
  - The aggregated sentiment scores are converted into "total sentiment scores" by multiplying the total count by the aggregated sentiment score.
  - The final transformation is achieved through dividing the total sentiment by the total number of *aspect counts*. This presents us with the "Average Sentiment".
  - This transformation is necessary to obtain a better input for the model as the sentiment score has been averaged by aspect-count per industry across the entire time-frame.
- **Financial Data**
  - The financial data can be imported directly from the database and requires no further cleaning or transformations as this data is already fully usable.
- **Gross Domestic Product (GDP):**
  - The DBnomics API allows easy access to a large amount of macroeconomic data. The GDP for the US in the specified time-frame (including 1-year prior) was extracted using the API.
  - This was subsequently transformed into the GDP growth-rate for each year to predict the next year of returns. The code for this is discussed in Appendix A.8.
- **Relative Strength Index (RSI):**
  - The RSI is a useful momentum oscillator and was necessary to include at this point to supplement the data. It can be calculated using the same data that was used to calculate the "log returns" which is discussed in the next sub-section.
  - Detailed analysis of how the momentum oscillator is calculated is contained in Appendix A.9.

Before looking at a sample of the predictor variables, it is necessary to look at the outcome variable, *i.e.* "Log Returns".

4.1.2 Outcome Variable

The outcome variable considered is the stock return for each stock. Due to the nature of returns there exists numerous outliers in the dataset, to negate the effect of these outliers the raw data is transformed into "Log Returns", making the distribution more normal. The returns data was calculated using the Yahoo Finance API by running a query on all the companies to obtain the "Adjusted Close" data for the specified time period and calculating the percentage change to obtain annual log returns which are used to predict the next year's annual log returns. Detailed discussion of the code required to produce the log returns is provided in Appendix A.7 also.

4.1.3 Input Data

In order to visualise the discussion of the variables in the previous sub-sections, and to provide a full account of the variables that were retrieved from the "Financial Data table" a sample of the input data is provided below:

artificial intelligence	automation	machine learning	BasicEPS	BTM	Gearing	CashToAssets	logMktCap	GDP	RSI	Return
0.000000	0.174054	0.000000	-4.50	0.922907	0.599272	0.156148	21.552091	0.092114	45.026530	-0.523096
0.000000	0.174054	0.000000	-0.43	0.067480	0.640043	0.134150	17.957376	0.092114	49.161003	-0.284026
0.585402	0.309676	0.657992	0.00	0.495232	0.076529	0.015566	21.777068	0.026520	52.450536	-0.012964
0.350863	0.357286	0.340114	0.03	0.460338	0.164667	0.436446	15.934952	-0.014990	41.855172	-0.178788
0.000000	0.309676	0.000000	3.14	1.028707	0.149366	0.006531	21.760744	-0.014990	59.319736	-0.554613
0.350863	0.357286	0.340114	4.88	0.548450	0.062941	0.101906	23.110672	-0.014990	50.983540	-0.219684
0.000000	0.309676	0.000000	0.87	0.732198	0.013000	0.099252	18.936640	-0.014990	42.815560	-0.262049
0.000000	0.309676	0.000000	1.68	0.899469	0.083913	0.020619	19.273039	-0.014990	39.937022	-0.566037
0.585402	0.309676	0.000000	1.05	1.365873	0.046002	0.100943	18.958131	0.107054	41.857405	0.852171
0.585402	0.309676	0.657992	0.00	0.882040	0.016110	0.073602	18.514811	0.026520	46.661045	-0.058417

This table excludes the average sentiment scores for "algorithm" and "bots" to enhance readability. The values for these were predominantly 0 by virtue of the random sampling to obtain this table.

4.2 Machine Learning Model Analysis

4.2.1 Model Selection

As discussed the two models being considered for regression are Boosting and Random Forest:

- **Boosting:**
  - Boosting is a machine learning ensemble technique where weak models are combined to create a stronger one. It trains models sequentially, with each subsequent model adjusting for the errors of the previous one.
- **Random Forest:**
  - Random Forest is also an ensemble technique that builds multiple decision trees and combines their predictions to produce a final output. Each decision tree in the forest is built using a random subset of the features and training data.

The two models are very similar and are both good models to use in a regression setting for stock returns prediction.

4.2.2 Hyperparameter Tuning & Model Performance

The input data, with sample shown above, was split into training data and testing data. The split for training data to test data was 70% to 30%. The necessary hyperparameters to use when fitting the model were obtained using Grid-Search Cross-Validation. This allows us to refit the model using combinations of each parameter to return the best values for those parameters so that the fitted model is optimised. Detailed discussion for this is provided in Appendix A.10 for both Boosting and Random Forest.

The metric to decide which model is best, and the core metric of this section, is "explained variance". This scoring metric was chosen because of how simple it is to understand - it indicates how well the model fits the data and is bound between 0 and 1 with higher scores indicating better performance. It is useful in the stock returns prediction setting because it measures the proportion of variance in the target variable that is explained by the model and is sensitive to the magnitude of the errors. It also aids against underfitting and overfitting making it ideal for this scenario.

- **Boosting - Model Performance:**
  - Best Explained Variance Score = 47.1%
  - Root Mean Squared Error (RMSE) = 15.77%

On the next page is a scatter plot of the predicted vs. actual log returns of the test observations. This shows the presence of outliers even after a log transformation which is unavoidable due to the nature of stock returns.

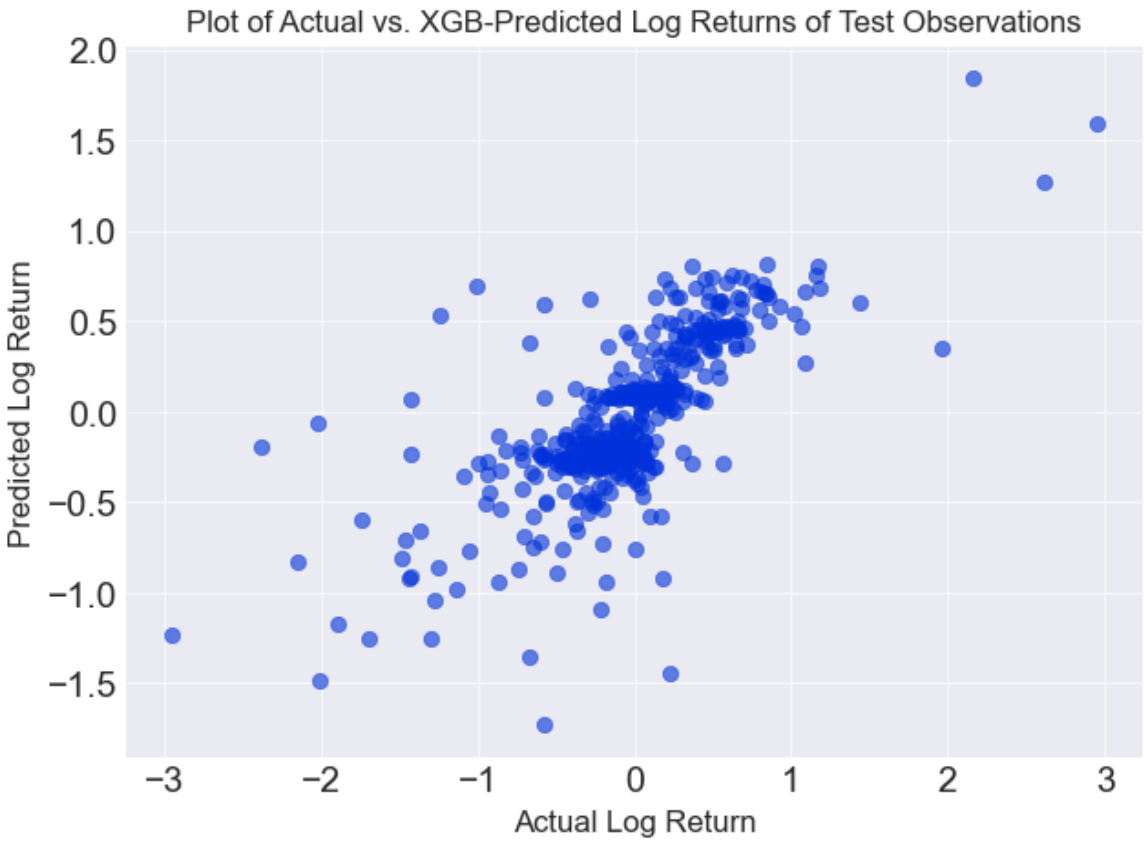

Figure 2: "Scatter Plot" showing Predicted Log Returns vs. Actual Log Returns for the Boosting Model.

- **Random Forest - Model Performance:**
  - Best Explained Variance Score = 47.5%
  - Root Mean Squared Error (RMSE) = 16.1%

On the next page is a scatter plot of the predicted vs. actual log returns of the test observations. This shows the presence of outliers even after a log transformation which is unavoidable due to the nature of stock returns.

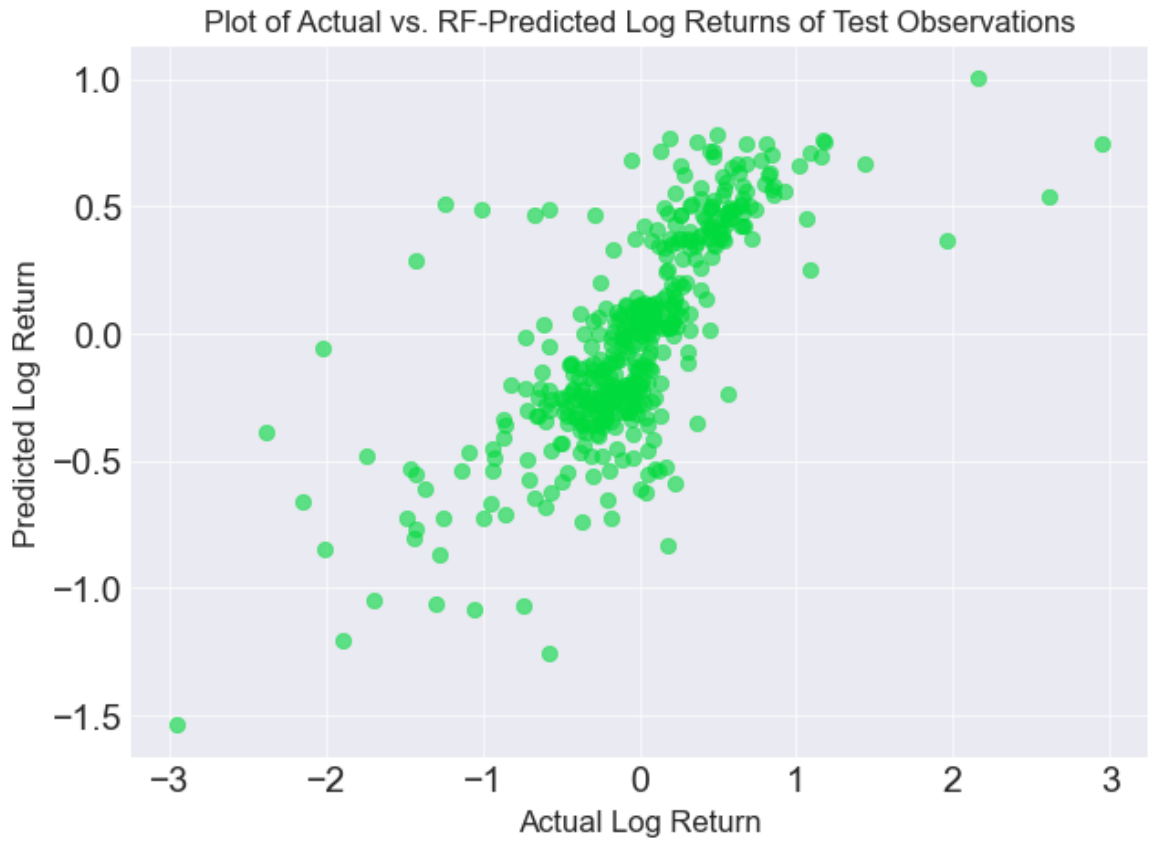

Figure 3: "Scatter Plot" showing Predicted Log Returns vs. Actual Log Returns for the Random Forest Model.

The RMSE values were included to give an idea of how many extreme errors are being penalised with both scores being acceptable. While either model would be suitable for implementation, Boosting is preferred as it is more computationally efficient and has a much lower RMSE on average.

5 Business Analysis

There are 2 main areas in which value is generated by the code written for this report. There is inherent value in the textual analysis performed on the annual 10-K reports. This code produces measurement of AI/ML integration as well as the sentiment towards AI/ML per industry. The code can easily be scaled up to include more industries with minimal changes, and the bubble chart provides an excellent understanding of these metrics through a visual aid. This does not need to be limited to only AI/ML research, the keywords themselves can be replaced for any other concepts of interest (such as ESG Scores).

The scalability and reusability of this code makes it worthwhile, with every dataset of interest being automatically stored in a database for ease of access later on. The benefits of such research can help the implementing business understand a market they are trying to enter, or better understand their competitors within the same industry. The value of this code depends on how it is implemented.

This data can also be utilised in the Machine Learning Model to predict stock returns prediction and this model performs well as it explains a significant proportion of the variance in log returns. The feature importance plot for Boosting is provided on the next page which shows that significant importance was attached to some of the sentiment score of the keywords.

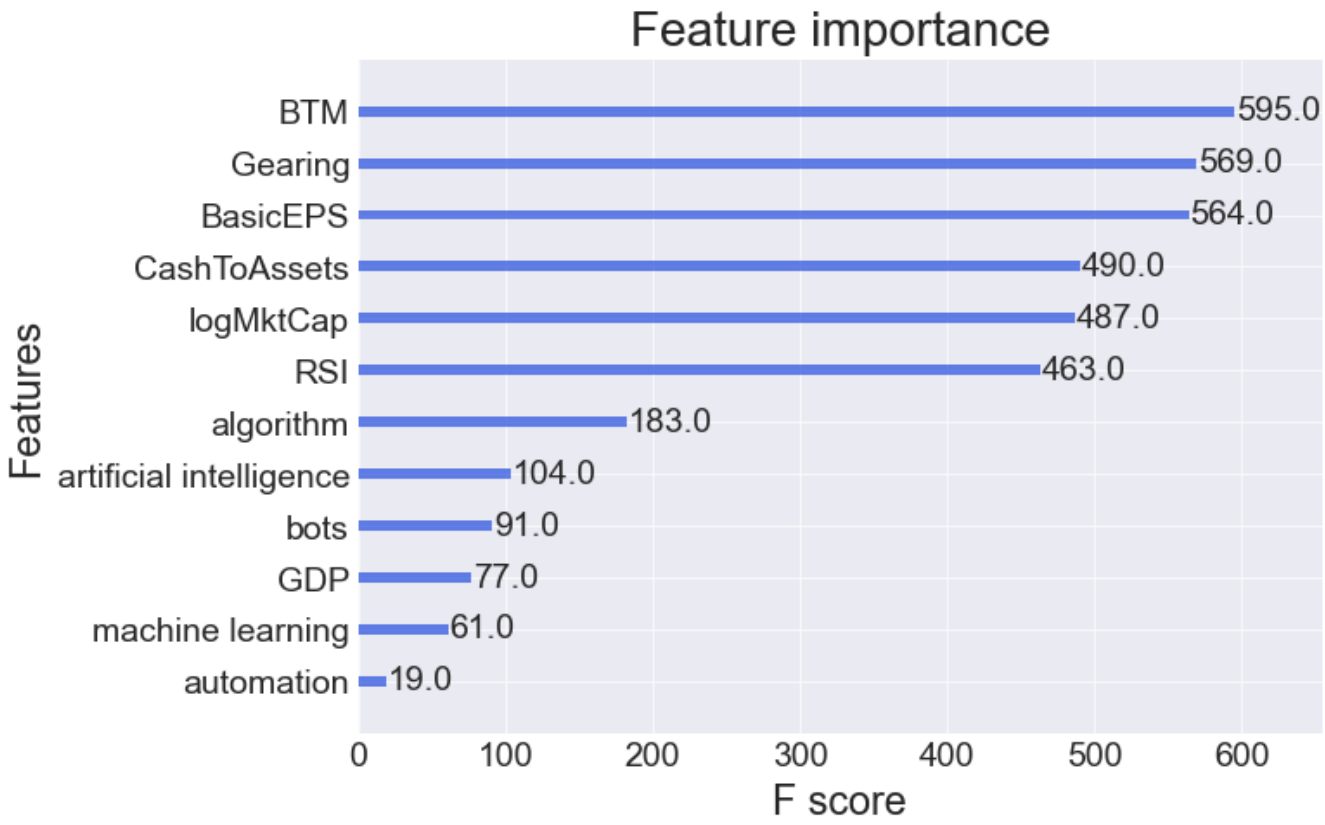

Figure 4: "Scatter Plot" showing Predicted Log Returns vs. Actual Log Returns for the Random Forest Model.

The real benefit of this model is that further sentiment analysis can be run using additional keywords to obtain further variables to use for prediction. This makes it possible to increase the accuracy of the model with minimal changes, and the new variables can also be used for industry/market analysis in a business context. With explained variance at approx. 50% this is a good model to implement but can certainly be improved. The limiting of the scope to only consider AI/ML integration and sentiment perhaps harmed model accuracy but provided valuable insight into industry-specific AI/ML integration and sentiment; it is possible to have the best of both worlds.

A Appendix A - Python Code

A.1 10-K Links - Web Scraping

```
import sqlite3 as lite
import pandas as pd
import requests
from bs4 import BeautifulSoup
import unicodedata
import re
import time
import sqlalchemy
# Parameters - Flow Control
SIC_list = ['6029', '6022', '6035', '6036', '6099', '4512', '4513', '4522', '4581', '6163',
            '6172', '6199', '6200', '6211', '6221', '6282', '0100', '0200', '0700', '0800', '0900']
form_type = '10-K'
filter_filings_from='20170101'
filter_filings_before=None

# Store CIK Codes & Accession Numbers in a Dictionary - Insert Scraped Data into DataBase
CIK_codes_dict = {}

for SIC in SIC_list:
    CIK_codes_dict[SIC] = {}
    endpoint = r'https://www.sec.gov/cgi-bin/browse-edgar'
    params_dict= {'SIC':SIC,
                'myowner':'exclude',
                'action':'getcompany',
                'output':'atom'}

    user_agent = {'User-agent' : '<evanmcgarry7@gmail.com>', 'Host': 'www.sec.gov'}

    response = requests.get(url=endpoint, headers=user_agent, params=params_dict)

    soup = BeautifulSoup(response.content, 'xml')
    entries = soup.find_all('entry')

    for entry in entries:
        CIK = entry.find('cik').text
        CIK_codes_dict[SIC][CIK] = {}
    links = soup.find_all('link',{'rel':'next'})

    while soup.find_all('link',{'rel':'next'}) != []:
        next_page_link = links[0]['href']
        time.sleep(0.5)
        response = requests.get(url=next_page_link, headers=user_agent)
        soup = BeautifulSoup(response.content, 'xml')
        entries = soup.find_all('entry')

        for entry in entries:
            CIK = entry.find('cik').text
            CIK_codes_dict[SIC][CIK] = {}
        links = soup.find_all('link',{'rel':'next'})

for SIC, CIK_dict in CIK_codes_dict.items():
    for CIK in CIK_dict.keys():
        endpoint = r'https://www.sec.gov/cgi-bin/browse-edgar'
        params_dict = {'action': 'getcompany',
            'CIK':str(CIK),
```

```
            'type':form_type,
            'datea':filter_filings_from,
            'dateb':filter_filings_before,
            'owner':'include',
            'output':'atom'}

        response = requests.get(url=endpoint, headers=user_agent, params=params_dict)
        soup = BeautifulSoup(response.content, 'xml')
        entries = soup.find_all('entry')
        conformed_name = soup.find('conformed-name').text
        # print("Completed: ", conformed_name)
        CIK_dict[CIK] = {}
        for entry in entries:
            if entry.find('filing-type').text == '10-K':
                accession_no = entry.find('accession-number').text
                CIK_dict[CIK][accession_no] = {}
                CIK_dict[CIK][accession_no]['company_name'] = conformed_name
                CIK_dict[CIK][accession_no]['cik'] = CIK
                CIK_dict[CIK][accession_no]['filing_date'] = entry.find('filing-date').text
                CIK_dict[CIK][accession_no]['filing_type'] = entry.find('filing-type').text
                CIK_dict[CIK][accession_no]['filing_href'] = entry.find('filing-href').text

                time.sleep(.5)
                response = requests.get(url=CIK_dict[CIK][accession_no]['filing_href'],
                headers=user_agent)
                soup = BeautifulSoup(response.content, 'xml')
                a = soup.select("a[href*=Archives]")
                link = a[0]['href']
                CIK_dict[CIK][accession_no]['direct_link_to_doc_filed'] = 'https://www.sec.gov' +
                link[8:] if '/ix?doc=' in link else 'https://www.sec.gov' + link

# Create SQLite Database
con = lite.connect('SEC_Filings.db')
with con:
    cur=con.cursor()
    cur.execute("DROP TABLE IF EXISTS Reports_Data")
    cur.execute("CREATE TABLE Reports_Data(SIC TEXT, CIK TEXT, Company TEXT,
    Filing_Date TEXT, Filing_Link TEXT, Accession_No TEXT PRIMARY KEY NOT NULL UNIQUE)");
    for SIC, CIK_dict in CIK_codes_dict.items():
        for CIK in CIK_dict.keys():
            if CIK_dict[CIK]:
                for accession_no in CIK_dict[CIK].keys():
                    rowi = (SIC, CIK, CIK_dict[CIK][accession_no]['company_name'],
                    CIK_dict[CIK][accession_no]['filing_date'],
                    CIK_dict[CIK][accession_no]['direct_link_to_doc_filed'], accession_no)
                    cur.execute("INSERT OR IGNORE INTO Reports_Data VALUES(?,?,?,?,?,?)", rowi)
```

In this code we have imported all necessary packages for the web-scraping. We set the parameters outside any loops to allow for flow control. The parameters include the SIC List which can be modified to include more sectors to denote more industries. The form-type can also be filtered, this could be changed to 10-Q for quarterly reports. The date can also be changed to set the time-frame.

Next it is necessary to set up a dictionary to store the CIK codes that are unique to each company. Another dictionary is set up within the dictionary to allow the SIC to act as the key. With this set up we begin to request the necessary webpages by setting the endpoint and all necessary parameters to access the company listings in XML format. We can then parse this page with BeautifulSoup to allow us to access the different elements/tags. We access each entry using CIK, meaning that each entry will be the filings for that specific company. However, each page only contains 40 filings that include filings other than the 10-Ks. As a result we need to obtain the link for the "next page" element on each webpage - which is denoted by ("rel": "'next'"). This is looped through to obtain each page of filings for each individual company, with a sleep timer of 0.5 seconds to prevent the code from

timing out due to an overload of requests.

Now that we have all necessary pages, we parse those webpages to allow us to retrieve the 10-Ks only for each company, the company being denoted by it's "conformed name". The 10-Ks are unique due to their "accession-number" which we retrieve through further parsing. All of the necessary data is then stored in the original dictionary of dictionaries that is defined outside the loop. The link to the 10-K is of the utmost importance here and is assigned to the variable "'link'" using the unique identifier "filing-href" in the XML. Before assigning the link to the dictionary, there is a potential pitfall which must be overcome. In order to ensure every single 10-K is readable we have to remove iXBRL formatting from any links that may have this formatting. The identification of this potential pitfall, as well as the numerous unique identifiers/tags used throughout this code was made easier thanks to a similar project undertaken by SigmaCoding on Kaggle.

With all of the necessary data stored in the dictionary, we upload this data to the database named "SEC Filings.db" as this scraping code can take a significant amount of time to complete. In case of any errors in subsequent analysis we can simply query this data from the SEC Filings database rather than wait for the code to run.

A.2 10-K Data Retrieval and Cleaning

```
import sqlite3 as lite
import pandas as pd
import requests
from bs4 import BeautifulSoup
import unicodedata
import re
import time
import sqlalchemy

# Query the 10-K Filings Database
db_name = "SEC_Filings.db"
table_name = "Reports_Data"

engine = sqlalchemy.create_engine('sqlite:///' + db_name, execution_options={"sqlite_raw_colnames": True})
df = pd.read_sql_table(table_name, engine)
df
user_agent = {'User-agent' : '<evanmcgarry7@gmail.com>', 'Host': 'www.sec.gov'} ## REMOVE LATER

# Retrieve Filing Documents as HTML and Clean Data

# Function to Clean HTML
def restore_windows_1252_characters(restore_string):
    """
    Replace C1 control characters in the Unicode string s by the
    characters at the corresponding code points in Windows-1252,
    where possible.
    """
    def to_windows_1252(match):
        try:
            return bytes([ord(match.group(0))]).decode('windows-1252')
        except UnicodeDecodeError:
            # No character at the corresponding code point: remove it.
            return ''
    return re.sub(r'[\u0080-\u0099]', to_windows_1252, restore_string)

# Create New DataFrame Column to Store the Normalised Text
df['Filing_Document_Text'] = ''

# Loop Through Filing Links
for i in range(0, len(df)):
    response = requests.get(df['Filing_Link'][i], headers=user_agent)
```

```
    soup = BeautifulSoup(response.content, 'lxml')
    filing_document = soup.find('body')
    filing_doc_text = filing_document.extract()
    filing_doc_string = str(filing_doc_text)
    text_bytes = bytes(filing_doc_string, 'utf-8')
    doc_soup = BeautifulSoup(text_bytes, 'html5')
    doc_text = doc_soup.html.body.get_text(' ', strip = True)
    doc_text_normalise = restore_windows_1252_characters(unicodedata.normalize('NFKD', doc_text))

    # Additional Cleaning Steps
    doc_text_normalise = doc_text_normalise.replace('Â', '').replace('  ', ' ').replace('\n', ' ')
    doc_text_normalise = doc_text_normalise.lower()

    # Store Cleaned Data in DataFrame
    df['Filing_Document_Text'][i] = [doc_text_normalise]
# Convert 10-K Data to String
df['Filing_Document_Text'] = df['Filing_Document_Text'].astype(str)

# Insert into SQLite Database
# Create SQLite Table
db_name2 = "SEC_Filings.db"
table_name2 = "10K_Data"

engine2 = sqlalchemy.create_engine('sqlite:///' + db_name, execution_options={"sqlite_raw_colnames": True}
df.to_sql(table_name2, engine2, if_exists='replace', index=False)
```

Running a query on the previously established database retrieves the stored information and, most importantly, the links to the 10-K. Now it is necessary to use those links to retrieve the 10-Ks. SigmaCode provided a function to clean the HTML which replaces C1 control characters with their corresponding Windows-1252 characters making the 10-Ks easier to read for later sentiment analysis.

A new column is set up in the DataFrame that we retrieved from the database and this will be used to store the normalised text of the 10-Ks. The filing links are looped through to request the actual 10-Ks, of which there are 3917. The "body" of the 10-Ks is accessed, the text extracted and converted to string format. The data is still unusable as it is, therefore further cleaning is done in the loop. The string is converted to bytes using the utf-8 format and this is where the text is cleaned using the function defined at the beginning. The text is then normalised by removing any special characters as they hold no value. Double spaces and line breaks are converted to single spaces and finally the normalised text is set to lower-case to allow for words to easily be identified.

The 10-Ks have successfully been converted into a usable string format. This code finishes with storing the 10-K text in a database so it can be retrieved for sentiment analysis.

A.3 Financial Data - Ticker to CIK Mapping

```
import pandas as pd
import requests
import sqlalchemy

# Retrieve Mentions Data as DataFrame
db_name = "SEC_Filings.db"
table_name = "10K_Data"

engine = sqlalchemy.create_engine('sqlite:///' + db_name, execution_options={"sqlite_raw_colnames": True})
df = pd.read_sql_table(table_name, engine)

df = df.drop(['SIC', 'Filing_Date', 'Filing_Link', 'Accession_No', 'Filing_Document_Text'], axis=1)
pd.set_option('display.max_rows', None)

headers = {'User-Agent': "evanmcgarry7@gmail.com"}
```

```
tickers_cik = requests.get("https://www.sec.gov/files/company_tickers.json", headers=headers)

tickers_cik = pd.json_normalize(pd.json_normalize(tickers_cik.json(), max_level=0).values[0])
tickers_cik["cik_str"] = tickers_cik["cik_str"].astype(str).str.zfill(10)
tickers_cik.columns= ['CIK', 'Ticker', 'Company_Name']
tickers_cik.drop(['Company_Name'], axis=1, inplace=True)

tickers_cik['CIK'].astype(int)
df['CIK'].astype(int)

df_full = df.merge(tickers_cik, how='left', on='CIK')
df_full.loc[df_full['Ticker'].isna()]
df_full.dropna(inplace=True)
df_full.reset_index(drop=True, inplace=True)
df_full = df_full.drop_duplicates()

tickers_list = []
for company in df_full['Company'].unique():
    tickers_list.append(df_full[df_full['Company']==company].sort_values
    (by='Ticker').head(1).values.squeeze())

df_tickers = pd.DataFrame(tickers_list, columns=['CIK', 'Company', 'Ticker'])
# Insert Tickers Data into Table in Database
db_name2 = "SEC_Filings.db"
table_name2 = "Ticker_Table"

engine2 = sqlalchemy.create_engine('sqlite:///' + db_name, execution_options={"sqlite_raw_colnames": True}
df_tickers.to_sql(table_name2, engine2, if_exists='replace', index=False)
```

In order to collect Financial Data relating to each company of interest it is necessary to map each company onto its ticker. Firstly we read in the 10K Data from the database and drop everything apart from the company name and CIK code. Fortunately, the SEC Edgar website provides a ".json" file that maps the CIK and tickers. We request this file directly from the website allowing for seamless integration into the code. The data retrieved from the json file is normalised to enable it for use.

The issue with this file is that the CIK codes used for mapping do not have the necessary "0"'s before each CIK code to match the database. This is an inconsistency on the SEC website but is easily fixed as we use ".zfill(10)" to fill in the missing zero values before the unique numeric values of the CIK codes. The DataFrame is then set up with the company name being dropped from the columns, leaving the CIK codes and their corresponding Ticker in the DataFrame. We then merge the DataFrame containing the necessary CIK codes onto the DataFrame obtained from the SEC json file by using the CIK Code as the common value. Any missing values are then dropped and duplicates are removed. We then set up a new list by looping through the merged DataFrame for the unique tickers and append these to the list which is used to create the final DataFrame that we input into the database.

There was a strange issue with dimensions of the pandas DataFrame which required us to run some of this, what would otherwise be seen as, superfluous code. However, it was necessary to allow successful storing of this very important data in the database.

A.4 Financial Data - Creation and Storing

```
import sqlalchemy
import pandas as pd
import yahooquery as yq
import sqlite3 as lite
from dateutil.relativedelta import relativedelta
import numpy as np
import datetime as dt

# Retrieve data from SQL database
```

```
engine = sqlalchemy.create_engine('sqlite:///' + "SEC_Filings.db",
execution_options={"sqlite_raw_colnames": True})
df_ticker = pd.read_sql_table('Ticker_Table', engine)

# Create empty dataframes to store financial data
df_fin_bal = pd.DataFrame()
df_fin_inc = pd.DataFrame()
df_fin_stock = pd.DataFrame()
bal_vars = ['asOfDate', 'CashAndCashEquivalents','TotalAssets','TotalDebt','TangibleBookValue',
'OrdinarySharesNumber']
inc_vars = ['asOfDate', 'BasicEPS']
stock_vars = ['date', 'adjclose']
# bal_sheet_error = []
bal_sheet_error_tickers = []
inc_stat_error_tickers = []

# Retrieve financial data from Yahoo Finance
for ticker in df_ticker['Ticker']:
    tick = yq.Ticker(ticker)
    df_bal_tick = tick.balance_sheet(frequency='a', trailing=False)
    df_inc_tick = tick.income_statement(frequency='a', trailing=False)
    try: # some companies without pricing info included in list
        df_stock_tick = tick.history(interval='1d', start='2019-04-01').reset_index(level=1)
        df_stock_tick['date'] = df_stock_tick['date'] - relativedelta(days=1)
    except:
        continue
    try:
        df_bal_tick = df_bal_tick[bal_vars]
        df_fin_bal = pd.concat([df_fin_bal, df_bal_tick], axis=0)

    except:
        if type(df_bal_tick) == pd.DataFrame:
            # bal_sheet_error.append(f'{ticker}: Missing columns {[var for var in bal_vars if var not in d
            bal_sheet_error_tickers.append(ticker)
        else:
            # bal_sheet_error.append(f'{ticker} Balance sheet not Available')
            bal_sheet_error_tickers.append(ticker)

    try:
        df_inc_tick = df_inc_tick[inc_vars]
        df_fin_inc = pd.concat([df_fin_inc, df_inc_tick], axis=0)

    except:
        if type(df_inc_tick) == pd.DataFrame:
            # print([var for var in inc_vars if var not in df_inc.columns])
            inc_stat_error_tickers.append(ticker)
        else:
            inc_stat_error_tickers.append(ticker)

    df_stock_tick = df_stock_tick[stock_vars]
    df_fin_stock = pd.concat([df_fin_stock, df_stock_tick], axis=0)


df_fin_bal.reset_index(drop=False, inplace=True)
df_fin_inc.reset_index(drop=False, inplace=True)
df_fin_stock.reset_index(drop=False, inplace=True)
# Remove companies with 0 assets or 0 cash
df_fin_bal = df_fin_bal.loc[(df_fin_bal['TotalAssets']!=0) & (df_fin_bal['CashAndCashEquivalents'])!=0]
df_fin_inc.fillna(value=0, inplace=True)
```

```
# Only take rows with if matched in both dataframes
df_fin_joined = df_fin_inc.merge(df_fin_bal, how='inner', on=['symbol', 'asOfDate'])
df_fin_stock = pd.DataFrame()

for ticker in df_ticker['Ticker']:
    tick = yq.Ticker(ticker)
    try: # some companies without pricing info included in list
        df_stock_tick = tick.history(interval='1mo', start='2019-04-01').reset_index(level=1)
        df_stock_tick['date'] = df_stock_tick['date'] - relativedelta(days=1)
    except:
        continue
    df_stock_tick = df_stock_tick[stock_vars]
    df_fin_stock = pd.concat([df_fin_stock, df_stock_tick], axis=0)
df_fin_stock.reset_index(drop=False, inplace=True)

df_fin_stock = df_fin_stock.loc[df_fin_stock['date'].apply(type)==dt.date]
df_fin_stock['date'] = df_fin_stock['date'].astype('datetime64[ns]')

df_fin_joined = df_fin_joined.merge(df_fin_stock, how='inner', left_on=['symbol', 'asOfDate'],
right_on=['symbol', 'date'])
df_fin_cleaned = pd.DataFrame()
# backfill missing values based on company
for ticker in df_fin_joined['symbol'].unique():
    df_fin_cleaned = pd.concat([df_fin_cleaned, df_fin_joined.loc[df_fin_joined['symbol']==ticker]
    .sort_values(by='asOfDate', ascending=True).fillna(method='ffill')])

# Remove rows with incomplete data
missing_ticks = df_fin_cleaned[df_fin_cleaned.isnull().any(axis=1)]['symbol'].unique()
df_fin_cleaned = df_fin_cleaned.loc[~df_fin_cleaned['symbol'].isin(missing_ticks)].reset_index(drop=True)

df_fin_cleaned = df_ticker.merge(df_fin_cleaned, how="right",
left_on="Ticker", right_on="symbol").drop("symbol",axis=1)
df_fin_cleaned['MarketCap'] = df_fin_cleaned['OrdinarySharesNumber'] * df_fin_cleaned['adjclose']
df_fin_cleaned['BTM'] = df_fin_cleaned['TangibleBookValue']/df_fin_cleaned['MarketCap']
df_fin_cleaned['Gearing'] = df_fin_cleaned['TotalDebt']/df_fin_cleaned['TotalAssets']
df_fin_cleaned['CashToAssets'] = df_fin_cleaned['CashAndCashEquivalents']/df_fin_cleaned['TotalAssets']
df_fin_cleaned['logMktCap'] = df_fin_cleaned['MarketCap'].apply(np.log)
df_fin_cleaned.drop(['CashAndCashEquivalents', 'TotalAssets', 'TotalDebt', 'TangibleBookValue',
'OrdinarySharesNumber', 'adjclose', 'MarketCap'], axis=1, inplace=True)
# Add to database
conn = lite.connect('Financial.db')
conn.close()
engine_fin = sqlalchemy.create_engine('sqlite:///' + 'Financial.db', execution_options={"sqlite_raw_colnam
df_fin_cleaned.to_sql("FinancialDataJoined", engine_fin, if_exists='replace', index=False)
```

The table mapping CIK codes and Tickers is imported from the DataBase and used to obtain the financial data. We set up empty dataframes to store the historical information from the balance sheet, income sheet and stock data separately. In addition, not all of the companies that we are examining will have historical information available from Yahoo Finance and these are filtered out with their Tickers stored in lists in case we need to identify them.

The loop is then constructed to retrieve the previous 5 years of financial data from the balance sheet and income statements of each company by virtue of their tickers. The items that we need to retrieve the values of are contained in the lists "bal-vars" and "inc-vars". The stock data is also retrieved in the loop as this is necessary later for the transformations. Any errors are filtered out through the "try" statements. This data is then concatenated into a single DataFrame with the items of interest as the column names. Some companies are noted to have 0 assets or cash (or both) in the balance sheet data and these are removed in the code after the loop. Some companies have NaN values for their EPS which is forward-filled in the code to facilitate analysis over the, already diminished, time-frame. These two DataFrames are merged based on the date and ticker, dropping any superfluous data that will not be usable due to a mismatch in timeframe between the balance sheet and income statement.

Next we query Yahoo Finance to obtain the historical stock data for each of the tickers in the database. A similar loop to above is constructed except this time the timeframe is specified since the queried stock data can stretch back much further than 2019. The Adjusted Close price for the end of each year from 2019 onwards is extracted in the loop and appended to the DataFrame to store the *Date* and *Adjusted Close* price of each stock. This data is then merged onto the final DataFrame that contains the financial statement data, using the ticker and date to merge correctly. Any missing values are back-filled to ensure that the model does not suffer from missing values. Any rows with incomplete data are subsequently removed and the final DataFrame has been cleaned extensively which allows it to be used for transformations of the data to calculate the key predictors.

The main financial predictors are calculated using this cleaned DataFrame with the code shown. Market Capital is calculated using the balance sheet and stock data. The rest of the predictors are calculated using the values obtained from the balance sheet for each category of interest. The raw data is then dropped from the DataFrame and the transformed data is maintained and inserted into the database for future use in the ML model.

A.5 Textual Analysis - Sentiment Score Aggregation and Keyword Word Count

```
import pandas as pd
import sqlalchemy
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer
from nltk.tokenize import word_tokenize, sent_tokenize
nltk.downloader.download('vader_lexicon')
nltk.downloader.download('punkt')

# Import the 10-K Data from the Database
db_name = "SEC_Filings.db"
table_name = "10K_Data"

engine = sqlalchemy.create_engine('sqlite:///' + db_name, execution_options={"sqlite_raw_colnames": True})
df = pd.read_sql_table(table_name, engine)

# Clean DataFrame
agri = ['0100', '0200', '0900']
fin = ['6172', '6199', '6200', '6211', '6221', '6282']
air = ['4513', '4522', '4581']
bank = ['6022', '6029', '6035', '6036', '6099']

df['Industry'] = ''
df.loc[df['SIC'].isin(agri), 'Industry'] = "Agriculture"
df.loc[df['SIC'].isin(fin), 'Industry'] = "Financial Services"
df.loc[df['SIC'].isin(air), 'Industry'] = "Aviation"
df.loc[df['SIC'].isin(bank), 'Industry'] = "Banking"

df = df.drop(['Filing_Link', 'Accession_No', 'SIC', 'Company'], axis=1)
# Sentiment Analysis
# Set 'Aspect' Words
word_search_list_2 = [{'artificial intelligence': ['artificial intelligence', 'artificial-intelligence',
                        ' ai.', ' ai,', ' ai ']},
                      {'deep learning': ['deep learning', 'deep-learning']},
                      {'computer vision': ['computer vision', 'computer-vision']},
                      'bots',
                      'automation',
                      {'machine learning': ['machine learning', 'machine-learning']},
                      'algorithm']
# aspects = [list(x.values())[0] if isinstance (x, dict) else x for x in word_search_list_2]

# Convert 10-K Text to String
```

```
# review_text = df['Filing_Document_Text'][0]

# State Sentiment Analyzer - Rule-based Approach
sia = SentimentIntensityAnalyzer()

# Create DataFrame to Store Output
df_sentiments = pd.DataFrame(columns=['Industry', 'Year', 'Aspect', 'Count', 'Sentiment'])

df['Filing_Date'] = pd.to_datetime(df['Filing_Date'], format='%Y-%m-%d')
# Perform Sentiment Analysis

out_list = []
for i, review_text in enumerate(df['Filing_Document_Text']):

    # Create Dictionary to Store Sentiment Scores
    aspect_sentiments = {}
    for aspect_item in word_search_list_2:
        if isinstance(aspect_item, dict):
            aspect = list(aspect_item.keys())[0]
            aspect_sentiments[aspect] = {'count': 0, 'sentiment': 0}
        else:
            aspect_sentiments[aspect_item] = {'count': 0, 'sentiment': 0}

    year = df.loc[i,'Filing_Date'].year
    industry = df.loc[i,'Industry']

    sentences = sent_tokenize(review_text)
    for sentence in sentences:
        for aspect_item in word_search_list_2:
            if isinstance (aspect_item, dict):
                aspect = list(aspect_item.keys())[0]
                keywords = list(aspect_item.values())[0]
                for keyword in keywords:
                    if keyword in sentence.lower():
                        sent_score = sia.polarity_scores(sentence)
                        aspect_sentiments[aspect]['count'] += 1
                        aspect_sentiments[aspect]['sentiment'] += sent_score['compound']
            else:
                aspect = aspect_item
                if aspect in sentence.lower():
                    sent_score = sia.polarity_scores(sentence)
                    aspect_sentiments[aspect]['count'] += 1
                    aspect_sentiments[aspect]['sentiment'] += sent_score['compound']

    for aspect in aspect_sentiments:
        count = aspect_sentiments[aspect]['count']
        sent = aspect_sentiments[aspect]['sentiment']
        out_data = {'Industry': industry, 'Year': year, 'Aspect': aspect, 'Count': count, 'Sentiment': sen
        out_list.append(out_data)

df_sentiments = pd.DataFrame.from_dict(out_list)
pd.set_option('display.max_rows', None)

df_agg = df_sentiments[['Industry', 'Aspect', 'Year', 'Count', 'Sentiment']].groupby(['Industry',
'Aspect', 'Year']).sum()
df_agg['Aggregated Sentiment'] = 0
df_agg.loc[df_agg['Count']!=0, 'Aggregated Sentiment'] = df_agg['Sentiment']/df_agg['Count']
df_agg.reset_index(drop=False, inplace=True)
df_agg.drop('Sentiment', axis=1, inplace=True)
```

```
df_agg = df_agg.loc[~df_agg['Aspect'].isin(['computer vision', 'deep learning'])].reset_index(drop=True)
df_agg=df_agg.loc[df_agg['Count'] > 5].reset_index(drop=True)

industry_counts = df_sentiments[['Industry', 'Year']].groupby('Industry').count()/len(word_search_list_2)
for industry in df_agg['Industry'].unique():
    df_agg.loc[df_agg['Industry']==industry, 'Average Mentions'] =
    df_agg['Count']/industry_counts.loc[industry].values

df_agg.to_sql("Aggregated_Sentiment_Scores", engine, if_exists='replace', index=False)
```

This code utilises the text of the 10-Ks which has been cleaned and stored in the database. The 10-K data is retrieved from the database and immediately cleaned to group the SIC codes into their respective industries. Additionally, unnecessary elements at this point in the process are dropped. They keywords are listed in the "Word Search List", which is set up as a dict in a list since numerous terms are used to refer to the same aspect in the 10-Ks. Without this we would be overestimating the number of mentions, and by extension, the level of AI/ML integration.

The 10-K data is analysed using Vader's "Aspect-Based Sentiment Analysis" that uses a "rule-based approach". The sentiment analyser is set up and the DataFrame used to store the variables of interest is created. The loop is then initiated to sort through each of the 10-Ks (of which there are 3917). First it is necessary to create a dictionary that stores both the number of times each keyword is mentioned (by reference to the values of the dict instead of the key) and the associated sentiment with each mention of the keyword. The year and industry to which this 10-K belongs are stored in variables to allow for easy identification in the final DataFrame.

A loop is set up to tokenize each sentence in the individual 10-Ks which allows for sentiment analysis. The aspects are the keys of the dictionary created before whilst the keywords are the values associated with this key - this is necessary to ensure that the count is attached to the key of the dictionary only and prevent double-counting. The sentiment score for each aspect is also stored in compound form which means it aggregates the "negative", "positive" and "neutral" sentiment scores to provide a single figure that provides overall sentiment and is bound between -1 and 1. The initial dictionary stores these values by adding on the count for each instance of the loop as well as adding on the compound score for each aspect's sentiment. The data being calculated is assigned to individual variables which are then appended on to the output dictionary for each instance of the loop.

This dictionary is converted into a DataFrame to allow us to aggregate the sentiment scores for each aspect per industry. This is done by taking the aspects where the total count is not equal to 0 and dividing the sentiment score by the overall count. This provides us with an aggregated sentiment that must also be bound between -1 and 1. The data is further cleaned by removing any aspects with a count less than 5 as this will essentially be irrelevant in the model and for data visualisation purposes. The final step in the cleaning process is to calculate the average mentions per industry as this is crucial for data visualisation in the next stage. This cleaned data is then stored in the database as it will be necessary to call for data visualisation and as input data for the ML model.

A.6 Data Visualisation - Bubble Chart

```
import matplotlib.pyplot as plt
import seaborn as sns
import sqlalchemy
import pandas as pd

engine = sqlalchemy.create_engine('sqlite:///' + 'SEC_Filings.db', execution_options={"sqlite_raw_colnames
df_sentiments = pd.read_sql_table('Aggregated_Sentiment_Scores', engine)

aspect_counts=df_sentiments[['Industry', 'Aspect', 'Count']].groupby(['Industry', 'Aspect']).sum().values
df_sentiments['Total Sentiment'] = df_sentiments['Count']*df_sentiments['Aggregated Sentiment']
avg_sentiments = df_sentiments[['Industry', 'Aspect', 'Total Sentiment']].groupby(['Industry', 'Aspect']).
avg_sentiments.reset_index(drop=False, inplace=True)
avg_sentiments.rename(columns={'Total Sentiment': 'Average Sentiment'}, inplace=True)

df_sentiments['Total Mentions'] = df_sentiments['Count']*df_sentiments['Average Mentions']
df_sentiments.drop('Average Mentions', axis=1, inplace=True)
avg_mentions = df_sentiments[['Industry', 'Aspect', 'Total Mentions']].groupby(['Industry', 'Aspect']).sum
```

```
avg_mentions.reset_index(drop=False, inplace=True)
avg_mentions.rename(columns={'Total Mentions': 'Average Mentions'}, inplace=True)

df_sentiments_avg = df_sentiments.merge(avg_sentiments, on=['Industry', 'Aspect'])
df_sentiments_avg = df_sentiments_avg.merge(avg_mentions, on=['Industry', 'Aspect'])
df_sentiments_avg.drop(['Aggregated Sentiment', 'Total Sentiment', 'Total Mentions'], axis=1, inplace=True

colors_hex = ['#00da3c', '#0031da', '#da009e', '#daa900']
plt.style.use('seaborn-darkgrid')
ax = sns.scatterplot(data=df_sentiments_avg, x="Aspect", y="Average Sentiment", size="Average Mentions",
                palette=colors_hex, hue="Industry", legend=True, sizes=(30, 150), alpha=0.8)
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.xticks(rotation=45)
plt.title("Bubble Chart of Aggregated Sentiment Score and Average Menrions of Industry Keywords")
plt.show()
```

The data obtained from the sentiment analysis loop is imported from the database and can be altered to create a bubble chart that plots the aggregated sentiment by industry for each aspect with each point in the scatterplot differing in size based on the average mentions of the keyword in that industry.

A.7 Model Input Data - Retrieval and Cleaning

```
import xgboost as xgb
import pandas as pd
import sqlalchemy
import yahooquery as yq
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.ensemble import RandomForestRegressor
import numpy as np
import requests
import matplotlib.pyplot as plt
import datetime as dt
import talib as ta

# Importing the data
engine = sqlalchemy.create_engine('sqlite:///' + "Financial.db",
execution_options={"sqlite_raw_colnames": True})
df_financial = pd.read_sql_table('FinancialDataJoined', engine)

engine = sqlalchemy.create_engine('sqlite:///' + "SEC_Filings.db",
execution_options={"sqlite_raw_colnames": True})
df_sentiments = pd.read_sql_table('Aggregated_Sentiment_Scores', engine)
df_SIC = pd.read_sql_table("10K_Data", engine)
df_SIC = df_SIC[['SIC', 'CIK']]

agri = ['0100', '0200', '0900']
fin = ['6172', '6199', '6200', '6211', '6221', '6282']
air = ['4513', '4522', '4581']
bank = ['6022', '6029', '6035', '6036', '6099']

df_SIC['Industry'] = ''
df_SIC.loc[df_SIC['SIC'].isin(agri), 'Industry'] = "Agriculture"
df_SIC.loc[df_SIC['SIC'].isin(fin), 'Industry'] = "Financial Services"
df_SIC.loc[df_SIC['SIC'].isin(air), 'Industry'] = "Aviation"
df_SIC.loc[df_SIC['SIC'].isin(bank), 'Industry'] = "Banking"

# Merging the Sentiment data
aspect_counts=df_sentiments[['Industry', 'Aspect', 'Count']].groupby(['Industry', 'Aspect']).sum().values
df_sentiments['Total Sentiment'] = df_sentiments['Count']*df_sentiments['Aggregated Sentiment']
```

```
avg_sentiments =
df_sentiments[['Industry', 'Aspect', 'Total Sentiment']].groupby(['Industry', 'Aspect']).sum()/aspect_coun
avg_sentiments.reset_index(drop=False, inplace=True)
avg_sentiments.rename(columns={'Total Sentiment': 'Average Sentiment'}, inplace=True)

df_sentiments_avg = df_sentiments.merge(avg_sentiments, on=['Industry', 'Aspect'])
df_sentiments_avg.drop(['Aggregated Sentiment', 'Average Mentions', 'Total Sentiment'],
axis=1, inplace=True)

# Merging the Financial data
tab = df_SIC.merge(df_sentiments_avg, on="Industry", how="left")

df_pivot = pd.pivot_table(tab, values='Average Sentiment', index=['CIK', 'Year'],
                          columns=['Aspect'], fill_value=0).reset_index(drop=False)
ind = tab[['CIK', 'Year', 'Industry']].drop_duplicates(ignore_index=True)

df_pivot = ind.merge(df_pivot, on=['CIK', 'Year'])
df_financial['Year'] = df_financial['asOfDate'].dt.year+1
df_pivot_joined = df_pivot.merge(df_financial, on=['CIK', 'Year'])

ticker = df_pivot_joined.pop('Ticker')
company  = df_pivot_joined.pop('Company')

df_pivot_joined.insert(1, 'Company', company)
df_pivot_joined.insert(2, 'Ticker', ticker)
df_pivot_joined.drop(['asOfDate', 'date'], axis=1, inplace=True)

# Retrieving the stock data
df_stock = pd.DataFrame()

for ticker in df_pivot_joined['Ticker'].unique():
    tick = yq.Ticker(ticker)
    df_stock = pd.concat([df_stock, tick.history(start='01-01-2019', interval='3mo')[0::4]], axis=0)

df_stock = df_stock.pop("adjclose").reset_index(drop=False)
df_stock["Year"] = df_stock["date"].apply(lambda x: x.year)
df_stock.pop("date")

# log return
for ticker in df_stock['symbol'].unique():
    df_stock.loc[df_stock['symbol']==ticker, 'Return'] = np.log(df_stock.loc[df_stock['symbol']==ticker, '

df_stock.dropna(inplace=True)
df_stock.drop('adjclose', axis=1, inplace=True)
df_stock = df_stock.rename(columns={"symbol": "Ticker"})
```

All of the necessary raw data is imported from the databases. The sentiment score is cleaned to make sure we are working with our desired total sentiment per industry per aspect - this code is perhaps redundant but affords certainty we are inserting the correct input data. This is then merged with the financial data which has already been cleaned for use. The only step necessary is to increase the Year by 1 as this denotes the year we will be predicting, rather than the year that the data pertains to.

Any unncessary columns are dropped from the merged DataFrame. The stock data is then queried from Yahoo Finance to obtain the adjusted close values which are now used to calculate the log returns - this is done for the period preceding the period we are predicting. This is done in the same way as before except this time the period has changed as it is being used in the prediction context. This data is merged onto the final DataFrame that contains all of the raw input data after we have calculated GDP and RSI.

A.8 Calculation of GDP

```
# Function for API call
def checkindicator(url):
    r= requests.get(url)
    r = r.json()
    periods = r['series']['docs'][0]['period']
    values = r['series']['docs'][0]['value']
    dataset = r['series']['docs'][0]['dataset_name']
    indicators = pd.DataFrame(values,index=periods)
    indicators.columns = [dataset]
    return indicators

# Download GDP from DBnomics
gdp = checkindicator('https://api.db.nomics.world/v22/series/OECD/DP_LIVE/USA.GDP.TOT.MLN_USD.A?observatio

# Convert Index to Datetime
gdp.reset_index(inplace=True)
gdp.rename(columns={'index': 'Year', 'OECD Data Live dataset': 'GDP'},inplace=True)
gdp['Year'] = pd.to_datetime(gdp['Year'])

# Add Row for recent GDP
new_row = {'Year': '2022-01-01', 'GDP': 25462722.0} # https://fred.stlouisfed.org/series/GDP
gdp.loc[len(gdp)] = new_row
new_row2 = {'Year': '2023-01-01', 'GDP': 26137992.0} # https://fred.stlouisfed.org/series/GDP
gdp.loc[len(gdp)] = new_row2
gdp['Year'] = pd.to_datetime(gdp['Year'])

# # Only Retain Year in Date
gdp['Year'] = gdp['Year'].dt.year
# Clean DataFrame
gdp = gdp.loc[gdp['Year'] > 2018]
gdp['GDP'] = gdp['GDP'].pct_change()
gdp.dropna(inplace=True)
gdp = gdp.reset_index(drop=True)

df_macro = gdp
```

DBnomics provides an API that is used to obtain the GDP data. This is done using the function that is defined at the beginning of the code which allows access to the specified API link and downloads the data. The API link is obtained from the website itself and can be in .csv or .xlsx or .json. The function specifies .json as that is the desired file type.

The function is called to retrieve the GDP data and it is stored in the GDP DataFrame. It was necessary to manually add in the GDP figures for 2022 and 2023 as these are not yet finalised and are based on estimates from DBnomics itself with a link provided to this chart from which the information was drawn in the references section below. The date in the DataFrame was converted to "year" and transformed to pertain to the period of prediction to match the other data. Any values before 2018 are dropped and any null values are dropped and the DataFrame is assigned to "df-macro" which will be merged with the final DataFrame after the calculation of RSI.

A.9 Calculation of RSI

```
df_val = pd.DataFrame()

for ticker in df_pivot_joined['Ticker'].unique():
    df_adjclose = yq.Ticker(ticker).history(start='2017-9-30', interval='1d')
    try:
        rsi = ta.RSI(df_adjclose['adjclose'])
    except:
        continue
```

```
    df_rsi = pd.DataFrame()
    df_rsi['RSI'] = rsi
    df_rsi.reset_index(inplace=True)
    df_rsi['date'] = pd.to_datetime(df_rsi['date'], utc=True)
    df_rsi['date'] = df_rsi['date'].dt.year + 1
    df_rsi = df_rsi.drop_duplicates(subset=['date'], keep='last', ignore_index=True, inplace=False)
    df_val = pd.concat([df_val, df_rsi], axis=0)

df_val = df_val.loc[df_val['date'] > 2018]

df_val.reset_index(drop=True, inplace=True)
df_val.rename(columns={'symbol': 'Ticker', 'date': 'Year'}, inplace=True)

df_macro_joined = df_pivot_joined.merge(df_macro, on='Year')
df_momentum_joined = df_macro_joined.merge(df_val, on = ['Ticker', 'Year'])
df_full = df_momentum_joined.merge(df_stock, on=['Ticker', 'Year'])

df_full.drop(["CIK", "Company", "Ticker", "Industry"], axis=1, inplace=True)
df_full = pd.get_dummies(df_full, drop_first=True)

df_full.dropna(inplace=True)
# Drop outliers which had undue influence over the model output
# Correspond to companies which went 20x or value fell by 95% over course of a year
df_full = df_full.loc[abs(df_full['Return'])<3]

Return = df_full.pop("Return")
df_full.drop(axis=1, labels=['Year'], inplace=True)
df_full.columns = [str(i) for i in df_full.columns]
```

The final variable calculated is the Relative Strength Index (RSI) which is done using the TAlib library. First define an empty DataFrame to store the values. The tickers are looped through to obtain the Adjusted Close prices for each stock for the specified time period and interval. The RSI is calculated with a 14-day trailing period.

Similar to GDP the RSI values are cleaned to obtain only the relevant values and are concatenated onto the empty DataFrame. With all the variables calculated, the DataFrames are merged onto the final DataFrame which contains all of the relevant input data for the ML Model. The "Returns" data is popped out as this is the outcome variable and can be treated separately from the predictors data.

A.10 Machine Learning Models - XGBoost and Random Forest

```
X_train, X_test, y_train, y_test = train_test_split(
    df_full, Return, test_size=0.3, random_state=20
)
model_xgb = xgb.XGBRegressor()

param_grid = {
    'learning_rate': [0.01, 0.05, 0.1, 0.5, 1],
    'max_depth': [3, 5, 10, 20],
    'n_estimators': [10, 50, 100, 250]
}

grid_search = GridSearchCV(estimator=model_xgb, param_grid=param_grid, scoring='explained_variance',
cv=10, n_jobs=-1)
grid_search.fit(X_train, y_train)
print(f"Best parameters: {grid_search.best_params_}")
print(f"Best score: {grid_search.best_score_:.3f}")
plt.style.use('seaborn-darkgrid')
plt.scatter(y_test, grid_search.predict(X_test), color='#0031da', alpha=0.6)
plt.xlabel("Actual Log Return", fontsize=12)
plt.ylabel("Predicted Log Return", fontsize=12)
```

```
plt.title("Plot of Actual vs. XGB-Predicted Log Returns of Test Observations", fontsize=12)
plt.show()
model_xgb = xgb.XGBRegressor(kwargs=grid_search.best_params_)

model_xgb.fit(X_train, y_train)
xgb.plot_importance(model_xgb, color='#0031da', alpha=0.6)
plt.show()
np.mean((y_test-grid_search.predict(X_test))**2)
RF = RandomForestRegressor()

paramgrid = {
             'n_estimators': [20, 100, 150, 250, 300, 500],
             'criterion': ['squared_error', 'absolute_error'],
             'max_features': ['sqrt', 'log2'],
             'max_depth': [3, 5, 10, 15, 25],
             }

G_CV = GridSearchCV(RF, param_grid=paramgrid, scoring='explained_variance', n_jobs=-1, cv=5)
G_CV.fit(X_train, y_train)
print(f"Best parameters: {G_CV.best_params_}")
print(f"Best Score: {G_CV.best_score_}")

np.mean((y_test-G_CV.predict(X_test))**2)

plt.style.use('seaborn-darkgrid')
plt.scatter(y_test, G_CV.predict(X_test), color='#00da3c', alpha=0.6)
plt.xlabel("Actual Log Return", fontsize=12)
plt.ylabel("Predicted Log Return", fontsize=12)
plt.title("Plot of Actual vs. RF-Predicted Log Returns of Test Observations", fontsize=12)
plt.show()

RF_model = RandomForestRegressor(kwargs=G_CV.best_params_)
RF_model.fit(X_train, y_train)

features = X_train.columns

importances = RF_model.feature_importances_

df_importance = pd.concat([pd.Series(features, name='Feature'), pd.Series(importances, name='Importance')]
df_importance.set_index('Feature', inplace=True)
df_importance.sort_values(by='Importance', ascending=True).plot(kind='barh', color='#00da3c')
plt.show()
```

The train / test data split is done on the input data and returns data as X and Y. The Boosting model undergoes grid-search cross-validation first returning the best score for performance comparison and the best parameters for the fitted model. A scatter plot is created to examine the predicted vs. actual returns as shown in the report. The Boosting model is then fit onto the data using the best hyperparameters (as defined in the parameter grid) returned from the grid-search.

The exact same process is carried out for Random Forest. This allows for comparison between the two models both by quantifying prediction accuracy and examining visual representation of feature importance and differences between actual and predicted returns.

A.11 References

- SigmaCoding https://www.kaggle.com/code/klcode/scraping-and-analysing-sec-filings-part-1/notebook#EDGAR-Database
- SigmaCoding https://www.kaggle.com/code/klcode/scraping-and-analysing-sec-filings-part-2/notebook

- World Economic Forum https://www.weforum.org/agenda/2017/05/2017-is-the-year-of-artificial-intelligence-here-s-why/
- Forbes https://www.forbes.com/sites/louiscolumbus/2019/08/15/why-ai-is-the-future-of-financial-services/?sh=5b52e5e03847
- SEC Edgar https://www.sec.gov/edgar/searchedgar/companysearch
- SEC https://www.sec.gov/news/speech/speech-bauguess-050318
- SKLearn https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html
- XGBoost https://xgboost.readthedocs.io/en/stable/python/python_api.html