Data Warehousing and Business Intelligence Project

on

Predicting Daily Returns in the S&P 500 index using Twitter and Google Trends

Billy Hanan
18179797

PGDip in Data Analytics – 2019

Submitted to: Dr. Pierpaolo Dondio

National College of Ireland

Project Submission Sheet – 2018/2019

Student Name:	Billy Hanan
Student ID:	18179797
Programme:	Postgraduate Diploma in Science in Data Analytics **Year:** 2019
Module:	Data Warehousing and Business Intelligence
Lecturer:	Dr. Pierpaolo Dondio
Submission Due Date:	12 May 2019
Project Title:	Predicting Daily Returns in the S&P 500 Index using Twitter and Google Trends
Word Count:	

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the references section. Students are encouraged to use the Harvard Referencing Standard supplied by the Library. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action. Students may be required to undergo a viva (oral examination) if there is suspicion about the validity of their submitted work.

Signature: ……

Date: ……

PLEASE READ THE FOLLOWING INSTRUCTIONS:

1. Please attach a completed copy of this sheet to each project (including multiple copies).
2. Projects should be submitted to your Programme Coordinator.
3. **You must ensure that you retain a HARD COPY of ALL projects**, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer. Please do not bind projects or place in covers unless specifically requested.
4. You must ensure that all projects are submitted to your Programme Coordinator on or before the required submission date. **Late submissions will incur penalties.**
5. All projects must be submitted and passed in order to successfully complete the year. **Any project/assignment not submitted will be marked as a fail.**

Office Use Only

Signature:	
Date:	
Penalty Applied (if applicable):	

Table 1: Mark sheet – do not edit

Criteria	Mark Awarded	Comment(s)
Objectives	of 5	
Related Work	of 10	
Data	of 25	
ETL	of 20	
Application	of 30	
Video	of 10	
Presentation	of 10	
Total	of 100	

Project Check List

This section capture the core requirements that the project entails represented as a check list for convenience.

- ☒ Used LaTeX template
- ☐ Three Business Requirements listed in introduction
- ☐ At least one structured data source
- ☐ At least one unstructured data source
- ☐ At least three sources of data
- ☐ Described all sources of data
- ☐ All sources of data are less than one year old, i.e. released after 17/09/2017
- ☐ Inserted and discussed star schema
- ☐ Completed logical data map
- ☐ Discussed the high level ETL strategy
- ☐ Provided 3 BI queries
- ☐ Detailed the sources of data used in each query
- ☐ Discussed the implications of results in each query
- ☐ Reviewed at least 5-10 appropriate papers on topic of your DWBI project

Predicting Daily Returns in the S&P 500 index using Twitter and Google Trends

Billy Hanan
18179797

May 11, 2019

Abstract

We investigate the predictive power of Google Trends and Twitter data on the daily returns of the S&P 500 index. For this purpose, a data warehouse was constructed within a Mysql database. Three sources of data were collected and stored in files. The three data sets consisted of historical S&P 500 time series tick data, google trends search volumes on the keyword "S&P 500" and sentiment scores of relevant tweets. The ETL process was implemented using bespoke Python code and a rudimentary analysis on the integrated data set was then performed. We discover that there is little predictive power in google search volume and the sentiment of relevant Twitter data on the directionality of daily S&P 500 movements.

1. Introduction

The price movement of markets reflect the collective decisions of a myriad of market participants. With the advent of the internet and social media age, we now have access to information relating to the motivations and opinions of these participants. As such, there has been much academic (and non-academic) work on the predictive power of google searches and Twitter data on stock market movements [1-8]. The conclusions on this topic can best be described as mixed. This project constitutes my own brief investigations into this area of research.

In this project, among the many tradable stocks and instruments in the financial world, I decided to focus solely on the S&P 500 index. This American stock market index is a weighted average of the value of 500 large companies listed on the NYSE, NASDAQ and Cboe BZX exchange. It is thus considered an excellent proxy for the overall value of the U.S. stock market and is thus arguably the most important stock market index in the world.

In this project, we monitor the public interest and sentiment toward the U.S. market via relevant google searches and tweets. Google provides a website [9] that allows one to view the relative volume of any keyword entered into the Google search engine over time. In this project, we are interested in searches for the keyword "S&P 500 index". The data collected thus represents a proxy for the public interest in the U.S. stock market over time.

Tweets containing the term “S&P 500” were also collected and sentiment scores for each were calculated. The average daily sentiment toward the S&P 500 was subsequently calculated. Again, this represents a source of data relating to the public opinion toward the U.S. markets.

Can these two sources of data help one predict the future movements in the S&P 500 index? For ease of analysis, the above data needed to be collected, integrated and stored in an easily accessible location. To this end, a Data Warehouse (DW) with the following requirements needed to be constructed:

1. The DW should facilitate BI queries on daily S&P 500 index data over time.

2. Daily volumes of relevant google searches should be stored.

3. Daily sentiment scores of relevant Tweets are to be recorded.

In the project specification, it states that students are “free to use any data warehousing tool set discussed in the course, or otherwise”. I initially attempted to use Microsoft’s SSIS, but after having some difficulty getting it operational on my MacBook, I decided to construct the DW in a Mysql database and implement the entire ETL process using Python.

2. Data Sources

A general description of the individual data sources used in this project have already been given in the introduction. Here we give specific details on each source used.

2.1 Yahoo! Finance S&P 500 historical price data

A historical time series of the S&P 500 index was required and is available from many sources. The data set used in this project was acquired from the Yahoo! Finance website [10] where it is updated on a daily basis. From here, one can easily download daily data into a csv file. The data downloaded covered the time period from 2004-01-01 to 2019-01-01. Here is a sample of the acquired data:

Date	Open	High	Low	Close	Adj Close	Volume
2004-01-02	1111.920044	1118.849976	1105.079956	1108.47998	1108.47998	1153200000
2004-01-05	1108.47998	1122.219971	1108.47998	1122.219971	1122.219971	1578200000
2004-01-06	1122.219971	1124.459961	1118.439941	1123.670044	1123.670044	1494500000
2004-01-07	1123.670044	1126.329956	1116.449951	1126.329956	1126.329956	1704900000

2.2 Google Trends Search Volume Index (SVI)

Google provides a website where one can access time series data on the search volume index (SVI) of any search term that has been entered into the google search engine. The SVI values are always scaled between 0 and 100 and represent a measure of the relative volume of

searches over the specified time frame. One can therefore identify days where the search volume was particularly heavy or light.

One can download this data as a csv file directly from the website. However there are constraints. Specifically, it will only allow you to retrieve daily data over a time period no greater than about 8 months. Beyond that, only weekly and monthly SVI values are provided.

As daily data is a clearly defined requirement for our DW, one could conceivably download SVI data in 8 month time windows spanning from 2004 to 2019. This is a laborious manual task and one is then still left with the problem of having to combine the data files together and rescale them so that the individual time series align with each other (a non-trivial process).

It was therefore decided to automate this entire process by using a Python module called PyTrends. This unofficial API allows you to programmatically acquire data from the Google Trends website. The python script *Collect_SP500_Google_Trends_Data.py* was written to:

1. Acquire daily SVI data for the search term "S&P 500 index" for every month from 2004-01-01 to 2019-01-01

2. Integrate these individual monthly segments together into a single time series

Please see appendix A.1 for the python code. Note that the algorithm used to integrate the monthly data segments was obtained from a 2018 blogpost by Franz B. [11]. Also note that Python 3.0 is required to successfully execute the script.

Here are details of the inputs into Google Trends that were applied when retrieving the data:

i. The topic *"S&P 500 Index (Market Index)"* was used as the search keyword rather than simply the string "S&P 500". The use of a Google Trends topic is superior as it leverages the context-aware algorithm implemented by Google.

ii. The search was restricted to the *Finance* category.

iii. The region chosen was *Worldwide.*

iv. *Web Search* was the applied google search domain

Here is a sample of the Google Trends data acquired:

date	SVI	isPartial
2004-01-01	0	FALSE
2004-01-02	40.42	FALSE
2004-01-03	13.76	FALSE
2004-01-04	39.56	FALSE
2004-01-05	38.7	FALSE

Note that the *isPartial* column above simply identifies if the SVI value quoted was an estimate.

2.3 Twitter Sentiment Data

Twitter has become the de facto platform for finance people to get real-time news and opinions on the markets [4]. Famously in 2013, stock prices momentarily plunged after a Twitter account belonging to the Associated Press was hacked and used to send a bogus report claiming the White House had been bombed and President Obama injured.
It was for this reason that it was stated in the business requirements that the DW needed to store daily sentiment data on S&P 500 related tweets. This data was collected and processed as follows:

1. Using the python script *TwitterScrape.py*, tweets containing the search term 'S%26P500' were collected for every year from 2006 to 2018.

2. Each annual Json tweet file obtained above was then fed into the python script *append_tweet_sentiment.py*. This script reads the tweet text and calculates a sentiment score for each tweet. An output Json file holding the original tweet data with the sentiment score appended is then generated.

The code for both scripts is shown in Appendix A.2 and A.3.

The sentiment score for each tweet was calculated using the generic sentiment Python analysis tool *TextBlob* which yields a score from -1 (negative) to +1 (positive). As can be seen in the Python script in appendix A.3, some trivial cleaning of tweets was performed before calculating the sentiment score of each sentence in a tweet. The average of these scores was then taken to supply the final sentiment score for the entire tweet.

The final processed tweet data with sentiment score is stored in a Json file. Here is sample of a typical entry for a tweet:

Key	Value
sentiment	0.2166666667
text	"Stock futures strong this morning after a strong end of day move yesterday. SP500 +6.25 NAS100 +11.25 Dow +72.00"
likes	0
replies	0
retweets	0
user	"MacMoov"
fullname	"Mark Anderson"
url	"/MacMoov/status/34027072"
html	"<p class=\"TweetTextSize js-tweet-text tweet-text\" data-aria-label-part=\"0\" lang=\"en\">Stock futures strong this morning after a strong end of day move yesterday. SP500 +6.25 NAS100 +11.25 Dow +72.00</p>"
id	34027072
timestamp	1177069527000

In summary then, the three data sources to be collected during the ETL process consists of three sets of files: A single csv file holding S&P 500 historical values, another csv file holding google trends SVI values and a set of Json files holding sentiment scores for S&P 500 related tweets.

3. Related Work

In a 2013 paper entitled "Quantifying Trading Behaviour in Financial Markets Using Google Trends" [1], the authors analysed the google trends data of various search terms related to finance and showed that one could utilise this information to construct a trading strategy that could be used to beat market returns. Specifically, they used weekly google trends data for the search term "*debt*" to determine when to buy and sell the Dow Jones Industrial Average (DJIA) index. They showed how their "google trends strategy" would have, in theory, given profits of 326% from 2004-2011(as opposed to a mere 16% return using a simple buy & hold strategy of the DJIA).

This is just one example of a myriad of papers claiming how internet data could be leveraged to produce out-sized returns. Though there are surprisingly few research papers investigating the correlation between google searches and market movements, there are a copious amount of papers investigating the relationship between Twitter data and markets. Here I give the briefest of reviews of some of the relevant work.

In one of the seminal papers in this area [2], Bollen et al. extracted various measures of the public mood from general tweets. Using these mood factors and the previous three days values of the DJIA as inputs into a neural network model, they were able to predict the direction (positive or negative) of daily returns in the DJIA with an accuracy of 87.6% (It should be noted though that the period over which they tested only spanned three weeks).

In work by Pagolu et al. [3], they similarly achieved an impressive prediction accuracy of 69.01% in predicting the daily direction of returns for Microsoft stock ($MSFT). In this study, the sentiment of Microsoft related tweets was calculated and categorised as either being positive, negative or neutral. The number in each category over the preceding three days were then fed into a simple Logistic regression model to predict the direction of returns for the following day.

The above strategy of categorising relevant tweets into two or three sentiment categories and using their counts as inputs to some model is prevalent in the literature. In work by Tahir et al. for example [4], a simple linear regression model was used to predict the daily close value of the FTSE 100 index. They ultimately claimed that a strong positive correlation exists.

Similarly, Rao et al. [5] looked at the DJIA and NASDAQ-100 indices along with some individual tech stocks and, after collecting and analysing over 4,000,000 tweets, concluded that their results "show high correlation (up to 0.88 for returns) between stock prices and twitter sentiments".

In another study by Mao et al. [6], the sentiment of tweets was not determined. Instead they performed a volume-based (as opposed to sentiment-based) analysis where they simply counted the daily number of tweets that mentioned S&P 500 stocks. They again found that the number of tweets is correlated to daily returns in the S&P 500 index. "We find that whether S&P 500 closing price will go up or down can be predicted more accurately when including Twitter data in the model".

All of the above studies quoted above are optimistic on the possibility that Twitter data can be leveraged to achieve enhanced returns. This runs contrary to the accepted wisdom in the investment industry that it is in fact difficult to achieve alpha. That is, it is not easy to beat the market.

And there are indeed a small number of studies that come to a less optimistic conclusion. In work by Umang Patel [7] where he uses the public mood surmised from twitter data to predict stock market movements in the NASDAQ, he concludes that the "twitter sentiment score can predict the movement of stocks if the sentiment is trending positive, not negative."

Ting et al. came to a more negative conclusion [8]. They collected 7 months of tweets that mentioned S&P 500 companies and defined a measure of market "bullishness". Namely, they defined

$$B(t) = ln\frac{1 + \mathrm{N_p}}{1 + N_n}$$

where $\mathrm{N_p}$ and $\mathrm{N_n}$ is the number of positive and negative tweets at time t. They concluded that there "is not much connection between bullishness and returns … On the other hand, previous day returns have a positive and significant effect on bullishness. Thus, returns affect bullishness in stock micro-blogs, but not vice versa".

In this project we focus on a market index (S&P 500) as in some of the studies above. However we examine the market data over a much longer time period (14 years) than those above. Surprisingly, the studies invariably look at periods not much larger than one year. This seems a significant flaw because markets are known to behave very differently over time.

Furthermore, I was unable to find any papers that look at both google trends and twitter sentiment in their analysis (as is done in the present study). As mentioned previously, it appears that the research area is dominated with the study of Twitter sentiment rather than google trends.

Regardless, I undertook this project primarily to see if I could replicate the optimistic forecasts that publicly available internet data could indeed be leveraged to achieve excess returns in the markets.

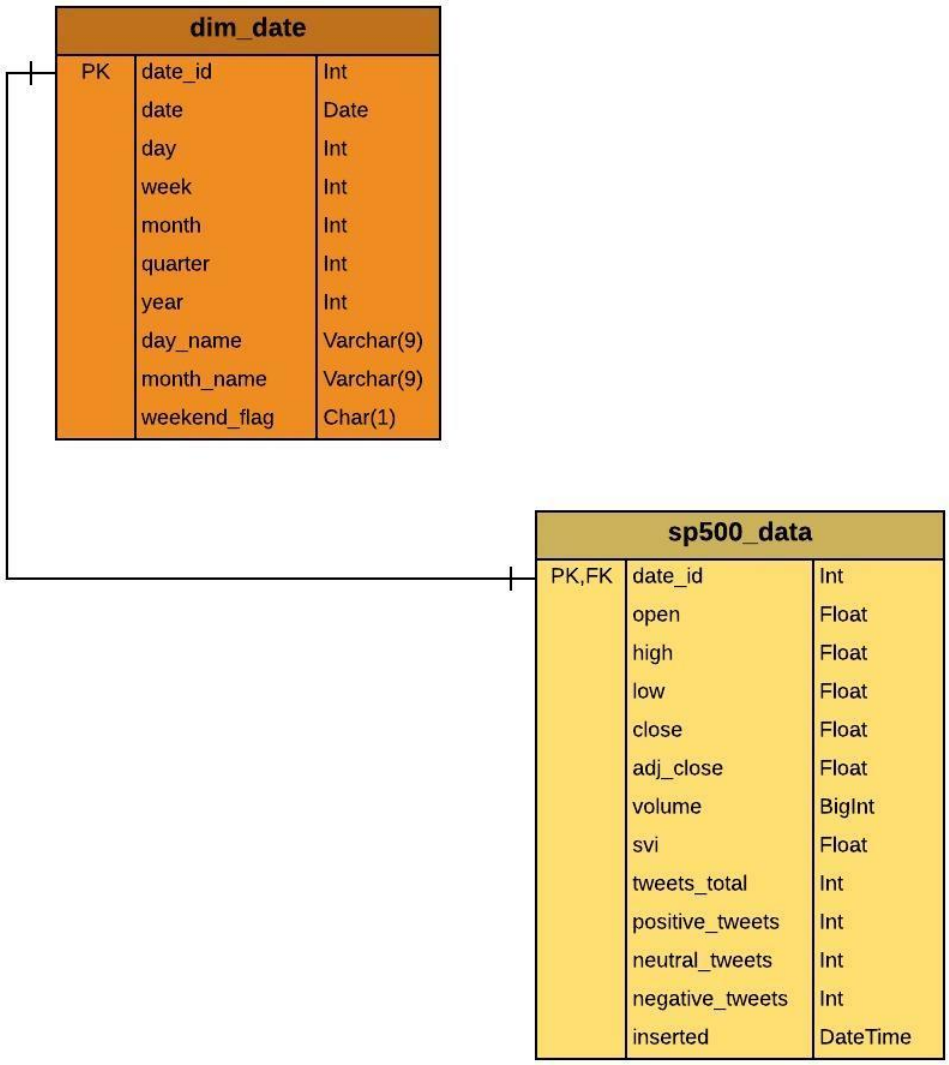
4. Data Model

The data warehouse design is actually incredibly simple as there is only a single dimension table. The database schema is displayed on the next page.

As we are only required to store and query daily data, each row in the dimension table *dim_date* corresponds to a specific date. Attributes relating to the date are also stored in the row. Below I show some sample values taken from the dimension table.

Field Name	Value
date_id	20130908
date	2013-09-08
day	8
week	36
month	9
quarter	3
year	2013
day_name	Sunday
month_name	September
weekend_flag	t

Though not essential and not stated in the original requirements, the above attributes will allow the user to execute queries on different granularities other than daily e.g. You can calculate the total number of tweets broken down by week, month, quarter or year. The dimension table will also allow you to retrieve the value of the S&P 500 index at close every Tuesday. In short, it allows the user to have some flexibility when querying.



The table *sp500_data* holds the required facts needed to meet the business requirements. The values are derived from the three data sources. The fields *open*, *high*, *low*, *close*, *adj_close* and *volume* hold tick data on the S&P 500 index for a specific date. Similarly, the *svi* field stores the google search volume index value and the tweet related fields holds tweet counts. Note that I have also added an *inserted* field into the fact table. This holds the date and time when the record was last inserted/updated. This was done to provide some minor transparency and auditing functionality.

5. Logical Data Map

The table below gives the details of each field in the source data and how it is transformed and loaded into the fact table (It should be noted that the dimension table is entirely independent of the source data. I will expand on this point in the next section). Each data source is given a different colour in the logical data map for clarity.

Source	Field Name	Fact Table Column	Transformation
S&P 500 Tick Data	Date	date_id	Surrogate key (lookup from dim_date)
	Open	open	None
	High	high	None
	Low	low	None
	Close	close	None
	Adj Close	adj_close	None
	Volume	volume	None
Google Trends Data	date	date_id	Surrogate key (lookup from dim_date)
	SVI	svi	None
	isPartial	N/A	
Twitter Data	sentiment	tweets_total positive_tweets neutral_tweets negative_tweets	Daily count of records in each sentiment category
	text	N/A	
	likes	N/A	
	replies	N/A	
	retweets	N/A	
	user	N/A	
	fullname	N/A	
	url	N/A	
	html	N/A	
	id	N/A	
	timestamp	date_id	Surrogate key (lookup from dim_date)

Note that N/A in the fact table column implies that the source data field is unused and discarded.

As you can see, the transformations are relatively simple. More details of these transformations will be given in the following section.

6. ETL Process

As will be demonstrated in the video presentation, the Data Warehouse (DW) can be created, cleaned down and populated by executing the following Python scripts respectively:

1. ***create_data_warehouse.py***
2. ***clean_data_warehouse.py***
3. ***etl_sp500.py***

The data warehouse and its tables were created in a MySQL database. Some key Python modules used include:

Module	Purpose
MySQLdb	Interacting with Mysql DB
pandas	Storing and transforming source data
textblob	Sentiment analysis of Twitter data

I will first give a brief description of the first two Python scripts above before giving a more detailed description of the ETL script. The code for all Python scripts is shown in appendix B.

6.1 create_data_warehouse.py

This executes the required sql scripts via Python that actually create the database imaginatively called *Data_Warehouse*. The tables *dim_date* and *sp500_data* are also created. It is important to note that the *dim_date* table is populated at this stage since it has no dependency on the source data.

This insertion of data into the dimension table was performed using a stored procedure which effectively takes in a start and end date and then populates the table with every date over this specified range (Note that the stored procedure and all other sql scripts used can be seen in appendix C). The primary key in *dim_date* is *date_id* and this critically will act as a surrogate key when loading data into the fact table.

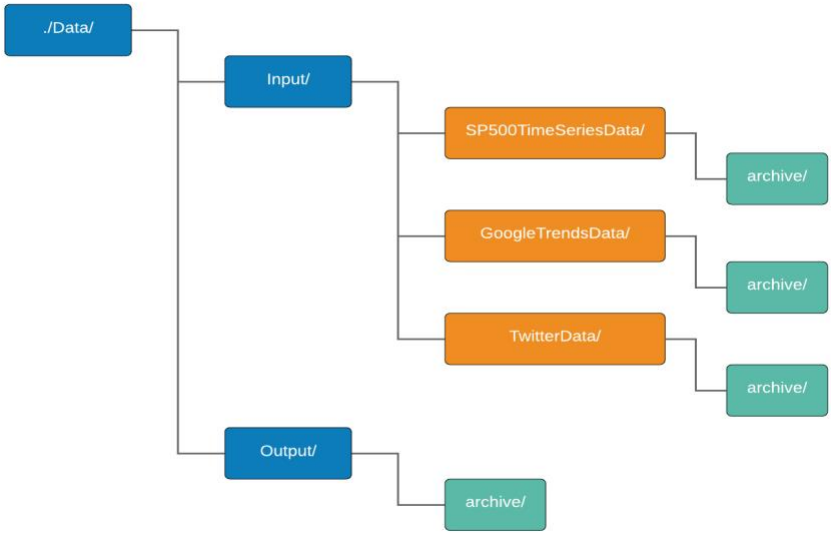
6.2 clean_data_warehouse.py

This script simply deletes all the contents in the fact table sp500_data by executing a truncate command.

6.3 etl_sp500.py

I will now give a high level description of the ETL process that is executed by this Python script. All the required functionality needed for extraction, transformation and loading were created in separate classes in the Python files extract.py, transform.py and load.py (See appendix D).

The diagram below shows the directory structure required by the ETL process. Note that the source data files are presumed to exist in their respective directories (colour-coded in orange) before ETL is executed.



On execution, these data files are read in and their data transformed. This transformed data is then written to the Output directory shown in the schematic above from where it is subsequently loaded to the data warehouse. Let us now describe each stage of this ETL process in more detail.

6.3.1 Extraction

As previously mentioned, the source data files are expected to exist in their respective input directories before the ETL process is executed (If they do not, a warning message will be written to the console).

The extraction is designed to automatically read in multiple files in an input directory. The only requirement is that the file names end with the appropriate extension (*.csv in the case of the S&P 500 tick and google trends data and *.json for the twitter data source files).

Note that after being read into memory, the source data files are copied to a time-stamped directory under the appropriate archive/ sub-directory. This feature was added so that one could easily track the history of uploads to the data warehouse.

At this point all the source file data is stored in pandas data frames waiting to be processed and transformed.

6.3.2 Transformation

The ultimate goal here is to take the source data stored in the pandas data frames and use it to create a new data file that is in the appropriate format to be loaded to the fact table. In most cases, the fields in the source data were either filtered out (i.e. Unused) or simply kept in their original form (i.e. No processing required).

There are two notable exceptions however. Each of the three data sources do have a date-related field. These needed to be replaced with the appropriate surrogate key stored in *dim_date*. This was done by simply reading the information in *dim_date* into a data frame and joining it to the other input data frames. The appropriate surrogate key was then matched up to each input record. Note that the integration of the three data sources into a single fact table record was very simple as a basic join on the surrogate keys of all three sources was all that was required.

The other non-trivial transformation required was the calculation of tweet counts. Recall that in the fact table we have the following columns: *tweets_total*, *positive_tweets*, *neutral_tweets* and *negative_tweets*. These are the number of tweets that fall into each sentiment category on any particular day.

Each tweet in the input source files has a sentiment score attached (between -1 to +1). The category a tweet falls into depends on the cut-off values *positive_threshold* and *negative_threshold* that are assigned in the python Twitter class (Again these values must be between -1 to +1). Here is the simple logic used to determine which category a tweet should be counted in:

Sentiment Category	Condition
Positive	sentiment_score > positive_threshold
Neutral	negative_threshold ≤ sentiment_score ≤ positive_threshold
Negative	sentiment_score < negative_threshold

The above logic is applied in the Python code to calculate the tweet counts that are assigned to *positive_tweets*, *neutral_tweets* and *negative_tweets* on any particular day. Note that *tweets_total* is a simple derived field (being simply the sum of the sentiment category counts).

The source data was fortunately of high quality and I therefore found little need to perform any cleaning operations. (Recall though that there was some trivial cleaning of tweet texts performed when the sentiment scores were being calculated during the data collection phase).

At the end of the above transformation process, the output data has been generated in a data frame and the final trivial step is to simply write this data to a csv file. This file, named *fact_table_data.csv*, is written to the ./Data/Output directory where it is ready for loading to the fact table.

6.3.3 Loading

Once the file *fact_table_data.csv* is created, it is then automatically loaded to the table *sp500_data*. Again, for auditability purposes, this file is moved to a time-stamped directory under ./Data/Output/archive.

There is the possibility that upon loading, you may overwrite some of the existing data in the fact table. An additional feature was therefore implemented to cater for such a scenario. Before loading begins, a check is done to see if such overwrites will occur. The loading will proceed regardless but any records that are about to be overwritten will be backed up to a file called *replaced_records.csv* and a warning message output to the console. This file is

archived to the same time-stamped directory as *fact_table_data.csv*. One can therefore restore the records to their original state if need be using this file.

Finally, upon loading, some simple statistics are printed to the console. These include the number of records inserted into the fact table and the number of existing records that have been overwritten.

7. Application

The purpose of the data warehouse was primarily to collect and integrate the different S&P 500 related data sources together for ease of analysis. The actual dimension table was of secondary importance. As such, rather than providing the BI queries asked for in the project spec, we instead attempt to build a rudimentary model that can be used to predict the direction of returns in the S&P 500 index. More specifically, the objective was to build a model that, using data from the present or past, could predict the direction of returns the following day.

This was done within a Jupyter notebook using the Python machine learning module *scikit-learn*. In the notebook environment, one is able to connect to the data warehouse and extract the data held in both the dimension and fact tables. These can then be easily combined and stored in a single pandas data frame that holds all the DW data in memory.

A target variable *y* was appended to the records in this data frame. This took on a value of 0 if the return the following trading day was negative and 1 if not. The prospective model ingests specified input features and outputs either a 0 or 1 as its prediction for the following day. Candidate features for the model include *close*, *volume*, *svi*, *tweets_total* or any other field (or derived field) from the fact table.

A random forest model was trained on a subset of the available data. This standard ensemble technique involves the training of a finite number of decision trees. For the results presented here, n=100 trees were used in the model. To avoid over-fitting and to obtain an accurate estimate of the predictive power of the model, we calculate the fraction of correct predictions, known as the classification accuracy, on a subset of the data that was not used in training. Specifically, we calculate the accuracy by performing k=10-fold cross validation.

We began by training a variety of single feature models and the results are summarised in the table below:

Input Feature	Accuracy
negative_tweets	0.51453
svi	0.51353
tweets_total	0.51216
neutral_tweets	0.50168
*return_today**	0.50118
positive_tweets	0.49762
*bullishness**	0.48859

It should be noted that the features marked with an asterisk are derived features. For example, the feature *bullishness* is calculated using the definition used by Ting et al. [8], namely

$$B(t) = ln\frac{1 + positive\_tweets}{1 + negative\_tweets}$$

It gave the poorest prediction accuracy of all the features above. Likewise the derived feature *return_today* defined as

$$return\_today(t) = \frac{close(\mathrm{t}) - close(t-1)}{close(t-1)}$$

also produced poor results. Overall though, the results are very disappointing.

For example, our data set spans 3775 trading days of which 2057 have positive (or zero) returns. Therefore, if one simply applied the dumb model of assuming every return tomorrow would be positive (or zero), one would achieve an accuracy of approximately 0.544 which beats the prediction accuracy achieved in any of the above models.

More complicated random forest models with multiple input features were also trained. But none achieved accuracies greater than the value of 0.51453 obtained from the single feature *negative_tweets* model.

Some improvement on the model accuracy was obtained when the input features were discretized. That is, we set the input feature to 1 if it's value is positive or zero, and 0 otherwise. The best performing random forest model obtained an accuracy of 0.55863 where the input features in the model were:

X = (Z(*Return_today(t)*), Z(*Return_today(t-1)*),
Z(*Return_today(t-2)*), Z(*svi_pct_change*), Z(*bullishness*))

Note that the function Z(x) discretizes the feature value (It is effectively the step function). One can see that we enter the discretized values of today's and the previous 2 days returns into the model. The derived variable *svi_pct_change* is defined as

$$svi\_pct\_change(t) = \frac{svi(\mathrm{t}) - svi(t-1)}{svi(t-1)}$$

and essentially represents the percentage change in the *svi* value from one day to the next.

8. Discussion and Conclusion

In previous studies, models based on Twitter data were built with prediction accuracies of 87.6% and 69.01% for the directional movement in the DJIA index and Microsoft stock respectively [2,3]. The accuracy of the models built in this project are distinctly underwhelming in comparison and struggle to beat the simple baseline model with accuracy of 54.4%.

Using a random forest model with a single input feature, accuracies well below this baseline value were obtained. The google trend feature *svi* and twitter features *negative_tweets* and *tweets_total* all provided accuracies exceeding 51%.
One might think that combining their weak predictive power would yield superior results. However, training a random forest model having these three features actually caused the model accuracy to degrade (A fact that was not mentioned in the previous section).

In an attempt to improve the model performance, the input feature values were discretized and accuracies of almost 56% were achieved. However, I do not have much faith in this model. For example, removing the feature *return_today(t-2)* so that there are now only 4 input features to the model yielded an accuracy of 52.1%. The sudden drop from approximately 56% to 52% is not encouraging for the robustness of the model. It also suggests that historical return data may have more predictive power than any twitter or google trend related data.

However, due to limited time, only a rudimentary analysis of the S&P 500 data is presented here in this project. One could for example try training alternative classifier models other than the random forest model analysed here e.g. Logistic regression, support vector machines, k-nearest neighbour.

The daily twitter and google trend data are also particularly noisy. Taking a moving average of these time series data could provide improved input features for a model.

We also need not restrict ourselves to simply predicting the direction of future returns. Ideally, one would also like to predict the magnitude of the returns. Furthermore, in this study we only examine daily returns. There is no reason why one would not want to examine returns on longer time scales e.g. Weekly and monthly returns.

Another important point to make is that the *TextBlob* sentiment algorithm used here is rather generic. In much of the academic work using Twitter data, much more advanced algorithms are used (and in some cases, bespoke sentiment generating engines are trained). One might also weight the tweet counts so that particularly influential twitter users have their tweets weighted more heavily [8]. An examination of the effectiveness of the sentiment scoring was also not performed.

In conclusion, the evidence presented in this project would suggest that data obtained from Twitter and Google Trends has little predictive power in determining the future movements of the S&P 500 index. However, much more analysis would need to be performed on the data before coming to a definitive conclusion.

Finally, a brief video presentation on this project can be found at the following URL:
https://www.loom.com/share/bbbb8c17b6884aec9a2fc12245f726ab

[1] T. Preis, H.S. Moat, H.E. Stanley, "Quantifying Trading Behaviour in Financial Markets Using Google Trends," *Nature, Scientific Reports* **3**, article number: 1684 (2013)

[2] J. Bollen, H. Mao, X. Zeng, "Twitter mood predicts the stock market," *Journal of Computational Science* **2** pp. 1-8 (2011)

[3] V. Pagolu, K.N. Reddy, G. Panda, B. Majhi, "Sentiment analysis of Twitter data for predicting stock market movements," 2016 International Conference on Signal Processing, Communication, Power and Embedded Systems.

[4] T.M. Nisar, M. Yeung, "Twitter as a tool for forecasting stock market movements: A short-window event study," *Journal of Finance and Data Science* **4**, pp. 101-119 (2018)

[5] T. Roa, S. Srivastava, "Analyzing Stock Market Movements Using Twitter Sentiment Analysis," Proceedings of the 2012 International Conference on Advances in Social Networks and Mining (ASONAM 2012).

[6] Y. Mao, B. Wang, W. Wei, B. Liu, "Correlating S&P 500 stocks with Twitter data," Proceedings of the First ACM International Workshop on Hot Topics in Interdisciplinary Social Networks Research 2012.

[7] U. Patel, "Twitter data Predicting Stock Price Using Data Mining Techniques", ResearchGate, Technical Report April 2016.

[8] Ting Li, Jan van Dalen, Pieter Jan van Rees, "More than just noise? Examining the information content of stock microblogs on financial markets," *Journal of Information Technology* **33**, pp. 50-69 (2018).

[9] https://trends.google.com

[10] https://finance.yahoo.com/quote/%5EGSPC/history/

[11] https://medium.com/@bewerunge.franz/google-trends-how-to-acquire-daily-data-for-broad-time-frames-b6c6dfe200e6

APPENDIX A.1

Collect_SP500_GoogleTrends_Data.py

```
#
# Execute this script to collect daily Google Trends data for the search term "S&P 500" from 2004 -> 2019
#
# Note that Google Trends will only supply you with daily Search Volume Indices (SVI) for about an 8 month time window
# In this script, we therefore individually query all the months from 2004 to 2019 to obtain daily data
# We then combine all this data into a single time series via a suitable rescaling (based on monthly SVIs)
#
# The details of the rescaling is described in the following blog-post
#
# https://medium.com/@bewerunge.franz/google-trends-how-to-acquire-daily-data-for-broad-time-frames-b6c6dfe200e6
#

import pandas as pd
from datetime import datetime, timedelta
from pytrends.request import TrendReq
from calendar import monthrange


# Timezone offset from UTC
off_set = '0'

# Login to Google
pytrend = TrendReq(tz=off_set)

# We will perform an advanced search for "S&P 500" using Google Trends Topics
topics = pytrend.suggestions('S&P 500')
searchItem = topics[0]
print("Search Term: " + searchItem['title'] + " ("+searchItem['type']+")")
keywords = [searchItem['mid']]

# Chosen category = 7 (Finance)
category = 7

# Worldwide
region = ''

# Web Search
domain = ''


#
# Execute a single search from 2004 -> 2018 (This returns monthly SVI values)
#

start_date = '2004-01-01'
end_date = '2018-12-31'
searchDates = str(start_date) + " " + str(end_date)
print("Searching: " + searchDates)

# Set search details
pytrend.build_payload(kw_list=keywords, cat=category, geo=region, timeframe=searchDates, gprop=domain)

# Retrieve Interest Over Time Information
df_monthly_svi = pytrend.interest_over_time()

# Change column names
df_monthly_svi.columns = ['interest','isPartial']
```

```
#
# Now execute a series of monthly requests from 2004 -> 2019
#

# first construct monthly query timeframes

monthlySearchDates = []

for year in range(2004,2019):

        for month in range(1,13):

                numDaysInMonth = monthrange(year, month)[1]
                start_date = str(year) + "-" + str(month).zfill(2) + "-01"
                end_date = str(year) + "-" + str(month).zfill(2) + "-" + str(numDaysInMonth)
                searchDates = str(start_date) + " " + str(end_date)
                monthlySearchDates.append(searchDates)


# We now send in a request for each month

df_daily_svi = pd.DataFrame()

for j in range(len(monthlySearchDates)):

        # Retrieve Search Dates
        searchDates = monthlySearchDates[j]

        # Retrieve daily google trend data for month
        print("Searching: " + searchDates)

        # Set search details
        pytrend.build_payload(kw_list=keywords, cat=category, geo=region, timeframe=searchDates, gprop=domain)

        # Retrieve Interest Over Time Information
        interest_over_time_df = pytrend.interest_over_time()

        # Rename columns
        interest_over_time_df.columns = ['interest','isPartial']

        # Retrieve SVI value for month
        svi_for_month = df_monthly_svi.iloc[j]['interest']

        # Rescale SVI values using monthly SVI
        interest_over_time_df['interest'] = interest_over_time_df['interest']*(svi_for_month/100.0)

        # Append monthly SVIs to our main data frame
        df_daily_svi = pd.concat([df_daily_svi, interest_over_time_df])
-


# Write results to file
df_daily_svi.to_csv(r'./sp500_google_trends.csv')
```

APPENDIX A.2

TwitterScrape.py

```
#
# I could not get the query_tweets() to work
# when I imported TextBlob
#
# So I have split them out into TWO python files
```

```
#
#               TwitterScrape.py - creates JSON file holding tweets & tweet info
#               append_tweet_sentiment.py - reads in this JSON file, generates and appends sentiment score for each tweet and writes output
to a JSON file
#
# PARAMETERS for Twitter Scraper
#
# limit: Max number of tweets retrieved - Multiple of 20 (Set to None to disable)
# limit: Max number of tweets retrieved - Multiple of 20 (Set to None to disable)
# poolside: Num of parallel threads (I think it splits the time frame up into segments and assigned to a thread)
#


#
#               Import modules
#

import json
import datetime
from twitterscraper import query_tweets


#
#               INPUTS
#

start_date = datetime.date(2013, 1, 1)
end_date = datetime.date(2013, 12, 31)
limit_size = None
numThreads = 100
query = 'S%26P500'
outputFileName = "tweets2013.json"


# Do you want the tweets printed to screen?
printTweets = False

#
#               Returns Tweet instances with attributes
#               ['likes', 'url', 'timestamp', 'replies', 'html', 'user', 'text', 'fullname', 'id', 'retweets']
#

list_of_tweets = query_tweets(query, limit=limit_size, begindate=start_date, enddate=end_date, poolsize=numThreads, lang='en')

print("\nNumber of Tweets found: " + str(len(list_of_tweets)))

#
#               Write to JSON file
#

jsonfile = open(outputFileName,"w")

list_of_json = []

for tweet in list_of_tweets: # for looping

   # Convert timestamp to string so that we can perform next step
   tweet.timestamp = datetime.datetime.strftime(tweet.timestamp, '%Y-%m-%d %H:%M:%S')

   # vars() converts object to dictionary
   list_of_json.append(vars(tweet))

json.dump(list_of_json, jsonfile)

#
#               Print tweets to screen
#

if printTweets == True:

   for tweet in list_of_json:
```

```
    print(tweet['text'])
```

APPENDIX A.3

append_twitter_sentiment.py

```
###############################################################################################
' Reads file (obtained by executing TwitterScrape.py) holding tweets in JSON format '
' It then calculates the sentiment of each tweet and writes all file content with   '
' the sentiment score appended to an output JSON file                               '
###############################################################################################

import sys
import pandas as pd
import re
from textblob import TextBlob
from math import floor


' Obtain name of input and output files '
if len(sys.argv) < 3:
        print("SentimentOfTweets.py")
        print("Reads in JSON file holding tweets (accessed via 'text' attribute)")
        print("The sentiment of each tweet is calculated and the JSON file contents + the tweet sentiment score are output to a JSON
file")
        print("[Usage] python SentimentOfTweets.py <Input file name> <Output file name>")
        exit()
else:
        inputFileName = sys.argv[1]
        outputFileName = sys.argv[2]


def calculate_tweet_sentiment(text):
        """ Calculate sentiment of tweet text """

        ' Clean up text '
        text = re.sub("[^A-Za-z0-9.:,'/?!#@]", " ", text)
        text = text.strip()

        sentiment_of_sentences = []
        blob = TextBlob(text)
        for sentence in blob.sentences:
                sentiment_of_sentences.append(sentence.sentiment.polarity)

        return sum(sentiment_of_sentences)/floor(len(sentiment_of_sentences))


' Read input file intp data frame '
print("[INFO] Reading contents of " + inputFileName)
df = pd.read_json(inputFileName)

' Calculate and append sentiment score ontp data frame '
print("[INFO] Calculating the sentiment scores of tweets")
df['sentiment'] = df['text'].apply(calculate_tweet_sentiment)

' Rearrange columms '
df = df[['sentiment','text','likes','replies','retweets','user','fullname','url','html','id','timestamp']]

' Write contents of data frame to csv file '
```

```
df.to_json(outputFileName, orient='records', force_ascii=False)
print("[INFO] Saved tweet information to file " + outputFileName)
```

APPENDIX B.1

create_data_warehouse_tables.py

```
#
#
#           Python Script to create the DB Tables
#
#           Note that it also populates the dim_date table
#

from Classes.mysqldatabase import MySqlDatabase


################################################################################
' The dim_date table will be populated with daily values for the following years '
################################################################################

start_date = '2000-01-01'
end_date = '2021-01-01'


db = MySqlDatabase()

print("WARNING: If it already exists, the database " + db.dbName + " (and all tables and data) will be deleted.")
reply = raw_input("Are you sure you want to proceed (y/n)? ")
```

```
if reply.lower() == 'y':

    ' Drop database '
    db.drop_database()

    ' Create database '
    db.create_database();

    ' Create and populate dimension table '
    db.create_dimension_tables()
    db.create_stored_procedures()
    db.populate_dim_date_table(start_date,end_date)

     ' Create fact table'
    db.create_sp500_data_table()

    ' Print out message '
    print("[INFO] Data Warehouse successfully created")

else:

    print("[INFO] Database creation was aborted as affirmative response was not provided")
```

APPENDIX B.2

clean_data_warehouse.py

```
#
#
#               Python Script to clean down the DB
#
#               Simply truncates the fact table
#

from Classes.mysqldatabase import MySqlDatabase

db = MySqlDatabase()

db.truncate_fact_table()
```

APPENDIX B.3

etl_sp500.py

```
#
#               Python Script to perform ETL
#
#
#     This script does the following:
#
#               1. Reads in CSV files from the following directories
#
#               /Data/Input/SP500TimeSeriesData
#               /Data/Input/GoogleTrendsData
#                               /Data/Input/TwitterData
#
#               2. Processes these files to produce a CSV file (that contains fact table data) in
#
#                               /Data/Output
#
```

```
#               3. Loads this output file into the fact table in DB
#
#               Note that the script also:
#
#                               - Archives files in time-stamped directories
#                               - Stores any over-written records to the file replaced_records.csv
#


from Classes.extract import Extract
from Classes.transform import Transform
from Classes.load import Load


#####################################
'    Read in all source data files '
#####################################

extract = Extract()
df_sp500_tick_data = extract.read_input_files('tick')
df_sp500_google_trends_data = extract.read_input_files('gtrends')
df_sp500_twitter_data = extract.read_input_files('twitter')


########################################
' Process each individual data source '
########################################

transform = Transform()
transform.process_sp500_tick_data(df_sp500_tick_data)
transform.process_sp500_google_trends_data(df_sp500_google_trends_data)
transform.process_sp500_twitter_data(df_sp500_twitter_data)


######################
' Merge data sources '
######################

transform.integrate_data()


#########################################################################
' Identify any records in fact table that will be over-written '
#########################################################################

transform.identify_duplicate_records()


#############################################################################################
' Write output csv files holding integrated data for loading (and duplicate records) '
#############################################################################################

transform.write_to_csv()


##############################
' Load output to fact table '
##############################

load = Load()
load.load_data()
```

APPENDIX C.1

create_dim_date_table.sql

```
USE Data_Warehouse;

CREATE TABLE dim_date
(
   date_id                                    INT PRIMARY KEY,
            date                                                DATE,
            day                                                 INT NOT NULL,
            week                                                INT NOT NULL,
            month                                       INT NOT NULL,
   quarter                                  INT NOT NULL,
            year                                                INT NOT NULL,
            day_name                                    VARCHAR(9) NOT NULL,
            month_name                                  VARCHAR(9) NOT NULL,
            weekend_flag                                CHAR(1) DEFAULT 'f'
);
```

APPENDIX C.2

create_stored_procedures.sql

```
DROP PROCEDURE IF EXISTS fill_dim_date;
DELIMITER //
CREATE PROCEDURE fill_dim_date(IN startdate DATE,IN stopdate DATE)
BEGIN
   DECLARE currentdate DATE;
   SET currentdate = startdate;
   WHILE currentdate < stopdate DO
      INSERT INTO dim_date VALUES (
                  YEAR(currentdate)*10000+MONTH(currentdate)*100 + DAY(currentdate),
```

```
            currentdate,
            DAY(currentdate),
            WEEKOFYEAR(currentdate),
            MONTH(currentdate),
            QUARTER(currentdate),
            YEAR(currentdate),
            DATE_FORMAT(currentdate,'%W'),
            DATE_FORMAT(currentdate,'%M'),
            CASE DAYOFWEEK(currentdate) WHEN 1 THEN 't' WHEN 7 then 't' ELSE 'f' END);
    SET currentdate = ADDDATE(currentdate,INTERVAL 1 DAY);
  END WHILE;
END
//
DELIMITER ;
```

APPENDIX C.3

create_sp500_data_table.sql

```
USE Data_Warehouse;

CREATE TABLE sp500_data
(
  date_id                   INT PRIMARY KEY,
        open                        FLOAT(10,3),
        high                        FLOAT(10,3),
        low                         FLOAT(10,3),
        close               FLOAT(10,3),
        adj_close           FLOAT(10,3),
        volume              BIGINT,
        svi                         FLOAT(10,3),
        tweets_total        INT,
        positive_tweets     INT,
        neutral_tweets      INT,
        negative_tweets     INT,
        inserted            TIMESTAMP
);
```

APPENDIX C.4

import_sp500_data.sql

```
LOAD DATA LOCAL INFILE '../Data/Output/fact_table_data.csv'
REPLACE INTO TABLE sp500_data
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
(
  date_id,
  open,
  high,
  low,
  close,
  adj_Close,
  volume,
  svi,
        tweets_total,
        positive_tweets,
        neutral_tweets,
        negative_tweets,
        inserted
);
DELETE FROM sp500_data where date_id='0';
```

APPENDIX D.1

mysqldatabase.py

```
#
# Read in a downloaded Yahoo Finance CSV file and load it to the mysql DB
#
# Note that I manually created the database "Data_Warehouse" by executing the following in mysql
#
#          CREATE DATABASE Data_Warehouse;
#

import sys
import MySQLdb
import pandas as pd
from subprocess import Popen, PIPE

class MySqlDatabase:

  def __init__(self):

    self.dbUser = 'root'
    self.dbPassword = 'mysql'
    self.dbHost = 'localhost'
    self.dbName = 'Data_Warehouse'
    self.dbPort = 3307

    self.connection = None
    self.cursor = None


  def connect(self):
    """ connect to the mysql database """
    try:
      self.connection = MySQLdb.connect(host=self.dbHost, user=self.dbUser, passwd=self.dbPassword, db=self.dbName,
port=self.dbPort)
      self.cursor = self.connection.cursor()
    except:
      print("[ERROR] An error occurred while establishing connection")


  def close_connection(self):
    """ close the database connection """
    try:
      self.cursor.close()
      self.connection.close()
    except:
      print("[ERROR] An error occurred while closing connection")
```

```
def execute_query(self, sQuery):
    """ execute a sql auery """
    try:
        self.cursor.execute(sQuery)
        return self.cursor.fetchall()
    except:
        print("[ERROR] An error occurred while executing query")


def execute_sql_script(self, sqlscript):
    """ Executes a sql script """
    try:
        sQuery = "source " + sqlscript

        ' DB login details are stored in file ~/.my.cnf '
        process = Popen('mysql %s' % (self.dbName), stdout=PIPE, stdin=PIPE, shell=True)
        output = process.communicate(sQuery)[0]
    except:
        print('[ERROR] ' + str(sys.exc_info()[1]))


def create_database(self, db_name=''):
    """ Create a database """
    try:
        if db_name == '':
            db_name = self.dbName

        print("[INFO] Creating database " + db_name)
        sQuery = "create database " + db_name
        conn = MySQLdb.connect(host=self.dbHost, user=self.dbUser, passwd=self.dbPassword)
        cursor = conn.cursor()
        cursor.execute(sQuery)
        row = cursor.fetchone()
        cursor.close()
        conn.close()
        print("[INFO] The database " + db_name + " has been created")
        return row
    except:
        print('[ERROR] ' + str(sys.exc_info()[1]))


def drop_database(self, db_name=''):
    """ Delete a database """
    try:
        if db_name == '':
            db_name = self.dbName

        print("[INFO] Dropping database " + db_name)
        sQuery = "drop database if exists " + db_name
        conn = MySQLdb.connect(host=self.dbHost, user=self.dbUser, passwd=self.dbPassword)
        cursor = conn.cursor()
        cursor._defer_warnings = True
        cursor.execute(sQuery)
        row = cursor.fetchone()
        cursor.close()
        conn.close()
        print("[INFO] Database " + db_name + " dropped")
        return row
    except:
        print('[ERROR] ' + str(sys.exc_info()[1]))


def drop_table(self, table):
    """ Remove the table """
    try:
        sQuery = "drop table " + self.dbName + "." + table
        self.connect()
        self.cursor.execute(sQuery)
        row = self.cursor.fetchone()
        self.close_connection()
        return row
```

```
    except:
        print('[ERROR] ' + str(sys.exc_info()[1]))


def truncate_table(self, table):
    """ truncate the table """
    try:
        sQuery = "truncate " + self.dbName + "." + table
        self.connect()
        self.cursor.execute(sQuery)
        row = self.cursor.fetchone()
        self.close_connection()
        return row
    except:
        print('[ERROR] ' + str(sys.exc_info()[1]))


def truncate_fact_table(self):
    """ truncate the fact table """
    print("[INFO] Truncating sp500_data fact table")
    self.truncate_table('sp500_data')
    print("[INFO] Contents of sp500_data fact table have been deleted")


def populate_dim_date_table(self, startDate, endDate):
    """ Populates the dim_date table with dates """
    try:
        self.connect()
        args = (startDate, endDate)
        result = self.cursor.callproc('fill_dim_date', args)
        self.connection.commit()
        self.close_connection()
        print("[INFO] Table dim_date has been populated with dates from " + startDate + " -> " + endDate)
        return result
    except:
        print('[ERROR] ' + str(sys.exc_info()[1]))


def read_date_surrogate_keys(self):
    """ Reads the primary keys and dates from dim_date table in DB """
    try:
        sQuery = "select date_id,date from " + self.dbName + ".dim_date"
        self.connect()
        self.cursor.execute(sQuery)
        data = self.cursor.fetchall()
        self.close_connection()

        ' Convert output into dataframe '
        df_keys = pd.DataFrame.from_records(list(data), columns=['date_key','date'])

        ' Convert datetime column to string '
        df_keys['date'] = df_keys['date'].astype(str)

        return df_keys
    except:
        print('[ERROR] ' + str(sys.exc_info()[1]))


def retrieve_fact_table_data(self):
    """ Read all contents from fact table """
    try:
        print("[INFO] Reading data in fact table")

        sQuery = "select * from " + self.dbName + ".sp500_data order by date_id"
        self.connect()
        self.cursor.execute(sQuery)
        data = self.cursor.fetchall()
        self.close_connection()

        ' Convert output into dataframe '
        df = pd.DataFrame.from_records(list(data))
```

```
        ' Define column names of data frame '
        if df.empty == False:
            df.columns = ['date_key','open','high','low','close','adj_close','volume','interest','total','positive','neutral','negative','inserted']

        return df
    except:
        print('[ERROR] ' + str(sys.exc_info()[1]))


def create_sp500_data_table(self):
    sqlscript = "./SqlScripts/create_sp500_data_table.sql"
    self.execute_sql_script(sqlscript)
    print("[INFO] Fact table has been created")


def create_stored_procedures(self):
    """ We create any required stored procedures needed for our database """
    sqlscript = "./SqlScripts/create_stored_procedures.sql"
    self.execute_sql_script(sqlscript)
    print("[INFO] Stored procedure fill_dim_date() has been created")


def create_dimension_tables(self):
    sqlscript = "./SqlScripts/create_dim_date_table.sql"
    self.execute_sql_script(sqlscript)


def import_sp500_data(self):
    sqlscript = "./SqlScripts/import_sp500_data.sql"
    self.execute_sql_script(sqlscript)
    print("[INFO] Successfully loaded processed data in output file fact_table_data.csv to the fact table")
```

APPENDIX D.2

extract.py

```
import sys
import glob
import pandas as pd
from Classes.archive import Archive


class Extract:
    """ Methods responsible for reading input data sources """

    def __init__(self):

        ' Location, file mask and file format for each data source  '
        self.source_dir =
{'tick':'../Data/Input/SP500TimeSeriesData/','gtrends':'../Data/Input/GoogleTrendsData/','twitter':'../Data/Input/TwitterData/'}
        self.file_mask = {'tick':'*.csv','gtrends':'*.csv','twitter':'*.json'}
        self.file_format = {'tick':'csv','gtrends':'csv','twitter':'json'}

        ' Label for each data source (Used in log messages) '
        self.log_label = {'tick':'S&P 500 Tick','gtrends':'Google Trends','twitter':'Twitter'}



    def read_input_files(self, data_src='tick'):
        """ Read input files holding raw (collected) data into data frame """
        try:
            ' Set location, file mask and file format of data source '
            inputDir = self.source_dir[data_src]
            filemask = self.file_mask[data_src]
            fileType = self.file_format[data_src]
```

```
        ' Print log message '
        print("[INFO] Extracting {} Data".format(self.log_label[data_src]))

        ' Obtain list of files in input directory '
        files = glob.glob(inputDir + filemask)

        ' Sort list '
        files.sort()

        ' Data frame to store all the data in '
        df_all_input_data = pd.DataFrame()

        ' Exit if no files are found '
        if len(files) == 0:
            print("[WARNING] No data matching " + filemask + " was found in the directory " + inputDir)
            print("[WARNING] Any data loaded to the fact table may therefore be incomplete")
            return df_all_input_data


        ########################################
        ' Read all files in input directory '
        ########################################

        ' Create time-stamped archive directory '
        inputFileArchive = Archive(inputDir)

        for inputfile in files:

            ' Read contents of input file '
            print("[INFO] Reading contents of file " + inputfile)
            if fileType == 'csv':
                df = pd.read_csv(inputfile,header=None)
            else:
                ' Assume files contain json '
                df = pd.read_json(inputfile)

            ' Append data to data frame '
            df_all_input_data = pd.concat([df_all_input_data,df])

            ' Archive input file '
            inputFileArchive.archive_file(inputfile)

        return df_all_input_data
    except:
        print('[ERROR] ' + str(sys.exc_info()[1]))
```

APPENDIX D.3

transform.py

```
import re
import sys
import datetime
import numpy as np
import pandas as pd
from textblob import TextBlob
from math import floor
from Classes.mysqldatabase import MySqlDatabase


class Transform:
    """ Holds methods responsible for processing the raw input data """

    def __init__(self):
```

```
    ' Define data frames that will hold the transformed data from our three data sources '
    self.df_sp500_tick_data = pd.DataFrame(columns=['date_key','open','high','low','close','adj_close','volume'])
    self.df_sp500_google_trends_data = pd.DataFrame(columns=['date_key','interest','isPartial'])
    self.df_sp500_twitter_counts = pd.DataFrame(columns=['date_key','negative','neutral','positive','total'])

    ' Define data frame that will hold records with duplicate keys '
    self.df_duplicates = pd.DataFrame()

    ' Data frame that will hold the processed and integrated data to be loaded to DB fact table '
    self.output_data = None

    ' Path and name of output files'
    self.outputFileName = '../Data/Output/fact_table_data.csv'
    self.duplicatesFileName = '../Data/Output/replaced_records.csv'


def process_sp500_tick_data(self, df_sp500_tick_data):
    """ Prepare the S&P 500 tick data for loading into Data Warehouse """
    try:
        print("[INFO] Processing S&P 500 Tick Data")

        ' If no data was ingested there is nothing to process! '
        if df_sp500_tick_data.empty == True:
            return df_sp500_tick_data

        ' Define connection to our DB '
        db = MySqlDatabase()

        ' Add column names to dataframe '
        df_sp500_tick_data.columns = ['date','open','high','low','close','adj_close','volume']

        ' Read surrogate keys from dim_date table '
        print("[INFO] Reading surrogate keys for date dimension from database")
        dim_date = db.read_date_surrogate_keys()

        ' Replace date with surrogate key in dim_date table '
        print("[INFO] Replacing dates with surrogate keys")
        df_sp500_tick_data = pd.merge(df_sp500_tick_data, dim_date, how='inner', on='date')
        df_sp500_tick_data.drop(['date'], axis=1, inplace=True)

        ' Rearrange columns in correct order '
        df_sp500_tick_data = df_sp500_tick_data[ ['date_key','open','high','low','close','adj_close','volume'] ]

        print("[INFO] S&P 500 ticker data has been processed")

        self.df_sp500_tick_data = df_sp500_tick_data
    except:
        print('[ERROR] ' + str(sys.exc_info()[1]))



def process_sp500_google_trends_data(self, df_sp500_google_trends_data):
    """ Prepare the S&P 500 Google Trends data for loading into Data Warehouse """
    try:
        print("[INFO] Processing Google Trends Data")

        ' If no data was ingested there is nothing to process! '
        if df_sp500_google_trends_data.empty == True:
            return df_sp500_google_trends_data

        ' Define connection to our DB '
        db = MySqlDatabase()

        ' Add column names to dataframe '
        df_sp500_google_trends_data.columns = ['date','interest','isPartial']

        ' Read surrogate keys from dim_date table '
        print("[INFO] Reading surrogate keys for date dimension from database")
        dim_date = db.read_date_surrogate_keys()

        ' Replace date with surrogate key in dim_date table '
```

```
        print("[INFO] Replacing dates with surrogate keys")
        df_sp500_google_trends_data = pd.merge(df_sp500_google_trends_data, dim_date, how='inner', on='date')
        df_sp500_google_trends_data.drop(['date'], axis=1, inplace=True)

        ' Rearrange columns in correct order '
        df_sp500_google_trends_data = df_sp500_google_trends_data[ ['date_key','interest','isPartial'] ]

        print("[INFO] S&P 500 Google Trends data has been processed")

        self.df_sp500_google_trends_data = df_sp500_google_trends_data
    except:
        print('[ERROR] ' + str(sys.exc_info()[1]))



def process_sp500_twitter_data(self, df_sp500_twitter_data):
    """ Prepare the S&P 500 Twitter data for loading into Data Warehouse """
    try:
        print("[INFO] Processing Twitter Data")

        ' If no data was ingested there is nothing to process! '
        if df_sp500_twitter_data.empty == True:
            return df_sp500_twitter_data

        ' Define connection to our DB '
        db = MySqlDatabase()

        ' Read surrogate keys from dim_date table '
        print("[INFO] Reading surrogate keys for date dimension from database")
        dim_date = db.read_date_surrogate_keys()

        ' Calculate date column '
        print("[INFO] Extracting dates from timestamp")
        df_sp500_twitter_data['date'] = df_sp500_twitter_data['timestamp'].apply(lambda x:str(x.date()))

        ' Replace date with surrogate key in dim_date table '
        print("[INFO] Replacing dates with surrogate keys")
        df_sp500_twitter_data = pd.merge(df_sp500_twitter_data, dim_date, how='inner', on='date')
        df_sp500_twitter_data.drop(['date'], axis=1, inplace=True)

        ' Aggregate the twitter data: Count the number of positive, neutral and negative tweets '
        twitter = Twitter()
        print("[INFO] Counting daily number of positive, negative and neutral tweets")
        df_twitter_counts = twitter.daily_counts( df_sp500_twitter_data[['date_key','sentiment']].copy() )
        print("[INFO] S&P 500 Twitter data has been processed")

        self.df_sp500_twitter_counts = df_twitter_counts
    except:
        print('[ERROR] ' + str(sys.exc_info()[1]))



def integrate_data(self):
    """ Merge the processed data sets into a single data frame """
    try:
        print("[INFO] Merging data sets")

        ' Create empty dataframe '
        df = pd.DataFrame()

        ' Merge all data frames together '
        df = pd.merge(self.df_sp500_tick_data, self.df_sp500_google_trends_data, how='outer', on='date_key')
        df = pd.merge(df, self.df_sp500_twitter_counts, how='outer', on='date_key')

        if df.empty == False:
            ' Replace empty values with mysql nulls '
            df.fillna('\N',inplace=True)

            ' Sort rows in data frame by date '
            df.sort_values(by='date_key',inplace=True)
```

```
            ' Add time-stamp column '
            df['inserted'] = str(datetime.datetime.now())[0:-7]

            ' Arrange columns in the same order as in DB fact table '
            df = df[['date_key','open','high','low','close','adj_close','volume','interest','total','positive','neutral','negative','inserted']]

        self.output_data = df
    except:
        print('[ERROR] ' + str(sys.exc_info()[1]))



def identify_duplicate_records(self):
    """ Obtain any fact table records that will be over-written when current data is uploaded """
    try:
        print("[INFO] Identifying any records that will be overwritten")

        ' Define connection to our DB '
        db = MySqlDatabase()

        ' Retrieve all the data currently stored in fact table '
        df_fact_table_data = db.retrieve_fact_table_data()

        if df_fact_table_data.empty == False:
            ' Find subset of records that will be overwritten '
            self.df_duplicates = pd.merge(df_fact_table_data, self.output_data['date_key'].to_frame(), how='inner', on='date_key')

            ' Replace any undefined entries with \N (mysql nulls) '
            self.df_duplicates.fillna('\N',inplace=True)

            ' Log message '
            if self.df_duplicates.empty == True:
                print("[INFO] No records in fact table will be overwritten for this upload")
            else:
                print("[WARNING] There are records in the fact table that will be overwritten")
                print("[WARNING] These original records can be found in the output archive directory")
        else:
            print("[INFO] No records in fact table will be overwritten for this upload")

    except:
        print('[ERROR] ' + str(sys.exc_info()[1]))



def write_to_csv(self):
    """ Writes all output files for loading and/or archiving """
    try:
        ' CSV data to be loaded to fact table '
        print("[INFO] Writing processed data to csv file for loading to DB")
        self.output_data.to_csv(self.outputFileName, header=True, index=False)

        ' File holding any records that will be over-written '
        print("[INFO] Writing records (if any) that will be over-written to csv file")
        self.df_duplicates.to_csv(self.duplicatesFileName, header=True, index=False)

        ' Log summary info '
        print("[INFO] {} records will be loaded to fact table".format(len(self.output_data)))
        print("[INFO] {} records currently in fact table will be over-written".format(len(self.df_duplicates)))
    except:
        print('[ERROR] ' + str(sys.exc_info()[1]))




class Twitter:

    def __init__(self):

        ' Set sentiment score threshold values that define positive and negative tweets '
        self.positive_threshold = 0.0
```

```
        self.negative_threshold = -0.0


    def daily_counts(self, df_tweets):
        """ Count the number of positive, neutral and negative tweets on each day """
        try:
            ' Append and populate sentiment_category column '
            df_tweets['sentiment_category'] = np.nan
            print("[INFO] Identifying positive, neutral and negative tweets")
            df_tweets.loc[ df_tweets['sentiment'] > self.positive_threshold, 'sentiment_category' ] = "positive"
            df_tweets.loc[ df_tweets['sentiment'] < self.negative_threshold, 'sentiment_category' ] = "negative"
            df_tweets.loc[ (df_tweets['sentiment'] >= self.negative_threshold) & (df_tweets['sentiment'] <= self.positive_threshold),
'sentiment_category' ] = "neutral"

            ' Drop redundant column from dataframe '
            df_tweets.drop('sentiment',axis=1,inplace=True)

            ' Add count column needed for pivot table '
            df_tweets['count'] = 1

            ' Perform a count of sentiment category for each date '
            df_counts = pd.pivot_table(df_tweets, index=['date_key'], columns=['sentiment_category'], values='count', aggfunc=np.sum)
            print("[INFO] Daily counts of positive, neutral and negative tweets completed")

            ' Replace nulls with zeros '
            df_counts.fillna(0, inplace=True)

            ' Shift date_key index into body of data frame '
            df_counts.reset_index(inplace=True)

            ' Append total count to data frame '
            df_counts['total'] = df_counts['positive'] + df_counts['negative'] + df_counts['neutral']

            return df_counts
        except:
            print('[ERROR] ' + str(sys.exc_info()[1]))



    def calculate_tweet_sentiment(self, text):
        """ Calculate sentiment of tweet text """
        try:
            ' Clean up text '
            text = re.sub("[^A-Za-z0-9.:,'/?!#@]", " ", text)
            text = text.strip()

            sentiment_of_sentences = []
            blob = TextBlob(text)
            for sentence in blob.sentences:
                sentiment_of_sentences.append(sentence.sentiment.polarity)

            return sum(sentiment_of_sentences)/floor(len(sentiment_of_sentences))
        except:
            print('[ERROR] ' + str(sys.exc_info()[1]))
```

APPENDIX D.4

load.py

```
import sys
from Classes.mysqldatabase import MySqlDatabase
from Classes.archive import Archive
```

```
class Load:
    """ Holds methods responsible for loading data into data warehouse """

    def __init__(self):

        ' Directory holding file to be loaded '
        self.load_directory = '../Data/Output/'

        ' Name of file to load '
        self.load_file_name = self.load_directory + 'fact_table_data.csv'

        ' Name of file holding records that will be over-written '
        self.duplicates_file_name = self.load_directory + 'replaced_records.csv'



    def load_data(self):
        """ Loads data to fact table and archives all files """

        self.load_to_fact_table()
        self.archive_files()



    def load_to_fact_table(self):
        """ Loads transformed data to fact table """
        try:
            ' Define connection to our DB '
            db = MySqlDatabase()

            ' Import Data '
            db.import_sp500_data()
        except:
            print('[ERROR] ' + str(sys.exc_info()[1]))



    def archive_files(self):
        """ Archive files to time-stamped directory """
        try:
            ' Archive csv file holding loaded data '
            outputArchive = Archive(self.load_directory)
            outputArchive.archive_file(self.load_file_name)
            outputArchive.archive_file(self.duplicates_file_name)
        except:
            print('[ERROR] ' + str(sys.exc_info()[1]))
```

APPENDIX D.5

archive.py

```
import os
import sys
import shutil
import datetime


class Archive:
    """ Responsible for archiving of files """
```

```
def __init__(self, srcDir):

    ' Directory holding file to archive '
    self.src_directory = srcDir

    ' Define archive directory '
    self.archive_directory = srcDir + "/archive/" + datetime.datetime.now().strftime('%Y-%m-%d_%H-%M-%S')

    ' Create archive directory '
    self.create_archive_dir()



def create_archive_dir(self):
    """ Create a time-stamped archive directory """
    try:
        ' Create new directory'
        os.makedirs(self.archive_directory)

        ' Print message '
        print("[INFO] Archive directory " + self.archive_directory + " has been created")
    except:
        print('[ERROR] ' + str(sys.exc_info()[1]))



def archive_file(self, file_name):
    """ Moves file into a time-stamped archive directory """
    try:
        ' Obtain name of file to archive '
        fileName = file_name.split('/')[-1]

        ' Construct full path name of file '
        fullPathName = self.src_directory + "/" + fileName

        ' Archive file '
        shutil.move(fullPathName, self.archive_directory)

        ' Print message '
        print("[INFO] " + fileName + " backed up to " + self.archive_directory)
    except:
        print('[ERROR] ' + str(sys.exc_info()[1]))
```