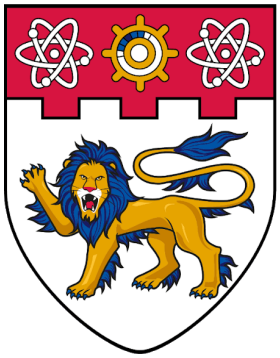


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Semester 2, AY2020-21

CZ4034 Information Retrieval

Course Assignment

Members	Matriculation Number
Phoe Chuan Bin	U1821679J
Chen Gangzhe	U1822840E
Lai Weng Hong	U1840470B
Lee Chia Zhe	U1920924D

Course Coordinator:
Erik Cambria

Table of Contents

Overview

In this age of advanced information, we are flooded with lots of information every day. Without any processing, these data are meaningless and unusable. In the stock market, every piece of news may have either a positive or negative impact on a certain stock. Social media platforms such as Twitter allow both retail and professional investors to voice their opinions and analysis on a stock at any time. It is time-consuming for investors to read through every single tweet to gauge the sentiment and forecast of the stock. In this assignment, we will be scraping tweets from Twitter to analyse whether a stock is bullish or bearish.

Data from Twitter are chosen as many researches show that there is significant correlation between twitter data and the stock market. We scraped data from 12 different twitter investing accounts. We then indexed the data using Solr. Using Python libraries such as *nltk* and *sklearn*, we conducted data preprocessing and classification. Sentiment analysis is used to gauge the mood expressed in the tweets (bullish or bearish). We built a web application which allows the user to query the database, sort results and update the database real-time.

In this application, the user can search for the stock that they are interested in and apply filters to the search so that they can get the news as close as possible to what they want. The user can choose to get the data from sources they prefer through the filter to feed into the analyser and they can get feedback regarding the stock that they are interested in from the system. With this, investors are able to save up a lot of time reading up news and analyse the stock manually.

YouTube link:
https://youtu.be/_tStMKGaR-g

Link to file with crawled text data, queries and their results, manual classifications, automatic classification results, and any other data for Questions 3 and 5:
https://drive.google.com/drive/folders/1NkwXK_vQtUAp_q2TcdM6YvlajRNZkNK8?usp=sharing

Link to source code:
https://github.com/chuanbinp/informationRetrieval

Question 1:

1) **How you crawled the corpus (e.g., source, keywords, API, library) and stored it (e.g., whether a record corresponds to a file or a line, meta information like publication date, author name, record ID)**

In this project, we crawled stock tweets from Twitter pages that provide up-to-date news or price projection estimates for equities based in the United States - specifically, stocks that are publicly traded in exchanges like the NASDAQ and NYSE and not OTC securities. Such tweets are depicted in Figure 1.

Figure 1. Example tweets that fits our criteria

Prepopulating our Database

After inspecting various sources, we came up with a comprehensive list of 12 pages that fit our criteria which can be seen in Figure 2.

Twitter Pages to be scrapped	Number of Tweets
EliteOptions2	2000
OnlyGreenTrades	1326 (max)
stockstobuy	2000
SeekingAlpha	2000
WarlusTrades	2000
Ultra_Calls	2000
TickerReport	2000

MarketRebels	2000
canuck2usa	2000
MarketBeatCom	2000
AmericanBanking	2000
TradeOnTheWire1	2000
Total	**23,326**

Figure 2. Twitter pages to be used scrapped for this project

This is done by using the Twitter API for developers and a Python library *tweepy*. Figure 3 shows the brief walkthrough of our scrapper, the full code can be found under *twitter_scraping.py*.

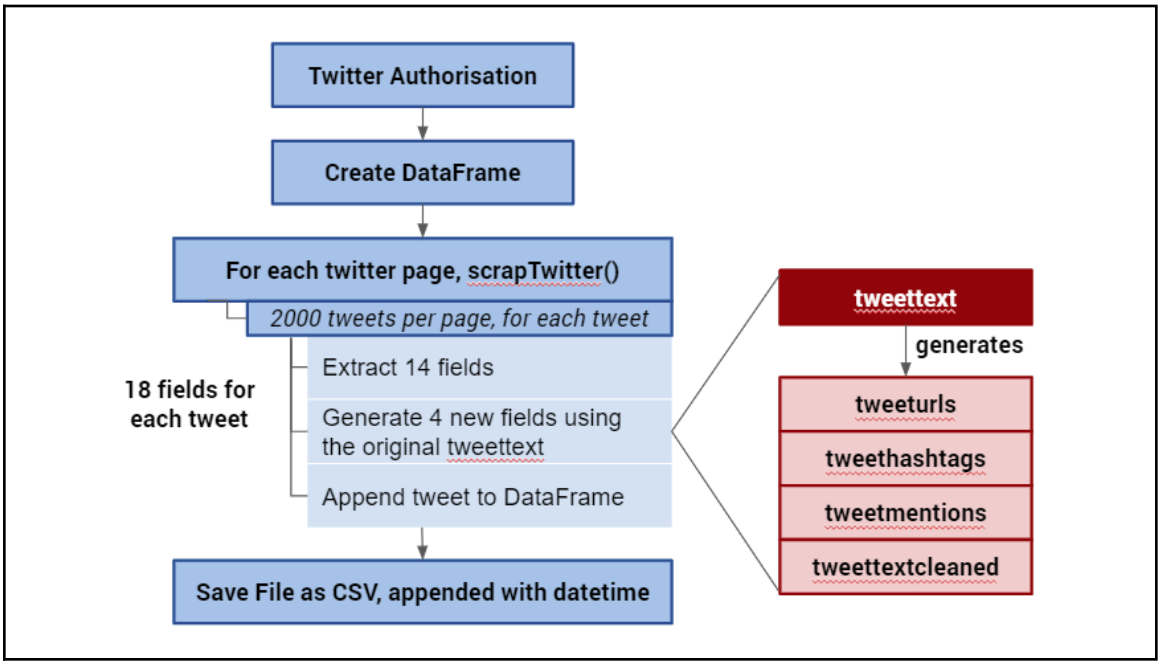


Figure 3. twitter_scraping.py code walkthrough

As Twitter API has a rate limit of 900 requests/15 minutes, and we have a large dataset to scrape, we ensured that our scrapper is robust enough to handle such “rate limit exceeded” errors. By using a try-except block, it allows our scrapper to wait and continue scraping after the 15-minute window is up.

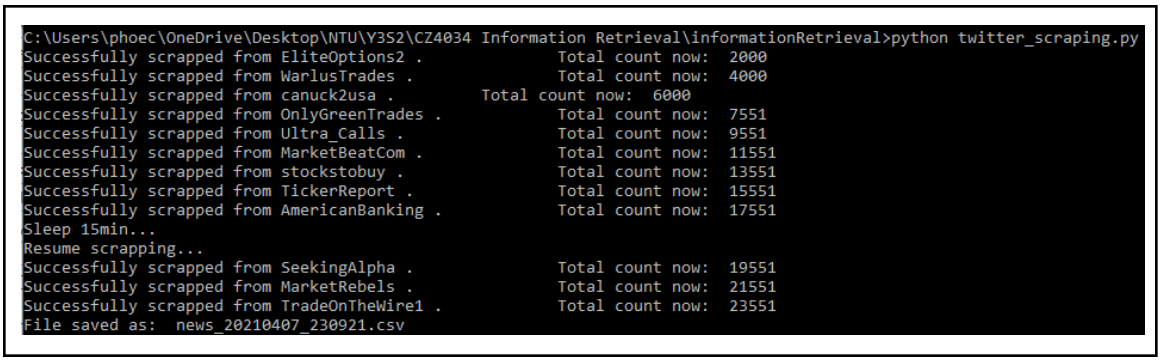


Figure 4. Scrapping robustness

We extracted 14 relevant fields from the returned JSON results. Furthermore, from the original *tweettext*, we utilised an external library *tweet-processor* to generate 4 extra fields. First, it helps to clean up the original tweet text by removing URLs, Mentions, Emojis, Smiley and Reserves. Second, it returns them as new fields.

Figure 5. Example tweet showing the 18 extracted fields and the tweet preprocessing

Cleaning the *tweettext* to *tweettextcleaned* is important as we will use it for indexing later. The other fields are extracted to be used in our frontend search engine to increase its usability.

Once this has been done, we run the *inject_data.py* script to inject it into our tweets collection in SOLR. To prevent duplicates, this script will:

1. Remove all existing data
2. Read in the csv file as a dictionary and inject into the SOLR tweets collection via API

Actively listening to new tweets for a real-time Database

To ensure that our data is always relevant to our users, we developed a worker script *twitter_stream_to_db.py* to listen to new tweets from all 12 Twitter pages. We utilise event-driven programming to achieve our goal of having a real-time dynamic database. Whenever there is a new tweet, we will:

1. Preprocess the tweet to have the same structure as our prepopulated tweets as the JSON format returned is different via Stream
2. Add that tweet to our database
3. Search and remove the oldest tweet based on *tweetcreatedts*
4. Inform server that there is a new tweet

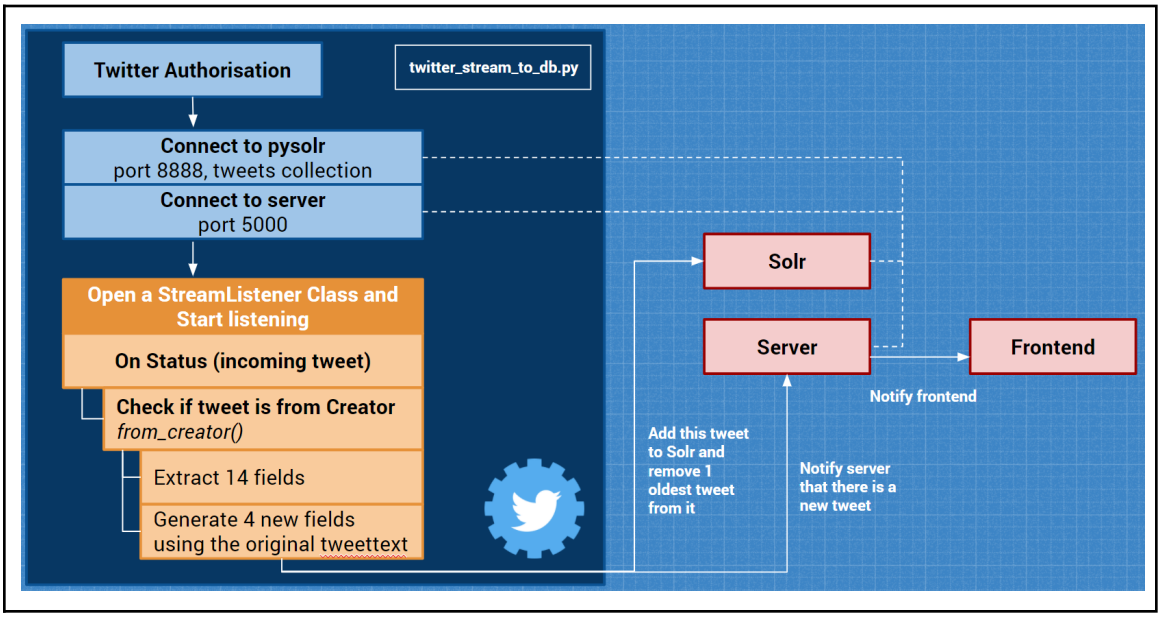


Figure 6. twitter_stream_to_db.py code walkthrough

The streaming ids that were used are listed below.

Twitter Handles	Streaming Ids
EliteOptions2	1018324467758465024
OnlyGreenTrades	748611168168644612
stockstobuy	430841130
SeekingAlpha	779131850023378944
WarlusTrades	89517375
Ultra_Calls	68559732
TickerReport	37564410
MarketRebels	1979190776

canuck2usa	61661638
MarketBeatCom	23059499
AmericanBanking	817007725666242561
TradeOnTheWire1	51912109

Figure 7. Streaming Ids for each Twitter handle

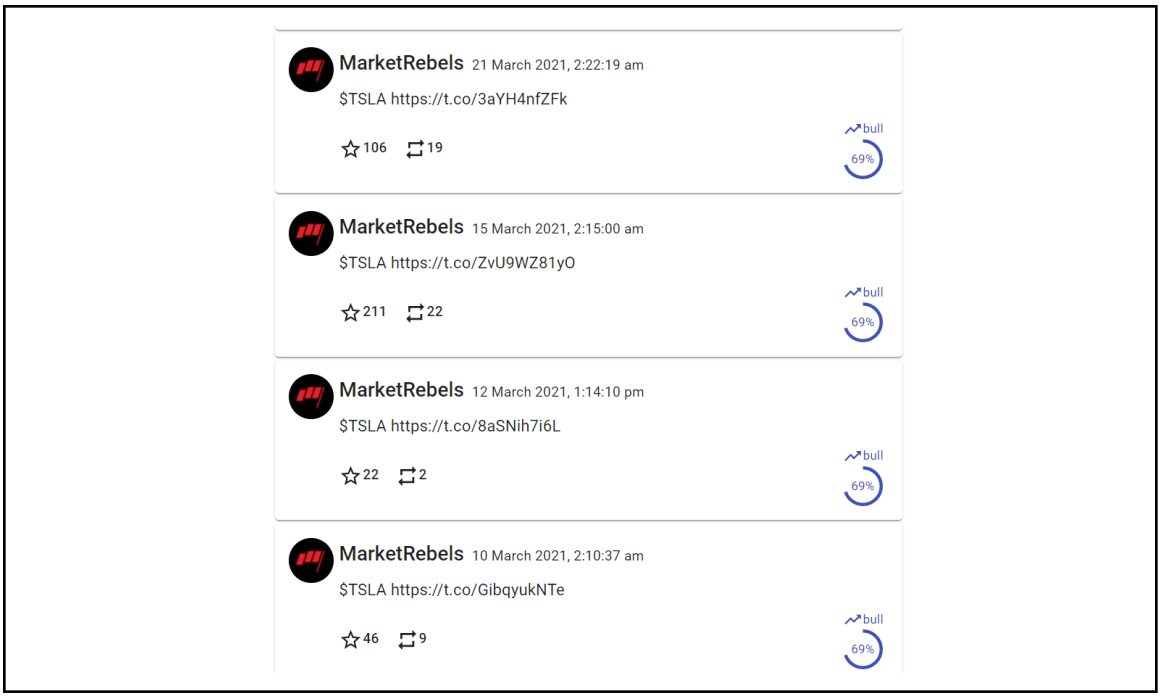
2) What kind of information users might like to retrieve from your crawled corpus (i.e., applications), with sample queries

From the application, the users may:

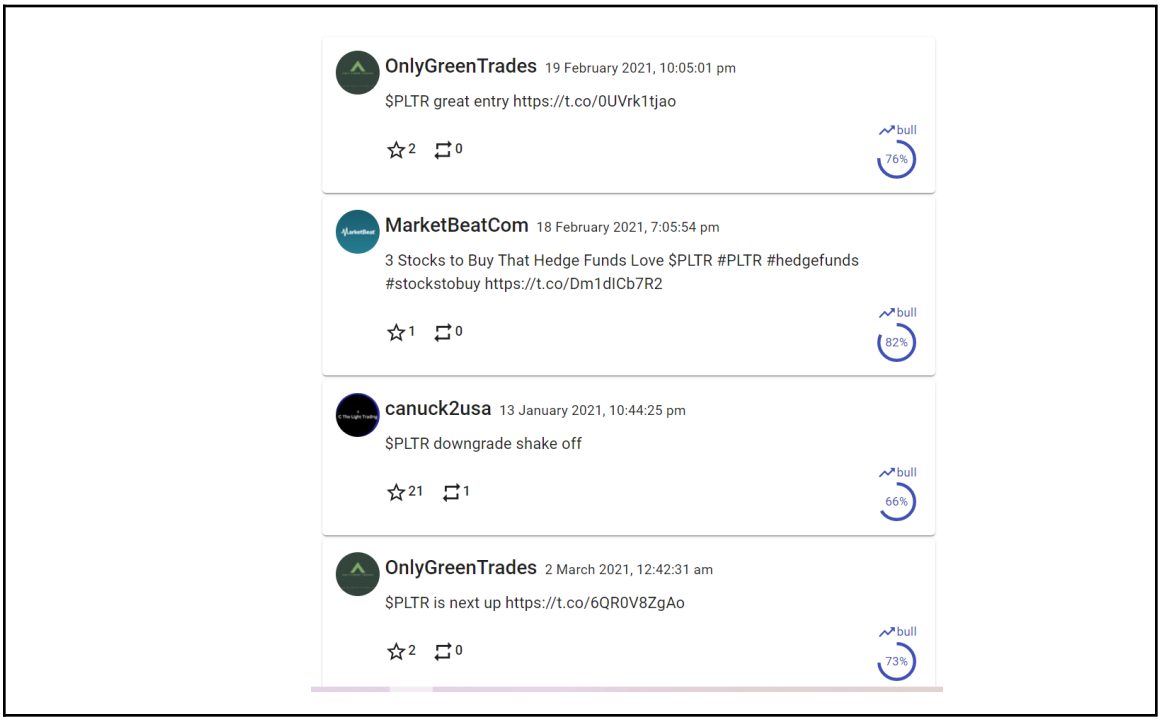
1. Retrieve the latest information in real time regarding the stock they bought to check if there are any changes or news that may affect stock price so that they can react immediately to it.
2. The users can also make a query on the application to estimate whether a stock is bullish or bearish so that they can have more precise decisions to buy or to sell.
3. Get the analysis about a stock from the sources they prefer, e.g. American Banking and OnlyGreenTrades to process so that the computed result is based on American Banking and OnlyGreenTrades only.

Sample queries:

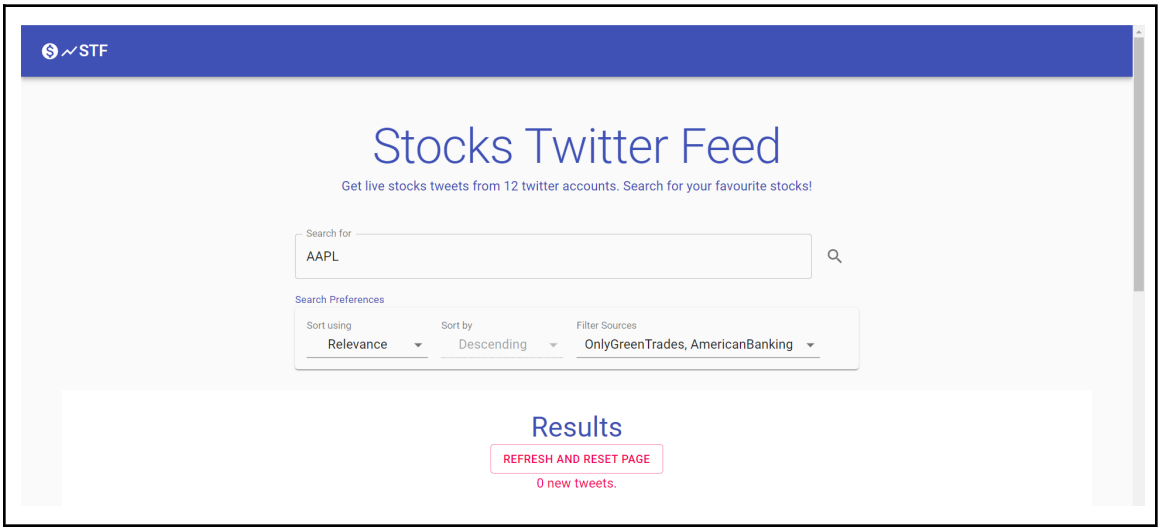
1. If the user bought Tesla, he/she can search for it to get an analysis regarding the stock anytime.

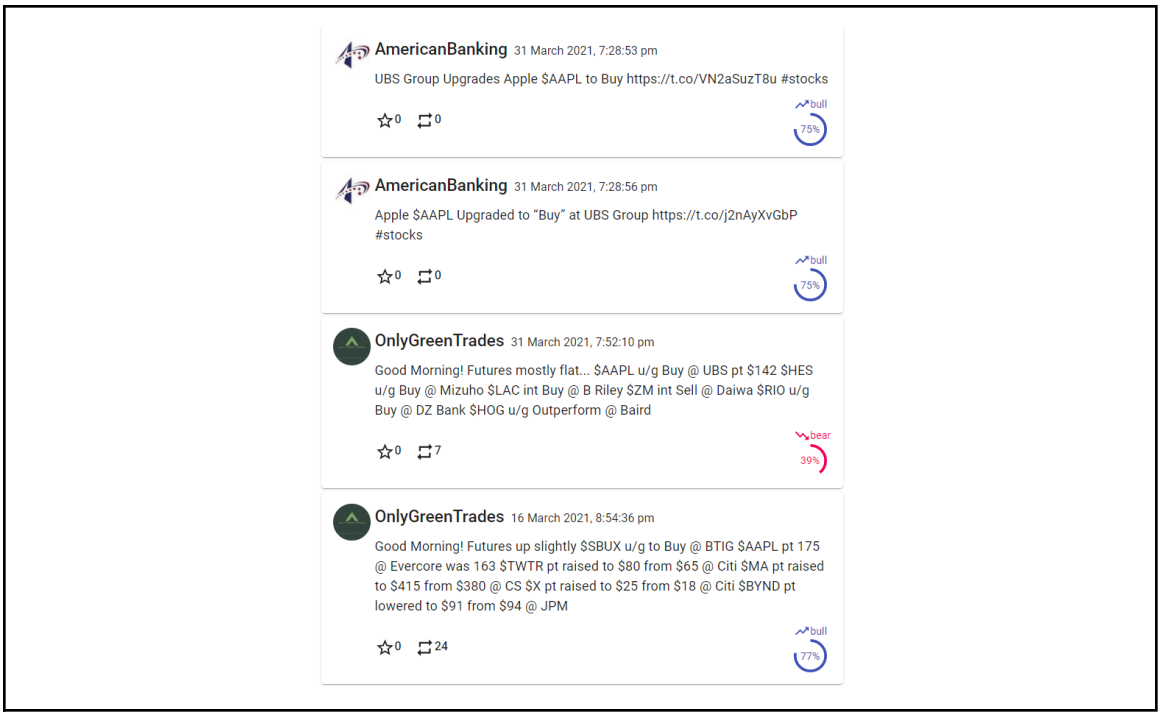


2. If the user thinks that PLTR has a good potential to rise, he can search for the tweets regarding the stock and get an analysis about the stock to help him make a more precise decision.



3. If the user likes sources from American Banking and OnlyGreenTrades, he/she can search for the stock using “Filter Sources” and choose American Banking and OnlyGreenTrades.





3) The numbers of records, words, and types (i.e., unique words) in the corpus

```
In [13]: data.shape
Out[13]: (23326, 18)

In [12]: data['tweettextcleaned'].str.split().str.len().sum()
Out[12]: 366912.0
```

Figure 8. Corpus length

We have a total of 23,326 records and 366,912 words in our *tweettextcleaned* field which is used for indexing. The fields are shown in Figure 9.

Field	**Description**
username	Username of the tweet originator
userdesc	Description of the tweet originator
userurl	URL to the tweet originator page
userpic	URL of the tweet originator's display picture
userlocation	Tweet originator's location

userfollowing	Number of users the tweet originator is following
userfollowers	Number of followers the tweet originator has
usertotaltweets	Total number of tweets the tweet originator has
usercreatedts	Timestamp of when the tweet originator's account is created
tweetid	Unique Id of this tweet
tweetcreatedts	Timestamp of when this tweet is posted
tweetretweetcount	Number of retweets this tweet has garnered
tweetfavcount	Number of favourites this tweet has garnered
tweeturls	URLs (http://..) present in the tweettext field
tweethashtags	Hashtags (#...) present in the tweettext field
tweetmentions	Mentions (@...) present in the tweetext field
tweettext	Original tweet text
tweettextcleaned	Cleaned tweet text, without any URLs, Mentions, Emojis, Smiley and Reserves

Figure 9. Data dictionary of scrapped fields

We did a simple visualisation of the top 50 most common terms in our corpus as shown in Figure 10 and noticed that it is dominated by stopwords like 'to', 'the', 'a'. The most common ticker symbol is '$tsla' followed by '$spx' which represents Tesla Motors and S&P 500 Index respectively.

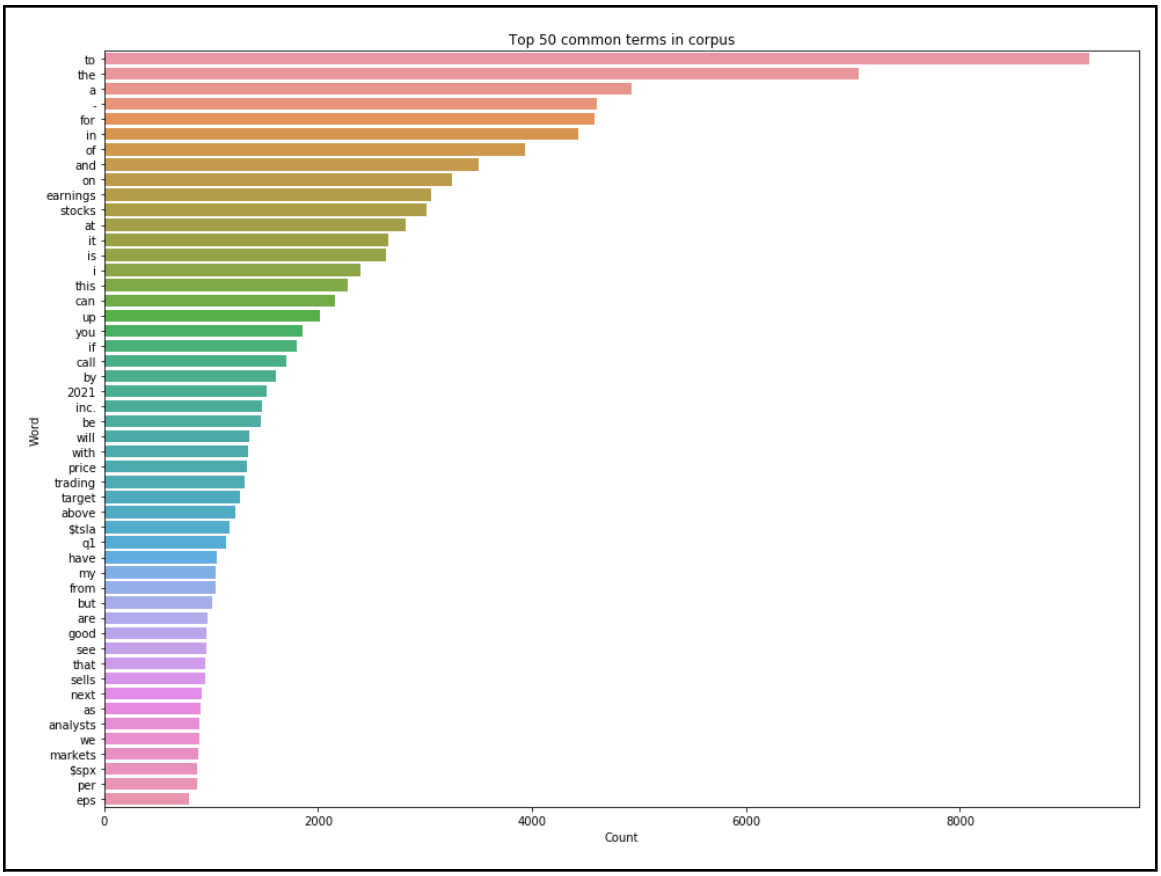


Figure 10. Top 50 most common words in our Tweet collection

Question 2:
Perform the following tasks:

1) Build a simple web interface for the search engine (e.g., Google)

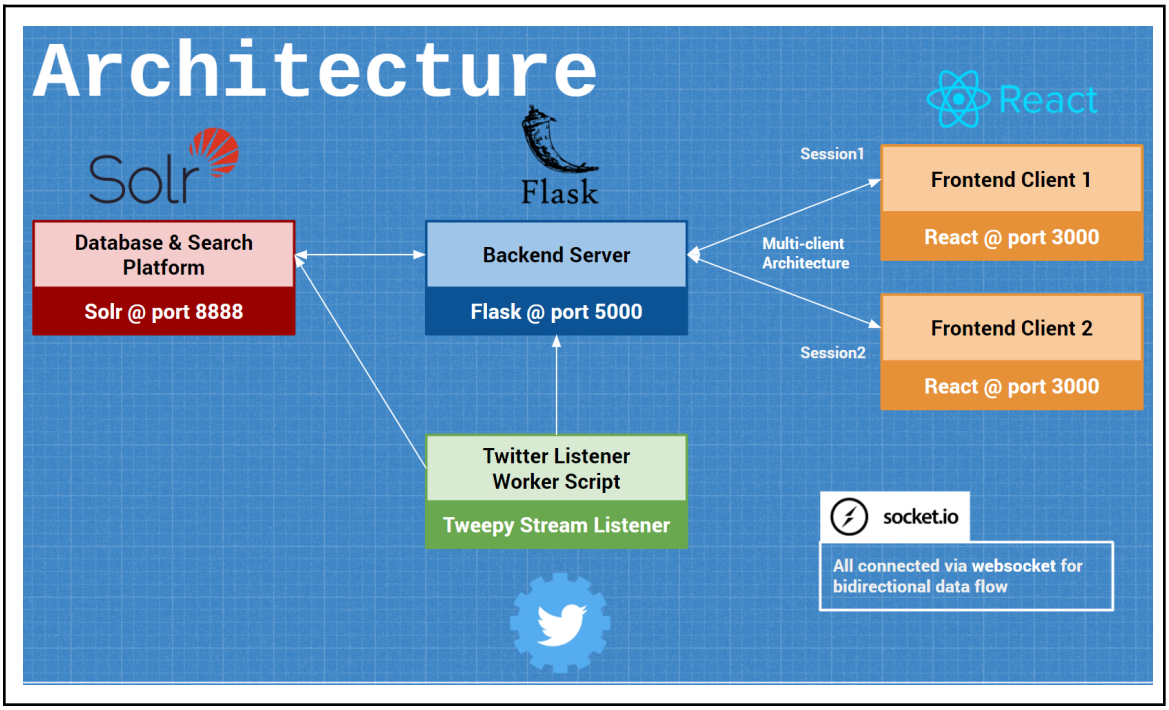


Figure 11. Client Server Architecture

Figure 11 shows our high-level architecture used to develop this project. Our flask server is running on *port 5000* and have 4 main routes that the clients and database can interact with, namely:

	Routes	**Triggered by**	**Description**
1	join	Client	When a new client connects, we use the *client_id* as our session id to create a new room. The client is added to their own room. We then get all data sorted in descending *tweetcreatedts* order and emit it back to that specific room.
2	search	Client	Retrieve data from SOLR based on the search parameters (q, fq and sort) and emit the data back to that specific room.
3	refresh_data	Client	*new_tweet_count_dict* that keeps track on the number of tweets that has just been

			added to the database via the Twitter Streamer and yet to be seen by **each client.**
			
			`new_tweet_count_dict = {`
			`  <client_session_id_1> : count1,`
			`  <client_session_id_2> : count2,`
			`  ....`
			`}`
			
			refresh_data will reset the specific count for that client to 0 and emit back the full refreshed data sorted in descending *tweetcreatedts* order.

4	streamer_new_tweet	Stream Listener	Server is informed that a new tweet has been received and added to the database. Each value in the *new_tweet_count_dict* is therefore incremented by 1, and each client is informed on this current count.

Figure 12. Server routes

Communication protocol for Information Transfer

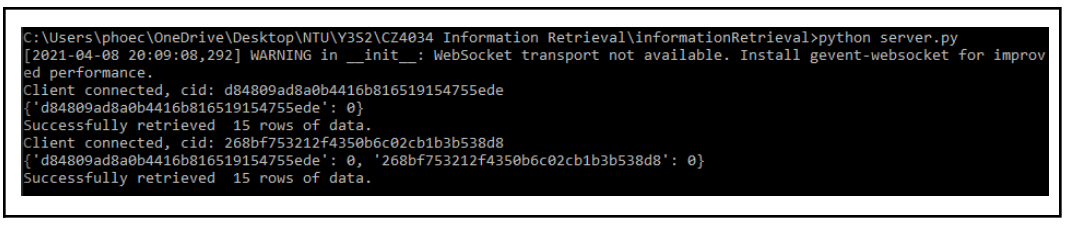
We chose to use Websocket via socketio instead of the classic REST APIs as it allows for:

1. Bi-directional flow of data from client to server and vice versa, whereas REST follows a unidirectional approach. This allows us to provide real-time feedback to clients when our listener receives new tweets.

2. WebSocket communication allows the client and server to talk independently of each other, whereas with the REST-based approach, either client is talking to the client or the server is talking to the client at any given time.

Unique Features of our Web Application

1. Multi-client Architecture
   Each client has their own room, identified uniquely by their *client_id*. This allows us to keep track and inform each client how many tweets have come in since they last refreshed their feed.

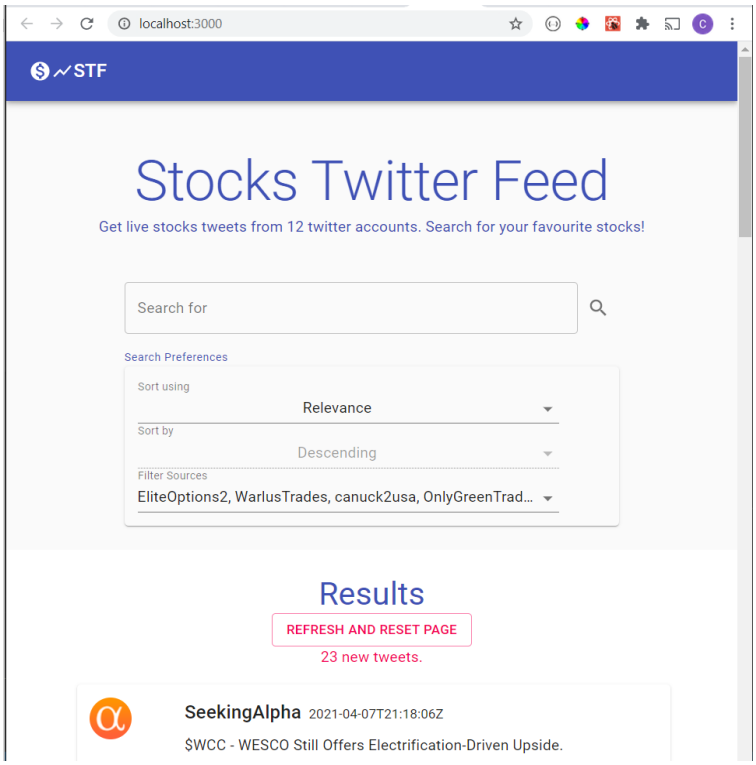


Server console

Figure 13. Connecting 2 clients to our server, displaying new_tweet_count_dict

2. Real-time dynamic database
   Live addition of new tweets from the 12 Twitter pages that we follow. Our SOLR database is therefore constantly changing as although we maintain a fixed number of records, the oldest tweet is always replaced by a new tweet. Refer to Figure 14.

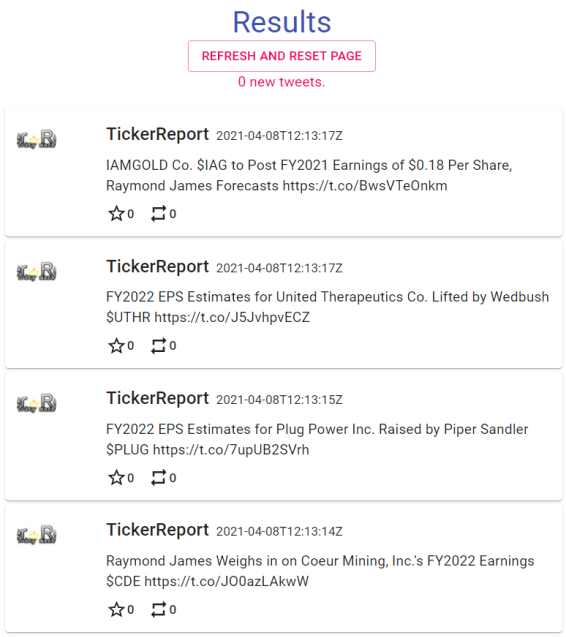
3. Real-time feedback to users
   Once data has been added to the database, our worker script will inform the server and thereafter each client of the total number of tweets that they have yet seen. Refer to Figure 14.



Before Refreshing (Client)

```
 tweetcreatedts:  2021-04-08 12:13:08
 tweettextcleaned:  Wedbush Comments on Comerica Incorporateds FY2022 Earnings $CMA stocks
Successfully added 1 new data.
Deleting the following:
  ['31df31e3-95cd-42cf-8cc0-ce4d19a149e1', ['OnlyGreenTrades'], ['2017-03-06T15:32:30Z']]
Successfully deleted  1 oldest data.

Adding the following:
 username:  TickerReport
 tweetcreatedts:  2021-04-08 12:13:13
 tweettextcleaned:  Q4 2021 EPS Estimates for Teck Resources Limited $TECK Reduced by Analyst
Successfully added 1 new data.
Deleting the following:
  ['e77b61d4-98a3-4e02-ac03-0de2449eef6d', ['OnlyGreenTrades'], ['2017-03-06T15:35:12Z']]
Successfully deleted  1 oldest data.

Adding the following:
 username:  TickerReport
 tweetcreatedts:  2021-04-08 12:13:14
 tweettextcleaned:  Raymond James Weighs in on Coeur Mining, Inc.s FY2022 Earnings $CDE
Successfully added 1 new data.
Deleting the following:
  ['129c58d9-91d8-4caf-8d5e-f20cb5438f9b', ['OnlyGreenTrades'], ['2017-03-06T15:56:15Z']]
Successfully deleted  1 oldest data.

Adding the following:
 username:  TickerReport
 tweetcreatedts:  2021-04-08 12:13:15
 tweettextcleaned:  FY2022 EPS Estimates for Plug Power Inc. Raised by Piper Sandler $PLUG
Successfully added 1 new data.
Deleting the following:
  ['1bd53cc9-140f-4b44-bcf8-6cd7a8f19428', ['OnlyGreenTrades'], ['2017-03-06T15:58:54Z']]
Successfully deleted  1 oldest data.

Adding the following:
 username:  TickerReport
 tweetcreatedts:  2021-04-08 12:13:17
 tweettextcleaned:  IAMGOLD Co. $IAG to Post FY2021 Earnings of $0.18 Per Share, Raymond James Forecasts
Successfully added 1 new data.
Deleting the following:
  ['e1f9d0ab-424b-46c3-959c-5ffa1cfe7011', ['OnlyGreenTrades'], ['2017-03-06T16:19:49Z']]
Successfully deleted  1 oldest data.

Adding the following:
 username:  TickerReport
 tweetcreatedts:  2021-04-08 12:13:17
 tweettextcleaned:  FY2022 EPS Estimates for United Therapeutics Co. Lifted by Wedbush $UTHR
Successfully added 1 new data.
Deleting the following:
  ['3ce78108-9441-4e90-bcfe-4352733d8c39', ['OnlyGreenTrades'], ['2017-03-06T16:30:12Z']]
Successfully deleted  1 oldest data.
```

Before Refreshing (Listener Console)



After Refreshing (Client)

Figure 14. Real-time database and user feedback

ReactJS is used to create our frontend client and is running on *port 3000*. Refer to Figure 15. Users can:

1. Enter their search term
2. Sort using Relevance, Favourite Count or Retweet Count
3. Sort by Descending or Ascending order
4. Filter the sources for their search results

The main functionality is Search which will communicate with the backend server to change the 'results' State variable.

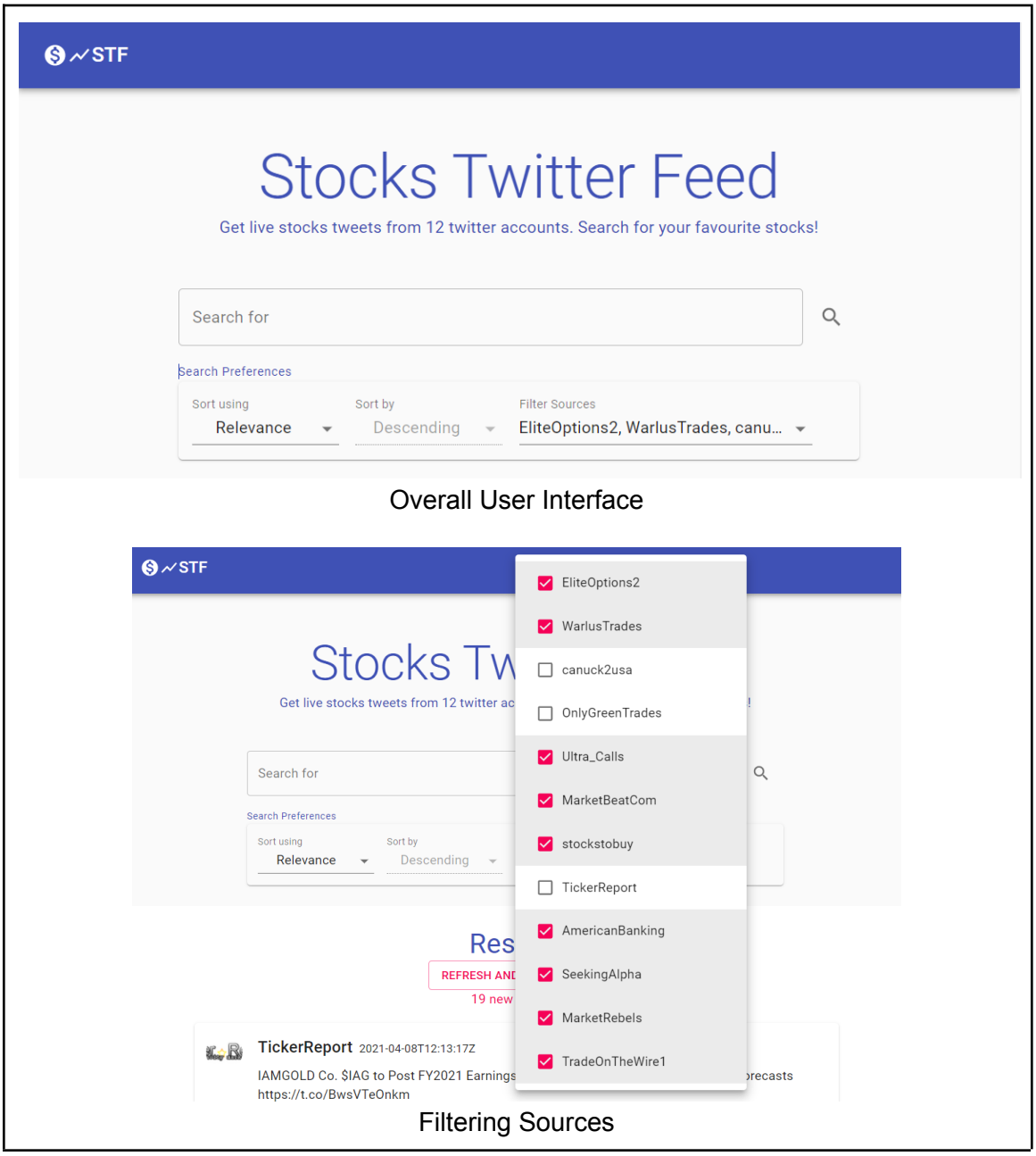


Figure 15. Frontend client user interface

Users can also click on the username and display picture to go to the tweet originator's twitter page on a new tab.

2) A simple UI for crawling and incremental indexing of new data would be a bonus (but not compulsory)

To facilitate the ease of use for our users, we built a simple python Graphical User Interface using *tkinter*. The steps to start the whole application, and the actions taken for each button is shown in Figure 16.

```
python guiInterface.py
```

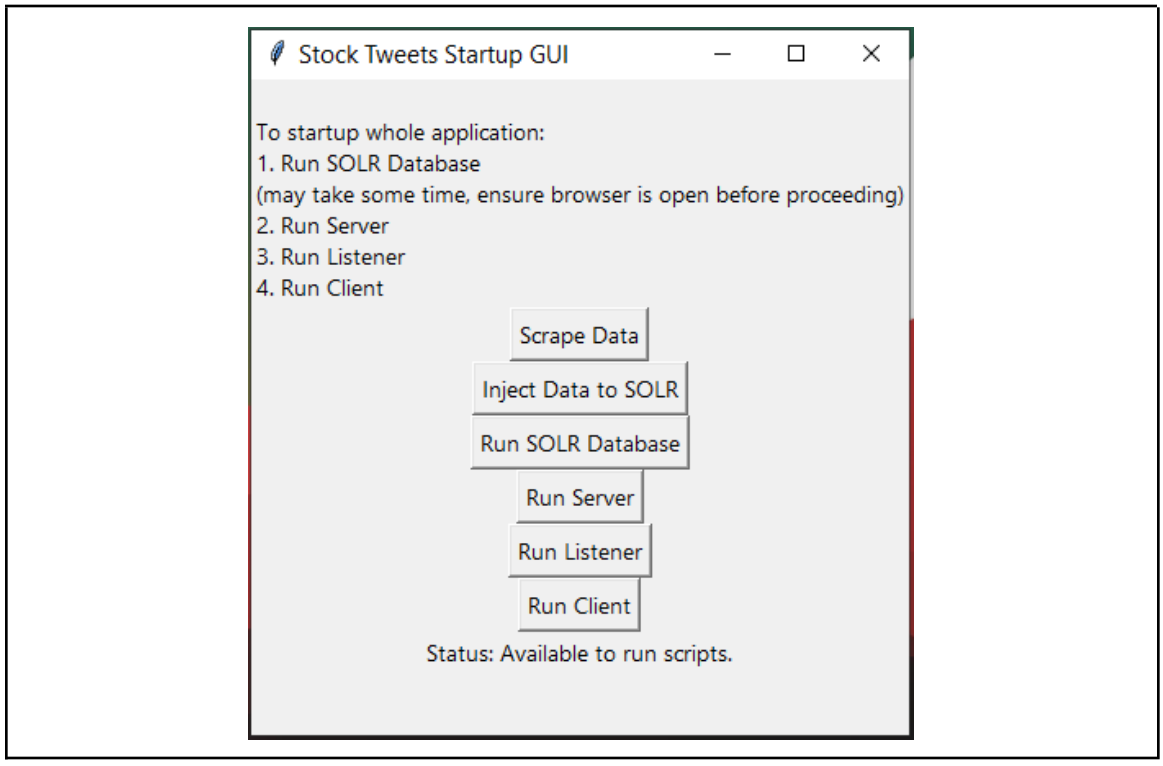
Each button will run the respective scripts in a new cmd line.

Scrape and Inject

1. Run SOLR Database
2. Scrape Data
3. Inject Data to SOLR

Start application

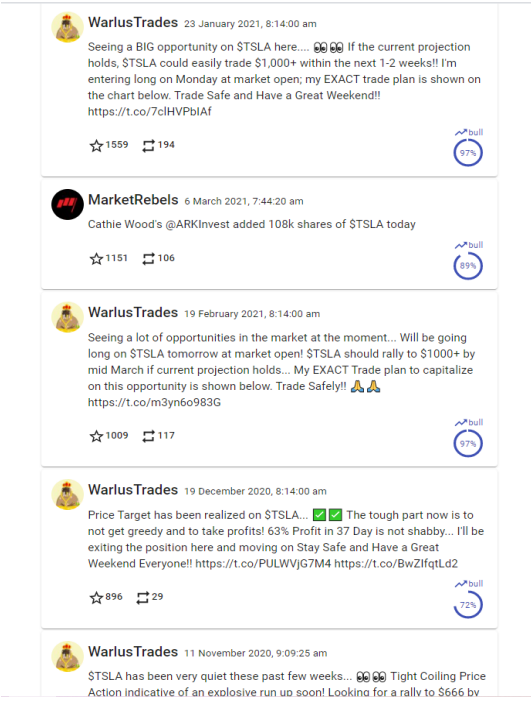
1. Run SOLR Database
2. Run Server
3. Run Listener
4. Run Client

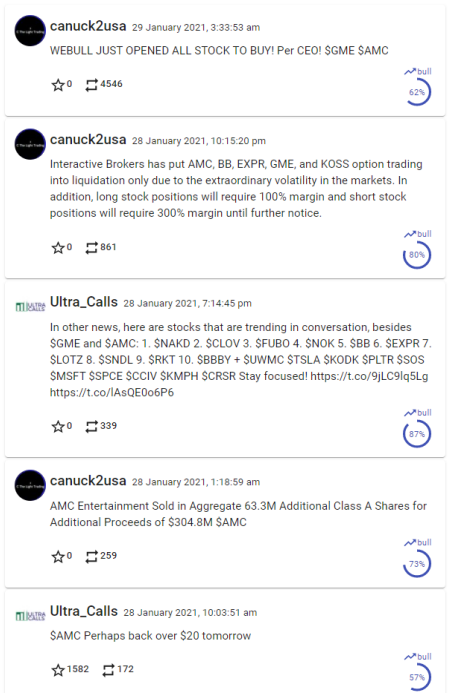
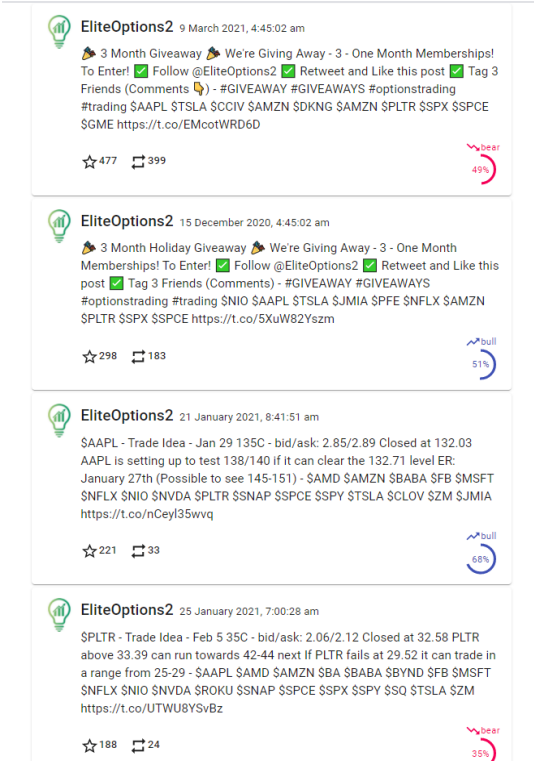


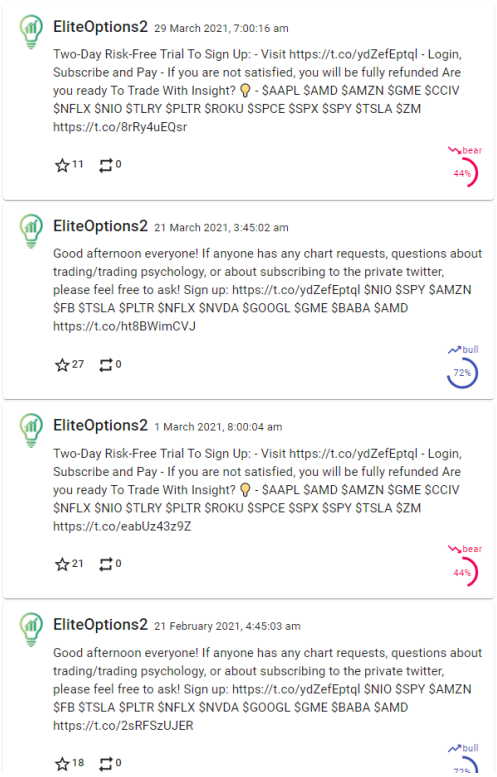
Button	Action
Scrape Data	Run *twitter_scraping.py*
Inject Data to SOLR	Run *inject_data.py*
Run SOLR Database	Change directory to solr/bin Startup solr at port 8888 Open browser http://localhost:8888/solr/#/
Run Server	Run *server.py*
Run Listener	Run *twitter_stream_to_db.py*
Run Client	Change directory to react_frontend Run *npm start* to startup React Application

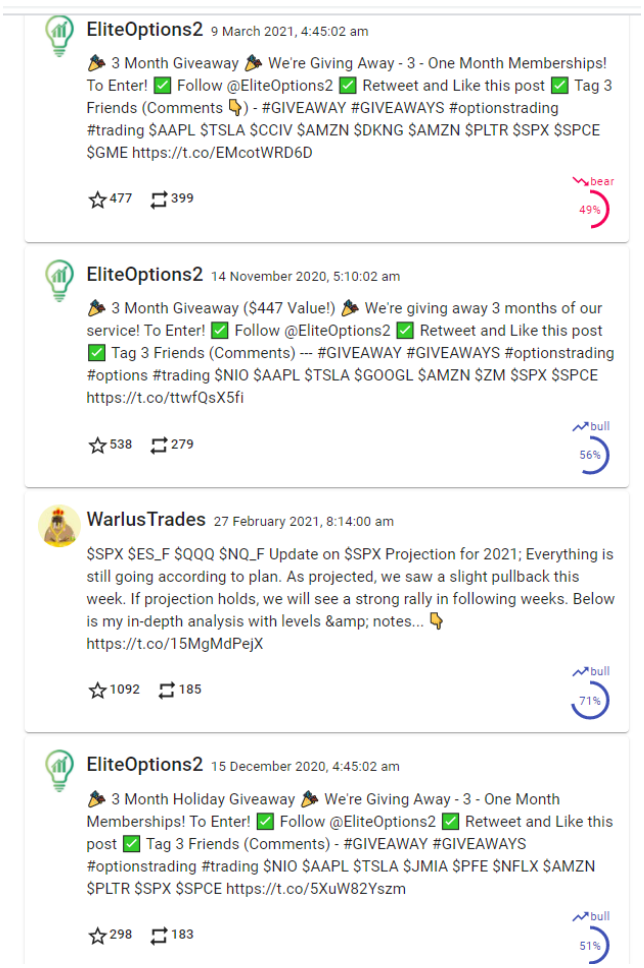
Figure 16. Python GUI to perform crawling, indexing and other actions

3) Write five queries, get their results, and measure the speed of the querying

Query	Results	Speed of Querying(ms)
Input: TSLA **Sort using:** Favourite count **Sort by:** Descending order **Sources:** EliteOptions2, OnlyGreenTrades, MarketRebels, canuck2usa, stockstobuy, SeekingAlpha, WarlusTrades, Ultra_Calls, TickerReport, MarketBeatCom, AmericanBanking, TradeOnTheWire1		512

Input: AMC **Sort using:** Retweet count **Sort by:** Descending order **Sources:** EliteOptions2, OnlyGreenTrades, MarketRebels, canuck2usa, stockstobuy, SeekingAlpha, WarlusTrades, Ultra_Calls, TickerReport, MarketBeatCom, AmericanBanking, TradeOnTheWire1		673
Input: PLTR **Sort using:** Retweet count **Sort by:** Descending order **Sources:** EliteOptions2		207

Input: NIO **Sort using:** Retweet count **Sort by:** Ascending order **Sources:** EliteOptions2, Ultra_Calls, SeekingAlpha		237

Input: SPX **Sort using:** Retweet count **Sort by:** Descending order **Sources:** EliteOptions2, OnlyGreenTrades, MarketRebels, canuck2usa, stockstobuy, SeekingAlpha, WarlusTrades, Ultra_Calls, TickerReport, MarketBeatCom, AmericanBanking, TradeOnTheWire1		893

Question 3:

Explore some innovations for enhancing the indexing and ranking. Explain why they are important to solve specific problems, illustrated with examples. Possible innovations include (but are not limited to) the following:

- Enhanced search (e.g., add histograms, timelines, pie charts, or word clouds)
- Interactive search (e.g., refine search results based on users' relevance feedback)
- Geo-spatial search (e.g., use map information to refine query results and improve visualization)
- Multimodal search (e.g., implement image or video retrieval)
- Multilingual search (e.g., enable your system to retrieve data in multiple languages)
- Multifaceted search (e.g., visualize information according to different categories)

Sorting

- Sort by time
  This allows the user to sort the tweet displayed from latest tweets to oldest tweets if they choose ascending and vice versa. As timing is very important in the stock market, sorting the tweets according to time will help the user to get the latest tweets so that they can react immediately if necessary. One example is the user wants to search for the latest tweets regarding Tesla.

- Sort by favourite counts
  This allows the user to sort the tweet displayed from tweet with highest favourite counts to tweet with lowest favourite counts if they choose descending and vice versa. A tweet with a higher number of likes tends to imply that it is agreed by more people as compared to one with lesser likes. Therefore, users might want to find those tweets with a higher number of like counts as it seems to be more reliable. One example is the user wants to search for the tweets regarding Tesla with the highest number of favourite counts.

- Sort by retweet counts
  This allows the user to sort the tweet displayed from tweet with highest retweet counts to tweet with lowest retweet counts if they choose descending and vice versa. People usually retweet when they find a tweet valuable to share with others. The higher retweet counts tends to imply that the tweet is more valuable and useful. Therefore, users might want to find those tweets with a higher number of retweet counts as it seems to be more reliable. One example is the user wants to search for the tweets regarding Tesla with the highest number of retweet counts.

Spellchecking and suggestions

Solr has a built-in spellchecker component that can check whether the user's query is valid and spelled correctly or not. It also has a suggest component that provides the user with the suggested query that is correctly spelled. We can activate the spellchecker and suggester by

modifying the *solrconfig.xml* file. The spellcheck distance measure that we have selected is the internal levenshtein distance. In Figure 17 below, we can see that when a user incorrectly spells “goldman" as “goldmen”, the spellchecker suggests a few replacements, with the correct spelling “goldman” being the best suggestion, followed by “golden”, “goodman" and “goldin”. Figure 18 shows the suggestions displayed in the web application.


```
{
  "responseHeader":{
    "status":0,
    "QTime":10},
  "response":{"numFound":0,"start":0,"numFoundExact":true,"docs":[]
  },
  "spellcheck":{
    "suggestions":[
      "goldmen",{
        "numFound":4,
        "startOffset":10,
        "endOffset":17,
        "origFreq":0,
        "suggestion":[{
            "word":"goldman",
            "freq":71},
          {
            "word":"golden",
            "freq":28},
          {
            "word":"goodman",
            "freq":1},
          {
            "word":"goldin",
            "freq":1}]}],
    "correctlySpelled":false,
    "collations":[
      "collation",{
        "collationQuery":"tweettext:goldman",
        "hits":71,
        "misspellingsAndCorrections":[
          "goldmen","goldman"]},
      "collation",{
        "collationQuery":"tweettext:golden",
        "hits":28,
        "misspellingsAndCorrections":[
          "goldmen","golden"]},
      "collation",{
        "collationQuery":"tweettext:goodman",
        "hits":1,
        "misspellingsAndCorrections":[
          "goldmen","goodman"]},
      "collation",{
        "collationQuery":"tweettext:goldin",
        "hits":1,
        "misspellingsAndCorrections":[
          "goldmen","goldin"]}]}}
```


Figure 17. Spellchecker suggestions for misspelled query “goldmen” in JSON format

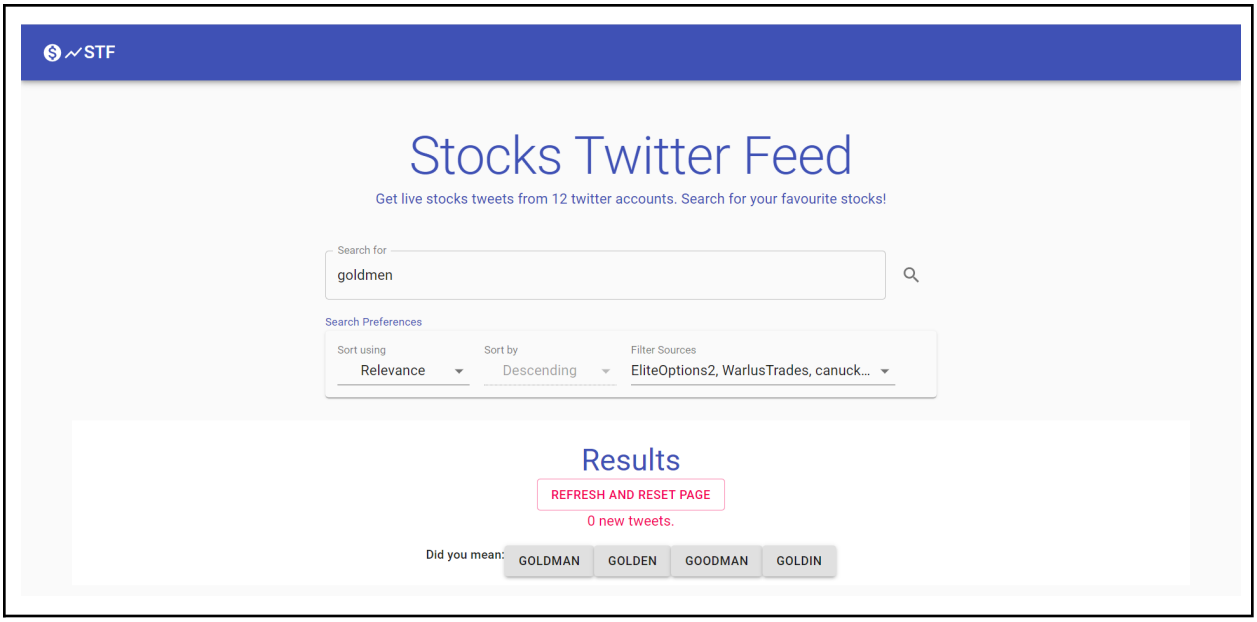


Figure 18. Spellchecker suggestion for misspelled query "goldmen" in web application

Character Mapping

As our system is purposed for querying financial and stock news and sentiments, we have to design our system to accept stock ticker symbols as query terms for specific stocks. For example, a user may query "tsla" or "$tsla" in order to search for Tesla company's stock. In order to map the stock ticker symbols to their corresponding stock name, we have utilised Solr's *MappingCharFilterFactory*.

The filter works by using an external file with all the mapping rules. A mapping rule is of the form "source" => "target". One example is "$tsla" => "tesla". In order to populate our mapping file, we have scraped all the stock ticker symbols and their corresponding stock name in the S&P 500. All the mapping rules can be found in the file *mapping.txt*. In addition to creating this file, we have also modified Solr's *managed-schema.xml* file to analyse queries based on the mapping rules.

```
" MMM "=>" 3M Company "
"$MMM"=>"3M Company"
" AOS "=>" A.O. Smith Corp "
"$AOS"=>"A.O. Smith Corp"
" ABT "=>" Abbott Laboratories "
"$ABT"=>"Abbott Laboratories"
```

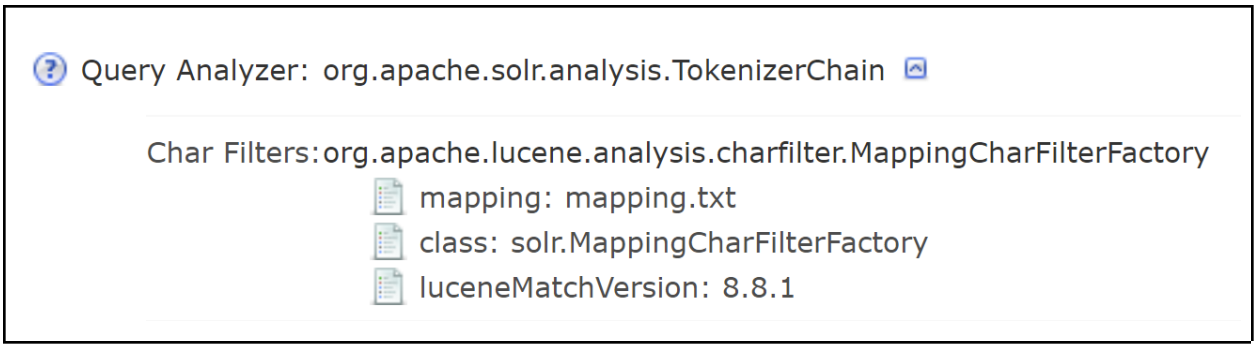


Figure 19. Example of mapping and screenshot of schema

Question 4: Classification

We decided to perform different classification models on the tweets to classify them into either bullish or bearish. The tweets collected contain either bullish, bearish and neutral sentiment. We manually labelled the data using these three categories, to classify the tweets as neutral vs opinionated. We further removed the “neutral” labelled tweets and focused on building the model using opinionated tweets only. We used Python *pandas, pyplot, numpy, nltk and sklearn* packages to perform data preprocessing and classification. Our final aim is to integrate the final classification model with our system to achieve on-the-fly classification. We have the trained model stored in *pickle* implemented in the system. When users access the home page, they are able to view the classified tweets. Once the page is updated or search being performed, the current tweets fetched will be classified immediately with the trained model stored in the system. The predicted class will be stored together with tweets itself in Solr.

1) **Motivate the choice of your classification approach in relation with the state of the art.**

The models we have chosen to perform classification are: *Logistic Regression, Decision Tree, Random Forest, Gradient Boosting, Linear SVC, Gaussian Naïve Bayes, K-Nearest Neighbours.*

Logistic Regression

Logistic regression is a linear model for classification. Its cost function is rather more complex than. Defined as the ‘sigmoid function’ or also known as the ‘logistic function’. The hypothesis of logistic regression tends it to limit the cost function between 0 and 1. In our model, we used *Randomized Search Cross Validation* to find the best estimators out of several solvers, penalties and C values.

Decision Tree

Decision Tree is used to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. It is able to handle both numerical and categorical data. The cost of using the tree in predicting data is logarithmic in the number of data points used to train the tree. The deeper the tree, the more complex the decision rules and the fitter the model. Similarly, we used a Randomized *Search CV* to determine the depth to be used. We also set some pre-pruning criterion to ensure the leaf nodes contain at least a fraction of the overall sum of the sample weights.

Random Forest

A random forest is a meta estimator that fits a number of decision tree classifiers on several sub-samples of the dataset. It uses averaging to improve the predictive accuracy and control over-fitting. Similar to Decision Tree Classifier, we also used *Randomized Search CV* to

determine the best maximum depth to be used. Some other parameters were also found along the way such as *bootstrap, criterion, min_samples_leaf*.

Gradient Boosting

Gradient Boosting builds an additive model in a forward stage-wise fashion. In each stage, *n_classes_* regression trees fit on the negative gradient of the binomial or multinomial deviance loss function. In this model, we used a Randomized *Search CV* to get the best learning rate as one of the parameters. Learning rate shrinks the contribution of each tree and there is a trade-off between *learning_rate* and *n_estimators.*

Linear SVC

The objective of a Linear SVC (Support Vector Classifier) is to fit the data and return a “best fit” hyperplane that divides or categorizes the data. After getting the hyperplane we can feed some features to the classifier to see what the “predicted” class is. It is effective in high dimensional spaces and highly versatile. However, it might not work well if the data contains significant non-linear relationships.

Gaussian Naïve Bayes

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes’ theorem with the “naive” assumption of conditional independence between every pair of features given the value of the class variable. In addition to the probabilities for each class, it also stores the mean and standard deviations for each input variable for each class. Predicted classes are calculated using the Gaussian Probability Density Function.

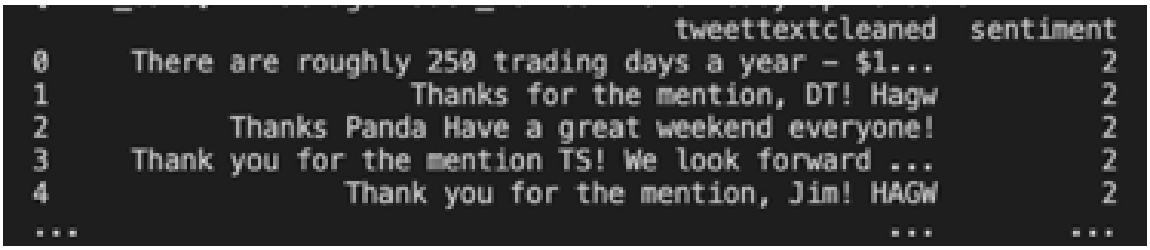
K-Nearest Neighbours

KNN (K-Nearest Neighbor) is a simple supervised classification algorithm we can use to assign a class to a new data point. KNN does not make any assumptions on the data distribution, hence it is non-parametric. It keeps all the training data to make future predictions by computing the similarity between an input sample and each training instance. It computes the distance between the new data point with every training data. The distance could be Euclidean distance, Hamming distance or Manhattan distance. The model will pick k entries in the data which are closest to the new data point. The most common class will finally be the class of the new data point.

2) **Discuss whether you had to preprocess data (e.g., microtext normalization) and why**

Data preprocessing is done on both the static data used to build the model and the dynamic data we retrieved from the twitter API to do the classification. We ensure the procedure of data preprocessing is similar for both sides of the data.

1. We took out the raw textual content of the tweets which is related to predict the sentiment and did preprocessing on it.
2. After getting all the tweets we need, we manually labelled 2000 tweets from all 10 sources with 200 tweets from each source. The types of labels include 0(bearish), 1(neutral), 2(bullish). We stored the data in csv files and passed it through the system for further processing.



	tweettextcleaned	sentiment
0	There are roughly 250 trading days a year - $1...	2
1	Thanks for the mention, DT! Hagw	2
2	Thanks Panda Have a great weekend everyone!	2
3	Thank you for the mention TS! We look forward ...	2
4	Thank you for the mention, Jim! HAGW	2
...	...	...

Figure 20: Training Data in Python Dataframe.

3. Convert to lowercase
   All the texts are converted into lower case to ensure similar words are mapped together as the same word. This helps to remove the redundancy caused by the differences between lowercase and uppercase. In our project, preserving capitalization is not important.
4. Remove non-alphabetical characters
   Punctuations and numbers are removed because they are less meaningful in this project.
5. Remove stop words
   Stop words are a set of commonly used words in a language. Examples of stop words in English are “the”, ”we” and etc. The intuition behind removing stop words is that, by removing low information words from the text, we can focus more on the important words instead of distinguishing text and classification.
6. Stemming
   Stemming is the process of making the words to their root form. The root form may not be a real root word but a canonical form of the original word. We use PorterStemmer() as Porter's Algorithm is known to be empirically effective in English.
7. Document-term matrix
   We used CountVectorizer() to convert all the documents into a document term matrix. It also enables the pre-processing of text data prior to generating the vector representation. This functionality makes it a highly flexible feature representation module for text.
8. *tf-idf* Matrix
   We applied *tf-idf* conversion to convert the document-term matrix into a *tf-idf* matrix which would be the final output of the data preprocessing step.

3) **Build an evaluation dataset by manually labeling 10% of the collected data (at least 1,000 records) with an inter-annotator agreement of at least 80%**

Two of our members manually labelled 2,000 tweets. The result of labelling is shown below:

	Bearish (0)	Neutral (1)	Bullish (2)	Total
Bearish (0)	755	17	31	803
Neutral (1)	23	701	11	735
Bullish (2)	8	9	445	462
Total	786	727	487	2000

Table 21: Comparison of labelled tweets.

We use Cohen's Kappa Measure to assess whether we have good inter-annotator agreement. According to the formula,

$$Kappa = \frac{[P(A) - P(E)]}{[1 - P(E)]}$$

$$P(A) = \frac{(755 + 701 + 445)}{2000} = 0.9505$$

$$P(E) = \frac{1}{2000^2}((786 * 803 + 727 * 735 + 487 * 462) = 0.3476$$

$$Kappa = \frac{0.9505 - 0.3476}{1 - 0.3476} \approx 0.9241$$

Since the Kappa Measure is above 0.8, the labelled data is considered to have a good inter-annotator agreement.

Before splitting into training and test data sets, we remove those neutral-sentiment documents for more accurate prediction of the sentiment whether it is bullish or bearish. We split our labelled tweets into training and testing data sets. 70% of all the labelled data was used to train the classifier while the remaining 30% was used to evaluate the model performance. The performance evaluation of different models trained is shown below.

Logistic Regression				
Class	**Precision**	**Recall**	**F-measure**	**Accuracy**

Bearish (0)	0.79	0.57	0.66	0.7804
Bullish (2)	0.78	0.91	0.84	
Weighted Average	0.78	0.78	0.77	

Table 22: Evaluation Result for Logistic Regression.

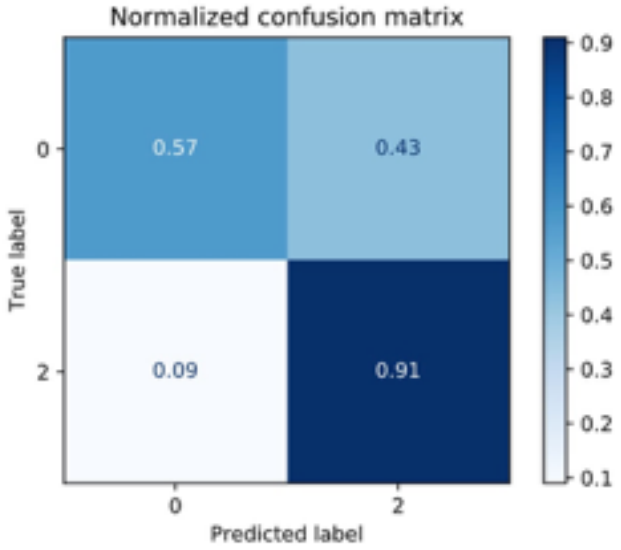


Figure 23: Confusion Matrix for Logistic Regression

Decision Tree				
Class	**Precision**	**Recall**	**F-measure**	**Accuracy**
Bearish (0)	0.75	0.57	0.65	0.7282
Bullish (2)	0.77	0.89	0.83	
Weighted Average	0.76	0.77	0.76	

Table 24: Evaluation Result for Decision Tree.

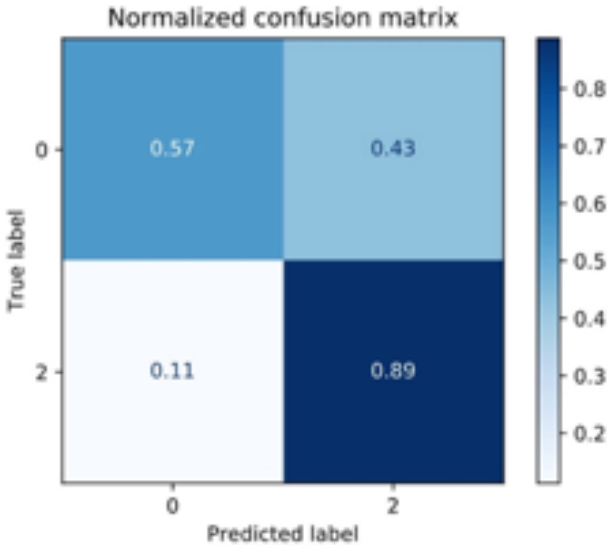


Figure 25: Confusion Matrix for Decision Tree

Random Forest				
Class	**Precision**	**Recall**	**F-measure**	**Accuracy**
Bearish (0)	0.84	0.59	0.70	0.8037
Bullish (2)	0.79	0.93	0.86	
Weighted Average	0.81	0.80	0.79	

Table 26: Evaluation Result for Random Forest.

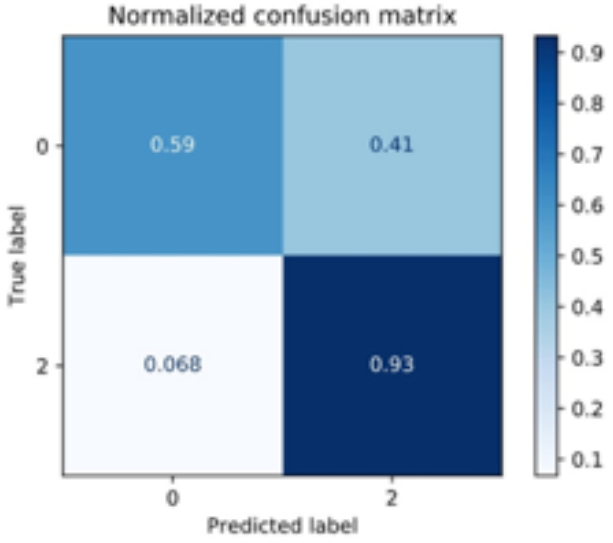


Figure 27: Confusion Matrix for Random Forest.

Gradient Boosting				
Class	**Precision**	**Recall**	**F-measure**	**Accuracy**
Bearish (0)	0.86	0.54	0.67	0.7944
Bullish (2)	0.77	0.95	0.85	
Weighted Average	0.81	0.79	0.78	

Table 28: Evaluation Result for Gradient Boosting.

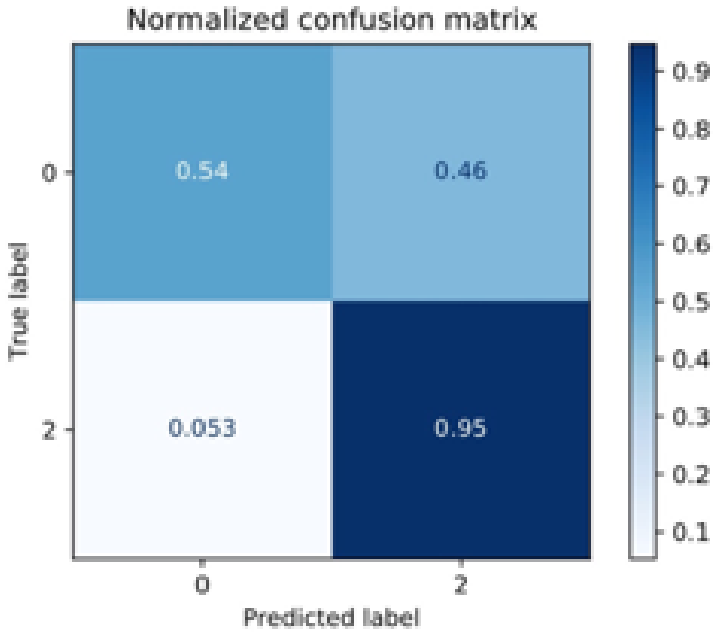


Figure 29: Confusion Matrix for Gradient Boosting.

K-Nearest Neighbours				
Class	**Precision**	**Recall**	**F-measure**	**Accuracy**
Bearish (0)	0.69	0.53	0.60	0.7336
Bullish (2)	0.75	0.86	0.80	
Weighted Average	0.73	0.73	0.72	

Table 30: Evaluation Result for K-Nearest Neighbours.

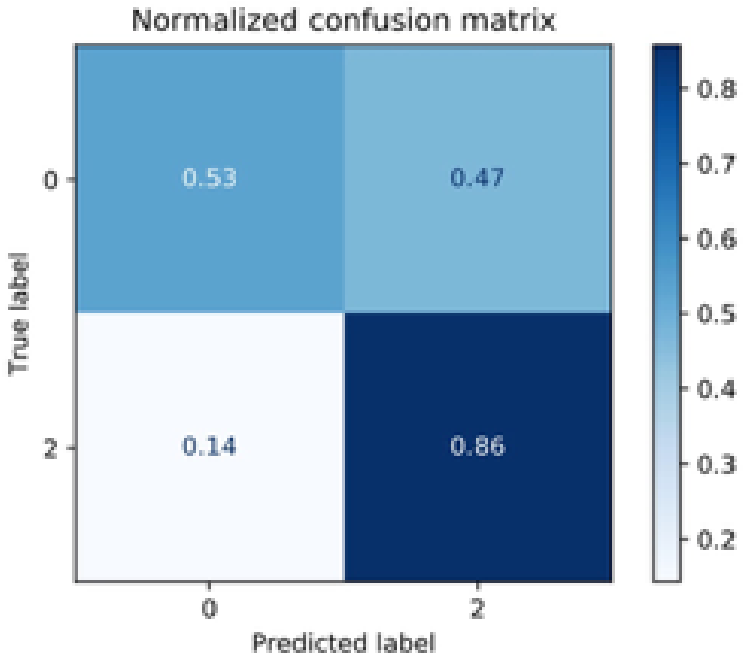


Figure 31: Confusion Matrix for K-Nearest Neighbours.

Linear SVC				
Class	**Precision**	**Recall**	**F-measure**	**Accuracy**
Bearish (0)	0.78	0.57	0.66	0.7757
Bullish (2)	0.77	0.90	0.83	
Weighted Average	0.78	0.78	0.77	

Table 32: Evaluation Result for Linear SVC.

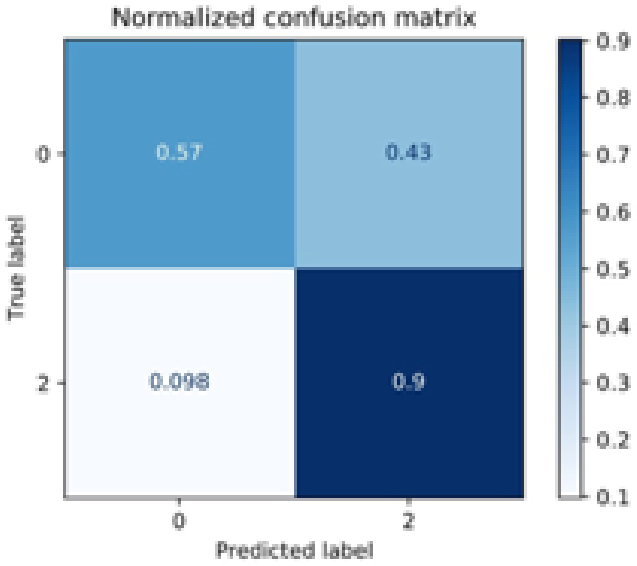


Figure 33: Confusion Matrix for Linear SVC.

Naïve Bayes				
Class	**Precision**	**Recall**	**F-measure**	**Accuracy**
Bearish (0)	0.50	0.85	0.63	0.6262
Bullish (2)	0.84	0.49	0.62	
Weighted Average	0.72	0.63	0.62	

Table 34: Evaluation Result for Naïve Bayes.

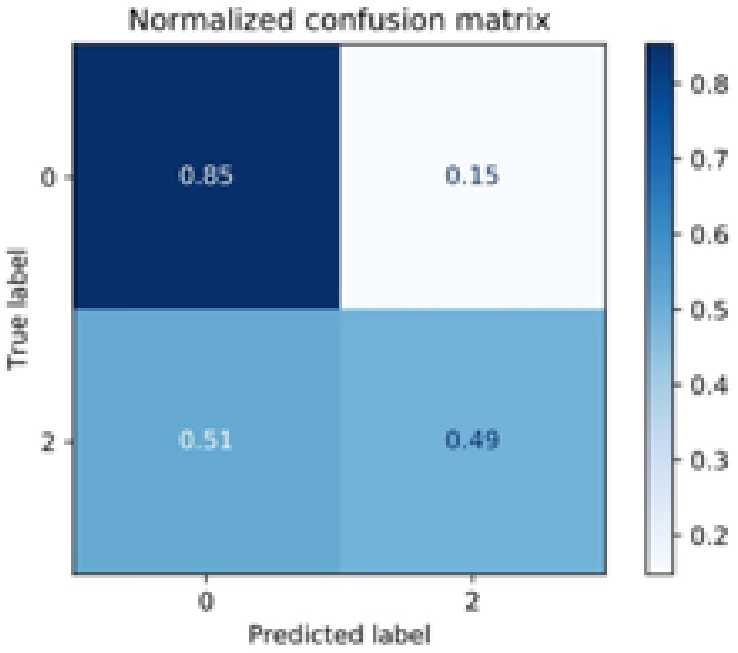


Figure 35: Confusion Matrix for Naïve Bayes.

Before discussing the result, we do a summary for the metrics data using the weighted average scores.

	Precision	Recall	F-measure	Accuracy
Logistic Regression	0.78	0.78	0.77	0.7804
Decision Tree	0.76	0.77	0.76	0.7282
Random Forest	0.81	0.80	0.79	0.8037
Gradient Boosting	0.81	0.79	0.78	0.7994
K-Nearest Neighbours	0.73	0.73	0.72	0.7336
Linear SVC	0.78	0.78	0.77	0.7757
Naïve Bayes	0.72	0.63	0.62	0.6202

Table 36: Summary of Evaluation Results for all Classifiers

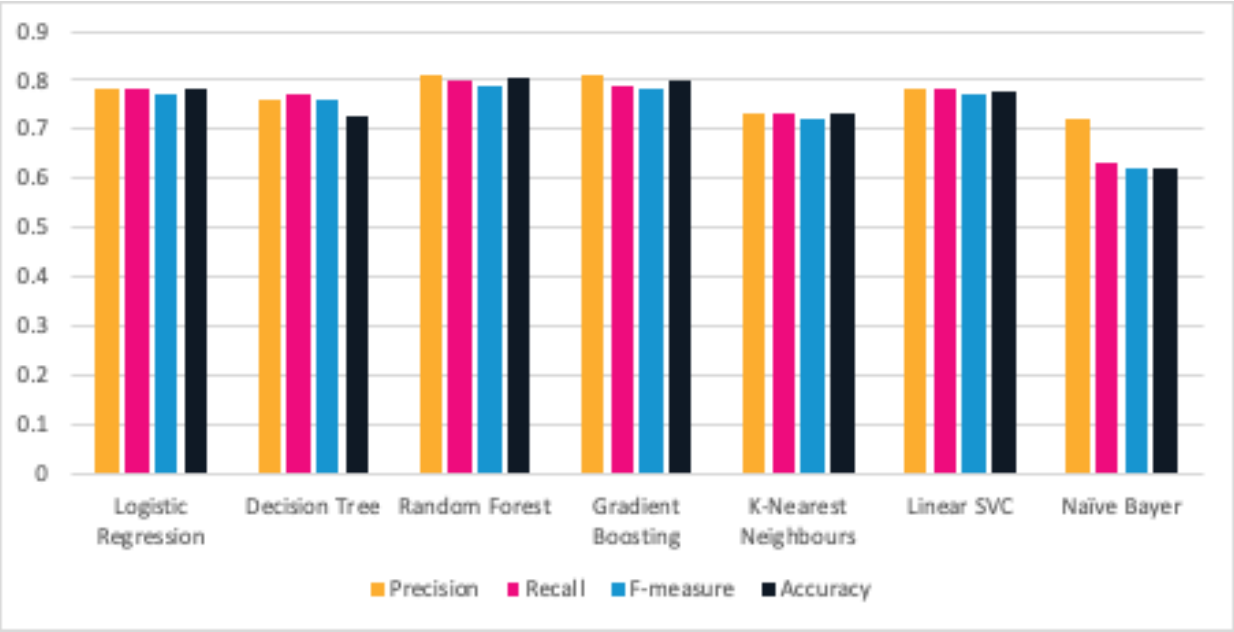


Figure 37: Summary of Model Comparison

Observation and Discussion:

1. Random Forest and Gradient Boosting Classification models have the highest precision.
2. Random Forest Classification model has the highest recall.
3. Random Forest Classification model has the highest F-measure.
4. Random Forest Classification model has the highest accuracy.
5. Most of the models have higher precision than recall except Decision Tree.

Question 5
Explore some innovations for enhancing classification.

To improve the accuracy of the classification model, we tried to apply the ensemble method for model stacking. Model stacking is an efficient ensemble method in which the predictions, generated by using various machine learning algorithms, are used as inputs in a second-layer learning algorithm. This second-layer algorithm is trained to optimally combine the model predictions to form a new set of predictions. The idea of constructing stacking model is shown below:

Figure 38: Construction of the predictive model by combining different models.

As shown in Figure x, the initial training data (X) has m observations and n features, used in training of M different models. The output of the level 1 models are then casted into second level training data which m x M. The M predictions have become features for this second level data. A second level model (or models) can then be trained on this data to produce the final outcomes which will be used for predictions. Base-models and meta-model need to be decided. Beta-model which is the level-0 models are the models fit the training model and whose predictions are compiled. Meta-model is the model that learns how to best combine the predictions of base models.

In our project, we selected logistic regression as our meta-model, while the rest of the classification models as base-models. After a round of training and comparing the accuracies, we removed linear SVC and K-nearest neighbours classifiers in the stacking model and we achieved a higher accuracy compared to using a single classification model alone in section 4. The comparison of the accuracy of the stacking model and other classification model done in section 4 is shown in Figure x. We can see that in all 4 measures which are precision, recall, F-measure and accuracy, the stacking model out-performed all other classification models.

Stacking Model				
Class	**Precision**	**Recall**	**F-measure**	**Accuracy**
Bearish (0)	0.85	0.62	0.71	0.8131
Bullish (2)	0.80	0.93	0.86	
Weighted Average	0.82	0.81	0.81	

Table 39: Evaluation Result for Stacking Model.

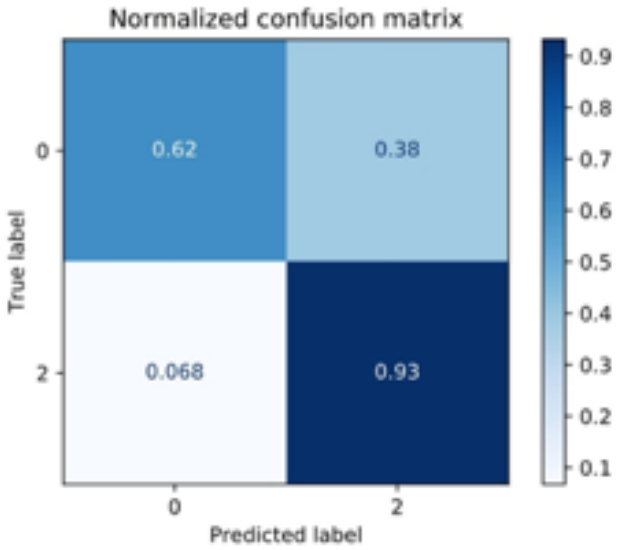


Figure 40: Confusion Matrix for Stacking Model.

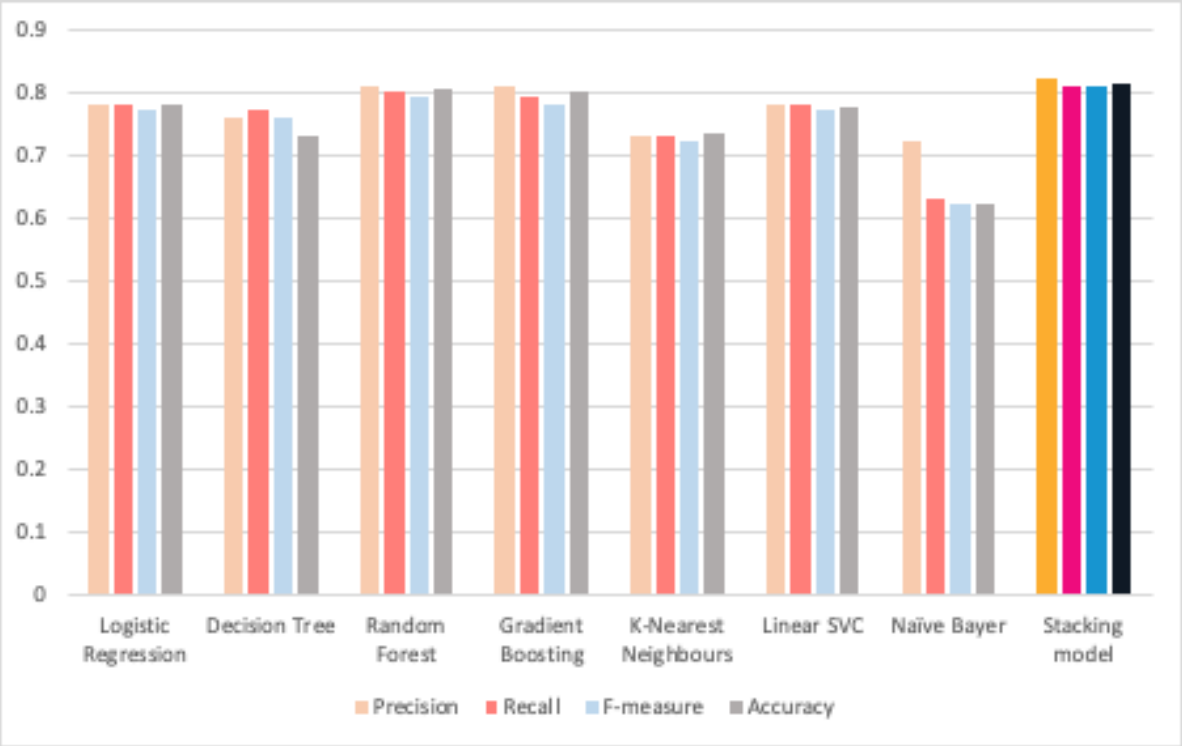


Figure 41: Comparison of stacking model with other classification models.