US Political Sentiment Analysis

Siem Berhane

[Analysis 3](#_Toc130812008)

[Introduction 3](#_Toc130812009)

[Research Into Existing Projects 5](#_Toc130812010)

[Sentiment Viz 5](#_Toc130812011)

[Understanding Political Twitter 6](#_Toc130812012)

[Sentiment Analysis Research 7](#_Toc130812013)

[Lexicon or Machine Learning Approach 7](#_Toc130812014)

[Vectorisation 7](#_Toc130812015)

[Machine Learning Algorithms 8](#_Toc130812016)

[UI 9](#_Toc130812017)

[Identification of User Needs 9](#_Toc130812018)

[Limitations 10](#_Toc130812019)

[Design 11](#_Toc130812020)

[Data Flow 11](#_Toc130812021)

[Class Diagrams 13](#_Toc130812022)

[Algorithms 13](#_Toc130812023)

[SentimentAnalysis 13](#_Toc130812024)

[CollectTweets 14](#_Toc130812025)

[GatherTweetData 15](#_Toc130812026)

[Resources Used 15](#_Toc130812027)

[UI Design 16](#_Toc130812028)

[Data Structures 17](#_Toc130812029)

[Code 25](#_Toc130812030)

[Hashing Algorithm 25](#_Toc130812031)

[Linked List and Node 26](#_Toc130812032)

[Sentiment Analysis Model 27](#_Toc130812033)

[Merge Sort Algorithm 29](#_Toc130812034)

[TwitterData 30](#_Toc130812035)

[DataProcessor 31](#_Toc130812036)

[TrainAlgorithm 33](#_Toc130812037)

[GatherTweetData 34](#_Toc130812038)

[CollectTweets 37](#_Toc130812039)

[TwitterDataStore 39](#_Toc130812040)

[CalculateTweetData 39](#_Toc130812041)

[MainWindow 41](#_Toc130812042)

[LayoutHandler 46](#_Toc130812043)

[MainClass 46](#_Toc130812044)

[Testing 49](#_Toc130812045)

[Evaluation 93](#_Toc130812046)

[User Feedback 93](#_Toc130812047)

[Objective Evaluation 93](#_Toc130812048)

[Next Steps 96](#_Toc130812049)

[New UI 97](#_Toc130812050)

Analysis

Introduction

Mention client that has asked to build this program for the same reason I have written here.

The invention of the internet has allowed more people to learn about events happening around the world than ever before. Social media has been instrumental in allowing the spread and discussion of news with it slowly overtaking more traditional news sources. A study conducted found that 71% of US adults aged 18-29 get their news online from places like websites, apps or social media[[1]](#footnote-1). It would be reasonable to say that huge sites like Twitter and Facebook are influencing the way people shape their views on the world.

People on the internet tend to form communities with people that share common interests. When it comes to politics, keeping to a single community could lead to being trapped in an echo chamber where people’s views aren’t challenged, and they are less likely to engage with different viewpoints. These sorts of communities can reinforce biases which could lead to members and the community slowly adopting more extreme views than what they initially started off with[[2]](#footnote-2).

I plan to build a desktop app that will gather tweets mentioning the keywords ‘republican’ or ‘democrat’ to see how the general Twitter user views the US parties. The program will analyse the tweets and predict users' opinion on whichever party is mentioned. This data will then be compared to the general public’s opinion to see how Twitter’s views compare to the everyday American.

Research Into Existing Projects

There are plenty of existing programs that can be used to gather the opinion of twitter users using a technique called *sentiment analysis*. Sentiment analysis determines if a piece of text is positive or negative and many businesses use it to monitor how the general public react to their products.

What I’ve come to realise is that most of the products available only present the data instead of actually doing something with it. There are a lot of different websites and tools that will scour the internet for keywords or topics and present huge amounts of data in various different ways but expect the users to draw their own conclusions from it. There are two products I would like to look at in more detail.

Sentiment Viz

Redo this, Brand24 isnt in this document – either add it or remove mentions of it

Sentiment Viz works in a similar way to Brand24. The user enters a keyword in both programs but the way the data is presented is quite different and it seems like both products are aimed at different end users. Brand24 seems to be aimed at businesses whereas Sentiment Viz seems to be more research based and more aimed at visualising Twitter users’ views on topics.

Sentiment Viz presents the data in many different ways. It plots the tweets on a graph with the pleasure on the *x* axis and how active the tweet is on the *y*. It displays the tweets as green or blue balls of varying sizes. The bigger the ball, the more confidence the algorithm has at predicting the emotion of it and the greener the ball is, the more positive the tweet is.

It also includes other features such as a word cloud and a map which is meant to show the location of certain tweets.

Understanding Political Twitter

Understanding Political Twitter[[3]](#footnote-3) is an article written by Duncan Grubbs and Meghan Mandi where they use sentiment analysis to get a better understanding of the political climate on Twitter around the world. They gathered tweets from 12 important political figures from around the world and calculated the average polarity and objectivity of each figure.

They analysed tweets from Donald Trump and compared the tweets that mention Fox News and CNN and plotted a graph from the data.

The graph tells us that most of the time Trump usually talks about Fox News in a neutral to positive light whereas he usually talks about CNN in a neutral to negative light. This data implies that Trump prefers Fox News to CNN.

Sentiment Viz is good for analysing the sentiment of Tweets but it isn’t focused on politics and is more of a general tool. The article that analyses the Tweets of political leaders has some interesting results, but it isn’t interested in the political views of the general public which is where my program will be focusing on.

There are a lot of academic papers being released to this day that use sentiment analysis to predict election results in different countries which shows that it is a relevant topic that people are still interested in. According to this paper written by Adam Bermingham and Alan F. Smeaton[[4]](#footnote-4), there are three ways people have used sentiment analysis to work with online data. Those are monitoring reactions to a certain event like a political debate, predicting things like stock market values by correlating public mood with socio-economic factors, and predicting future elections. A lot of the papers I looked at only predict future election results but don’t tend to compare the data they collected with current views to see if Twitter users hold different opinions to the general public which is what I plan to do with my program.

Sentiment Analysis Research

Lexicon or Machine Learning Approach

The two main ways sentiment analysis is carried out is by either a lexicon based approach or machine learning. The lexicon approach consists of creating a list of words that are likely to show up and give them a score based on the emotional tone. Words with a positive tone will be given a positive score and the stronger the emotional tone, the bigger the score. The same is done for negative words but they are given a negative score instead.

One advantage of this method is that complex algorithms won’t need to be implemented. However, I would have to either find or create a list of words each with individual set scores. Trying to find one online that matches up with what I want to do could prove difficult and creating my own list would be labour intensive. Another disadvantage is that all the words have the same score no matter the context. The context of the text can completely flip the emotional tone of certain words which a lexicon based approach doesn’t consider. These are some of the reasons I have decided to take the machine learning approach.

Using a machine learning algorithm would involve gathering data and then transforming said data into vectors so the algorithm can use it to predict its polarity. The algorithm needs to be trained with a dataset. This approach solves all the issues with the lexicon method since finding a dataset of tweets to do with US politics will be a lot easier and the methods to transform the words into vectors account for context.

The main disadvantages for this method is the fact that algorithms don’t tend to perform well when fed data that contains emojis and punctuation. To get around this, I will remove all the extra noise that comes with the tweets so all that is left is just text.

Vectorisation

The two methods I researched to transform text into vectors is the bag of words method (count vectorizer) and the TF-IDF vectorizer.

The bag of words method creates a list of all the unique words in the whole text document, and records how many times each word appears in each sentence. This list is then fed into the algorithm.

The TF-IDF vectorizer calculates a value for each word which depends on two factors. The frequency of the word in a text and the number of texts in the whole dataset that contain that word. It is proportional to the first factor and inversely proportional to the second. To calculate the TF-IDF score, the Term Frequency (TF) and the Inverse Document Frequency (IDF) need to be calculated first.

Term Frequency

TF is the frequency of which a word appears in a document and is calculated using this formula:

Where is the frequency of the term t in document d

Inverse Document Frequency

The IDF measures the frequency of a word in the whole set of data. The words that have a low frequency will give a higher IDF value. It is calculate using this formula:

Where |D| is the total number of documents in the whole set and is the number of documents in the dataset which contain the word t. The greater the frequency of a term in the whole dataset, the closer the fraction is to 1 which means the idf value is closer to 0.

The TF-IDF value is calculated by finding the product of TF and IDF:

Since TF-IDF accounts for the frequency across the whole dataset, it will give lower scores to common words such as *‘if’*, *‘and’*, *‘but’* etc. whereas the bag of words method would give those words high scores since they are so frequent even though they provide little emotional substance. TF-IDF transforms words into vectors based on how important they are whereas bag of words only cares about frequency. For this reason, I will implement the TF-IDF vectorizer.

Machine Learning Algorithms

There are multiple algorithms that can be used for sentiment analysis but the main ones that I considered were Naive Bayes, Logistic Regression and Support Vector Machines (SVM). Ultimately, I decided on using Naive Bayes since it is a simple yet effective algorithm for what I plan to do.

The Naive Bayes algorithm uses Bayes theorem which is:

Where is the conditional probability of A given B and is the conditional probability of B given A. It is called naive since it considers each input as independent from the others.

The algorithm needs to be trained by using a suitable dataset before it can be used to predict the polarity of text. I’ll be using a dataset called Sentiment140 which contains 1.6 million tweets extracted using the Twitter API. The dataset has given each tweet a polarity score between 0 (negative) and 4 (positive) and also contains extra data like the id of the tweet, the date it was tweeted, the user that tweeted it and more. For my program, I’ve reduced the size from 1.6 million to 100,000 tweets.

I’ll be using the pandas library to manipulate the dataset and the scikit-learn framework to implement the TF-IDF vectorizer and the Naive Bayes algorithm. Twitter has an API that can be used for collecting tweets.

UI

There are many python libraries and frameworks that can be used to add graphics to projects including pygame, Tkinter and PyQt. Tkinter tends to be better suited for smaller projects and pygame is better suited for games instead of apps. PyQt’s main use is desktop apps which my project will be so I have decided to use that.

Survey + Interview with client

Create both – analyse both

**Survey**

explain the purpose of each question + analyse results

**interview**

analyse results and how this might impact the scope of the project like how ideas change bc of it

Identification of User Needs

1. Collect tweets that mention the keywords *democrat/democrats* or *republican/republicans*
2. Predict the polarity of each tweet and gather data to do with each tweet and twitter user
3. Allow the user to specify the types of data they would like to accompany the tweets
4. Display all the tweets collected to the user along with all the data gathered
5. The user should be able to choose to view data regarding the whole dataset of tweets or just individual tweets
6. The user should be able to filter out the collected tweets based on certain factors like polarity or which party the tweet is about and view data regarding those tweets as a whole
7. The user should be able to view the tweets in ascending/descending order based on different factors such as the data tweeted or the number of followers each user has and view data regarding those tweets as a whole
8. The user should be able to compare the collected data to current real-world data to see if the opinions on Twitter are different to real life
9. The user should be able to compare the collected data to previous real-world data such as previous elections

Objectives

Go through each IUN and break them up into smaller objectives. Split it up as much as possible – into every step needed to achieve it. E.g. objective 1 – look at how the code works for inspiration to split it up

Add UI objectives too

1. Collect tweets that mention the keywords *democrat/democrats* or *republican/republicans*
   1. Authorise Twitter API access
   2. Set out queries stating what is being looked for
   3. Specify which pieces of data to be returned
   4. Organise the data so that all the data for each user-tweet combination is in one place
2. Predict the polarity of each tweet and gather data to do with each tweet and twitter user
   1. Talk about creating a suitable model – training, testing, gathering test + training data, the .csv file
3. Allow the user to specify the types of data they would like to accompany the tweets
   1. Add a text prompt before the program runs asking the user to choose the type of data to be returned
   2. The user should be able to select from a set of data types – drop box maybe? Will need to update UI design
   3. Store the input
   4. When searching for tweets, make sure user specified data is returned
4. Display all the tweets collected to the user along with all the data gathered

Data Flow

This shows how the flow of data as the program is executed

Search for Tweets

Find location if available

The polarity is predicted

Get rid of unnecessary characters

Return specified data

The data is stored in relevant lists e.g. all positive tweets stored in a list etc.

The data is stored in a class

Data from .csv file is imported

Program starts

Export data to .csv file

Calculate information to do with data set like mean number of followers of users etc.

Display everything

Store the data in a hash table

Limitations

1. The Naive Bayes model won’t be 100% accurate so some tweets will be misunderstood. This will be especially prevalent with Tweets that include sarcasm
2. The access level my Twitter developer account has only allows the most recent tweets to be searched and can only
3. The version of Nominatim the program will be using only allows up to around 300 requests at a time

Design

Modular Structure of Program

This shows how each of the main aspects of the program will work

Keep this

Program

Machine Learning Models

Save the models

Get rid of unnecessary data

Train the TF-IDF vectorizer

Train the Naïve Bayes Model

Train Models

Get training & testing data

Read data from dataset csv file

Prepare Data

Getting Tweets

Storing

Store all the data in a class

Specify types of data to be returned from search

Hash table

Lemmatize

Processing Text

De-contract

Remove unnecessary characters

Extra Data

Location

Polarity

Processing Tweets

Searching

Store it in a suitable form

Authenticate developer account

Create queries

Store in separate lists based on certain factors

Class Diagrams

Maybe redo this so that it is clearly readable

|  |
| --- |
| **TwitterData** |
| text  cleanText  sentiment  location  username  lang  repub  metrics  userMetrics  key |
| \_\_getRepub(repub)  \_\_getSentiment(sent)  \_\_metricsDict(metrics)  \_\_setDate(timeData) |

Algorithms

Keep descriptive paragraph for each algorithm but change bullet points to pseudocode

Explain in the paragraph how it achieves different objectives

SentimentAnalysis

To predict the polarity of text using the Naive Bayes machine learning algorithm, a model of it will need to be trained using pre-existing data. The text in the dataset I will be using has already been cleaned up so there is no need to include any of that in the training algorithm.

1. Read data from the dataset
   1. The dataset is stored as a .csv file so the pandas library will be used to manipulate the data
   2. The dataset contains lots of unnecessary data that is not needed to train the models which will be removed
2. Store the text and polarity in separate lists so that the index of each piece of text is the same as the index of its polarity
3. Split the text and polarity lists into testing and training lists where the first 80% of the elements go into the training list and the other 20% into the testing lists
4. Using scikit-learn, create a TF-IDF vectorizer model using the training text list and save it
5. Use the model to transform the test-text list
6. Train the Naive Bayes model by inputting the vectorised training text data alongside the training polarity data and save the model
7. Use the trained model to predict the polarity of the testing text list
8. Calculate the accuracy of the model
9. Once the accuracy is of a satisfactory level, save it using the pickle library

CollectTweets

The Twitter API is what I will be using to get the tweets the program needs. I will be searching for the most recent tweets that talk about the Democrats or Republicans. To do so, the API needs a query that has all the factors needed to get the most relevant tweets.

1. Authorise the API client by inputting my account’s bearer token
2. Create separate queries to find Democrat and Republican tweets
   1. The tweets will only mention either the Democrats or Republicans, not both
   2. All the tweets must be in English
   3. It will exclude retweets
3. Search for tweets mentioning the Democrats and store them in a list. As well as the text, the API will also return the following data:
   1. The time it was tweeted
   2. The tweet id
   3. The tweet metrics (number of likes, number of retweets etc.)
   4. The username
   5. The location of the user (if available)
   6. The public metrics (number of followers, the number of tweets the account has tweeted etc.)
4. Search for tweets mentioning the Republicans and do the same thing
5. Store the data for each tweet in a dictionary and then store all dictionaries in a list
   1. Data to do with the user who published the Tweet and data regarding the tweet itself will have to be stored in separate dictionaries. This is because different methods must be used to access user data compared to tweet data

GatherTweetData

This algorithm will handle the tweets and the data gathered from it by the previous algorithm. It will use multiple APIs and libraries with the main ones being Python’s regular expression library, the Nominatim API which is used when dealing with locations, and the spacy library which is used in natural language processing.

1. Receive the two dictionaries for the Republicans and Democrats containing all the data gathered from the previous algorithm
2. Combine the two dictionaries for each party into one for each
3. Clean the text
   1. Removes links, hashtags, possessive apostrophes and all characters that aren’t in the English alphabet
   2. De-contracts any contracted words e.g., *we’ve* becomes *we have*
   3. Each word is lemmatized and converted to its base word. E.g., the words *runs, running* and *ran* would all be converted to the word *run*.
4. Use the Naive Bayes model to predict the polarity of each tweet
5. If the user has their location set in the US, find the state they’re located in using Nominatim
6. For each individual tweet, store the data in a class and add that class into a list so that all the data for each tweet can be accessed

Resources Used

**PyQt**

Python binding for the C++ framework. It comes with features that will help in the creation of the UI. Some of the features that will be used are the buttons, labels and various layouts that organise the different UI elements in specific ways such as grids or vertically.

**Geopy**

Geopy is a python client that gives the user access to many geocoding services such as Google and Bing Maps by abstracting their APIs. The one I’ll be using will be Nominatim. Nominatim provides a geocoding service which can look up locations based on textual inputs such as the location provided by a Twitter user.

**Pandas**

Pandas is a python library that is used when working with large sets of data. It can analyse, clean, export and manipulate data. I will be using it to prepare data from the initial dataset by removing unnecessary columns and empty spaces.

**spaCy**

spaCy is a Python library that has lots of natural language processing (NLP) capabilities. It comes with features such as tokenization and part-of-speech tagging. The main feature I will be using will be lemmatization which finds the base form of words.

**Tweepy**

Tweepy is a python library that gives the user access to the Twitter API. I will be using Tweepy to search for tweets that mention the Democrats and Republicans and find data associated with the tweet and user.

**Sklearn**

Sklearn, also known as sci-kit learn, is a python library that contains many implantations of different machine learning algorithms, including the one I will be using. it is designed to be user friendly and is widely used in industry and academia.

**Pickle**

Pickle is a python module that allows the user to serialize and deserialize Python objects. I will use it to store the trained machine learning module so it can be re used without having to train a new model every time.

UI Design

What will be displayed when the program is run

The username will be a button. Pressing it will change the general data to the specific data about that tweet and user

The refresh button will cause the display to go back to what it was like when it was first run

The *Filters* and *Sort* sections will be drop down buttons

Data Structures

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Data Type** | **Example/Explanation** | **Class Found In** |
| table | Array | The hash table | HashTable |
| len | Integer | The length of the hash table | HashTable |
| twitterObj | Node | Stores all the data to do with the tweets along with a pointer to the next node in the linked list | HashTable |
| index | Integer | The index the object will be stored in the hash table | HashTable |
| node | Node | Stores the node found in the linked list | HashTable |
| lList | LinkedList | Stores the linked list stored in the index | HashTable |
| lastNode | Node | The last node found in a linked list | HashTable |
| key | Integer | The hashing key | HashTable |
| obj | TwitterData | The tweet data | HashTable |
| lastNode | Node | The final node in a linked list | HashTable |
| data | TwitterData | The tweet data | Node |
| next | Node | The node it points to | Node |
| head | Node | The head of the linked list | LinkedList |
| lastNode | Node | The last node of the linked list | LinkedList |
| endOfList | Bool | Returns true once the loop reaches the end of the linked list | LinkedList |
| temp | Node | Stores the node that is currently being compared | LinkedList |
| nodeFound | Bool | Returns true if the specified node has been found | LinkedList |
| C **MAYBE GET RID** | Integer | Counter used to test linked list traversal | LinkedList |
| id | integer | The id of the node that is being searched for | LinkedList |
| \_\_TRAINPERCENTAGE | Float | The percentage of the initial list that will become the training list. The remaining data will become the testing list | SentimentAnalysis |
| text | List | Stores the tweet text | SentimentAnalysis |
| sentiment | List | Stores the polarity of the tweet | SentimentAnalysis |
| NB | MultinomialNB | Scikit-learn’s Naïve Bayes algorithm | SentimentAnalysis |
| textData | List | Stores the tweet text | SentimentAnalysis |
| sentimentData | List | Stores the polarity of the tweet | SentimentAnalysis |
| splitTextData | Tuple | Stores the training and testing datasets of the texts | SentimentAnalysis |
| splitSentimentData | Tuple | Stores the training and testing datasets of the polarity | SentimentAnalysis |
| trainText | List | The text that will be used to train the ML algorithm | SentimentAnalysis |
| testText | List | The text that will be used to test the ML algorithm | SentimentAnalysis |
| trainSentiment | List | The polarities that will be used to train the ML algorithm | SentimentAnalysis |
| testSentiment | List | The polarities that will be used to test the ML algorithm | SentimentAnalysis |
| vectText | Tuple | The vectorized training and testing text | SentimentAnalysis |
| vectTrain | Tuple | The vectorized training text | SentimentAnalysis |
| vectText | Tuple | The vectorized testing text | SentimentAnalysis |
| sentiment | numpy.ndarray | The polarity of the text predicted by the trained model | SentimentAnalysis  TwitterData |
| vectData | Tuple | The vectorized text of the tweet | SentimentAnalysis |
| model | MultinomialNB | The Naïve Bayes model | SentimentAnalysis |
| vect | TFidfVectorizer | Scikit-learn’s TF-IDF vectorizer | SentimentAnalysis |
| splitData | Integer | The index the data will be split at to create the training and testing | SentimentAnalysis |
| modelPath | string | The file path of the Naïve Bayes model | SentimentAnalysis,  GatherTweetData |
| vectModelPath | string | The file path of the vectorizer model | SentimentAnalysis,  GatherTweetData |
| file | string | The file path that the pickle functions will use to save/load from | SentimentAnalysis |
| tweetData | List | The list of tweet data | SentimentAnalysis |
| data | List | The data to be the models will use to predict polarity | SentimentAnalysis |
| items | List | The list of items to be sorted | Algorithms |
| asc | Bool | Is true when the list is to be sorted in ascending order | Algorithms |
| midpoint | Integer | The midpoint of the list | Algorithms |
| leftHalf | List | The items of the list up to and including the midpoint | Algorithms |
| rightHalf | List | The items of the list after the midpoint | Algorithms |
| mergedItems, merged | List | Merge the ordered left and right sides of the initial list | Algorithms |
| text | String | The tweet text | TwitterData |
| cleanText | String | The cleaned text | TwitterData,  GatherTweetData |
| sentiment | String | The sentiment/polarity of the text | TwitterData,  GatherTweetData |
| location, loc | String | The state of the user | TwitterData,  GatherTweetData |
| username | String | The username of the user | TwitterData,  GatherTweetData |
| lang | String | The language of the tweet | TwitterData,  GatherTweetData |
| repub, isRepub | Bool | Returns true if the tweet mentions the Republican party | TwitterData |
| metrics, tweetMetrics | Dictionary | Data such as how many likes, retweets etc each tweet has | TwitterData,  GatherTweetData |
| userMetrics | Dictionary | Data such as how many followers the user has, tweet count etc. | TwitterData,  GatherTweetData |
| key | Integer | The last 5 digits of the id to be used as the hashing key | TwitterData,  GatherTweetData |
| newSent | String | Stores the properly formatted sentiment value imported from the sentiment field in the .csv file | TwitterData |
| lst | List | A list of the properly formatted values from the *user\_metrics* field | TwitterData |
| metrics | String | The value stored in the *user\_metrics* field in the .csv file | TwitterData |
| part | String | Each element stored in *metrics* | TwitterData |
| newPart | String | Stores the properly formatted version of *part* | TwitterData |
| metricsDict | Dictionary | Stores the dictionary version of *lst*. The odd elements are the keys and the even ones are the values | TwitterData |
| timeData | String | The time and date the tweet was published | TwitterData |
| time | String | The time the tweet was published | TwitterData |
| date | String | The date the tweet was published | TwitterData |
| \_\_nlp | Spacy | Importing spacy library so text can be lemmatized | DataProcessor |
| analysis | SentimentAnalysis | Instance of the SentimentAnalysis class | DataProcessor |
| twitterObject | TwitterData | Instance of the TwitterData class where only the sentiment and text are stored | DataProcessor |
| sentiment | String | The sentiment of the tweet imported from the .csv file | DataProcessor |
| text | String | The text from the tweet imported from the .csv file | DataProcessor |
| temp | String | Temporarily stores the text to be cleaned | DataProcessor |
| text | String | Stores the text to be de-contracted | DataProcessor |
| doc | String | Stores the text to be lemmatized | DataProcessor |
| finalDoc | String | Stores the lemmatized text | DataProcessor |
| text | String | The text to be cleaned | DataProcessor |
| filepath | String | The file path of the dataset | TrainAlgorithm |
| tweetData | List | List of the instances of TwitterData class created for all tweets | TrainAlgorithm |
| rawTweets | Dataframe | The contents of the .csv file | TrainAlgorithm |
| tweets | Dataframe | *rawTweets* but all the unnecessary columns have been removed | TrainAlgorithm |
| newText | List | The cleaned text for each tweet | TrainAlgorithm |
| index | Integer | The row number in the dataframe | TrainAlgorithm |
| tweetSenti | String | The sentiment of the tweet found at row number *index* | TrainAlgorithm |
| TweetText | String | The cleaned text of the tweet found at row number *index* | TrainAlgorithm |
| modelPath | String | The file path the machine learning model will be stored at | TrainAlgorithm,  GatherTweetData |
| vectModelPath | String | The file path the vectorizer model will be stored at | TrainAlgorithm,  GatherTweetData |
| score | Float | The sentiment imported from the .csv file. It is given as a number | TrainAlgorithm |
| sentiment | String | The sentiment of the tweet based on what was imported from the .csv file | TrainAlgorithm |
| lemmatise | Bool | Returns true if the text is to be lemmatized | DataProcessor,  GatherTweetData,  TrainAlgorithm |
| \_\_maxTweets  \_\_MAX\_TWEETS | Integer | Maximum number of tweets to be searched | GatherTweetData  CollectTweets |
| \_\_geoLocator | Nominatim | Instance of the Nominatim API to search locations | GatherTweetData |
| \_\_MAX\_LOCATION\_COUNT | Integer | Maximum number of locations to be searched | GatherTweetData |
| \_\_tweetCollector | CollectTweets | Instance of *CollectTweets* class | GatherTweetData |
| repubData | Tuple | All the Republican tweets and their corresponding data | GatherTweetData |
| demData | Tuple | All the Democrat tweets and their corresponding data | GatherTweetData |
| repubDict | List | A list of dictionaries containing the data of r*epubData* | GatherTweetData |
| demDict | Dictionary | A list of dictionaries containing the data of *demData* | GatherTweetData |
| object | Dictionary | The elements of *demDict* and *repubDict* | GatherTweetData |
| df | Dataframe | Dataframe combining *repubDict* and *demDict* | GatherTweetData |
| repubObjList | List | List of TwitterData instances for Republican tweets | GatherTweetData |
| demObjList | List | List of TwitterData instances for Democratic tweets | GatherTweetData |
| tweetObjList | List | Combination of repubObjList and demObjList | GatherTweetData |
| reader | OrderedDict | Reads the .csv file | GatherTweetData |
| lst | List | List version of the *reader* variable | GatherTweetData |
| object | Dictionary | Elements of the *lst* variable | GatherTweetData |
| twitterList | List | List of dictionaries containg all the tweet data | GatherTweetData |
| tweet | Dictionary | Elements of *twitterList* | GatherTweetData |
| tempTextList | List | Stores the text of the tweet | GatherTweetData |
| cleanText | String | The clean text of the tweet | GatherTweetData |
| data | dictionary | Dictionary containing all the Twitter data for a tweet | GatherTweetData |
| sentiment | String | Returns true if the tweet is about Republicans | GatherTweetData |
| country | String | The country of the user | GatherTweetData |
| location | JSON | All the location details of the Twitter user | GatherTweetData |
| state, tweetState | String | The state of the user | GatherTweetData |
| twitterDataList,  tweetDataList,  userDataList | List | Stores a list of dictionaries | GatherTweetData,  CollectTweets,  CollectTweets |
| \_\_client | tweepy.Client | The Twitter API client | CollectTweets |
| \_\_REPUBLICAN\_QUERY | String | Query to find Republican tweets | CollectTweets |
| \_\_DEMOCRAT\_QUERY | String | Query to find Democrat tweets | CollectTweets |
| repubTweets,  demTweets, | JSON | The tweets and data the query returned | CollectTweets |
| userDataList | List | List of dictionaries that contain the data collected from each tweet about the user | CollectTweets |
| user | Dictionary | Element of the *userJSON*  variable | CollectTweets |
| userData | Dictionary | Stores the data from *repubTweets* and *demTweets* concerning the user | CollectTweets |
| userJSON | JSON | Part of the JSON file that contains data about the user | CollectTweets |
| tweetJSON | JSON | Part of the JSON file that contains data about the tweet | CollectTweets |
| tweetDataList | List | List of Dictionaries that contain the data collected about the tweet | CollectTweets |
| tweetData | Dictionary | Stores the data from *repubTweets* and *demTweets* concerning the tweet | CollectTweets |
| tweet | List | Element of the *tweetJSON* variable | CollectTweets |
| strId | String | String version of the tweet ID | CollectTweets |
| key | String | Last 5 characters of *strId* | CollectTweets |
| intKey | Integer | Integer version of *key* | CollectTweets |
| **start here** |  |  |  |
| completelLIst | List | List containing every tweet searched | TwitterDataStore |
| repubTweetsLIst | List | List containing every tweet mentioning the Republicans | TwitterDataStore |
| demTweetList | List | List containing every tweet mentioning Democrats | TwitterDataStore |
| posList | List | List containing every tweet with a positive polarity | TwitterDataStore |
| negList | list | List containing every tweet with a negative polarity | TwitterDataStore |
| app | QApplication | Needed to start event loop for display | MainClass |
| sorting | Algorithms | Instance of the *Algorithms* class | MainClass,  CalculateTweetData |
| tds | TwitterDataStore | Instance of *TwitterDataStore* class | MainClass |
| gtd | GatherTweetData | Instance of *GatherTweetData* class | MainClass |
| tweetData, calcData | CalculateTweetData | Instance of the *CalculateTweetData* class | MainClass,  MainWindow |
| currentList | Enum | The list that is currently being displayed | MainClass |
| hashTable | Hash Table | A hash table | MainClass |
| tweets | List | List of all the *TwitterData* objects for each tweet searched | MainClass |
| object, obj | TwitterData | Element of *tweets* | MainClass |
| window | QMainWindow | The window that will be displayed | MainClass |
| list | List | List of items to be hashed | MainClass |
| item | TwitterData | Element of *list* | MainClass |
| key | Integer | Hashing key | MainClass |
| index | Integer | Integer that maps to each value in *currentList* | MainClass |
| tweetsList, cList | List | The list that is currently displayed | MainClass,  MainWindow, |
| tweetDict | Dictionary | Maps each user’s follower count to their id for easy access | MainClass |
| sortedList | List | The sorted list of tweets | MainClass |
| tempSortedList | List | List of sorted values | MainClass |
| sortedTweets | List | The *TwitterData* objects for each element of *tempSortedList* | MainClass |
| posTweets, num | Integer | Number of positive tweets | CalculateTweetData |
| negTweets | Integer | Number of negative tweets | CalculateTweetData |
| repubTweets, num | Integer | Number of Republican tweets | CalculateTweetData |
| demTweets | Integer | Number of Democrat tweets | CalculateTweetData |
| posPerc | Float | Percentage of tweets that are positive | CalculateTweetData |
| negPerc | Float | Percentage of tweets that are negative | CalculateTweetData |
| demPerc | Float | Percentage of tweets that mention the democrats | CalculateTweetData |
| repubPerc | Float | Percentage of tweets that mention the republicans | CalculateTweetData |
| meanFollowers | Integer | Mean number of followers a user has | CalculateTweetData |
| medianFollowers | Integer | Median number of followers a user has | CalculateTweetData |
| maxFollowers | Integer | Max number of followers | CalculateTweetData |
| minFollowers | Integer | Min number of followers | CalculateTweetData |
| data | Dictionary | All the data to be displayed | CalculateTweetData |
| followersList | List | List of the user’s follower count | CalculateTweetData |
| lst | List | List of all the *TwitterData* objects | CalculateTweetData |
| obj | TwitterData | Element of *lst* | CalculateTweetData |
| tweets | List | List of tweets to be displayed | MainWindow |
| mainClass | MainClass | Instance of the *MainClass* object | MainWindow |
| boldFont | QFont | Font applied to buttons that will be displayed | MainWindow |
| buttons | QButtonGroup | Stores all the buttons that display tweet data and maps them to the corresponding user’s ID | MainWindow |
| widgetDict | Dictionary | Stores the *stackWidget* objects with the ID of the corresponding user being the value of the dictionary | MainWindow |
| stackWidget | QStackedWidget | List of all the data that can be displayed about the general dataset and individual users | MainWindow |
| tweetsWidget | QWidget | Block that stores all the tweet widgets to be displayed | MainWindow |
| tweetLayout, vertLayout | QVBoxLayout | Layout that causes all the items add to it to be orgainsed vertically | MainWindow |
| tweet | TwitterData | Element of *tweets* | MainWindow |
| tweetContainer | QWidget | Contains the username and tweet text | MainWindow |
| tweetContainerLayout | QVBoxLayout | Causes the username widget to be vertically above the text widget | MainWindow |
| username | QPushButton | Button that displays the data regarding the specific tweet. The text of the button is the user’s username | MainWindow |
| tweetLable | QLabel | The tweet text | MainWindow |
| scroll | QScrollArea | Allows the user to scroll through the data | MainWindow |
| gridLayout | QGridLayout | Organises the widgets in a grid layout | MainWindow |
| gridWidget | QGridWidget | The main widget that contains everything to be displayed. *gridLayout* is applied to it | MainWindow |
| buttonID | Integer | The ID of the button pressed | MainWindow |
| tweetData | TwitterData | The data of the tweet that *buttonID* maps to | MainWindow |
| vertLayoutWidget | QWidget | Widget that *vertLayout* is applied to | MainWindow |
| party | String | The party the tweet is talking about | MainWindow |
| dataDict | Dictionary | Dictionary of all the data to be displayed | MainWindow |
| item | String | Key for each key-value pair in *dataDict* | MainWindow |
| horLayout | QHBoxLayout | Layout that causes all the items added to it to be organised horizontally | MainWindow |
| horLayoutWidget | QWidget | Widget for *horLayout* to be applied to | MainWindow |
| title | QLabel | Stores *item* so it can be displayed | MainWindow |
| data | QLabel | Stores the value that *item* corresponds to so it can be displayed | MainWindow |
| metrics | String | Each element in *tweetData.userMetrics* dictionary | MainWindow |
| titleText | String | The text of *title* | MainWindow |
| key | Integer | The key used to get the data from *widgetDict* | MainWindow |
| filterComboList | List | List of ways to filter the tweets | MainWindow |
| userComboList | List | List of ways to filter the users | MainWindow |
| filterCombo | QComboBox | Combo box of *filterComboList* | MainWindow |
| userCombo | QComboBox | Combo box of *userComboList* | MainWindow |
| refreshButton | QPushButton | Button that refreshes the window | MainWindow |
| menuWidget | QWidget | Widget that is anchored to the top of the screen | MainWindow |
| menuWidgetsList | List | List of widgets to be added to *menuWidget* | MainWindow |
| menuLayout | QHBoxLayout |  |  |

Code

Annotate using text boxes where objectives and skills have been met

Comments will be used to explain what the code does, annotations link to objectives + skills

Hashing Algorithm

from Backend.LinkedList import Node, LinkedList

class HashTable:

def \_\_init\_\_(self, len):

self.table = [None] \* len

self.len = len

# The last 5 digits of id will be key, the twitterObj will be obj

def addItem(self, key, obj):

twitterObj = Node(obj)

index = self.\_\_hashingAlgorithm(key)

if self.table[index] != None: # If there is already an object at that point

self.\_\_addToLinkedList(twitterObj, self.table[index]) # Add to the objects linked list

else:

self.table[index] = self.\_\_createLinkedList(twitterObj)

# Returns the item with the specified key

def getItem(self, key):

node = None

index = self.\_\_hashingAlgorithm(key)

lList = self.table[index]

# Looks for the node in the linked list

if lList.head.data.key != key:

node = lList.findNode(key) # The node was not the head of the list

else:

node = lList.head # The node was the head of the list

return node.data

def \_\_createLinkedList(self, node):

lList = LinkedList(node)

return lList

# Adds node to the end of a linked list

def \_\_addToLinkedList(self, node, lList):

lastNode = lList.getLastNode()

lastNode.next = node

def \_\_hashingAlgorithm(self, key):

return key % self.len

def traverseListTest(self, index):

lList = self.table[index]

lList.traverseListTest()

# Used to test if the algorith works

def testHashing(self):

for llist in self.table:

if llist == None:

continue

lastNode = llist.getLastNode()

if lastNode.data.id != llist.head.data.id:

llist.traverseListTest()

continue

print(f"This is the only item stored {llist.head.data.id}")

The hash table takes the last 5 digits of the Twitter ID as the hashing key. The items are added to a linked list when assigned an index. If there is a collision, the item is added to the linked list stored at the index.

Linked List and Node

# Class of the obj which will hold data + pointer to next obj

class Node:

def \_\_init\_\_(self, d):

self.data = d

self.next = None

class LinkedList:

def \_\_init\_\_(self, h):

self.head = h

self.lastNode = h.next

# Finds the node at the end of the list

def getLastNode(self):

endOfList = False

temp = self.head

while not endOfList:

if temp.next == None:

endOfList = True

else:

temp = temp.next

return temp

# Finds the node with the specified ID

def findNode(self, id):

temp = self.head

nodeFound = False

while not nodeFound:

if temp.data.key != id:

temp = temp.next

else:

nodeFound = True

return temp

The Node class stores all the data for each element in the linked list. *self.next* points towards the next Node object in the linked list. *self.data* stores the TwitterData object of the item. The two functions are used to traverse the list.

Sentiment Analysis Model

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.naive\_bayes import MultinomialNB

from sklearn.metrics import accuracy\_score

import pickle

class SentimentAnalysis:

def \_\_init\_\_(self):

self.\_\_TRAINPERCENTAGE = 0.8

# Gets the text from each tweet

def \_\_getListOfTweets(self, tweetData):

text = []

for tweet in tweetData:

text.append(tweet.text)

return text

# Gets the sentiment from each of the tweets

def \_\_getListOfSentiments(self, tweetData):

sentiment = []

for tweet in tweetData:

sentiment.append(tweet.sentiment)

return sentiment

# Trains and tests a machine learning and vectorization model

def trainModels(self, data, modelPath, vectModelPath):

NB = MultinomialNB()

# Split into train and test data

textData = self.\_\_getListOfTweets(data)

sentimentData = self.\_\_getListOfSentiments(data)

splitTextData = self.\_\_testTrainSplit(textData)

splitSentimentData = self.\_\_testTrainSplit(sentimentData)

trainText = splitTextData[0]

testText = splitTextData[1]

trainSentiment = splitSentimentData[0]

testSentiment = splitSentimentData[1]

vectText = self.\_\_trainVectoriser(trainText, testText, vectModelPath)

vectTrain = vectText[0]

vectTest = vectText[1]

# Trains the model

NB.fit(vectTrain, trainSentiment)

sentiment = NB.predict(vectTest) # Tests the model

# Save model

self.\_\_savePickle(modelPath, NB)

# How accurate the model is

print("accuracy: {}".format(accuracy\_score(testSentiment, sentiment)))

def predictPolarity(self, data, modelPath, vectModelPath):

vectData = self.\_\_featureExtraction(data, vectModelPath)

model = self.\_\_loadPickle(modelPath)

sentiment = model.predict(vectData)

return sentiment

def \_\_trainVectoriser(self, trainText, testText, modelPath):

vect = TfidfVectorizer()

vectTrain = vect.fit\_transform(trainText)

vectTest = vect.transform(testText)

self.\_\_savePickle(modelPath, vect)

return (vectTrain, vectTest)

# Gives a numerical value based on how common the word is

def \_\_featureExtraction(self, text, modelPath):

vect = self.\_\_loadPickle(modelPath)

return vect.transform(text)

def \_\_testTrainSplit(self, data):

splitData = int(round(len(data) \* self.\_\_TRAINPERCENTAGE))

trainData = data[:splitData]

testData = data[splitData:]

return trainData, testData

# Loads the specified .pickle file

def \_\_loadPickle(self, file):

return pickle.load(open(file, 'rb'))

# Saves data as a .pickle file

def \_\_savePickle(self, file, obj):

f = open(file, 'wb')

pickle.dump(obj, f)

f.close()

This algorithm uses the sci-kit learn library to build a naïve bayes machine learning and TF-IDF vectorizer model. It receives the text and its corresponding sentiment as a parameter through the *trainModels* function. It then splits the two into testing and training sets. The training set is used to build the two models and the testing set is used to see how accurate the models are. The two models are then saved as .pickle files using the pickle library.

Merge Sort Algorithm

class Algorithms:

def mergeSort(self, items, asc):

# The recursion will stop when the list has been divided into single items

if len(items) <= 1:

return items

else:

# Splits the list into halves by finding the midpoint

midpoint = (len(items)-1) // 2

leftHalf = items[0:midpoint+1]

rightHalf = items[midpoint+1:len(items)]

leftHalf = self.mergeSort(leftHalf, asc) # Recursive call on left half

rightHalf = self.mergeSort(rightHalf, asc) # Recursive call on right half

# Call funtion to merge both halves

if asc:

mergedItems = self.\_\_mergeAscending(leftHalf, rightHalf)

else:

mergedItems = self.\_\_mergeDescending(leftHalf, rightHalf)

return mergedItems

def \_\_mergeAscending(self, left, right):

merged = [] # New list for merging the items

indexLeft = 0

indexRight = 0

# While there are still items to merge

while indexLeft < len(left) and indexRight < len(right):

# Find the lowest of the two items being compared

if left[indexLeft] < right[indexRight]:

merged.append(left[indexLeft])

indexLeft += 1

else:

merged.append(right[indexRight])

indexRight += 1

# Add to the merged list any remaining data from left list

while indexLeft < len(left):

merged.append(left[indexLeft])

indexLeft += 1

# Add to the merged list any remaining data from right list

while indexRight < len(right):

merged.append(right[indexRight])

indexRight += 1

return merged

# Works the same as ascending

def \_\_mergeDescending(self, left, right):

merged = []

indexLeft = 0

indexRight = 0

while indexLeft < len(left) and indexRight < len(right):

if left[indexLeft] < right[indexRight]:

merged.append(right[indexRight])

indexRight += 1

else:

merged.append(left[indexLeft])

indexLeft += 1

while indexLeft < len(left):

merged.append(left[indexLeft])

indexLeft += 1

while indexRight < len(right):

merged.append(right[indexRight])

indexRight += 1

return merged

Recursive merge sort algorithm that can sort lists in ascending or descending order.

TwitterData

class TwitterData():

def \_\_init\_\_(self, text, cleanText=None, sentiment=None, loc=None, creationDate=None, username=None,

lang=None, key=None, tweetMetrics=None, republican=None, userMetrics=None):

self.text = text

self.cleanText = cleanText

self.sentiment = self.\_\_getSentiment(sentiment)

self.location = loc

self.username = username

self.lang = lang

self.repub = self.\_\_getRepub(republican)

self.metrics = tweetMetrics # Dictionary of metrics such as likes, retweets etc.

self.userMetrics = self.\_\_metricsDict(userMetrics)

self.key = key # Will be used as the hashing key

self.\_\_setDate(creationDate)

# Returns true if the tweet mentions the Republicans

def \_\_getRepub(self, repub):

isRepub = True

if repub == 'TRUE':

isRepub = True

else:

isRepub = False

return isRepub

# Gets rid of the unnecessary characters found in the sentiment column in the .csv file

def \_\_getSentiment(self, sent):

newSent = sent.replace("[", "")

newSent = newSent.replace("]", "")

newSent = newSent.replace("'", "")

return newSent

# Converts the metrics column from a string to a dictionary

def \_\_metricsDict(self, metrics):

lst = []

metrics = metrics.split(" ")

for part in metrics:

newPart = part.replace("{", "")

newPart = newPart.replace("}", "")

newPart = newPart.replace(",", "")

newPart = newPart.replace(":", "")

newPart = newPart.replace("'", "")

lst.append(newPart)

metricsDict = {lst[i]: lst[i + 1] for i in range(0, len(lst), 2)}

return metricsDict

# Gets the time and date

def \_\_setDate(self, timeData):

date = timeData[0:10]

time = timeData[11:19]

return (time, date)

This class is used to store all the data regarding a Tweet and its user. The data is pulled from the .csv file and passed through to the class via the parameters in the *\_\_init\_\_* function. The data is then manipulated if needed, such as *republican* and *userMetrics* before being stored in global variables so they can be accessed by any classes.

DataProcessor

from geopy.geocoders import Nominatim

from Backend.MachineLearningModel import SentimentAnalysis

from Backend.TweetScraper import CollectTweets

from Backend.TextData import TwitterData

import pandas as pd

import spacy

import re

import csv

class DataProcessor:

def \_\_init\_\_(self):

# Initialise all apis & other stuff

self.\_\_nlp = spacy.load("en\_core\_web\_sm")

self.analysis = SentimentAnalysis()

# Create object of TwitterData class to store data

def \_storeData(self, sentiment, text):

twitterObject = TwitterData(sentiment=sentiment, text=text)

return twitterObject

# Reduce the tweet to bare text - remove everything else e.g. punctuation, numbers etc.

def cleanTweet(self, text, lemmatise):

temp = text.lower()

# Followed part of this article - https://catriscode.com/2021/05/01/tweets-cleaning-with-python/

# Removing hashtags and mentions

temp = re.sub("@[A-za-z0-9\_]+", "", temp)

temp = re.sub("#[A-za-z0-9\_]+", "", temp)

# Removing links

temp = re.sub(r"http\S+", "", temp)

temp = re.sub(r"www.\S+", "", temp)

temp = re.sub("'s", "", temp) # Get rid of possesive apostrophes

temp = self.\_decontract(temp) # Decontract contracted words

# Removes all characters that aren't letters or whitespace

temp = re.sub('[^A-Za-z ]+', '', temp)

if lemmatise:

temp = self.\_lemmatize(temp)

return temp

# Decontracts words eg won't -> will not etc.

def \_decontract(self, text):

# specific

text = re.sub(r"won\'t", "will not", text)

text = re.sub(r"can\'t", "can not", text)

# general

text = re.sub(r"n\'t", " not", text)

text = re.sub(r"\'re", " are", text)

text = re.sub(r"\'d", " would", text)

text = re.sub(r"\'ll", " will", text)

text = re.sub(r"\'t", " not", text)

text = re.sub(r"\'ve", " have", text)

text = re.sub(r"\'m", " am", text)

return text

# Finds the root of the word

def \_lemmatize(self, text):

doc = self.\_\_nlp(text)

finalDoc = ""

for token in doc:

finalDoc += token.lemma\_ + ' '

return finalDoc

This is the parent class of *TrainAlgorithm* and *GatherTweetData* and contains functions that are used by both of them. It uses multiple libraries such as *re* and *spacy*. *re* is python’s regular expressions library and the function *cleanText* uses it to remove unnecessary characters from the Tweets passed through. *spacy* is the library that provides many natural language processing tools and is used to lemmatize words in this class.

TrainAlgorithm

# Subclass of DataProcessor

class TrainAlgorithm(DataProcessor):

def \_\_init\_\_(self):

super().\_\_init\_\_()

# Prepares the data from the .csv file so it can be used to train the ML model

def preprocessText(self, filepath):

tweetData = []

rawTweets = pd.read\_csv(filepath)

tweets = rawTweets.drop(columns=["From-User-Id", "To-User-Id", "Language", "Retweet-Count", "PartyName", "Id",

"Negativity", "Positivity", "Uncovered Tokens", "Total Tokens"]) # Removes the unneeded columns

tweets.dropna(axis=0, inplace=True) # Deletes items with empty cells

newText = []

for index in tweets.index:

newText.append(self.cleanTweet(tweets['Scoring String'][index], False))

# Creates a new column in the dataframe and adds the cleaned text to it

tweets['clean\_text'] = newText

# Creates an instance of the TwitterData class to store all relevant data

for index in tweets.index:

tweetSenti = self.\_\_getSentiment(tweets['Score'][index])

tweetText = tweets['clean\_text'][index]

tweetData.append(self.\_storeData(tweetSenti, tweetText))

# Exports to .csv file to see if the data has been manipulated correctly

tweets.to\_csv("CSV\_Files/test.csv")

# The file paths that the ML and vectorizer models will be stored at

modelPath = 'Models/political\_election\_tweets\_NB\_test.pickle'

vectModelPath ='Models/political\_election\_tweets\_vect\_test.pickle'

# Train the model using the data from the .csv file

self.analysis.trainModels(tweetData, modelPath, vectModelPath)

# Get the sentiment based on the score given by the .csv file

def \_\_getSentiment(self, score):

sentiment = None

if score > 0:

sentiment = 'positive'

else:

sentiment = 'negative'

return sentiment

# Call the function stored in its parent class

def \_storeData(self, sentiment, text):

return super().\_storeData(sentiment, text)

# Call the function stored in its parent class

def cleanTweet(self, tweet, lemmatise):

return super().cleanTweet(tweet, lemmatise)

This inherits from *DataProcessor*. It reads the training dataset which is stored as a .csv file. It uses the pandas library to remove all the unnecessary columns. It uses the *cleanText* function it inherited from *DataProcessor* to clean the text and stores it as a *TwitterData* object using the inherited *storeData* function. It then uses this data to train the machine learning and TFIDF vectorizer models.

GatherTweetData

# Subclass of DataProcessor

class GatherTweetData(DataProcessor):

def \_\_init\_\_(self, numOfTweets):

super().\_\_init\_\_()

self.\_\_maxTweets = numOfTweets

self.\_\_geoLocator = Nominatim(user\_agent="NEA\_Project")

self.\_\_tweetCollecter = CollectTweets(self.\_\_maxTweets)

# Used for testing

def testTweets(self):

# Collects small number of tweets

testTweetsDems = self.\_\_tweetCollecter.searchDemocratTweets()

testTweetsReps = self.\_\_tweetCollecter.searchRepublicanTweets()

tweetsDems = self.\_\_organiseData(testTweetsDems)

tweetsReps = self.\_\_organiseData(testTweetsReps)

# Writes the text data to a .txt file

with open(r'Project\_Code/tweets.txt', 'w') as f:

f.write("DEMOCRATS\n\n")

for tweet in tweetsDems:

f.write(f"{tweet['text']}\n// \n")

f.write("REPUBLICANS\n\n")

for tweet in tweetsReps:

f.write(f"{tweet['text']}\n// \n")

# Gets the tweets that have been collected, finds the rest of the data such as polarity &

# location before exporting it to a .csv file

def getTweets(self):

# Searches for tweets

repubData = self.\_\_tweetCollecter.searchRepublicanTweets()

demData = self.\_\_tweetCollecter.searchDemocratTweets()

# Adds the data to a big dictionary

repubDict = self.\_\_organiseData(repubData)

demDict = self.\_\_organiseData(demData)

# Gets rid of all characters apart from text

repubDict = self.cleanTweet(True, repubDict)

demDict = self.cleanTweet(True, demDict)

# Gets the senitment using trained ML and vectorizer models

repubDict = self.\_\_getSentiment(repubDict, 'Models/political\_election\_tweets\_NB.pickle',

'Models/political\_election\_tweets\_vect.pickle')

demDict = self.\_\_getSentiment(demDict, 'Models/political\_election\_tweets\_NB.pickle',

'Models/political\_election\_tweets\_vect.pickle')

# List of TwitterData objects

repubObjList = []

demObjList = []

# Finds location, assigns what party the tweet menitons and

# Creates TwitterData object of it before appending to a list

for object in repubDict:

object['republican'] = True

object['location'] = self.\_\_getLocation(object['location'])

repubObjList.append(self.createTwitterObj(object))

for object in demDict:

object['republican'] = False

object['location'] = self.\_\_getLocation(object['location'])

demObjList.append(self.createTwitterObj(object))

# Combines the party's lists/dictionaries to a combined one

tweetObjDict = repubDict + demDict # This is used to create a dataframe

tweetObjList = repubObjList + demObjList

# Creates dataframe so it can be easily exported to a .csv file

df = pd.DataFrame(tweetObjDict)

df.to\_csv('twitter\_obj.csv', mode='a', index=False, header=False)

return tweetObjList

# Gets the data from the .csv file

def csvToListOfData(self):

twitterObjList = []

with open('twitter\_data\_no\_duplicates.csv') as f:

reader = csv.DictReader(f)

lst = list(reader) # Converts the .csv to a list

# Creates TwitterData object and appends to list

for object in lst:

twitterObjList.append(self.createTwitterObj(object))

return twitterObjList

# Gets polarity of all tweets and adds them to the dictionary

def \_\_getSentiment(self, twitterList, modelPath, vectPath):

for tweet in twitterList:

tempTextList = [] # The vectoriser only takes lists, not strings

tempTextList.append(tweet['text'])

tweet['sentiment'] = self.analysis.predictPolarity(tempTextList, modelPath, vectPath)

return twitterList

# Removes unnecessary characters from the text and lemmatises it

# Calls its parent function

def cleanTweet(self, lemmatise, twitterList):

for tweet in twitterList:

cleanText = super().cleanTweet(tweet['text'], lemmatise)

tweet['clean\_text'] = cleanText

return twitterList

# Creates a class to store all the data

def createTwitterObj(self, data): #id=data['id']

twitterObj = TwitterData(data['text'], loc=data['location'], creationDate=data['created\_at'],

username=data['username'], lang=data['lang'], tweetMetrics=data['metrics'],

cleanText=data['clean\_text'], sentiment=data['sentiment'], republican=data['republican'],

userMetrics=data['user\_metrics'], key=int(data['id']))

return twitterObj

# Finds the state the user is situated in, if available

# Reads the data from a JSON file

def \_\_getLocation(self, loc):

# The state is always the second to last item

state = None

location = self.\_\_geoLocator.geocode(loc, addressdetails=True) # Returns a JSON file

print(type(location))

try:

country = location.address.split(',')[-1]

tweetState = location.raw['address']['state']

if country == ' United States':

state = tweetState

except KeyError:

state = 'US' # If the user hasn't specified the state they live in but they live in the US,

# Location will be set to the US

except:

state = None

return state

# Sorts all the data into a single dict

def \_\_organiseData(self, data):

twitterDataList = []

userData = data[0] # List of dictionaries

tweetData = data[1]

for i in range(len(userData)):

userTweetData = {}

userTweetData['username'] = userData[i]['username']

userTweetData['user\_metrics'] = userData[i]['user\_metrics']

userTweetData['location'] = userData[i]['location']

userTweetData['text'] = tweetData[i]['text']

userTweetData['created\_at'] = tweetData[i]['time\_created']

#userTweetData['id'] = str(tweetData[i]['id'])

userTweetData['key'] = tweetData[i]['key']

userTweetData['lang'] = tweetData[i]['lang']

userTweetData['metrics'] = tweetData[i]['metrics']

userTweetData['republican'] = False

userTweetData['clean\_text'] = None

userTweetData['sentiment'] = None

twitterDataList.append(userTweetData)

return twitterDataList

This class also inherits from *DataProcessor*. It gathers all the Tweets that will be used, finds the relevant data such as the location of the user and the sentiment of the Tweet before exporting all the data to .csv file. The data returned when the location the Tweet user is passed through to the *Nominatim* API is in the form of a JSON file, which the function *\_\_getLocation* handles. This happens once before the actual program is run. When the program is executed, the class pulls the data from the .csv file and returns it to the class that called it.

CollectTweets

import tweepy

class CollectTweets:

def \_\_init\_\_(self, maxTweets):

# Authorise API

# The password has been censored

self.\_\_client = tweepy.Client("AAA\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*hLL")

self.\_\_MAX\_TWEETS = maxTweets

# Get tweets in English mentioning only one of the parties from a US based user that isn't a retweet

self.\_\_REPUBLICAN\_QUERY = "(republican OR republicans) lang:en -(democrat OR democrats) -is:retweet"

self.\_\_DEMOCRAT\_QUERY = "(democrat OR democrats) lang:en -(republican OR republicans) -is:retweet"

# Searches tweets and returns specified data

def searchRepublicanTweets(self):

# The data returned is in the form of a JSON file

repubTweets = self.\_\_client.search\_recent\_tweets(query=self.\_\_REPUBLICAN\_QUERY, tweet\_fields=['created\_at', 'text', 'id', 'lang', 'public\_metrics'],

expansions='author\_id', user\_fields=['username', 'location', 'public\_metrics'], max\_results=self.\_\_MAX\_TWEETS)

# Have to deal with user and tweet data seperately

userData = self.\_\_formatUserDetails(repubTweets.includes['users'])

tweetData = self.\_\_formatTweetDetails(repubTweets.data)

return (userData, tweetData)

# Same as searchRepublicanTweets

def searchDemocratTweets(self):

demTweets = self.\_\_client.search\_recent\_tweets(query=self.\_\_DEMOCRAT\_QUERY, tweet\_fields=['created\_at', 'text', 'id', 'lang', 'public\_metrics'],

expansions='author\_id', user\_fields=['username', 'location', 'public\_metrics'], max\_results=self.\_\_MAX\_TWEETS)

userData = self.\_\_formatUserDetails(demTweets.includes['users'])

tweetData = self.\_\_formatTweetDetails(demTweets.data)

return (userData, tweetData)

# Adds data to a dictionary

def \_\_formatUserDetails(self, userJSON):

userDataList = []

for user in userJSON:

# Will contain the location, username and metrics of the user and their account

userData = {}

userData['location'] = user['location']

userData['username'] = user['username']

userData['user\_metrics'] = user['public\_metrics']

userDataList.append(userData)

return userDataList # Returns a list of dictionaries

# Adds data to a dictionary

def \_\_formatTweetDetails(self, tweetJSON):

tweetDataList = []

for tweet in tweetJSON:

tweetData = {} # Will contain the text, time created at, tweet ID,

# language & metrics of the tweet

tweetData['text'] = tweet['text']

tweetData['time\_created'] = tweet['created\_at']

tweetData['key'] = self.\_\_getKey(tweet['id'])

tweetData['lang'] = tweet['lang']

tweetData['metrics'] = tweet['public\_metrics']

tweetDataList.append(tweetData)

return tweetDataList # Returns a list of dictionaries

# Returns the last 5 digits of the ID

def \_\_getKey(self, id):

strId = str(id)

key = strId[-5::]

intKey = int(key)

return intKey

*CollectTweets* uses Tweepy which is a python library that gives access to the Twitter API. It searches for Tweets mentioning either the Democrats or Republicans with the criteria explained in the comments. The API returns the query as a JSON file. The relevant data is selected from it and stored in dictionaries based on whether the data was about the Tweet or the user. The two dictionaries are then returned to the *GatherTweetData* class.

TwitterDataStore

# Will hold all the data in different lists based on different factors

class TwitterDataStore:

# List of twitter data objects

def \_\_init\_\_(self):

self.completeList = []

self.repubTweetsList = []

self.demTweetsList = []

self.posList = []

self.negList = []

This is used to store *TwitterData* objects based on which party the Tweet is talking about and its sentiment.

CalculateTweetData

import statistics

class CalculateTweetData:

def \_\_init\_\_(self, sorting):

self.sorting = sorting # The merge sort class

# Gets the all the data to do with the datset that is currently displayed

# Returns a dictionary of all the data which will be displayed

def generalData(self, tweetList):

# Finds num of +ve & -ve tweets to do with each party

posTweets = self.\_\_numOfPositveTweets(tweetList)

negTweets = len(tweetList) - posTweets

repubTweets = self.\_\_numOfRepubTweets(tweetList)

demTweets = len(tweetList) - repubTweets

# Finds the percentage of all the values

posPerc = self.\_\_percentageOf(posTweets, len(tweetList))

negPerc = self.\_\_percentageOf(negTweets, len(tweetList))

demPerc = self.\_\_percentageOf(demTweets, len(tweetList))

repubPerc = self.\_\_percentageOf(repubTweets, len(tweetList))

# Uses the statistics library to find the mean & median. Rounds to nearest whole number

meanFollowers = round(statistics.mean(self.\_\_followersList(tweetList)), 0)

medianFollowers = round(statistics.median(self.\_\_followersList(tweetList)), 0)

# Uses merge sort algorithm to sort list from highest follower count to lowest

followerSortedList = self.sorting.mergeSort(self.\_\_followersList(tweetList), False)

maxFollowers = followerSortedList[0] # Account with most followers will be first in the list

minFollowers = followerSortedList[-1] # Account with least followers will be last in the list

data = {

'Number Of Positive Tweets:': posTweets,

'Number Of Negative Tweets:': negTweets,

'Positive Tweets %:': posPerc,

'Negative Tweets %:': negPerc,

'Number of Republican Tweets:': repubTweets,

'Republican Tweets %:': repubPerc,

'Positive Republican Opinion, YouGov, 21/2/23: ': '44.3%',

'Negative Republican Opinion, YouGov, 21/2/23: ':'52.9%',

'Number of Democratic Tweets:': demTweets,

'Democratic Tweets %:': demPerc,

'Positive Democrat Opinion, YouGov, 21/2/23: ': '45.7%',

'Negative Democrat Opinion, YouGov, 21/2/23: ': '52.1%',

'Mean Number of Followers:': meanFollowers,

'Median Number of Follwers:': medianFollowers,

'Max Number of Followers:': maxFollowers,

'Min Number of Followers:': minFollowers

}

return data

# Gets the follower count from TwitterData instance of each tweet

# And appends it to a list

def \_\_followersList(self, lst):

followersList = []

for obj in lst:

followersList.append(int(obj.userMetrics['followers\_count']))

return followersList

# Increases by 1 if the sentiment of the tweet is positive

def \_\_numOfPositveTweets(self, tweetList):

num = 0

for tweet in tweetList:

if tweet.sentiment == 'positive':

num += 1

return num

# Increases by 1 if the party mentioned is the Republicans

def \_\_numOfRepubTweets(self, tweetList):

num = 0

for tweet in tweetList:

if tweet.repub == True:

num += 1

return num

# Finds the % to 2 d.p.

def \_\_percentageOf(self, n1, n2):

return round((n1/n2) \* 100, 2)

This class calculates all the general data regarding whichever list of Tweets is currently displayed. It finds the mean and median number of followers the users have, the number and percentage of Republican, Democrat, positive and negative Tweets. It also uses the merge sort algorithm to find the lowest and highest number of followers.

MainWindow

from Frontend.Layouts import LayoutHandler

from enum import Enum

from PyQt6.QtCore import Qt

from PyQt6.QtGui import QFont

from PyQt6.QtWidgets import (

QGridLayout,

QWidget,

QLabel,

QComboBox,

QPushButton,

QMainWindow,

QVBoxLayout,

QHBoxLayout,

QScrollArea,

QButtonGroup,

QStackedWidget)

class MainWindow(QMainWindow):

def \_\_init\_\_(self, tweets, mainClass, cList, calcData):

super().\_\_init\_\_()

self.main = mainClass # Instance of the main class

self.calcData = calcData # Instance of the CalculateTweetData class

self.currentList = cList # The current list displayed

self.boldFont = QFont()

self.boldFont.setBold(True)

self.\_\_initialiseMenu()

self.\_\_initScreen(tweets)

# Initialises the screen

def \_\_initScreen(self, tweets):

self.buttons = QButtonGroup() # Group of buttons that have a unique ID

self.widgetDict = {}

self.stackWidget = QStackedWidget() # List of pre made widgets

tweetsWidget = QWidget(self)

tweetLayout = QVBoxLayout(self)

# Displays the username and tweet as a single object that can be interacted with via button (username)

# The username is displayed as a button and the tweet is stored as a label

for tweet in tweets:

# The username and tweet are added to a vertical layout which is then added to another vertical layout

tweetContainer = QWidget(tweetsWidget)

tweetContainerLayout = QVBoxLayout(tweetsWidget)

username = QPushButton(tweetsWidget, text=tweet.username)

username.setFont(self.boldFont) # The username will be bold to help make it stand out

tweetLabel = QLabel(tweetsWidget, text=tweet.text)

tweetLabel.setWordWrap(True) # Text moves on to a new line if it's too big

# Add the username and text to the same layout

tweetContainerLayout.addWidget(username)

tweetContainerLayout.addWidget(tweetLabel)

# Add the button to the button group, the tweet key will be the button ID

# This makes it easy to retrieve the tweet data from the hash table

self.buttons.addButton(username, tweet.key)

# Add it to the main tweet layout

tweetContainer.setLayout(tweetContainerLayout)

tweetLayout.addWidget(tweetContainer)

# Return the ID of the button clicked

self.buttons.idClicked.connect(self.\_\_usernamePressed)

tweetsWidget.setLayout(tweetLayout)

# Able to scroll through the tweets

scroll = QScrollArea(self)

scroll.setVerticalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAlwaysOn)

scroll.setHorizontalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAlwaysOff)

scroll.setWidgetResizable(True)

scroll.setWidget(tweetsWidget)

# Will contain the data to do with the tweet/tweets

self.rightWidget = QWidget(self)

self.rightWidget.setLayout(self.\_\_rightLabelText())

self.widgetDict[0] = self.stackWidget.addWidget(self.rightWidget)

self.stackWidget.setCurrentIndex(self.widgetDict[0])

# Able to scroll through the data

rightScroll = QScrollArea(self)

rightScroll.setVerticalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAlwaysOn)

rightScroll.setHorizontalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAlwaysOff)

rightScroll.setWidgetResizable(True)

rightScroll.setWidget(self.stackWidget)

# The left side are the tweets & the right side is the data

self.gridLayout = QGridLayout(self)

self.gridLayout.addWidget(scroll, 1, 0, 1, 1, alignment=Qt.AlignmentFlag.AlignLeft)

self.gridLayout.addWidget(rightScroll, 1, 1, 1, 2)

gridWidget = QWidget(self)

gridWidget.setLayout(self.gridLayout)

self.setCentralWidget(gridWidget)

# Displays the specific data if a button is pressed

def \_\_usernamePressed(self, buttonID):

# Get the data from the hash table using the buttonID as the key

tweetData = self.main.getItemFromHash(buttonID)

vertLayout = QVBoxLayout()

vertLayoutWidget = QWidget()

# Makes the data appear more readable to the user

if tweetData.repub:

party = 'Republican'

else:

party = 'Democrat'

if len(tweetData.location) == 0:

tweetData.location = 'Not Available'

dataDict = {'Username: ': tweetData.username,

'Sentiment: ': tweetData.sentiment,

'Location: ': tweetData.location,

'Party: ': party}

# Loops through the data so it can be displayed

for item in dataDict:

horLayoutWidget = QWidget()

horLayout = QHBoxLayout()

title = QLabel(text=item)

data = QLabel(text=dataDict[item])

# Adds the data to a horizontal layout

horLayout.addWidget(title)

horLayout.addWidget(data)

horLayoutWidget.setLayout(horLayout)

vertLayout.addWidget(horLayoutWidget)

# Loops through all the metrics dictionary and adds them to display

for metric in tweetData.userMetrics:

horLayoutWidget = QWidget()

horLayout = QHBoxLayout()

# Makes the data more presentable to the user

if metric == 'followers\_count':

titleText = 'Number of Followers: '

elif metric == 'following\_count':

titleText = 'Number of Accounts Following: '

elif metric == 'tweet\_count':

titleText = 'Number of Tweets: '

else:

titleText = 'Number of Lists Added to: '

title = QLabel(text=titleText)

data = QLabel(text=tweetData.userMetrics[metric])

# Adds to a horizontal layout

horLayout.addWidget(title)

horLayout.addWidget(data)

horLayoutWidget.setLayout(horLayout)

vertLayout.addWidget(horLayoutWidget)

vertLayoutWidget.setLayout(vertLayout)

self.widgetDict[buttonID] = self.stackWidget.addWidget(vertLayoutWidget)

self.\_\_updateRightLabel(buttonID)

# Displays the general data

def \_\_rightLabelText(self):

cList = self.main.getCurrentList()

dataDict = self.calcData.generalData(cList) # Gets the general data

vertLayout = QVBoxLayout()

for item in dataDict:

horWidget = QWidget()

horLayout = QHBoxLayout()

title = QLabel(text=item)

# If the data in the dictionary isn't a string, convert it to one

if not isinstance(dataDict[item], str):

data = QLabel(text=str(dataDict[item]))

else:

data = QLabel(text=dataDict[item])

# Adds to horizontal layouy

horLayout.addWidget(title)

horLayout.addWidget(data)

horWidget.setLayout(horLayout)

vertLayout.addWidget(horWidget)

return vertLayout

# Changes the data displayed based on what has been selected

def \_\_updateRightLabel(self, key):

self.stackWidget.setCurrentIndex(self.widgetDict[key])

# Initialises the main bar at the top of the window

def \_\_initialiseMenu(self):

filterComboList = ['Filters', 'Republicans', 'Democrats', 'Positive', 'Negative']

userComboList = ['Sort', 'Most Followers', 'Least Followers', 'A-Z', 'Z-A']

# Creates drop down boxes

self.filterCombo = self.\_\_addComboBox(filterComboList)

self.userCombo = self.\_\_addComboBox(userComboList)

# Resets the window

refreshButton = QPushButton(parent=self, text='Refresh')

# Calls these functions when one of the options is pressed

self.filterCombo.activated.connect(self.\_\_updateFilterState)

self.userCombo.activated.connect(self.\_\_updateUserState)

refreshButton.clicked.connect(self.\_\_resetLabels)

# Pins them to the top of the window

menuWidget = QWidget()

menuWidgetsList = [self.filterCombo, self.userCombo, refreshButton]

menuLayout = LayoutHandler.createHBox(parent=self, widgets=menuWidgetsList)

menuWidget.setLayout(menuLayout)

self.setMenuWidget(menuWidget)

# Resets the window

def \_\_resetLabels(self):

self.main.refreshDisplay()

# The index of the button corresponds to which button was pressed

def \_\_updateFilterState(self, index):

self.main.filterTweets(index)

# The index of the button corresponds to which button was pressed

def \_\_updateUserState(self, index):

if index == 1: # Most followers

self.main.orderByFollowerCount(False)

elif index == 2: # Least followers

self.main.orderByFollowerCount(True)

elif index == 3: # A-Z

self.main.orderByUsername(True)

elif index == 4: # Z-A

self.main.orderByUsername(False)

def \_\_addComboBox(self, text):

cBox = QComboBox(self)

cBox.addItems(text)

return cBox

# Enums

class FilterComboState(Enum):

COMPLETE = 0

REPUB = 1

DEM = 2

POS = 3

NEG = 4

class UserComboState(Enum):

SORT\_USERS = 0

MOST\_FOLLOWERS = 1

LEAST\_FOLLOWERS = 2

class LayoutType(Enum):

HORIZONTAL = 0

VERTICAL = 1

GRID = 2

PyQt6 is used in the class to create the display. It create objects containing the text and username of each Tweet and adds it to the left side of the display. The right side displays the data regarding all the Tweets currently displayed. If the username of a Tweet is pressed, the right side data will change to display the data about the specific Tweet and the user who posted it. It also allows the user to filter which Tweets they can see by adding drop down boxes which lets them select which Tweets they want displayed based on a variety of factors such as which party the Tweets mention. It uses Enums to figure out which drop down button was pressed.

LayoutHandler

from PyQt6.QtWidgets import (

QHBoxLayout,

)

class LayoutHandler:

# Loops through all the widgets and adds them

# To the layout

def createHBox(parent, widgets):

horLayout = QHBoxLayout(parent)

for w in widgets:

horLayout.addWidget(w)

return horLayout

This class is used to more easily create horizontal layouts.

MainClass

from enum import Enum

import sys

from Backend.TextProcessing import GatherTweetData

from Backend.TwitterDatabase import TwitterDataStore

from Backend.Hashing import HashTable

from Backend.SortingAlgorithms import Algorithms

from Frontend.Display import MainWindow

from Frontend.CalculateTweetData import CalculateTweetData

from PyQt6.QtWidgets import QApplication

class CurrentList(Enum):

COMPLETE = 0

REPUB = 1

DEM = 2

POS = 3

NEG = 4

class MainClass:

def \_\_init\_\_(self):

# Starts the event loop, needed to display the window

self.app = QApplication(sys.argv)

# Initalising all the classes

self.sorting = Algorithms()

self.tds = TwitterDataStore()

self.gtd = GatherTweetData(10)

self.tweetData = CalculateTweetData(self.sorting)

self.currentList = CurrentList.COMPLETE

self.\_\_searchTweets()

self.hashTable = HashTable(len(self.tds.completeList) \* 2) # Create the hash table

self.\_\_hash(self.tds.completeList) # Add tweets to hash table

self.\_\_displayTweets(tweetsList=self.tds.completeList)

sys.exit(self.app.exec()) # Allows the user to close the program

# Gets tweets from .csv file and adds them to relevant lists

def \_\_searchTweets(self):

tweets = self.gtd.csvToListOfData()

for object in tweets:

self.\_\_storeData(object)

# Creates a window and displays all the data

def \_\_displayTweets(self, tweetsList=None):

self.window = MainWindow(tweetsList, self, self.currentList, self.tweetData)

self.window.show()

# Add item to hash table

def \_\_hash(self, list):

for item in list:

self.hashTable.addItem(item.key, item)

# Gets item from hash table

def getItemFromHash(self, key):

return self.hashTable.getItem(key)

# Displays the list of tweets selected from the drop down box

def filterTweets(self, index):

self.currentList = CurrentList(index)

currentList = self.getCurrentList()

self.window.close() # Closes the window so a new one can be created

self.\_\_displayTweets(currentList)

# Sorts the tweets in alphabetical order of their usernames

def orderByUsername(self, asc):

tweetsList = self.getCurrentList()

tweetDict = {}

# Adds tweets to a dictionary with their key so they can be easily accessed

for tweet in tweetsList:

tweetDict[tweet.username.upper()] = tweet.key

sortedList = self.\_\_sortItems(tweetDict, asc)

self.window.close()

self.\_\_displayTweets(sortedList)

# Same as orderByUsername

def orderByFollowerCount(self, asc):

tweetsList = self.getCurrentList()

tweetDict = {}

for tweet in tweetsList:

# Creates dictionary with the follower count as the key and the hash key as the value

# Allows easy access to the key

tweetDict[int(tweet.userMetrics['followers\_count'])] = tweet.key

sortedList = self.\_\_sortItems(tweetDict, asc)

self.window.close()

self.\_\_displayTweets(sortedList)

# Uses the merge sort algorithm to sort the tweets

def \_\_sortItems(self, dict, asc):

tempSortedList = []

sortedTweets = []

# Creates a list of all the keys in the dictionary and sorts them

tempSortedList = self.sorting.mergeSort(list(dict.keys()), asc)

for tweet in tempSortedList:

sortedTweets.append(self.getItemFromHash(dict[tweet]))

return sortedTweets

# Gets the list of tweets that's currently displayed

def getCurrentList(self):

cList = []

if self.currentList == CurrentList.COMPLETE:

clist = self.tds.completeList

elif self.currentList == CurrentList.REPUB:

clist = self.tds.repubTweetsList

elif self.currentList == CurrentList.DEM:

clist = self.tds.demTweetsList

elif self.currentList == CurrentList.POS:

clist = self.tds.posList

else:

clist = self.tds.negList

return clist

# Resets the window

def refreshDisplay(self):

self.\_\_displayTweets(self.tds.completeList)

# Adds tweets to relevant lists

def \_\_storeData(self, obj):

self.tds.completeList.append(obj)

if obj.repub:

self.tds.repubTweetsList.append(obj)

else:

self.tds.demTweetsList.append(obj)

if obj.sentiment == 'positive':

self.tds.posList.append(obj)

else:

self.tds.negList.append(obj)

x = MainClass() # Calls the program

This is the main class that is called to run the program. It stores instances of almost every class. It calls the function that adds Tweets to the hash table, manages sorting the Tweets using the merge sort algorithm. It adds the Tweets to the lists in the *TwitterDataStore* class and calls the function to create the display.

Testing

Add end user testing in a similar fashion to this w/ what has been tested, the outcome, evidence of outcome and any bugs/ final thoughts added @ end

The numbering convention of the tests match up with the user objectives specified in the Analysis section. Extra details related to each of the tests will be given with the screenshots if needed

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test No.** | **Description** | **Test Data** | **Expected Outcome** | **Actual Outcome** | **Changes Needed** |
| 1.1 | The program should be able to search for and return tweets | Call the *testTweets* function from the G*atherTweetData* class | The text of 10 tweets will be exported to a text file and the number of Tweets pulled stated by the Twitter Developer Portal will increase from 110 to 120 | As expected | None |
| 1.2 | The tweets gathered should be the most recent ones | Call the *testTweets* function from the G*atherTweetData* class | The time the Tweets were tweeted will be exported to a .csv file. The time should be very similar to the current time | As expected | None |
| 1.3 | The tweets gathered should have the words *Democrat/s* or *Republican/s* but not both. Capitalisation doesn’t matter | Call the *testTweets* function from the G*atherTweetData* class | The text of 20 tweets will be exported to a text file | As expected | None |
| 2.1 | The ML model should be able to predict the polarity of any text given to it | ‘Today has been a great day’.  ‘Today has been terrible’ | The ML model should predict the first sentence as positive and the second one as negative | As expected | None |
| 2.2 | Check if the ML model has an acceptable level of accuracy | 20% of the training dataset | The accuracy of the model represented as a decimal | As expected | None, the model performed very well. |
| 2.3 | The program should assign a polarity to every tweet pulled | Call the *getTweets* function from the G*atherTweetData* class | All the tweets will be given a polarity | As expected | None |
| 2.4 | The program should have gathered the specified data from each Tweet and user | Call the *getTweets* function from the G*atherTweetData* class | A .csv file with all the specified pieces of data as headings | As expected | None |
| 3.1 | UI should be displayed when the program is run | Run the program | The display should show | As expected | None |
| 3.2 | The correct options should show when the drop down boxes are selected | Select the drop down boxes | The correct options should be displayed | As expected | None |
| 3.3 | General data about the whole dataset should be displayed to the right of the tweets | Run the program | The data should be displayed on the right with the tweets taking up 2/3 of the space | As expected | None |
| 3.4 | The data displayed should be correct | Use excel to calculate the data | The data from excel should match up with the data displayed | As expected | None |
| 5.1 | Pressing the username for each tweet displays the data to do with the selected tweet | Press the username button for 3 tweets | The general data is replaced with tweet specific data | As expected | None |
| 5.2 | The Tweet specific data is correct | Press the username button for the same tweets as *5.1* | The data matches up within a reasonable range with their Twitter account | As expected | None |
| 5.3 | Pressing the refresh button refreshes the display | Press the refresh button | The Tweet specific data is replaced with the general data | As expected | None |
| 6.1 | The *Republicans filter works* | Press the *Republican* option in the drop down box | Only Republican Tweets are displayed | As expected | None |
| 6.2 | The *Democrats* filter works | Press the *Democrat* option in the drop down box | Only Democrats Tweets are displayed | As expected | None |
| 6.3 | The *Positive* filter works | Press the *Positive* option in the drop down box | Only Tweets the ML model classed as positive are displayed | As expected | None |
| 6.4 | The *Negative* filter works | Press the *Negative* option in the drop down box | Only Tweets the ML model classed as negative are displayed | As expected | None |
| 6.5 | Pressing the filter buttons changes the data displayed on the right side of the screen | Press each of the filter drop down buttons | The general data will change according to the button pressed | As expected | None |
| 6.6 | The functionality of the username button still works for the *Republican* filter | Press the *Republican* filter drop down button and press three different username buttons | The general data will change to the Tweet specific data and it will match with the user’s Twitter account | As expected | None |
| 6.7 | The functionality of the username button still works for the *Democrat* filter | Press the *Democrat* filter drop down button and press three different username buttons | The general data will change to the Tweet specific data and it will match with the user’s Twitter account | As expected | None |
| 6.8 | The functionality of the username button still works for the *Positive* filter | Press the *Positive* filter drop down button and press three different username buttons | The general data will change to the Tweet specific data and it will match with the user’s Twitter account | As expected | None |
| 6.9 | The functionality of the username button still works for the *Negative* filter | Press the *Negative* filter drop down button and press three different username buttons | The general data will change to the Tweet specific data and it will match with the user’s Twitter account | As expected | None |
| 7.1 | Sorting by *Most Followers* works | Press the *Most Followers* option in the drop down box | The Tweets will be displayed in order of most to least followers | As expected | None |
| 7.2 | Sorting by *Least Followers* works | Press the *Least Followers* option in the drop down box | The Tweets will be displayed in order of least to most followers | Failed | Change the parameter of the *orderByFollowerCount* function in the *MainWindow* class |
| 7.3 | Sorting usernames alphabetically from *A-Z* works | Press the *A-Z* option in the drop down box | The Tweets will be displayed in order of their usernames from A-Z | As expected | None |
| 7.4 | Sorting usernames alphabetically from ­*Z-A* works | Press the *Z-A* option in the drop down box | The Tweets will be displayed in order of their usernames from Z-A | As expected | None |
| 7.5 | The functionality of the username button still works for the *Most Followers* sort | Press the *Most Followers* drop down button and press three different username buttons | The general data will change to the Tweet specific data and it will match with the user’s Twitter account | The user *ProjectLincoln* must’ve changed their location.Apart from that, as expected | None |
| 7.6 | The functionality of the username button still works for the *Least Followers* sort | Press the *Least Followers* drop down button and press three different username buttons | The general data will change to the Tweet specific data and it will match with the user’s Twitter account | As expected | None |
| 7.7 | The functionality of the username button still works for the *A-Z* sort | Press the *A-Z* drop down button and press three different username buttons | The general data will change to the Tweet specific data and it will match with the user’s Twitter account | As expected | None |
| 7.8 | The functionality of the username button still works for the *Z-A* sort | Press the *Z-A* drop down button and press three different username buttons | The general data will change to the Tweet specific data and it will match with the user’s Twitter account | As expected | None |
| 8.1 | The user can compare the data presented with real-world data | Run the program | The right side data will include general the data along with data collected from YouGov concerning the two parties | As expected | None |

**1.1**

Before test:

After test:

**1.2**

The time when the Tweet times were exported

There’s roughly a 20 second difference between the time of the tweets being posted and the current time. The 20 second delay is most likely due to other factors so we can safely say the tweets being pulled are the most recent ones

**1.3**

**2.1**

**2.2**

**2.3**

Since there’s too many tweets to show in a reasonable amount of screenshots, I’ve displayed the first, last and middle 20 tweets

**2.4**

The fields to be returned are:

1. The date and time the tweet was posted
2. The text of the tweet
3. The tweet ID
4. The language of the tweet
5. How many retweets, replies, likes, quote tweets and impressions the tweet has
6. The username of the user who posted the tweet
7. The location of the user, if available
8. How many follwers, accounts following, tweets posted and lists added to for the account that posted the tweet
9. Which party the tweet mentions
10. The polarity of the tweet

The fields that have been returned are:

1. created\_at
2. text
3. id
4. lang
5. metrics
6. username
7. location
8. user\_metrics
9. republican
10. sentiment

**3.1**

**3.2**

The options that should show for the *Filters* drop down box are:

1. Republicans
2. Democrats
3. Positive
4. Negative

The options that should show for the *Sort* drop down box are:

1. Most Followers
2. Least Followers
3. A-Z
4. Z-A

**3.3**

**3.4**

The fields that are able to be calculated through Excel are:

1. Number of positive tweets
2. Number of negative tweets
3. Number of Republican tweets
4. Number of Democrat tweets

This is the command used to find the number of negative tweets:

*=COUNTIF(K2:K961, "['negative']")*. The *“[‘negative’]”* is changed to *“[‘positive’]”* to find the number of positive tweets

This the command used to find the number of Republican tweets:

*=COUNTIF(I2:I961, "TRUE")*. The *“TRUE”* is changed to a *“FALSE”* to find the number of Democrat tweets

**5.1**

First tweet in the dataset

Tweet from the middle of the dataset

Last Tweet from the dataset

**5.2**

Around 30 days have passed between the creation of the dataset and this test being carried out so slight variations to the data is to be expected.

Top of the dataset

Middle of the dataset

Bottom of the dataset

**5.3**

Before

After

**6.1**

Top of dataset

Middle of dataset

End of dataset

**6.2**

Top of the dataset

Middle of the dataset

Bottom of the dataset

**6.3**

Top of the dataset

Middle of the dataset

Bottom of the dataset

**6.4**

Top of the dataset

Middle of the dataset

Bottom of the dataset

**6.5**

Before

Republicans

Democrats

Positive

Negative

**6.6**

Before

Top of the dataset

Middle of the dataset

Bottom of the dataset

**6.7**

Before

Top of the dataset

The location is only displayed if the user is in the US

Middle of the dataset

Bottom of the dataset

Text

Description automatically generated

**6.8**

Before

Table

Description automatically generated with low confidence

Top of the dataset

Text

Description automatically generatedA screenshot of a tree

Description automatically generated with medium confidence

Middle of the dataset

Graphical user interface, text, application, chat or text message

Description automatically generatedText

Description automatically generated with medium confidence

Bottom of the dataset

Text

Description automatically generated

**6.9**

Before

Top of the dataset

Text

Description automatically generated with medium confidence

Middle of the dataset



Bottom of the dataset

A picture containing application

Description automatically generatedGraphical user interface

Description automatically generated

**7.1**

Top of the dataset

Graphical user interface, text, application, email

Description automatically generated

Middle of the dataset

Bottom of the dataset

**7.2**

Top of the dataset: failed test

Top of the dataset: Successful test

Middle of the dataset

Bottom of the dataset

**7.3**

Top of the dataset

Bottom of the dataset

**7.4**

Top of the dataset

Bottom of the dataset

**7.5**

Before

Top of the dataset

Middle of the dataset

Bottom of the dataset

**7.6**

Before

Top of the dataset

Middle of the dataset

Bottom of the dataset

**7.7**

Before

Top of the dataset

Middle of the dataset

Bottom of the dataset

**7.8**

Before

Top of the dataset

Middle of the dataset

Bottom of the dataset

**8.1**

Evaluation

User Feedback

James Dixon few more questions + analyse it + conclusion, could move this to end of section

**Is the UI intuitive and easy to use?**

Yes, it was very obvious what to do

**How accurately would you say the program predicted the polarity of the tweets?**

I think the program worked well at predicting polarity. As a whole, however, it seems to struggle understanding sarcasm

**Were there any bugs that you encountered?**

No

**What could be improved?**

Addition of visual representation of statistics to make it easier to visualise the tends in the population as a whole

Objective Evaluation

**Objective 1**

Collect tweets that mention the keywords *democrat/democrats* or *republican/republicans*

**How has it been achieved?**

The tweets have been collected via the Twitter API. All the tweets mention either on or the other parties but neither

**How could it be improved?**

I wasn’t able to collect Tweets in real time due to the changes made to the Twitter API. Finding a new API that gets around the limitations could be one way to improve it

**Objective 2**

Predict the polarity of each tweet and gather data to do with each tweet and twitter user

**How has it been achieved?**

Passing through the text of each Tweet through the machine learning model to get the sentiment. The data was gathered by specifying which pieces of data were to be returned from the Twitter API query.

**How could it be improved?**

I think the machine learning model was overfitted since its accuracy on the test data was very high, but it misunderstood quite a few tweets. It also struggled with sarcasm. It could be improved by retraining the model using a different testing set to try and improve its accuracy.

**Objective 3**

Display all the tweets collected to the user along with all the data gathered

**How was it achieved?**

The general data was displayed alongside the specific data when the username was pressed

**How could it be improved?**

Adding some visual elements such as graphs to help visualise the data

**Objective 4**

Allow the user to specify the types of data they would like to accompany the tweets

**How was it achieved?**

It wasn’t. Due to the changes made to the Twitter API, I wasn’t able to implement this feature

**How could it be improved?**

Finding a different API that can avoid these limitations

**Objective 5**

The user should be able to choose to view data regarding the whole dataset of tweets or just individual tweets

**How was it achieved?**

Making the username of each Tweet a button that changes the general data on the right to show the specific data regarding the Tweet

**How could it be improved?**

It would be interesting to be able to look at other Tweets the user tweeted regarding the two parties and see how their views compare. Adding a link to the actual Tweet would’ve been useful too

**Objective 6**

The user should be able to filter out the collected tweets based on certain factors like polarity or which party the tweet is about and view data regarding those tweets

**How was it achieved?**

The drop down box labelled *Filters* contains buttons that only display the Tweets that mention one of the parties or only displays Tweets that have been classified as positive etc.

**How could it be improved?**

Implementing a feature that allows multiple boxes to be selected. E.g. selecting the *Republicans* and *Positive* options so that positive Tweets mentioning the Republican party are displayed

**Objective 7**

The user should be able to view the tweets in ascending/descending order based on different factors such as the data tweeted or the number of followers each user has and view data regarding those tweets

**How was it achieved?**

The drop down box labelled *Sort* contains buttons that sorts the Tweets in order of follower count and in alphabetical order of their usernames

**How could it be improved?**

Same as *Objective 6.* Adding a way to select multiple options at the same time

**Objective 8**

The user should be able to compare the collected data to current real-world data to see if the opinions on Twitter are different to real life

**How was it achieved?**

Data from *YouGovAmerica* which tracks that tracks how registered US voters view the two parties is displayed alongside the general data so the user can easily compare the two values

**How could it be improved?**

Adding more data that relates to the opinions collected and the opinions presented by *YouGovAmerica* would’ve been useful. Things like the percentage difference between the two. Visual elements such as graphs could’ve improved the readability of the data.

**Objective 9**

The user should be able to compare the collected data to previous real-world data such as previous elections

**How was it achieved?**

It wasn’t.

**How could it be improved?**

Actually implementing it. Finding a dataset of previous election data and adding that to the display. If a dataset doesn’t exist, creating one myself would’ve been an option

Next Steps

Overall, I think the program works well and meets most of the objectives set out in the Analysis stage, but it needs improvements in certain areas. The main area being the machine learning model.

The model works well predicting the opinion of simple sentences where it is obvious what the opinion is. It also works with more complex pieces of text but begins to fall short when sarcasm is involved.

It also miscategorises Tweets that are talking negatively about a person who represents one of the parties while mentioning the other. For example, a Tweet talking about how the user doesn’t like the policies Joe Biden has implemented and thinks the Republicans should be in power would be classed as a negative Tweet, due to their opinion about Joe Biden, but would also be a Republican tweet since it mentions them. Therefore, it would contribute to the number of negative Republican Tweets even though the user has a positive opinion about them.

The steps I would take to help solve this problem would include:

1. Looking into different, possibly more complex, machine learning algorithms
2. Find more datasets that can be used to test and train the model. Possibly some that include sarcasm so it can be better equipped to detect it
3. Find a way to

Another way the program can be improved is by adding more visual elements to the data such as graphs and charts. These would make the data more readable and more interesting to look at than just text. These could be better improved if the data collected covered a greater length of time. That would allow the user to see how opinions have changed over time. It could also allow users to view Tweets within a certain time period to see how users reacted to certain events such as elections or big events.

Some of things that could be added:

1. Pie charts comparing the percentage of positive and negative or Republican and Democrat Tweets
2. A map showing where Tweets with their location available have been Tweeted from
3. Scatter graphs showing how the number of positive, negative, Republican and Democrat Tweets have changed over time
4. Scatter graphs that compare the data stated in the previous bullet point to real world data

Adding the ability to compare the collected data to how previous generations viewed each of the parties throughout history would’ve been an interesting feature. The same visual elements stated previously could be applied but with this new, historical data.

New UI

This will be a dropdown box containing the buttons *Text Display, Graphs* and *Map*

Different scatter graphs could be shown displaying the number of positive, negative, republican or democrat tweets against time

A map of the USA will be shown, and each dot represents a Tweet from that area

1. "More than eight-in-ten Americans get news from digital devices." 12 Jan. 2021, <https://pewrsr.ch/2MZqns7>. Accessed 25 May. 2022. [↑](#footnote-ref-1)
2. "Group Polarization - What is Psychology?." <http://psychology.iresearchnet.com/papers/group-polarization/>. Accessed 8 Sept. 2022. [↑](#footnote-ref-2)
3. "Understanding Political Twitter. Using tweet sentiment analysis to…." 6 May. 2020, <https://towardsdatascience.com/understanding-political-twitter-ce3476a38377>. Accessed 6 Jun. 2022. [↑](#footnote-ref-3)
4. "On Using Twitter to Monitor Political Sentiment and Predict Election ...." 13 Nov. 2011, <https://aclanthology.org/W11-3702.pdf>. Accessed 21 Nov. 2022. [↑](#footnote-ref-4)