Technical Guide

Project Title: PandemicAnalysis: A data analysis platform for COVID-19 using Twitter

Student 1 Name: Jaime de Vivero Woods

Student ID: 19447494

Student 2 Name: Alen Tom Joy

Student ID: 18313576

Stream: CASE4

Project Supervisor Name: Renaat Verbruggen

Document Type: Technical Guide

Date of completion: 06/05/2023

Table of Contents

Abstract

This project performs sentiment analysis on publicly available tweets about the COVID-19 pandemic to determine how negative or positive they were (sentiment). The tweets were stored in a database and a number of classifier models were evaluated to find the most accurate. We developed a Django web-app to provide graphs and statistics of the models. The website allows users to select time frames and graphs to display results, the sentiment is then calculated for that time frame and results are displayed.

Motivation

Last year, during our 3rd-year project brainstorming sessions, the idea of performing sentiment analysis on Twitter data came up. Intrigued by the potential insights and challenges it presented, we researched this topic. However, as we delved deeper into the complexities involved, we realised that it would be a project that required more time and resources than we had available last year. We made the decision to pursue a different project that aligned better with our timeline and available resources. When this year's project cycle began, we decided to revisit the Twitter sentiment analysis project that we postponed. Our primary objective for this project is to analyse and compare different models for predicting sentiment using Twitter data as our dataset. We believe that Twitter is an ideal source for conducting sentiment analysis since it offers a huge amount of opinions, emotions, and perspectives. Twitter has millions of users worldwide that provide a diverse range of opinions and perspectives. This diversity enhances the accuracy and reliability of sentiment analysis by ensuring a more comprehensive representation of public sentiment across various demographics, locations, and interests. Another advantage is Twitter's character limit of 280 characters, which aids sentiment analysis by encouraging users to convey their thoughts and emotions in a concise manner. The brevity of tweets simplifies the process of extracting and analysing sentiment,

making it more efficient. Additionally, Twitter provides geolocation data for tweets, enabling sentiment analysis of particular regions or countries.

The purpose of the project is to evaluate different models of predicting sentiment analysis. The system displays tweet data and sentiment results in different graphs. The system achieved this by using Textblob's polarity function and the predictor models we developed.

Research

During the research process, our initial plan was to use the Twitter API to gather a sample of tweets and perform tagging on the dataset to train our models. However, our supervisor advised us to use a tagged dataset, which resulted in us finding an open-source pre-tagged dataset that updates the tweets on a monthly basis. We only discovered later that the dataset was tagged automatically using TextBlob. After selecting the dataset, we conducted thorough research on various models for sentiment prediction. We explored the data requirements, working principles, advantages, and disadvantages of each model. Additionally, we referred to the CA4010 module on Data Warehousing & Data Mining to identify algorithms that we could implement and evaluate. Based on our research, we selected a combination of classifiers and clustering methods including Naïve Bayes, Decision Tree, KNN, K-Means, and Logistic Regression. We opted for Naïve Bayes and Decision Tree Classifiers as they were well-suited for handling dichotomous categorical data, which aligned with our dataset. We decided to use KNN to investigate the accuracy of a model based on Euclidean Distance. We also decided to incorporate K-Means as a clustering model for comparison purposes, although we expected that its accuracy would be lower than the other models due to the dichotomous nature of our dataset.

The final model we implemented was Logistic Regression. As a supervised machine learning algorithm, logistic regression is well-suited to binary classification tasks, such as sentiment analysis. It can be used for sentiment analysis by treating each word in the text as a feature and training a model to predict the sentiment based on the frequency of these features. Additionally, logistic regression is computationally efficient, enabling the analysis of large datasets without the need for significant computational resources.

Design

Architecture Overview

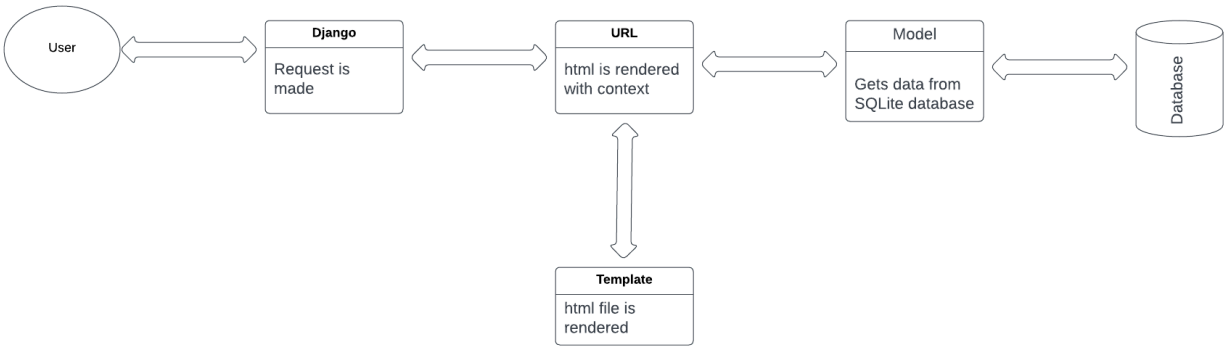
Backend

The backend architecture of our system consists of a SQLite database that stores the tweet dataset in two tables. These two tables are “Tweets” and “TweetPolarity”. The “Tweets” table contains all tweet data parsed from the dataset, while the other contains 9 less attributes, but has a polarity attribute. The data from tables is used to calculate the graphs and passed to the frontend of the application. The predictor models were developed using tweet data collected from our dataset. These models are graphed using a specific graphing function and are passed to the frontend when a GET request is received. To improve the overall performance of our system, we decided to calculate the polarity values of tweets only once during the addition of tweets to the database. This helped improve the speed and efficiency of the system. To further improve the speed of loading graphs to the UI, we saved the calculated graphs locally as HTML files after they were computed for the first time. By doing this, we could call the precomputed graphs using GET requests instead of calculating them in real-time. This decision led to an exponential improvement in performance, and it made the system much more responsive and user-friendly.

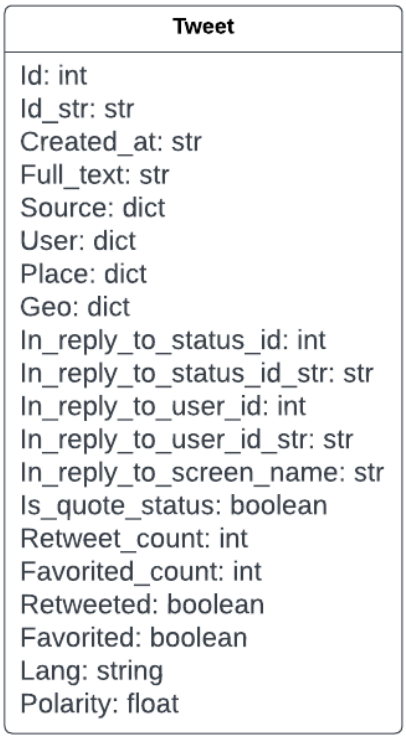
Frontend

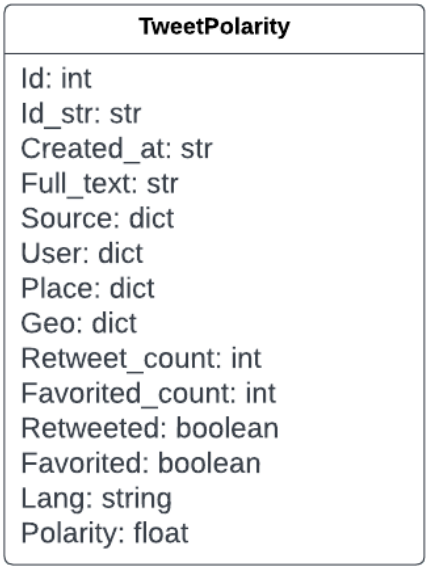
The client-side UI was developed using Django, Python, HTML, CSS and Javascript. The frontend follows a Model-View-Template (MVT) architecture. When a user accesses a page, a GET request is sent to the corresponding URL. Django then maps the URL to a view that interacts with the model and template, finally rendering a template. These views use database functions to select and graph data from tables.

High-Level Design

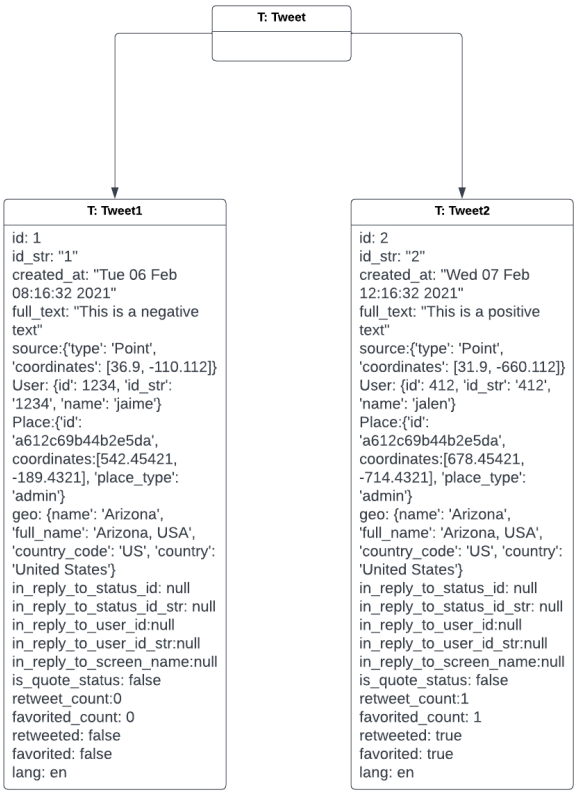


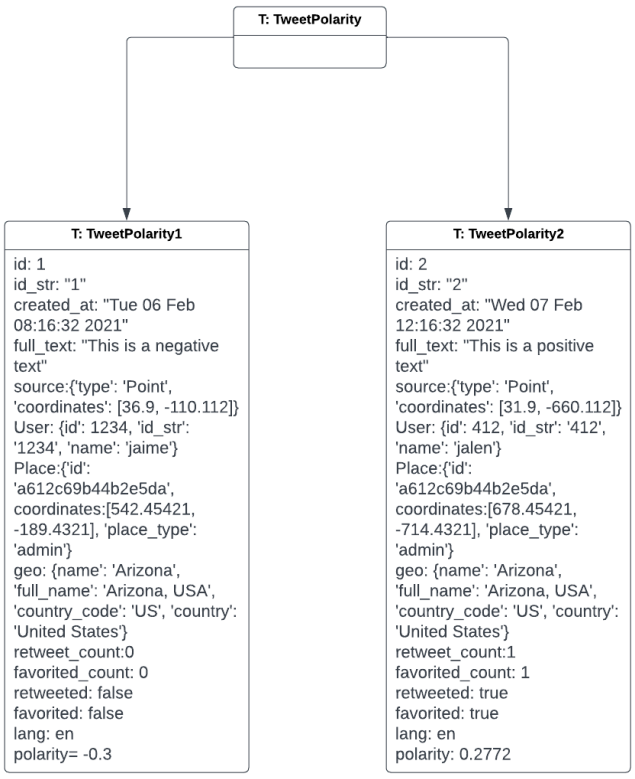
Class Diagrams:





Object Diagrams:





Implementation

Tweet Hydration and JSONL Parsing

The first step in the implementation was to automate the retrieval of dehydrated tweets from our dataset source: https://ieee-dataport.org/open-access/coronavirus-covid-19-geo-tagged-tweets-dataset. This proved difficult as the site requires a user account in order to download datasets and there is no API that can be used to provide access. The dataset we retrieved contained only the tweet ID and tagged sentiment score, as Twitter's recent policy changes do not allow for the public storage of raw tweet data. To retrieve the full tweet data from the tweet ID, the dataset has to undergo a process known as hydration. This required Twitter Developer accounts to gain access to the API keys, which were used with a library called "twarc" that allowed for automated hydration of the tweets.

After the tweets were hydrated, we had to parse the JSONL files produced for each month in order to be added to the database. To parse the JSONL files we made a file called "decoder.py" which contained a function "jsonl_to_json()". This function, when given an input path, parses the JSONL files and returns a list of dictionaries where each dictionary is a tweet in the file.

Database Creation

In the "models.py" file we defined two database models: "Tweet" and "TweetPolarity". These models contained every attribute from the hydrated tweet data. After this, we made a file called "addition.py" in which we developed the functions that used the function "jsonl_to_json()" to parse the JSONL files and add the tweets to the DB. For the "TweetPolarity" model, the polarity attribute was calculated using Textblob's polarity function on the full text to give it a polarity score.

Developing Predictor Models

This is an outline of the development of classifier models. Full details can be found in the Model Evaluation document.

Naïve Bayes Classifier

During the development of the Naive Bayes classifier we randomly selected tweets in our database from different months and manually gave them a sentiment score (positive or negative). Our initial results showed an accuracy of the classification of only 50%. When evaluating the results we noticed that values were mostly negatives. We refined the training and test sets until achieving good results, while evaluating the amount of false positives and negatives. The results showed a sharp increase in accuracy to 80%, with a smaller number of false negatives being presented in the results.

Decision Tree Classifier

We then developed a Decision Tree classifier using the same training and test data. Our initial results showed an accuracy of 69%. When analysing the classification of each test data individually we found that there were many false negatives. We also noticed that the accuracy was not consistent with results ranging from 65-71% accuracy, changing each time we ran our test. We concluded that to achieve a reliable accuracy we would need a much more extensive set of data and testing and Naive Bayes worked better for sentiment prediction.

K-Nearest Neighbour (KNN) Classification

We decided to compare the accuracy of two classifiers and then test a new model using Euclidean Distance for sentiment analysis, using the same training and test sets.We were interested to see how this new approach would perform in predicting the sentiment of text. To predict the sentiment of the text we first had to transform the dictionary into a 2D matrix for the clustering to be able to be performed. Depending on the number of neighbours we defined, the accuracy of the model changed. At the default value of 2 the average accuracy was 54%, when analysing the results we noticed that all of the wrong predictions were false negatives. With a k value at 3 the accuracy decreased to 49%. At k values of 4 the accuracy was 51% and at 5 the accuracy was 46%.

K-means Clustering

The last clustering method we evaluated was K-means clustering. Again, we used the same training and test sets. Like with KNN, to predict the sentiment of the text we first had to transform the dictionary into a 2D matrix to enable clustering to be performed. We used the same training and test sets as our Naive Bayes Classifier model. Our initial tests with a k value of 2 showed an accuracy of 51%, but had results as low as 46%. At k = 3, the accuracy was slightly lower with consistent results of 49% and a minimum of only 2% recorded once. At k=4 accuracy had a max value of 60% but we noted a minimum value of only 5%, meaning that 2 out of the 34 tests were accurately predicted. Analysing the results to examine if there were more false positives or negatives was difficult due to the fact that the categorical labels had been converted to numerical values for the matrix for K-means to be produced. Positive values were transformed into “1” and negative values into “0”. The results contained a large number of false positives.

Built-In Textblob Classifier

Our next step was to compare the accuracy of the models we trained up to now with TextBlob’s “polarity()” function. We measured the accuracy of the function with the same test set used for the other models. The accuracy was 86%, higher than all of the models we trained. For this reason, we decided to use this as our model to graph the sentiments.

Logistic Regression Model

Logistic regression (LR model) is a statistical model used to analyse the relationship between a dependent variable (also known as the response variable) and one or more independent variables (also known as predictor variables). We believed that it would be suitable for sentiment analysis as it is also a binary classification algorithm like Naïve Bayes and Decision Trees. Our research suggests that it performs close to or better than Naïve Bayes for our use case.

Initially, we used a small set of 55 tweets to train our model and 35 test tweets. The results showed an accuracy of 75%. We concluded after some research that this performance was inadequate and suspected the cause was the low amount of training data. We also found that

our training data itself was problematic in the way that it was inherently biassed, being centred around tweets concerning Covid-19. There was likely to be a higher distribution of negative tweets than positive tweets and this was interfering with the model's ability to learn patterns in the data, leading to many false positives and negatives in the confusion matrix. We expanded our training set to contain over 300 tweets, now including tweets from the Sentiment140 dataset that is widely used to train sentiment analysis classifiers. Our new training set was more balanced with almost equal amounts of positive and negative data.

Further Analysis and Update of Training and Test Sets

After developing the logistic regression model we realised that both the test and training sets were too small to achieve high accuracy. We found a publicly available dataset on Kaggle with tagged tweets containing sentiment scores (positive or negative). We collected and added 250 tweets to the training set and 100 tweets to the test set. We determined that a larger test set would demonstrate the performance of the developed models more accurately. We added a total of 250 tweets to the training set for all of the other models too. We ran the tests again with the test set now containing 135 tweets for each model, and the resulting accuracies are as follows:

.

- Naive Bayes Classifier: 83% accuracy - increased by 2% from previous tests.
- Logistic Regression: 85% accuracy - increased by 11% from initial tests.
- Decision Tree Classifier: 77% accuracy - increased by 8% from previous tests
- K-Means Clustering: 50% accuracy - remained the same as in previous tests
- KNN Clustering: 65% accuracy - same accuracy results as in previous tests
- TextBlob's Built-In Classifier: 90% accuracy - increased by 4% from previous tests.

The change in accuracy demonstrated not only the consistency of TextBlob, but it also showed that the other predictors improved from the larger training data. For K-means, the big increase might be because of the increased training set, but we determined that the results were still not reliable enough to make a consistent prediction.

Calculating Sentiment Scores

After comparing the accuracy of various sentiment analysis models, we determined that Textblob's polarity function was the most reliable. To calculate sentiment scores for specific date ranges, we created a series of functions. We developed functions to calculate the sentiment of specific date ranges. We developed a series of functions to calculate the total number of tweets in a month, to calculate the polarity by year, month, week and day and to select a number of tweets in a given date range.

We calculated sentiment scores by month, year, week and key dates. To perform these calculations we developed a series of functions. Firstly, we made a function to calculate the daily polarity in every month called "get_daily_polarity()". First, we developed a function called "get_daily_polarity()" which calculated the daily polarity for each month by iterating over the TweetPolarity table and adding the total polarity for each day. Then we made a function to calculate polarity by month, called "get_all_polarity()". This function calculated the monthly polarities by calling the "get_daily_polarity()" function for each month, and returned a dictionary where the keys were the months calculated and the values were the polarity score. We then made a function to get polarity by week, this function was called "get_polarity_by_week()" and took in a list of dates, iterated over them and calculated the daily polarity for them using the same method as "get_daily_polarity()". The last function we made was "get_polarity_by_key_date()". This function found the daily polarities of a number of key dates which we passed to it by finding the dates in the "TweetPolarity" table and following the same method as "get_daily_polarity()".

Graphing Results

To visualise the sentiment scores, we created barcharts and line plots using the matplotlib library in Python. We made functions to plot the data by all months as a barchart and line plot where each point or bar represents the average polarity in a month. We then made a line plot to view daily polarity scores by month. For tweet count we developed a heatmap, which showed what countries the tweet data was coming from and what amount of tweets came from each country. We developed this using the geotag from the tweet data, and calculating the total number of tweets per country in a month's range. This visualisation helped us understand the distribution of tweets and identify any trends or patterns in the data. Finally,

we created barcharts, confusion matrices and ROC curves to visualise our model data. We made these using the accuracy scores we calculated earlier.

Website Development

We developed 5 different pages, a home page, a daily page, a key dates page, a models page and an about page. We made a view for each of the different HTML pages in “views.py” where we called the previously defined functions to calculate the polarity and graph the data in real time and rendered the information onto the HTML page. On the home page we displayed a barchart and line plot of each month's polarity and a heatmap with a slider to view the changes in frequency by country on a month to month basis. The sliders were made in html and a JS script was added to iterate over each slide and set the current as active and the last option as inactive. We then added CSS to the pages and increased the usability, learnability and appearance as we introduced more features. For example, on the daily page, we included a slider to select which month to view and added a table of data to analyse major changes in sentiment. The key dates page compared the polarity at key dates to the polarity of the week, month, and year the key date occurred. We displayed this information as a bar chart. Lastly, we developed the models page which shows the accuracy of every model in a bar chart, provides information about every model and contains confusion matrices to gain insight into how these models perform.

Testing

We made unit tests for each of our functions as we developed them. These ensured that we did not break any functionality as we refactored code. We developed unit tests using django’s library “django.test.testcase”. We also made integration tests using django’s library, in this case also using the package “Client”. For integration testing we tested the views functions for each page. In these tests a GET is made and the correct html being run with the correct response context is checked. For performance testing we used the developer console on Chromium-based and Firefox browsers. This allowed us to see the timings for various requests and identify areas where we could speed up performance, for example by precalculating graphs and functions where they take up too much load time. Specific testing results can be found in the Performance Testing document.

CI/CD

For our CI/CD process we utilised a cloud hosted server and installed a gitlab runner to handle updates to the repository. Everytime new code is pushed to production the runner will use scripts to teardown the previous instance, update the repository and run tests before starting the django server again.

Sample Code

Some sample code of how we parsed jsonl:

```
def jsonl_to_json(jsonl):
    # returns list of dictionaires - each dictionary is the

    with open(jsonl) as f:
        list_jsonl = list(f)

    return [json.loads(jline) for jline in list_jsonl]
```

Code of database models and functions to add tweets to DB:

```
class TweetPolarity(models.Model):
    id = models.IntegerField(primary_key=True)
    created_at = models.CharField(max_length=200, null=True)
    id_str = models.CharField(max_length=750, null=True)
    full_text = models.CharField(max_length=280, null=True)
    geo = models.CharField(max_length=500, null=True)
    entities = models.CharField(max_length=500, null=True)
    source = models.CharField(max_length=500, null=True)
    retweet_count = models.CharField(max_length=500, null=True)
    favorite_count = models.CharField(max_length=500, null=True)
    favorited = models.CharField(max_length=500, null=True)
    retweeted = models.CharField(max_length=500, null=True)
    polarity = models.CharField(max_length=8, null=True)
    user = models.CharField(max_length=500, null=True)
```

Part of code for logistic regression mode:

```
    processed_train = train_tweet[['text', 'label']]
    processed_test = test_tweet[['text', 'label']]

    # 3. Tokenization, stemming and lemmatization
    tokenizer = RegexpTokenizer(r'\w+')

    train_tweet['text'] = train_tweet['text'].apply(tokenizer.tokenize)
    train_tweet['text'] = train_tweet['text'].apply(lambda x: stemming_on_text(x))
    train_tweet['text'] = train_tweet['text'].apply(lambda x: lemmatizer_on_text(x))

    test_tweet['text'] = test_tweet['text'].apply(tokenizer.tokenize)
    test_tweet['text'] = test_tweet['text'].apply(lambda x: stemming_on_text(x))
    test_tweet['text'] = test_tweet['text'].apply(lambda x: lemmatizer_on_text(x))

    # Train a TF-IDF vectorizer on the training data
    vectorizer = TfidfVectorizer()
    X_train = vectorizer.fit_transform(unprocessed_train['text'])
    y_train = unprocessed_train['label']

    # Transform the test data into feature vectors using the trained vectorizer
    X_test = vectorizer.transform(unprocessed_test['text'])
    y_test = unprocessed_test['label']

    # Can probably train all models with these matrices in the future (if dev time allows)

    xy_train_xy_test = [X_train, y_train, X_test, y_test]

    return [train_tweet, test_tweet, processed_train, processed_test, xy_train_xy_test]
```

Code for KNN:

```
##### K-Nearest Neighbour Clustering ###
vectorizer = CountVectorizer()

trainx_text = []
train_y = []
for data in train: # iterate over training data and add it to list to be transformed to matrix
    train_y.append(data[1]) # score is data[1] which is either 'neg' or 'pos'
    trainx_text.append(data[0]) # tweet text data is [0]

train_x = vectorizer.fit_transform(trainx_text) # transform train data into matrix for clusterin

test_text = []
for data in test: # iterate over test data to add to list which will be transformed to matrix
    test_text.append(data[0]) # add all text values without the pos or neg values to an array

test_x = vectorizer.transform(test_text) # transform test data into matrix for clustering

model = KNeighborsClassifier(n_neighbors=2)
model.fit(train_x, train_y) # train KNN

# Use the model to predict the sentiment of the test data
test_y = model.predict(test_x)
```

Code for Naive Bayes Classification:

```
def get_nbc_accuracy(): # given a list of training data
        cl = NaiveBayesClassifier(train)
        return cl.accuracy(test)
```

Code for some of the graphing functions:

```
def barchart_by_month(dict):
    labels = list(dict.keys())
    sizes = dict.values()

    colors = np.random.rand(len(labels), 3) # make random colours for each bar of the bar chart
    fig, ax = plt.subplots(figsize=(10,10))
    ax.bar(labels, sizes,color=colors, align="center")

    ax.set_xticks(np.arange(len(labels)))

    ax.set_xticklabels(["Feb 2020", "Mar 2020", "Apr 2020", "May 2020", "Jun 2020", "Jul 2020", "Aug 2020",
              "Sep 2020", "Oct 2020", "Nov 2020", "Dec 2020", "Jan 2021", "Feb 2021", "Mar 2021",
              "Apr 2021", "May 2021", "Jun 2021", "Jul 2021", "Aug 2021", "Sep 2021", "Oct 2021",
              "Nov 2021", "Dec 2021", "Jan 2022", "Feb 2022", "Mar 2022", "Apr 2022", "May 2022",
              "Jun 2022", "Jul 2022", "Aug 2022", "Sep 2022", "Oct 2022", "Nov 2022", "Dec 2022"])

    plt.setp(ax.get_xticklabels(), rotation=45, ha="right", rotation_mode="anchor")

    ax.set_title("Sentiment Score by Month")
    ax.set_xlabel("Date")
    ax.set_ylabel("Sentiment Score")

    handles = []
    for i, label in enumerate(ax.get_xticklabels()):
        handles.append(ax.bar(0, 0, color=colors[i], label=label.get_text())[0])
    ax.legend(handles, labels, title="Date", loc="upper left", fontsize="small")

    mpld3.save_html(fig, "All Barchart Monthly")
```

```
def lineplot_by_month(dict):

    labels = list(dict.keys())
    sizes = dict.values()

    fig, ax = plt.subplots(figsize=(6, 6))

    ax.plot(labels, sizes)

    ax.set_xticks(np.arange(len(labels)))

    ax.set_xticklabels(["Mar 2020", "Apr 2020", "May 2020", "Jun 2020", "Jul 2020", "Aug 2020",
              "Sep 2020", "Oct 2020", "Nov 2020", "Dec 2020", "Jan 2021", "Feb 2021", "Mar 2021",
              "Apr 2021", "May 2021", "Jun 2021", "Jul 2021", "Aug 2021", "Sep 2021", "Oct 2021",
              "Nov 2021", "Dec 2021", "Jan 2022", "Feb 2022", "Mar 2022", "Apr 2022", "May 2022",
              "Jun 2022", "Jul 2022", "Aug 2022", "Sep 2022", "Oct 2022", "Nov 2022", "Dec 2022"])

    plt.setp(ax.get_xticklabels(), rotation=45, ha="right", rotation_mode="anchor")

    ax.set_title("Sentiment Score by Month")
    ax.set_xlabel("Date")
    ax.set_ylabel("Sentiment Score")

    mpld3.save_html(fig, "All Monthly Sentiment Lineplot")
    return mpld3.fig_to_html(fig)
```

Code of some functions from views.py:

```
def daily(request):
    plots = get_monthly_html_files("./PandemicAnalyser/templates/monthly")
    return render(request, 'daily.html', {
        'Mar2020': plots[0], 'Apr2020': plots[1], 'May2020': plots[2], "Jun2020": plots[3],
        "Jul2020": plots[4],
        "Aug2020": plots[5], "Sep2020": plots[6], "Oct2020": plots[7], "Nov2020": plots[8], "Dec2020": plots[9],
        "Jan2021": plots[10],
        "Feb2021": plots[11], "Mar2021": plots[12], "Apr2021": plots[13], "May2021": plots[14], "Jun2021": plots[15],
        "Jul2021": plots[16],
        "Aug2021": plots[17], "Sep2021": plots[18], "Oct2021": plots[19], "Nov2021": plots[20], "Dec2021": plots[21],
        "Jan2022": plots[22],
        "Feb2022": plots[23], "Mar2022": plots[24], "Apr2022": plots[25], "May2022": plots[26], "Jun2022": plots[27],
        "Jul2022": plots[28],
        "Aug2022": plots[29], "Sep2022": plots[30], "Oct2022": plots[31], "Nov2022": plots[32] #, "Dec2022": plots[33]
    })
```

```
def index(request):

    yplots = get_yearly_html_files("./PandemicAnalyser/templates/yearly")

    #hmaps = get_monthly_heatmap() # - save html files locally to increase speed
    hmaps = get_html_heatmap("./PandemicAnalyser/templates/heatmaps")

    return render(request, 'index.html', {#'plot': fig_plot, 'piechart':fig_piechart,
                                          'barchart': yplots[0], 'lineplot': yplots[1],
                                          'Mar2020': hmaps[0], 'Apr2020': hmaps[1],
                                        'May2020': hmaps[2], 'Jun2020': hmaps[3], 'Jul2020': hmaps[4],
                                        "Aug2020": hmaps[5], "Sep2020": hmaps[6], "Oct2020": hmaps[7], "Nov2020": hmaps[8], "Dec2020": hmaps[9],
                                        "Jan2021": hmaps[10],
                                        "Feb2021": hmaps[11], "Mar2021": hmaps[12], "Apr2021": hmaps[13], "May2021": hmaps[14], "Jun2021": hmaps[15],
                                        "Jul2021": hmaps[16],
                                        "Aug2021": hmaps[17], "Sep2021": hmaps[18], "Oct2021": hmaps[19], "Nov2021": hmaps[20], "Dec2021": hmaps[21],
                                        "Jan2022": hmaps[22],
                                        "Feb2022": hmaps[23], "Mar2022": hmaps[24], "Apr2022": hmaps[25], "May2022": hmaps[26], "Jun2022": hmaps[27],
                                        "Jul2022": hmaps[28],
                                        "Aug2022": hmaps[29], "Sep2022": hmaps[30], "Oct2022": hmaps[31], "Nov2022": hmaps[32], "Dec2022": hmaps[33]})
```

Code of some of the unit tests:

```
def test_add_tweet_polarity(self):
        add_tweet_polarity(self.test_path)
        tweet_count = TweetPolarity.objects.filter(id=123).count()
        self.assertEqual(tweet_count, 1)

        # Test that a tweet is added to the db
        add_tweet_polarity(self.test_path)
        tweet_count = TweetPolarity.objects.filter(id=123).count()
        self.assertEqual(tweet_count, 1)

        # Test the polarity is correct
        added_tweets = TweetPolarity.objects.filter(id=123)
        for tweet in added_tweets:
            self.assertEqual(tweet.polarity, '0.0')  # expected polarity for test tweet
```

Code of integration tests:

```
Jaime
@patch("PandemicAnalyser.Graphs.barchart.barchart_models")
@patch("PandemicAnalyser.Predictor.logisticreg.get_lr_cm")
def test_models_view(self, mock_barchart_models, mock_get_lr_cm):
    mock_barchart_models.return_value = "<p> Test<p>"
    mock_get_lr_cm.return_value = "<p> Test 2<p>"

    response = self.client.get('/models/')

    # Check that page loads correctly
    self.assertEqual(response.status_code, 200)

    # Check that models.html is used
    self.assertTemplateUsed(response, 'models.html')

    # Check that the barchart and confusion matrix are in the response context
    self.assertIn('barchart', response.context)
    self.assertIn('confusionmatrix', response.context)
```

Problems Encountered/Solved

Dataset Retrieval:

We found it difficult to automate the dataset retrieval process, due to the necessity of having to log-in to the source website. We were able to circumvent this problem by using a python library called “twillio”. This enabled us to virtualise a browser-like entity that can navigate through the site, input credentials and retrieve the download links for the dataset.

Hydration:

Due to the large size of the dataset (~400,000 tweets), it took 3 - 4hrs to hydrate them all. We mitigated this problem somewhat by dropping tweets that have an associated sentiment label of 0 (indicating a neutral tweet) and by saving the hydrated tweets into JSONL files.

Data Processing:

The size of the dataset made it very difficult to process data in an efficient manner. We found that performing real time operations on the SQLite database often took tens of minutes. To save time, we attempted to use pandas dataframes whenever possible as we found the performance to be much better.

Databases:

Initially, we planned to use a document type NOSQL database like MongoDB to store our tweet dataset, as it would give us greater flexibility and improved performance over SQL-based databases due to the size and type of our data. We discovered only when we got to the database construction stage, that the framework we planned to use, PyMongo, was not officially supported by Django. Django has no official support for any NOSQL database. This led us to being forced to use SQLite, which had performance that was far too slow to enable any user interactivity in real-time, such as ranged sliders for the graphs in order to adjust the timeframe for which they are displayed.

Interactive Graphs:

Due to the slowness in database response, we were unable to implement some of the interactive features mentioned in the functional spec such as a ranged slider for the dates.

Instead in order to significantly boost performance, we precalculated the graphs and stored them locally.

Model Performance:

We found that none of the standard classifiers could not quite match the performance of TextBlob's built-in polarity function. This meant we had to use this polarity score, instead of any of our trained classifiers for our sentiment calculations in order to get the most accurate results.

Webapp Self-hosting:

We had initially planned to host the website on a server we had built from spare parts, but we found that there were several issues with this. Firstly, there were the security concerns that came from allowing incoming connections to our home network that bypass any firewall. We would need to expend significant time and resources to make this connection secure. There were also performance concerns as while our ISP offers gigabit download speeds, upload speeds are restricted to only 50 mbit/s, which would become a huge bottleneck for our server. The only solution was to move to a cloud-hosted service.

Model Training:

We found that training our models using only the covid tweets provided lower accuracy than desired. We were able to significantly improve the precision, recall and F1-score of our models by adding tweets from the sentiment140 Kaggle dataset, which is specifically designed for researchers in the field of sentiment analysis.

Dataset Updates:

Recently, Elon Musk acquired Twitter and made many changes that negatively impacted developers such as us. One such change was the removal of the "academic research" tier, which we had previously applied to and were granted permission to use. This tier enabled us to hydrate and look up up to two million tweets per month, which was sufficient for our ~400,000 tweet dataset.

The changes Twitter have made now make it impossible to use tweet lookup for free tier users. This can now only be done by purchasing the prohibitively expensive ($100 per month) basic tier, which still only allows for retrieval of 10,000 tweets per month

(https://developer.twitter.com/en/portal/products/basic). As a result, it will be unfeasible for us to update the dataset weekly or monthly as previously planned.

Results

The goal of our project was to research various machine learning classifier models and assess their performance in the task of sentiment analysis. Using this data, we planned to make a data analysis platform that evaluates sentiment on tweets concerning the Covid-19 pandemic.

- Naive Bayes Classifier: **83**% accuracy
- Logistic Regression: **85**% accuracy
- Decision Tree Classifier: **77**% accuracy
- K-Means Clustering: **50**% accuracy
- KNN Clustering: **65**% accuracy
- TextBlob Polarity Function: **90%** accuracy

From our research, we have discovered that for our specific dataset, the most accurate standard classification model is Logistic Regression. We have identified a number of reasons for why this is the case. Logistic Regression is a linear model that can be well-suited for datasets where the relationship between the features and the target variable is approximately linear. This was the case for our dataset, which could explain why Logistic Regression outperformed other models like Decision Tree Classifier or K-Means Clustering, which are less suited for capturing non-linear relationships.

Logistic Regression is a commonly used model for binary classification tasks, which includes sentiment analysis. It works by modelling the probability of a tweet belonging to a particular sentiment class (e.g., positive or negative). This may have allowed Logistic Regression to better capture the sentiment information in the tweets compared to other models that are not specifically designed for binary classification. The performance of a model can be highly dependent on the features used for training. It's possible that the feature selection or feature engineering process used for our dataset was better suited for Logistic Regression, which could have contributed to its higher accuracy. It's also possible that since we did not optimise the hyperparameters used for training each specific model, Logistic Regression happened to perform better with the default hyperparameters.

If the optimization of hyperparameters was possible within the time constraints we faced, it may have been possible for the Logistic Regression model to potentially match or outperform

the TextBlob polarity function, as the difference in accuracy between them was only 5%. The polarity function works by first tokenizing the input text into individual words, and then using a pre-trained machine learning model to classify each word as either positive, negative, or neutral. From our research, it was unclear exactly which classification model is used by the polarity function. The algorithm then assigns a polarity score to the text based on the proportion of positive and negative words. The polarity score ranges from -1 (completely negative) to +1 (completely positive), with 0 indicating a neutral sentiment.

The TextBlob polarity function also takes into account the intensity of the sentiment expressed in the text. For example, a text with a higher frequency of strongly positive or negative words will receive a higher polarity score than one with mostly neutral words. This capability, and the optimization of hyperparameters are likely major contributors to why the polarity function showed the best performance.

Future Work

We have identified several areas of improvement for our project that could be addressed in the future.

Model Optimization:
In the future, we could look into optimising the hyperparameters of each model to achieve even greater accuracy. Additionally, we could also add more training and test data.

Accounting for Emojis:
Tweets often contain emojis from which valuable sentiment information can be gleaned. In the future, we could account for this by weighting the emoji in conjunction with the text based on the text ID of the emoji (e.g. 🙂 = :slightly_smiling_face: weight: + 15% positivity).

Accounting for Gifs:

Gifs are another way commonly used to express sentiment found in tweets. While it would be difficult and complex to use computer vision to try and analyse a gif, usually just the filename of the gif is enough of a descriptor.