Development of a tool for the emotional interpretation of text using sentiment analysis.

**Prasharn Vitnu Selvaranjan**

**19 April 2023**

**Word Count -**

**BSc Data Science and Computing Project Report, Birkbeck College, University of London, 2023**

*This report is the result of my own work except where explicitly stated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.*

**Abstract:**

This project report outlines the process of developing an application for the sentiment analysis of tweet text. A brief description of sentiment analysis is given, and its uses are outlined. To develop this application, principles of data science and computing are utilised. The result is the development of an accurate sentiment classifier (with a validation accuracy of above 0.750), comprehensive APIs, and a usable application user interface (UI). The process by which the dataset was pre-processed, and the choice of logistic regression algorithm is discussed. The report features an explanation of API tools used as well as design choices made regarding the UI and its development. Project development and decisions are evaluated and further potential improvements on how to produce a better classifier are discussed. Furthermore, the potential flaws in the API and how these can be identified and rectified are explained. A brief discussion on UI improvement is also included.

Table of contents:

1. Introduction

2. Aims, Objectives, and Design

3. Development

* Classifier
* API (Application Programming Interface)
* UI (User Interface)
* Testing

4. Analysis

5. Conclusion

6. References

8. Bibliography

**Introduction:**

When discussing the wellness, attitude, or sentiment of an individual, although easy for some, it can feel almost impossible for others. Unless explicitly stated, there is much depth to understanding the attitudes of another. To help in understanding, interpretation can be broken down into three dimensions [1]:

Affective – The individuals’ feelings towards the topic.

Example: Donald Duck is annoying.

Behavioural – Actions and how the attitude is expressed.

Example: I punched Donald Duck.

Cognitive - Values, Beliefs, and Ideas.

Example: Donald Duck is unclean.

Despite the differing wording and background, the reader can infer that the individual’s sentiment towards Donald Duck is negative in all three statements. How is it that the reader can tell? If unimportant and connective words are removed, all that is left in the statements are the topic (Donald Duck) and words implying a sentiment (annoying, punched, unclean). The reader then infers that the sentiment attached to the word is how the individual feels. In essence, this is how a sentiment analysis algorithm operates. Sentiment analysis utilises machine learning and natural language processing to extract and classify subjective information.

To develop such an algorithm for sentiment analysis, the model would require vast amounts of training data to learn the importance of keywords (e.g., annoying, punched, unclean). Ideally, the training data should be of a similar format and length as the planned input.

Twitter is a popular social media platform that provides a space for individuals and organisations to express their thoughts and opinions. It also has a character limit of 280 with most tweets being written in the Roman alphabet. This makes it an excellent source for training data and application inputs.

The implementation of sentiment analysis on twitter allows us to better understand how individuals feel about a subject by analysing whether the tweets are mostly positive or negative. For example, if a beverage company released a new drink and most of the tweets relating to it had a negative sentiment as well as mentioning the colour of the drink. It could be assumed that the public do not like the colour of the new drink.

Furthermore, sentiment analysis can and has been used to analyse or follow the public opinion of topics. A good example of this would be the tracking of public opinion of candidates by political campaigns.

Similarly, sentiment analysis can be used on tweets relating to a certain topic to study how sentiment has changed in retrospect. For example, researchers could track the sentiment around a product over the last 10 years, observing the trends and impact of previous marketing campaigns.

In customer service, a sentiment analysis tool would allow businesses to respond to issues more quickly and efficiently; quickly identifying if there is a system failure by observing the current sentiment around one of their products.

In practice, this could be used to find the sentiment of the whole document or smaller sections such as chapters, paragraphs, or sentences. For example, this could be used to identify the specific emotions conveyed in each statement within an item review or more simply whether it is positive or negative.

**Aims, Objectives, and Design:**

To develop a tool that can be used to classify tweets by sentiment comprising of a classifier, UI, and the appropriate APIs. It should adhere to the following requirements:

**Functional requirements:**

The application must be able to accurately classify tweets into either positive or negative categories of sentiment with an accuracy of at least 0.70.

The application must be able to collect tweets using Twitter’s API.

The application must be able to take usernames, hashtags, and sentences as inputs.

The application should appropriately visualise the results of the analysis.

**Non-Functional requirements:**

The application must have an accuracy above 0.50 when classifying tweets into the positive and negative categories.

The application must uphold good data security practices, protecting against security threats and attacks.

The application must protect against GDPR and similar breaches by ensuring that personal data or identifiable information is not disclosed or stored without consent.

The application should be accessible and usable through a well-designed user interface.

The application should be available when needed with minimal disruptions.

The application should operate with quickly with minimal response delays.

**Languages and technology used:**

Python was used for its versatility, ease of use and many useful libraries. Flask for its smaller learning curve and React JS as it allows for more efficient UI building.

**Diagrams: Use case, class/general model/sequenceDiagram

Description automatically generated**

**Use case diagrams, Class diagrams:**

**Development: Classifier**

**Sourcing Data:**

Using Kaggle and similar platforms, three potential datasets were found. Potential-dataset-1 [2] featured 7776 lines of tweets. These tweets were labelled with specific emotions (anger, fear, joy, sadness, surprise) rather than positive and negative. Potential-dataset-2 and 3 featured tweet information as well as binary classifications of sentiment (Positive or Negative). However, after exploration, it was found that Potential-dataset-2 [3] had incorrectly labelled target values. Furthermore, it was also only 31,963 records long compared to the bigger Potential-dataset-3 (1,048,575 entries). Potential-dataset-3 [4] was a more balanced dataset, featuring an equal number of positive and negative tweets.

Potential-dataset-3 was picked over Potential-dataset-2 due to its larger and correctly labelled data. Potential-dataset-1 was not used – the six potential classifications and small dataset would make it difficult to guarantee accuracy and for my proposed application, high accuracy is desired. A simple positive or negative response is a sufficient classification for the uses previously outlined.

**Data exploration and cleaning:**

Before pre-processing, it is important to understand the true dimensions of the dataset, as well as what columns of information are contained. Through this exploration, the following changes were made:

To ensure the model was not biased towards certain tweets, duplicate values were removed using pandas.dataFrame.drop\_duplicates. This removed 18534 records from the dataset. Furthermore, it was found that more of the removed entries were labelled 0 (Negative) so the data was balanced by dropping randomly selected Positive data. Finally, the columns at indexes 1, 2, 3, and 4 were removed as they do not feature information used in the algorithm. This was done using pandas.dataFrame.drop.

**Pre-processing:**

As part of pre-processing, an iterative approach was used on the dataset to both simplify the data and to ready it for modelling. The steps taken are as follows:

Initially, the data was made lowercase as part of pre-processing. This can help to reduce the dimensionality of the data. This standardisation of the text makes it more consistent. This can be beneficial in sentiment analysis as it reduces the variation in text cases. However, capitalisation can be important in understanding sentiment. Upper case letters can be used to emphasize certain words, ultimately changing the overall sentiment of the sentence. Proper nouns would also be lost, impacting the sentiment of the text. An example of this could be the sentence “Happy Feet is okay. It could be better”. The sentence references the movie “Happy Feet”, stating that it is okay and could be improved. With only lower-case letters used in the algorithm, this sentence is more likely to be identified as a Positive statement.

To identify the ideal dataset characteristics, several versions of the dataset were produced to be developed into models. The different datasets vary in whether they featured punctuation and in whether they contained both upper and lower-case letters. These experiments identified that all tweet information should be made lowercase.

URLs are links to websites and do not imply a sentiment. Hence, they have been removed in pre-processing to prevent their impact to the sentiment deciding algorithm. This was done using the regex expression re.sub('((www.[^s]+)|(https?://[^s]+))',' ',dataFrame). The ‘www.’ matches the beginning of a URL. Similarly, ‘https?://’ matches the strings ‘http:// or ‘https://’. The logical OR operation ‘|’ is then used. ‘[^s]+’ matches any following characters that are not whitespace. The matches are substituted with a blank space.

As the dataset features tweets, it was expected that the data will feature @usernames. As @usernames do not have a sentiment and can obscure the learning process, they were removed. To remove these, the code re.sub(r"@\w+\b",' ',dataset) was used. This regex expression matches the ‘@’ symbol and any letters that follow it, substituting them with a blank space.

The most used words such as connective words generally do not carry a significant meaning on their own. These are called stop words and their removal improves accuracy and efficiency by reducing the size of the dataset. The selection of these words is straightforward and can be done using the nltk.corpus.stopwords module. The additional words: rt, rts and retweet were included as these terms appear regularly, but do not impact the sentiment of the tweet. To remove the stop words, the string(tweet) is split into a list of words. A new list is made by appending the word in the list that are not found in the stop words. The ‘join’, reverses the split, making the full sentence once again but with the stop words removed.

Like case-sensitivity, punctuation can be indicative of sentiment and its removal should be carefully considered. Exclamation and question marks can be deeply associated with the intensity of a sentiment. Take the example “I love Shakespeare!”. The exclamation mark makes the statement more strongly positive. Consequently, several test datasets were produced with differing punctuation constraints. Through this experimentation, it was found that the best performing datasets featured no punctuation, hence the dataset with no upper-case letters or punctuation was used.

Repeat characters were removed from the dataset. This was to shorten elongated words into their simplest form and to allow for the creation of a more precise and concentrated dictionary. For example, ‘Loooove’ would be shortened to ‘love’. This would also mean that words spelt with double letters would also be shortened, rendering them nonsensical. For example, ‘happy’ would be simplified to ‘hapy’. Double letters are mostly used to assist in the pronunciation of words so, their simplification should not impact sentiment. However, it is noteworthy that certain words such as ‘hoop’ and ‘teen’ would be simplified to ‘hop’ and ‘ten’ which are words with very different meanings.

Numbers do not imply sentiment; therefore, they were removed from the tweet column. This was also done with the regular expression module using re.sub('[0-9]+', '', dataset). This code removes one or more numeric values, replacing them with an empty string.

To simplify the tweet text further, stemming was used. This was used instead of lemmatization as it is faster and still preserves the root of the word, helping to produce a smaller and more concise dictionary of tweeted words. To do this, an instance of the porter stemmer is created and applied to the words in the dataset, returning a list of the words from the dataset but, in their stemmed root form. It is worth noting that there are some drawbacks to stemming, such as the loss of context and more ambiguity.

The tweet column was tokenised, breaking it down into chunks which are smaller and more manageable. This is also a necessary step before vectorising the tweet data. This was done using the ‘tokenize’ method from the class ‘RegexpTokenizer’. As it was used with the expression (r'\w+'), it matches consecutive alphanumeric characters and tokenises them.

TF-IDF vectorisation was used. There are several benefits to this. Using this method, it is possible to identify the most important features of the tweets and remove any unimportant phrases. TF-IDF also allows for the comparison of texts of varying lengths.

TF-IDF (Term Frequency – Inverse Document Frequency) is a statistic that represents the importance of a word in a document within a larger collection of documents. It is calculated by finding the Term frequency (the frequency of the term in the document) and dividing it by the inverse document frequency (the commonness of the word in the whole corpus).

TF = Number of times term t appears in document d / Total number of terms in document d

IDF = log(Total number of documents / Number of documents with term t in it)

TF-IDF = TF \* IDF

= (Number of times term t appears in document d / Total number of terms in document d) \* (log(Total number of documents / Number of documents with term t in it))

A TF-IDF score is then returned for each term. The higher the score, the more important they should be to the document. This is because they occur regularly in the document but less regularly in the whole corpus.

When vectorising using TFIDF, there were some hyperparameters specified. These were:

‘ngram\_range’ – This determines the size of potential ngrams of the text. For example, if ngram\_range = (1,1). Only single words otherwise known as unigrams would be considered. Likewise, ngram\_range = (1,2) would consider couples of words, commonly known as bigrams.

‘max\_df’ determines the maximum frequency of terms considered. For example, max\_df = 0.05 would only consider values in the bottom 5% of the corpus. Similarly, ‘min\_df’ determines the minimum frequency of terms considered. For example, min\_df = 0.01 would only consider values above the bottom 1% of the corpus. The two hyperparameters together would therefore mean that only words found between the bottom first and fifth percentiles are considered.

**Hyperparameters and the use of the grid search:**

Once the training data had been pre-processed, the next step was to find the optimal classification algorithm and hyperparameters. This process can be repetitive and tedious so to speed up the process, a grid search was used. A grid search can assist in finding the best combination of predefined hyperparameters systematically, producing a more accurate classifier. However, a drawback is that the grid search can be computationally expensive and requires powerful hardware to be run on larger datasets with many hyperparameters. Another drawback is that the grid search only explores combinations of predefined hyperparameters; an attribute which is unhelpful when dealing with continuous hyperparameters such as the parameter C used in SVM model training. Like with most tuning practices, there is a risk of overfitting the model to the training data although, the use of validation data allows for the assessment of this and adjustment of the hyperparameters accordingly, minimising its affects.

In addition to the initial train test split for model training and accuracy, the x\_train and y\_train data was further split to produce a validation set. This is particularly important when tuning hyperparameters. This is because it allows for the evaluation of the model’s performance using unseen data, assisting in the identification of overfitted algorithms. The photo below depicts this.

A picture containing timeline

Description automatically generated

A diagram visualising the two train/test splits. [5]

By a process of trial and improvement, it was found that the most optimal TFIDF vectoriser hyperparameters were (ngram\_range=(1,1), max\_df = 0.05 , min\_df= 1). This reduced the dimensions of the vectorised dataset massively allowing for the more efficient training of models. Vectorisation creates a sparse matrix which has greater dimensionality than the original dataset. To make the matrix dense and to increase the efficiency of the model training, the dimensionality reduction technique ‘TruncatedSVD’ was used. By using this, the feature size was selected and reduced to 50.

Initially, an attempt at model development using the whole dataset was made. However, due to the large size of the dataset and the many features after TFIDF vectorisation, this approach was abandoned until better hardware was made available. The dataset was randomly sampled to 10,000 records to make experimental model training more efficient. To ensure that the model testing is as unbiased as possible, Stratified k-fold cross-validation was used.

There are several appropriate models when developing an algorithm for the binary classification of text. Several experimental models were tested with differing hyperparameters. A brief description of the algorithms, screenshots of the hyperparameters, and results are given below:

Support Vector Machines (SVM) find a hyperplane that separates the data points into different classes. A well-tuned SVM will find a hyperplane that divides the classes with a large margin whilst keeping incorrectly classed data points to a minimum. As linear SVM is being used, the hyperparameter we are tuning is the ‘C’ parameter. The larger ‘C’ is the narrower margin. In contrast, a smaller ‘C’ value would produce a wider margin. The drawback of an excessively large margin is that the model would misclassify more points with the hyperplane and could underfit the model. Too small a margin and the model might become overfitted to the training data. Generally, SVM is a good multi-class classifier which performs well with datasets containing outliers. However, as the dataset being modelled is very large and the goal is a binary classifier, SVM was not expected to produce the most optimal results.

Using the random subset of 10,000 records with 50 TFIDF vectorised features, a model with a training accuracy of 0.654 was produced using the following hyperparameters. As the validation accuracy was similar at 0.650, the model had not been overfitted to the dataset.

A screenshot of a computer

Description automatically generated with low confidence

Text

Description automatically generated

The ‘linear’ kernel was used due to its efficiency. This kernel assumes that the data is linearly separable. The ‘poly’ kernel separates the decision boundary non-linearly. The hyperparameters ‘degree’ and ‘gamma’ were defined. ‘degree’ defines the order of the polynomial function. A higher degree will allow the SVM to capture the more intricate relationships within the data but at the risk of overfitting. In contrast, a lower degree will reduce the risk of overfitting but will produce a simpler more linear decision boundary; losing some of the relationships present in the highly dimensional dataset. The hyperparameter ‘gamma’ defines the size and shape of the decision boundary. ‘gamma’: [‘scale’] uses the standard deviation of the features to ensure that boundary is properly balanced between all features. This can be useful when dealing with features of different scales. ‘gamma’: [‘auto’] computes the gamma value as 1/number\_of\_features. Hence, models trained with an ‘auto’ SVM will be biased towards more common features. It is worth noting that this should not be an issue when dealing with larger datasets. When running the same dataset using a ‘poly’ type kernel the following results were yielded:

Text

Description automatically generated

Text

Description automatically generated

As shown, the ‘poly’ kernel produced a slightly better training accuracy of 0.651 and a validation accuracy of 0.644. This was a relatively good accuracy and was not overfitted; however, it did not fulfil the accuracy requirement outlined. Hence, other models were explored.

Decision Trees is a simple algorithm that recursively splits the data at nodes. Each node acts as a test case for an attribute, eventually dividing the dataset into a collection of classifying branches. Like SVM, decision trees are resistant to outliers. However, without pruning they are prone to overfitting. Two hyperparameters were defined. These were ‘min\_samples\_leaf’ and ‘max\_depth’. ‘min\_samples\_leaf’ defines the minimum size of a leaf node. This hyperparameter can help prevent overfitting by producing a more generalised decision tree. Setting this value too low will overfit the model as it produces a large tree with too many leaf nodes specific to the training data. Setting this value too high will produce an underfitted small tree which performs poorly. ‘max\_depth’ defines the maximum distance between the root of the tree to the leaf nodes. A greater ‘max\_depth’ produces a tree with a longer decision path. Careful selection of this value can help express some of the more complex patterns in the dataset, however, there is risk of overfitting.

An initial attempt at fitting the decision tree was made using the below hyperparameters. As shown, the best performing model of those tested had the hyperparameters ‘max\_depth’ = 18 and ‘min\_samples\_leaf’: 30. They produced a high training accuracy of 0.740. This met our functional requirement of an accuracy above 0.70 however, the validation accuracy was only 0.599. The accuracy difference was therefore 0.141 which suggested that the model had been overfitted.

A screenshot of a computer

Description automatically generated with medium confidence

Text

Description automatically generated

A further two attempts were made using decision trees. In the first reattempt, changes were primarily made to the ‘max\_depth’ parameter. This is due to the best performing hyperparameter being the lowest of the tested options. The best performing model also had a ‘min\_samples\_leaf’ value at the upper end of those tested. The same results were given as to those found in the initial attempt. These were a training accuracy of 0.740 and a validation accuracy of 0.599. This suggests that the values selected were not comprehensive enough and should be changed to reduce the overfitting.

A screenshot of a computer

Description automatically generated with medium confidence

Text

Description automatically generated

In the second reattempt, the ‘min\_samples\_leaf’ values were adjusted to cover a greater range whereas the ‘max\_depth’ hyperparameter was reduced to much lower numbers below 10. As expected, this lowered the training accuracy (0.675) to a value closer to that of the validation accuracy (0.643). Further adjustments could be made to make these values more similar and further reduce the overfitting however as the validation accuracy was still relatively low and worse than the SVM performance, further attempts at modelling using other algorithms were made.

Text

Description automatically generated

Text

Description automatically generated

Random Forest algorithms combine multiple decision trees to improve classification performance. As each decision tree within the random forest is trained on a randomly selected subset of the data, the models produced are generally more resistant to overfitting. Like the previous two algorithms tested, they perform well against outliers. Due to the several decision trees used when training the model, they handle larger datasets well. This, however, can be computationally demanding. To mitigate this and allow for hyperparameter testing, the model was trained on the same smaller subset of the data (approximately 10,000 records after random sampling) used previously. Had performance been exceptional, a model would’ve been trained on a larger dataset later. The hyperparameters being explored are those of decision trees i.e., ‘min\_sample\_leaf’ and ‘max\_depth’.

Initial attempts at training a random forest classifier produced a high training accuracy of 0.878 but a validation accuracy of only 0.662. The accuracy difference was 0.216, which suggested that the model had been overfitted to the dataset. The ‘max\_depth’ hyperparameter that performed best was that at the bottom of the range chosen (75). Hence, a lower range of values were later tested. The ‘min\_sample\_leaf’ value that performed the most well was 11 but as the ‘max\_depth’ was lowered, it was expected that the minimum samples of each leaf would increase. Hence, a range including 11 as well as greater ‘min\_samples\_leaf’ values were used in the next grid search experiment.

A screenshot of a computer

Description automatically generated with medium confidence

Text

Description automatically generated

As shown by the large accuracy difference of 0.216, the changes to the hyperparameters did not reduce overfitting in this attempt.

Text

Description automatically generated

Text

Description automatically generated

Smaller ‘max\_depth’ values were then used; however, this yielded the same results with a training accuracy of 0.800 and a validation accuracy of 0.661. This test suggested that the best hyperparameters were ‘max\_depth’ : 10, ‘min\_samples\_leaf’: 12. Despite the accuracy difference needing further investigation, this algorithm performed better than both SVM and Decision Trees but doesn’t meet our functional requirement of an accuracy above 0.70. It is also computationally demanding so is difficult to experiment with further. Nonetheless, Random Forest performed relatively well and could produce a powerful classifier.

Text

Description automatically generated

Text

Description automatically generated

The Naïve Bayes algorithm is particularly powerful when tasked with binary classification tasks. Particularly when dealing with text data. This makes it an ideal algorithm to classify the sentiment of tweets. It isn’t computationally demanding so could utilise the large and highly dimensional dataset available. Conversely, it also performs well with small datasets so should yield good results during the smaller experimental grid search. A downfall of the algorithm is its strong reliance on independence. This means that it might not capture relationships between the many different features of the vectorised dataset. This also makes the algorithm prone to misclassification when dealing with outliers or unseen data. The Naïve Bayes hyperparameter ‘var\_smoothing’ helps to prevent some of these issues by ensuring that all features are used in the classification decision, making for a more generalisable model. Naïve Bayes is based on Bayes’ Theorem.

The initial attempt at training the model using Naïve Bayes produced good results. The training accuracy of 0.620 and validation accuracy of 0.610 were very similar with an accuracy difference of only 0.010. Hence, we could assume that the model had not been overfitted to the dataset. The grid search stated that the best value for the hyperparameter ‘var\_smoothing’ was 1.

Text

Description automatically generated

Text

Description automatically generated

In the second experimental test of Naïve Bayes, more values around the value 1 for the hyperparameter were used. This produced the same results, stating that the best value for ‘var\_smoothing’ was still 1. As the accuracy of 0.610 was well below 0.70 and not as accurate as the result of the polynomial SVM classifier or Random Forest, it was decided that Naïve Bayes would not be the algorithm used to develop the classifier.

A screenshot of a computer

Description automatically generated with medium confidence

Text

Description automatically generated

Logistic regression can be a simple, yet powerful algorithm for binary classification. The features of the dataset are mapped onto the sigmoid function by giving each feature a value between 1 and 0. These numbers corresponding to each of the two classifications. During training, the weighted sums are adjusted to minimise the error of the target label and the prediction using a loss function. When testing the function, the weighted sum of the input features is then passed into the mapped sigmoid function. This will produce a value between 1 and 0. All values above 0.5 will be classed as 1 and all values below 0.5 will be classed as 0. This is an efficient algorithm for binary classification; however, it does assume that feature relationships are linear. This can make this algorithm vulnerable to outliers and multicollinearity.

There were two hyperparameters that were adjusted. The first was ‘penalty’. This is the regularisation parameter and has the potential to reduce overfitting by regulating the impact of each feature. The selected value ‘l2’ is known as Ridge Regression and adds the squared value of the coefficients when calculating the loss function. This reduces overfitting by moderating the impact of individual features, subsequently, producing a more even distribution of the coefficients with smaller values. The second hyperparameter ‘C’ defines the strength of this regularisation; smaller values producing more regularised results but with greater bias compared to larger ‘C’ values which produce more complex models with reduced bias but produced greater variance.

As shown below, initial tests with logistic regression produced very good results. The training accuracy was 0.661 and the validation accuracy was 0.665. As there is only a -0.004-accuracy difference, the model has not been overfitted to the training data. Although results varied depending on ‘C’ value, the results were better than all previously tested algorithms and was closer to our target accuracy of 0.70. Hence, it was decided that logistic regression would be the algorithm used to build the sentiment classifier.

A screenshot of a computer

Description automatically generated with low confidence

Text

Description automatically generated

Further exploration of the logistic regression model exposed that adjustment of the hyperparameters did not affect performance of the model significantly. Notice how in both sets of screenshots below, the best hyperparameters are very different but performance is similar. As a result of this observation, it was decided that the main model would be built using the default hyperparameters for simplicity.

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

To build the classifier that will make up most of our applications backend, a logistic regression model was trained using 500,000 randomly selected records. Due to time and hardware constraints, it was not feasible to use more than this; particularly as the ‘TruncatedSVD’ parameter ‘n\_components’ still needed adjustment. This parameter was given a value of 1110. Like the TFIDF vectoriser, this was done through a process of trial and improvement. This was important as the model would need sufficient feature space to develop an accurate algorithm without exceeding time constraints. When building the Logistic Regression model, the hyperparameter ‘max\_iter’ was set to 3000. This was because the default maximum number of iterations of the solver being used (‘lbfgs’) is 100. By increasing the max iterations to 3000, we allow the solver more opportunities to converge the algorithm.

To measure the accuracy of the final model the accuracy score and confusion matrix were both printed. As shown below, performance was very good and exceeded the requirement of 0.70 accuracy when testing against validation data achieving an accuracy of 0.751. The confusion matrix shows that the model accurately classified positive tweets with 37,008 (true positive) correctly classified. Similarly, 33,443 (true negative) tweets were accurately classified negative. There were more false positives classified using this model with 13,355 tweets compared to the 9,944 false negatives. These results suggest that the model is bias towards positive tweet classifications.

Text

Description automatically generated

**Chart, treemap chart

Description automatically generated**

**Development: APIs:**

The API was implemented using the Flask library in Python. The API is RESTful (Representational State Transfer). The benefits of using a REST API is that it is very lightweight, scalable and independent from other tiers of the application.

The UI and API communicate via POST requests. The API has two restful endpoints. They are ‘getSentenceSentiment’ and ‘getHashtagSentiment’.

getSentenceSentiment accepts JSON in the request body. This JSON contains the text to be analysed. It then returns the original text as well as the sentiment result in JSON.

Similarly, the second restful endpoint (getHashtagSentiment), accepts a hashtag to be analysed in the form of JSON. This returns a dictionary of original tweets relating to the hashtag as well as the corresponding sentiments for each tweet along with the tweet authors username.

The JSON body needs to be validated to ensure that we are receiving correctly formatted data from the client (UI). For example, ensuring that the endpoint for hashtags only receives a hashtag and not sentences with spaces in them.

When invalid requests are made, it is important that they are handled with a relevant response such as the correct HTTP code. Hence, successful requests with the correct JSON body return a HTTP 200 response. Conversely, requests with malformed or invalid JSON bodies return a HTTP 400 response. Errors regarding the classifier or API return a HTTP 500 response.

To ensure that the API and all the endpoints were functioning correctly, a variety of testing was performed against it. Manual testing was conducted using Google’s Postman tool.A variety of test requests were sent to the API and the responses were analysed to ensure the API returned the expected outcome. This included both positive and negative testing. Positive testing consisted of sending happy path responses and negative testing included sending invalid requests to the endpoints. Edge cases were also tested e.g. Sending one word to the sentence endpoint. Most the test cases were successful however one of the edge cases which was unsuccessful was the scenario where a stop word is solely sent to the ‘getSentenceSentiment’ endpoint. This causes an error due to the fact the stop word is filtered out in the pre-processing step and an empty array is inputted to the classifier. The API however gracefully handles this scenario by returning a HTTP 500 response. However, to make the application more robust, potential future workarounds include filtering the user input in the API by validating the body to ensure stop words are not sent by itself.

Manual end-to-end testing was also performed to test the API’s integration with the UI and the classifier. This will be further elaborated in the UI section.

**Talk about codes life**

**Mention twitter keys and how acquired and limitations**

**Development: UI**

The technology behind the User Interface (UI) is ReactJS. This is a JavaScript framework developed by Facebook and used by many others such as Netflix, Instagram and AirBnB. Alternative frameworks such as AngualarJS and VueJS were considered however ReactJS was used due to its UI focused designs, scalability, and usability.

React’s quick start guide [6] was used to create the skeleton for the UI. The Material UI library [7] was used as the component library for the UI, i.e. for buttons, textbox’s etc.

The root file for the UI is the index.js which renders the App.js file. The app.js uses the react BrowserRouter to route between the different pages.

The homepage will be <http://localhost:3000/>, which allows the user to navigate to the text page (<http://localhost:3000/text>) or the twitter page (<http://localhost:3000/twitter>). Both these pages consist of a textbox allowing the user to input their query and a submit button to fire the HTTP request.

The UI src folder also consists of a data directory and a pages directory. The pages directory consists of the components for targeting each API endpoint. The data directory consists of a service.js file which is a common functionality used between both page components. The service.js file consists of a function (getData) which performs a HTTP POST request to a given endpoint with a given body. The reason for abstracting the common functionality into a separate data directory was to adhere to object orientated programming principles. Abstraction is used to avoid code duplication and allow better reusability amongst components.

HTTP requests in JavaScript are asynchronous. This means the requests return a promise immediately and not the actual expected data response. However, when the server has the response available this will eventually be sent to the client. This can cause a lot of problems in the UI due to data not being returned immediately and users not having a fluid experience. JavaScript’s asynchronous nature is handled by using the fetch method along with ‘.then’ to wait for the promise to be returned before calling the subsequent function.

A loading icon has been implemented for when the user is waiting for a HTTP request to be returned to the UI, allowing the user experience to be uninterrupted. Once the HTTP requests return a response, these responses are set in the components state. Ternary operators are used to query the state to ensure valid results are available to display to the user. If valid results are available, the text/tweet and sentiment are displayed to the user along with an author for tweets.

User experience testing was performed on the UI from a random sample. Most participants of the testing enjoyed the user journey and an improvement suggested was the loading icon which was thereafter implemented.

As mentioned previously, manual end-to-end testing was also performed to explore all possible routes for the application. A variety of inputs were tested on all the pages. This allowed the testing of the UI, API, and classifier as well as the integration between all these components.

To ensure best coding practices all code was formatted using the VS Code built-in formatter and appropriate linting tools were used for each technology stack i.e. ESLint for JavaScript files and PyLint for Python files.

**Analysis:**

The development of this tool was successful. Despite time and hardware constraints, a well performing algorithm was produced that accurately classifies sentiment of tweet text. However, some improvements could potentially be made by further exploration. For example, are more negative tweets posted than positive tweets? If so, is it still best practice to balance the dataset?

Another improvement would be to explore and identify other potential stop words. When setting the min and max hyperparameters of the TFIDF vectoriser, it was found that most of the initially included terms did not improve the accuracy of the algorithm. This suggested that many stop words still remained within the corpus. To identify these, all the tweet text could be flattened and then printed. Words that do not hold any sentiment could then be removed from the dataset.

Rather than stemming, the use of lemmatisation (or another process that actually utilises a lexicon) could be explored however, due to the greater complexity of lemmatisation, hardware improvements would need to be made to make this feasible.

Another potential improvement to the exploratory phase would be to develop an iterative approach to identify the best combination of punctuation and capitalisation when pre-processing the dataset. This contrasts with the trial and improvement method used and would hopefully identify which punctuation, if any, should be included in the tweet text.

Regarding the exploratory model building, investigation of the logistic regression algorithm is needed to identify why changes to the hyperparameters did not affect the model performance. Likewise, as accuracy was good, further exploration of the random forest classifier should be conducted to find the best hyperparameters. To allow for this, more extensive pre-processing should be conducted to reduce the number of records and features included. This would be facilitated using more powerful computers with greater processing power.

The APIs function correctly, linking the classifier, UI and retrieving tweet text. However, some improvements could still be made. Once such improvement would be testing the API. This could be done using testing libraries such as Pytest. Thorough unit testing would ensure that the API functionality is working as expected and that no future changes break it. Integration testing between the API and UI, and the API and classifier would ensure that future changes do not affect functionality. Some examples of future performance tests would be sending multiple and concurrent requests to see how the API handles these. Once the limit of the application is known, horizontally scaling the application can be considered using technologies such as using Kubernetes and Docker with a load balancer to run the API on multiple nodes [8]. A benefit of this would be that if a node were to go down, traffic would be redirected to another node, making the application highly available. This would also make the application easier to deploy. Using a load balancer to distribute traffic to the API would allow more nodes to be used and improve the performance of the API by allowing for concurrency and more requests being allowed. Regression testing of the endpoints could be conducted using the postman collection runner, automating the running of a collection of predefined requests. Authentication would ensure the security of the API. This could be done using SSH certs or user credential verification ensuring that only authenticated users will be able to access the API. HTTPS encryption will allow all traffic between the client and server to be encrypted, ensuring all data is protected from interception/man in the middle attacks. It would also be important to implement a deterrent to DDoS attacks. Furthermore, a personal twitter API key is currently in use. If users were to request inappropriate requests this will cause personal harm – so a business API key should be used.

The UI In hindsight, there are several aspects of the tool that require further exploration and development.

Talk about future domain hosting etc. along with Kubernetes docker highly available application for user.

Jest unit testing

Cypress automated UI testing

UI Validation

**Code clean up with linting, sonarqube for static analysis, ide tool for error checking**

**Conclusion:**

The project development process allowed for the development of a sentiment classifier with a good accuracy of 0.75 however, the pre-processing stage was minimalistic and could be improved to produce a more accurate model. Other algorithms such as random forest could be further explored to produce a more accurate model, improving the overall performance of the application. The APIs produced operate correctly; connecting the desired endpoints however, as mentioned, further testing could expose some weaknesses, particularly in security. The results are presented using a website as the user interface that adheres to good usability practices however there are several improvements that could be made to make it more accessible. To conclude, the project goals and requirements were aspirational. This being considered, the outcome of the application was good, and development overall was successful.

**References**

[1] - Mcleod, S. (2023) *Attitudes and behavior in psychology*, *Simply Psychology*. Available at: https://www.simplypsychology.org/attitudes.html (Accessed: April 1, 2023).

[2] - Chauhan , S. (2018) *Mail2sawan/emotions-tagging-using-advance-NLP*, *GitHub- mail2sawan/Emotions-Tagging-using-Advance-NLP*. Available at: https://github.com/mail2sawan/Emotions-Tagging-using-Advance-NLP (Accessed: January 1, 2023).

[3] - Vidiyala, R. (2020) *Twitter tweets data for sentiment analysis*, *Kaggle*. Available at: https://www.kaggle.com/datasets/ramyavidiyala/twitter-tweets-data-for-sentiment-analysis (Accessed: December 12, 2022).

[4] - KAZANOVA, Μ.Α.Ρ.Ι.Ο.Σ.Μ.Ι.Χ.Α.Η.Λ.Ι.Δ.Η.Σ. (2017) *Sentiment140 dataset with 1.6 million tweets*, *Kaggle*. Available at: https://www.kaggle.com/datasets/kazanova/sentiment140 (Accessed: February 1, 2023).

[5] - Bland, G. (2020) *Train/test split and cross validation - A python tutorial - ALGOTRADING101 blog*, *Quantitative Trading Ideas and Guides - AlgoTrading101 Blog*. Available at: https://algotrading101.com/learn/train-test-split/ (Accessed: January 1, 2023).

[6] - *Getting started* (2021) *Create React App*. Available at: https://create-react-app.dev/docs/getting-started (Accessed: April 18, 2023).

[7] - *Material-ui: A popular react UI framework* (no date) *UI*. Available at: https://v4.mui.com/ (Accessed: April 19, 2023).

[8] - Banerjee, S. (2023) *Kubernetes vs Docker: Understanding the differences and advantages*, *Civo.com*. Available at: https://www.civo.com/blog/kubernetes-vs-docker-a-comprehensive-comparison (Accessed: April 1, 2023).

**Bibliography**

Caputo, R. (2019) *Double Trouble: How Children and Adults Decide When to Use Double Consonants*, *Washington University in St.Louis: Open Scholarship Institutional Repository*. Available at: https://openscholarship.wustl.edu/art\_sci\_etds/1979/ (Accessed: January 1, 2023).

Burkov, A. (2019) *The hundred-page machine learning book*. Quebec City: Andriy Burkov.

Goyal, G. (2023) *Twitter sentiment analysis using Python: Introduction & techniques*, *Analytics Vidhya*. Available at: https://www.analyticsvidhya.com/blog/2021/06/twitter-sentiment-analysis-a-nlp-use-case-for-beginners/ (Accessed: March 15, 2023).

Ahmed, M.S. *et al.* (2017) *Robustification of naïve Bayes classifier and its application for microarray gene expression data analysis*, *BioMed research international*. U.S. National Library of Medicine. Available at: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5564130/#:~:text=The%20na%C3%AFve%20Bayes%20classifier%20(NBC,the%20location%20and%20scale%20parameters (Accessed: February 1, 2023).