Abusive Language Detection Against Immigrants and Women

1. Introduction

1.1 Project Aims

The aim of this project is to design an effective system for detection of abusive language towards immigrants and women. The project is based on a similar project run online,[1], and as such I will be comparing my work and results to the results of other participants of the competition. As a result of there being others working with the same data set that I have used for this project, a suitable goal of the project was to achieve the highest accuracy possible and to compare this accuracy to other participants of the competition. This would be done through a thorough understanding of natural language processing and experimentation with different forms of classifiers.

1.2 Intended Audience

The intended audience of this project is anyone interested in natural language processing, but also anyone interested in the practical application of offensive speech detection online. This is especially beneficial in the current world, given the rise of hate speech online on popular platforms such as Twitter.[2] While this project was developed and tested on a specific set of tweets, with some modification it could be used to classify tweets in real time. I would argue, therefore, that the techniques and models explored in this project would be beneficial to anyone intending to implement an automated hate speech detector into their website. Equally, this project would be beneficial to law enforcement agencies and governments looking to curb the rise in online hate speech.

1.3 Project Scope

The scope of this project adapted over time, due to the fact that at the start of the project the libraries and concepts I was going to be working with were largely unknown to me. As a result of this, it was difficult to estimate the amount of time it would take to complete pieces of work.

1.4 Approach

The way that I approached carrying out the project was through regular meetings with my supervisor, coupled with a Kanban approach to working. This meant that I was focused on small sections of deliverable work that I could then discuss with my supervisor and plan out what to work on next. This also meant that the scope was able to remain flexible throughout the project, and I was able to pivot quickly in the event that circumstances changed, which they did at some parts of the project.

The way in which I ran the experiments that lead to my outcomes changed throughout my project as the data available to me changed, and as such can be split into two methods. Initially, I did not have access to the test set that others who attempted this problem had, and as such I combined the two data sets I did have (train set and dev set),

[1] https://competitions.codalab.org/competitions/20011

[2] Banks, J., 2010. Regulating hate speech online. *International Review of Law, Computers & Technology*, *24*(3), pp.233-239.

and then split this one set into multiple splits using a method known as *K-Fold Cross Validation*. Once I was able to get access to the third data set during my project, I approached the project in the following way, which is considered best practice:

- Implement a method of converting the corpus into features.
- Select a classifier model.
- Provide the model with the train set and features.
- Predict whether the tweets in the dev set were offensive or not.
- Compare these predictions to actual results.
- Record results.
- Experiment with the classifier parameters and features.
- Repeat

Once I had done this to a point where I was happy with each of the different methods I had implemented, I then generated predictions for the test set and compared these predictions to the actual results. These scores were then used for my final results.

1.5 Ethical Considerations

It is important to mention the ethical considerations that come with a project like this. Although this project makes use of large amounts of user generated content in the form of data sets used to train classifier models and generate features, all of this data was published alongside peer reviewed publications, and thus any ethical issues when compiling and distributing this data is left to the authors of the original publications.

There are also a number of expletives used throughout this paper which the reader will encounter, which are absolutely necessary due to the strict academic need to show the kind of data that has been worked with during this project.

1.6 Summary of Outcomes

There were a number of key outcomes that came out of this project, and they are as follows:

- Contrary to initial predictions, selection of feature extraction method plays a far larger role in the accuracy of results returned than the selection of classifier model.
- The combination of Term Frequency–Inverse Document Frequency (TF/IDF) as the method of feature extraction and a Support Vector Classification (SVC) classifier model provided the most accurate predictions, as measured by specific metrics that are discussed later in this report.
- Although the combination of TF/IDF feature extraction and SVC as the classifier model yielded the most accurate results, this combination also had the longest program run time by far.
- The combination of Word2Vec word embeddings as features and SVC as the classifier model was the best balance between shorter program run time and accuracy of results.
- Although a combination of feature extraction and classifier model was implemented that was highly accurate in its predictions, when compared to others working with the same data sets the most successful combination was not as accurate as teams using Neural Networks in their systems.

2. Background

2.1 Project Context

The wider context of the problem of detection of abusive language towards women and immigrants is clear, as the rise of social media platforms and the anonymity that they provide has precipitated an increasing amount of online abuse, with one survey suggesting 18% of teenagers experienced abusive language while communicating online.[3] Given the frequency of abusive language use online, I would argue that there is a clear need for accurate automated detection methods of said language.

The main aim of this project is focused around research and analysis into how best to approach the design and implementation of a program for detection of abusive language towards women and immigrants. I have chosen these two groups specifically due to the large amount of abuse they receive online.[4] In particular, immigrants have faced a rise in hate speech in part due to the refugee crisis,[5] and women have historically always suffered from hate speech.[6] The main project focus of detection of abusive language splits into a multitude of sub problems, but the main three are feature extraction, model selection, and experiment setup. It is through working through these three problems that I was able to produce a varying different sets of results with varying accuracy.

Another aim is to produce a piece of software that, when given a tweet via the command line, will be able to accurately predict if the tweet is offensive or not towards the target groups. This was successfully developed using the research outcomes I found from the main aim of the project.

2.2 Identified Problem

As identified in my initial plan, the pervasive nature of abuse towards women and immigrants online shows that this is a problem that needs solving. Although the focus of this project was experimentation and analysis, the outcomes and findings are useful in determining the best approach in developing an accurate automated method for detection of abuse towards women and immigrants. Using the findings of the project, I was able to implement a script that takes a single tweet as an input and returns a JSON blob containing

[3] Kawate, S. and Patil, K., 2017. Analysis of foul language usage in social media text conversation. *International Journal of Social Media and Interactive Learning Environments*, *5*(3), pp.227-251.

[4] Basile, V., Bosco, C., Fersini, E., Nozza, D., Patti, V., Pardo, F.M.R., Rosso, P. and Sanguinetti, M., 2019, June. Semeval-2019 task 5: Multilingual detection of hate speech against immigrants and women in twitter. In *Proceedings of the 13th International Workshop on Semantic Evaluation* (pp. 54-63).

[5] Bosco, C., Patti, V., Bogetti, M., Conoscenti, M., Ruffo, G.F., Schifanella, R. and Stranisci, M., 2017. Tools and resources for detecting hate and prejudice against immigrants in social media. In *SYMPOSIUM III. SOCIAL INTERACTIONS IN COMPLEX INTELLIGENT SYSTEMS (SICIS) at AISB 2017* (pp. 79-84). AISB.

[6] Cresti, M., Martino, V. and Rosola, M., Kate Manne. Down Girl. The Logic of Misogyny, Oxford University Press, 2017, pp. XXIV, 338. *APhEx*.

an offensive/not offensive prediction for the tweet. This could be quickly added to an online platform to aid in hate speech detection.

2.3 Likely Stakeholders

In terms of likely stakeholders in the problem area, those that would benefit from developing and implementing an automated system for the detection of abuse language would be social media sites and law enforcement agencies. There is a vested interest among social media site owners to stop and remove abusive language on their platforms in order to make their platform more welcoming and more pleasant to use, as end users will likely stop using the site if they are receiving abuse. Equally, some of the abusive language that could be posted online could constitute as hate speech, which is a crime in the United Kingdom[7] as well as many other nations, and therefore it would be beneficial to law enforcement agencies if they could automatically detect this abusive language.

This project would improve the first two steps in the hate crime prosecution process, namely reporting and investigation of said crime. By automatically detecting hate speech, platforms would be able to report the comments, police would be able to investigate said comments and report them to the Crown Prosecution Service as necessary. Prosecution could then take place as normal but the previous two steps would be enacted faster and with less manual intervention, for example in the form of human moderators.

2.4 Theory Associated with Problem Area

The focus of this project is around machine learning, of which the three main areas are models, features, and experiment setup. The majority of the work on this project involved experimenting with these three areas in different formats and combinations to try and achieve the best accuracy results possible.

Model selection involves making use of different classifier models for more accurate hate speech detection. There are two main differences in the models that I used, prediction accuracy and program run time. I made use of four different classifiers that I chose for varying reasons that I will elaborate on below.

- The first model I used was a Decision Tree.[8] This model works by looking for a common word in the corpus that can split the data into offensive/not offensive tweets, and then repeating this step again with a new word. It is possible to set a maximum number of leaves for the tree which will improve run time but potentially lower the accuracy of the model. I chose this model to begin with as it was a simple model to understand and implement as it requires minimal data preparation to implement correctly.
- The second model that I implemented was the Random Forest model.[9] This model is "a meta estimator that fits a number of decision tree classifiers on various sub-

[7] https://www.cps.gov.uk/hate-crime

[8] https://scikit-learn.org/stable/modules/tree.html

[9] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting". This was a logical follow on from using the Decision Tree classifier due to the similarity in the way that they work.

- The third model I implemented was Logistic Regression.[10] This model involves using a Logistic Function[11] in order to model probability and generate a prediction. I chose this model due to the belief that it would be a good balance between slightly longer run time and improved prediction accuracy.
- The fourth and final model that I implemented was Support Vector Classification.[12] This model works by plotting a line through the corpus matrix in order to make its classifications. This model was chosen in the belief that it would provide the best accuracy, albeit in exchange for the longest run time.

Feature extraction encompasses the tokenisation of the corpus and the subsequent extraction of features that will be used to train the selected model. I ended up using three forms of feature extraction, a Count Vectorizer[13], TF/IDF extraction[14], and a Word2Vec[15] model.

- The Count Vectorizer was by far the least effective form of feature extraction as it went through the corpus and selected the 50 most common words excluding English "stop words", such as "and, if, it, to" for example. Although this was not very effective in terms of accuracy, it was a quick and simple form of feature extraction to implement and was a good starting point for the project.
- The Word2Vec model is a more effective method of feature extraction that involves grouping similar words together to produce word embeddings, that can then be tokenized and used as features to train a classifier model. I was given access to a Word2Vec model that was trained on a large corpus that produced a very effective set of word embeddings that I then tokenized and used to produce highly accurate results.
- Term frequency–inverse document frequency or TF/IDF is the final form of feature extraction I implemented. This vectorizer works by putting weightings on features to determine how important they are to documents in the corpus. I made use of parameters such as "n grams" to specifically point the vectorizer to look for common phrases as well as words based on characters.

[10] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[11] https://en.wikipedia.org/wiki/Logistic_function

[12] https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

[13] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

[14] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

[15] https://radimrehurek.com/gensim/models/word2vec.html

In terms of experiment setup, I went through varying setups involving differing forms of feature extraction and models together, with varying parameters. I made sure to record and export all of these results during the course of my work in order to catalogue the variation in results.

- One of the major changes in experimental setup throughout my work on this project was the change from using k-fold cross validation[16], to the best practice train/dev/test split. The best practice for work on machine learning algorithms involves training your model on the train data set and then testing it on the dev data set, before implementing changes to the model or feature extraction to improve results. Only once you are satisfied with your model and feature extraction should you perform the final test on the test data set to produce your final results. Unfortunately, I did not have access to the test data set until halfway through the project, so I combined the train and dev data sets and implemented k-fold cross validation. This performed the train/dev/test splits and I then combined the results to create an average.

2.5 Constraints on Approach

As stated in my introduction, I was originally constrained from approaching the project in, what is considered as, the best practice method for developing a model to classify offensive tweets due to not having access to the test data set that is used to generate final results, and ultimately compare these results to other teams who have worked on the same problem. This constraint was solved halfway through the project and I was able to work in the best practice approach for the remainder of the project.

Due to this constraint, I had to adopt an alternative approach for the initial period of my work on this project, involving using k-fold cross validation splits to alleviate the issue created by not having a test set. This proved to be a viable approach to the project as the results and outcomes generated during this period of the project were essentially similar to when I approached the project with best practice in mind. The trends methods I found to get the best results were the same and could be adopted when I was given access to the test set.

2.6 Existing Solutions

There were a large number of varying methods and research presented by others participating in the competition this project is based on. They range from machine learning methods, similar to the work that I have to done, to deep learning. For example, there was use of Support Vector Machines similar to what I have used. Although there was use of both machine learning and deep learning techniques, deep learning was by far the most popular method of approaching the problem with 70% of teams making use of deep learning.[17]

[16] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html

[17] Zampieri, M., Malmasi, S., Nakov, P., Rosenthal, S., Farra, N. and Kumar, R., 2019. Semeval-2019 task 6: Identifying and categorizing offensive language in social media (offenseval). *arXiv preprint arXiv:1903.08983*.

Further to this, the most successful teams to participate made use of deep learning, with the highest ranked team to not use deep learning placing 6th.

As well as use of varying models and feature extraction, there was also a wide range of different methods of data pre-processing used. These methods ranged from making use of additional training material such as Word2Vec models and/or external data sets, to using sentiment analysis models for prediction, or even using offensive word lists.[18] Some teams used twitter specific tokenizers, for example NTLK TweetTokenizer. Teams also made efforts to normalise URL's, hashtags, and elongated words to improve results. There were also teams that converted emojis to text to be able to analyse them effectively.

2.7 Methods and Tools

My project was developed using Python 3.7.6,[19] and in particular I made use of 3 specific libraries to complete the project: Sci-Kit Learn, Gensim, and Numpy. In terms of the structure of my program, each of my models is made up of a form of classifier (For example, a decision tree of Logistic Regression), and a form of feature extraction that helps to train the model with features that are deemed as important markers for if a tweet is offensive or not.

Sci-Kit Learn[20] is a library that I used to give me access to a wide number of different classifier models that I could train my corpus on and use to generate predictions for if a tweet is offensive or not. As well as this, at one stage I also used this library to extract features from the corpus as a further method of training the classifier model.

Gensim[21] is a library that allowed me to implement a form of neural networks into the project, using a model of feature extraction called Word2Vec. Word2Vec works by creating word embeddings for each word in the corpus and then looking at what words are "near" that word in order to determine links. I implemented two forms of feature extraction in my project, with one Word2Vec model that was trained on my data set, and another Word2Vec model that was trained on a much larger data set of tweets. Both of these models were used in my work and both produced varying results. I also used Word2Vec to analyse what words were "nearest" to certain words to enable me to see what words are commonly used alongside what could typically be considered as abusive language.

NumPy[22] is a library that extends the operations that Python can perform on data structures, particularly arrays and matrices. This is useful for my project given that Sci-Kit Learn makes extensive use of sparse matrices to represent the predictions output by the classifier models, and by using NumPy it is much easier to record these results.

[18] Zampieri, M., Malmasi, S., Nakov, P., Rosenthal, S., Farra, N. and Kumar, R., 2019. Semeval-2019 task 6: Identifying and categorizing offensive language in social media (offenseval). *arXiv preprint arXiv:1903.08983*.

[19] https://www.python.org/

[20] https://scikit-learn.org/

[21] https://radimrehurek.com/gensim/

[22] https://numpy.org/

Joblib[23] is a library that allows you save models that have been trained by Sci-kit learn as Joblib files for later use. This is extremely useful as often the bulk of the time spent by a function generating predictions is spent training the model. By being able to save and load classifier models that are pre-trained, you only have to train the model once and can simply load the pre-trained model each time you want to use it. This is significantly faster and a useful method of cutting down on runtime.

[23] https://joblib.readthedocs.io/en/latest/

3. Specification and Design

3.1 Approach to Solving the Problem

The exploratory nature of this project meant that my approach to the project changed over the duration of the work. As mentioned previously, there were changes in the experiment setup, methods of feature extraction, and classifier models. I have illustrated the way in which the program works in a flow chart that can be found below.

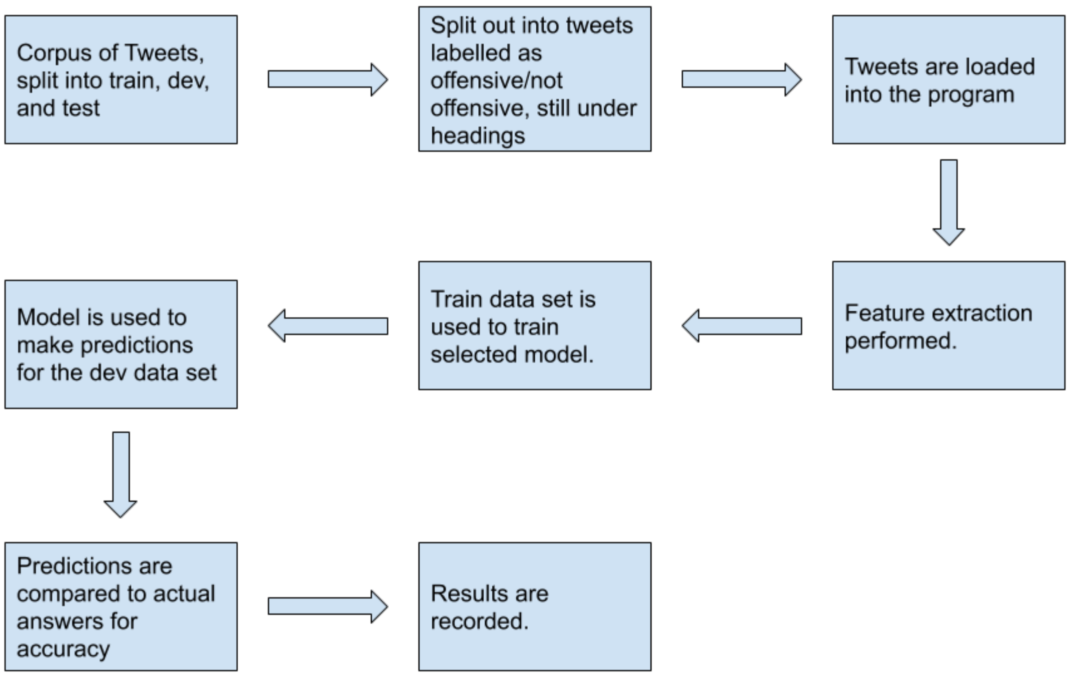

Corpus of Tweets, split into train, dev, and test
Split out into tweets labelled as offensive/not offensive, still under headings
Tweets are loaded into the program
Feature extraction performed.
Train data set is used to train selected model.
Model is used to make predictions for the dev data set
Predictions are compared to actual answers for accuracy
Results are recorded.

Figure 1

The reasoning behind why there was so much of an evolution in all areas of the project over the duration is due to my limited experience with the tooling and concepts used in the project prior to beginning work. As a result of this, many of the concepts and tooling I initially used were not best practice, specifically the use of k-fold cross validation to substitute a train/dev/test split in data sets, or they did not provide very accurate predictions. The final iteration of the project, as shown in the diagram above, implements all of the concepts and tooling that I found to produce the most accurate predictions, as measured by precision, recall, and f-1 score.

As I have shown in the diagram above, the data flow of the program is relatively simple and remains the same despite variations in experimental setup, feature extraction, and model selection. The diagram does, however, abstract a lot of the workings going on underneath the simple step descriptions I have included. For example, the step "Tweets are loaded into the program" abstracts the need to load the tweets and their corresponding labels into feature vectors in such a way that Sci-Kit Learn can make use of the corpus.

Another flaw in the diagram is that it does not show the evolution that this project went through over the course of my work on it. For example, although I was always comparing the predictions to the actual labels to determine the accuracy of the system, the way in which I calculated this accuracy did vary. In the final iteration of the program I had one set of predictions to compare to the actual tweet labels, but in earlier iterations I had 5 different sets of predictions. This was due to my use of k-fold cross validation - the use of which and its use case I have previously explained. As a result of this, I had to average these 5 results together to get an overall result thus complicating the step of result generation.

The way in which I recorded my results was to include a function that first noted down the parameters of the experiment to the results file and then copied down the classification report containing metrics, such as precision, recall, and f-1 score of the experiment. All of this information was recorded in a CSV for two main reasons. First, because Python has a built in module for handling CSV files, and secondly because CSV files can store large amounts of data in a tabulated format that can be printed to the command line or displayed in a spreadsheet program such as Excel, making them a versatile method of displaying the results collected. As a result of recording my results automatically, every time an experiment was run, I had an extensive catalogue of results that I have been able to examine and analyse, allowing me to accurately present a summary of outcomes as to what the most effective methods for detection of abusive language are.

Another key step that is abstracted in program flow chart under the "tweets are loaded into the program" step is data pre-processing. This step involves normalising the tweets so to improve the overall accuracy of the results produced by the experiment. I performed varying levels of data pre-processing throughout the project as I learned more about natural language processing. In the final state of the project, the most effective form of pre-processing I used was to allow for the use of "n grams" to specifically look for common phrases instead of words. This vastly improved the accuracy of my results.

I have previously touched on the variations in feature extraction and model selection, and I will go into depth about how exactly I implemented these varying methods that make up the core of the experimental setup.

3.2 Metrics Used

Precision, Recall, and F-1 score are the key metrics by which the accuracy of predictions made by the system are made - an explanation of each metric and the formula to calculate it will be provided now.

Precision is defined as the number of True Positives divided by the sum of True Positives and False Positives.[24] What this metric measures is the number of predicted positive cases (in this case, offensive tweets), compared to the number of those predictions that were actually offensive.

$$\text{Precision} = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

Figure 2

[24] https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9

Recall is defined as the number of True Positives divided by the sum of True Positives and False Negatives.[25] What this metric measures is the number of offensive tweets the model is labelling correctly as a True Positive by including the cases that the model is incorrectly predicting as offensive in the formula.

$$\text{Recall} = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

Figure 3

F1-Score is the final and perhaps most important metric that will be used in this paper to determine the accuracy of predictions made by the models used. F-1 Score is defined as Precision multiplied by Recall, divided by the sum of Precision and Recall, multiplied by 2[26]. The F-1 Score is necessary as it strikes a balance between Precision and Recall thus showing a good overall measure of the way a model is performing.

$$\text{F1} = 2 \text{ x } \frac{Precision * Recall}{Precision + Recall}$$

Figure 4

[25] https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9

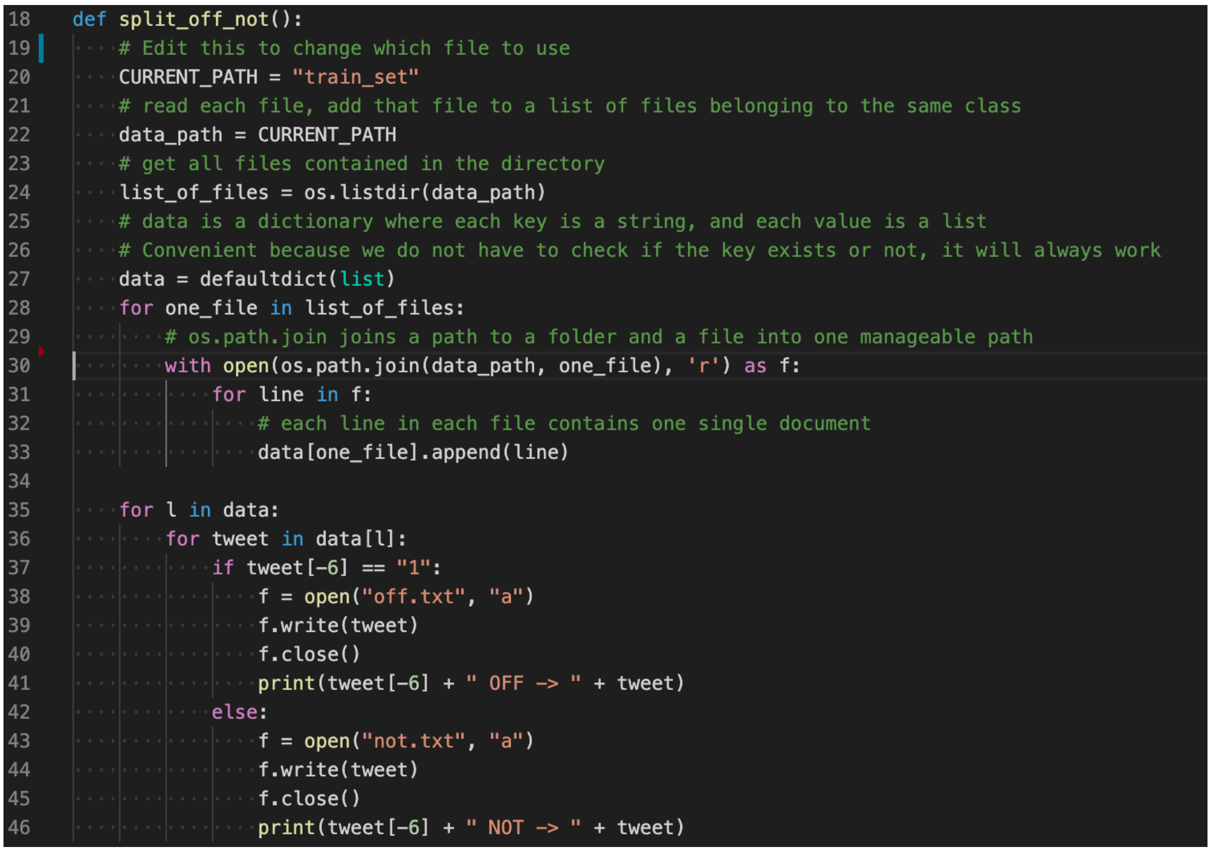
[26] https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9

4 Implementation

4.1 Data Handling

The first part of the project that I worked on was the handling of the data sets as the libraries that I utilised required data to be formatted in a specific way before you can make use of the corpus to train the classifier models or extract features from them. To do this in Python, I made use of the OS[27] library as my method of file handling to load the tweets from the text files I had them stored in. I used this library due to the fact that it is built into Python and so did not require another dependency to get the program running.

Once the text files have been loaded, the next problem I had to solve was the splitting of the training data (in this case, the train data set) into separate offensive and not offensive text files, based on their labels within the original train set text file. The reason this had to be done was to facilitate the creation of a Python list of tweets with corresponding labels that Sci-Kit Learn can use to train a classifier model, as well as compare predictions to actual tweet labels to determine the accuracy of said model. The screenshot below shows the function that I wrote to perform this task, along with comments explaining what the loading lines do. The step of splitting the tweets into their separate offensive/not offensive text file only needs to be done once, as the files are saved.



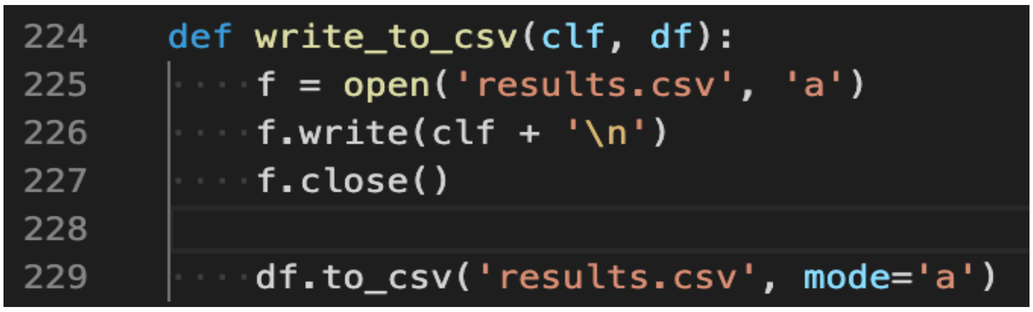
```python
def split_off_not():
    # Edit this to change which file to use
    CURRENT_PATH = "train_set"
    # read each file, add that file to a list of files belonging to the same class
    data_path = CURRENT_PATH
    # get all files contained in the directory
    list_of_files = os.listdir(data_path)
    # data is a dictionary where each key is a string, and each value is a list
    # Convenient because we do not have to check if the key exists or not, it will always work
    data = defaultdict(list)
    for one_file in list_of_files:
        # os.path.join joins a path to a folder and a file into one manageable path
        with open(os.path.join(data_path, one_file), 'r') as f:
            for line in f:
                # each line in each file contains one single document
                data[one_file].append(line)

    for l in data:
        for tweet in data[l]:
            if tweet[-6] == "1":
                f = open("off.txt", "a")
                f.write(tweet)
                f.close()
                print(tweet[-6] + " OFF -> " + tweet)
            else:
                f = open("not.txt", "a")
                f.write(tweet)
                f.close()
                print(tweet[-6] + " NOT -> " + tweet)
```

Figure 5

[27] https://docs.python.org/3/library/os.html

Following on from splitting the tweets out, I needed to load the separate offensive/not offensive files into the program. The way in which I did this essentially involved recycling the loading code shown in the “split_off_not” function shown above as they essentially need to perform the same role of loading the tweets into a list, and because of this I will not show the “load_file” function that performs this task.

The final form of data handling that needed to be implemented was ensuring that all results were automatically recorded for later analysis. It was important to record this information in some form of permanent storage otherwise results would be lost any time the terminal was closed. Another requirement was the need for this function to be modular and separate from the “main” function due to the amount of times that this function would need to be run.



```
def write_to_csv(clf, df):
    f = open('results.csv', 'a')
    f.write(clf + '\n')
    f.close()

    df.to_csv('results.csv', mode='a')
```

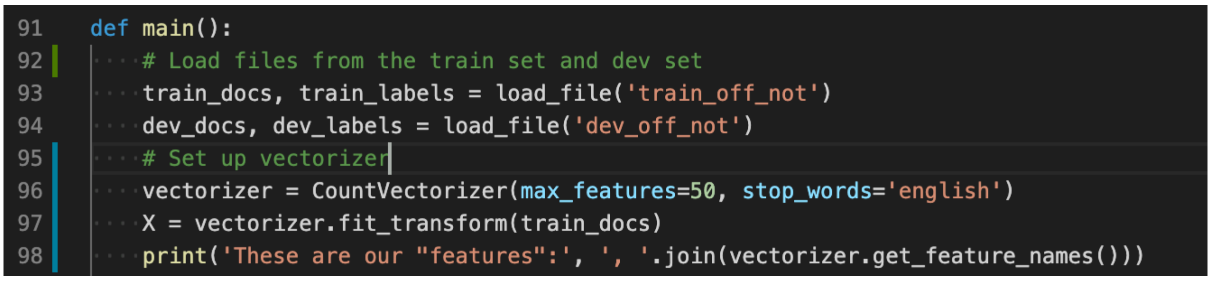
Figure 6

As can be seen in the code excerpt above, the “write_to_csv” function is a small and modular function that is parsed the classifier used, as well as a Pandas data frame containing the results from the classifier predictions. It then opens the “results.csv” file to amend it first with the experimental setup, namely the classifier used and its parameters, followed by a new line in order to make the structure of the results file clearer. Following on from this, the function makes use of the “to_csv” function from the NumPy library that writes a Pandas data frame directly to a CSV file. The result of running this function is the results file is updated with the experiment setup as well as the results of the experiment, and they are ready for further examination and analysis at a later date.

4.2 Feature Extraction

4.2.1 Count Vectorizer

The first form of feature extraction that I implemented was a Count Vectorizer. The way in which I implemented this was by using a method built into the Sci-kit Learn library. The way in which this method works is by collecting a list of the most common words in the corpus it is provided with and use those as the features with which train the model you have selected. In my case, I limited the vectorizer to 50 words but also made use of the “stop_words=English” parameter to skip over common linking words, for example “and, or, if”. I experimented with including more features but the trade off in run time did not yield any significant improvement in the accuracy of the predictions made when using a Count Vectorizer.



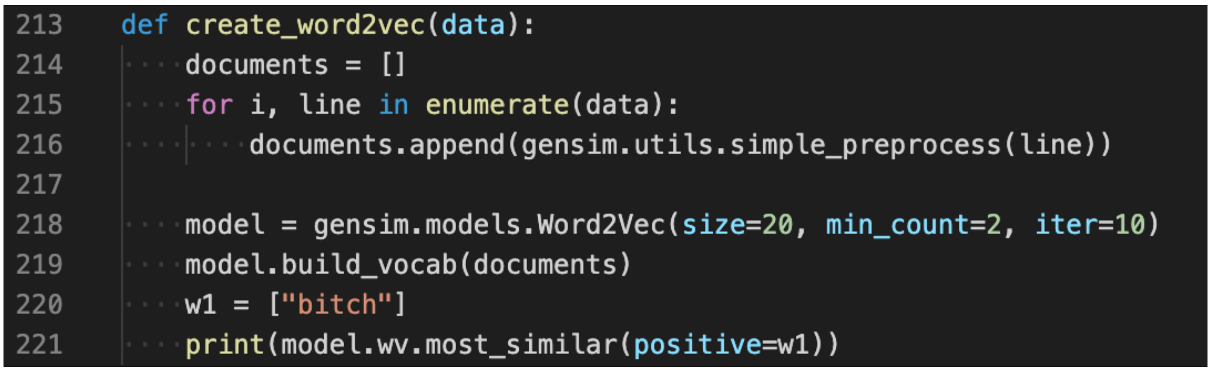
```
91  def main():
92      # Load files from the train set and dev set
93      train_docs, train_labels = load_file('train_off_not')
94      dev_docs, dev_labels = load_file('dev_off_not')
95      # Set up vectorizer
96      vectorizer = CountVectorizer(max_features=50, stop_words='english')
97      X = vectorizer.fit_transform(train_docs)
98      print('These are our "features":', ', '.join(vectorizer.get_feature_names()))
```

Figure 7

As can be seen in Figure 7, this code excerpt shows that the train data set and the dev data set are loaded into the program. Once this is done, the vectorizer is initialised with the parameters desired, and then the train set corpus is parsed to the vectorizer which will produce a list of features for use by the classifier models, shown on Line 97. For verbosity, the list of features that is generated by the vectorizer is printed to the console. Although the Count Vectorizer was the simplest form of feature extraction that was implemented, it was the worst performing in terms of accuracy of predictions.

4.2.2 Word2Vec Word Embeddings

Following on from the Count Vectorizer, the next step for feature extraction was exploring Word2Vec models as a way of getting a better set of features to train classifier models with. Word2Vec[28] works by creating word embeddings for each word in the corpus and then looking at what words are "near" that word in order to determine links between words. The use of Word2Vec in this project occurred in two stages. Initially the corpus that was provided as part of this project was used to train a Word2Vec model, and the subsequent word embeddings were used as features to train classifier models.



```
def create_word2vec(data):
    documents = []
    for i, line in enumerate(data):
        documents.append(gensim.utils.simple_preprocess(line))

    model = gensim.models.Word2Vec(size=20, min_count=2, iter=10)
    model.build_vocab(documents)
    w1 = ["bitch"]
    print(model.wv.most_similar(positive=w1))
```

Figure 8

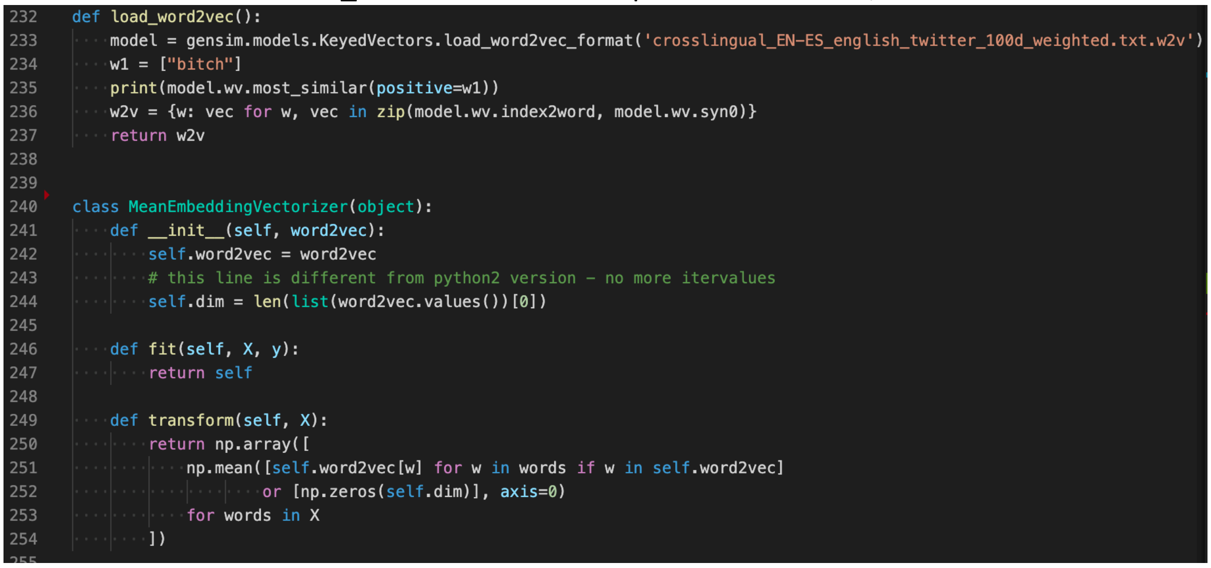
As shown in the code excerpt above, the "create_word2vec" function takes a corpus of tweets and formats it in such a way that Gensim can create the word embeddings. Once this is done, a Word2Vec model is initialised with parameters. In this instance, the parameters are a size of 20 embeddings, where a word has to be used twice to be considered, and the number of times to iterate through the corpus is set to 10. The corpus

[28] https://radimrehurek.com/gensim/models/word2vec.html

of tweets is then provided to the Word2Vec model, finishing the process of model creation. For verbosity and to ensure the model has been created correctly, the model is tested by printing out the most similar words to “bitch”, as well as the distance between our test word and the words most similar to it. Although the initial use of Word2Vec was an improvement on the previous use of a Count Vectorizer, it did not provide the improvement desired. Following research into how to improve the usefulness of Word2Vec models, the author determined that the small corpus size was impacting the usefulness of the word embeddings produced by this Word2Vec model.

The second iteration of the use of Word2Vec in this project was an attempt to try and improve the quality of the word embeddings produced by Word2Vec. As the main issue was the size of the corpus being used to train the model was too small, the solution that the author implemented was to use a pre-trained Word2Vec model.[29] This Word2Vec model was trained on a much larger corpus and, as such, its word embeddings were more accurate than the embeddings produced by the model trained on the smaller corpus. The use of this second model did require an adjustment in the codebase to make use of the model.

The function “load_word2vec” loads the pre-trained model, which is stored as a



```
232  def load_word2vec():
233      model = gensim.models.KeyedVectors.load_word2vec_format('crosslingual_EN-ES_english_twitter_100d_weighted.txt.w2v')
234      w1 = ["bitch"]
235      print(model.wv.most_similar(positive=w1))
236      w2v = {w: vec for w, vec in zip(model.wv.index2word, model.wv.syn0)}
237      return w2v
238
239
240  class MeanEmbeddingVectorizer(object):
241      def __init__(self, word2vec):
242          self.word2vec = word2vec
243          # this line is different from python2 version - no more itervalues
244          self.dim = len(list(word2vec.values())[0])
245
246      def fit(self, X, y):
247          return self
248
249      def transform(self, X):
250          return np.array([
251              np.mean([self.word2vec[w] for w in words if w in self.word2vec]
252                      or [np.zeros(self.dim)], axis=0)
253              for words in X
254          ])
```

Figure 9

“w2v” file in the root of the repository. Once again for verbosity, the model is tested and the most similar words to “bitch” are printed out to test that the model has been loaded correctly. In order to use the word embeddings as features for classifier models, they need to be formatted into NumPy in a sparse matrix, otherwise Sci-kit Learn will not be able to read them. This process is handled using a class found online,[30] although I did have to make a number of changes to update the code from Python2 to Python3. Line 236 collects the index value of the word embeddings and returns them formatted so that when “MeanEmbeddingVectorizer” is called as part of the classifier functions, it can take all of the

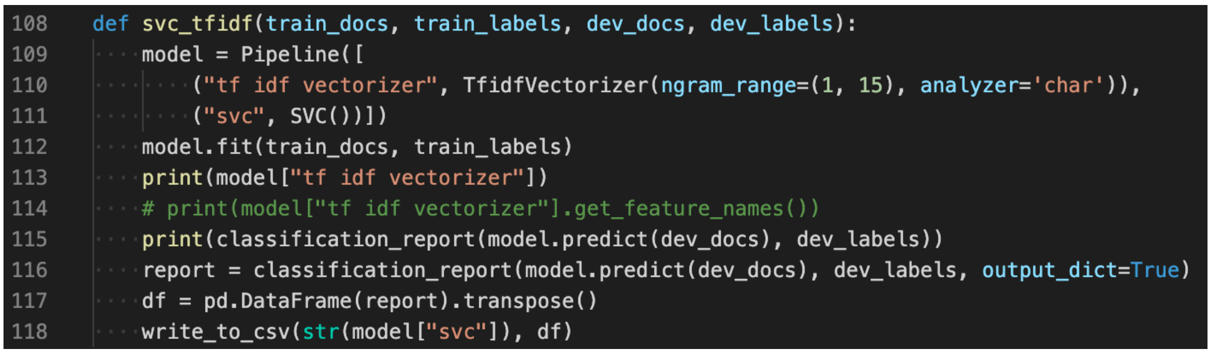
[29] Camacho Collados, J., Doval, Y., Martínez-Cámara, E., Espinosa-Anke, L., Barbieri, F. and Schockaert, S., 2020. Learning cross-lingual word embeddings from Twitter via distant supervision. *ICWSM*.

[30] http://nadbordrozd.github.io/blog/2016/05/20/text-classification-with-word2vec/

embedding index values and format them into a NumPy matrix, which is returned and used as features to train the classifier model. The use of this form of Word2Vec model did produce the results that were originally hoped for with the use of Word2Vec and did vastly improve the accuracy of the predictions produced by the classifier models.

4.2.3 TF/IDF

The final form of feature extraction that was implemented in this project was the use of "Term frequency–inverse document frequency" or TF/IDF. This form of feature extraction was implemented as a result of continued research into alternative means of feature extraction, and the intended effect when combined with any classifier was to be a significant improvement in results. This form of feature extraction works by putting weightings on words depending on how frequent and important they are to the corpus and determining its list of features from these weightings.



```python
def svc_tfidf(train_docs, train_labels, dev_docs, dev_labels):
    model = Pipeline([
        ("tf idf vectorizer", TfidfVectorizer(ngram_range=(1, 15), analyzer='char')),
        ("svc", SVC())])
    model.fit(train_docs, train_labels)
    print(model["tf idf vectorizer"])
    # print(model["tf idf vectorizer"].get_feature_names())
    print(classification_report(model.predict(dev_docs), dev_labels))
    report = classification_report(model.predict(dev_docs), dev_labels, output_dict=True)
    df = pd.DataFrame(report).transpose()
    write_to_csv(str(model["svc"]), df)
```

Figure 10

Above is an example of a function making use of an SVC classifier with a TF/IDF vectorizer. In this function, the feature extraction is built into a pipeline block of code along with the classifier model, in this case SVC. The pipeline is a feature of Sci-kit Learn which simplifies and shortens the code required. This can be compared to the code required to get the Count Vectorizer working. As such, all of the work done in the latter stages of the project made extensive use of the pipeline as a method of saving time and space.

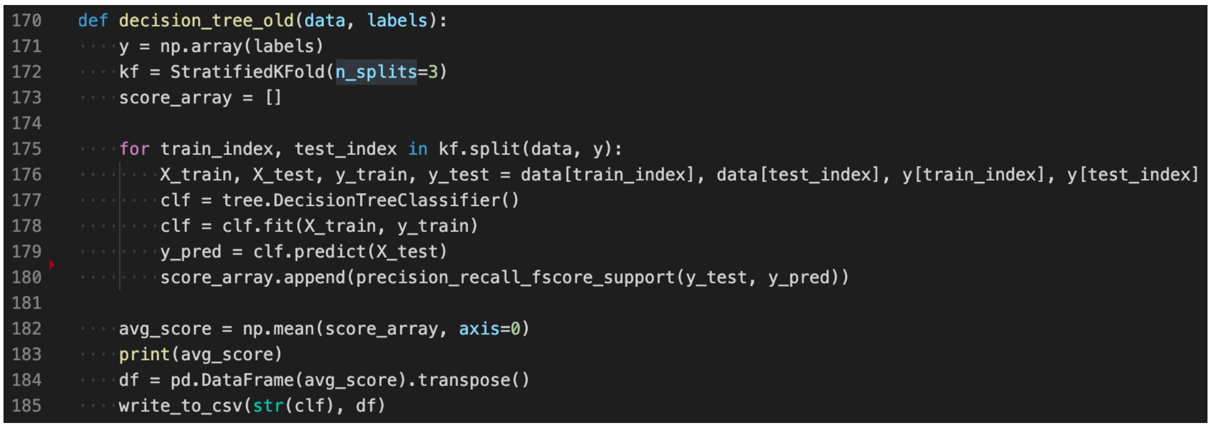
On the first use of the TF/IDF vectorizer, all of the parameters were set to default to see what impact this would have on results. Initially this did not improve results, which came as a surprise to the author as the expectation was an improvement. Upon further research into how this vectorizer worked, it was determined that the vectorizer was attempting to only tokenize words which was stopping the vectorizer from picking up on key phrases in the corpus. This problem was solved by first setting the analyser parameter to "char", and second the parameter "n_gram_range" was used to look for common groupings of characters between 1 to 15 characters long. The combination of these two parameters meant that the vectorizer could now pick up on key phrases in the corpus instead of just words as tokens. Due to these changes, there was a significant improvement in the accuracy of predictions made.

Despite this improvement in results, it is important to mention now that the combination of the TF/IDF vectorizer with the SVC classifier resulted in a huge increase in the run time of the program. The average run time of all other vectorizers and models was around 1 minute, but the run time with these particular parameters increased to over 20 minutes.

4.3 Classifier Model Selection

4.3.1 Decision Tree

Selecting classifier models to implement is the next key section of the project implementation to discuss. As mentioned previously in this report, 4 forms of classifier model were selected and implemented throughout the duration of this project. First, I will discuss the first model that was implemented during the project, the Decision Tree. It must be now said that due to the changes in experimental setup that was undergone throughout the project, the way that this classifier was implemented did undergo significant changes which I will now highlight.

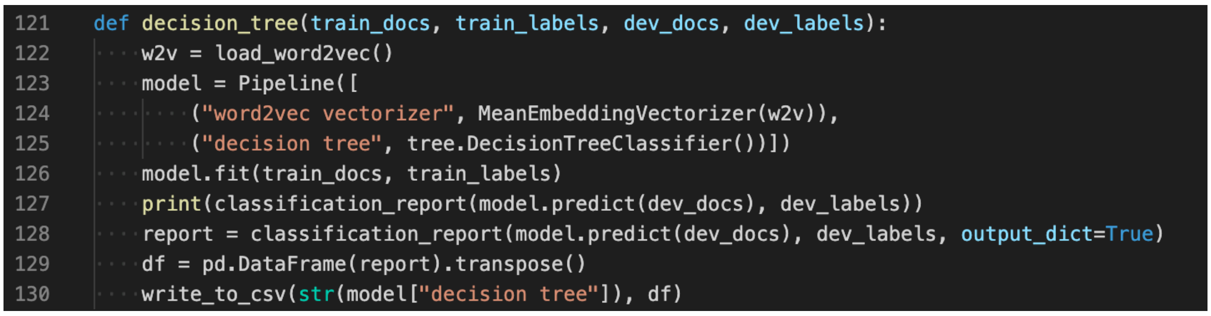


```python
def decision_tree_old(data, labels):
    y = np.array(labels)
    kf = StratifiedKFold(n_splits=3)
    score_array = []

    for train_index, test_index in kf.split(data, y):
        X_train, X_test, y_train, y_test = data[train_index], data[test_index], y[train_index], y[test_index]
        clf = tree.DecisionTreeClassifier()
        clf = clf.fit(X_train, y_train)
        y_pred = clf.predict(X_test)
        score_array.append(precision_recall_fscore_support(y_test, y_pred))

    avg_score = np.mean(score_array, axis=0)
    print(avg_score)
    df = pd.DataFrame(avg_score).transpose()
    write_to_csv(str(clf), df)
```

Figure 11

The code excerpt above shows the first complete implementation of the Decision Tree classifier. This initial version of the function made use of k-fold cross validation and as such is structured differently to the later versions of the classifier models that were implemented. It is also important to highlight the use of *Stratified K-Fold* here, which is a variant of k-fold cross validation that keeps the same proportion of labels across the splits in each section of data used. What this means is that if there is a 45/55 split overall between the two labels in the corpus, the splits will echo this and have a 45/55 split in labels. The function is parsed the data to train the model along with a set of labels. The labels have to be converted to a NumPy array in order to be in the correct format for use with k-fold cross validation. Once this is done, the data and labels are split into however many splits are desired for this particular run of the experiment - in this case, 3 is chosen. A for loop is run for the number of splits with the results of each run of the experiment being appended to the “score_array”, from which an average of the results will be calculated once all of the splits have concluded. The model is retrained every time a split is run with new data as with each new split the section of the corpus used for training has changed. The average of all the results is then printed to the command line for verbosity. Next, the results are formatted in preparation to be written to the results.csv file by converting the results into a Pandas data frame. Finally, the “write_to_csv” function is called with the classifier used and its parameters and the results of the experiment.

As stated previously this implementation of the Decision Tree classifier, although effective in providing useful results at the beginning of the project, was updated once the setup of the experiments changed upon gaining access to the “test” data set. The final iteration of the Decision Tree classifier phased out the use of k-fold cross validation as it was no longer necessary.



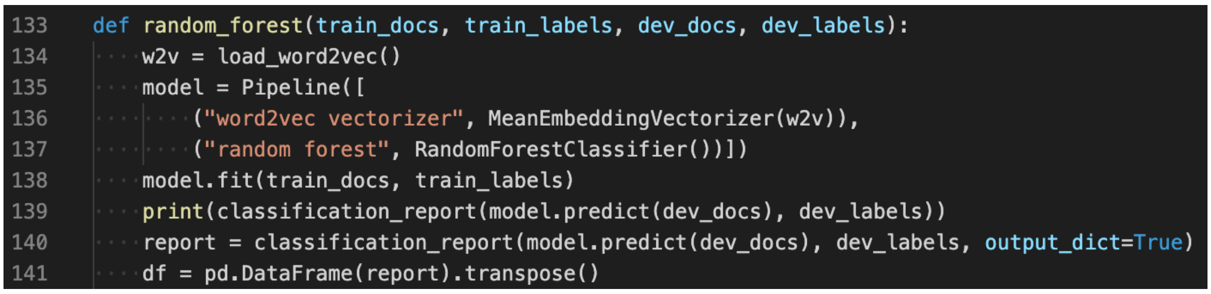
```
121  def decision_tree(train_docs, train_labels, dev_docs, dev_labels):
122      w2v = load_word2vec()
123      model = Pipeline([
124              ("word2vec vectorizer", MeanEmbeddingVectorizer(w2v)),
125              ("decision tree", tree.DecisionTreeClassifier())])
126      model.fit(train_docs, train_labels)
127      print(classification_report(model.predict(dev_docs), dev_labels))
128      report = classification_report(model.predict(dev_docs), dev_labels, output_dict=True)
129      df = pd.DataFrame(report).transpose()
130      write_to_csv(str(model["decision tree"]), df)
```

Figure 12

The excerpt above is the final iteration of the Decision Tree classifier. As shown previously in this report, this implementation of the classifier makes use of the Pipeline feature from the Sci-kit Learn library. As well as this, this function makes use of the Word2Vec vectorizer. The function is parsed the train docs and labels required to train the classifier model, as well as the dev docs and labels required to make predictions and grade how accurate those predictions were. Next, the Word2Vec model is loaded using the function “load_word2vec” that I have shown previously. Following on from this, the model is trained and predictions are made, which are printed out in the form of the “classification report” which is a method from the Sci-kit Learn library that returns values such as precision, recall, and f-1 score for the predictions made. Line 128 collects the classification report in the form of a dictionary, using the “output_dict=true” flag to achieve this. Using two different formats for the classification report was necessary due to the fact that the dictionary format is needed to write the results to a CSV file, but if you print this format out to the console it is very difficult to read, thus resulting in the use of two different formats. Finally, the classification report is converted to a Pandas data frame as we have seen already, before being parsed to the “write_to_csv” function along with the classifier parameters which are called from the Pipeline object.

4.3.2 Random Forest

The next form of classifier that was implemented was an upgraded version of the Decision Tree, the Random Forest classifier. Similar to the Decision Tree, the way that this classifier was implemented changed over the course of the project for the same reasons stated previously. Due to the almost exact similarity between the implementations of the two outdated functions, the outdated version of the Random Forest classifier function will not be shown in this paper.



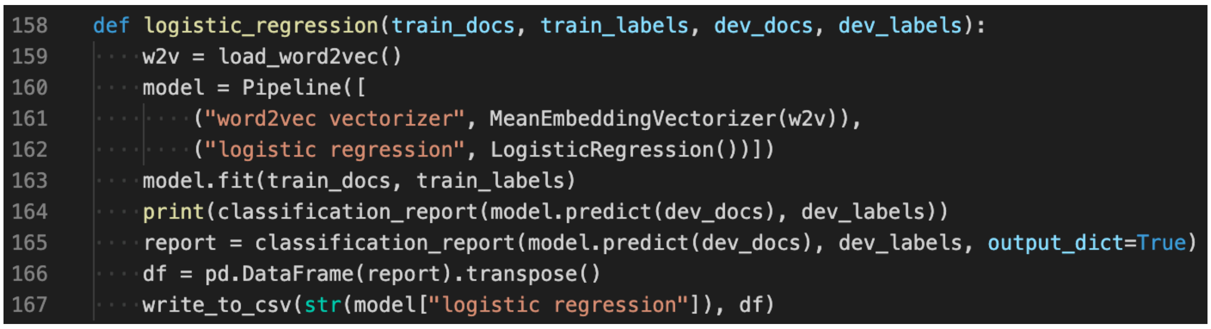
```
def random_forest(train_docs, train_labels, dev_docs, dev_labels):
    w2v = load_word2vec()
    model = Pipeline([
            ("word2vec vectorizer", MeanEmbeddingVectorizer(w2v)),
            ("random forest", RandomForestClassifier())])
    model.fit(train_docs, train_labels)
    print(classification_report(model.predict(dev_docs), dev_labels))
    report = classification_report(model.predict(dev_docs), dev_labels, output_dict=True)
    df = pd.DataFrame(report).transpose()
```

Figure 13

As can be seen in the code excerpt above, the Random Forest and Decision Tree functions are broadly similar in the way that they are implemented. This is mainly due to two reasons. First, the way in which these two classifiers work is broadly the same, with the Random Forest classifier just being a grouping of a number of Decision Trees together. Second, with the use of the Pipeline feature from Sci-kit Learn, the process of implementing different classifier models is greatly simplified and largely consists of changing single lines of code usually. The function flows the same way as the Decision Tree function as the two data sets and their labels are parsed to the function. Next, the Word2Vec model is loaded to use as the form of feature extraction for this particular experiment. The model is trained and two different formats of classification report are generated for the same reasons as previously discussed. Finally, this information is formatted for recording in the results file and the "write_to_csv" function is called. Line 137 and Line 142 are the main differences in-between this and the Decision Tree function, where Random Forest is used in place of the Decision Tree.

4.3.3 Logistic Regression

The third classifier that was implemented as a part of this project was Logistic Regression. This classifier was implemented during the latter stages of the project so was never implemented making use of k-fold cross validation, so the implementation shown is the first and final iteration of the Logistic Regression function. Once again, I made use of the Sci-kit Learn Pipeline feature[31] to structure the function, and as such the majority of the code is similar to functions that have already been shown in this report.



```
158  def logistic_regression(train_docs, train_labels, dev_docs, dev_labels):
159      w2v = load_word2vec()
160      model = Pipeline([
161              ("word2vec vectorizer", MeanEmbeddingVectorizer(w2v)),
162              ("logistic regression", LogisticRegression())])
163      model.fit(train_docs, train_labels)
164      print(classification_report(model.predict(dev_docs), dev_labels))
165      report = classification_report(model.predict(dev_docs), dev_labels, output_dict=True)
166      df = pd.DataFrame(report).transpose()
167      write_to_csv(str(model["logistic regression"]), df)
```

Figure 14

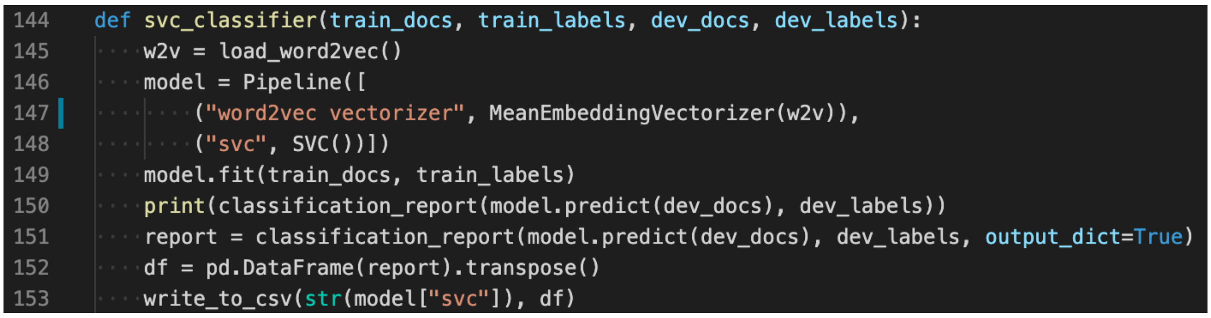
Above is the code excerpt showing the implementation for the Logistic Regression classifier. As stated previously, it is broadly similar to the Decision Tree and Random Forest functions and follows the same flow with the same requirements and outputs. The only major changes in this function are on Line 162 and Line 167, which both contain the code specific to the Logistic Regression implementation.

4.3.4 SVC

The final classifier model that was implemented was the Support Vector Machine classifier. Similar to the Logistic Regression function, SVC was implemented in the latter stages of the project thus was never implemented using k-fold cross validation. Furthermore, SVC was implemented in two different formats: one format making use of the

[31] https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

Word2Vec form of feature extraction and one format making use of “Term frequency–inverse document frequency” or TF/IDF. The implantation of SVC using TF/IDF has already been discussed earlier in this paper therefore I will focus on the implementation of SVC making use of the Word2Vec form of feature extraction.



```
144  def svc_classifier(train_docs, train_labels, dev_docs, dev_labels):
145      w2v = load_word2vec()
146      model = Pipeline([
147          ("word2vec vectorizer", MeanEmbeddingVectorizer(w2v)),
148          ("svc", SVC())])
149      model.fit(train_docs, train_labels)
150      print(classification_report(model.predict(dev_docs), dev_labels))
151      report = classification_report(model.predict(dev_docs), dev_labels, output_dict=True)
152      df = pd.DataFrame(report).transpose()
153      write_to_csv(str(model["svc"]), df)
```

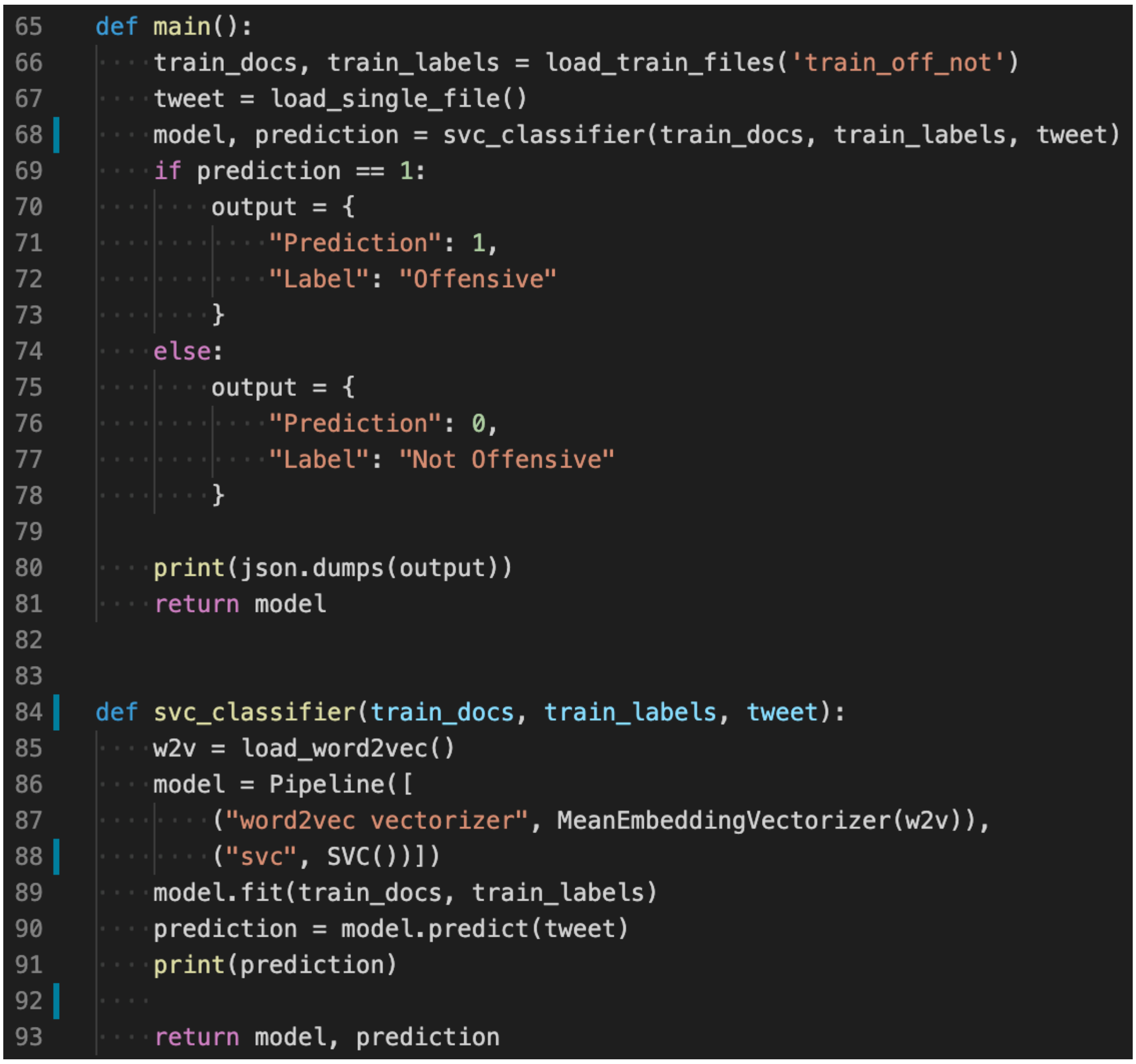
Figure 15

Again, this implementation is broadly similar to others that have been discussed already in this paper. The only changes in this block of code to others are on Line 148 and Line 153 which contain specific references to the classifier used.

4.4 Single Tweet Program V1

Once the experiments had been run and the best methods to make the most accurate predictions regarding whether a tweet was offensive or not became clear, a decision was made by the author to implement a program that took a single tweet as an input and returned a prediction for that tweet. The reason for this is that making predictions for large data sets often has little use in terms of real-world application; what is much more likely is the need for a single tweet to be examined for offensive content. This program would be much more readily adaptable into a REST API that would be of much greater use to any interested parties. The single tweet program recycles a large amount of code from the main code used to perform the experiments mentioned previously. Most of the recycled code is data handling or copying of the Word2Vec feature extraction methods. There are some differences in those functions however, for example a new function had to be written to load and format a file with a single tweet in as opposed to a file with a substantial number of tweets in. It must be said now that although the SVC classifier combined with TF/IDF feature extraction provided the most accurate prediction, the trade off in vastly increased run time did not make sense for this particular program. With the idea of making this readily adaptable for a REST API, a 20-minute run time to return a single prediction didn’t seem sensible. As a result, I have decided to use the second-best combination of the

SVC classifier combined with the Word2Vec feature extraction, which returns a result in 1 minute with only slightly worse accuracy.



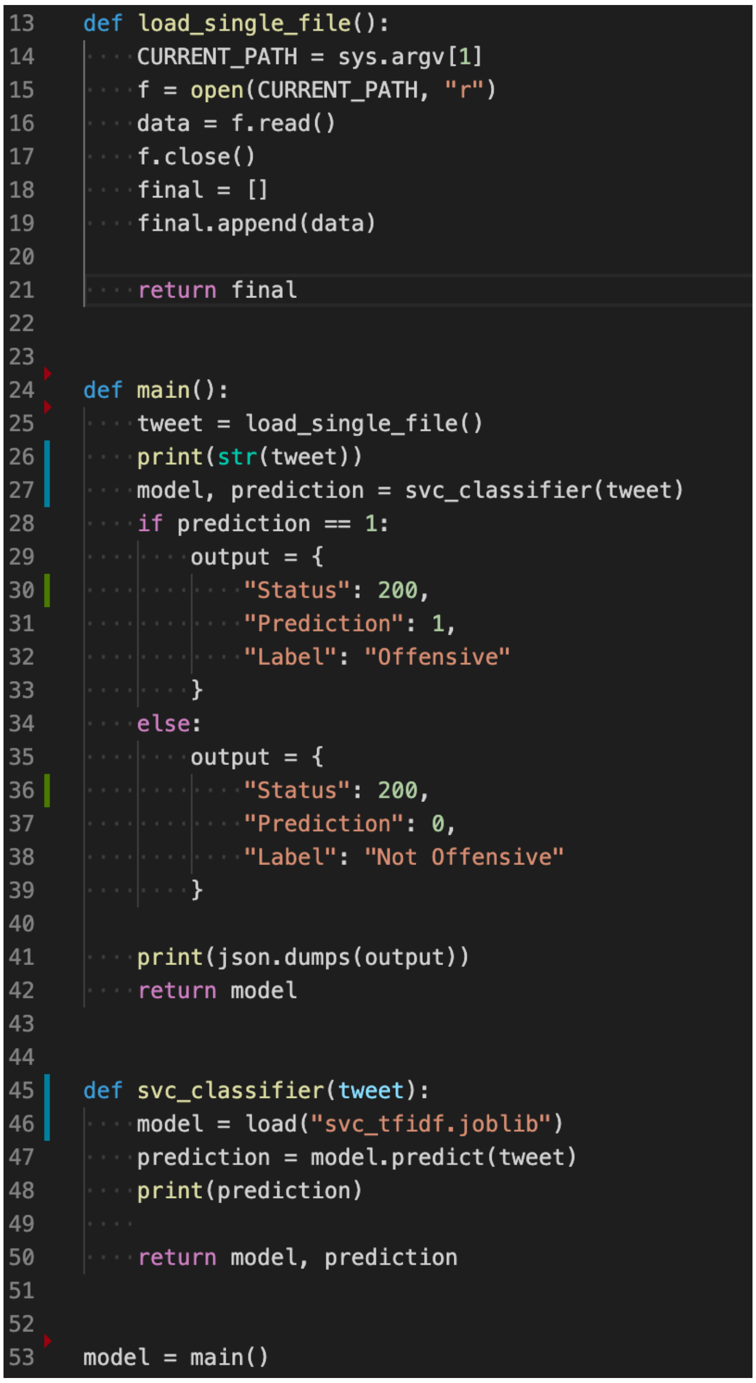
```python
def main():
    train_docs, train_labels = load_train_files('train_off_not')
    tweet = load_single_file()
    model, prediction = svc_classifier(train_docs, train_labels, tweet)
    if prediction == 1:
        output = {
            "Prediction": 1,
            "Label": "Offensive"
        }
    else:
        output = {
            "Prediction": 0,
            "Label": "Not Offensive"
        }

    print(json.dumps(output))
    return model


def svc_classifier(train_docs, train_labels, tweet):
    w2v = load_word2vec()
    model = Pipeline([
        ("word2vec vectorizer", MeanEmbeddingVectorizer(w2v)),
        ("svc", SVC())])
    model.fit(train_docs, train_labels)
    prediction = model.predict(tweet)
    print(prediction)

    return model, prediction
```

Figure 16

The code excerpt above shows the main variations in the single tweet program in comparison to the main program. The training data is still loaded using the same function, but the next line loads a single tweet that will be used to make a prediction on. Next, the classifier function is called with the training data and the tweet to be categorised as offensive/not offensive. Once the model is trained and a prediction is made, instead of generating a classification report, the prediction is printed for verbosity and then returned along with the model. The prediction takes the form of “1” or “0”, and as based on the prediction a dictionary is created for either offensive/not offensive. This model is then returned as a JSON blob, which is the output expected if this were to be used as a REST API.

4.5 Single Tweet Program V2

In the first version of the single tweet program it was highlighted that, due to run time constraints, the combination of SVC and TF/IDF feature extraction could not be used despite the fact this combination provided the most accurate results. Later in the project, a solution was found to this problem that involved saving a pre-trained model and exporting it into a file. By doing this and loading the subsequent pre-trained model in the single tweet program, the run time was drastically cut down therefore the use of SVC with TF/IDF was much more practical.

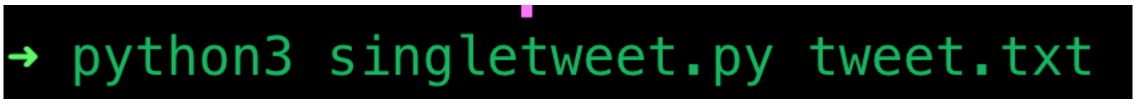


```python
def load_single_file():
    CURRENT_PATH = sys.argv[1]
    f = open(CURRENT_PATH, "r")
    data = f.read()
    f.close()
    final = []
    final.append(data)

    return final


def main():
    tweet = load_single_file()
    print(str(tweet))
    model, prediction = svc_classifier(tweet)
    if prediction == 1:
        output = {
            "Status": 200,
            "Prediction": 1,
            "Label": "Offensive"
        }
    else:
        output = {
            "Status": 200,
            "Prediction": 0,
            "Label": "Not Offensive"
        }

    print(json.dumps(output))
    return model


def svc_classifier(tweet):
    model = load("svc_tfidf.joblib")
    prediction = model.predict(tweet)
    print(prediction)

    return model, prediction


model = main()
```

Figure 17

As you can see in the code excerpt above, the final version of this program is significantly smaller due to the removal of any need to train a classifier model. As such, the code is short enough that it can be shown in one screenshot. Anything that was needed to load training files has been removed as well as any code that was used to load the pre-trained Word2Vec model. As well as this, any lines actually training a Sci-kit learn model ready to make predictions have been removed. All that is left is the above, showing the loading of the tweet from a text file and formatted slightly to allow for Sci-kit learn to load the tweet and read it. Finally, the pre-trained model using SVC and TF/IDF is loaded and used to form the prediction. This prediction is then collected and formatted into a dictionary before being converted to JSON and outputted to the command line for verbosity.

In the screenshot below, you can see how the program is run from the command line with the program being called, followed by location of the text file containing the tweet that will be analysed, and a prediction be made as to whether the tweet is offensive or not offensive.



```
→ python3 singletweet.py tweet.txt
```

Figure 18

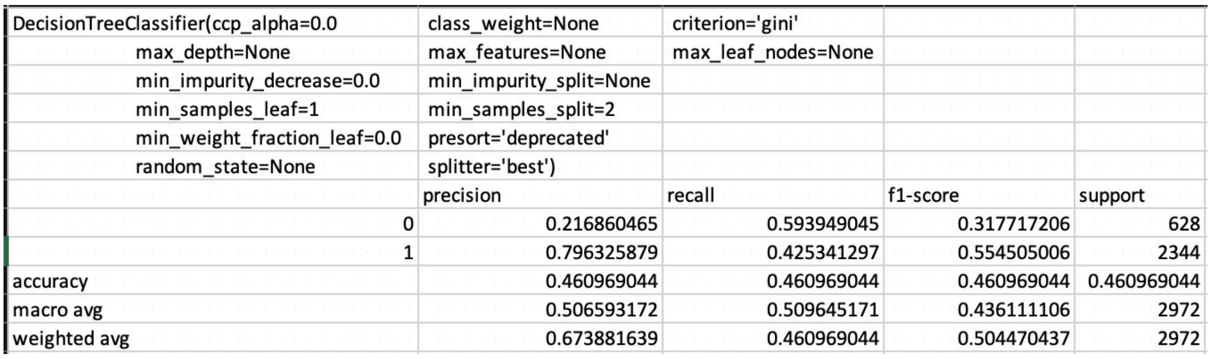
This is a significant improvement to the single tweet program which allows it to make use of the more accurate combination of SVC and TF/IDF feature extraction with a much faster runtime of 1 minute, instead of the 20-30 minutes the run takes when including model training time. This improvement in the system was made possible due to the inclusion of the Joblib model that allows for saving and loading of Sci-kit learn models once they have been trained.

5. Results and Evaluation

5.1 Experiment Results

In this section the results that were collected throughout the duration of the project will be presented and analysed and summary outcomes will be drawn. All of the following results are collected by training the classifier model using the train data set and predictions are then made on the test data set. These tests were run at the end of the project and only run once so to preserve the best practice of the experimental setup. Each model implemented was run with each form of feature extraction. It should also be mentioned now that in all of the following results “0” means “Not Offensive” and “1” means “Offensive”. As well as this, the test data set that we are making predictions on is made up of 1252 offensive tweets and 1720 not offensive tweets.

The first form of feature extraction implemented was the Count Vectorizer and although this was the simplest form of feature extraction that was implemented, it also produced the least accurate results across all of the four classifier models that were implemented. The first model to be examined is the Decision Tree which was also the first model to be implemented.

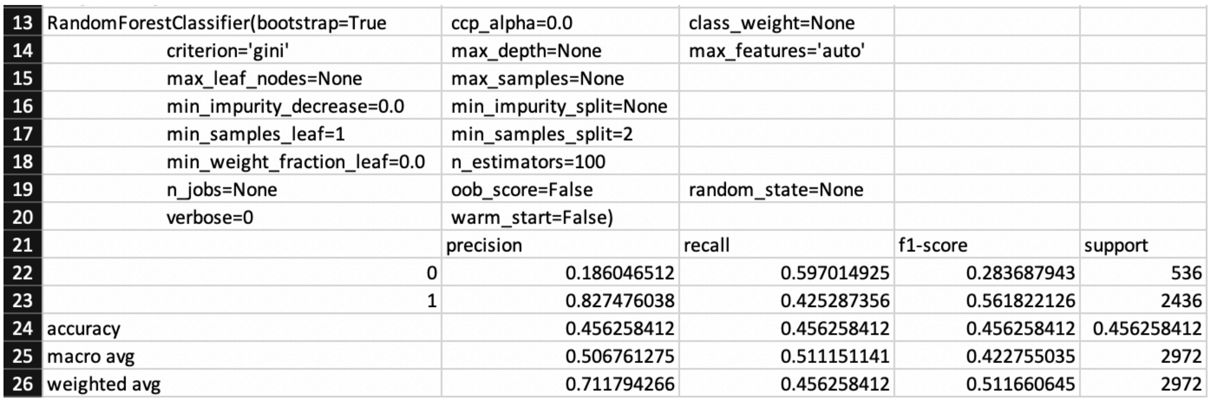


DecisionTreeClassifier(ccp_alpha=0.0	class_weight=None	criterion='gini'		
max_depth=None	max_features=None	max_leaf_nodes=None		
min_impurity_decrease=0.0	min_impurity_split=None			
min_samples_leaf=1	min_samples_split=2			
min_weight_fraction_leaf=0.0	presort='deprecated'			
random_state=None	splitter='best')			
	precision	recall	f1-score	support
0	0.216860465	0.593949045	0.317717206	628
1	0.796325879	0.425341297	0.554505006	2344
accuracy	0.460969044	0.460969044	0.460969044	0.460969044
macro avg	0.506593172	0.509645171	0.436111106	2972
weighted avg	0.673881639	0.460969044	0.504470437	2972

Figure 19

As can be seen in the results above, the Decision Tree combined with a Count Vectorizer as the form of feature extraction does not perform well, with an overall F-1 Score of 0.46. This shows that over half of the predictions made by this model were incorrect. If we examine the results further, we can see that this particular setup scores very highly in Precision when classifying tweets as offensive, with a score of 0.80, but the model also scores very poorly in terms of Precision when classifying not offensive tweets, with a score of 0.22. We can infer from these results that this particular combination of model and classifier does not form a particularly effective form of classifier. The extremely low Precision score when classifying not offensive tweets combined with the low number of tweets actually classified as not offensive suggests that, even when the model does classify a tweet as not offensive, it is actually a false positive and should be offensive. If we examine the way this model classifies offensive tweets, the high rate of Precision combined with an average Recall scores suggests that this model does not return a large number of results when classifying offensive tweets, but the majority of the predictions it makes on this group are correct. The key takeaway from this model is that it performs better when classifying offensive tweets, as shown by the comparison in F-1 Scores between not offensive and offensive of 0.32 and 0.55 respectively. This improvement when classifying offensive tweets is caused by the rates of return between not offensive and offensive, with not offensive having a much higher Recall score and thus returning more results, but also having a much lower Precision score which suggests that the model is getting a large number of these predictions incorrect.

The next classifier that will be examined when combined with a Count Vectorizer is the Random Forest. This classifier performed almost exactly the same as the Decision Tree overall in terms of F-1 Score but had some interesting differences in Precision and Recall scores, due to the difference in the way the classifiers function.

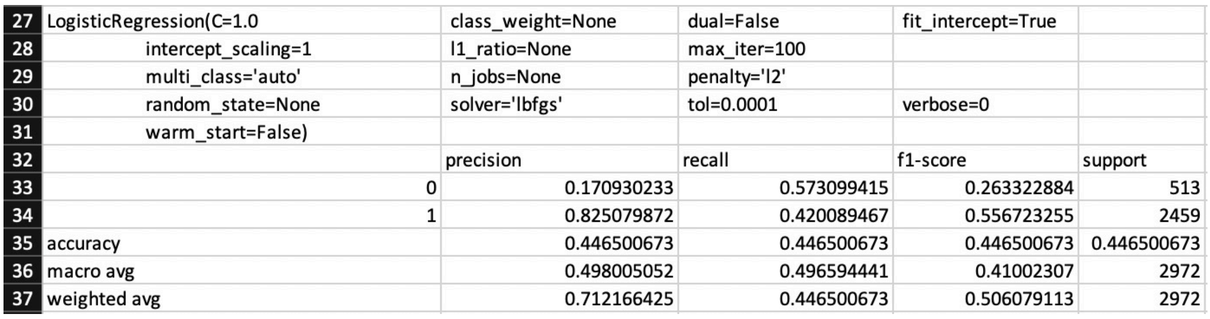


13	RandomForestClassifier(bootstrap=True	ccp_alpha=0.0	class_weight=None		
14	criterion='gini'	max_depth=None	max_features='auto'		
15	max_leaf_nodes=None	max_samples=None			
16	min_impurity_decrease=0.0	min_impurity_split=None			
17	min_samples_leaf=1	min_samples_split=2			
18	min_weight_fraction_leaf=0.0	n_estimators=100			
19	n_jobs=None	oob_score=False	random_state=None		
20	verbose=0	warm_start=False)			
21		precision	recall	f1-score	support
22	0	0.186046512	0.597014925	0.283687943	536
23	1	0.827476038	0.425287356	0.561822126	2436
24	accuracy	0.456258412	0.456258412	0.456258412	0.456258412
25	macro avg	0.506761275	0.511151141	0.422755035	2972
26	weighted avg	0.711794266	0.456258412	0.511660645	2972

Figure 20

As we can see, the overall F-1 Score is exactly the same as the Decision Tree with 0.46. Again, this model had largely similar Recall and Precision scores for not offensive tweets, suggesting that the model is returning a large number of results for not offensive but is getting these predictions almost entirely wrong due to the large number of false positives in the predictions that it has made. The converse is true when it comes to offensive tweets, with a lower Recall score of 0.43 and a much higher Precision score of 0.83, suggesting that the model is returning a smaller proportion of offensive tweets but the predictions for these tweets are much more accurate, with a low number of false positives in the predictions. Despite this, the results are broadly similar to the Decision Tree classifier and mostly show little improvement despite the expectation that this classifier would improve results. These metrics combine to provide the predictable results of the Random Forest having a worse F-1 Score when it comes to classifying not offensive tweets with a score of 0.2 but an improved F-1 Score when it comes to classifying offensive tweets with a score of 0.56. Overall, the Random Forest classifier when combined with a Count Vectorizer has an F-1 Score of 0.46 which is not as accurate as could be hoped.

The third classifier that will be examined is Logistic Regression. The results for this classifier when combined with a Count Vectorizer can be seen below.

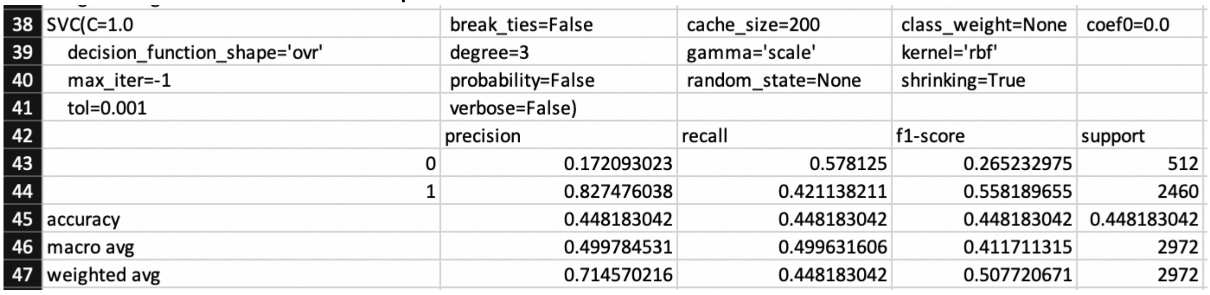


27	LogisticRegression(C=1.0	class_weight=None	dual=False	fit_intercept=True	
28	intercept_scaling=1	l1_ratio=None	max_iter=100		
29	multi_class='auto'	n_jobs=None	penalty='l2'		
30	random_state=None	solver='lbfgs'	tol=0.0001	verbose=0	
31	warm_start=False)				
32		precision	recall	f1-score	support
33	0	0.170930233	0.573099415	0.263322884	513
34	1	0.825079872	0.420089467	0.556723255	2459
35	accuracy	0.446500673	0.446500673	0.446500673	0.446500673
36	macro avg	0.498005052	0.496594441	0.41002307	2972
37	weighted avg	0.712166425	0.446500673	0.506079113	2972

Figure 21

As can be seen in Figure 21, this classifier had nearly identical results to the Random Forest classifier in essentially every field. The only differences between the two are for Precision when classifying not offensive tweets and Recall when classifying not offensive tweets. In both of these cases, Logistic Regression performed slightly poorer than the Random Forest with scores of 0.17 and 0.57 respectively. This led to a worse F-1 Score when predicting not offensive tweets, suggesting that this model is the worst so far at classifying not offensive tweets when compared to the other two classifiers that have been examined previously. Equally, Logistic Regression has the weakest overall F-1 Score of any classifier so far when combined with a Count Vectorizer.

The final classifier to be examined when combined with a Count Vectorizer is the SVC classifier. The results of this experiment can be seen below.



38	SVC(C=1.0	break_ties=False	cache_size=200	class_weight=None	coef0=0.0
39	decision_function_shape='ovr'	degree=3	gamma='scale'	kernel='rbf'	
40	max_iter=-1	probability=False	random_state=None	shrinking=True	
41	tol=0.001	verbose=False)			
42		precision	recall	f1-score	support
43	0	0.172093023	0.578125	0.265232975	512
44	1	0.827476038	0.421138211	0.558189655	2460
45	accuracy	0.448183042	0.448183042	0.448183042	0.448183042
46	macro avg	0.499784531	0.499631606	0.411711315	2972
47	weighted avg	0.714570216	0.448183042	0.507720671	2972

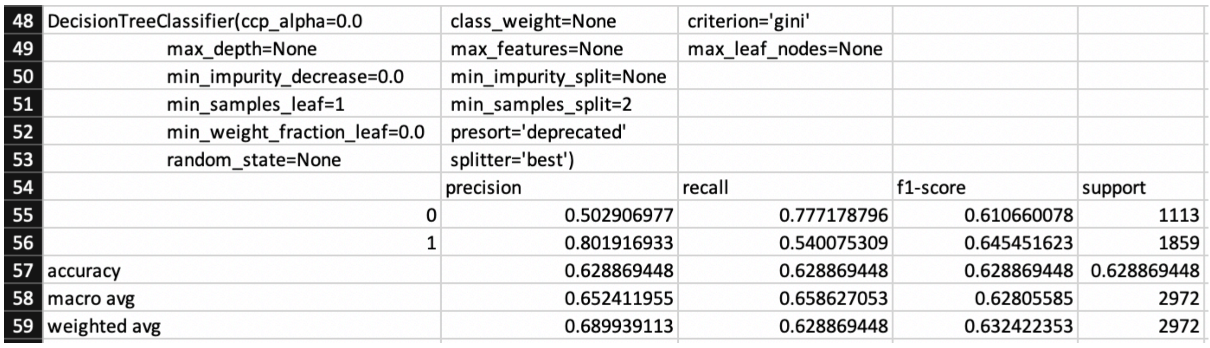
Figure 22

These results are essentially identical to the results for Logistic Regression, and as such further analysis is not required. All that can be said for this experiment is that the same conclusions drawn for Logistic Regression apply for SVC as well.

In conclusion, all 4 of the different classifier models that were implemented throughout the duration of this project performed very similarly when combined with the count vectorizer as the form of feature extraction used. All of the results collected from these experiments are relatively poor, with quite a low degree of accuracy overall in the predictions made by these models. In particular, all of the models above suffered from a combination of high recall/low precision when it came to classifying not offensive tweets. As mentioned previously, this is due to the fact that the models are returning too many results when looking for not offensive tweets, and any predictions of not offensive are overwhelmingly false positive. The conclusion to draw from these results is that the Count Vectorizer is not an effective method of feature extraction and cannot be relied upon to generate accurate results. It must be said that this could be owing to the parameters used for the Count Vectorizer; that of a maximum of 50 features and only including basic stop words, but based on the research done it seemed that the Count Vectorizer was a basic form of feature extraction and a large amount of time spent changing parameters would only yield a small improvement in accuracy scores and as such was not worth it.

The next form of feature extraction that was implemented was using Word2Vec to create word embeddings, that would then be used as the features for models to use as part of training. It has been discussed previously that Word2Vec word embeddings were created in two different ways throughout the project so I must mention now that all of the following results are making use of the larger pre-trained Word2Vec model, due to the poor quality of results provided by the Word2Vec model trained on the training corpus. Following the same pattern as with the Count Vectorizer, I will analyse how this form of feature extraction impacted the results of each different classifier model, starting with the Decision Tree.

Instantly it can be seen that there has been a huge improvement in results across

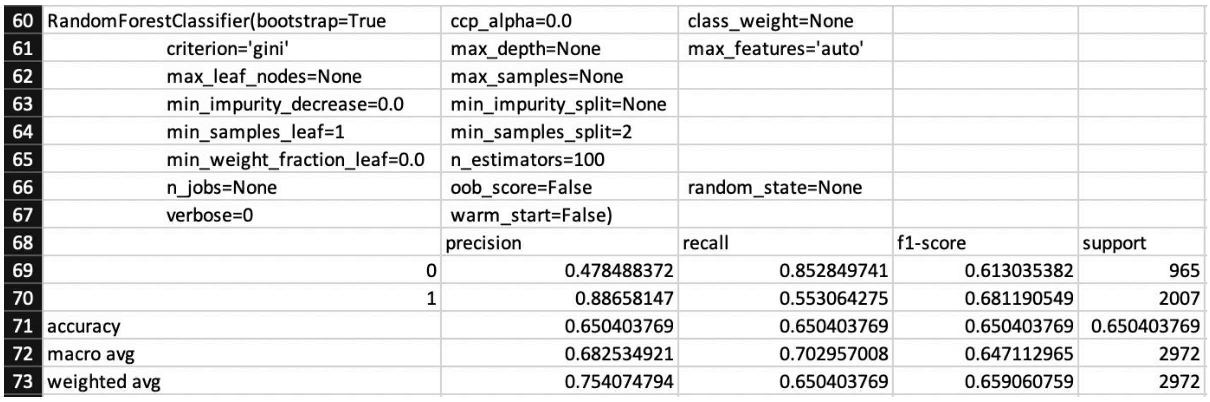


48	DecisionTreeClassifier(ccp_alpha=0.0	class_weight=None	criterion='gini'		
49	max_depth=None	max_features=None	max_leaf_nodes=None		
50	min_impurity_decrease=0.0	min_impurity_split=None			
51	min_samples_leaf=1	min_samples_split=2			
52	min_weight_fraction_leaf=0.0	presort='deprecated'			
53	random_state=None	splitter='best')			
54		precision	recall	f1-score	support
55	0	0.502906977	0.777178796	0.610660078	1113
56	1	0.801916933	0.540075309	0.645451623	1859
57	accuracy	0.628869448	0.628869448	0.628869448	0.628869448
58	macro avg	0.652411955	0.658627053	0.62805585	2972
59	weighted avg	0.689939113	0.628869448	0.632422353	2972

Figure 23

Precision, Recall, and F-1 Score for both not offensive and offensive tweets due to the use of the Word2Vec word embeddings as features. When classifying not offensive tweets, there is a clear increase in both Recall and Precision. The Precision score for not offensive tweets is 0.50, which is still only an average score, but coupled with the higher Recall score of 0.78 it suggests that this model is better at correctly classifying not offensive tweets and has less false positives in its predictions, although half of its predictions are still false positives. There is also a minor improvement in Precision and Recall for offensive tweets, suggesting a slight improvement in returning more results for offensive tweets but also that these results returned are mostly correct predictions with few false positives. Overall, there is a significant improvement in the accuracy of the model when compared to the same model with a Count Vectorizer as its form of feature extraction, with the averaged F-1 Score of not offensive and offensive tweets reaching 0.63.

The next classifier that will be examined is the Random Forest classifier. Following a similar pattern to the Decision Tree, when combined with Word2Vec feature extraction there was an improvement in the accuracy of predictions.

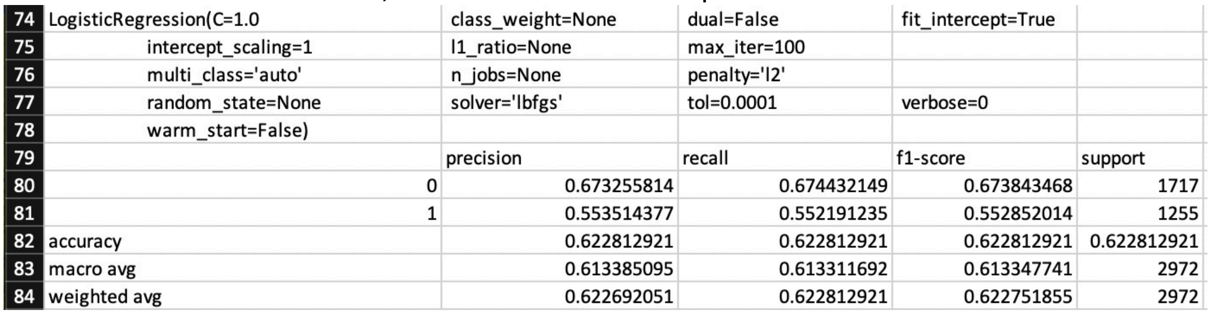


60	RandomForestClassifier(bootstrap=True	ccp_alpha=0.0	class_weight=None		
61	criterion='gini'	max_depth=None	max_features='auto'		
62	max_leaf_nodes=None	max_samples=None			
63	min_impurity_decrease=0.0	min_impurity_split=None			
64	min_samples_leaf=1	min_samples_split=2			
65	min_weight_fraction_leaf=0.0	n_estimators=100			
66	n_jobs=None	oob_score=False	random_state=None		
67	verbose=0	warm_start=False)			
68		precision	recall	f1-score	support
69	0	0.478488372	0.852849741	0.613035382	965
70	1	0.88658147	0.553064275	0.681190549	2007
71	accuracy	0.650403769	0.650403769	0.650403769	0.650403769
72	macro avg	0.682534921	0.702957008	0.647112965	2972
73	weighted avg	0.754074794	0.650403769	0.659060759	2972

Figure 24

Again, there is a significant improvement in all fields when compared to the same model using the Count Vectorizer as its form of feature extraction. This particular model has a high recall for not offensive tweets, but a low precision. As has been previously discussed, this combination of high recall/low precision suggests that while this model returns a large number of results for not offensive tweets, the Precision of these results is poor with a score of 0.48, suggesting the majority are false positives. The converse is true when it comes to offensive tweets, where a much lower rate of Recall at 0.55 is balanced with a very high Precision score of 0.89. This suggests that while this model is good at returning not offensive tweets, usually the tweets it has returned and classed as not offensive are false positive and are actually offensive tweets. It also means that while this model is excellent at not classifying not offensive tweets as offensive, it does miss a large number of offensive tweets, which is represented by the low Recall score. Overall, it is clear that the Random Forest classifier when combined with Word2Vec feature extraction still performs slightly better at classifying offensive tweets than the same model when combined with a Count Vectorizer, with an F-1 Score of 0.68 compared to 0.61 for not offensive tweets. When averaged, these two scores form the overall accuracy score as 0.65, which is a significant improvement from the 0.46 score when this classifier was used with a Count Vectorizer.

The next classifier that will be examined is Logistic Regression. It would be reasonable to expect, based on the last two sets of results when using Word2Vec as the method of feature extraction, that there will be an improvement in results for this classifier.

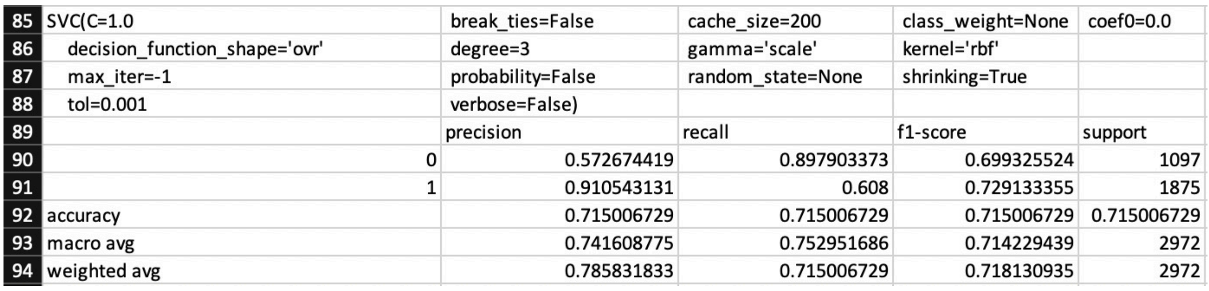


74	LogisticRegression(C=1.0	class_weight=None	dual=False	fit_intercept=True	
75	intercept_scaling=1	l1_ratio=None	max_iter=100		
76	multi_class='auto'	n_jobs=None	penalty='l2'		
77	random_state=None	solver='lbfgs'	tol=0.0001	verbose=0	
78	warm_start=False)				
79		precision	recall	f1-score	support
80	0	0.673255814	0.674432149	0.673843468	1717
81	1	0.553514377	0.552191235	0.552852014	1255
82	accuracy	0.622812921	0.622812921	0.622812921	0.622812921
83	macro avg	0.613385095	0.613311692	0.613347741	2972
84	weighted avg	0.622692051	0.622812921	0.622751855	2972

Figure 25

As expected, there is an improvement in results when compared to the use of Logistic Regression combined with a Count Vectorizer. There is a significant improvement in the overall classification of not offensive tweets, with both Precision and Recall improving with regard to these tweets. Conversely, while Recall for offensive tweets did improve, there was a significant drop in Precision when classifying offensive tweets. What is interesting about the results for this classifier when combined with Word2Vec, is the balance between Precision and Recall for both not offensive and offensive tweets. For not offensive tweets, there is still a higher Recall score than for offensive tweets, with 0.67 compared to 0.55 respectively. There is a significantly improved Precision score, however, of 0.67. This suggests that this model is now returning a smaller number of not offensive tweets and the predictions that are made by the model, with regard to not offensive tweets, are mostly accurate. If we examine offensive tweets, we can see that there has been a significant drop in Precision, suggesting a large increase in the number of false positive predictions being made when classifying offensive tweets. This drop in Precision is balanced out by an increase in Recall, suggesting that this model has improved at returning more results for offensive tweets, which is balanced with more of the predictions being inaccurate. All of these results combine to create an overall increase in accuracy of the predictions being made, with an F-1 Score of 0.62.

The final classifier to be examined when using Word2Vec is the SVC classifier. The expectation for this experiment is a significant improvement in the accuracy of predictions made by this model.



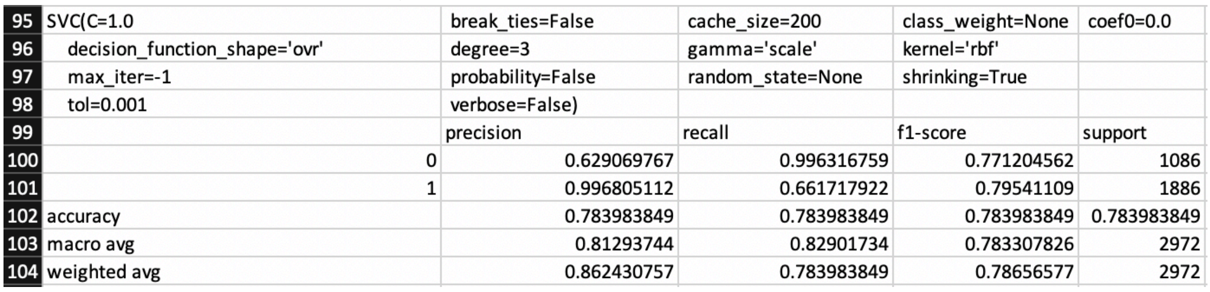
85	SVC(C=1.0	break_ties=False	cache_size=200	class_weight=None	coef0=0.0
86	decision_function_shape='ovr'	degree=3	gamma='scale'	kernel='rbf'	
87	max_iter=-1	probability=False	random_state=None	shrinking=True	
88	tol=0.001	verbose=False)			
89		precision	recall	f1-score	support
90	0	0.572674419	0.897903373	0.699325524	1097
91	1	0.910543131	0.608	0.729133355	1875
92	accuracy	0.715006729	0.715006729	0.715006729	0.715006729
93	macro avg	0.741608775	0.752951686	0.714229439	2972
94	weighted avg	0.785831833	0.715006729	0.718130935	2972

Figure 26

As can be seen by the results, the expectations for this experiment were met and there was a significant improvement in results, with the overall accuracy being the best results seen so far in this project. If we examine the results for not offensive tweets, we can see a huge improvement in Recall with a score of 0.90. This high Recall score, combined with a much lower Precision score of 0.57, further suggests that this model is returning a large number of tweets as not offensive, but a large portion of the results returned are false positives and have been classified incorrectly. Similar to the classifiers that we have examined previously when using Word2Vec as the method of feature extraction, this classifier is much better at classifying offensive tweets. The SVC model still has a high precision/low recall trade off when classifying offensive tweets, but it has much better scores than other models. The Recall score of 0.61 suggests that it is returning a larger number of offensive tweets than other models and this combined with the very high Precision score of 0.91 suggests that those results returned are correct to a near perfect degree, with a very low number of false positive predictions made. These results combine to have an overall accuracy F-1 Score of 0.72, which is the strongest score seen throughout this project so far.

In conclusion, using a Word2Vec model trained on a large corpus to create word embeddings, and using those word embeddings as features for the classifier models, is a much more effective form of feature extraction than a Count Vectorizer. Across all four of the classifiers implemented as part of this project, there were significant improvements in the overall accuracy of predictions made. There is a common drawback, however, to using Word2Vec word embeddings as features that appears to be difficulty when classifying not offensive tweets. All four classifiers, to varying degrees, had high recall/low precision problems when classifying not offensive tweets. As well as this, all four classifiers had the converse problem of low recall/high precision when classifying offensive tweets. This suggests that when using Word2Vec word embeddings as features, this particular Word2Vec model provides features that lead classifier models to overclassify tweets as not offensive, resulting in large numbers of false positives. That being said, this form of feature extraction is still a significant improvement over the Count Vectorizer, and when combined with the SVC classifier provided the best results seen so far in this project.

The last form of feature extraction that was implemented as part of the project was TF/IDF. This form of feature extraction was implemented near the end of the project and as such a decision was made to only implement it for the most successful classifier model so far, in an attempt to achieve the strongest results possible. As a result of this, the only classifier model TF/IDF was implemented with was the SVC classifier.



95	SVC(C=1.0	break_ties=False	cache_size=200	class_weight=None	coef0=0.0
96	decision_function_shape='ovr'	degree=3	gamma='scale'	kernel='rbf'	
97	max_iter=-1	probability=False	random_state=None	shrinking=True	
98	tol=0.001	verbose=False)			
99		precision	recall	f1-score	support
100	0	0.629069767	0.996316759	0.771204562	1086
101	1	0.996805112	0.661717922	0.79541109	1886
102	accuracy	0.783983849	0.783983849	0.783983849	0.783983849
103	macro avg	0.81293744	0.82901734	0.783307826	2972
104	weighted avg	0.862430757	0.783983849	0.78656577	2972

Figure 27

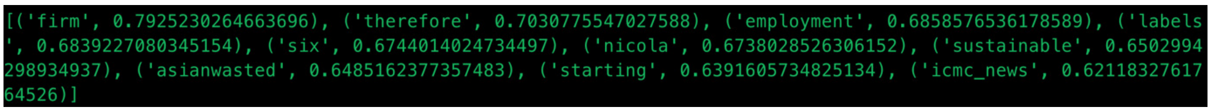
As can be seen in the above results, the combination of TF/IDF and the SVC classifier provided the best results in the project. If we examine the results for not offensive tweets, we can see a score of 0.996 which is the highest score we have seen. The results also show a Precision score of 0.63 for not offensive tweets. These two scores combined suggest that this model is returning the largest proportion of not offensive tweets yet. Despite this, the low precision score suggests that the model is still suffering from a large number of false positives being returned. This means that a large number of offensive tweets are being incorrectly classified as not offensive, which is an obvious issue. When it comes to classifying offensive tweets, this model has both an improved Recall and Precision score when compared to the same model using Word2Vec word embeddings as features. With a Recall score of 0.66, the model is better at returning results for offensive tweets but is still missing a large number. The Precision score of 0.996, however, suggests that when a tweet is returned and predicted as offensive, it is essentially guaranteed to be a correct prediction. What this means is that this model very rarely gets a prediction of an offensive tweet wrong, showing that you can trust its detections of offensive tweets, if not its detections of not offensive tweets.

In conclusion, the most accurate predictions achieved by any model were found in the combination of the SVC classifier and TF/IDF feature extraction. What I should mention is the major drawback that comes with this combination; that being the extreme increase in program run time when using this combination, from an average run time of 1 minute up to an average of 20 minutes. This trade-off is definitely worth it in this experimental setting where speed of results is not an issue, but this combination would not be an effective choice in a situation where you may want real-time feedback. One such situation that this could occur would be if you wanted to implement an automated offensive tweet detection system thus using SVC with TF/IDF feature extraction would not be an effective choice, due to having to wait 20 minutes for each tweet to be analysed and a prediction returned. Therefore, while SVC and TF/IDF is the most effective combination, it is highly situational and its use should be carefully considered based on the requirements of the system being designed.

5.2 Word2Vec Nearest Word Analysis

In the previous section I mentioned that two Word2Vec models were used in this project, one trained on the training sets provided as part of the project and one trained on a much larger corpus of tweets. In this section of the report, I will explore why a second Word2Vec model was required, by utilising the "most similar"[32] feature of the Word2Vec library. What this feature does is show the word embeddings that are "nearest" to a provided word, as well as to what degree they are near to the word that has been provided. Both Word2Vec models will be provided with the same word, in this case "bitch", to see what word embeddings are considered nearest in the two Word2Vec models.

The first model that will be examined is the model trained on the training set of data provided as part of this project, which consists of 3783 offensive tweets and 5218 not offensive tweets.



```
[('firm', 0.7925230264663696), ('therefore', 0.7030775547027588), ('employment', 0.6858576536178589), ('labels
', 0.6839227080345154), ('six', 0.6744014024734497), ('nicola', 0.6738028526306152), ('sustainable', 0.6502994
298934937), ('asianwasted', 0.6485162377357483), ('starting', 0.6391605734825134), ('icmc_news', 0.62118327617
64526)]
```

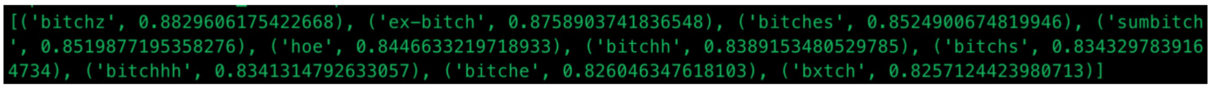
Figure 28

As you can see above, the word embeddings that are most similar to "bitch" in this model often have no relevance to words that could be considered offensive. For example, the third nearest word to "bitch" in this model is "employment". What this means is that based on these features, a classifier model would be highly likely to consider the word "employment" as an offensive word and would be more likely to classify a tweet as offensive if it contained this word. This is obviously not correct as the word employment is not deemed as offensive. As a result, the word embeddings and features generated by this Word2Vec model are, unfortunately, not particularly useful to us. This is most likely due to the small corpus size that the model was trained on as Word2Vec models function much better and provide better embeddings when trained on hundreds of thousands of tweets, as

32 https://tedboy.github.io/nlps/generated/generated/gensim.models.Word2Vec.most_similar.html

opposed to 9001 in this case. This resulted in the subsequent use of a second Word2Vec model that was trained on a much larger corpus.

In order to improve the word embeddings provided by the original model, I found and made use of a pre-trained Word2Vec model.[33] As stated above, the second Word2Vec model was trained on a much larger corpus, and as a result the new model provided much more useful word embeddings. This is the main reason why this second Word2Vec model was used as the method of feature extraction for the classifier models, and not the original model trained on the smaller corpus.



```
[('bitchz', 0.8829606175422668), ('ex-bitch', 0.8758903741836548), ('bitches', 0.8524900674819946), ('sumbitch
', 0.8519877195358276), ('hoe', 0.8446633219718933), ('bitchh', 0.8389153480529785), ('bitchs', 0.834329783916
4734), ('bitchhh', 0.8341314792633057), ('bitche', 0.826046347618103), ('bxtch', 0.8257124423980713)]
```

Figure 29

As can be seen in the above screenshot, the words that are nearest to “bitch” in the second Word2Vec model make a lot more sense. Not only are the words “nearer” to our provided word, but they could all be considered offensive. In fact, the vast majority of the most similar words to “bitch” in this model are either misspellings of “bitch” or are deliberate attempts to avoid getting caught by an abusive language detector by changing letters, for example “bxtch”. As well as this, another misogynistic term “hoe” is considered very similar to “bitch” in this model, suggesting that the two words are used together regularly in the corpus of tweets used to train this model. This screenshot is a perfect example of why this second Word2Vec model had to be used, as it provided word embeddings that were much more suitable as features and thus provided a significant boost in the accuracy of results.

5.4 Results from Single Tweet Program

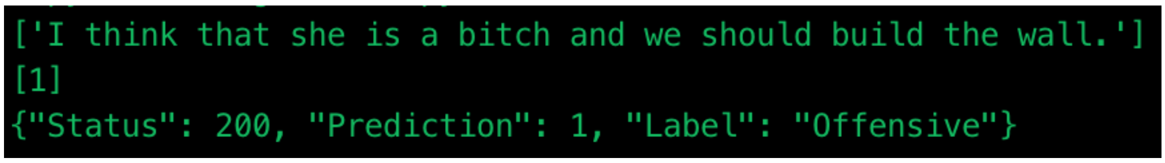
Following on from the overall results of the experiments run as part of this project, it is important to show how the outcomes from the project can be adapted and implemented into something more useful to parties interested in the outcomes of the project. This takes the form of a program that takes in a single tweet and returns a prediction of offensive/not offensive for the tweet, instead of an analysis of accuracy which is returned from all of the other functions shown so far. It has been discussed already how this program was designed and implemented - with constant thought as to how to make the outcomes of the project useful and adaptable into a system that could be implemented by a social media platform like Twitter for example.

In the first version of this program, a decision was made to make use of the second most accurate classifier model/feature extraction combination, due to the fact that using SVC with TF/IDF feature extraction would have made the system impractical due to run time. As a result, the first version of the program implemented SVC with Word2Vec word embeddings as features. As such, the system took in a tweet and returned a prediction as to if the tweet was offensive or not. Based on the results discussed previously, there is a danger with using this particular model that the prediction will not be correct, with this

[33] Camacho Collados, J., Doval, Y., Martínez-Cámara, E., Espinosa-Anke, L., Barbieri, F. and Schockaert, S., 2020. Learning cross-lingual word embeddings from Twitter via distant supervision. *ICWSM*.

combination only achieving an accuracy score of 0.72. The greatest flaw in this particular combination is its tendency to classify tweets as not offensive when they are actually offensive - a false positive prediction.

The issue with long run times was solved in the second version of the program by using Joblib to save and load pre-trained Sci-kit Learn models, thus vastly cutting down on the program run time and making it practical to implement SVC with TF/IDF feature extraction. This gave a significant improvement to the accuracy of the prediction provided by the program. Although there is still a danger that tweets classed as not offensive are false positives and are actually offensive, the danger of this occurring is reduced when compared to using SVC with Word2Vec word embeddings, as evidenced by the improved Precision score when classifying offensive tweets. As well as this, due to the extremely high Precision score when classifying offensive tweets, the program is almost guaranteed to have correctly predicted the contents of a tweet when that tweet is predicted as offensive. Overall, the inclusion of loading a pre-trained model has greatly improved the functionality of this single tweet implementation.



```
['I think that she is a bitch and we should build the wall.']
[1]
{"Status": 200, "Prediction": 1, "Label": "Offensive"}
```

Figure 30

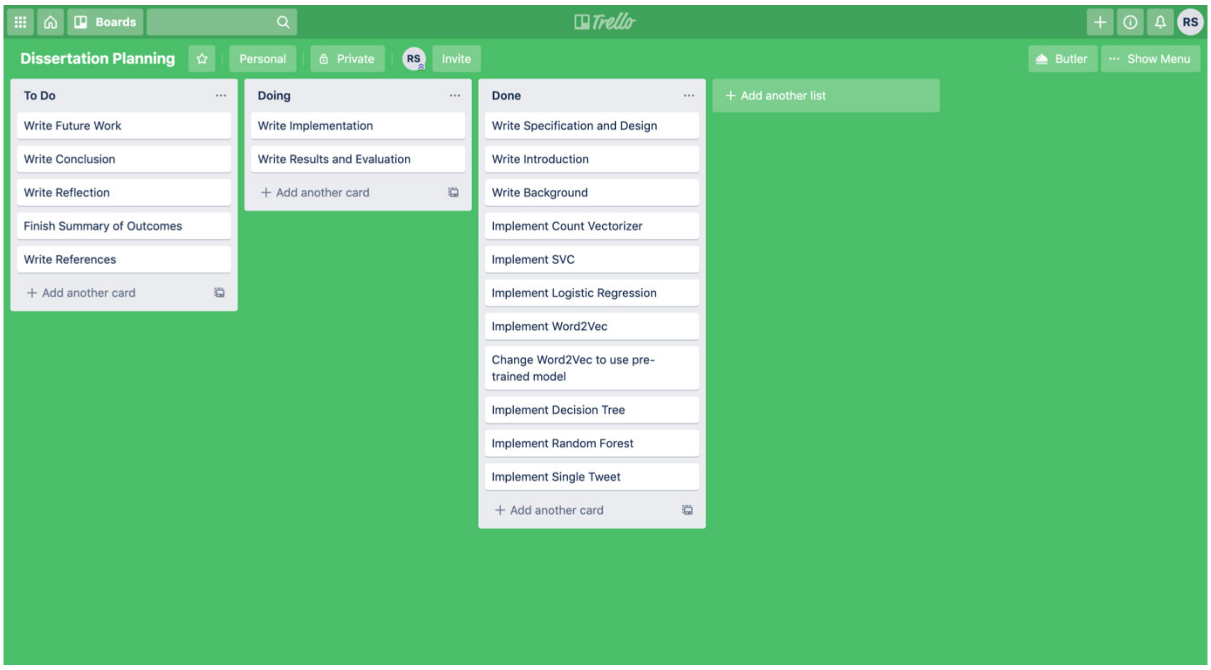
Included above is a screenshot of the command line output of the program showing the tweet that is being analysed and a prediction being made on, followed by a JSON object with relevant information contained. The JSON object has a status code of 200, showing that the request was processed successfully. It has a prediction which will always be either 0 for not offensive or 1 for offensive, and finally a label that includes a human readable output of what the prediction is. The system could easily be expanded to also include the tweet that was analysed in the JSON object returned if so desired.

This final iteration of the single tweet program is a much-improved iteration on the original version, but still has some drawbacks. For example, although the accuracy of the model used is much better than SVC with Word2Vec word embeddings as features, it is still not perfect and does have a tendency to classify offensive tweets as not offensive mistakenly, although not in the same number as other models. As a result, this program acts more of a “proof of concept” than anything, in showing how an automated offensive tweet detection system might be designed, implemented, and function. In its current state, this system could not be implemented straight away into a large social media platform or be used by law enforcement simply due to the error margin involved, and the large number of tweets that would be mistakenly classified as not offensive. However, with some adaptation, this program could be improved either through implementing better feature extraction or classifier models to improve the overall accuracy. As well as this, the program would need some kind of “wrapper” around it to provide the infrastructure necessary to use the code with a REST API.

5.6 Evaluation of Project Management

In terms of project management, I decided work in the style of the Agile Methodology.[34] Agile involves splitting up the work to be completed into small sections of work that can be completed in a shorter period of time rather than large sections of work. For example, instead of planning to simply "implement the classifier models I have chosen", I planned to implement one at a time. This allowed me to complete a piece of work and gain feedback on it before moving forward, allowing me to pivot in another direction if necessary. The use of Agile was especially useful to me given my limited experience with natural language processing – I could not estimate the complexity of a large number of the implementation tasks due to my need to learn how to use the libraries and tooling I would be using as part of this project. Therefore, being able to see clearly the tasks that needed to be completed helped me to know if I was failing to complete tasks quickly enough.

Specifically, I adopted the Kanban[35] style of Agile as my method of organising my work. I chose this style of working due to its focus on fast feedback loops and small deliverable sections of work. By having a Kanban board, I was able to split the work up into smaller sections and make these tasks visible by displaying them on a virtual board. This meant that I could immediately see where I was in the project by glancing on the Kanban board. This also had the effect of focusing my attention onto tasks that needed to be completed to allow the project to move forward. The tooling used to create and maintain the Kanban board was Trello,[36] an online tool for creating Kanban boards. Below is a screenshot showing the Kanban board that was used throughout the project.



To Do	Doing	Done
Write Future Work	Write Implementation	Write Specification and Design
Write Conclusion	Write Results and Evaluation	Write Introduction
Write Reflection	+ Add another card	Write Background
Finish Summary of Outcomes		Implement Count Vectorizer
Write References		Implement SVC
+ Add another card		Implement Logistic Regression
		Implement Word2Vec
		Change Word2Vec to use pre-trained model
		Implement Decision Tree
		Implement Random Forest
		Implement Single Tweet
		+ Add another card

Figure 31

[34] https://agilemanifesto.org/

[35] https://www.atlassian.com/agile/kanban

[36] https://trello.com/

As you can see in the screenshot above, the overall work of the project is broken down into smaller sections that allowed me to focus on what tasks needed to be completed in what order. For example, the implementation tasks needed to be completed before the implantation and evaluation sections could be written. This focus on smaller sections of work was reinforced by weekly meetings with my dissertation supervisor. By meeting once a week, I was able to present solutions to problems and get feedback on the work done, creating a fast feedback loop that allowed me to act on feedback and pivot away from work that was not going in the right direction. This flexibility meant that throughout the duration of the project I was always on track and felt that I was working well. Conversely, if I ran into problems or was in danger of not meeting a deadline for a piece of work, there were regular and frequent opportunities for me to express this to be supervisor and gain help and guidance to solve the issue. This greatly aided my work on the project and allowed me to ensure that all sections of work were completed on time and to a high standard.

Overall, this method of working was successful due to the fact that the project was completed on time and to a high standard, with all of the original aims of the project being met. The focus on fast feedback loops meant that anytime I met complications or was not heading in the right direction I was able to correct the problem without having wasted too much time on work that was not moving the project forward. One change that could be made to improve the management of the project, however, would be the use of a physical Kanban board as opposed to a virtual one. This is due to the fact that at times keeping the virtual board up to date became a secondary priority and therefore was not completely up to date. As a physical board is much more prominent than a virtual one, by using a physical board this problem would likely not occur as often.

In order to keep the code base organised and stored somewhere safely, I opted to store the code base in a Github repository. The first benefit this provided was this guaranteed that the code was backed up somewhere online so that, in the event my local copy was destroyed, I would not have lost my work and I would be able to continue working on the project with little difficulty. The second benefit this provided was it allowed me to store multiple versions of the project through different commits being stored online. This meant that in the event I needed to access an old version of the code, it was stored on Github for me. This would have been useful if I needed to restore to an older working version of the code, but it was also useful while I was writing this report as it allowed me to examine old versions of the code to refresh my memory. The final benefit that storing the code in Github provides is that going forward, if someone wants to collaborate on this repository and expand the project, then the code is online and available for collaboration.

6. Future Work

6.1 Improved Accuracy Results

The main thrust of any future work on this project should have the aim of improving the accuracy of the models used. Although the models and methods of feature extraction used in this project so far did have some success at classifying offensive tweets, I do not believe that any are accurate enough for implementation in the real world. As such, the focus of any further work needs to be finding ways to improve the accuracy of results provided by the models used. There are a number of ways that this could be accomplished, for example different methods of feature extraction could be implemented, or new classifier models could be implemented to try and improve the accuracy of results. Alternatively, another way of achieving better accuracy could be through exploring neural networks.

In terms of different forms of feature extraction that could be implemented, one option to implement in the future could be a Hashing Vectorizer.[37] This vectorizer has a number of advantages over others that have been used as part of this project but the main advantage is the extremely low memory use of this vectorizer. This would be particularly useful for the single tweet program as this would speed up the loading time of the pre-trained model used to make predictions. It is possible that, when combined with parameters making use of "n_grams" and tokenizing by characters instead of words, that this vectorizer could produce an improvement in the accuracy of results. If this was combined with faster loading times when using Joblib, this vectorizer could provide a significant improvement to the overall functionality of the single tweet program and make it faster at returning results with little loss in accuracy of those results.

As well as implementing completely new forms of feature extraction, there is an argument to be made for improving the methods of feature extraction currently used in the project. For example, an idea for future work could be improvement of the data that is used to select features from in the current iteration of the project. Currently, features for training the classifier models are either selected from the corpus of training data provided to me as part of the project, or from the pre-trained Word2Vec model that is trained on hundreds of thousands of tweets collected. It could be argued that the Count Vectorizer, Word2Vec word embeddings, and TF/IDF vectorizer could produce better features to train the classifier models on if they had access to larger and more carefully constructed data to select features from. This work could involve anything from more extensive normalising of the data to using completely new data sets to select features from.

There is a large scope for future work on this project involving the implementation of new classifier models. The four models that were implemented as part of this project were all chosen for specific reasons, but there are a large number of classifier models that have been unexplored as a result of this. Although I will examine some of the possible classifier models that could be implemented in the future, this is by no means an exhaustive list. One

[37] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.HashingVectorizer.html

potential classifier model that could be implemented is the Naïve Bayes algorithm.[38] There are a number of variations of this algorithm that could be explored to discover which provided the best results, including Gaussian Naïve Bayes and Complement Naïve Bayes. Another potential classifier model that could be implemented is the AdaBoost classifier.[39] This classifier works by first fitting a classifier over the data set and then fitting additional classifiers over that, while simultaneously adjusting the weights of incorrectly classified cases. This has the effect of focussing subsequent classifiers on the more difficult cases to classify. Either of these classifier models could provide a boost to results or, alternatively, other classifier models would also potentially provide an increase in the accuracy of results. The key takeaway is that, if I had had more time on the project, I would have explored a much wider range of classifier models. This is not to say that I believe I did not cover enough classifier models as I selected the models that I felt would provide the best results, but it does mean that in the pursuit of better accuracy new classifier models is a valid endeavour for future work.

It has already been discussed that the most successful teams participating in solving this problem online made use neural networks as their preferred method of making predictions.[40] Therefore, the next logical step for this project would be to incorporate neural networks as the method of classifying tweets as offensive/not offensive. If I had more time to complete this project, the starting point for expanding this project to use neural networks would be TensorFlow.[41] I would choose this starting point due to the fact that TensorFlow is a machine learning neural network library for Python, meaning that you could build on top of the existing code base and would not have to start the project again in another programming language. I believe that by using neural networks to make predictions on the data set, you would likely see a significant improvement in the accuracy of results returned, due to the strong performance of teams utilising neural networks when interacting with the same data sets used as part of this project.

In conclusion, although all of the original aims of the project have been met there is a large scope to continue working on this project in terms of just improving the accuracy of results that are returned from the predictions made by classifier models. The benefit of improving the accuracy of results returned is clear in ensuring that any system implemented that makes use of the outcomes of this project, for example a single tweet analyser, would be more accurate and make less mistakes. This is key if a large social media platform or a law enforcement agency were attempting to implement an automated abusive language detection system as a large number of false positive classifications would undermine the usefulness of the system.

[38] https://scikit-learn.org/stable/modules/naive_bayes.html

[39] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

[40] Zampieri, M., Malmasi, S., Nakov, P., Rosenthal, S., Farra, N. and Kumar, R., 2019. Semeval-2019 task 6: Identifying and categorizing offensive language in social media (offenseval). *arXiv preprint arXiv:1903.08983*.

[41] https://www.tensorflow.org/

6.2 Implementation as a REST API

Alongside improving the accuracy of the results returned by the classifier models used as part of this project, another idea for future work on this project would be to take the single tweet program proof of concept and fully develop it as an automated abusive language detection system. I believe the best way to expand on the single tweet program would be to adapt it for use as a REST API. The single tweet proof of concept already returns a JSON object but the code would need to be modified to be parsed the tweet that it is classifying.

If I had more time and was going to begin this piece of work, the way that I would go about implementing this change would be to design the program for a fully serverless environment. This could take place on any of the major cloud providers but I have the most experience with AWS therefore this would be the platform that I would use. The solution I would implement would involve hosting the code in AWS Lambda and connecting AWS API Gateway to it to allow for the code to function as a REST API properly. This solution would provide a number of benefits. For example, by implementing this solution you would essentially guarantee that the API would always be available and functioning. Another benefit is you would not have to worry about hosting and maintaining the API yourself, which would cut down on the up-front cost of purchasing something to host the API on. The final benefit of this implementation would be the ability to set up a deployment pipeline for changes made to the code. By hosting the code on AWS, it would make it easy to deploy code pushed to the master branch in Github into AWS automatically, as this is a widely done practice that has been implemented in the industry.

In order to implement the solution suggested above, the code base would need to undergo some adaptation. For example, currently the program loads the tweet from a text file from local memory. This would have to be changed to take a tweet from a request to the REST API. Depending on how you wanted to use the system, this could be triggered by a manual API request or by a webhook that is pushed to the API. As well as this, there would need to be some solution that handles the loading of the pre-trained model used for generating the prediction for the tweet being analysed as in the proof of concept I designed, this model is loaded from local memory as well. Perhaps the most important change that would have to be implemented, however, is the addition of robust security to the program. There is no security implemented in the current version of the program due to the fact that the program is run locally and not hosted anywhere online that could be accessed by anyone. If this program were to be adapted into a REST API, you would need to include robust security to ensure that only verified users could make use of the API, otherwise you would potentially leave the API open to misuse. There are a number of ways that you could go about implementing this security. One option would be using JSON Web Tokens (JWT) as a method of allowing a user to remain authenticated for multiple uses of the API during a certain time frame. Another option would be to use an API key that could be given to verified users and would allow them to authenticate with the API Gateway instance.

By implementing the solution described above in combination with the outcomes described as part of this project, you would have a reasonably accurate automated abusive language detection system. This system could be utilised as a third-party system for abusive language detection that could be utilised to analyse tweets for offensive content. Equally, the system could be further adapted by websites with large amounts of user generated content to automatically flag abusive language, particularly towards women and immigrants. Equally, law enforcement could use this system as a method of flagging abusive language. It must be said now, however, that due to the large number of false positive classifications of not offensive tweets, this system could not exist on its own as the only method of abusive language detection. There will still need to be some level of human intervention before any final judgement can be made. For example, social media platforms could still rely on users reporting offensive content to catch cases where content had been mistakenly labelled as not offensive. There would also need to be some level of human intervention for examining tweets that have been potentially mistakenly been labelled as offensive, even though the results suggest that false positive predictions for offensive tweets are very rare.

7. Conclusions

7.1 Summary of Aims and Results

The main aim of this project, as stated at beginning of this report, was “to design an effective system for detection of abusive language towards immigrants and women”. I would argue that this aim has been met, as I have demonstrated throughout this report that I have designed a system that can effectively classify abusive language towards immigrants and women. Although this was the main aim of the project, this aim can be split down into a number of smaller sub-aims for the project. For example, a sub-aim that fed into the main aim of the project was the need to implement a number of classifier models in order to ascertain which produced the most accurate results. To complete both of these aims, a strong knowledge of natural language process was required, and I had to learn this as the duration of the project went on. All of these aims were met throughout the course of the project.

As has been discussed in the body of this report, the work that has been completed as part of this project is not perfect, and as such the systems that were designed and explored as part of this project were also not perfect. There were varying degrees of success with some models suffering from very low accuracy of predictions compared to other models that were largely accurate in their predictions of tweets. The most accurate classifier model used was the SVC classifier, and the least accurate classifier model used was the Decision Tree classifier. Although the classifier model used did impact results, the model selected did not have as much of an effect as the method of feature extraction that was used. Classifier models that were combined with the Count Vectorizer as its method of feature extraction always performed the worst, often providing results that were wrong over half of the time. The most successful combination of classifier model and method of feature extraction was the SVC classifier combined with TF/IDF feature extraction. This provided results that suggested that this combination would almost always be correct if it had predicted a tweet as offensive but was significantly worse when predicting tweets as not offensive, with significantly more false positives for these classifications. This exploration of classifier models and feature extraction methods led to a number of designs of systems for the detection of abusive language towards women and immigrants that could all be considered effective, to varying degrees.

Following on from the classifier models and methods of feature extraction that were explored as part of this project, a further aim was added of developing a proof of concept program to show how the outcomes of the experiments run as part of this project would be implemented into something more applicable to the real world. The result of this aim was the development of a program that takes in a single tweet and returns a prediction as to whether that tweet is offensive or not. This proof of concept program went through a number of versions before its final iteration was completed. Initially the program was using a sub-par classifier model and feature extraction combination, which did reduce the usefulness of the program to a potential interested party. A solution to an issue with program run time was found, however, and in the final iteration of the program, the aim of having an effective proof of concept program as met.

8. Reflections

8.1 Key Learnings

Due to my lack of knowledge regarding natural language processing prior to beginning this project, all of the work done as part of this project was essentially new learnings for me. What I have gained throughout this project is a deep understanding of machine learning and natural language processing. There were a few key learnings, however, that I feel are necessary to point out here.

Firstly, I learned that although classifier model selection is important, the selection of method feature extraction has a far greater impact on the accuracy of results. This was shown quite conclusively through my results, with different classifiers using the same feature extraction method often having very similar results. This was a surprise to me as prior to performing these experiments and analysing the results, I had expected the classifier model selection to play a far greater role in determining the accuracy of results produced. This revelation led to me putting a greater focus on researching potential different forms of feature extraction, and as such is a key learning for anyone working on natural language processing.

Another key learning that I gained by working on this project was the need to save and load pre-trained classifier models. As I worked on the single tweet program, I was initially limited from using the most accurate classifier model and feature extraction combination due to it having a 20-minute program run time. In order to solve this issue, I had to research and find a method of saving a pre-trained classifier model as this was what was taking up the bulk of the program run time. By solving this problem, I was able to vastly improve the accuracy of predictions made by the single tweet program and therefore make the proof of concept much more useful to an interested party. This was a key learning for anyone working with natural language processing and machine learning.

9. References

1. Banks, J., 2010. Regulating hate speech online. *International Review of Law, Computers & Technology*, *24*(3), pp.233-239.

2. Kawate, S. and Patil, K., 2017. Analysis of foul language usage in social media text conversation. *International Journal of Social Media and Interactive Learning Environments*, *5*(3), pp.227-251.

3. Basile, V., Bosco, C., Fersini, E., Nozza, D., Patti, V., Pardo, F.M.R., Rosso, P. and Sanguinetti, M., 2019, June. Semeval-2019 task 5: Multilingual detection of hate speech against immigrants and women in twitter. In *Proceedings of the 13th International Workshop on Semantic Evaluation* (pp. 54-63).

4. Bosco, C., Patti, V., Bogetti, M., Conoscenti, M., Ruffo, G.F., Schifanella, R. and Stranisci, M., 2017. Tools and resources for detecting hate and prejudice against immigrants in social media. In *SYMPOSIUM III. SOCIAL INTERACTIONS IN COMPLEX INTELLIGENT SYSTEMS (SICIS) at AISB 2017* (pp. 79-84). AISB.

5. Cresti, M., Martino, V. and Rosola, M., Kate Manne. Down Girl. The Logic of Misogyny, Oxford University Press, 2017, pp. XXIV, 338. *APhEx*.

6. Zampieri, M., Malmasi, S., Nakov, P., Rosenthal, S., Farra, N. and Kumar, R., 2019. Semeval-2019 task 6: Identifying and categorizing offensive language in social media (offenseval). *arXiv preprint arXiv:1903.08983*.

7. Camacho Collados, J., Doval, Y., Martínez-Cámara, E., Espinosa-Anke, L., Barbieri, F. and Schockaert, S., 2020. Learning cross-lingual word embeddings from Twitter via distant supervision. *ICWSM*.