Abusive Language Detection Against Immigrants and Women

[Introduction 3](#_Toc38622022)

[Project Aims 3](#_Toc38622023)

[Intended Audience 3](#_Toc38622024)

[Project Scope 3](#_Toc38622025)

[Approach 3](#_Toc38622026)

[Summary of Outcomes 4](#_Toc38622027)

[Background 5](#_Toc38622028)

[Project Context 5](#_Toc38622029)

[Identified Problem 5](#_Toc38622030)

[Likely Stakeholders 6](#_Toc38622031)

[Theory Associated with Problem Area 6](#_Toc38622032)

[Constraints on Approach 8](#_Toc38622033)

[Existing Solutions 8](#_Toc38622034)

[Methods and Tools 9](#_Toc38622035)

[Specification and Design 10](#_Toc38622036)

[Approach to Solving the Problem 10](#_Toc38622037)

[Implementation 12](#_Toc38622038)

[Data Handling 12](#_Toc38622039)

[Feature Extraction 13](#_Toc38622040)

[Supplementary Materials 16](#_Toc38622041)

Introduction

# 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]](#footnote-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 other people 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 compare this accuracy to other participant of the competition. This would be done through a thorough understanding of natural language processing, and experimentation with different forms of classifiers.

# 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]](#footnote-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. Therefore, I would argue that the techniques and models explored in this project would be beneficial to anyone looking at implementing 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.

# 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. Therefore, it was hard to estimate the amount of time it would take me to complete pieces of work.

# 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 focussed 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), and then split this one set into multiple splits using a method known as *K-Fold Cross Validation*. I was able to get access to the third data set during my project, and once I had this, 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

[CHANGE THIS TO DIAGRAM?]

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.

# Summary of Outcomes

In terms of the important outcomes from the project, there were a number of interesting outcomes, as follow:

* Word embeddings based on Word2Vec were the most effective way of determining features for the models to use.
* SVC was the most effective model at classifying tweets from the corpus accurately.
* Your classifiers use word embeddings, but it’s not that word embeddings are for determining features for classifiers to use, they are general-purpose, semantically rich numerical representations of words.

[REWRITE THIS]

Background

# 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]](#footnote-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 focussed 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]](#footnote-4). In particular, immigrants have faced a rise in hate speech due to the refugee crisis[[5]](#footnote-5), and women have historically always suffered from hate speech[[6]](#footnote-6). The main project focus of detection of abusive language splits into a lot of sub problems, but the main 3 are feature extraction, model selection, and experiment setup. It is through working through these 3 problems that I was able to come up with varying different sets of results with varying accuracy.

Another aim is to produce a piece of software that 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.

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

# 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 nicer to use, as end users will stop using the site if they are receiving abuse. Equally, some of the abusive language that could be posted online could constitute hate speech, which is a crime in the United Kingdom[[7]](#footnote-7) as well as many other nations, and therefore it would be beneficial to law enforcement agencies if they could automatically detect the 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.

# 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]](#footnote-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]](#footnote-9). This model is “a meta estimator that fits a number of decision tree classifiers on various sub-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]](#footnote-10). This model involves using a Logistic Function[[11]](#footnote-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]](#footnote-12). This model works by plotting a line through the corpus matrix in order to makes 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 tokenization 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, TF/IDF extraction, and a Word2Vec 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 Gensim 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.

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[[13]](#footnote-13) 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 get 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 for me, and I then combined the results to create an average.

# Constraints on Approach

As stated in my introduction, I was originally constrained from approaching the project in what is considered the best practice method for developing a models 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 did have 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.

# 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[[14]](#footnote-14). 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[[15]](#footnote-15). 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 emoji’s to text to be able to analyse them effectively.

# Methods and Tools

My project was developed using Python 3.7.6[[16]](#footnote-16), and in particular I made use of 3 specific libraries to complete the project: Sci-Kit Learn, Gensim, and Numpy. In terms of what makes up 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.

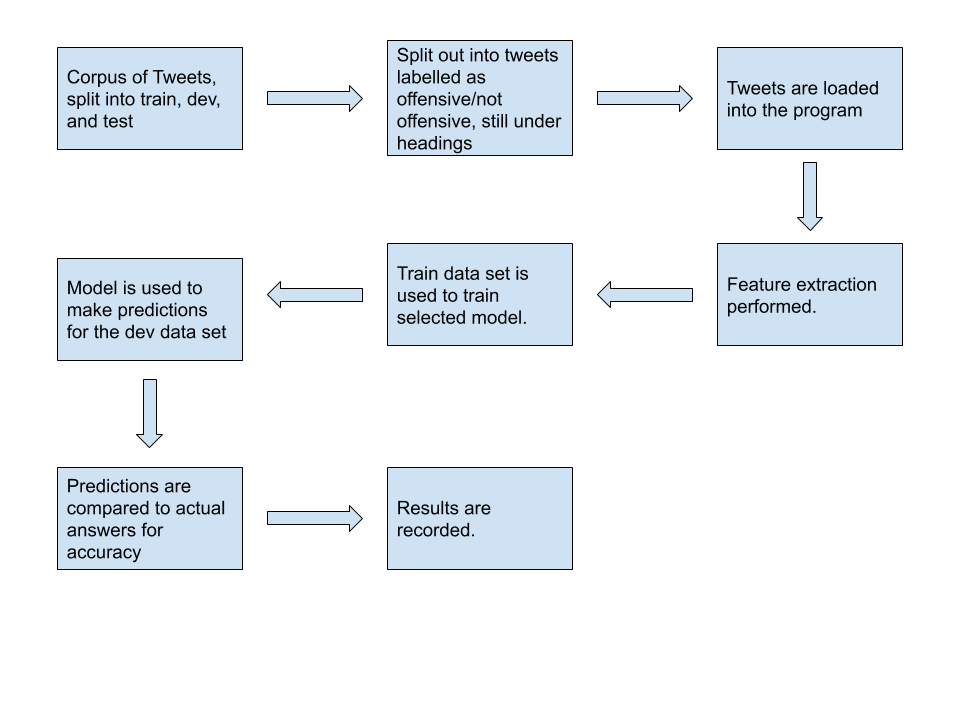
Sci-Kit Learn[[17]](#footnote-17) is a library that I used to give me access to a wide number of different classifier models that I could train my corpus on, and use to generate predictions for if a tweet is offensive or not. As well as this, at once stage I also used this library to extract features from the corpus as a further method of training the classifier model.

Gensim[[18]](#footnote-18) 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, 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 see what words are commonly used along with what could typically be considered as abusive language.

Numpy[[19]](#footnote-19) 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.

Specification and Design

# 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.

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, a lot 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 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. However, the diagram does 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 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.

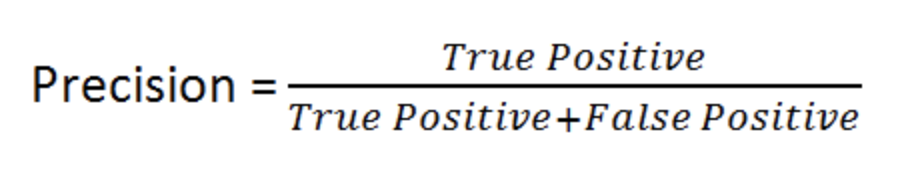
The way in which I recorded my results was to include a function that first noted down the parameters of the experiment to the results file, and then copied down the classification report containing metrics such as precision, recall, and f-1 score of the experiment. All of this information was recorded in a CSV, for two main reasons. First, because Python has a built in module for handling CSV files, and second 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 go over and analyse, allowing me to accurately present a summary of outcomes as to what the most effective methods for detection of abusive language are.

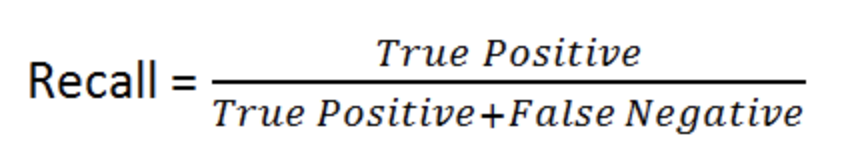
Another key step that is abstracted in program flow chart under the “tweets are loaded into the program” step is data pre-processing. This step involves normalising the tweets so as to improve the overall accuracy of the results produced by the experiment. I performed varying levels of data pre-processing throughout the project as I learned more about natural language processing. In the final state of the project, the most effect 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 already previously touched on the variations in feature extraction and model selection, and I will go into depth about just how I implemented these varying methods that make up the core of the experimental setup.

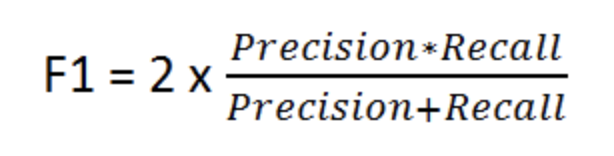
# Metrics Used

Precision, Recall, and F-1 score are the key metrics by which the accuracy of predictions made by the system are made. And as such, 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[[20]](#footnote-20). 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.

Recall is defined as the number of True Positives divided by the sum of True Positives and False Negatives[[21]](#footnote-21). 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.

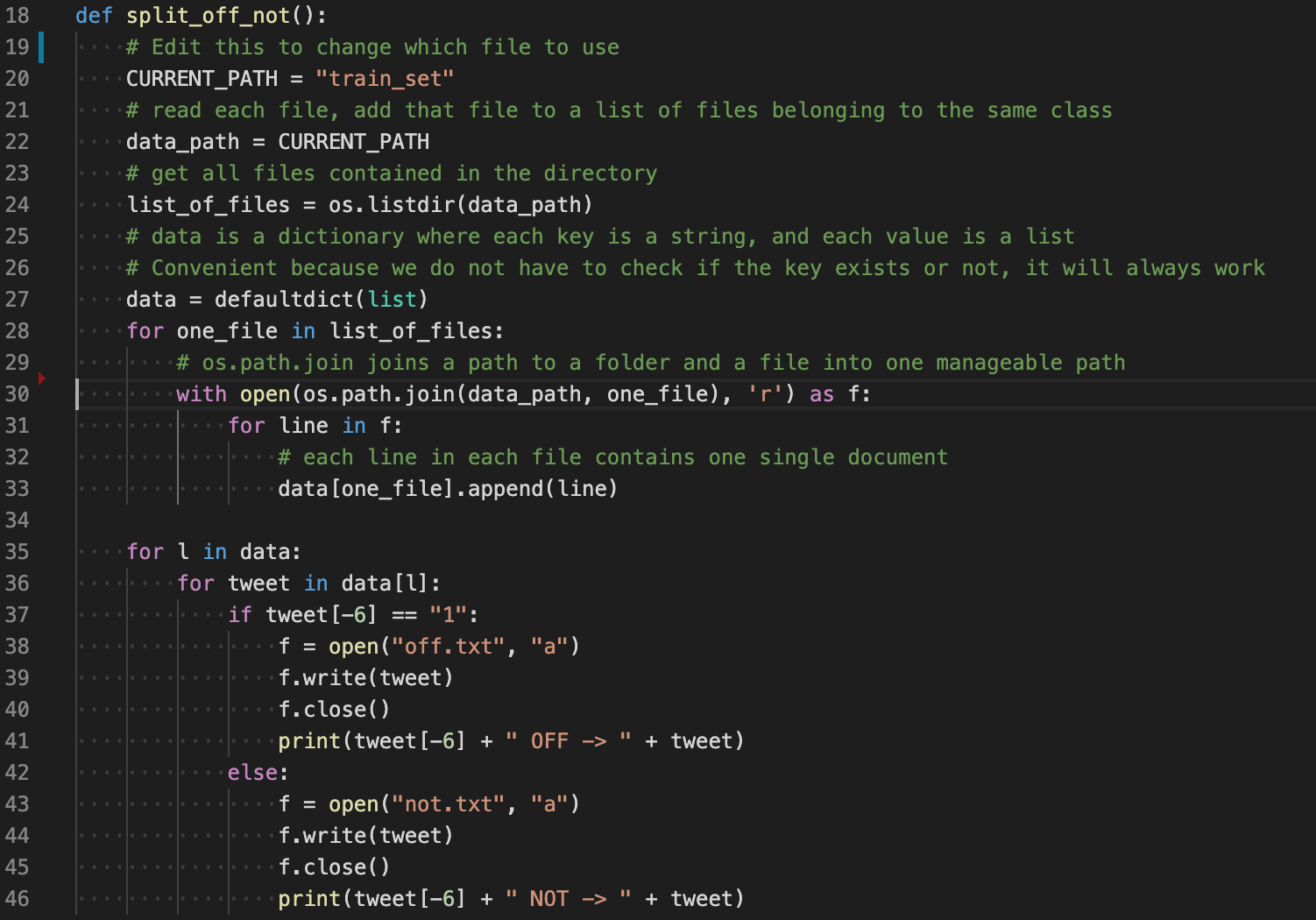
F1-Score is the final, and perhaps most important metric that will be used in this paper to determine the accuracy of predictions made by the models used. F-1 Score is defined as Precision multiplied by Recall, divided by the sum of Precision and Recall, multiplied by 2[[22]](#footnote-22). The F-1 Score is necessary as it strikes a balance between Precision and Recall, thus showing a good overall measure of the way a model is performing.



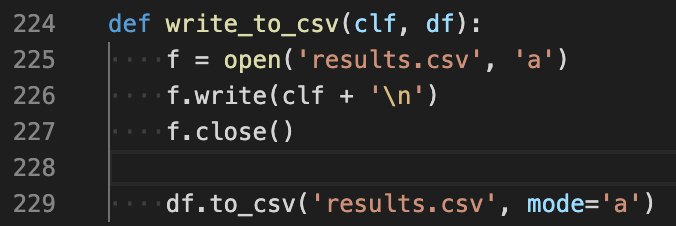
Implementation

# Data Handling

The first part of the project that I worked on was the handling of the data sets, as the libraries that I made use of 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[[23]](#footnote-23) 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 as such did not require another dependency to get the program running.

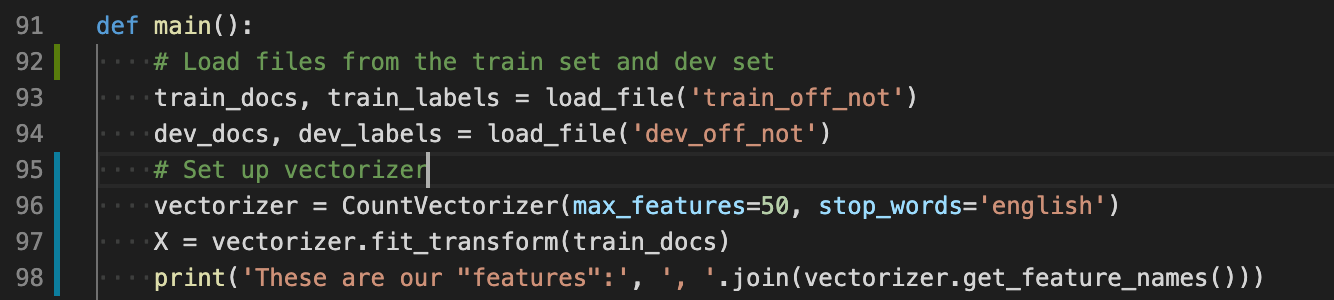
 Once the text files have been loaded, the next problem that I had to solve was the splitting of the training data, in this case the train data set, into separate offensive and not offensive text files, based on their labels within the original train set text file. The reason that this had to be done is 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.

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.

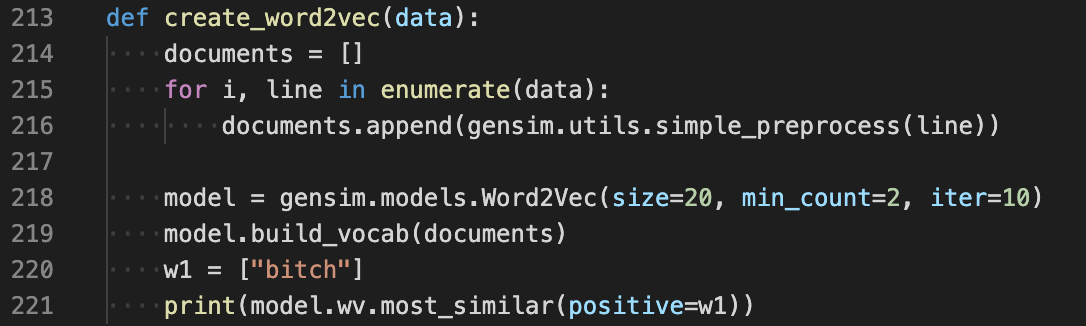
The final form of data handling that needed to be implemented was ensuring that all results were automatically recorded for later analysis. It was important to record this information in some form of permanent storage, as otherwise results would be lost any time the terminal was closed. Another requirement 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.

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.

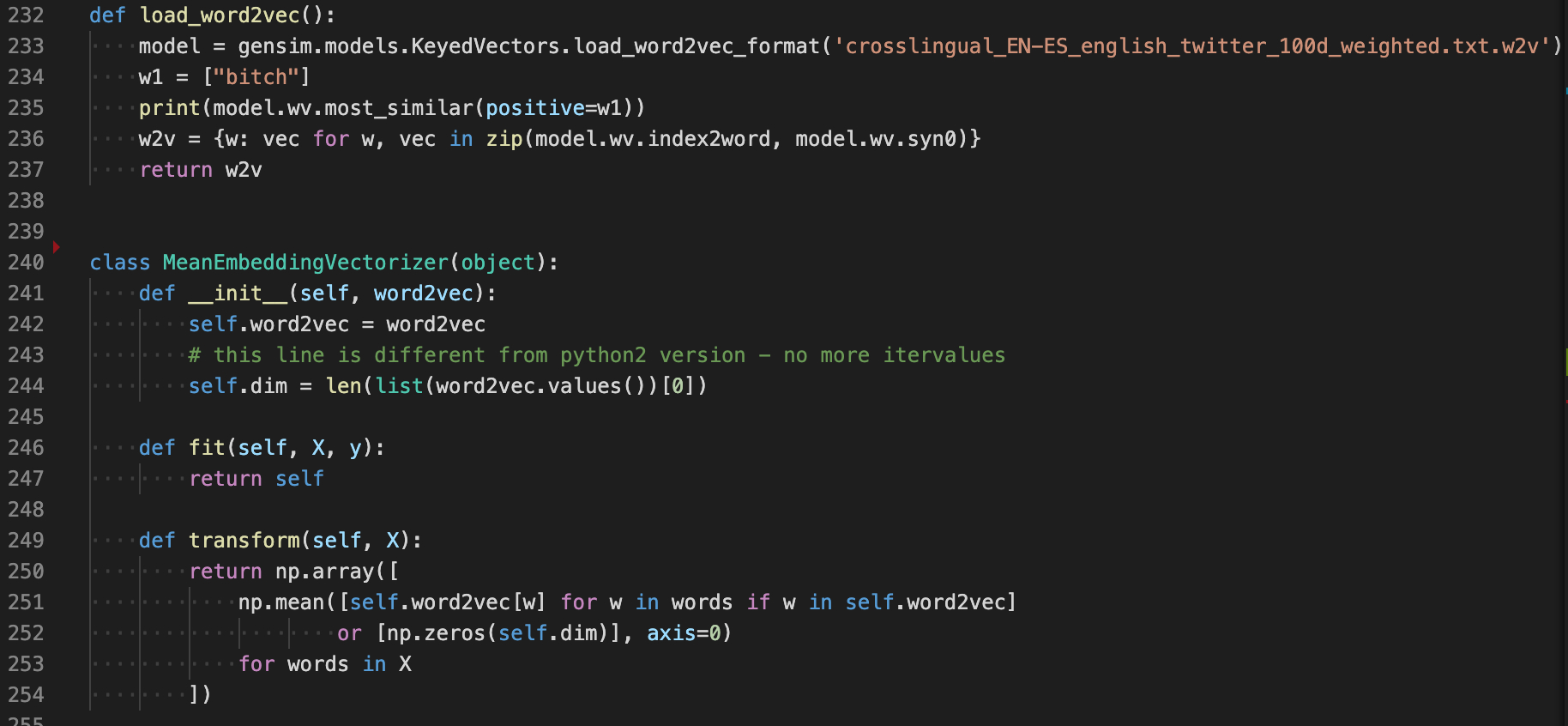
# Feature Extraction

 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 and use those as the features to train the model you have selected with. In my case, I limited the vectorizer to 50 words, but I also made use of the “stop\_words=English” parameter to skip over words common linking words, for example “and, or, if”. I did experiment 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.

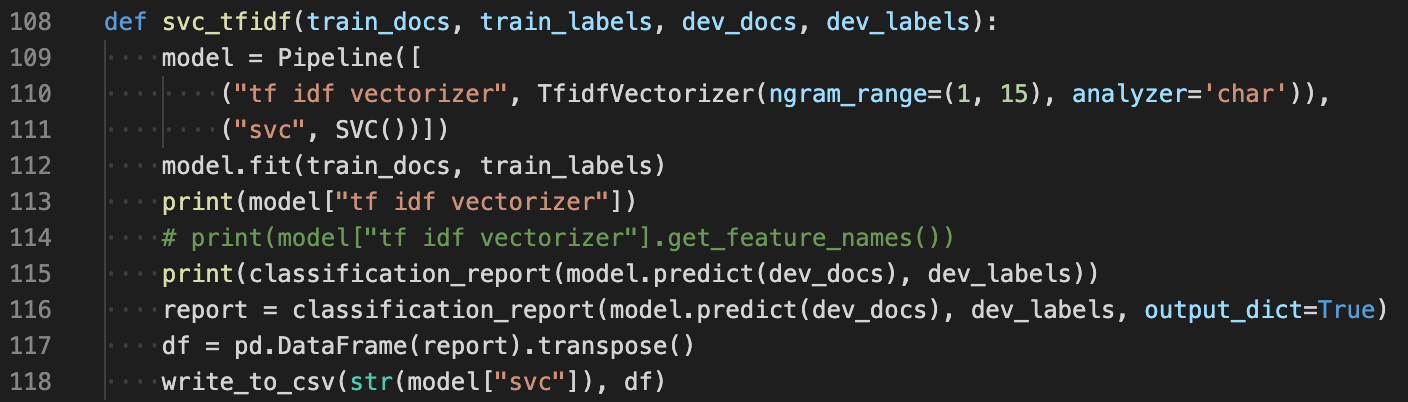
As you can see in the screenshot of the code, I load in the train data set and the dev data set. Once this is done, initialise the vectorizer with the parameters I want, and then pass the train set corpus to the vectorizer in order to get our list of features, which is shown on line 97. For verbosity, the list of features that the vectorizer has generated are printed to the console. Although the Count Vectorizer was the simplest form of feature extraction that I implemented, it was also the worst performing in terms of accuracy of predictions.

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[[24]](#footnote-24) works by creating word embeddings for each word in the corpus, and then looking at what words are “near” that word, to determine links between words. The use of Word2Vec in this project occurred in two stages. Initially a Word2Vec model on trained on the provided corpus of tweets and the subsequent word embeddings created from this model was used to train classifier models.

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 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. Due to the fact 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. [Find out where Luis got this model from and cite]. 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 “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[[25]](#footnote-25), 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 embedding index values and format them into a NumPy matrix, which is returned as 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.

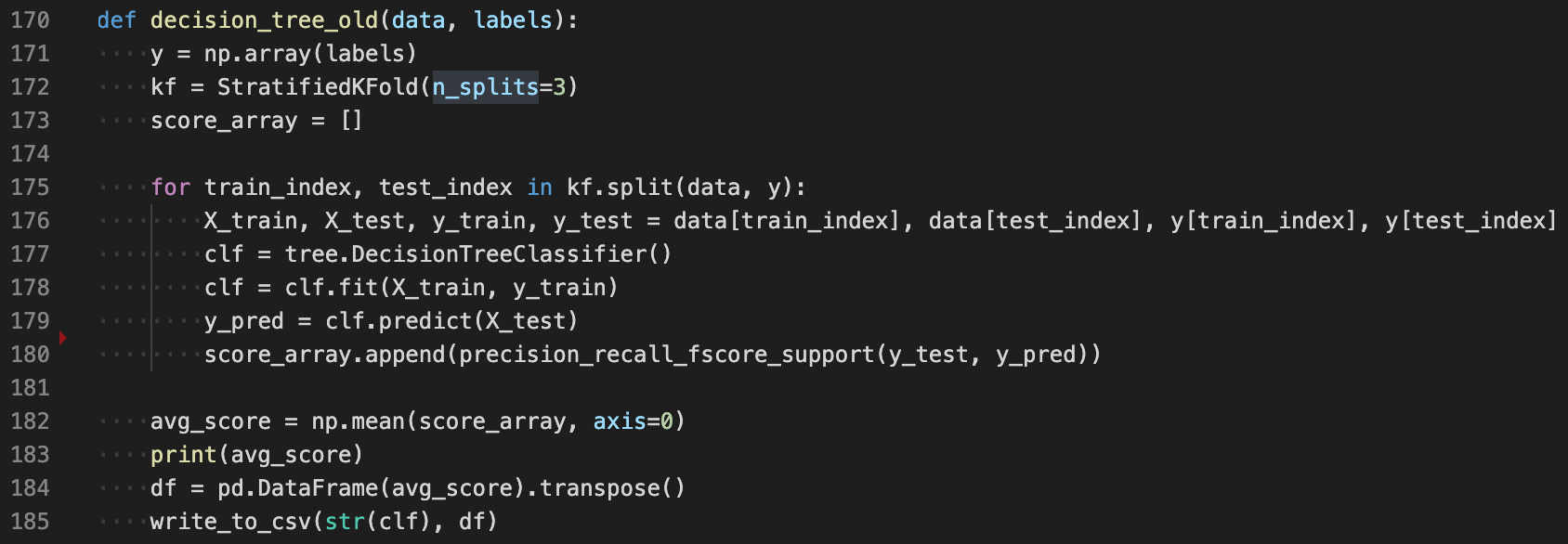
The final form of feature extraction that was implemented in this project was the use of “Term frequency–inverse document frequency” or TF/IDF. This form of feature extraction was implemented as a result of continued research into alternative means of feature extraction, and the intended effect when 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.

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.

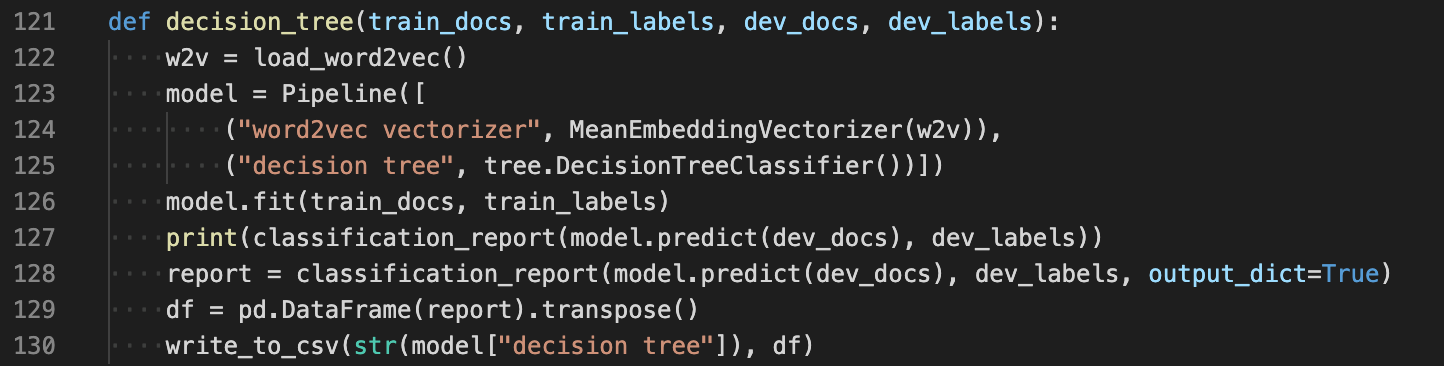
On the first use of the TF/IDF vectorizer, all of the parameters were set to default to see what impact this would have on results. Initially this did not improve results, which came as a surprise to the author as the expectation was an improvement. Upon further research into how this vectorizer worked, it was determined that the vectorizer was attempting to tokenize only 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.

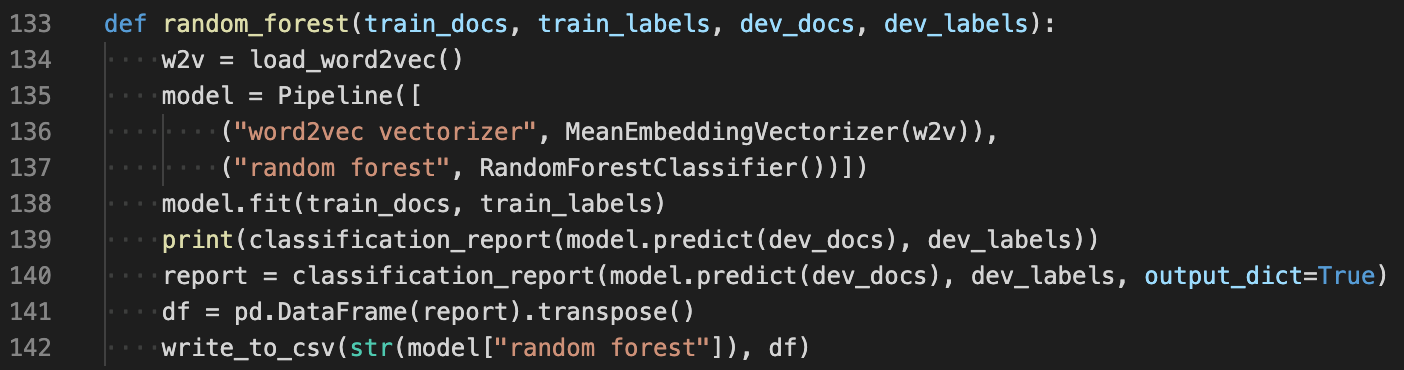
# Classifier Model Selection

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. The first that will be discussed was the first model that was implemented, the Decision Tree. It must be said now 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 will be highlighted now.

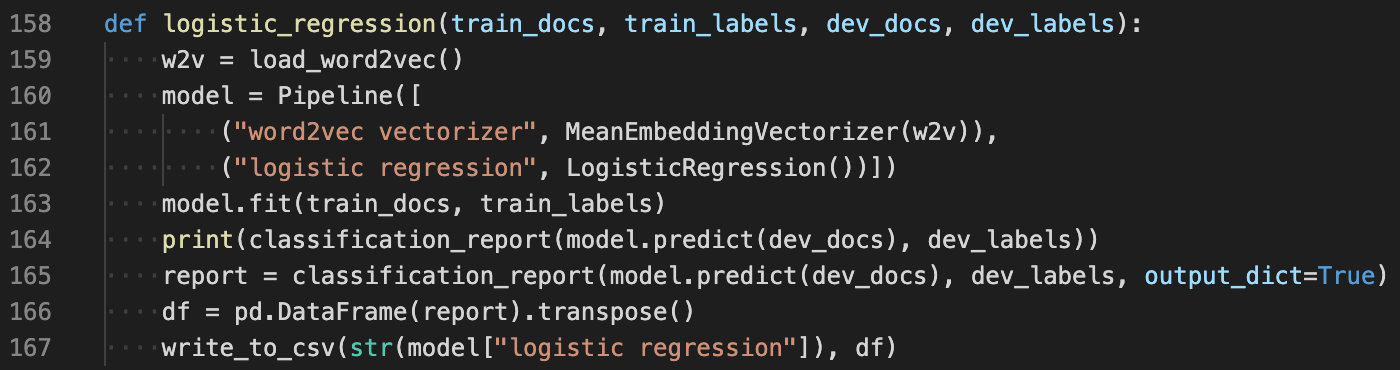
The code excerpt above shows the first complete implementation of the Decision Tree classifier. This initial version of the function made use of k-fold cross validation and as such is structured differently to the later versions of the classifier models that were implemented. It is also important to 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.

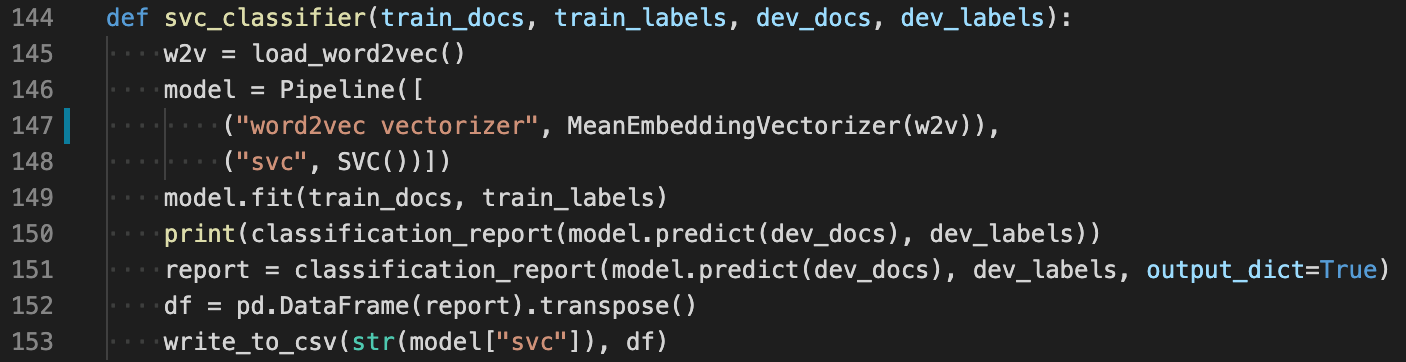
The excerpt above is the final iteration of the Decision Tree classifier. As shown previously in this report, this implementation of the classifier makes use of the Pipeline feature from the Sci-kit Learn library. As well as this, this function makes use of the Word2Vec vectorizer. The function is parsed 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.

 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.

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, with 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.

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 and as such 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. However, it must be said once again that due to the use of the Pipeline feature, the Logistic Regression function is broadly similar to the previous two classifier functions.

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.

 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 and as such was never implemented using k-fold cross validation. However, SVC was implemented in two different formats: one format making use of the 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, and as such I will focus on the implementation of SVC making use of the Word2Vec form of feature extraction.

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.

# Single Tweet Program

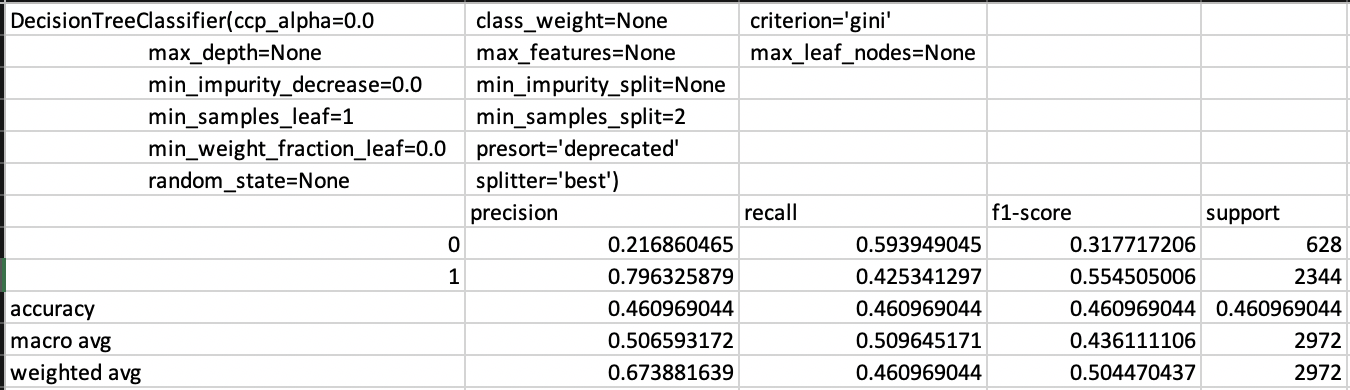
Once the experiments had been run and it became clear what were the best methods to use in order to make the most accurate predictions regarding whether a tweet was offensive or not, a decision was made by the author to implement a program that took a single tweet as an input and returned a prediction for that tweet. The reasons for this were that making predictions for large data sets often has little use in terms of real-world application, what is much more likely is a single tweet needs 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 though, 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.

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.

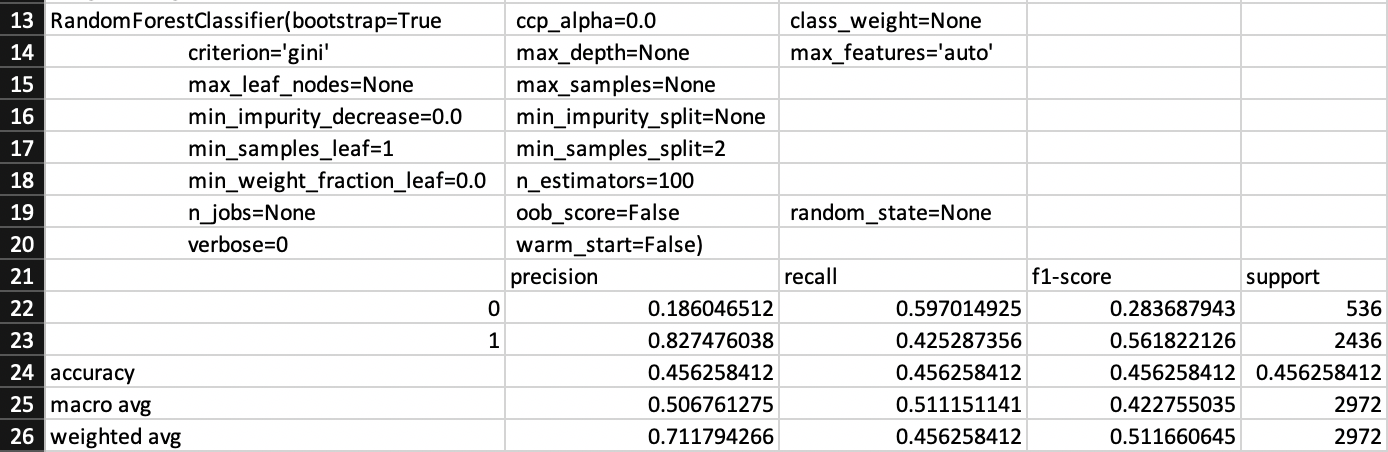
Results and Evaluation

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, 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”.

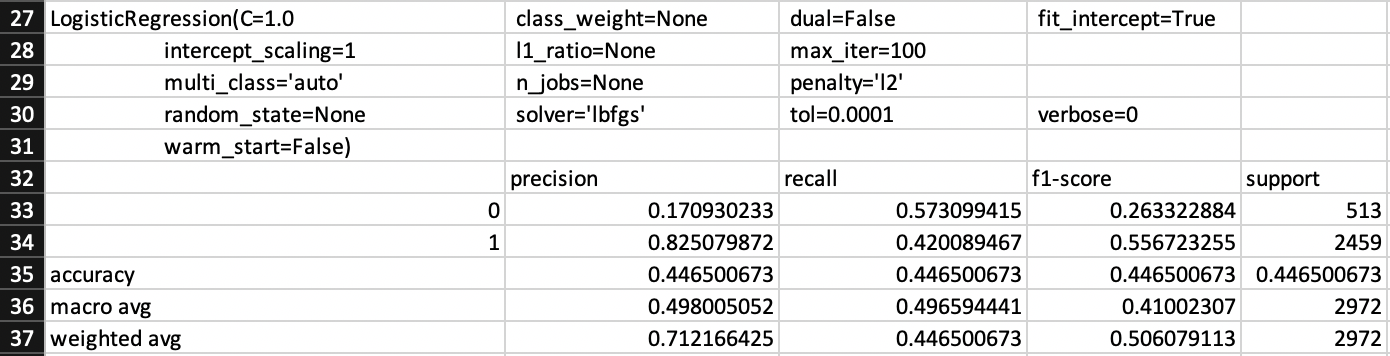
# Count Vectorizer

 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.

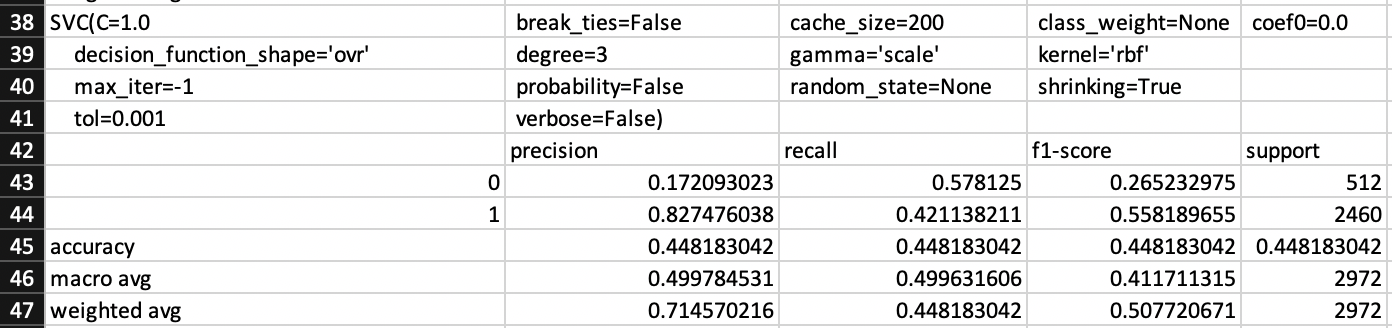
As can be seen in the results above, the Decision Tree combined with a Count Vectorizer as the form of feature extraction does not perform well, with an overall F-1 Score of 0.46. This shows that over half of the predictions made by this model were incorrect. If we examine the results further, we can 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. What this means is the Decision Tree classifier combined with the Count Vectorizer classifies far too many tweets as offensive when they are actually not, resulting in a large number of False Positive results. This assessment of the model performance is confirmed by examining the recall scores of the model, which shows a low Recall score of 0.43 when classifying offensive tweets, showing that the model is getting a large number of offensive predictions incorrect. These defects in the model combine to show the relatively poor F-1 Score that has already been discussed. 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 in classification of offensive tweets is only down to the fact this model over classifies tweets as offensive and as such has a large number of false positives.

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.

As can be seen, the overall F-1 Score is exactly the same as the Decision Tree with 0.46. The Random Forest classifier also performed exactly the same in terms of Recall score for both not offensive and offensive tweets, again showing that this model is getting a large number of predictions of offensive tweets incorrect. This model ended up having a better Precision score than the Decision Tree for classifying offensive tweets with 0.83, but a worse Precision score when classifying not offensive tweets with 0.19. What this suggests is that the Random Forest suffers from the same flaw as the Decision Tree, which is having a large number of false positives when classifying offensive tweets. 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 0.28, 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 very accurate.

 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.

As can be seen, 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 so far. Equally, Logistic Regression has the worst overall F-1 Score of any classifier so far when combined with a Count Vectorizer.

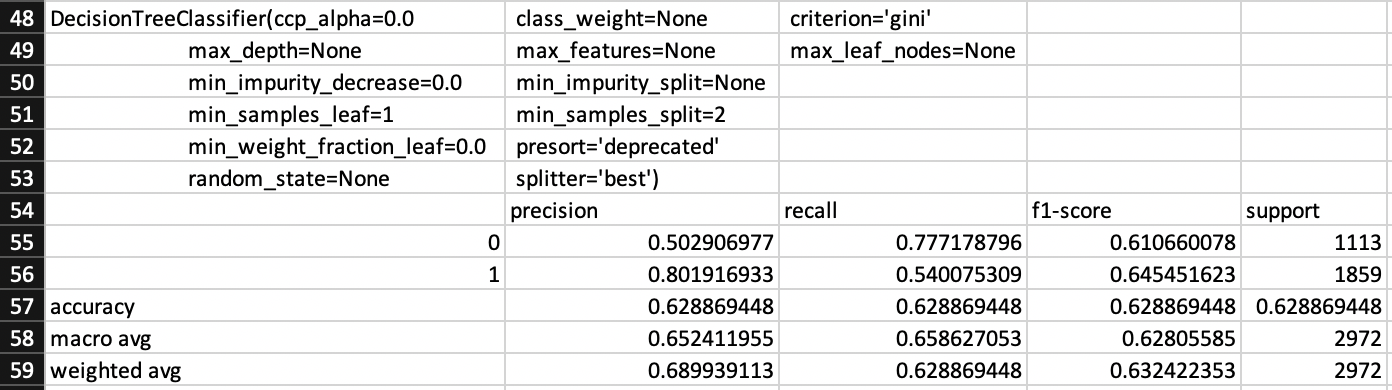
 The final classifier to be examined when combined with a Count Vectorizer is the SVC classifier. The results of this experiment can be seen below.

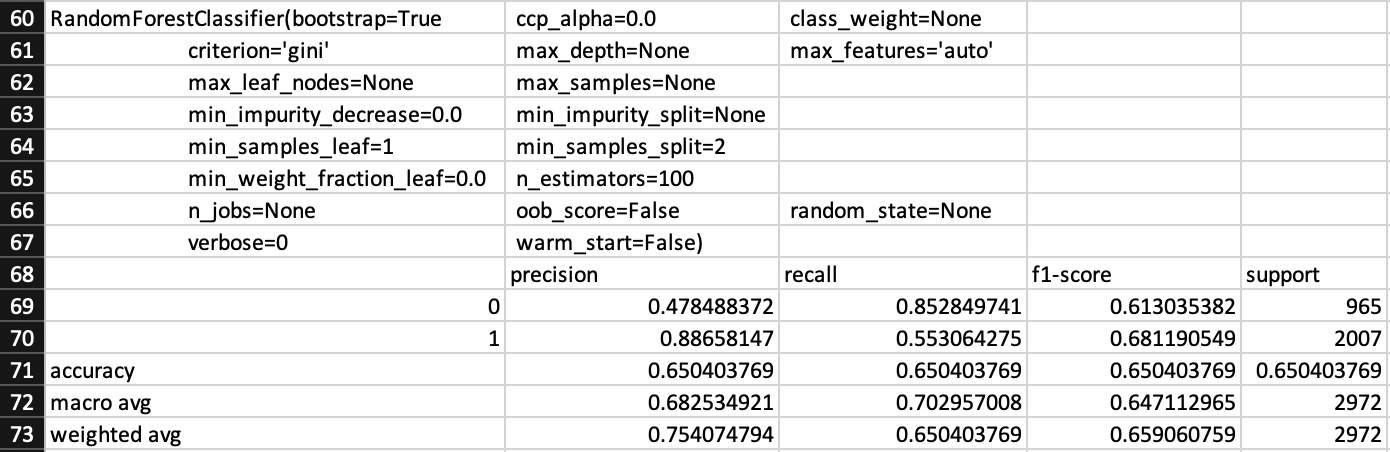
These results are essentially identical to the results for Logistic Regression, and as such little extra analysis can be done. All that can be said for this experiment is that the same conclusions drawn for Logistic Regression apply for SVC as well.

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. However, there was a high degree of precision when it came to Precision scores for offensive tweets. As I have mentioned above though, this is due to the fact that these models are classifying offensive tweets in large amounts and generating large numbers of false positives. 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 down 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.

# Word2Vec for Feature Extraction

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 it must be mentioned 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.

Immediately it can be seen that there has been a huge improvement in results across Precision, Recall, and F-1 Score for both not offensive and offensive tweets due to the use of the Word2Vec word embeddings as features. We can see a significant improvement in the both the Precision and Recall score for not offensive tweets, suggesting that this model is classifying more tweets as not offensive and getting more of them right. However, the Precision score is still 0.50 which suggests that the model is now getting a large number of false positives for not offensive tweets. Despite these drawbacks, there is still a significant improvement in the overall accuracy of the model, 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.

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. This suggests that this model is good at identifying relevant cases of not offensive tweets, but is poor at only returning though cases.

Supplementary Materials

1. [https://competitions.codalab.org/competitions/20011](https://competitions.codalab.org/competitions/20011#results) [↑](#footnote-ref-1)
2. Banks, J., 2010. Regulating hate speech online. *International Review of Law, Computers & Technology*, *24*(3), pp.233-239. [↑](#footnote-ref-2)
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. [↑](#footnote-ref-3)
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). [↑](#footnote-ref-4)
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. [↑](#footnote-ref-5)
6. Cresti, M., Martino, V. and Rosola, M., Kate Manne. Down Girl. The Logic of Misogyny, Oxford University Press, 2017, pp. XXIV, 338. *APhEx*. [↑](#footnote-ref-6)
7. <https://www.cps.gov.uk/hate-crime> [↑](#footnote-ref-7)
8. <https://scikit-learn.org/stable/modules/tree.html> [↑](#footnote-ref-8)
9. <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html> [↑](#footnote-ref-9)
10. <https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html> [↑](#footnote-ref-10)
11. <https://en.wikipedia.org/wiki/Logistic_function> [↑](#footnote-ref-11)
12. <https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html> [↑](#footnote-ref-12)
13. <https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html> [↑](#footnote-ref-13)
14. 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*. [↑](#footnote-ref-14)
15. 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*. [↑](#footnote-ref-15)
16. <https://www.python.org/> [↑](#footnote-ref-16)
17. <https://scikit-learn.org/> [↑](#footnote-ref-17)
18. <https://radimrehurek.com/gensim/> [↑](#footnote-ref-18)
19. <https://numpy.org/> [↑](#footnote-ref-19)
20. <https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9> [↑](#footnote-ref-20)
21. <https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9> [↑](#footnote-ref-21)
22. <https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9> [↑](#footnote-ref-22)
23. <https://docs.python.org/3/library/os.html> [↑](#footnote-ref-23)
24. <https://radimrehurek.com/gensim/models/word2vec.html> [↑](#footnote-ref-24)
25. <http://nadbordrozd.github.io/blog/2016/05/20/text-classification-with-word2vec/> [↑](#footnote-ref-25)