Classification Challenge Code Writeup

Candidate 13343

Introduction

In `writeup.pdf`, I explain my process of selecting a model that classifies a set of Wikipedia Talk Page comments either as an attack or not. The full code for the analysis and all models are in the `writeup.rmd` file, from which the PDF file is compiled. The full code for the final submitted model is presented in `code.rmd` and compiled to `code.pdf`, with the details of the model explained in the writeup.

I first load and explore any statistics or patterns of the data. Then I create a feature representation of the data using a bag-of-words approach and add a few custom features. I then test several learning algorithms including Naive Bayes, LASSO, SVM, Random Forest, and Multi-Layer Perceptron model. I select the best performing model based on its F1 score on a validation set and then use the selected model to predict labels for the 100,000 observations in the test set and submit these predictions to the online challenge.

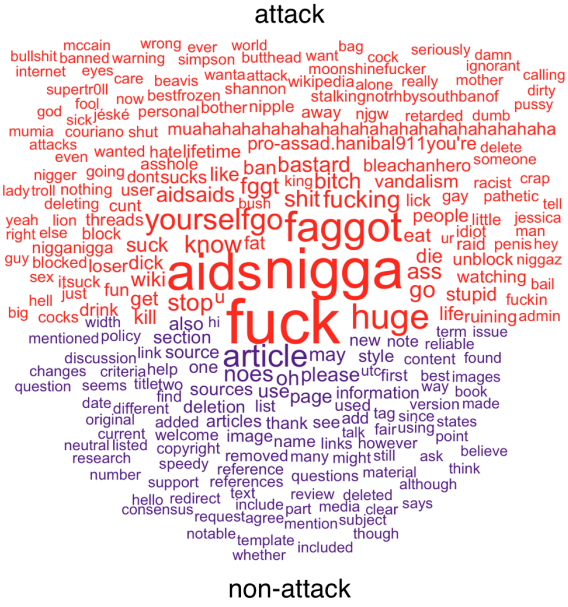
Loading and Exploring the Data

The data are comprised of two sets: **(1)** 15,000 observations in the training data ("training set") with comment text labelled either as *1* - an attack or *0* - not an attack, and **(2)** 100,000 observations in unlabeled held-out data ("testing set") for which the selected model will predict labels. Table 1 shows the distribution of the class labels. We can see that it is unbalanced with there being a larger proportion of non-attack comments.

Table 1: Distribution of labels in training data

Attack label	Frequency
0	11778
1	3222

The wordcloud below gives a brief overview of the top features across comment type in the training set, excluding punctuation, symbols, and English stopwods. **Content warning:** It is apparent from this visualisation that attack comments exhibit more profane, aggressive language.



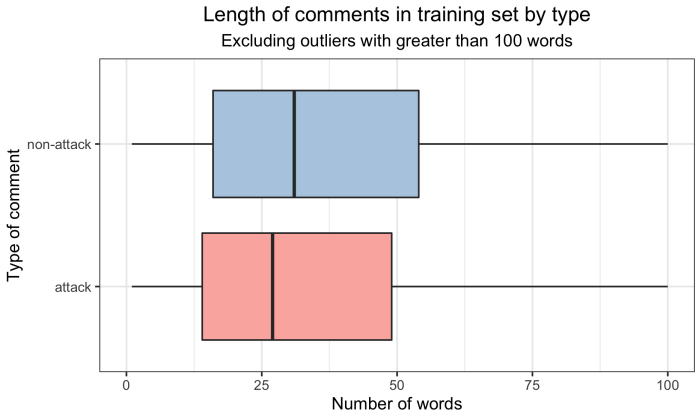
Descriptive Analysis

Length of comments

I first explored differences of comment lengths (based on number of words) between attacks/non-attacks. We can see the difference is not statistically significant, and the distribution of comment lengths is shown in the boxplot below.

Table 2: Welch Two Sample t-test: `length_noattack` and `length_attack`

Test statistic	df	P value	Alternative hypothesis	mean of x	mean of y
0.6508	4411	0.5152	two.sided	82.17	80.12

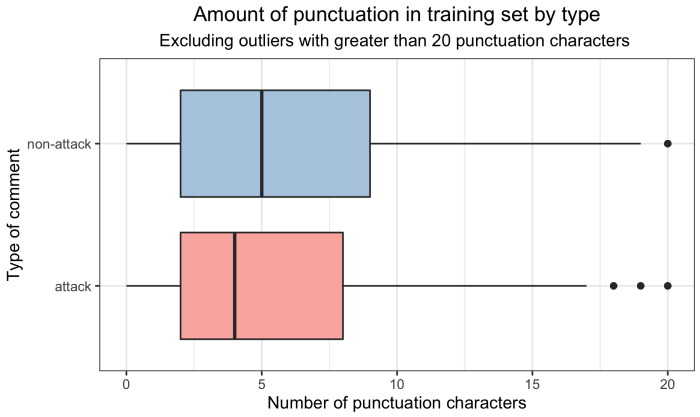


Amount of punctuation

Moreover, the average number of punctuation characters doesn't significantly differ either, with attack comments only having about 0.1 more on average. This is shown by the t-test and boxplot below.

Table 3: Welch Two Sample t-test: `length_punc_noattack` and `length_punc_attack`

Test statistic	df	P value	Alternative hypothesis	mean of x	mean of y
-0.1454	3766	0.8844	two.sided	10.76	10.86



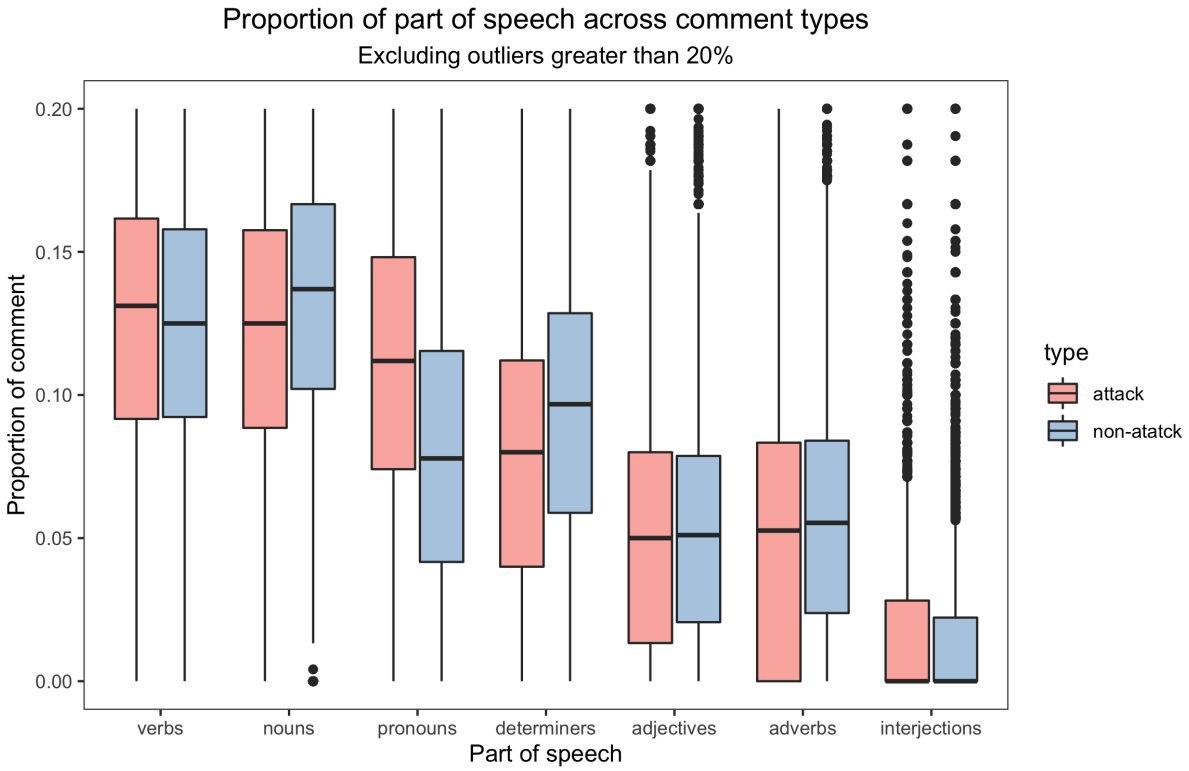
Parts of speech

Next, I looked at the average proportion of different parts of speech across each comment type. This was done by using `spacyr` to carry out part of speech tagging on each unique word in the corpus. I then used each part of speech as an associated dictionary key, and applied the custom dictionary to the data, where each comment was represented as a vector of word proportions (to normalize for comment length).

The differences in proportion of parts of speech between the two comment types are shown in the table and boxplot below (which excludes proper nouns, symbols and numerals because the proportions are too small to visualise). We can see that, generally, each type of comment contains relatively the same proportion of parts of speech, with attack comments featuring more pronouns, possibly indicating they are more personal.

Table 4: Average proportion of each part of speech in each comment

	Attack	Non-attack
adjectives	0.058	0.062
adverbs	0.062	0.058
determiners	0.097	0.083
interjections	0.017	0.022
nouns	0.161	0.147
numerals	0.013	0.007
pronouns	0.090	0.130
proper_nouns	0.001	0.000
symbols	0.003	0.001
verbs	0.145	0.150



Number of words in all caps - custom feature

Words in all capital letters tend to indicate shouting or anger online, which might suggest an attack. To find the number of all caps words in each comment I first removed all URLs in the text to avoid capturing all caps URLS. Then I counted the number of all caps words with two or more letters. Below is the beginning of the comment with the highest number of capital letter words ("OH NOES" repeated over one thousand times).

```
## [1] "OH NOES OH NOES OH NOES OH NOES OH NOES OH NOES OH NOES OH NOES OH NOES OH NOES"
```

Below is the beginning of another example of a comment with many capital letters. The above is not labelled as an attack, whereas below is.

```
## [1] "IT'S NO USE!!!!  I DIDNT EVEN GET A CHANCE TO ADD LINKS AND STUFF BECAUSE YOU"
## [2] "OR WHOEVER ELSE KEEP MESSING IT UP!! YOU GUYS ARE ANNOYING LET ME AT LEAST GET"
## [3] "A CHANCE TO FINISH THE DAMN PAGE!"
```

Below we can see the number of all caps words differs across comment type, with attack comments typically containing a greater amount of all caps words. Therefore, I decided to add this as a custom feature, as perhaps the number of capital words would help discriminate between comment type.

Table 5: Welch Two Sample t-test: `num_capital_words_noattack` and `num_capital_words_attack`

Test statistic	df	P value	Alternative hypothesis	mean of x	mean of y
-4.159	3521	3.273e-05 * * *	two.sided	1.2	5.49

Number of exclamation marks - custom feature

The number of exclamation marks might indicate anger, which might more likely be an attack online. The beginning of comment below is the comment in the training set with the greater number of exclamations, which is labelled as an attack.

```
## [1] "Wikipedia, I have one itty bitty question. LEAVE MY EDITS ALONE!!!!!!!!!!!!!!!!!"
```

We can see in the table and t-test below that indeed comments which are attacks have a greater number of exclamation marks, on average. Therefore, I added this as a custom feature as well, since other punctuation is excluded from feature representation.

Table 6: Welch Two Sample t-test: `num_exclamation_mark_noattack` and `num_exclamation_mark_attack`

Test statistic	df	P value	Alternative hypothesis	mean of x	mean of y
-3.573	3238	0.0003574 * * *	two.sided	0.2697	1.817

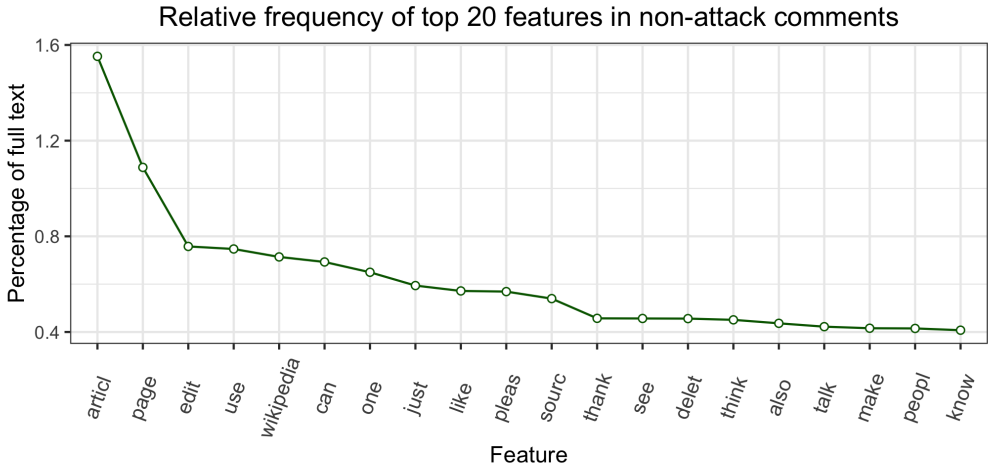
Feature Representation

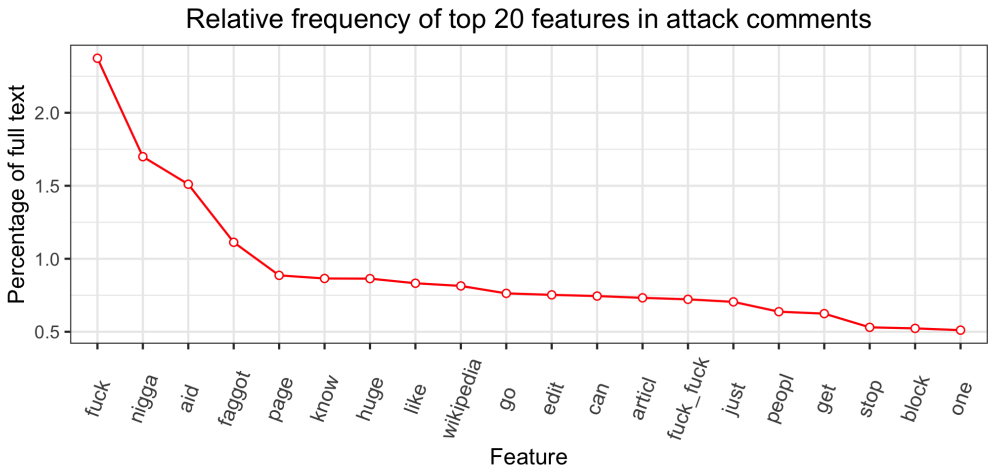
Creating a document-feature-matrix

After exploring the data, I chose to use a bag-of-words approach for feature representation. I transformed the texts for both the training and testing set into document-feature-matrices (dfms), where each document (i.e., each comment) is represented by a vector of word counts for each unique word in the corpora. I removed all punctuation, numbers, urls, symbols, words with only 1 letter, and English stopwords from the texts in each dataset. I removed these as most of these features would not provide much discriminative information (aside from exclamation marks, later added back).

I also included bi-grams in each dfm to identify possible collocations, and, again, to only include the most important and discriminant features, I trimmed the dfm to only include words that occur in more than 10 documents. Finally, I stemmed the words to prevent collinearity. For training the models and using them for prediction, I made sure the features were identical between the two datasets by joining on the common words and creating a combined dfm.

Next, I identified and removed more top frequency words across both comment types which were likely to be non-discriminative. **Content warning:** Shown in the below plots are the top 20 words in each comment type and their relative frequencies. I decided to trim the common features "article", "page", "edit", "wikipedia", "can", "one", "just", and "like" from the dfm.





Profanity count - custom feature

Next, I created another custom feature which counts the frequency of profanity words in each document. This could again be an indicator of an attack. To do so, I applied a dictionary of profanity words, which was a merging of the profanity dictionaries from the `lexicon` r package and from the "Offensive/Profane Word List" from Luis von Ahn's Research Group. In total, the dictionary had 1222 profanity words.

Attack comments contain an average of 8.02 profane words, where as non-attack comments contain an average of 0.5. A t-test was performed on comments with at least 1 profane word, and we can see that attack comments still have a significantly higher amount.

Table 7: Welch Two Sample t-test: `profanity_noattack` and `profanity_attack`

Test statistic	df	P value	Alternative hypothesis	mean of x	mean of y
-3.729	2022	0.0001977 * * *	two.sided	2.4	12.78

Sentiment and emotion - custom feature

Next, I added more custom features which indicate the comment sentiment and emotion, using the NRC lexicon. This pre-made dictionary corresponds words across two sentiments (positive and negative) and eight emotions (anger, anticipation, disgust, fear, joy, sadness, surprise, and trust).

I applied the NRC dictionary to the dfm where word counts were weighted to be proportions to normalize for document length. We can see that on average, comments which are attacks have greater anger, disgust, fear, sadness, and surprise, and overall greater negative sentiment. Therefore, I used these values as features in the dfm, and excluded the emotions of anticipation and joy, for which there was no significant difference.

Table 8: NRC Lexicon average sentiment and emotion in each document

	anger	anticipation	disgust	fear	joy	negative	positive	sadness	surprise	trust
non_attack	0.022	0.035	0.011	0.026	0.025	0.050	0.093	0.022	0.015	0.060
attack	0.049	0.034	0.052	0.046	0.027	0.116	0.075	0.042	0.019	0.052

Table 9: Welch Two sample t-test for difference across comment type

emotion	p_value
anger	1.15e-51
anticipation	0.677
disgust	5.83e-84
fear	8.35e-29
joy	0.184
negative	9.08e-118
positive	4.94e-21
sadness	4.24e-32
surprise	8.84e-05
trust	3.56e-07

Model Selection

After representing the features in the data as bag-of-words and adding custom features, I also weighed the dfm using tf-idf weighting, which is a combination of term frequency (to remove rare terms) and inverse document frequency (weights words that appear in the most documents). I tested both the weighted and unweighted dfms in the analysis.

To begin, I subsetting a labelled, held-out dataset from the training data to use as a validation set to approximate the performance for the test set. I used 3,000 observations for the validation set, leaving 12,000 observations to train the models on (80% of the training data).

Naive Bayes

The first model I tested was Naïve Bayes, a supervised learning model popularly used for text data. The model uses Baye's rule to determine the class posterior probabilities of each word, and then aggregates to determine the class probabilities for each document, assuming conditional independence. My first submission to the competition was using the Naive Bayes model on the dfm without any of the custom features mentioned above. This resulted in an F1 score of 0.5938 on 20% the testing set. For my second submission, I added the custom features to the model (exclamation count, capital words count, sentiment/emotion proportion scores) and this yielded an F1 score of 0.64558 on 20% of the testing set.

I ran the model on both the unweighted and tf-idf weighted dfm, and performance of the Naive Bayes model (including all custom features) on the training and validation sets is shown below, with the best validation score bolded below.

Table 10: Naive Bayes model performance

dfm_type	dataset	precision	recall	f1	accuracy
unweighted	training	0.678	0.636	0.656	0.776
unweighted	**validation**	**0.656**	**0.615**	**0.635**	**0.765**
weighted	training	0.717	0.628	0.670	0.779
weighted	validation	0.674	0.583	0.625	0.755

LASSO regression

Next, I applied a Lasso regression model, which performs regression but adds a shrinkage penalty to large coefficients and shrinks them towards zero using the L1 norm. This is useful because often text data has many features but is sparse, and a model with regularization allows for a sort of feature selection of the most important words.

I implemented the model using 5-fold cross-validation to select the optimal tuning parameter lambda, which additionally controls the way in which coefficients are shrunken. Since the class distribution is imbalanced, I also added class weights (shown in the table below) to the model hyperparameters to give more attention to the minority class (attack).

Table 11: Class label weights

Class	Weight	Frequency
1	0.2148	7811
0	0.7852	2189

Despite the advantages of this learning model, the LASSO on both the weighted and unweighted dfm performed worse than Naive Bayes. The log-lambda value was the same for both dfm types, with the weighted dfm performing slightly better on the validation set. Perhaps the pre-processed dfm was not sparse or informative enough for the model to be effective and would instead benefit from using ridge regression and the L2 norm, where coefficients are not completely zeroed out. These are questions to explore in further analysis.

Table 12: Lasso model performance

log_lambda	dfm_type	dataset	precision	recall	f1	accuracy
-5.276	unweighted	training	0.594	0.726	0.654	0.794
-5.276	unweighted	validation	0.508	0.669	0.577	0.753
-5.276	weighted	training	0.618	0.756	0.680	0.813
-5.276	**weighted**	**validation**	**0.513**	**0.669**	**0.581**	**0.755**

Support Vector Machine

I next tried using a linear support vector machine (SVM) on the data, which finds a separating hyperplane between the two classes. I expected the model would perform well since textual data is high-dimensional and therefore is very likely linearly separable. Therefore, I would expect an SVM with a linear kernel that finds a linear decision boundary to perform well.

I again added class weights for the model to focus on observations inversely proportional to the class distribution. I tuned the cost hyperparameter C, for values of 0.001, 0.01, 0.1, 1, 5, and 100. The cost C acts as a budget for the amount and degree to which the margin (i.e., the distance from the hyperplane to each class) is violated by the observations. This indeed was able to outperform the two previous models, and the top values are shown below. Oftentimes lower values of C created high fit but high variance, and higher values potentially introduced too much vias by allowing for more violations to the margin.

Table 13: Linear SVM model performance

dfm_type	dataset	cost	precision	recall	f1	accuracy
unweighted	training	0.01	0.681	0.618	0.648	0.768
unweighted	validation	0.01	0.638	0.603	0.620	0.757
weighted	training	0.01	0.772	0.755	0.763	0.846
weighted	**validation**	**0.01**	**0.659**	**0.643**	**0.651**	**0.778**
unweighted	training	0.10	0.736	0.820	0.775	0.868
unweighted	validation	0.10	0.599	0.692	0.642	0.786
weighted	training	0.10	0.824	0.909	0.864	0.927
weighted	validation	0.10	0.601	0.688	0.641	0.785
unweighted	training	1.00	0.823	0.926	0.872	0.935
unweighted	validation	1.00	0.566	0.690	0.622	0.776

Random Forest

The random forest (RF) model fits decision trees to many boostrapped sample of the training data, where at each split only a portion of all features is considered for the split. This model therefore reduces the variance of a single decision tree by taking multiple samples, and it also decorrelates the trees by adding regularization and preventing overfitting. This works well for text classification since RF is suited for dealing with high-dimensional and noisy data.

To select the model hyperparameters, I used a randomized grid search over discrete values of different tuning parameters, shown in the table below. First I tuned "mtry", the number of features to consider at each tree split (with values between 0.5 to 3.5 times the square root of the total number of features). Other hyperparamters I tuned included ones which control the tree size and complexity including "min.node.size", the minimal node size, "max.depth", the maximal tree depth, and the overall number of trees to build, "num.trees". A model which is too complex may overfit the training data, so therefore hyperparameter tuning is important. I again added a class weights hyperparameters to account for the class imbalance.

Table 14: Hyperparameter values used for grid search for random forest model

hyperparameter	values
mtry	35, 70, 105, 140, 176, 211, 246
min.node.size	1, 3, 5
num.trees	50 - 500
max.depth	50 - 250

I tested 40 different hyperparameter combinations on both the weighted and unweighted dfms, and the top 5 model results are shown below. The best performing model had an F1 score of 0.66 on the validation set, with the hyperparameters shown below in the bolded row in the table. This model was submitted as my third submission and yielded a 0.6587 F1 score on 20% of the test data.

Table 15: Random forest model performance

dfm_type	mtry	min node size	num trees	max depth	dataset	precision	recall	f1	accuracy
unweighted	35	1	373	242	training	0.856	0.968	0.909	0.961
unweighted	35	1	373	242	validation	0.611	0.715	0.659	0.798
unweighted	35	3	415	220	training	0.859	0.963	0.908	0.959
unweighted	35	3	415	220	validation	0.616	0.707	0.658	0.796
unweighted	105	3	313	123	training	0.930	0.973	0.951	0.976
unweighted	105	3	313	123	validation	0.643	0.674	0.658	0.788
weighted	35	3	371	158	training	0.874	0.947	0.909	0.955
weighted	**35**	**3**	**371**	**158**	**validation**	**0.637**	**0.683**	**0.660**	**0.791**
weighted	105	1	330	162	training	0.927	0.986	0.956	0.982
weighted	105	1	330	162	validation	0.642	0.671	0.656	0.787

Multi-Layer Perceptron Network Model

Lastly, I tried the multi-layer perceptron network model (MLP). This domain of artificial neural networks perform well for many classification problems, including text classification. There are a number of tuning parameters whose performance tend to vary on the dataset at hand. However, because of the complexity of neural networks, they are also more difficult to identify problems and adjust the model accordingly.

Because of the time and computational constraint of running the models, I again performed a random grid search over different hyperparameter values, along with fixed hyperparameter values of: class weights, binary cross-entropy for the loss function (common for binary classification), and accuracy for the metric.

The other hyperparameter values are shown in the table below. I selected three of the more popular optimizers, a set of a few different values for the number of nodes used in the first layer of the model

("units"), different epochs which tells the model how many cycles to go through the training set, and dropout rates ("dropout") which "determines the rate at which units are dropped for the linear tranformation of the inputs" (source: Quanteda Classifiers).

Table 16: Hyperparameter values used for grid search for MLP model

hyperparameter	values
optimizer	adam, rmsprop, sgd
units	256, 512, 1024, 2048
epochs	5-15
dropout	0.1 - 0.9 (increments of 0.1)

Moreover, I simplified the dfm I was using for this model because after a few tries it appeared the model performed better on this particular dataset when it was higher-dimensional. Therefore, I re-processed a dfm to only exclude stopwords and URLs, and I still carried out the steps of stemming the words, adding the custom profanity counts, sentiment/emotion scores and number of capital letters. I only used a tf-idf weighted dfm for training the MLP.

The first five model results are shown in the table below.

Table 17: MLP model performance

units	epochs	dropout	optimizer	dataset	precision	recall	f1	accuracy
512	11	0.6	adam	training	0.992	0.998	0.995	0.998
512	11	0.6	adam	validation	0.677	0.578	0.623	0.753
512	14	0.1	rmsprop	training	0.999	0.984	0.991	0.992
512	14	0.1	rmsprop	validation	0.702	0.574	0.632	0.755
256	13	0.6	adam	training	0.995	0.998	0.997	0.998
256	13	0.6	adam	validation	0.664	0.602	0.631	0.761
1024	10	0.4	adam	training	0.995	0.999	0.997	0.999
1024	10	0.4	adam	validation	0.699	0.531	0.604	0.736
512	10	0.8	sgd	training	0.691	0.741	0.715	0.824
512	**10**	**0.8**	**sgd**	**validation**	**0.616**	**0.686**	**0.649**	**0.789**

Above, because the training scores were lower but validation scores still relatively high for the gradient descent optimizer, "sgd", I assumed the model was still underfitting to the data. Since SGD in deep learning exposes one observation at a time to the network as input, it is a slower algorithm than the others.

Therefore, to continue tuning the model, I fixed the rest of the parameters (at 512 units and a 0.8 dropout rate) and attempted to improve the model performance by increasing the number of epochs. I expected this would improve performance since it gives the model more cycles through the data to learn it. This was successful and the results are shown in the table below for the SGD optimizer at different numbers of epochs, with 70 epochs resulting in the best performance.

Between 70 and 100 epochs, the validation set F1 score does not improve much, nor does the test set score in the competition. This suggests the model starts to overfit during an epoch between these two. Given more time, this is something that should be validated by looking at values in between and by cross-validating the results on other folds of the dataset.

Table 18: MLP model performance with SGD optimizer

epochs	dataset	precision	recall	f1	accuracy
20	training	0.7420	0.8070	0.7730	0.8640
20	validation	0.6070	0.7040	0.6520	0.7950
30	training	0.7760	0.8610	0.8160	0.9150
30	validation	0.6270	0.7140	0.6680	0.8570
35	training	0.7980	0.8780	0.8360	0.9080
35	validation	0.6500	0.7090	0.6780	0.8650
45	training	0.8080	0.9120	0.8570	0.9260
45	validation	0.6460	0.7210	0.6810	0.8110
55	training	0.8404	0.9366	0.8859	0.9473
55	validation	0.6560	0.7159	0.6847	0.8677
70	training	0.8749	0.9553	0.9133	0.9604
70	**validation**	**0.8720**	**0.6719**	**0.6893**	**0.8720**
100	training	0.8796	0.9729	0.9239	0.9650
100	validation	0.6687	0.7110	0.6892	0.8713

Final Submissions and Conclusions

Below is a table summary of the final submissions submitted to Kaggle as discussed in the previous sections. For each submission, it features the model type, adjustment from the previous submission, CV F1 score, and test F1 score. The MLP model using the SGD optimizer with 70 epochs had the best performance on both the validation and test data, bolded below.

Table 19: Kaggle submissions

submission	model	adjustment_made	cv_f1	test_f1
1	Naive Bayes	none	0.5890	0.5938
2	Naive Bayes	custom features	0.6350	0.6456
3	Random Forest	use RF model	0.6600	0.6587
4	MLP	use MLP model	0.6680	0.6904
5	MLP	increase to 45 epochs	0.6810	0.6952
6	**MLP**	**increase to 70 epochs**	**0.6893**	**0.7033**
7	MLP	increase to 100 epochs	0.6892	0.6913

In this analysis, I explored a number of ways to approach classifying text data. Although a many models performed similarly, they each use a unique approach to mapping and predicting the data. Although it ended up that the Multi-Layer Perceptron Network model was the best performing, selecting the best tuning parameters and text pre-processing steps for this model was limited by computational efficiency and time constraints. Moreover, there was not a very principled approach to understanding the model.

Therefore, with the pure aim of prediction, the MLP approach is superior. However, when it comes to understanding how the texts are used as data, interpretability, and error troubleshooting, other models such as SVM or Naive Bayes are perhaps more suitable. In future work, I would explore a wider range of hyperparameters for each model as well as explore different custom features and data cleaning aside from what is initially done here (e.g. look at use of pronouns, misspellings, emojis, readability/lexical diversity scores).