# Milestone 3

## Approach

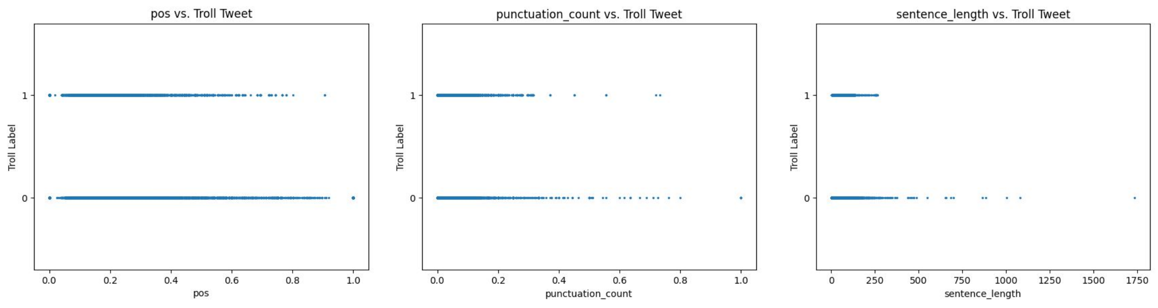
The data for this project came from an existing dataset from the online source Kaggle. The dataset itself is a single json file that consists of a string (the Tweet) and an integer (the classification). Originally it contained 20,000 sets of data.

For this dataset, the context about the Tweets aren't provided. The only context given is how the labellers labeled the data. Specifically, the labellers of the dataset looked for 'aggression' in the text, and label it '1' if aggression exists and '0' if it doesn't. To give an idea of the structure of the json file, a few lines are presented below:

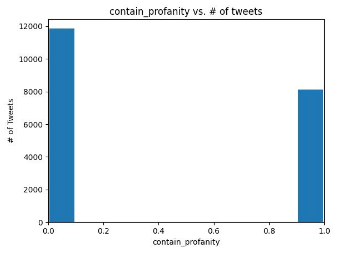
```
{"content": " Get fucking real dude.","annotation":{"notes":"","label":["1"]},"extras":null}
```

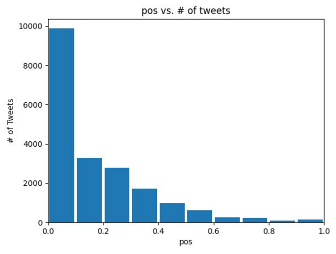
```
{"content": " She is as dirty as they come  and that crook Rengel  the Dems are so fucking corrupt it's a joke. Make Republicans look like","annotation":{"notes":"","label":["1"]},"extras":null}
```

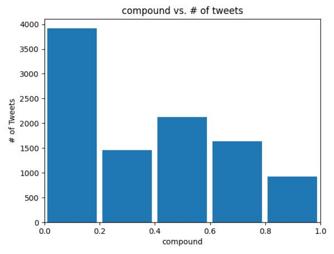
The distribution and correlation of the data was analyzed before we started to code our solution to see if the distributions made sense and so that we could form hypotheses for our models. The graphs below show the effect that a feature has on the output label. Positive sentiment is a value that was harvested from Vader, and fits what we hypothesized since non troll Tweets have a distribution that skews towards positive sentiment. There are other interesting correlations with a Tweets punctuation and length. The larger the count, the less likely a Tweet is to be a troll Tweet.



Distributions of feature values are also important to analyze. Distributions show how many of each value a feature has, which can be important in the case that the data isn't represented enough. It's also useful to know how distributions differ and how they correlate to the population distribution.







For Tweets that contain profanity, there's almost 60% that don't and 40% that do, which is a fairly evenly distributed split. In the middle graph, it seems like very little Tweets show signs of positive sentiment. This isn't a bad distribution of data since just because Tweets don't contain positive sentiment doesn't mean that they do contain negative sentiment. As for compound sentences, it seems like there's a distribution of all types of sentences, with many that are not compound sentences.

Natural Language Processing (NLP) was our first method to use because the project is centered around trying to understand human language, specifically troll language. It was hypothesized that NLP would perform very well because it is the branch of machine learning that is centered around human language.

Secondly, decision trees were chosen because the metrics we gathered during preprocessing are all quantitative. It would be interesting to see how a very simple model like a decision tree would compare against a more complex model like NLP.

## NLP

Sklearn, gensim, and nltk did most of the heavy lifting for this machine learning approach. Specifically, gensim was used for Doc2Vec, sklearn was used to bring in machine learning models to train on Doc2Vec's output, and nltk was used for tokenizing and preprocessing the Tweets.

| **Package** | **Version** |
|---|---|
| sklearn | 0.23.2 |
| gensim | 3.83 |
| nltk | 3.5 |
| pandas | 1.1.1 |
| seaborn | 0.11.0 |

This section will dive into the technical aspect of the project, mostly analyzing the code. The program was written in Python and primarily uses external machine learning libraries for core functionality. Only a handful of important packages are needed, which are outlined above. The steps below show the broad steps for the code:

1. Read and shuffle the dataset
2. Get train, test split
3. Generate models
4. Fit and predict each model
5. Plot graphs

The first step in our code is to read the data from the csv file that we preprocessed. Once read into a dataframe, the data is shuffled. The same random state is given across runs so that the shuffle can be uniform between runs. Also, during the shuffle, the columns remain in the same order, but the rows themselves are put into a random order. Next, the data was split with 20% for testing and 80% for training.

To find the best hyperparameters for this method, a method similar to grid search was programmed manually. The user provides the model name and hyperparameters in a file, and our implementation parses the file and runs those models. How the technical implementation works is that we read the csv file, match the model to a mode in sklearn, then fit and test each model in the file.

Different models are used in the natural language processing method because Doc2Vec (and typically other NLP methods) don't actually do the classification of the text. What models like Doc2Vec do is that they convert text into a vector of numbers that can then be trained by typical machine learning algorithms. We train a Doc2Vec model in our code which then returns a vector representation for word embeddings. These embeddings are then passed into the different machine learning models as specified in the csv file.

As mentioned above, Doc2Vec simply converts strings into vector representation numbers. It does this by trying to maximize the average log probability, like in the equation below [3]. Doc2Vec is a network that is trained to try to predict the next word in a string, has to be trained, and has hyperparameters as well.

$$\frac{1}{T}\sum_{t=k}^{T-k}\log p(w_t|w_{t-k},...,w_{t+k})$$

Where w is a word, t is the number of words, k is the number of words to predict.

To make the runtime faster, there is a caching layer where trained models are loaded from a .model file. That way, our program doesn't need to spend time and compute resources retraining already trained models. The program will go through each classifier and if it exists, loads it, but otherwise will create the classifier which is saved locally. This allows testing of a wide range of classifiers and hyperparameters without hard coding classifiers and with the benefit of significantly reducing runtime. Joblib is used to save the classifier locally.

For performance evaluation, sklearn's classification_report, accuracy_score, and confusion_matrix were all used to report statistics on how each model performed. These metrics were based on the accuracy, precision, recall, F1 score, as well as the false positive, false negative, true positive, and true negative values of a model. The results of these metrics are explored in the evaluation and analysis section.

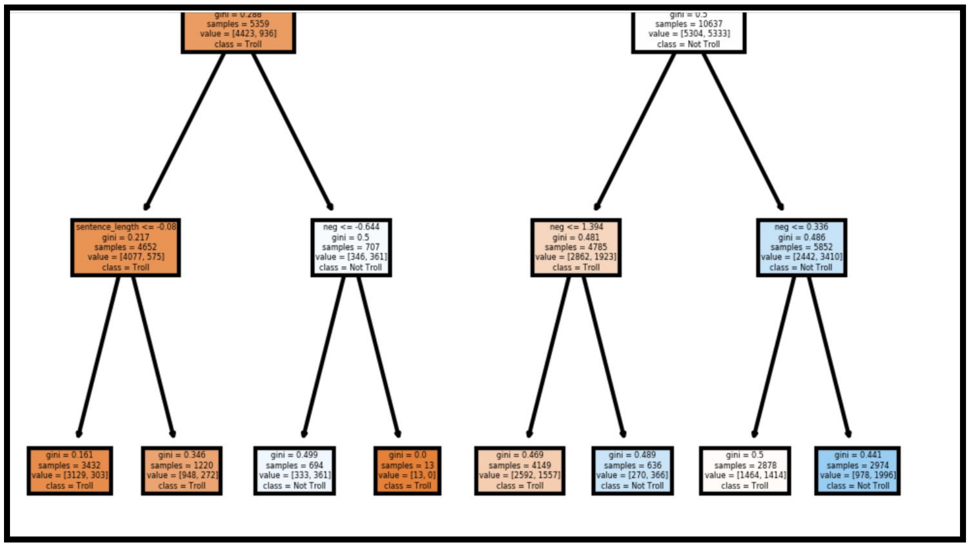
**Decision Tree**
Another method of prediction that was implemented was a decision tree. This was done using sklearn's package for the Python programming language. This decision tree was built through calculating the Gini impurity (or Gini index) on each feature in each row of the training data.

The Gini index is the proportion of the samples that belong to the class c for the particular node. This is essentially a measure of how often a randomly chosen element from the set would be incorrectly labeled if it was randomly labeled according to the distribution of labels in the subset. This is calculated through summing the squared probabilities of each class:

$$I_G = 1 - \sum_{j=1}^{c} p_j^2$$

The way that sklearn's decision tree module works is that the feature with the highest Gini index is the feature that is split on. This is done recursively on the training set until there is no more data to train on. Below is a demonstration of the tree with a max depth of 3. The top value is the feature it is split on at each node.

Figure 1: Decision Tree with Depth of 3



The data was split into a roughly 70/20/10 split for training, validation, and test data. This was done using train_test_split from sklearn.

The decision tree used in the project was tested under no constraints, as well as a variety of thresholds, as discussed in the following section. This includes maximum tree depth and minimum Gini impurity threshold.

- **Maximum Depth (max_depth)** - stops the tree from splitting once the maximum depth of the tree is reached. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than the number of samples
- **Minimum Gini Impurity Decrease (min_impurity_decrease) -** a node in the tree will only be split if the split results in a decrease of the impurity greater than or equal to this value

A coarse grid search was run on the validation set in order to gauge the best performing set of hyperparameters to run on the untouched test data. This was done using GridSearchCV from sklearn library. The coarse grid search tested a variety of parameters in order to get a high level perspective about which hyperparameters performed the best. From the results of the coarse grid search, a fine grid search was conducted to test more specific parameters. The best performing combination of hyperparameters was then tested on the untouched test set and evaluated.

## Evaluation

## NLP

As briefly mentioned above, to find the best classifier for the sentence vectors generated from Doc2Vec, a testing framework was created in order to test a wide range of classifiers and hyperparameters. This framework read a list of classifiers and hyperparameters provided in classificationdata.csv, located in the data folder. This file contains a list of classifiers and hyperparameters. For example, here are some of the entries in the document:

```
20,KNeighborsClassifier,5,0,0
20,KNeighborsClassifier,10,0,0
20,GaussianNB,1e-9,0,0
20,SVC,"poly",500,2
20,SVC,”poly",10,3
```

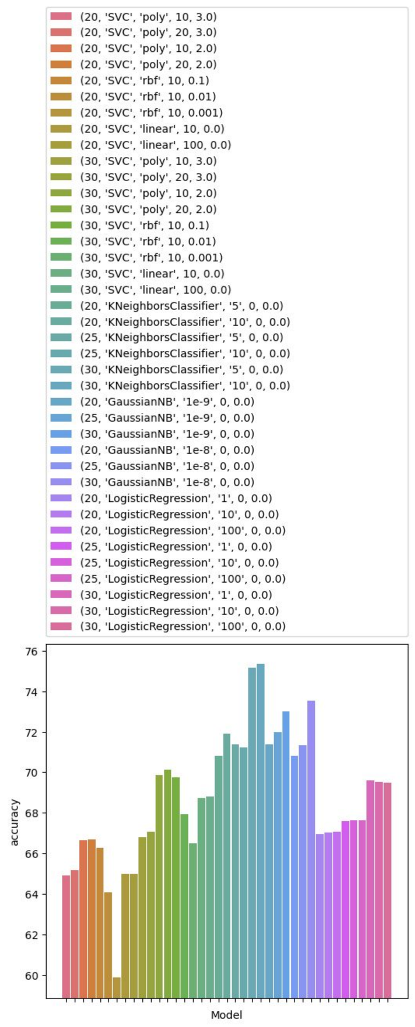
The first column is the number of vectors that Doc2Vec will generate with the word embeddings, the second column is the name of the classifier, and the third through fifth columns are any related hyperparameters. For each classifier, not all of the hyperparameters are necessarily used. For example, KNeighborsClassifier only uses one hyperparameter unlike SVC which uses 3. The hyperparameter for KNearestNeighbors is n-nearest neighbors, and SVC requires the kernel, C, and the gamma/degree value. Specifically, this projects explores the following models to learn Doc2Vec’s word embeddings vectors:

1. KNeighborsClassifier
2. DecisionTreeClassifier
3. GaussianNB
4. LogisticRegression

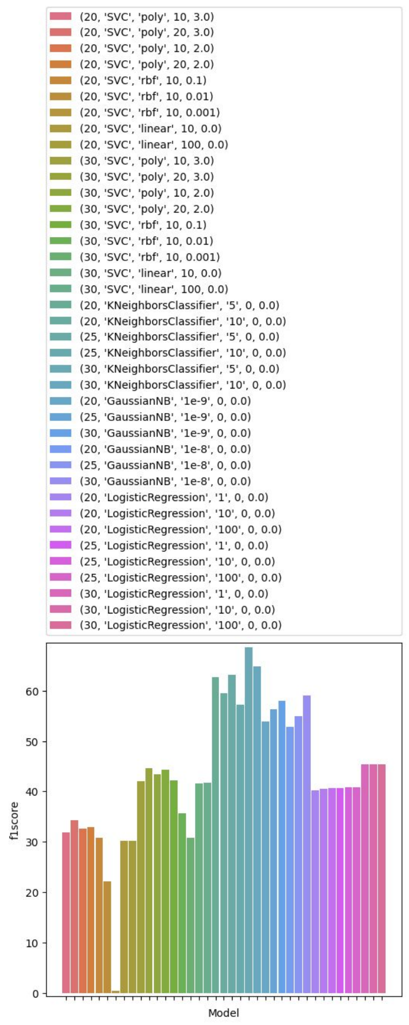
Using our training and testing data as described earlier, NLP was tested using the list of classifiers provided in the classification csv file. Each classifier had a confusion matrix image run for it after the test, and the average accuracies, F1 scores, and recall scores were compiled into individual images.

There were several ways that the NLP implementation was tested. The number of vectors that NLP generates per instance was varied between 20-30. Several different classifiers were tested as well. These include KNearestNeighbors, GaussianNB, LogisticRegression, and SVC, and a wide range of hyperparameters were explored for each of these. The list of parameters used will be present in the classificationdata.csv document and in the Appendix section.

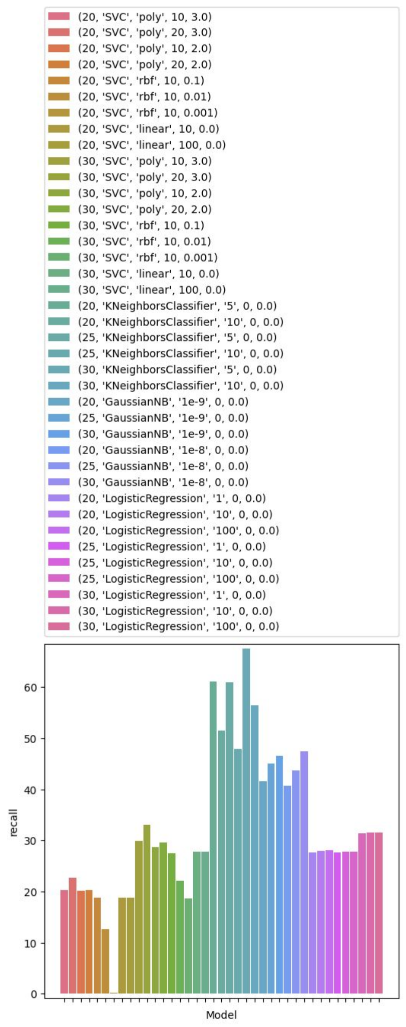
Accuracy:



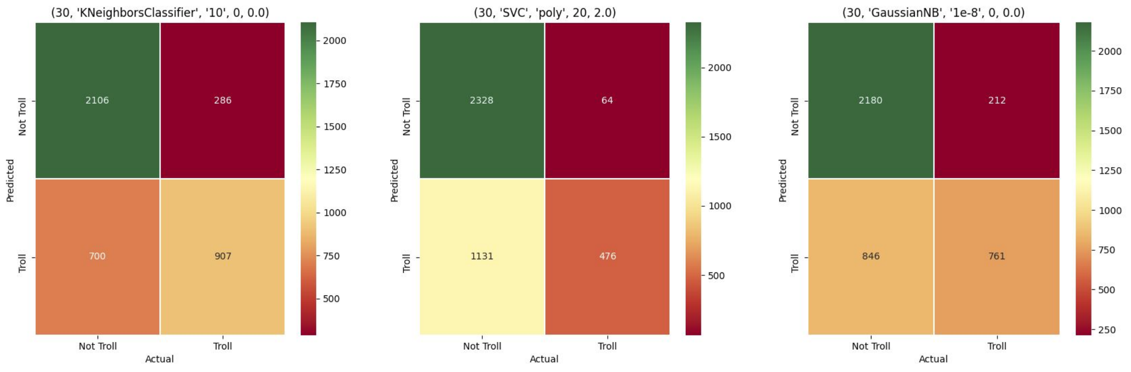
F1 Score:



Recall:



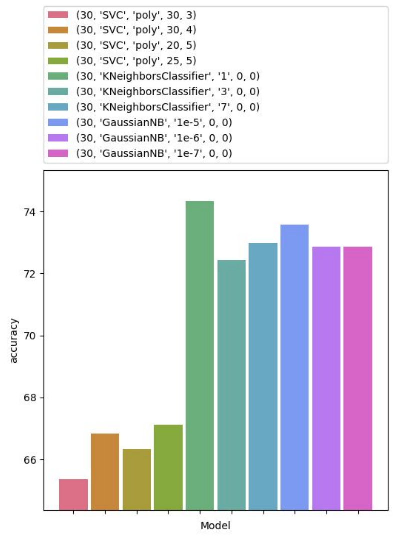
By looking at the graphs above, it's possible to see that the KNN models performed best across all classification metrics. KNN with K = 5 and K = 10 neighbors both got above 75% accuracy. Other models to note that did particularly well are the polynomial SVC model and the GaussianNB model. The highest accuracy for these models were 70% and 73%, respectively.



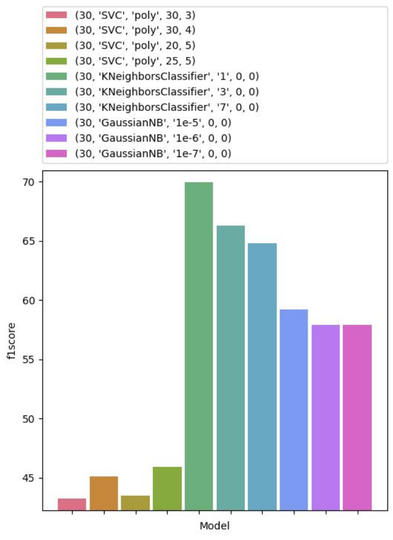
To find the best hyperparameters for these models, another run with a more fine search of hyperparameters was conducted. In other words, the KNN, SVC, and GaussianNB models were looked into more closely to find hyperparameters that get better results. The models for LogisticRegression and certain types of SVCs were dropped at this point.

The graphs below show the fine grid search with more tightly knit parameters. Fine grid search of parameters is meant to find a better performing model. Looking at how the models performed below, none reached above 75%, which was the high for KNN in the coarse grid search above. Though this is unusual, all this means is that the hyperparameters in the coarse grid search happened to perform the best. The Appendix section features the exact outputs from testing models.

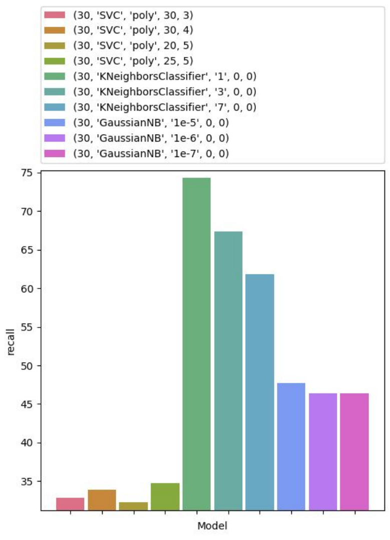
Accuracy:



F1 Score:

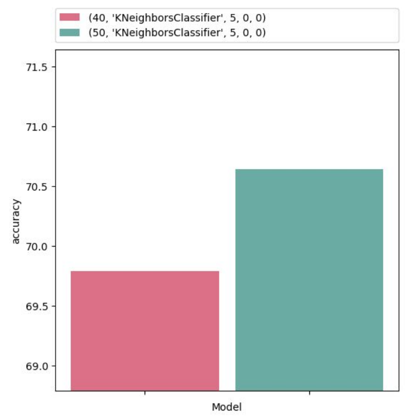


Recall:

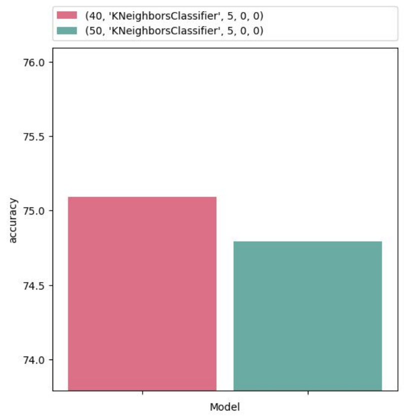


Finally, different parameters were applied to the best models from the fine grid search. For these runs, the Doc2Vec vector lengths were increased to see if a higher value would increase the accuracy. Another parameter that was tested was the preprocessing of the Tweet during runtime. One doesn't do any preprocessing to each Tweet, another only converts it to lowercase, and the final converts it to lowercase and removes common stopwords. Accuracies for those can be seen below:

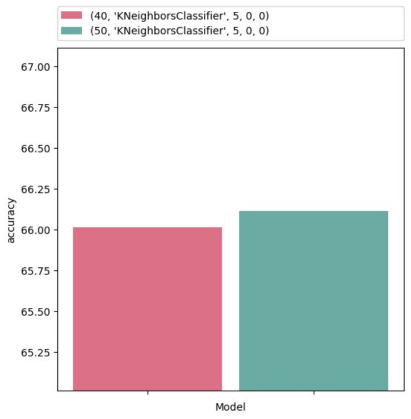
No change



Lowercase



Lowercase, No stop words



It can be concluded from above that increasing the Doc2Vec vector size didn't have much of an impact on the accuracy. The KNN model still performed at around 75% accuracy. No change to the Tweet caused the accuracy to fall significantly from just converting the Tweet to lowercase. The accuracy fell about 5% when the processing was taken away. On the other hand, when more preprocessing was added, and stop words were removed, the accuracy fell even more. By removing the stop words, the accuracy fell by about 9% compared to converting the Tweet to lowercase.

**Best Performing Hyperparameters**

| **Hyperparameters** | **Accuracy** | **Precision** | **Recall** | **F1 Score** |
|---|---|---|---|---|
| **KNN K=5** | 0.75 | 0.76 | 0.72 | 0.73 |
| **SVC, poly, C=20, D=2** | 0.70 | 0.78 | 0.63 | 0.62 |
| **GaussianNB, s=1e-8** | 0.73 | 0.75 | 0.69 | 0.70 |

All in all, KNN with K=5 and Doc2Vec's vector size as 30 performed the best by looking primarily at accuracy, but also at recall and the F1 score. This model reached a peak accuracy of 75.34%. Other models also performed well, like polynomial SVC at 70% and GaussianNB at 73%.

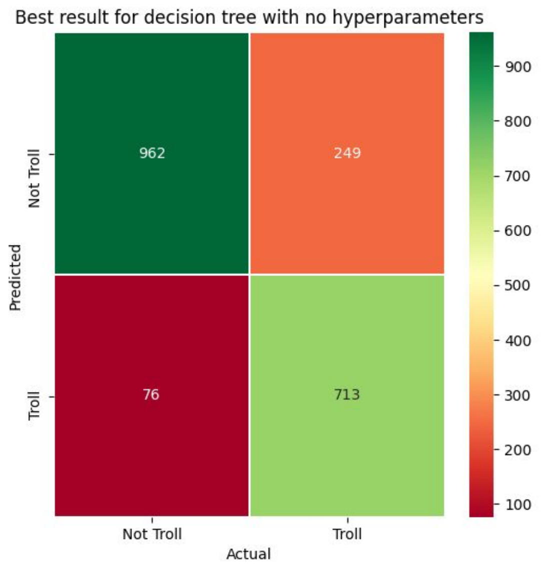
**DecisionTree**

The tree was originally trained without any thresholds to gauge the general performance of the tree. The results are as detailed in the table below.

**Decision Tree Performance (No Threshold)**

| **Accuracy** | **Precision** | **Recall** | **F1-score** |
|---|---|---|---|
| 0.85 | 0.75 | 0.94 | 0.85 |

With no thresholding, the tree ended with a depth of 43 and 2736 leaves, a significantly size decision tree. Performance was relatively high with no thresholds, with an accuracy close to 87 percent. This included more false positives than false negatives, with an average recall score higher than precision.The figure below gives a general sense of the accuracy, precision, and recall of the decision tree with no hyperparameters under the test set.



In an attempt to improve the performance of the algorithm, a coarse grid search was run on the training data in order to determine the best range of hyper parameters to threshold the tree on. This includes maximum depth of the tree and the minimum impurity decrease on a given node. Since the decision tree run with no thresholds yielded such a large depth, a wide array of depth values were tested in the coarse grid search. This is to give a high-level estimate of where the tree best performs under this dataset. The table below shows a detailed output of the coarse grid search. It displays the accuracy of the combination in the lightest shade, followed by precision in the middle, then recall in the darkest shade.

## Coarse Grid Search Results

| | **Min Impurity** | | | | |
|---|---|---|---|---|---|
| **Max Depth** | **.1** | **.01** | **.001** | **.0001** | |
| **4** | 0.61 | 0.67 | 0.68 | 0.686 | **Accuracy** |
| | 0 | 0.58 | 0.655 | 0.631 | **Precision** |
| | 0 | 0.559 | 0.378 | 0.474 | **Recall** |
| **6** | 0.61 | 0.67 | 0.68 | 0.687 | |
| | 0 | 0.58 | 0.654 | 0.642 | |
| | 0 | 0.559 | 0.381 | 0.453 | |
| **10** | 0.61 | 0.67 | 0.68 | 0.703 | |
| | 0 | 0.58 | 0.654 | 0.636 | |
| | 0 | 0.559 | 0.381 | 0.56 | |
| **20** | 0.61 | 0.67 | 0.68 | 0.758 | |
| | 0 | 0.58 | 0.654 | 0.677 | |
| | 0 | 0.559 | 0.381 | 0.725 | |
| **40** | 0.61 | 0.67 | 0.68 | 0.788 | |
| | 0 | 0.58 | 0.654 | 0.703 | |
| | 0 | 0.559 | 0.381 | 0.791 | |
| **80** | 0.61 | 0.67 | 0.68 | 0.787 | |
| | 0 | 0.58 | 0.654 | 0.701 | |
| | 0 | 0.559 | 0.381 | 0.792 | |

As seen from the results of the coarse grid search, a minimum impurity decrease of less than 0.01 performed extremely poorly, with a precision and recall of 0 for each run. Performance increased as maximum depth increased, and leveled out around a max_depth of 40, which was around where the tree stopped when tested with no thresholds.

Performance also increased as minimum Gini impurity decreased, performing best in the 0.0001 range. Overall, the performance is actually lower using the selected hyperparameters compared to the not thresholded model. This means the values chosen, primarily for minimum Gini impurity, were not best for the model. For the fine grid search, depths near 40 should be explored, in addition to minimum Gini impurities equal to and lower than 0.0001.

The fine grid search explores the hyperparameters mentioned above. The table below describes the results of the fine grid search.

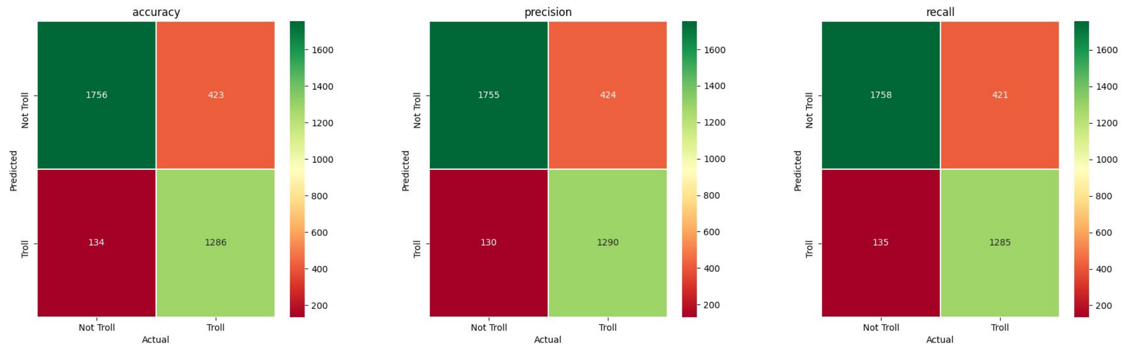
**Fine Grid Search Results**

| | **Min Impurity** | | | | | |
|---|---|---|---|---|---|---|
| **Max Depth** | **.0001** | **.00005** | **.00001** | **.000005** | **.000001** | |
| **30** | 0.787 | 0.818 | 0.841 | 0.839 | 0.841 | **Accuracy** |
| | 0.7 | 0.729 | 0.744 | 0.742 | 0.743 | **Precision** |
| | 0.796 | 0.855 | 0.901 | 0.902 | 0.904 | **Recall** |
| **40** | 0.787 | 0.819 | 0.842 | 0.843 | 0.85 | |
| | 0.701 | 0.728 | 0.747 | 0.746 | 0.750 | |
| | 0.799 | 0.857 | 0.904 | 0.906 | 0.910 | |
| **45** | 0.788 | 0.819 | 0.842 | 0.844 | 0.844 | |
| | 0.699 | 0.729 | 0.749 | 0.747 | 0.743 | |
| | 0.798 | 0.858 | 0.908 | 0.909 | 0.908 | |
| **50** | 0.788 | 0.821 | 0.842 | 0.841 | 0.844 | |
| | 0.701 | 0.731 | 0.744 | 0.748 | 0.742 | |
| | 0.8 | 0.858 | 0.904 | 0.908 | 0.905 | |

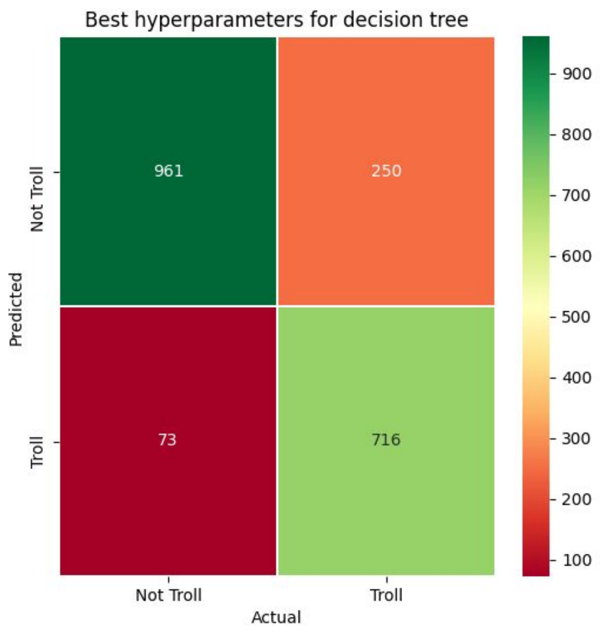
From the fine grid search, it is evident that the model performed much better with a max depth over 40 and a minimum impurity index of 1e-6. The combination of hyperparameters performed nearly identical, however 40/1e-6 performed slightly better than the rest, which will be investigated further on the test data.

The tables below give a general sense of the accuracy, precision, and recall of the best performing metrics under the validation set.

## Performance Metrics for Fine Grid Search



The best performing hyperparameters were then run against the untouched test data to determine actual performance. The figure below shows the confusion matrix for the best performing hyperparameters run with the test data.



The table below describes the performance of the best set of hyperparameters (max_depth 40/min_impurity_decrease 1e-6), as compared to the tree with no threshold that was tested first.

**Best Performing Hyperparameters**

| **Hyperparameters** | **Accuracy** | **Precision** | **Recall** | **F1 Score** |
|---|---|---|---|---|
| **Max_depth: 40, Min_impurity_decrease: 1e-6** | 0.85 | 0.75 | 0.91 | 0.82 |
| **No Threshold** | 0.85 | 0.75 | 0.94 | 0.85 |

From the table, it is evident that the adjusted hyperparameters did not have much of an impact on the performance of the tree. These performance values are not surprising based on the values of the hyperparameters. The tree thresholded ended with a depth of 43, meaning that the threshold of 40 was extremely close to the natural limit. The minimum Gini impurity decrease was also extremely small, around 1e-6. This means that the tree allowed most split to occur no matter the impurity; the Gini impurity decrease would need to be extremely small to threshold any values.

It is interesting to note that the best performing model was the one closest to not being thresholded. Performance may be limited due to the quality of the dataset. Performance could be improved through augmenting the dataset with more meaningful features or with feature reduction. This could also be improved by testing other hyperparameters offered by the decision tree classifier.

**Contribution**

- Kemal Fidan: Wrote base of nlp method, made confusion matrix and NLP plots, helped with report
- Nicholas Corbin: Wrote nlp method to include testing framework, helped with report
- Shreyank Patel: Wrote the base code for the decision tree, created data graphs, helped with the report
- Racheal Dylewski: Implemented grid search for decision tree plus some tweeks, helped with a large portion of report