

# Detecting Troll Tweets

**Kemal Fidan, Nicholas Corbin, Shreyank Patel, Racheal Dylewski**

Department of Electrical Engineering and Computer Science

Knoxville, Tennessee 37996

[kfidan, ncorbin2, spatel91, rdylewsk]@vols.utk.edu

## Abstract

On the internet, trolls are people who show anti-social and aggressive behavior in otherwise normal conversations. Many studies have shown a growing community of online trolls for a wide variety of reasons, and consequently, a need for blocking troll behavior has also risen. This study explores different machine learning approaches to best detect troll text from Twitter. Between a natural language processing approach and a decision tree approach, we were able to predict troll text at 85% accuracy.

## Introduction

Troll text is typically concerned with flaming or intentionally upsetting or joking with people on the internet. Identifying this type of text is important so that people's feelings aren't hurt or that they aren't offended. Pew Research Center found that 70% of 18 to 24 year olds who use the internet had experienced harassment [1]. Since there is no way of preventing someone on the internet to behave like a troll, identifying this type of text can be helpful in hiding what people do not want to see.

The purpose of this work is to explore and compare two machine learning algorithms performance on classifying troll text. The machine learning algorithms classify Tweets from Twitter as 'troll' or 'not troll'. In other words, this work dives into a binary classification problem to predict whether or not a Tweet is troll or not. In this project, troll text is defined as text containing aggression or targeted contempt. Ultimately, the project implements a decision tree algorithm as well as natural language processing to correctly classify Tweets. Also, for validation, grid search, cross validation, and graphs are presented.

## Dataset

### Data

The data for this project came from an existing dataset from the online source Kaggle [2]. 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.

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

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:

<pre>{"content": " Get fucking real dude.", "annotation": {"notes": "", "label": ["1"]}, "extras": null}</pre>
<pre>{"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}</pre>

The dataset was from a fairly reliable source, however accuracy cannot be guaranteed from an online source. To make sure, the description of how the data was labeled and then ran over a fraction of the dataset to see if the labeling would match the labeling that was already there. This provided a fair sanity check for the quality of the dataset. A few sets of data that did not match the team's definition of troll were filtered out through this method. The final dataset chosen seems to match the definition of aggression and is fairly accurate in its labeling. However, the dataset is limited to the Twitter platform and the dataset's definition of troll text.

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 in Figure 1 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.

As for the percentage of 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

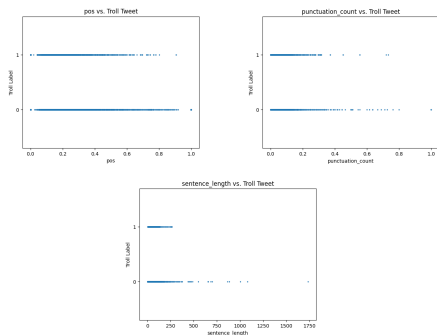


Figure 1: Correlation of data to the label

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.

## Preprocessing

The data from Kaggle was very complete. There were no NaNs or unknown data points in either dataset. However, the dataset still went through heavy preprocessing in order to better fit the requirements of the project. The first augmentation that was done was to convert the data in csv format instead of json format.

While the team did not sanity check the entire dataset (20,000 rows), rows where there was no agreement on the labeling were removed from the dataset entirely in order to not train the models on inaccurate data. This was done over time and only if someone on the team came across a mislabeled Tweet. This is because we pre-screened the dataset and agreed with most of the labels, so the dataset was considered the ground truth.

The data was also separated into additional features based on the content of the Tweet string. The updated dataset consists of a string (the Tweet), an integer (the classification), length of sentence, vader sentiment analysis values, punctuation count, binary classification of whether the string has profanity, and the number of profane words in the Tweet. The string is the text we are trying to classify that contains a short phrase or sentence. The integer classification is either a '0' or a '1'. A '1' means it is troll text, a '0' means it is non-troll text. The table below demonstrates a few sets of data and what they look like.

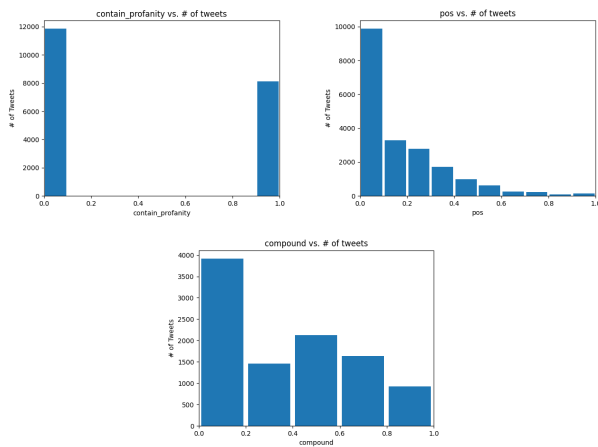


Figure 2: Distribution of data

Content	“Get fucking real dude”	“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	1	1
Len	22	129
Neg	0	0.098
Neu	1	0.743
Pos	0	0.159
Compound	0	0.2023
Punct_count	0.4545	0.0387
Profanity	1	1
Num_profanity	1	1

During runtime, the Tweets underwent some preprocessing before being fed into the machine learning models. Our team experimented with converting the Tweets to lowercase and removing stop words to see if the preprocessing would have an effect on the resulting accuracy. Since decision trees don't need scaled data, and the natural language processing method only looks at the Tweet, no scaling of the data was conducted.

## Approach

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

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

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

Figure 3: Doc2Vec, maximizing the average log probability. Where  $w$  is a word,  $t$  is the number of words,  $k$  is the number of words to predict.

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 earlier, Doc2Vec simply converts strings into vector representation numbers. It does this by trying to maximize the average log probability, like in the equation in Figure 3 [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.

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:

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. Figure 5 is a demonstration of the tree with a max depth of 3. The top value is the feature

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

Figure 4: Gini index calculation

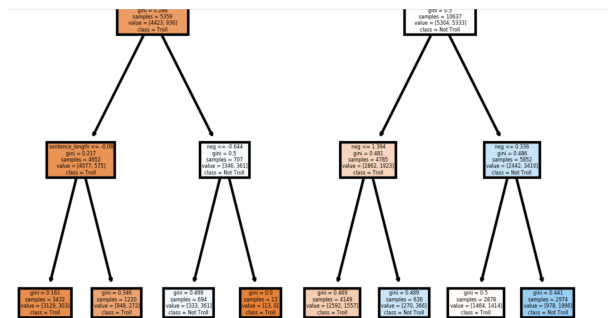


Figure 5: Decision tree with depth of 3

it is split on at each node.

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 earlier, 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.

By looking at Figure 6, 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 in Figure 8 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%,

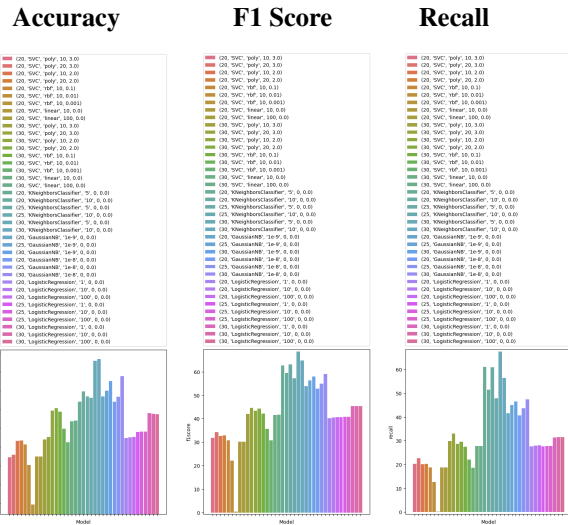


Figure 6: Coarse grid search results

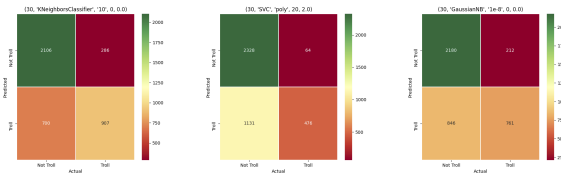


Figure 7: Confusion matrices for the top performing models

which was the high for KNN in the coarse grid search in Figure 8. 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.

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

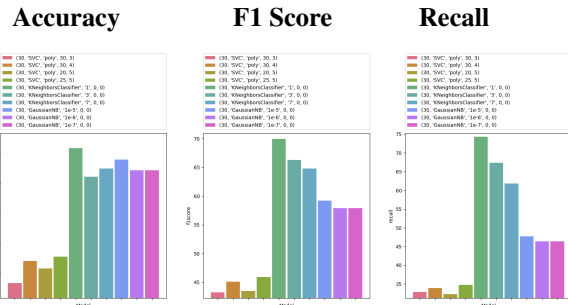


Figure 8: Fine grid search results

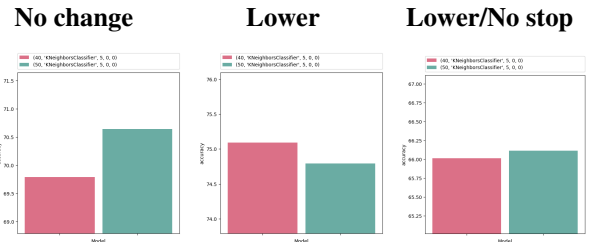


Figure 9: Comparison between different preprocessing methods

for those can be seen in Figure 9.

It can be concluded 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.

Hyperparams	Accuracy	Precision	Recall	F1 Score
<b>KNN K=5</b>	0.75	0.76	0.72	0.73
<b>SVC, poly, C=20, D=2</b>	0.70	0.78	0.63	0.62
<b>GaussianNB, s=1e-8</b>	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%.

## Decision Tree

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.

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. Figure 10 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

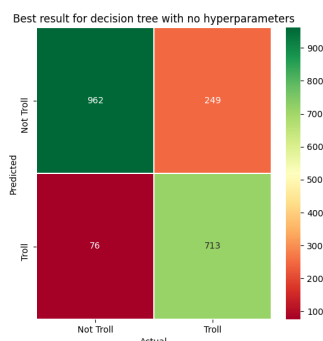


Figure 10: Confusion matrix for the no hyperparameters

	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	

Figure 11: Coarse grid search results

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.

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,

	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	

Figure 12: Fine grid search results

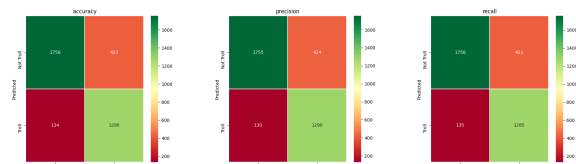


Figure 13: Performance metrics for fine grid search

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 earlier. The table below describes the results of the fine grid search.

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.

Figure 13 give a general sense of the accuracy, precision, and recall of the best performing metrics under the validation set.

The best performing hyperparameters were then run against the untouched test data to determine actual performance. Figure 14 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.

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



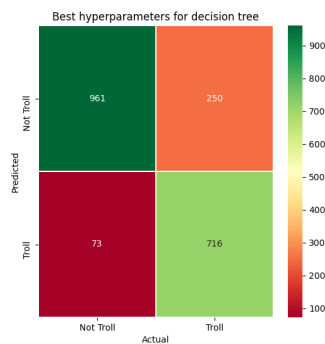


Figure 14: Confusion matrix for the best hyperparameters

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.

## Discussion

Throughout the project, there were a couple of interesting findings in both implementations. For the NLP method, it was assumed that the Doc2Vec model would learn information about the Tweets, as well as the Tweet's label. However, when it was time to implement the model, the Doc2Vec model was not giving back its predicted labels. After researching more about Doc2Vec, it was found that it is only used to convert text into a paragraph vector, so there is no direct predicting associated with it for this project's troll text use case. The paragraph vectors were then fed into different machine learning models to get predictions for our troll Tweet dataset.

When it was time to pick the models to train the Doc2Vec vectors with, online articles wrote that the GaussianNB model was a model that worked great with NLP. However, when different models were run against it, the GaussianNB model didn't hold up to the highest standards from previous work. Though the model performed the second highest, it had a 2% lower accuracy than the KNN model, which was an interesting finding.

Different preprocessing methods for each Tweet were also conducted during runtime. In the best case, the Tweet was converted to lowercase and then fed into Doc2Vec. It was

interesting to see that over preprocessing the Tweets resulted in a much lower accuracy than leaving the original Tweet. The accuracy fell about 4% by over preprocessing the Tweet, which made it apparent that removing the stop words from the Tweet removed valuable information for Doc2Vec.

For the decision tree, it was most interesting to find that the tree performed best under no restraints or very loose restraints. When performing a grid search of various values for the hyper-parameters, it was found that the more restrictive the threshold, the worse the model performed. The best performing tree reached the large depth of close to 40, and with a minimum impurity decrease of an almost negligible value of  $1e-6$ . This likely means that the classification was more complex, and that all features were influential in the decision making process. This is also surprising, however, as one would expect thresholding the splitting criteria to result in better performance and avoid overfitting.

Perhaps the most unexpected result from the project was that the decision tree method performed better than the NLP method, by about 10% in accuracy. The decision tree was run on information that was extracted from each Tweet, which resulted in concrete information about each Tweet that could be learned by the model. On the other hand, the slang and informal nature of Tweets probably made it difficult for NLP to curate the best set of paragraph vectors, which then made it hard to predict for the machine learning models.

## Conclusion

The purpose of this work is to explore and compare two machine learning algorithms performance on classifying troll text. The first uses Doc2Vec to harvest paragraph vectors, then trains different models with these vectors. The second is a decision tree algorithm that is trained with preprocessed data.

The findings between the models show that the decision tree method has a higher accuracy of 85% compared to the NLP method which has a peak accuracy of 7%. The accuracy, precision, and recall scores for the decision tree implementation are better than the NLP model by at least 0.09. However the NLP model excels in precision by 0.01. All in all, the findings are backed by cross validation, coarse and fine grid search, detailed metrics, and visual graphs.

## References

1. <https://www.pewresearch.org/internet/2014/10/22/online-harassment/>
2. <https://www.kaggle.com/dataturks/dataset-for-detection-of-cybertrolls>
3. <http://proceedings.mlr.press/v32/le14.pdf>

## Future Work

For future work, it would be fascinating to see how the Doc2Vec model could be improved. Maybe removing punctuation from Tweets would increase the change that Doc2Vec provides a more accurate paragraph embedding. Also, replacing slang words or acronyms could result in a

more formal blob of text for Doc2Vec to train on. Neural networks could also be tested for prediction. These models can find complex relationships between features and labels, which might uncover association between some of the data's features and the troll text label that the current models overlooked.

## Contributions

- **Kemal Fidan:** For Milestone 1, Kemal was involved in the discussions on Miro and Discord when picking a project topic. As for Milestone 2, he drafted the report for submission. For Milestone 3, he wrote base of NLP method, made confusion matrix and NLP plots, and helped with report. For Milestone 4, he made powerpoint and ported the report to latex.
- **Nicholas Corbin:** Wrote nlp method to include a testing framework, helped with report For Milestone 1, Nick helped to brainstorm ideas for the project, and participated in narrowing down the project to the current problem. Then for Milestone 2, Nick created a master document of possible datasets and outlined information about these datasets in order to choose the best dataset. Nick then used Kemal's base code for the NLP method to create a testing framework that would allow the team to test many different models quickly for Milestone 3. He also worked on the report, much of which was also used for milestone 4.
- **Shreyank Patel:** For Milestone 1, Shreyank worked on brainstorming ideas for the final project and finding pre-existing datasets that can be used to tackle the problem identified. During Milestone 2, Shreyank worked on adding new features to the existing dataset and preparing the csv file to be used for Milestone 3. Shreyank wrote the base code for the decision tree which was later expanded upon by Racheal Dylewski for Milestone 3. Shreyank also worked on creating the graphs under the datas section to be used for the final project report. Additionally for Milestone 3, Shreyank helped with the report.
- **Racheal Dylewski:** For Milestone 1, Racheal helped brainstorm topics and finalize the decision. For Milestone 2, Racheal helped pick the dataset and finalized the report. During Milestone 3, Racheal implemented grid search for decision tree as well as implemented tweeks to the exisiting code. She also helped with a large portion of report, writing the solution and evaluation sections for the decision tree implementation. She also wrote part of the introductory and data description sections of the report to be used in Milestone 4.