Classifying Toxic Comments

By Carson Loveless & Caden Silberlicht

We wanted to learn how we can help classify toxic comments so they can be filtered out, whether through a mandatory system or if users elect to do so. For this task we wanted to see what algorithms we can use to classify these comments, as well as which ones can do so more efficiently. Furthermore, we wanted to see if we could identify what words are most likely to be found in a toxic comment. When we were searching for a topic to cover in this project, we were fascinated by the concept of an algorithm that can accurately guess the sentiment of a particular sentence. While we attempted to answer these questions, we were challenged in all steps of the way. Some of those tasks included: preprocessing our data, learning how to use libraries to handle most of the process, and finding the best ways to quantify and make sense of the results. After we went through solving all these problems on the way, it was clear that while both of these algorithms perform well with larger amounts of data, overall, one of them performed much better.

We used two algorithms for classification in this assignment: Logistic Regression and Naive Bayes (Multinomial). We chose these algorithms because from what we were able to gather online [2], these two algorithms both fit our goals well and provided us with a new challenge to implement. Both of these algorithms have well-documented functions and models as a part of the scikit-learn Python library, but before we could use these algorithms to train and test data, we had to find data to use for training and testing. For our input data, we had a csv including the comment and the classification of toxic or not toxic. Our output was the trends of these two classifiers testing accuracy over different amounts of data.

What we wanted to answer:

- Whether Naive Bayes or Logistic Regression was more effective for determining if comments are toxic.
- How different amounts of training data changed the level of accuracy of classification.
- What words were most likely to classify a comment as toxic.

What challenges we faced:

- Pre-processing our data.
- Using scikit-learn to implement a Naive Bayes and Logistic Regression model
- How to interpret output data.

To tackle these challenges, we first needed to prepare our data for training and testing. We set about this task by first reading the comments and tags from comments, and tokenizing the words in comments and determining if a comment was 0 - not toxic or 1 - toxic. With the tokenized words of the sentence, we used a list of stopwords from a previous assignment, and customized it to further filter out unnecessary data. And after converting that data back to

sentences, we used the *CountVectorizor* class in scikit-learn to convert the sentences and tags into a bag of words for classification. Initially we read the csv into a Pandas *DataFrame*. From there, we incremented through the data. For tokenization we used a Python library called NLTK. This library is focused on processing text, and has many helpful functions for doing so. One of these functions, *regexp_tokenize*, allowed us to tokenize whenever we saw any delimiter we thought necessary to remove what was not necessary. Using the *detokenize* function, we then turn each set of tokens back into a sentence and add each to a list. We then used the *CountVectorizor* class to transform the list of comments into a dataframe bag of words, and once we used the *toarray* function, our data was now in a bag of words to be used with our classifiers.

At this point, we had to learn how to use the *LogisticRegression* and *MultinomialNB* classes of scikit-learn in order to train our classifier. For Naive Bayes, we elected to use the Multinomial algorithm. Multinomial Naive Bayes makes use of the Bayes' Theorem, which states that:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

Via Wikipedia: https://en.wikipedia.org/wiki/Bayes%27_theorem

What this formula is used for is finding the conditional probability of an event happening based on many features. We first are finding the frequency of words associated with each possible outcome. For example: the word 'annoying' could have 50 1's, which represent a toxic comment, and 11 0's, which represent a comment that is not toxic. From there we find the total probability of having a toxic or not toxic comment from this data, and we also find the probability of each word being in a toxic comment. For testing any given comment, we will find all of its words for which we have collected data for, and multiply all of those probabilities together to find the probability that a given comment is toxic. The decision is determined simply based upon whether it is below .5 or at .5 and above. Multinomial Naive Bayes would typically go about these calculations for each possible outcome if there are more than 2, however, since we only used one, the algorithm is simpler.

With all of this in mind, when it came to implementing the Naive Bayes algorithm, we used the *MultinomialNB* class from scikit-learn. The process was very simple, it takes one line only, which was `nb = MultinomialNB().fit(X,Y)` where X is the bag of words and Y is the row of all the tags to represent whether the comment was toxic or not from a Pandas DataFrame we created. After that, we passed test data in the form of a bag of words into the *predict* function and then used the *accuracy_score* function to compare the models predictions with the actual classifications of the test data.

Logistic regression is a popular choice for binary classification. The output is always between 0 or 1. The closer the value is to either 0 or 1, the more likely the given feature fits into that respective class. For our project, this would mean the closer the output is to 1, the higher the

probability that the comment is toxic. Logistic regression uses a sigmoid function to map these outputs. The equation for logistic regression is as follows.

$$y = \frac{e^{(b_0+b_1X)}}{1+e^{(b_0+b_1X)}}$$

From [4]

Where X is the input, y is the prediction, b_0 is the bias, and b_1 is the coefficient attached to the input. b_0 and b_1 are the predicted weights. Implementing this algorithm turned out to be quite simple just like the Naive Bayes. It could also be done in one line using the scikit-learn *LogisticRegression.* `lr = LogisticRegression().fit(X,Y).` Where X is the bag of words and Y is the training labels. After creating the Logistic Regression object, we used the *predict()* function to get the predictions, and then used *accuracy_score* to compute the accuracy of our predictions.

We knew from the beginning we planned to use the NLTK algorithm to tokenize our sentences, but when we used the basic *tokenize* function, we found a lot of inconsistencies in our data and a lot of very strange tokens. This is where *regexp_tokenize* became important. We input a large variety of delimiters to tokenize by in a very quick and simple way. We included all numbers as we felt they were not helpful for classification, all \<char> values, and all symbols present on the keyboard, and finally white space characters. After that, we used a simple iterative function to go through our text file for stopwords. When we initially started creating our bag of words, we elected to use Python's built-in functions to do it in a very simple manner. When it came time to test the effectiveness of it, it was evident that our algorithm was highly inefficient. It was at this point we found the *CountVectorizor*, another scikit-learn tool that allowed us to convert sentences into a bag of words. We now had to figure out how to turn the tokens from each sentence back into a sentence with all the stopwords, and that is where *detokenize*, a part of the *TreebankWordDetokenizer* class in NLTK. Finally we had reached the challenge of obtaining our test data. Through our reading, we were already aware of the *train_test_split* function from scikit-learn, but we wanted to use the same test data for all of our tests and this would not allow it. Unfortunately, after trying many different options, and running into problem after problem, we ended up using *train_test_split* to turn a portion of our input data into test data.

Here is some pseudocode to give a quick rundown of our workflow:

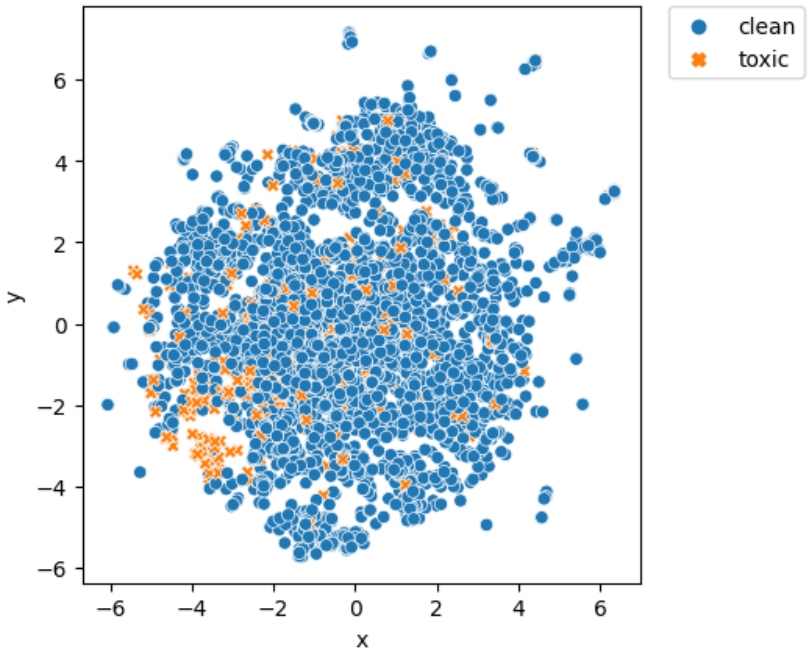
```
for each sample size(tiny,small,medium,big):
     Sample = Read csv for sample size
     Tokenize each comment
     Removes words on stoplist
     Detokenize each comment
     Pass comments into CountVectorizor
     Transform countvector to bag of words
     Obtain tags
```

```
Split data into training and test data
For Naive Bayes and Logistic Regression classifiers:
     Train classifier
     Predict test results
     Compare predictions to find accuracy
```

The data we used comes from Kaggle's *Toxic Comment Classification Challenge* [1] (Disclaimer: the dataset for this challenge contains text that may be considered profane, vulgar, or offensive). The data consisted of user id's, their comment, and whether the comment met the criteria for 5 different forms of toxicity. We were not as concerned with the category that this fell under, but rather whether a comment was "toxic" or not. In this dataset, the different descriptions for toxicity included: toxic, severe_toxic, obscene, threat, insult, or identity_hate. We chose to simply encompass all of these categories as a single tag that described the data as toxic or not toxic. We considered the possibility of taking advantage of the Multinomial Naive Bayes algorithm's strength in categorizing a comment based on what type of toxic trait it falls under, however given the way the data was given to us, the scores being 0 or 1 for each category and having multiple of these tags possible at once, it did not make sense to attempt to find the category that was predicted as we would not have any results to compare those predictions against. Here is a small sample of a line given from the .csv file for our data:

id	comment_text	toxic	severe_toxic	obscene	threat	insult	identity_hate
0000997932d777bf	Explanation Why the edits made under my username Hardcore Metallica Fan were reverted? They weren't vandalisms, just closure on some GAs after I voted at New York Dolls FAC. And please don't remove the template from the talk pa	0	0	0	0	0	0

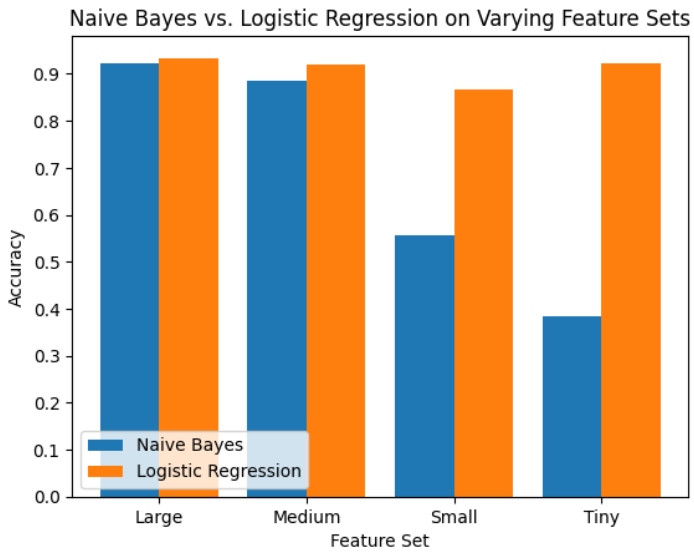
Once we had finished our training and testing, we had to decide what data we wanted to show. The first, and simplest of these, was to find the testing accuracy of our classifiers, and see how this accuracy changed depending on how much training data we provided. We made various smaller, varying sizes of files from the original data (we were unable to process the whole file as there were over 500,000 lines of text to handle), and with this data we compared the results of accuracy for our training data. By using varying data sizes for each algorithm ('tiny' - 42 entries, 'small' - 147 entries, 'medium' - 875 entries, and 'large' - 7238 entries), we were able to clearly notice how these different algorithms perform with different sample sizes. Note that for all the previous listed amounts of entries, 30% of them were used for testing. It is important when building a classifier, or any machine learning algorithm, that we find one that performs the best for our sample size. So it was helpful to get an idea for the scalability of our program. The following is a t-SNE plot of the largest feature set we used.



T-SNE plot of our large feature set.

As you can see from the scatter plot, the majority of the comments in our data were clean. There is only a relatively small group of "toxic" words that closely relate to each other, while others are shockingly similar to words in "clean" comments.

When we began this assignment, we struggled to find a model that would help us classify the Wikipedia comments. Finding the algorithms that we wanted to investigate further took some time, but the two we chose both gave largely different results. When it came to a large sample size, both of these algorithms started to level out at about 91 percent test accuracy. Most interesting of all, however, was the difference in effectiveness of these algorithms as the sample size increased. We have provided a graph made using the *matplotlib* library below:



Naïve Bayes Probability of a Word Appearing in a Comment for Each Type of Comment		
Ranking of Likelihood	Toxic Comment	Non-toxic Comment
1	f**k	article
2	page	page
3	f*****g	talk
4	people	edit
5	s**t	time
6	life	articles
7	stupid	people
8	stop	make
9	a**	good
10	article	made

What is immediately apparent is that Logistic Regression had very similar results regardless of the sample size that was used for training and testing. On the other hand, with Naive Bayes, there is a clear difference in its performance as we increase the sample size. We believe that Naive Bayes has a significant improvement in performance due to the fact that as it trains, earlier on, when there is far less data, small amounts of data can vastly change how it outputs. Along with that, it is easy to imagine that as the classifier's vocabulary increases, its ability to make these predictions improves.

In the table above, the classification for some of the words associated with toxic comments make sense, however many are shared between both categories. The reason is that these are the words that generally appear often in comments. In the end, we were able to determine that Logistic Regression performs better consistently. Where we were challenged in this project was scaling our algorithm to handle much larger data, as well as adjusting our program to be more efficient when the runtimes were extremely slow. While we were able to improve our runtimes, we failed to scale our trainer to handle much larger amounts of information at one time. While we had hoped to answer the question about what words were most likely to make a comment toxic or not toxic, we did not succeed. Regardless of that, we still wanted to find out what words are most common in toxic and non toxic comments, so we used code from another source [3] to give us results. We failed to find the words ourselves due to our lack of experience with scikit-learn. In this short amount of time, it was difficult to learn the intricacies of the library, and we gained only a fundamental understanding of the library.

This project gave us great insight into the world of text classification. The problems we encountered led us to take an entirely different approach than we had originally hoped. Our collective knowledge of data pre-processing was greatly improved by this task. This project allowed us to learn the process of the Naive Bayes and Logistic Regression model, as well as various methods of visualizing data that previously we had never done before. If we had the chance to do it all over again, I believe we would process our data differently, so as to find a way to make this a multi-class classifier, rather than a binary classifier. All in all, this project was a phenomenal learning experience, and we are excited to be able to use this new-found knowledge of machine learning in our future endeavors.

References:

[1] - https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge
[2] - https://monkeylearn.com/blog/classification-algorithms/
[3] - https://stackoverflow.com/questions/50526898/how-to-get-feature-importance-in-naive-bayes
[4] - https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-logistic-regression/
https://www.upgrad.com/blog/multinomial-naive-bayes-explained/
https://towardsdatascience.com/multinomial-na%C3%AFve-bayes-for-documents-classification-and-natural-language-processing-nlp-e08cc848ce6

https://datatofish.com/scatter-line-bar-charts-using-matplotlib/
https://seaborn.pydata.org/generated/seaborn.scatterplot.html
https://danielmuellerkomorowska.com/2021/01/05/introduction-to-t-sne-in-python-with-scikit-learn/
https://thatascience.com/learn-machine-learning/bag-of-words/
https://www.mygreatlearning.com/blog/multinomial-naive-bayes-explained/