STA 141C: Toxic Comment Classification

Yiting Kuang | April Vang | Haolin Li

1. Introduction

Cyber-bully has always been a large issue in the world. Negative comments are often seen on the internet in different social media platforms and they can be categorized as “toxic”, “severe toxic”, “obscene”, “threat”, “insult” and “identity hate”. Since it is impossible to ask everyone to not post negative comments on the internet, what we can do is to build a model that is capable of detecting different types of toxicity of comments that prevents the users from posting negative comments to the public.

Despite the fact that there is already a Perspective model that tries to detect toxicity, the current model still makes errors. This is our motivation for this project, with doing the Kaggle Challenge on Toxic Comment Classification, we want to build a model to classify the toxicity of the comments as accurate as possible.

We reached a highest accuracy of 0.987130 with our KNN algorithm. For this, the non-processed data, Wiki Fasttext, Twitter GloVe data were used. Long short-term memory and gated recurrent unit filtering were also accounted for in the implementation of the algorithm.

2. Methodology

2.1 Dataset

The dataset we are working with contains a large number of Wikipedia comments which have been labeled by human raters about each comment’s toxicity. There are two sets of datasets included, the training set and the test set. There are 159571 comments in the training set while there are 153164 comments in the test set. The following plot shows the label frequencies in the training set. We can see that “Toxic” class is classified the most because it is a general category in toxicity.

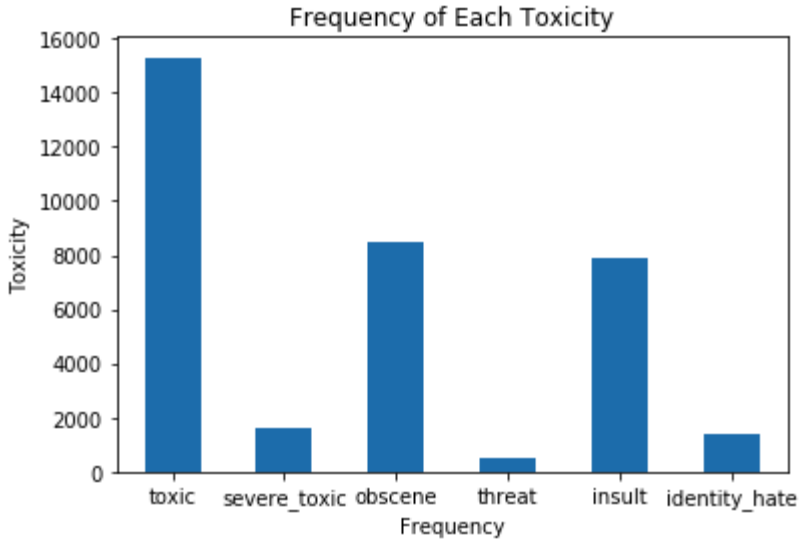

Figure 1: Label Frequencies in Training Set

Additional Datasets (for KNN Algorithm):

- test_processed.csv (153,000 X 2), trian_processed.csv (160,000 X 7) where the data comments were processed and cleaned for easier reading.
- wiki-news-300d-1M.vec.zip: 1 million-word vectors trained on Wikipedia 2017, UMBC web-based corpus and statmt.org news dataset (16B tokens). Used to train and test.
- glove.twitter.27B.zip: Twitter (2B tweets, 27B tokens, 1.2M vocab, uncased, 200d vectors) containing the word vectors obtained from the corpora. Used to train and test.

2.2 Algorithms/Classifiers

From the Scikit-learn library, we selected two algorithms: **LogisticRegression** and **RandomForestClassifier**, as we have learned about them in class. As mentioned in the introduction, our goal is to build a model to predict the probability of each class of each comment. The very first model we build was using logistic regression. The idea of using logistic regression is to implement the model for each output column with X_train and Y_train, then make a prediction for X_test. Finally store the results into Y_test with creating class_names within the test dataset. The idea of random forest is similar to logistic regression. As random forest algorithm gives a discriminating power for each factor that contributes to a better prediction, we observed that the model random forest is doing not better than the logistic model, logistic model has an accuracy of 0.97609 while random forest has 0.90573, both in the cases using stemming. Additionally, random forest has a longer run time than logistic regression.

RNN Model: When working with the RNN method, the hard part is getting the dataset structured in an organized way since the comments do not have specific targets, they have time series (the sentences is read in one direction) and we are trying to match it to its correct category. For this, using RNN requires pre-processing to get the better results. We have found pre-processed data to be used and hypothesize that apply our RNN method will yield more accurate classification compared to when using the non-processed data. We tokenized the words and then use the Fasttext and GloVe and then rsplit was applied since it was space separated already.

3. Implementation

3.1 Library

The main library we are using in this project is Scikit-learn, as we want to implement the algorithms and classifiers within the library. The Scikit-learn project provides a wide selection of classification, regression and clustering algorithms including support vector machines, random forests and more. The robustness of Scikit-learn being easy to use, powerful and flexible for research projects makes it perfect for us to use it for our project and other machine learning projects.

In order to fully maximize the use of Scikit-learn in our project, we utilized some of its robust features. First, we used the TfidfVectorizer and CountVectorizer for the preprocessing and feature extraction, as we have learned these two extractions in lecture and they are the best implementations we have known of. Then, we imported the LogisticRegression and the RandomForestClassifier classes to build our models. Finally, we chose to use make_union and criss_val_score features to combine our vectors and validate our training data.

3.2 Preprocessing (Stemming)

One of the most popular methods when dealing with preprocessing text data is stemming. For grammatical reasons, text comments are going to be used in different forms of a word, such as get, got,

and getting. Also, there are families of related words with similar meanings, such as democracy and democratic. Therefore, our goal is to use stemming to reduce the form of a word to its common base form such as colors to color, and different to differ. In other words, stemming removes morphological affixes from words, leaving only the word stem.

We decided to use one of the stemmers we are familiar with, that we learned in lecture, EnglishStemmer. The library we used here is the NLTK library. Despite there are non-English comments in the dataset, we observed that most of them are English comments so we figured stemming words to English stem forms would be our choice. We believed that reducing words to their stem forms would be valuable to our model. As a result, we observed that using EnglishStemmer improved our accuracy from 0.97375 to 0.97609 in our Logistic Regression models.

3.3 Extraction: CountVectorizer and TfidfVectorizer

We used the stemming before applying the CountVectorizer and the TfidfVectorizer. Both vectorizers implement both tokenization and occurrence counting in a single class. They count the number of times a token shows up in the document and uses this value as its weight. By using two different vectorizers, we observed that TfidfVectorizer improves our accuracy from to 0.97609, in the case of using stemming. Following are some parameters of CountVectorizer and TfidfVectorizer we set that improve the result.

1. **stop_words**: Since vectorizers just count the occurrences of each word in its vocabulary, extremely common words like "a," 'the', 'and' will become very important features since they add little meaning to the text. We try to improve our model by not taking those words into account. We set the parameter stop_words = 'english,'a built-in list, which is a list of words we do not want to use as features.
2. **ngram_range**: An n-gram is a string of n words in a row. We use the parameter ngram_range=(a,b), where a is the minimum and b is the maximum size of ngrams we want to include in our features. We use the default ngram_range=(1,1) because by intuition, many toxic words such as "f*ck,", "n*gger," and "b*tch" are just 1-word long.
1. **max_features:** Both vectorizers choose the words that occur most frequently to be in the vocabulary and drop everything else. This parameter is rather subjective. We try different values and eventually set max_features = 30000 and 10000.

3.4 KNN Method and Keras

A KNN standard model was used with word embeddings and implemented using Keras. Specifically, the key steps in the model include:

- Apply the Fasttext (wiki-news-300d) and Twitter (GloVe) embeddings in concatenation. For words without word vectors, replace it with the vector word for "something".
- Use the *SpatialDropout1D*, and apply the filters of *Long Short-Term Memory* and *Gated Recurrent Unit*
- Account for unique and all-cap words.
- Set output dense layer.

We've also converted the text into ASCII for filtering and easier reading. Key parameters used were ~ Batch size: 32, Epochs: 80 (with early exit at epoch 3), Sequence Length: 500, num_folds = 10.

4. Result and Thoughts

4.1 Logistic Regression

As mentioned in the introduction, our ultimate goal of this project is to classify each comment to each class with the probability that predict the different classes on the test set. To thoroughly observe the quality of our use of models and classifiers, we calculated the prediction of the test set. The accuracy of the test set is the prediction of the comments within the test set. We tried out different combinations of models and classifiers, in the hope of getting the best accuracy and have the best model. We generated 5 different combinations with logistic regressions, random forest algorithms and different stemmings and extractions. The best combination was the logistic regression classifier with TfidfVectorizer and stemming in both words and character. What we learned from this is that logistic regression model is the best model for this classification problem. The reason that the best model with the highest accuracy is because its use in TfidfVectorizer instead of CountVectorizer. Another important reason is that it stemmed not only the word but also each character in the comments. The benefit of using 'char' analyzer is that people often confuse negative words with additional characters, using 'char' analyzer can potentially detect them, and 'char' analyzer helps in detecting foreign languages in the test set as well.

Table: Accuracy of Different Combinations

Combination	Accuracy of Test
Logistic Regression with TFIDF & English Stemmer	0.97609
Logistic Regression in TFIDF without Stemming	0.97375
Logistic Regression with CountVectorizer & Stemming	0.94872
Logistic Regression with TFIDF in words and character	**0.97624**
Random Forest with TFIDF & English Stemmer	0.90573

4.2 KNN Algorithm

Table: KNN Accuracy Rate Vs. Time on Original and Processed Data

	Original (Non-Processed) Test Data		Processed Test Data	
Epoch	Run Time (s)	Accuracy Rate	Run Time (s)	Accuracy Rate
1	6583	0.985104	5397	0.982455
2	8968	0.986704	5262	0.985030
3	7990	**0.987130**	5503	0.986515

Surprisingly, applying the KNN algorithm to the original data performs better for accurately classifying comments compared to when using the processed data. For this we reject our hypothesis. It is important to note however that the run time for the non-processed data took longer compared to the processed data. We see a trade-off between accuracy and time. Additionally, we were trying to implement Alexander Burmistrov's 0.9872 method (3rd Place on Kaggle). One of the main differences is how we used Wiki Fasttext (wiki-news-300d, 1-Million-word vector) instead of the Crawl Fasttext (crawl-300d, 2-Million-word vector). This led to an accuracy difference of 0.0001 to four decimal places. We've attempted to use Crawl, however, using it takes a considerable amount of runtime; for better accuracy, we can try Crawl with the non-processed data, or for better runtime, we can try Wiki and the processed data when using our KNN algorithm.