**Hate Speech Identification**

Authors: Joseph Seiler, Jonathan Touma

**Abstract**

The paper reviews the making of our hate speech identifying tool and the results we found through testing. Large quantities of hateful and harassing messages are posted online daily. These types of message only serve to harm others and to halt any meaningful discussion. We have undergone the task of creating a tool that is able to take in text messages and output a label for the message. The label indicates whether or not the message is believed to contain some form of hateful or racist ideas. The end goal of this project is to have an accurate model for predicting such a classification, and to be able to determine different types of discrimination and hate speech.

**Introduction**

According to a survey done by the Anti-Defamation League, thirty-seven percent of adults have experienced online harassment and threats (Council on Foreign Relations, Laub). Another study shows that forty-two percent of adolescents have “experienced at least one direct (individual) discriminatory incident” (American Psychological Association, Tynes). This problem affects a substantial portion of the population and all age groups. Many companies that run these types of online boards might want a way to find these types of hateful messages and take some sort of action. Of course, it would be exceedingly impractical to do this by hand. The man-hours and resources required to comb through millions of messages per day is much too high. This is the reason we decided on using data mining methods and computing power to help solve the problem. We want a way to have a computer read and classify messages as hateful in real-time. The posts that are classified as such could then be flagged for further review by admins or moderators. Our tool would narrow the space of messages that need human intervention and thus save companies a considerable amount of time and money.

**Solution**

Our solution was to use publicly available data to build models that could perform classifications on the hatefulness or racism in a block of text. The data we are building models from is a collection of tweets off of Twitter.com. To implement our solution, we decided on Python for our programming language for the extensive data mining support. We want to achieve the highest possible performance, so we tested multiple algorithms for classification and compared the results. Python’s SkLearn library provided us with the implementations of the algorithms.

**Experiments**

**Data**

The data consisted of a dataset we got from Kaggle.com titled “Twitter Sentiment Analysis”. The dataset thirty-two thousand records, each record consisting of a tweet’s contents and its classification as hateful or not hateful. The file size is three megabytes. Approximately ninety percent of the data is positively classified, meaning the dataset is unbalanced and we need to use F1-score to measure performance. Through manual inspection, we found that the dataset is not classified perfectly, which is to be expected with such a large dataset classified by humans. For example, the tweet “retweet if you agree!” was classified as hateful. This should not affect our modeling and testing since we split the dataset in order to evaluate performance. However, it is worth noting that the model could be overfit to this dataset and its errors and would not perform well in a real-world environment.

**Experimental Setup**

**Joey you write about the cleaning you did on the data you beast 8==D**

Once the data was cleaned up, we were ready to build a model for classification. It is difficult to work with data that is consistent of strings. Ideally, we need a numerical representation of the data which would make it possible to work with the libraries that implement the algorithms we need. From a list of strings, each element containing the contents of a tweet, we use a CountVectorizer to transform the tweets. Each unique word in the strings is added to the dictionary and creates another column in the matrix. The tweets are transformed so that each column of an individual tweet is equal to the number of times the record mentioned the word aligned to that column.

It was then time to fix the weights for each record. A TfidfTransformer would allow us to achieve this. When counting the number of times a word appears in a document, the longer a document is, the more likely it is to have higher counts for more words. This means that long documents will be deemed more similar to other long documents just due to having high counts. Similarly for small documents. Using TFIDF weights for the terms means that this effect is circumvented, and we can get weights that better describe the definition of a document. With this model for describing tweets, we could then begin classification.

**Experimental Results and Analysis**

To classify tweets, we look at four algorithms. Decision tree, neural network, naïve Bayes, and K-nearest-neighbor. The performance metric we used for checking performance was the F1-score due to the imbalanced dataset. We test the algorithms by splitting the dataset into a training and test set. We also stratify the split so that both the training and test set contain the same proportions of each classification. The model of the dataset consisted of the thirty-two thousand records and the dictionary size of each unique word was thirty-eight thousand. This matrix was absolutely huge, containing over a billion elements. A matrix of this size was extremely difficult to work with. Normally, this size matrix would not be able to be stored because it exceeds the maximum size of Python arrays. However, the weights were stored in a compressed sparse row matrix. While in this form, the data could be played with. The problem arose when using the naïve Bayes and K-nearest-neighbor algorithms in SkLearn. These algorithms began by converting our matrix into an array. This would immediately crash the program because there were too many elements in the array or memory errors would be thrown. Thus, we needed a way to decrease the number of attributes in the dictionary.

We needed a way to reduce the number of unique words so that the dictionary’s size could be reduced. The first thing we did was to make the cleaning process stricter. Initially, words with apostrophes were kept as unique words so that some of the meaning that comes with those words could be kept. We later decided to discard apostrophes which reduced the size of the dictionary slightly. This reduction in size was not enough to allow the two algorithms to run without crashing. The next idea we had was to use a spelling corrector. Every spelling mistake would count as a unique word in the dictionary, and there are many ways to spell a word incorrectly. Ideally, by correcting a word’s spelling first, and then adding it to the dictionary, the number of unique words would be reduced. Two correctors were tried. TextBlob spelling corrector, and a spelling corrector based off Peter Norvig’s spell checker. Both drastically increased the runtime. Each word took between one to five seconds to correct. With thirty thousand tweets and up to one-hundred fifty words in each, the runtime would become impractical very quickly.

The next thing we tried was principle component analysis or PCA. However, this had the same problem as naïve Bayes and K-nearest-neighbor where it needed to transform the compressed matrix to an array and would crash the program. Lastly, we tried feature selection. Feature selection can be used to remove attributes that do not meet a certain variance threshold. Features with low variances are present in more records meaning they do not contribute greatly to the differences between records. SkLearn’s VarianceThreshold worked well with the compressed matrix, meaning it would not crash the program when attempting dimensionality reduction. This allowed us to remove as many attributes as we want, while trying to find a balance between low number of attributes and high performance. This can be seen in the table below.

Algorithms without feature selection – 37,000 features

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **F1-Score** | **Runtime (seconds)** |
| Neural Network | 0.70 | 1278 |
| Decision Tree | 0.58 | 10.9 |
| Naïve Bayes | CRASH | CRASH |
| K-Nearest-Neighbors | CRASH | CRASH |

Algorithms with feature selection – 7,000 features

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **F1-Score** | **Runtime (seconds)** |
| Neural Network | 0.68 | 346 |
| Decision Tree | 0.56 | 8.25 |
| Naïve Bayes | CRASH | CRASH |
| K-Nearest-Neighbors | CRASH | CRASH |

The tables above show the effects of feature selection. The model went from thirty-seven thousand attributes to just seven thousand. The variance threshold that achieved this was 0.00002. We settled on this threshold because it reduced the attributes and runtime by a significant amount while having only a small reduction in performance, giving us the best of both worlds. However, we were still seeing memory errors when running naïve Bayes and K-nearest-neighbors. The reason for this is possibly due to the fact that these algorithms are more memory intensive than the others since they are unable to run with compressed matrices. To get these algorithms to run, on top of feature selection, we subsampled the matrix by splicing out and using only the first five thousand records. This smaller matrix was finally able to be turned into an array without using up all of the memory in our machines. Results can be seen in the table below.

Algorithms with feature selection and subsampled matrix

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **F1-Score** | **Runtime (seconds)** |
| Neural Network | 0.68 | 346 |
| Decision Tree | 0.56 | 8.25 |
| Naïve Bayes | 0.26 | 1.06 |
| K-Nearest-Neighbors | 0.21 | 0.17 |

As we can see from above, all the work put into getting these two algorithms to work was for naught. They performed much worse than the other two algorithms while also requiring much more memory to run. The lower performance is most likely attributed to the fact that there were fewer records to compare with since the matrix was subsampled. At least we were thorough.

**Conclusion**

We have found through experimentation the best algorithm for the purposes of classification. The neural network was best suited for this task and graded highest on performance. However, it was also the costliest, in terms of runtime. To achieve our goal of moderating online forums, the neural network implementation could be used to flag posts as hateful. It would need to be tested in a real-world environment to see if it is only working well on the dataset it was built from or if the model translates well to real forums. It is possible that, due to the slow run time, the neural network would not be able to keep up with the huge volume of incoming text in large-scale websites. Further improvements to the model and more machine power would be needed to keep up.

**References**

Laub, Z. (2019, April 11). Hate Speech on Social Media: Global Comparisons. Retrieved May 12, 2019, from <https://www.cfr.org/backgrounder/hate-speech-social-media-global-comparisons>

Tynes, B. M. (2015, December). Online racial discrimination: A growing problem for adolescents. Retrieved May 12, 2019, from <https://www.apa.org/science/about/psa/2015/12/online-racial-discrimination>