Watch Your Mouth, Virginia!

Kyle Cheng (kwc9ap), Iain Muir (iam9ez), Boris Topalov (bnt4yb)

December 9 2021

1 Dataset

`https://www.kaggle.com/c/detecting-insults-in-social-commentary/data?select=train.csv`

2 Abstract

This project analyzes offensive language on the internet through the lens of Machine Learning. We use Natural Language Processing (NLP) models to categorize text commentary taken from social media websites into two categories: offensive language and non-offensive language. This paper is divided into 6 sections: Abstract, Introduction/Motivation, Methods/Experiments, Results, Conclusion, and References. We first introduce the data set and set up why we are seeking to detect offensive language on the internet. Next, we detail the machine learning algorithms we used and share the results of the models' performance, including statistics on precision, recall, and F1 scores. Finally, we conclude with our key takeaways, potential applications of this project, and project references.

3 Introduction and Motivation

Cyberbullying and exposure to offensive language are issues for internet users of all ages. These issues can have negative impacts on mental health, and in the case of children, psychological development. Software tools that combat offensive language on the internet exist, however, many of them focus on post-publication censorship. Our proposal takes a slightly different approach and aims to detect potentially offensive language before it is posted to the internet and made public. Often, users who post harmful or offensive language online do not think about the consequences of their actions. Our goal is to create a machine learning model that detects potentially offensive language, and provides recommendations on how to express a similar point using less explicit language. This type of model would not only reduce harmful language, but also be an educational tool on how to ethically engage with others over the internet.

Offensive language detection has become one of the most common uses of natural language processing (NLP). While originally executed through statistical inference, modern day NLP models generally utilize deep neural networks. A related paper (Susanty et al.) submitted to IEEE 2019 trained an Artificial Neural Network (ANN) on a sigmoid activation function. The ANN was able to classify tweets from a testing dataset as "offensive", or "non-offensive", at 96.8% accuracy.

4 Methods and Experiments

In order to classify social commentary as an insult or not, we utilized Natural Language Processing and tested multiple Classification models. We tested a Multinomial Naive-Bayes Classifier as our baseline, a Linear Support Vector Machine (scikit-learn's Stochastic Gradient Descent Classifier), and two separate Keras Neural Networks. Our neural network architecture for both models contained a series of convolution layers, dropout layers, pooling layers, and simple dense layers. The model also included an initial embedding layer. Additionally, our neural networks had to be tailored to our NLP problem, but also had to be simple enough to avoid overfitting. For all four of our models, we evaluated performance based on validation or test accuracy.

For our data preprocessing, we used two separate pipelines. The pipeline for our Naive-Bayes and SVM classifiers comprised of a StemmedCountVectorizer and a Term Frequency-Inverse Document Frequency (tf-idf) Transformer. The StemmedCountVectorizer converts words to their root forms (for example, running-¿run) and transforms a given text into a vector on the basis of the frequency of each word that occurs in the entire text. We created a custom class for this step in the pipeline, which inherits from scikit-learn's CountVectorizer yet also incorporates Natural Language Toolkit's Snowball Stemmer. The TF-IDF Transformer associates each word in a document with a number that represents how relevant each word is in that document. The higher the number, the relevant the word. Each of these preprocessing steps are able to be replicated using scikit-learn Pipelines. We did not need to implement a train-test split because the source data was already separated into two separate files. However, we did create a validation set using 20% of the train set. Our neural network pipeline comprised of a text cleaner, a tokenizer, and a sequence padder. The text cleaner removes stop words, punctuation, non-breaking spaces, and urls. The tokenizer splits text into sequences of tokens or words. The sequence padder pads data examples so every example will fit in neural network input layers. We decided to use two separate pipelines, one for our two scikit-learn models and another for our two Keras models, because we wanted to align the different libraries' preprocessing methods with their respective models. We attempted to incorporate both pipelines at various stages of our project but this resulted in worse performance across the board.

After preprocessing and initial model testing, we began the hyperparameter tuning phase. Specifically, for the Naive Bayes Classifier, we used a Five-Fold Cross-Validated Grid Search to determine the optimal hyperparameters for the model. The only hyperparameter we tweaked was the learning rate (alpha), from a range of 1 to 1e-4. After tuning, our final learning rate used was 0.1. For the SVM, we once again used a Five-Fold Cross-Validated Grid Search, yet tweaked the learning rate (alpha), penalty, and loss; alpha range from 0.006 to 0.00006, penalty was either l2, l1, or elasticnet, and loss was either hinge or squared hinge. Our final SVM hyperparameters consisted of a Learning Rate of 0.0006, with the best loss metric determined to be Squared Hinge and the best Penalty Metric determined to be Elastic Net.

Tuning the hyperparameters for our NLP models proved to be more complicated. Due to the skewed nature of the dataset, and the inherent complexity of neural networks, our model was classifying every data point as a negative and struggled to identify true positives in the dataset, regardless of hyperparameters used. This resulted in the model's performance metrics remaining relatively stagnant with successive epochs. We tested this hyperparameter tuning process by using different tuning objectives, including accuracy, precision, and recall, but we were unable to yield strong results. As a result, we are not confident with the results of our neural network results. After completing the tuning process, the final hyperparameters for our first neural network (NN I) consisted of 128 filters in the Convolutional Layers, 32 units in the Dense Layers, and a Learning Rate of 0.001. For our second neural network (NN II), the final hyperparameters consisted of 192 units in the Dense Layers and a Learning Rate of 0.01.

Experimentation in this project revolved around hyperparameter tuning and data cleaning. We tested the performance of our models with various learning rates and optimizers, including the Stochastic Gradient Descent optimizer, Adam optimizer, and Nadam optimizer. We found that the optimizer we used had a significant impact on performance, with Adam and Nadam resulting in the strongest model performance. We also tested various data cleaning methods. In our final Keras models, we implemented functions that remove urls, remove whitespace, and remove stopwords from each data point. While iterating on these text cleaning functions, we tested model performance to ensure that cleaning text improved performance.

5 Results

Given that our project is a binary classification problem, we are evaluating our machine learning algorithms using a standard confusion matrix, ROC curve, and scikit-learn's classification report function. Specifically, we have paid close attention to the accuracy of the model and the precision/recall trade-off. Our baseline model, the Multinomial Naive-Bayes Classifier, had varying levels of success. Despite accuracy and precision (weighted average) of 81.0%, the model's recall was a terrible 12.0% for the positive case. At its face, the model performs well, and when the classifier predicts positive it is usually correct (low amount of false positives). That being said, the model does not predict positive nearly as much as it should (low amount of true positives), resulting in a low recall score. Our initial SVM model performed much better on its first attempt, with an accuracy score of 82.0% and weighted average precision and recall scores of 81.0% and 82.0%, respectively. Most notably, the recall for the positive class jumped to 55.0%. After tuning the SVM, the model slightly improved with an accuracy of 83.0%, precision of 82.0% and recall of 83.0%. For our Keras models, the first post-tune Neural Network scored 78.2%, 60.9%, and 51.7% for accuracy, precision and recall, respectively, while the second post-tune Neural Network scored 77.5%, 57.7%, and 58.8%.

For a more detailed look at the weighted average performance of our models, see Table 1. Additionally, see Table 2 for a breakdown of performance by the positive and negative class. Lastly, see the following figures for each of the model's Confusion Matrices: Figure 1 for the Naive Bayes Classifier, Figure 2 for the Support Vector Machine, Figure 3 for the first Neural Network, and Figure 4 for the second Neural Network.

Table 1: Weighted Average Performance Metrics

Classifier	**Accuracy**	**Precision**	**Recall**	**F1-Score**
Pre-Tune MultinomialNB	77.0%	80.0%	77.0%	70.0%
Post-Tune MultinomialNB	81.0%	81.0%	81.0%	79.0%
Pre-Tune SVM	82.0%	81.0%	82.0%	81.0%
Post-Tune SVM	83.0%	82.0%	83.0%	81.0%
Pre-Tune Neural Net I	75.0%	76.0%	75.0%	76.0%
Post-Tune Neural Net I	78.2%	60.9%	51.7%	55.6%
Pre-Tune Neural Net II	82.0%	81.0%	82.0%	80.0%
Post-Tune Neural Net II	77.5%	57.7%	58.8%	58.2%

Table 2: Positive vs. Negative Class Performance

	Negative (0)		Positive (1)	
Classifier	**Precision**	**Recall**	**Precision**	**Recall**
Pre-Tune MultinomialNB	76.0%	99.0%	88.0%	12.0%
Post-Tune MultinomialNB	82.0%	96.0%	77.0%	40.0%
Pre-Tune SVM	85.0%	90.0%	66.0%	55.0%
Post-Tune SVM	84.0%	95.0%	77.0%	49.0%
Neural Net I	84.0%	89.0%	63.0%	51.0%
Neural Net II	85.0%	92.0%	71.0%	54.0%

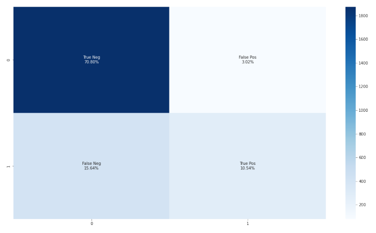

Figure 1: Naive Bayes Classifier Confusion Matrix

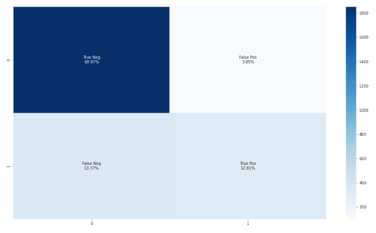

Figure 2: SVM Confusion Matrix

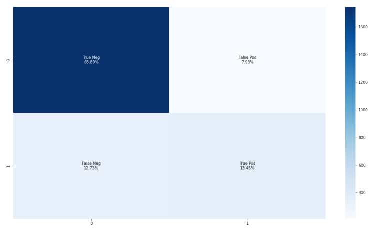

Figure 3: Neural Net I Confusion Matrix

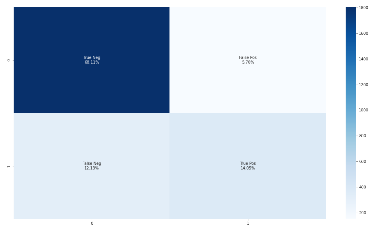

Figure 4: Neural Net II Confusion Matrix

6 Conclusion

After running our data through multiple models, we found that our SVM model performed the best with an accuracy of 83%. It was especially strong at correctly recognizing the positive case, with a positive-case precision and recall of . We think this is important since it's good to be a bit more liberal with our model's classification.

From this project, our team learned valuable lessons. Firstly, how important the proportionate representation and quantity of data can be. The dataset we used ended up not being optimal, and many of the competitors in this dataset's original Kaggle competition shared similar thoughts. The classes were heavily imbalanced (about 70% negative and 30% positive), and we did not have enough data to prevent complicated models like neural nets from simply memorizing the training data. As such, overfitting was a huge concern. We also learned the impact of hyperparameter tuning. As we tuned hyperparameters and continued to tailor them to our models, we saw small, but consistent increases in performance. Overall, we are proud of our work and believe that our project can have great impact in the real world by preventing cyberbullying and making the internet a safer place.

While we consider our project a success, we acknowledge several limitations and biases in our methodology. One big limitation is that our classification metrics are not yet at the level needed for reliable use in a production environment. Our best model, the SVM, had a precision of 77% on the positive case, meaning 23% of examples classified as insults, weren't actually insults. If users consistently experience misclassification of their comments, they will be much less likely to trust our system, even on comments correctly classified. Additionally, our models have a high level of bias, stemming from the unrepresentative nature of our dataset. Our dataset was collected from online message boards, of which can often have different online communities and social norms. The nature of which message boards sampled could have a huge effect on the ability of our model to generalize to unsampled message boards. Finally, the dataset was simply was not large enough in size to properly represent the complexities of human language and insults. In future works, we could explore using weighted ensemble classifiers, which recent research has shown to be very promising (Upadhyay et al., 2020).

7 References

- Keras Hyperparameter Tuning: `https://www.tensorflow.org/tutorials/keras/keras_tuner`
- Natural Language Toolkit: `https://www.nltk.org/data.html`
- NLP Pipeline: `https://medium.com/@eiki1212/natural-language-processing-naive-bayes-class`
- StemmedCountVectorizer: `https://towardsdatascience.com/machine-learning-nlp-text-classif`
- Susanty et al: `https://ieeexplore.ieee.org/abstract/document/8834452`
- Upadhyay et al: `https://ieeexplore.ieee.org/document/9185641`
- Text Cleaning: `https://medium.com/geekculture/nlp-with-tensorflow-keras-explanation-and-`

8 Contribution

- Kyle Cheng (kwc9ap): Completed the bulk of the initial coding, aided coding for the Keras Neural Networks, and wrote the Conclusion section.
- Iain Muir (iam9ez): Wrote bulk of the code for the scikit-learn data preprocessing and hyperparameter tuning, wrote Methods and Results section.
- Boris Topalov (bnt4yb): Wrote bulk of the code for the Keras Neural Networks, wrote part of Abstract and Introduction section.