Toxic Comment Classification Challenge

Sameer Ganesh Shinde
Student ID: 19644693
NetID: sgshinde
sgshinde@uci.edu

Pranav Udupa Shankaranarayana
Student ID: 65085145
NetID: pudupash
pudupash@uci.edu

Abstract

This document demonstrates the approaches and methods used for addressing the famous Kaggle competition, "Toxic Comments Classification Challenge". With an effort to solve this problem with the "mean class-AUC" as a performance measure, we implement and analyze Decision Trees, Logistic Regression, LSTM, XGBoost and ensemble techniques. In this report, we present our understanding and learning from the comprehensive empirical analysis and implementation of these multiple approaches.

1 Introduction

The Toxic Comment Challenge by Kaggle provides a focus on building a learner which can effectively classify a comment into six individual labels including toxic, severely toxic, obscene, threat, insult and identity hate. Through techniques of Natural Language Processing demonstrated below, this problem can be expressed in the form which can be solved using several Machine Learning techniques. We explore, analyze and contrast these NLP and ML techniques under the evaluation metrics like AUC-ROC to identify an optimal solution to the problem.

2 Data Exploration and Pre-processing

Data exploration and visualization was very important while designing an algorithm to solve this problem. Visualizing how the comment lengths were distributed aided in understanding the requirement of processing power required to train and predict the data. We also plotted, the number of comments with multiple classes tagged, which gave us better idea about the complexity of problem. During our research about data visualization techniques, a tutorial by (Jagan, 2018) was substantial in learning the effective way of visualizing data.

Similarly, plotting of class labels vs number of comments translating into the tag helped us visualize the types of comments with their distribution across multiple classes. More than 90 % comments were tagged as Clean.

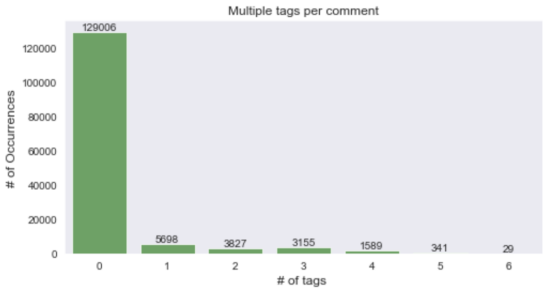

Figure 1: Number of comments tagged as multiple classes

2.1 Data Visualization of raw data

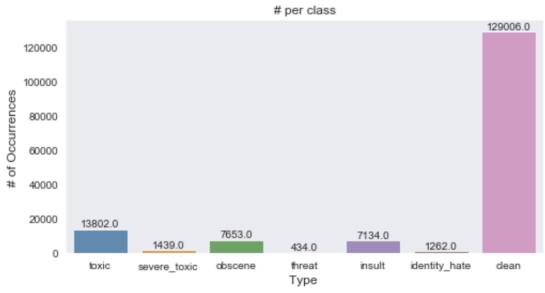

Figure 2: Distribution of comments across all classes

Further data analysis provided more valuable insights about the underlying challenge. A violin plot by seaborn demonstrates the relationship between the length of comments and the possibility of the comment being toxic. This illustrates that length of sentences do not seem to be a significant indicator of toxicity. This information was pivotal

in the data cleaning and pre-processing steps to reduce the length of input during training.

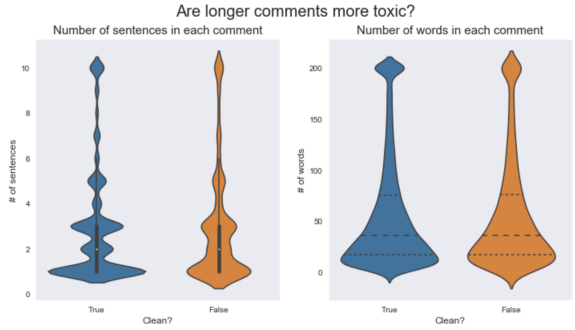

Figure 3: Violin plot of comment length vs Toxic

Another important aspect of data was unravelled with the help of combination matrix revealing relationship of cooccurences of classes.

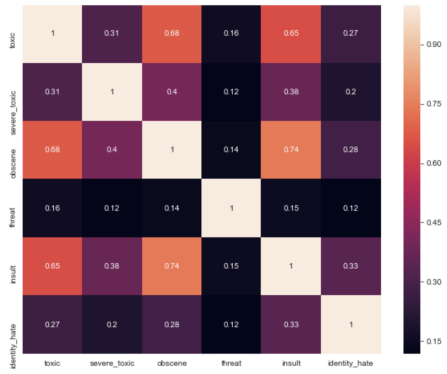

Figure 4: Matrix of Co-occurences

2.2 Data Cleaning

Several aberrant grammatical characteristics in the underlying language seldom provide valuable insights to solve this classification problem. Some of these characteristics include Non-ASCII characters, punctuation, letter casing, numerical information and extremely long words ($>$ 30 letters). Hence, we perform a data cleanup by removing these grammatical characteristics from words and recombining them into a sentence.

2.3 Removing Stopwords

Stopwords such as "the, be, are" provide no value to our classification problem (Jabeen, 2018). Hence, we implement a pre-processing technique that splits sentences into words and re-combines them ignoring all the words that we deem as stopwords. This set of stop words are obtained using the stopwords corpus in the nltk library. The

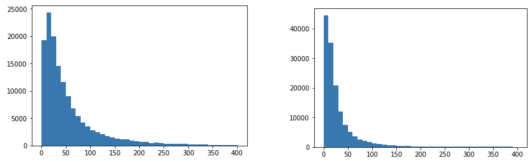

Figure 5: Word count distribution in sentences before(left) and after(Right) removing stop words.

stop word removal technique provides added benefits including a reduced dictionary size, removal of high frequency invaluable words, etc.

2.4 Stemming

Words in our dictionary used to train the models have a specific level grammatical/lexical inflection which could make our learners ineffective against the variance in the level of inflection in the test data. To solve, this we use a form of suffix removal technique called Stemming. In our implementation we use the popular Porter stemmer provided by the nltk library to stem all the words in our training, test and validation data.

2.5 Lemmatization

While Stemming is an efficient way of finding root words, we could also use our inherent knowledge of the language(English) being used in the data set. Since each word in English has a root word (lemma) that its derived from, we could use this knowledge to uniquely identify a word, irrespective of its inflections. We use such a form of lemmatization provided by the nltk package to identify the lemma of every word in our dataset. Although, this technique is slower than Stemming, it is extremely effective in pre-processing our data.

2.6 TF-IDF Vectorizer

Machine Learning algorithms are trained with discrete numerical data as input features. However, our techniques used above are inadequate to represent our data in a way the Machine Learning algorithms can understand. We use a form of feature extraction provided by sklearn called TF-IDF vectorizer.

A naive approach to vectorize our input data is to extract only those words which occur frequently. However, the existence of extremely frequently words might provide no valuable insight about a given sentence. To solve this, TF-IDF uses the term frequency in accordance with the inverse document frequency which helps disregard such

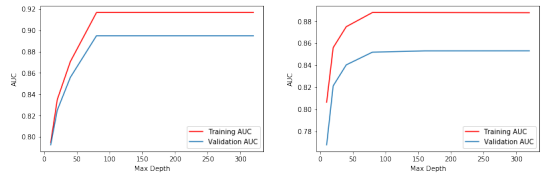

Figure 6: Mean AUC growth for Multi-label(Left) vs Single Label(Right) classification as a function of max-depth.

words. We use this vectorized numerical representation of a sentence to train, validate and predict the data.

2.7 Glove: Word Embedding

To solve the problem with sequential models, we require words in sentences to have a contextual references. Therefore its important to convert the textual data into numeric data while maintaining the contextual references between words in a sentence. **Glove** is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space. During our experiments, we used Glove 50d uncased data, which comprises of pre-trained word vectors.

3 Learning Techniques

3.1 Decision Tree

3.1.1 Motivation

Looking at the training data, it is clear that the decision of classifying a comment as toxic flows naturally from the existence of certain toxic words. A decision-tree can be viewed as a good solution for this kind of representation of the problem. Since the Information gain from each toxic word in the corpus would be high, even a shallow decision tree can be expected to yield good results.

3.1.2 Implementation

Multi-label classification vs Single label classification: A decision tree supports both multi-label and single label classifications. Here we contrast both of them with varying max-depth to analyze their relative efficiencies.

The multi-label classification yields better AUC results at every max-depth. Going further, we will optimize our hyper parameters with this technique.

3.1.3 Hyper-parameter tuning

Here we consider several different hyper parameters that we can tune for an efficient decision tree including **maximum depth**, **minimum sample split**, and **maximum leaf nodes**.

Max depth: The Mean AUC vs max depth illustrated above (Figure 6) shows that at max-depth=80 we have the ideal validation AUC. The AUC does not grow with further increase in the max depth which can be attributed to the fact that certain toxic words in the corpus have sufficient information gain to allow a shallow decision tree. Augmenting this max depth will either result in over-fitting or no added value to the efficiency.

Minimum sample split Always splitting a given node without sufficient sample data below the nodes might result in over fitting the data. Hence we demonstrate how AUC grows and determine the ideal fraction of the samples a node must have to warrant a split.

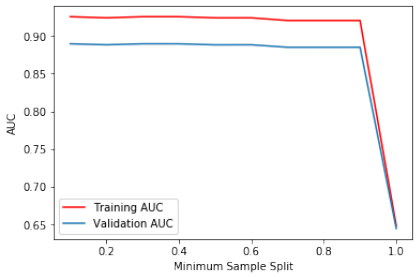

Figure 7: Mean AUC growth against Minimum sample split (Fraction of sample data required to warrant a split)

The ideal value for a minimum fractional sample split for a node with optimal fitting is 0.3.

Max leaf nodes Like max depth, a ceiling limit on the value of max leaf nodes can be tuned to avoid over fitting of the data. As illustrated by Figure 8, the ideal value for max leaf nodes the decision tree must have to avoid over fitting the data is 80. Any value $>$ 80 would result in over fitting.

3.2 Logistic Regression

3.2.1 Motivation

Classifying individual class labels disjointedly can be regarded as a linear classification problem. Since individual class label prediction is a form of

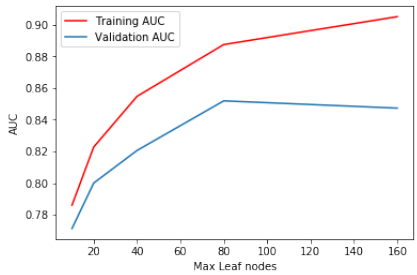

Figure 8: Mean AUC growth as a function of Maximum number of leaf nodes

Training AUC	Validation AUC	Test AUC
0.919	0.895	0.913

Table 1: Final AUCs with optimal hyper parameters using decision trees.

binary classification, classification using dichotomous Logistic Regression should be an efficient approach.

3.2.2 Implementation

Since multi-label prediction is not an intuitive approach with logistic regression techniques, we train and predict individual class labels disjointedly. Because of this, we also have the added benefit of optimizing individual hyper parameters granularly for each class label.

3.2.3 Hyper-parameter tuning

Here we consider several different hyper parameters that we can tune for an efficient logistic regression classifier including Maximum iterations, Penalty, and C.

Maximum iterations: We analyze how the Mean AUC varies as a function of maximum iteration and find that the optimal value where the classifier converges is 10.

Grid search: Optimizing a fixed value of hyper parameters for all class labels might not yield the most optimal results. Hence, to identify these optimal values for individual class labels we perform an extensive **Grid search** over all possible combinations of the hyper parameters. For the penalty, we use l1 and l2 as the possible penalty techniques. Maximum iterations range between 8-100 and C takes the 10 increasingly logarithmic values. We arrived at contrasting optimal hyper parameters findings for individual classes.

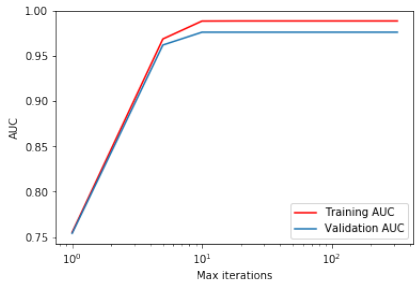

Figure 9: Mean AUC growth against Maximum Iterations

Class	Penalty	Max iter	C
Toxic	l2	8	2.78
Severely Toxic	l1	20	1
Obscene	l1	10	1
Threat	l1	40	1
Insult	l2	8	464.158
Identity Hate	l2	20	1

Table 2: Optimal hyper parameters found for individual classes.

Training AUC	Validation AUC	Test AUC
0.985	0.977	0.971

Table 3: Final AUCs with optimal hyper parameters using Logistic Regression.

3.3 Long Short Term Memory

3.3.1 Motivation

From the Sequential Models course by (Andrew NG), we understand that sequence models like RNNs are very good at identifying the temporal relationship between the data. This will help us in approaching Toxic comments classification problem, because there exist a sequential relationship between words in a sentence. To learn more about this, we explored Gated Recurrent Unit (GRU) and Long short term memory (LSTM) models. In most of the natural language processing problems, LSTM performs fairly well. Therefore we decided to explore LSTM models.

3.3.2 Implementation

Using a vanilla LSTM model would give a good performance on natural language processing problems. But it will fail to detect cases like,

One comment mentioning

"He said, Tobby is a disgusting person"

and another with

"He said, Tobby is an amazing guy"

Since next word, is predicted or weighted based on last word in LSTM, cases like these will give us errors and therefore it is very important for us to have a future knowledge to correctly predict the output. Bidirectional RNNs help us solve this problem. Therefore we decided to use Bidirectional RNN with LSTM. This modification gave us better performance.

Details: In our bidirectional RNN model with LSTM, we have used 5 different layers. First layer is Embedding layer, which takes the preprocessed cleaned data and converts it into glove vector representation. We have kept the maximum size of of each vector to be 50. Therefore LSTM takes 50 as input size and produces output with 0.1 dropout and recurrent dropout. This result is max pooled with Global Max Pooling layer for down-sampling which will help us in identifying the most important features. This result is then connected to a dense layer with **Exponential Linear unit** (which performed better than Rectified Linear Unit) in our experiments. This output is fed to final dense layer with drop out 0.1. Final layer applies sigmoid function to predict the values of output neurons. We followed (Anil) tutorial for understanding how LSTM is applied.

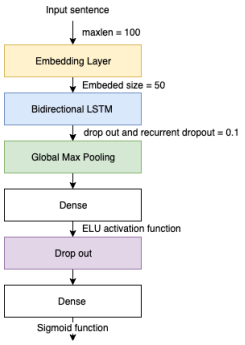

Figure 10: Bidirectional RNN with LSTM model

3.3.3 Hyper-Parameter Tuning

Due to limited power resource, we restricted our parameters to Drop out, Activation function and Optimizer. We performed the experiments with dropout values of 0.1, 0.3, and 0.5 with both activation functions: ReLU and ELU for 5 epochs.

Dropout ranging: In both of our experiments, we observed that that increasing dropout to 0.5 is not helping as it is ignoring lot of neurons in that layer. Therefore drop out factor of 0.1 gave us the best result. This logically also makes sense for penalizing unimportant features during training.

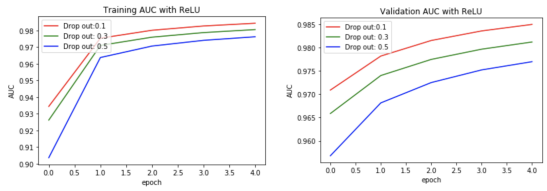

Figure 11: Comparison of Training vs Validation AUC with ReLU

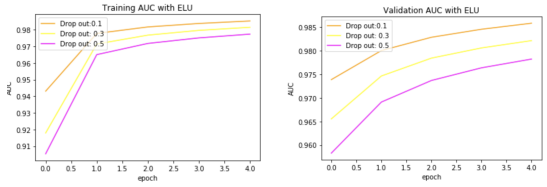

Figure 12: Comparison of Training vs Validation AUC with ELU

ReLU vs ELU (Liu): ReLU is known for facing problems with negative values. For negative values, it outputs 0. A ReLU neuron is dead if its stuck in the negative side and always outputs 0. Because the slope of ReLU in the negative range is also 0, once a neuron gets negative, its unlikely for it to recover. Such neurons are not playing any role in discriminating the input and is essentially useless. Over the time you may end up with a large part of your network doing nothing. ELU handles this problem graciously by producing small output when input is negative. During our experiments found that, ELU converges faster and learns better. Therefore for our final model we chose ELU to be the activation function

Adam Optimzer vs RMSProp:

RMSProp	Validation
1	0.9690
2	0.9736
3	0.9752
4	0.9761
5	0.9767

Adam	Validation
1	0.9739
2	0.9800
3	0.9828
4	0.9845
5	0.9858

Table 4: Comparison of AUC between Adam and RMSprop optimizer with ELU activation function in Bidirectional LSTM

3.4 XGBoost

3.4.1 Motivation

XGBoost(Documentation) is an advanced gradient boosted decision tree classifier. As we have

Training AUC	Validation AUC	Test AUC
0.9853	0.9858	0.9749

Table 5: Final AUCs with optimal hyper parameters using Bidirectional RNN with LSTM .

learned in boosting, the trees are built sequentially such that each subsequent tree aims to reduce the errors of the previous tree. Each of these weak learners contributes some vital information for prediction, enabling the boosting technique to produce a strong learner by effectively combining these weak learners. Such a method could be useful in solving this problem. Therefore we decided to perform one experiment with XGBoost learner as well.

3.4.2 Implementation

During our experiment we didn't perform hyperparamater tuning for this algorithm because it is was very time consuming with the help of grid or random search cv. We predefined all input arguments that control the performance of the classifier.

We kept number of estimators to be 600, max depth to be 3 and minimum child weight to be 1.

Training AUC	Validation AUC	Test AUC
0.9638	0.9599	0.9035

Table 6: Final AUCs with XGBoost.

3.5 Ensemble Method

3.5.1 Motivation

From the above ML techniques, we deduce that there are some techniques which produce strong learners while others produce weak learners. This renders an opportunity to use ensemble based techniques which combines our weak and strong classifiers to produce a stronger classifier which could classify some outlier data points more effectively.

3.5.2 Implementation

Here, we implement a voting ensemble between Decision Trees, Logistic Regression and an SVM classifier. For Decision Trees and Logistic Regression we train with the hyper parameters we optimized above. Since Logistic regression performed efficiently with regard to the AUC-ROC metric, we provide Logistic Regression an added weight in the voting step while Decision-Trees and SVM have equal weights.

Training AUC	Validation AUC	Test AUC
0.988	0.978	0.97

Table 7: Final AUCs using a 3-model Voting Ensemble Classifier.

4 Comparative Analysis

4.1 AUC

As demonstrated in the above techniques, we consistently use AUC as an evaluation metric to identify the optimal hyper parameters. Performance measures like accuracy and loss are poor representations of the underlying optimization problem since the distribution of data is particularly skewed towards non-toxic comments. Hence, we calculate AUCs for individual classes and use the mean of these as a performance measure.

4.2 Relative performance analysis

Name	Training	Validation	Test
Logistic Reg.	0.984	0.977	0.971
D-Trees	0.918	0.895	0.913
LSTM	0.985	0.986	0.975
XGBoost	0.964	0.960	0.903
Ensemble	0.988	0.978	0.97

Table 8: Final AUCs for optimal hyper parameters across all models.

From the above Table 8, we see that while Logistic Regression, LSTM and Ensemble techniques have comparable performance on test data, LSTM would be the recommended model for this classification problem.

5 Distribution of Work

Sameer: LSTM, XGBoost, Data Visualization
Pranav: Linear Regression, Decision Trees and Ensemble
Combined: Data Preprocessing

Acknowledgments

We would like to thank Prof. Xiaohui Xie for teaching us complex concepts of Machine learning in easy way. We would also like to thank

Deying Kong and Yoshitomo Matsubara for being prompt in solving our questions and guiding us through.

References

Andrew NG. *Sequential Models course*. Coursera.

Anil. *Bidirectional LSTM*, volume 1.

XGBoost Documentation. *Introduction to Boosted Trees*.

Hafsa Jabeen. 2018. *Stemming and Lemmatization in Python*.

Jagan. 2018. Kernel discussion on eda. 21.

Danqing Liu. A practical guide to relu. *Medium*.

A Appendices

We present the code for the above report in the Github repository below. Github URL `https://github.com/sameershinde14/ToxicCommentsChallengeKaggle/`