Ahmad Khan, David Cheng

Identifying Toxic Comments Using Deep Learning

Overview

One of the greatest current challenges to the free and open internet is the potential of use of online platform for abuse and harassment. In this project, a series of deep learning models were trained over online comments data, gathered from *Kaggle's Toxic Comment Classification Challenge**, to help predict and identify comments that were abusive and harassing. These models were then subsequently evaluated for their performance against a baseline lower bound.

Methodology

The comments in the dataset (total approximately 150K), could potentially fit, non-exclusively, into any of six categories: "toxic", "severe toxic", "obscene", "threat", "insult", and/or "identity hate" or none. Models were trained on 70% of the data and tuned for the best parameters and architecture, using an unseen validation set that comprised of randomly selected 30% of the original data, Finally the best model was then tested against the hidden test dataset provided by Kaggle. The metric for evaluation for the models was primarily accuracy with precision and recall used as additional metrics for further evaluation. The loss function used was binary cross-entropy.

The *Keras* deep learning framework, with a *Tensorflow* backend was used to generate these classification models. Training was conducted in Ubuntu virtual machines with *Cuda* GPUs.

Baseline Prediction

Before training a deep learning model, a baseline was established to serve as a lower bound in performance for any model to be considered 'good'. The challenge with the data was that the classes were extremely unbalanced. For every class (toxic, obscene etc) at least 90% of all comments were not identified as positive for that class. In some extreme cases like the Threat class, that percentage was higher than 99.5%.

Therefore, a naive but ultimately misleading way to get the 'best' accuracy would be to just predict every comment and for class as 0 or non toxic/obscene etc. In other words the baseline for our model can be thought of as an 'unmoderated forum.' where no comments are flagged as inappropriate. Predicting everything as non-offensive and nothing as offensive or toxic, returned 96.3% in accuracy. Alternatively, using scikit learn's (subset) accuracy metric (metric predicts as correctly classified if and only if every single class was correctly predicted) the (subset) accuracy

was 89.8% for the baseline. The accuracy for each class itself using this method is simply the % of comments that are positive for each class as shown below:

Figure 1 - Baseline Accuracy for Each Class (Total 150K Comments in Dataset)

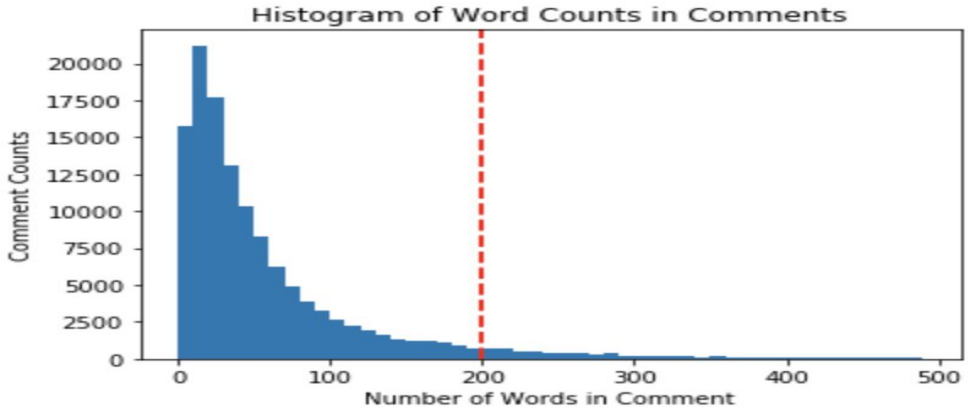
	Toxic	**Severe Toxic**	**Obscene**	**Threat**	**Insult**	**Identity Hate**
% of Comments	9.54	0.98	5.26	0.31	4.87	0.88
Class Accuracy %	90.46	99.02	94.74	99.69	95.13	99.12

Data Preprocessing

After establishing a baseline, data was preprocessed so it could be fed into different deep learning models. Using Keras' tokenizer, all punctuation and formatting was removed from the text and words were all converted to lower-case. In addition:

1) The words were each mapped to unique integers so there could be a numeric vector representation for each comment. Only the top 250,000 occurring words were mapped to avoid rare or unique words that would not generalize well for learning.
2) All comments were capped at 200 words. This means that any comment greater than 200 words only included its first 200 words and nothing after that. This was done to ensure that the neural nets all had the same size of inputs for each input vector. The 200 word limit was discovered by looking at the distribution of word lengths for all comments plotted below. As it can be seen that there are very few comments with more than 200 words with the bulk less than 50.
3) For comments that were less than 200 words, 'left padding' with zeros was used. This assigned 0 integer values to all positions prior to the first word in the comment until the 200 length limit was reached.

Figure 2 - Distribution of Comment Lengths



Model Training and Selection

Different neural net models were then tried to see which could perform best over the training data.

Convolutional Neural Networks

The CNN architecture for the model that was used is as follows . First an Embedding layer converts the comments, into a lower dimensional embedding matrix, that represents each word in the comment as a low dimensional vector. One Hot Encoding was initially used but discarded due to high dimensionality and sparsity of the one hot encoded vector matrix which led to lots of inefficiencies in model training (and poorer performance).

Then a 1 dimensional convolutional layer with multiple filters and a sliding kernel (window) is passed over the data to help identify textual relationships between different words. A 1D Convolutional Layer was used since sliding in 1 dimension (where the words themselves are the single dimension) over different windows could help identify N grams that could be predictive of the toxicity of a comment. After Convolutions, Max Pooling is applied to help reduce the dimensionality of the data and help make strong relationships more pronounced. At each step Dropout of 20% is introduced to help with better generalization of the model, and prevent certain weights from solely dominating the results. Finally, a normal Dense hidden layer is used and a softmax activation function is applied to the output of the hidden layer to predict the probability of each of the 6 comment classifications.

While the above architecture is fairly standard, to further help improve the performance different parameter choices were experimented over to see which combinations of parameters and architectures could help the best performance

a) **Experiment with different number of convolutional and pooling layers**

Instead of using only 1 convolutional layer, 2 and even 3 convolutional + max pooling layers were used in the architecture. The goal was to see if adding more layers helped improve the performance of the model. Results are summarized below and show that as the layers increase we actually see declining model performance. Therefore only a single convolutional + pooling layer was selected for the final model.

Figure 3 - CNN Model Performance with Different Number of Convolutional Layers

Number of Convolutional and Max Pooling Layers	Train Accuracy	Validation Accuracy
1	0.9900	0.9814
2	0.9896	0.9782
3	0.9868	0.9769

b) Experimenting with different activation functions

Multiple activation functions were also tried (over a single convolutional layer CNN) to see which one can perform the best. Performance is generally best for the relu or tanh activation functions on the validation set. For the final model Relu was used.

Figure 4 - Performance for each activation function

Activation Function	Validation Loss	Validation Accuracy
Sigmoid	0.05680	0.9800
Relu	0.05536	0.9817
Tanh	0.05576	0.9816
Elu	0.0567	0.9816

c) Experimenting with Different CNN hyperparameter choices

CNNs have many parameters like the number of filters to use for each convolutional layer, the window size of each filter, the stride at each step, and the pool size used for pooling. In addition the embedding matrix dimension size and the number of neurons in the final hidden dense layer can also be altered. In this project, a brute force grid search was performed over a few combinations of each of these parameters. (Note: because of memory and time constraints, only a few values could be tried upon and this list does not represent a necessarily optimal parameter choice).

Specifically these specific parameters took the following values during grid search:
Filter = 64, 128
Kernel Size = 2,3,4,5
Embedding Dimension = 32,64,128

Other parameters were held constant (batch size = 128, activation function = relu, convolved layers = 1, stride = 1, pool size = 2)

The top 3 models generated (as measured by validation accuracy) are shown below along with their validation loss and accuracy. Note: these models are not that much worse from the bottom 3 models so it seems that tweaking parameters alone does not lead to a huge gain in performance on this data set.

Figure 5 - Top 3 Model Parameter Choices

Model Rank	Filters	Kernel Size	Embedding Dimension	Hidden Dimension	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy
1	64	5	128	128	0.0292	0.9888	0.0520	0.9821
2	128	2	128	128	0.0313	0.9876	0.0499	0.9820
3	64	4	128	128	0.0297	0.9883	0.0523	0.9820

Figure 6 - Bottom 3 Model Parameter Choices

Model Rank	Filters	Kernel Size	Embedding Dimension	Hidden Dimension	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy
22	64	2	32	128	0.0395	0.9845	0.0517	0.9805
23	64	3	128	128	0.0311	0.9879	0.0524	0.9805
24	64	3	32	128	0.0359	0.9861	0.0537	0.9800

Generally speaking, it appears that the more complex the model in terms of embedding dimensions then the better the accuracy. All the top 3 models have the hidden dimension as the highest value of 128. This makes sense because a larger embedding dimension allows the CNN model to learn more detailed word relationships. High Window (Kernel) sizes like 5 and 4 are also more likely to be better performing models as this allows us a CNN to learn over a greater string of words at each stride in the filter layer.

Final CNN Model Evaluation

The final CNN model using the above optimized parameters therefore is as follows (Filters = 64, Kernel Size = 5, Embedding Dimension = 128, Activation Function = Relu, Convolutional Layers = 1, along with Hidden Layer Dimension = 128, Batch Size = 128, Pool Size = 2)

It's performance on validation data relative to the baseline can be summarized as below (Note: the threshold for positive classification was a score of 0.5 or more)

Figure 7 - Final CNN Model Performance relative to Baseline

Model	Loss	Unweighted Accuracy	Precision	Recall	Subset Accuracy
Final CNN	0.053	0.982	0.750	0.731	0.912
Baseline	N/A	0.963	0.000	0.000	0.897

Despite a heavily imbalanced set of classes, the model beats the baseline by every metric above. It's accuracy (both subset and unweighted) increases by 1.5% over baseline. More importantly when we break down Precision and Recall by each class (instead of weighted averaging it over all classes), we see that for some classes like Toxic, Obscene, Insult, the model does very well exceeding precision and recall by 70% for each of these). This is obviously better than the unmoderated baseline where we had 0 precision and recall as we were optimizing on very high accuracy but at the cost of zero predictive power for positive comments.

Figure 8 - Precision and Recall for Each Predicted Class

	Toxic	Severe Toxic	Obscene	Threat	Insult	Identity Hate
Precision	0.740	0.529	0.809	0.00	0.736	0.808
Recall	0.813	0.424	0.822	0.00	0.702	0.131

There is however one area where the model fails significantly. It fails at identifying any threats. This might be because the % of positive comments with Threats flagged is extremely low at only 0.3% so the model is unable to learn from such a small set of positive examples. Setting a lower score as a threshold for classifying comments as threats might help with identifying more threats.

Long Short Term Neural Networks

In addition to a CNN, a LSTM model was also built to see if the performance could be improved by using an alternative deep learning algorithm. Long short term memory neural networks are a subset of recurrent neural networks (RNNs) which allow the model to learn from a temporal sequence of events by using inputs from one event as inputs to the next event. This is especially suitable for text data since sentences are a sequence of words where one word can impact the meaning of other words before or after it. LSTMS in addition are optimized to allow for retention of not only short term memory but also long term memory (words that appeared a long time ago). Finally, in bidirectional LSTMs the first recurrent layer is duplicated. The original

information is fed through one copy of the layer, while a reversed copy of the information is fed through the duplicated layer. This allows for better connections to be made across larger gaps and gives LSTMs their 'long term memory' association.

Architecture

The LSTM implementation used is as follows. An embedding layer is used first, in the same manner as in the CNN. After that a bidirectional recurrent layer is introduced. A high level way of conceptualizing this would be one feeding the matrix incrementing the index up from 0, the other feeding the matrix into the duplicate RNN layer by decrementing the index from the max index to 0. One dimensional max pooling is used after along with dropouts at every step. Finally a dense hidden layer with sigmoid activation transforms into predicted class probabilities.

LSTM Model Evaluation

The LSTM performs better than the CNN beating the CNN by 1% in subset accuracy and 0.1% in unweighted accuracy. It's Precision is also better but at the expense of slightly worse Recall.

Figure 7 - LSTM Model Performance relative to Baseline on Validation Set

Model	Loss	Accuracy	Precision	Recall	Subset Accuracy
LSTM	0.048	0.9831	0.823	0.671	0.921
Final CNN	0.053	0.9821	0.750	0.731	0.912

Out of Sample (Test) Evaluation

With the models tuned, final evaluation was done on held out test data which had never been seen before. This test data was the test set provided by Kaggle for the competition with the class labels completely invisible to participants. Predictions were generated on the test data and submitted. The performance was as follows. Note: Unfortunately, the metric for evaluation for the competition was changed to AUC. So the tests results below are for AUC and not accuracy.

Model	Test Score (AUC)
Baseline	0.5000
CNN Model	0.9638
LSTM Model	0.9630
Blended Ensemble (50% CNN + 50% LSTM)	**0.9705**

Both models when evaluated against AUC do similarly with the CNN slightly better. Baseline does much worse at only 0.500 AUC. Therefore, just 2% increase in accuracy has a dramatic improvement over the baseline in terms of AUC.

Both the CNN and LSTM model predictions were also blended with equal weights (50%) each so the final ensemble prediction was just prediction = (CNN prediction + LSTM prediction)/2. By ensembling the 2 models, AUC out of sample increased significantly up to 0.9705. Thus by ensembling final model performance was improved even more.

Conclusion

In this project, different deep learning models(CNN + LSTMs) were trained over comments data to identify comments that were inappropriate along 6 different criteria. A baseline prediction was established and the models were tuned and tweaked over 70% of the data (train) and then evaluated over a 30% validation split of the data. Final models outperformed the baseline by 2% in accuracy in validation and much more by 0.47 when measuring by AUC on the test data.

Further improvements could however be made including:

1) Using pre-trained word embedding models like word2vec and glove for getting better word embeddings
2) Using punctuation and upper case as unique words instead of removing it
3) Converting misspelled words into the most similar word in the corpus (using some sort of similarity metric like levinshtein distance)
4) Using ensembles of many more different models to blend in predictions from many uncorrelated models
5) Use AUC for training as well since the classes are so imbalanced making accuracy a less useful metric for model evaluation