Definition

Project Overview

Toxicity is a common problem in online forums and comment sections. Many people have had the experience of opening the comment section of an article only to be met with insults, obscenity, and hatred. Extensive human moderation is one solution, but at large scale it's quite costly, and it can be a harrowing task citep:verge-2019-facebook-moderation. The ability to augment human moderation with machine learning models to filter out the worst and most obvious toxic comments is critical.

In this project, I created a classifier to identify several types of toxicity in online comments, including threats, obscenity, insults, and identity-based hate. This problem was originally posed in the Toxic Comment Classification Kaggle competition citep:kaggle-toxic-comments.

Problem Statement

The solution is to create a machine learning model that takes in the text of a comment and outputs a score for each of the toxicity labels, denoting the probability that the comment falls under one of those categories of toxicity. The scores would be used to flag comments for manual inspection by human moderators.

Metrics

The main evaluation metric will be the mean label-wise ROC AUC:

$$\frac{1}{L}\sum_{l=1}^{L} AUC_l \tag{1}$$

where L is the number of labels (in this case 6). ROC AUC is a good metric for this problem because it's

Additionally, the ROC AUC, precision and recall for each individual label will be analyzed as certain labels will be easier to identify than others.

	Label					
	Toxic	Severe Toxic	Obscene	Threat	Insult	Identity Hate
% of Examples	9.57%	0.878%	5.43%	0.308%	5.06%	0.947%
# of Examples	21,348	1,962	12,140	689	11,304	2,117

Table 1: Summary data for labels

Analysis

Data Exploration

The Kaggle dataset contains two files: the training data with 159,571 examples, and the test data with 153,164. In the test data, there are 89,186 examples without labels–these are Kaggle's final test set–and 63,978 with labels. Since this project wasn't intended to be uploaded to Kaggle, I threw out the unlabeled examples in the test set and combined the remaining examples with the training set for a final size of 223,549 examples across my training, validation, and test sets.

Table 1 shows some summary data on each label. There are quite a few examples for the toxic, threat, and insult labels, but very vew for threat. In all, 22.2% of examples have at least one positive label.

These comments have some problematic characteristics. There is quite a bit of punctuation in some that will have to be stripped out, and many comments have an IP address stamped at the end which will also have to be removed.

Exploratory Visualization

Figure 1 shows the correlations between labels. The highest correlations are between each pair of toxic, obscene, and insult. Threat is not highly correlated with any other label because it's so low volume.

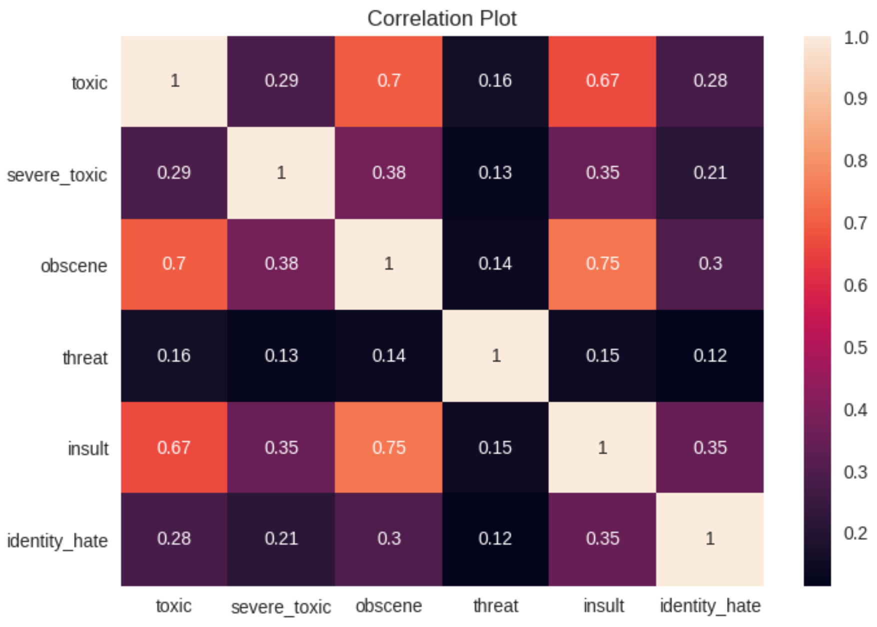

Figure 1: Pearson correlation coefficients for each pair of labels

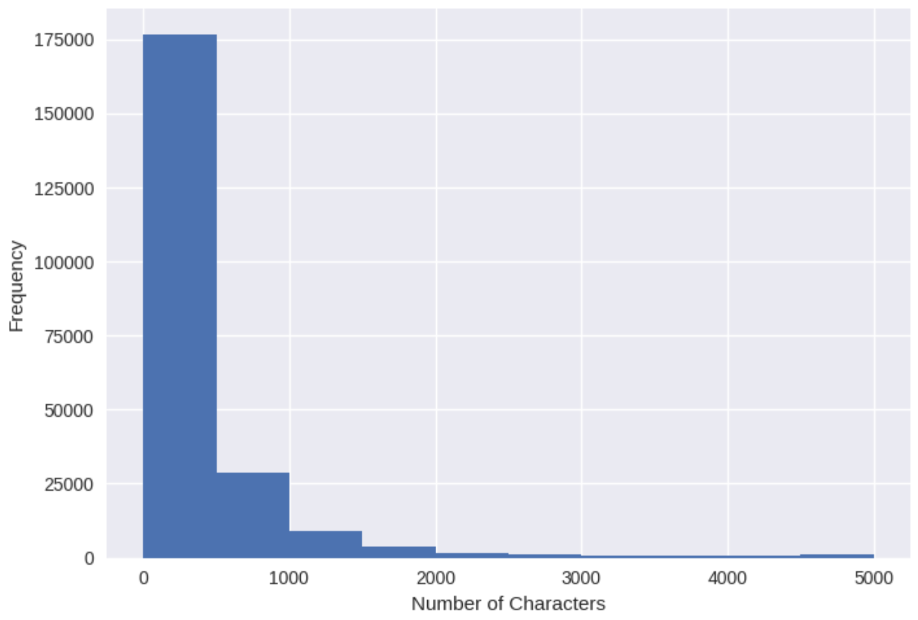

Figure 2: Distribution of the number of characters per example

Investigating the comment texts themselves, most comments are fairly short. Figure 2 shows the distribution of character lengths: 79.0% of examples are less than 500 characters in length, but there is a long tail of longer comments up to 5,000 characters.

Figure 3 shows a similar pattern for word counts: most comments are fairly short–54.0% are less than 40 words–while there is also a long tail of comments up to 2,000 words.

Algorithms and Techniques

The classifier I used is a multiayer network consisting of an embedding layer, one or many LSTM layers, one or many fully connected layers (these depend on the hyperparameters of the run), and finally a 6-node layer with sigmoid activation, one for each class label. LSTMs should be much more performant

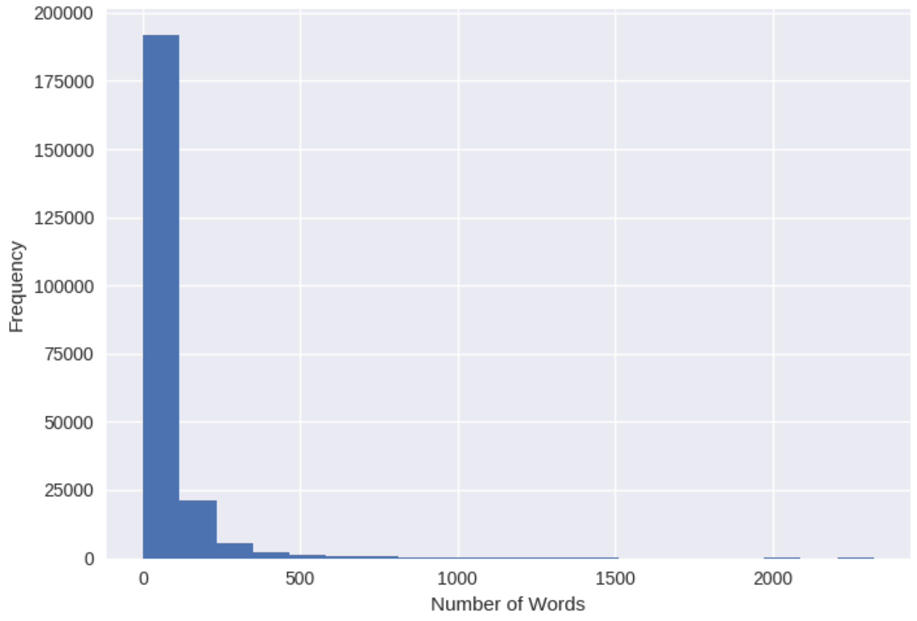

Figure 3: Distribution of the number of words per example

on this problem than non-recurrent approaches because context is very important in identifying some of these classes. For example, identifying identity hate will require the model to understand the context in which an identity is being invoked: a comment saying "I'm <X identity> and therefore I hate <something>" is not identity hate, whereas "I hate <X identity>" is. A simple bag of words approach would not do a very good job of telling one from the other. On the other hand, obscenity is probably easy to identify with just a bag of words: either there's an obscene word in the comment or there isn't.

Benchmark

Jigsaw/Conversation AI, the company that sponsored the Kaggle competition, have their own model citep:jigsaw-model-project, a convolutional neural network trained with GloVe word embeddings. Their model achieved an ROC AUC of 0.96857. It's not clear to me whether this is on the exact same dataset as used in this project, but it's still a good benchmark nonetheless.

Methodology

Data Preprocessing

The preprocessing is done in prep.ipynb and works as follows:

1. Each comment is cleaned by removing anything that looks like an IP address (see the note Data Exploration), converting all whitespace into spaces, removing punctuation, and lowercasing the text.
2. Each comment is tokenized using nkltk's $\text{word}_{\text{tokenize}}$ function, stop words–any word that exists in nltk's english stop words dictionary–are removed, misspellings of edit distance <= 2 are removed by using SymSpell's dictionary (this step was added later; see note in Refinement), and finally each token is lemmatized using nltk's WordNetLemmatizer
3. A dictionary of the 10,000 most common words from the previous step is created.

4. Each comment is encoded using the dictionary. Comments are padded or truncated to the max length of 500 words (before preprocessing this would handle 98.7% of comments in the dataset), and then each word is converted into an integer representation based on the dictionary. If a word is not included in the dictionary, it is replaced with a "missing in vocab" token. The final vocabulary size after this step is 10,002: 10,000 words plus the padding token and the missing token. Finally, the original length of the comment is prepended to the encodings and the labels are prepended to that to make one single array: [label1, label2, . . . , label6, comment length, word1, word2, . . . , word500]

5. The results are split into training, validation, and test sets. First, 10% is reserved for the test set, then 20% of the remaining (18% of the total) is reserved for the validation set and the rest (72%) is used for training. To make sure there are enough examples of each class in each set, these splits are stratified based on a class label (this step was added later; see note in Refinement). For simplicity, that class label is calculated as the maximum positive label for each comment (1-6). Ideally it would be stratified so that each combiniation of labels is considered its own class, but there ends up not being enough examples for that method to work.

Implementation

For each iteration, the training and evaluation happens in test.ipynb. The process is as follows:

1. Upload the dictionary and training and validation datasets to S3 so they're available to the SageMaker estimator.

2. Create an estimator and load both the training and validation datasets into memory. For the training set, samples are weighted inversely to the frequency of their class (this step was added later; see note in Refinement and see Data Preprocessing step #5 for the class calculation). The model is trained using the Adam optimizer and binary cross entropy loss with a batch size of 512. During training, the loss function is printed for the training set after each epoch, along with a set of metrics calculated on the validation set: accuracy, precision, recall and ROC

AUC. For each of those metrics, each of the individual class-wise metrics are printed along with the average across classes. As mentioned above, the average ROC AUC is the primary metric.

3. After training is completed, I looked at the log outputs to see the final ROC AUC, and whether it continued to increase during training or if it started overfitting.

4. The last run to create the final model used hyperparameter optimization. The hyperparameters tested were:

   - embedding dimension of 32 or 64
   - number of LSTM layers between 1 and 4
   - number of hidden layers and their dimensions: [100], [100, 64], [100, 64, 32], [100, 64, 32, 16]

   I configured SageMaker to read the average ROC AUC printed to the log and use that as the objective measure and early stopping metric. In all, 20 models were trained before settling on the best model.

After hyperparameter tuning, the best model that will be evaluated in Results looks like figure 4.

Refinement

The initial run I tried didn't use stratified sampling to create the training/validation/test sets, didn't try to correct misspelled words, used a 32-dimension embedding layer, 1 LSTM layer, and 1 100-dimension fully connected layer. When I tried to run the validation metrics, I ran into a problem: there weren't any samples with the threat label in the validation set! So I needed to update the train/val/test splitting to stratify on the class label. With 6 independent labels, that was a bit tricky. Ideally, I wanted to stratify on each possible combination of labels (2^6 combinations) to make sure each data set has a representative sample, but unfortunately there aren't enough samples in each combination for that to work. So instead I went with a simpler but cruder method: an example is labeled with its max positive class (1-6 for each of the labels, or 0 if all labels are 0). I re-ran the train/val/test split using that class to stratify, and I also added a step to the training script

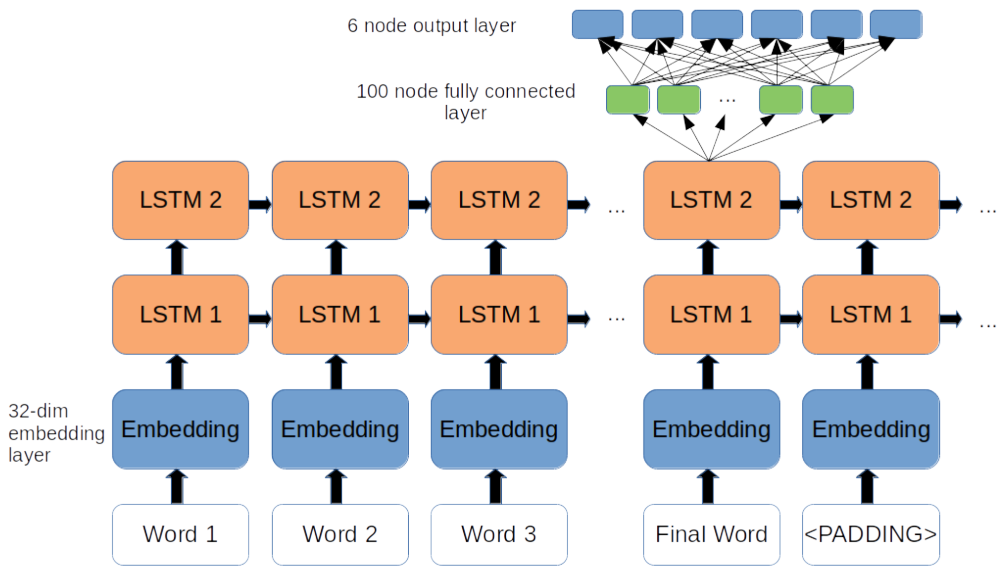

Figure 4: Final network structure

to weight examples based on that class: inversely proportional to the class's prevalence in the training set.

The second run used the same hyperparameters as the first run, with the only difference being those class weights. The results are shown in table 2

	Label						
	Average	Toxic	Severe Toxic	Obscene	Threat	Insult	Identity Hate
ROC AUC	0.872	0.903	0.875	0.920	0.857	0.917	0.761
Accuracy	96.9%	92.5%	98.9%	96.2%	99.7%	95.5%	98.9%
Precision	49.2%	60.6%	38.3%	65.2%	41.7%	55.6%	33.8%
Recall	40.0%	62.8%	26.0%	64.1%	11.5%	54.5%	20.7%

Table 2: Validation set results for run 2

Results

Model Evaluation and Validation

Justification

Conclusion

References