Project Progress Report, ECE – 539

Title: **Toxic Comment Classification**

Name: Deepan Das
UW Net ID: 9080158406

Contents:

Abstract: *In the following document, I have described two baseline models that accomplish the task of classifying toxic comments in a multiclass setting. The first primitive method uses a basic TF-IDF and Logistic Regression setting to accomplish this task whereas the second method uses a Deep Learning model, namely the LSTM model to accomplish this task. The first method, as evaluated by Kaggle's evaluator indicates an accuracy score of about 0.97, whereas the Deep Learning model improves on it and generates an accuracy of 0.983.*

---

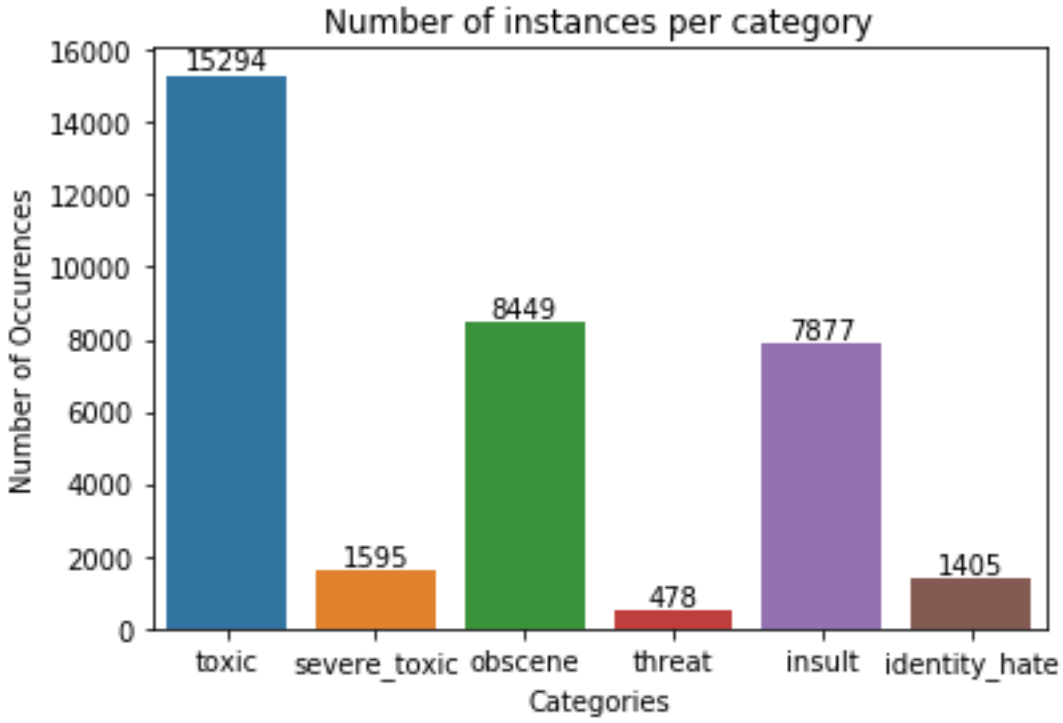
1. Exploratory Data Analysis

Understanding the dataset is an extremely vital task before we go on to build any model. The exploratory data analysis task on this dataset led to the following insights: The dataset being used is from the Kaggle Toxic Comment classification challenge.

a) Categorical Segregation:

Total Comments		
1.	Toxic	15294
2.	Severe Toxic	1595
3.	Obscene	8449
4.	Threat	478
5.	Insult	7877
6.	Identity Hate	1405
7.	None	143346

This tabular data has been visualized as a graph here:



Insights: We see that most comments are categorized as Non-toxic. This gives us a relatively smaller pool for the multiclass data. This also means that a comment can be classified as more than one category as is evident from the following graph. The X-Axis shows the various comments and the Y-Axis shows the number of categories they are classified into. We can see that almost all toxic comments are classified into more than one category.

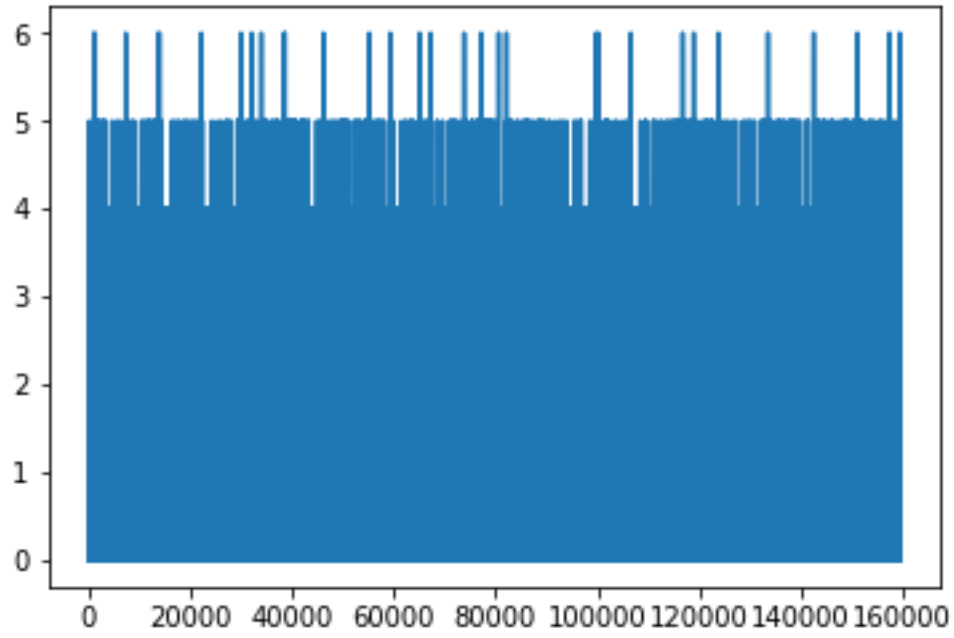


Fig: Figure representing the number of classes a comment has been classified into in the Training Set

---

2. Baseline Model 1: TFIDF Representation + Logistic Regression

Text data requires a set of steps to be applied on it to prepare it even before one can use it for modeling.

- Text must be parsed first to remove words, called **tokenization**
- Encode these words as integers or floating-point values for use as input to an algorithm

The three possible ways are:

1. Convert text to word count vectors with CountVectorizer
2. Convert text to word frequency vectors with Tf-idfVectorizer
3. Convert text to unique integers using HashingVectorizer

Bag of Words Model:

We want to perform classification of text pieces, so the text piece becomes an input and the class label becomes the output for our algorithm. Algorithms take vectors of numbers as input and therefore we need to convert our text to fixed length vectors of numbers. This is a simple model as it throws away all of the order information in the words and focuses on the occurrence of words in a document. The principle steps involved are:

- Assign each word a unique number
- Any document we see can be encoded as a fixed length vector with the length of the vocabulary of known words.
- Value in each position is filled with the count or frequency of each word in the encoded document

The scikit library in Python provides 3 different schemes that we can use and we can understand them before we proceed to build the first baseline model.

Count Vectorizer: Word Counts

Simple way to tokenize a collection of text documents and build a vocabulary of known words, but also to encode new documents using that vocabulary. The process is as follows:

1. Create an instance of CountVectorizer Class
2. Call fit() function in order to learn a vocabulary from one or more documents
3. Call transform() function on one or more documents as needed to encode each as vector.

This returns an encoded vector with a length of the entire vocabulary and an integer count for the number of times each word appeared in the document. These vectors contain a lot of zeros and therefore they are sparse vectors. These encoded vectors can then be used in any Machine Learning task.

Word Frequencies

Calculate word frequencies: Calculate Term Frequency-Inverse Document Frequency which are the components of the resulting scores assigned to each word.

- Term Frequency: Summarizes how often a given word appears within a document
- IDF: Downscales words that appear a lot across documents

This highlights those words that are more interesting, e.g. frequent in a document but not across documents. The weight of a term that occurs in a document is simply proportional to the term frequency.

The term frequency term can be calculated as the following double normalization method. This prevents a bias towards longer documents by dividing the raw frequency by the frequency of the most commonly occurring term in the document.

$$tf(t,d) = 0.5 + 0.5\frac{f_{t,d}}{max\{f_{t',d} in\ d\}}$$

The inverse document frequency is a measure of how much information the word provides. Logarithmically scaled inverse fraction of the documents that contain the word. Obtained by dividing the total number of documents by the total number of documents that contain the word.

$$idf(t,D) = log\frac{N}{|d\ in\ D: t\ in\ d|}$$

TF-IDF is essentially, a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. The statistic, then essentially becomes a product of the above two terms.

Model: Multinomial Logistic Regression:

Multinomial Logistic Regression is a classification method that generalizes logistic regression to multiclass problems and predicts the probabilities od the different possible outcomes of a categorically distributed dependent variable, given a set of independent variable. The Python scikit learn library has an inbuilt tool that can be used for this task and for our first baseline model, this classifier is used to generate a submission file that can be put to test at Kaggle's own evaluator.

The score for this Baseline model(Kaggle score):

Private Score: 0.9761

Public Score: 0.9772

3. Baseline Model 2: LSTM using Keras and TF backend

The preprocessing and data exploration techniques are pretty much the same as the initial baseline. We have to keep in mind the fact that most comments are not of huge lengths. Most comments are well below the length of 100. This initial exploratory task will help us later while building the model.

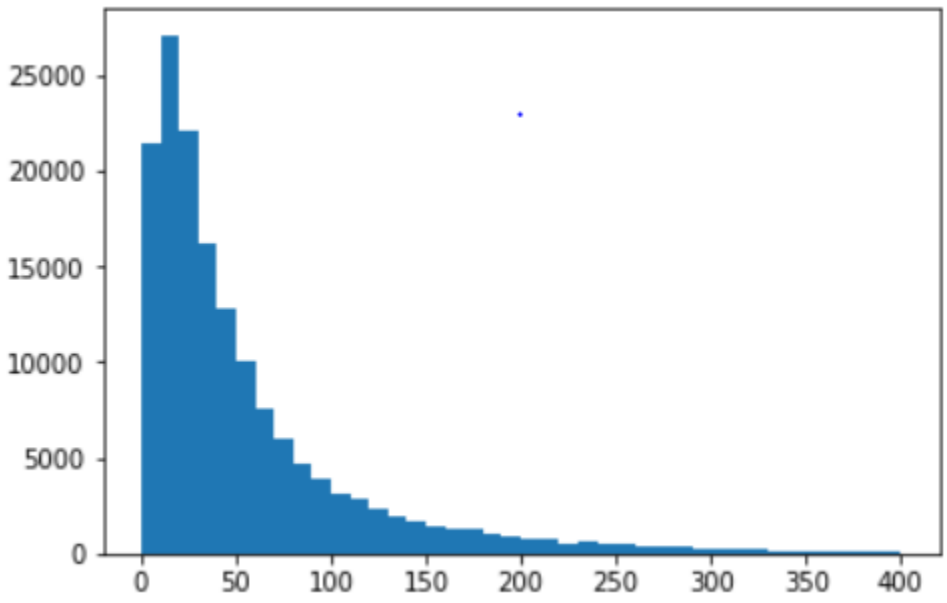


Fig: The length of various comments shown in a histogram plot. The lengths of various comments is along the X-Axis. The Y-Axis denotes the number of such comments.

Before we feed the data into the model being used, we would need to preprocess the words, like in the previous model:

- Tokenization: Break down sentences into unique words
- Indexing: Put the words in a dictionary like structure and given them an index for each
- Index Representation: Represent the sequence of words in the comments in the form of index and feed this chain of index into our LSTM.

By using Keras and the TensorFlow backend, we can implement all the steps quickly. The dictionary can hold a specified number of words and this can be done by specifying the Max-Features option when tokenizing the text corpus. Keras then turns the words into index representations. Proceeding, we have to keep in mind that the input to the LSTM model has to be the same for all comments. This is tackled by incorporating “Padding”. Based on the histogram above, we can safely take the maximum length to be 200. Following this, we can build our model.

The input to the model is a length 200 vector and this is passed on to the Embedding Layer.

Embedding Layer:

The Embedding layer, projects the words it receives to a defined vector space depending on the distance of the surrounding words in a sentence. Embedding allows us to cut down on the huge dimensions we would have had to deal with otherwise. The output of this layer is essentially a list of coordinates of the words in this vector space. Before we use the Embedding layer, we need to define a size of this vector space wherein we determine the various coordinates. This is a tunable parameter but I fixed it at 128. This tensor output from the Embedding layer has been fed into a LSTM Layer.

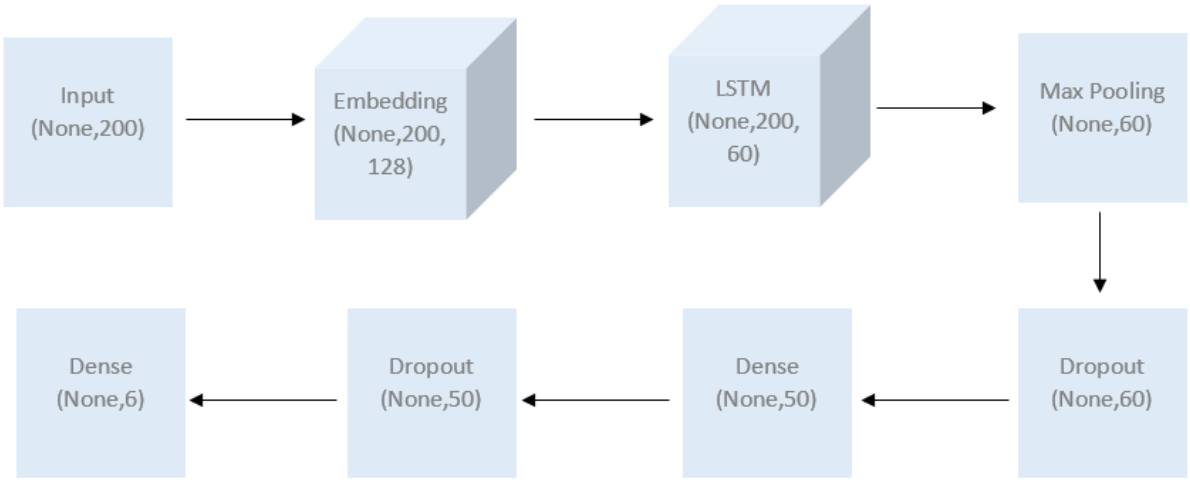


Fig: Visual representation of the model with sizes of the data being denoted at each layer.

The Model: The LSTM is set to produce an output that has a dimension of 60. An LSTM or RNN works by recursively feeding the output of previous networks into the input of the current network and would take the final output after a set number of recursions. However, we might be interested in using the unrolled output of each recursion as the result to pass to the next layer. The LSTM layer runs 200 times because in each iteration it receives a 128-dimensional coordinate representation of each word and produces a 60-length representation for each word. To use this output, we need to convert it into a 2D tensor. For this, we use the MaxPooling operation which gives us a 60 length data point for each sentence. Different variants of pooling could be used. This is passed to a Dropout layer which disables some nodes in the next layer so that the whole network can generalize better. The dropout rate is set at 10%. This is followed by a dense layer and this produces an output of dimension 50. After another dropout layer, we feed the output to a sigmoid layer for the classification task. The Optimization method being used is Adam and the loss being used is the binary cross entropy loss. The model is trained using a batch-size of 32 and runs for 2 epochs. A 10% size validation dataset is also passed on along. The accuracy metric is used here and the update schema is as follows:

```
Train on 143613 samples, validate on 15958 samples

Epoch 1/2

143613/143613 [==============================] - 1731s 12ms/step - loss: 0.0707 -
acc: 0.9777 - val_loss: 0.0489 - val_acc: 0.9818

Epoch 2/2

 46080/143613 [========>.....................] - ETA: 20:04 - loss: 0.0448 - acc:
0.9832
```

As we can see, an accuracy of 98.32% is achieved using this model.

Why I used Keras? This method cuts down a lot of time and saves us the agony of defining the right dimensions for matrices and checking dimensionality matching before we could proceed with the model evaluation. The time saved could be spent on experimenting with different variations on the model.

The summary of the model can be seen as follows:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 200)               0
_________________________________________________________________
embedding_1 (Embedding)      (None, 200, 128)          2560000
_________________________________________________________________
lstm_layer (LSTM)            (None, 200, 60)           45360
_________________________________________________________________
global_max_pooling1d_1 (Glob (None, 60)                0
_________________________________________________________________
dropout_1 (Dropout)          (None, 60)                0
_________________________________________________________________
dense_1 (Dense)              (None, 50)                3050
_________________________________________________________________
dropout_2 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_2 (Dense)              (None, 6)                 306
=================================================================
Total params: 2,608,716
Trainable params: 2,608,716
Non-trainable params: 0
_________________________________________________________________
```

Proceeding Ahead:

A solid baseline has been developed for this project to move ahead. With a fundamental Deep Learning based model being implemented, I plan to do the following:

1. Use popular pre-trained word embeddings like GloVe and FastText and compare the results
2. Use a self developed word embedding model
3. Implement Attention networks in this model
4. Possibly try to implement the most recent CapsuleNet model developed by Hinton.

References:

1. https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge/data : Toxic Comment Classification Challenge
2. https://arxiv.org/pdf/1506.00019.pdf : Recurrent Networks