Toxic Comment Classification

Deepan Das*
Dept. of Electrical and Computer Engineering
University of Wisconsin, Madison
NetID: 9080158406, ddas27@wisc.edu

Abstract

Detection and classification of abusive language online is a difficult NLP challenge as there are endless number of online comments being published every second, all with different contexts and with each consisting of words from different known and unknown set of words. The goal of this work was to explore and implement some of the baseline approaches available online as well as to try and implement advanced concepts that would help model the data in a better fashion. Through a comprehensive Exploratory Data Analysis and visualization setting, we can also aim to better understand the underlying patterns and behaviors that exist amongst these comments.

1 Introduction

Humans have built extensive models of expressing their thoughts via numerous means. The internet has not only become a credible method for expressing one's thoughts, but is also rapidly becoming the single largest means of doing so. In this context, the increasing levels of threat and abuse from certain sections of the online community renders a fair degree of inadequacy to facilitate proper conversations. Quite naturally, one area that requires attention is the identification of negative online behavior and consequentially, restricting their scope. This can be done by developing a classification module for the various types of toxic comments that we may be interested in finding. More recently, Deep Learning methods have been used as a potential method for abusive and toxic comment detection. The following work explores the

*UW Madison - ECE

usage of a very basic Logistic Regression classifier and then moves on to explore Deep Learning based approaches based primarily on Sequential Models, especially the Bidirectional LSTM architecture. Following that, an attempt to address the problem of overfitting has been made by using the techniques of Data Augmentation. An attempt to explore Attention Networks has also been made. The principal motivation to take up this topic was my personal willingness to learn Sequential models and the various implementations, while using the vast library of excellent kernels made available on the Kaggle website.

The remainder of the document is organized as follows: Section 2 discusses the Dataset and some Exploratory Data Analysis performed on the dataset. The EDA reveals some insightful information about the dataset and informs us about how to proceed with this task. Moreover, observations and recommendations from competition winners are taken into account before proceeding ahead with the task. The following sections describe the various models and approaches used to perform this task. It includes a simple Logistic Regression based approach and Sequential Model based Deep Learning approaches. This is followed by the compilation of the results and the ending discussion and remarks section, where we draw and discuss several observations from this task.

2 Dataset and EDA

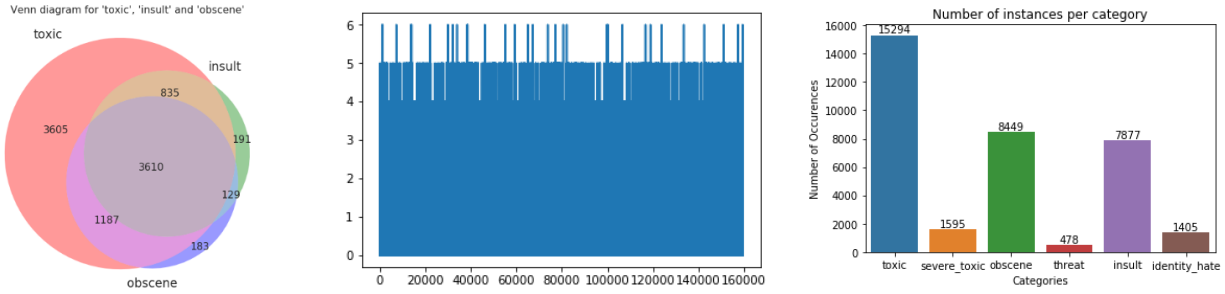
The Dataset used for this task is sourced from a Kaggle competition [1] and is titled as the "Jigsaw/Conversation AI Toxic Comment Classification Challenge Dataset". The creator have so far built a range of publicly available models served through the Perspective API and created this competition to enable participants to build a multi-headed model that is capable of detecting different types of toxicity like threats, obscenity, insults, and identity based hate better than their models. The dataset is composed of comments from Wikipedia's talk page edits. The various categorizations for the comments are: toxic, severe toxic, obscene, threat, insult, and identity hate. The training dataset consists of 160k training samples and the test set consists of 153k samples. Understanding the dataset is an extremely vital task and there are several insights to be drawn from the dataset.

A categorical segregation of the training set reveals the following numbers, presented in the following table. This tabular data has also been presented as a graph and it can be clearly observed that most of the comments in the training set do not belong to any toxic category. Apart from that, an interesting fact to be observed is the overlap of several categories. This can be clearly shown in the Ensuing graph and Venn diagram.

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

Table 1: Table showing distribution of various categories in Training data.

Another interesting data analysis that has been performed is the analysis of the correlation across several parameters shown in the following figures.



(a) (b) (c)

Figure1.a: Venn-Diagram showing the overlap between instances of classes: toxic, insult and obscene; 1.b: Graph showing the number of labels each comment belongs to. It can be clearly seen that almost all comments belong to more than one class; 1.c: Bars showing the number of instances in a class

A few insights can also be drawn from the forum discussion and contributions from the competition champions:

- Most of the complexity lies in the embedding layer and changes in sequential model architecture doesn't affect the results by much. Therefore, it becomes imperative to choose a well-suited Embedding Matrix.
- I found out that most comments are not that long (<50 words). However, longer sentences did correspond to more toxic behavior. And therefore, longer comments were used(~size 100)
- I found out that most comments are not that long(<50 words). However, longer sentences did correspond to more toxic behavior. And therefore, longer comments were used(~size 100)
- CNN-Based approaches are difficult to work with.

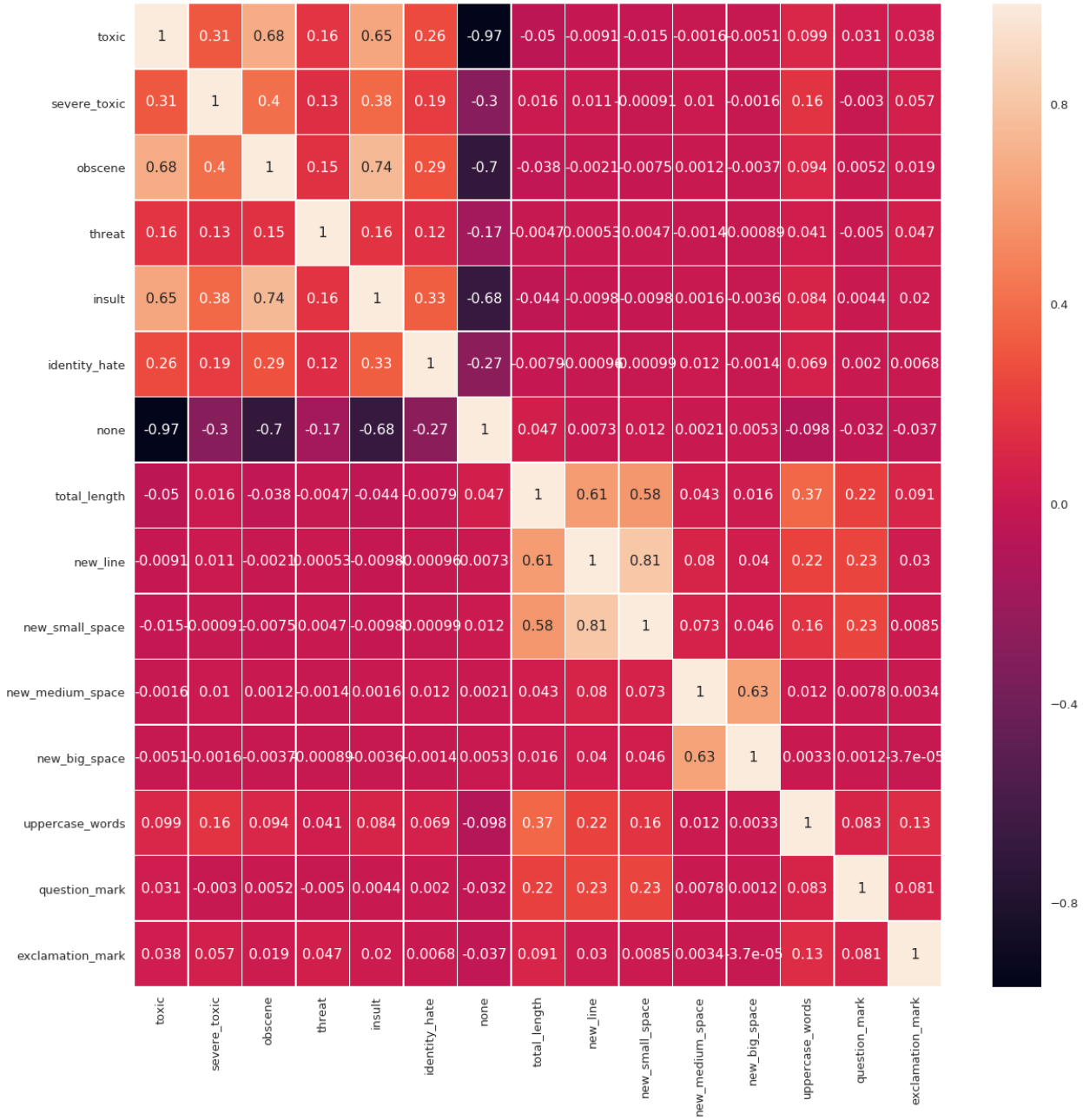


Figure2: Correlation chart developed on the basis of features extracted from the training dataset

3 Approach-I: Logistic Regression using TF-IDF

3.1 Setup and Pre-Processing

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

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

3.1.1 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 sci-kit 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.

3.1.2 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.

3.1.3 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.

3.2 Model: Multinomial Logistic Regression

Multinomial Logistic Regression is a classification method that generalizes logistic regression to multiclass problems and predicts the probabilities of 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.

4 Approach-II: Bi-LSTM and Random Embedding

4.1 Setup and Pre-Processing

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.

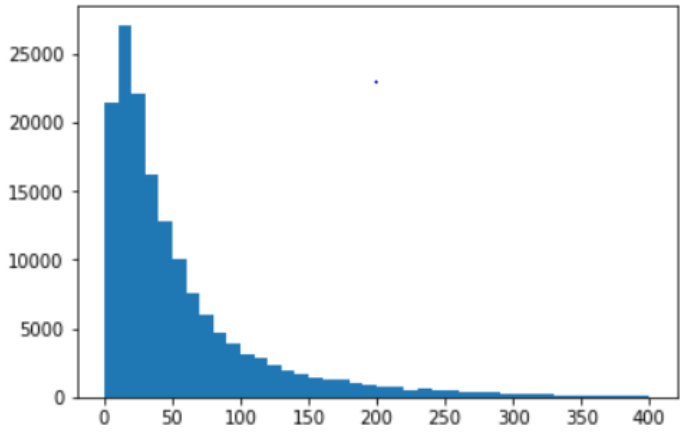


Fig 3: Histogram showing the length of various comments, X-Axis denotes length of various comments and Y-Axis denotes number of such comments

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, which is set randomly based on a Uniform Normal distribution.

4.2 Sequence Models

Human intelligence builds on its previous learnings and is heavily dependent on the understanding of previous events. Our thoughts have persistence. Traditional Neural Networks cannot model this behavior, except for Recurrent Neural Networks. A chunk of neural network A is fed some input x and it outputs a value h. A loop allows information to be passed from one step of the network to the next. The chain like structure found in Recurrent Neural Networks might make us believe that they are intimately connected to sequences and lists.

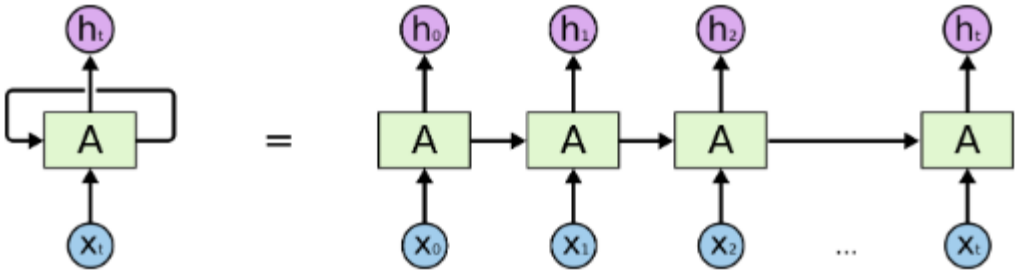


Fig: Recurrent Neural Network: Packed and Unrolled

4.2.1 The Problem of Long-Term Dependencies and LSTMs

RNNs, as it turns out cannot learn long term dependencies. As the gap between two instances grows, RNNs becomes unable to learn to connect the information. This problem was explored in depth and solved by Hochreiter and Bengio et al [6]. Long Short Term Memory networks are explicitly designed to avoid the long term dependency problem. The core idea behind LSTMs is the way the cell state is affected by the input to it and the previous output, and how the cell state, in return affects the present output. This is controlled by three gates: the Forget gate, Reset gate and the Output gate. There are variations to the fundamental LSTM structure, leading to structures like the **Gated Recurrent Unit** and **Peephole LSTMs.**

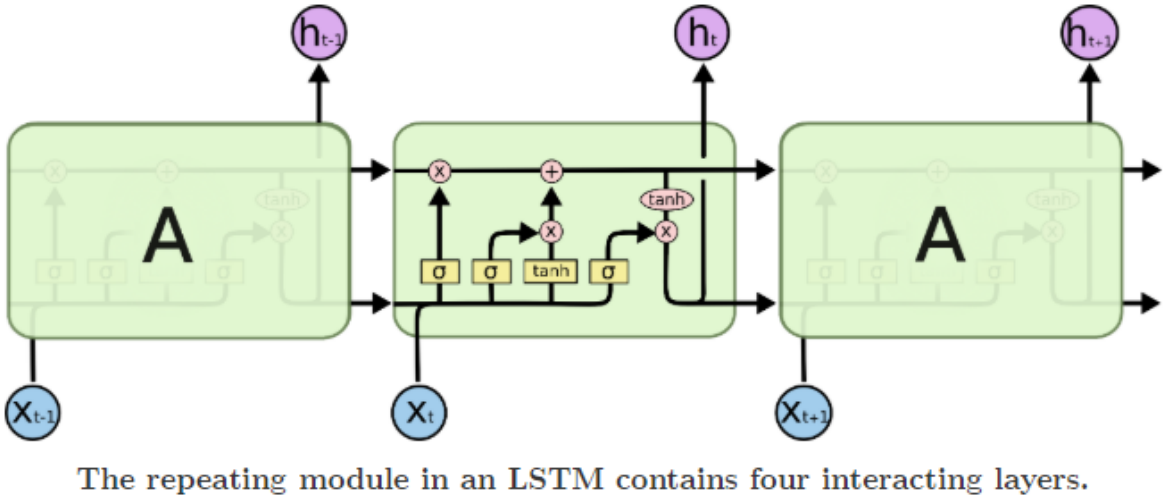


Fig: LSTM Module

4.3 Model

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. 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 Max-Pooling 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 model shows a Kaggle Score of: **0.9761** which is an improvement over the Baseline Logistic Regression Model. The schema is as follows:

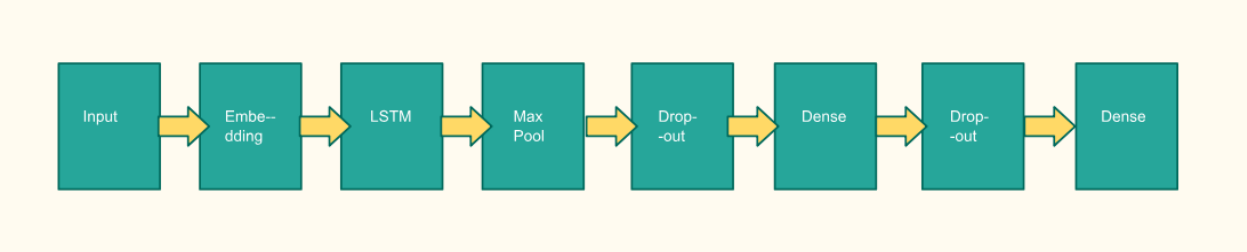


Fig 4: Schematic of the Model used in this approach

5 Approach-III: Bi-LSTM with GloVe Embedding and Further Improvements

5.1 Preprocessing and Setup

The preprocessing and setup is the same as the previous model in this case. Keras is used as the API with TensorFlow as backend. The principal motivation for this work has been derived from **Jeremy Howard's** submission at the Kaggle competition website. The improvements made on this model are all independently developed and are my work.

5.2 Model

This model is a slightly different from the previous model. The Embedding matrix used here is derived from GloVe Embedding representation and the results prove that the usage of GloVe for word embedding improves performance of the model. The model uses an Embedding layer after the input layer that provides an output of a 200-dimensional embedding vector for each word vector. This is followed by the Bidirectional LSTM, followed by the Max-Pooling layer and a Fully Connected layer. A dropout layer is added after this for regularization before a Dense layer. This last dense fully connected layer is followed by a softmax classifier for the multiclass classification task. Finally the schematic of the model can be seen as follows:

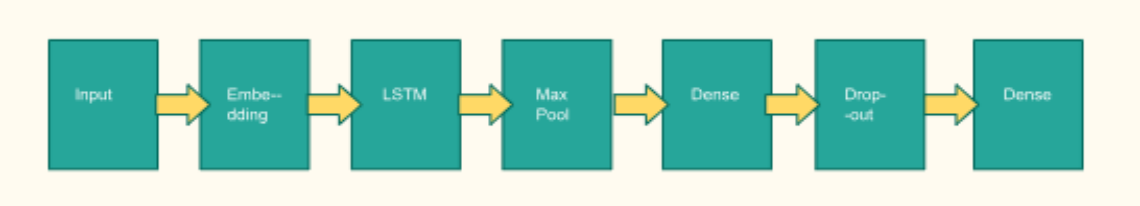


Fig 5: Schematic of the Model used in this approach

5.2.1 GloVe: Global Vectors for Word Representations

GloVe is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on co-occurrence statistics from various corpus and the resulting representations exhibit interesting linear substructures of the word vector space. GloVe is essentially a log-bilinear model with a weighted least-squares objective. The main intuition underlying the model is the simple observation that ratios of word-word co-occurrence probabilities have the potential for encoding some form of meaning. The training objective of GloVe is to learn word vectors such that their dot product equals the logarithm of the words' probability of co-occurrence. Owing to the fact that the logarithm of a ratio equals the difference of logarithms, this objective associates (the logarithm of) ratios of co-occurrence probabilities with vector differences in the word vector space. Because these ratios can encode some form of meaning, this information gets encoded as vector differences as well. For this reason, the resulting word vectors perform very well on word analogy tasks, such as those examined in the word2vec package.

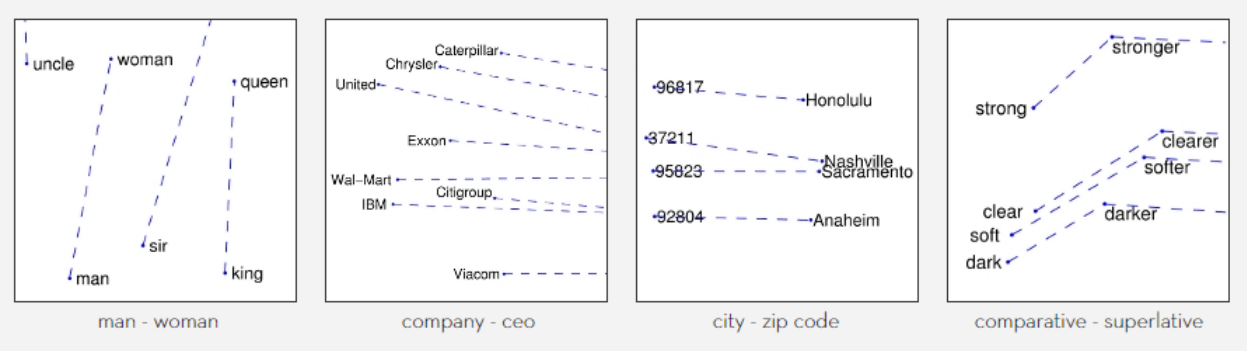


Fig6: The underlying concept that distinguishes man from woman, i.e. sex or gender, may be equivalently specified by various other word pairs, such as king and queen or brother and sister. To state this observation mathematically, we might expect that the vector differences man - woman, king - queen, and brother - sister might all be roughly equal. This property and other interesting patterns can be observed in the above set of visualizations.

5.3 Improvements: Data Augmentation

On investigating the results from the approach discussed here, it is found that the model overfits at the end of the second epoch itself. This could be because Deep Sequential models are data hungry and need a lot of data to train well. This problem is addressed by providing similar contextual data by translating the original English data to another language and translating it back to English. This method is applied with three different languages, in accordance with Pavel Otsyakov's Machine Translation module based on TextBlob. The languages chosen are: Spanish, French and German [3]. The textBlob library uses Google Translate API. The usage of this method helps train the model better and the training loss goes below validation loss at the end of the third epoch. Additionally, a highly improved training and validation accuracy is obtained.

6 Approach-IV: Hierarchical Attention Networks

6.1 Attention Mechanism

In the paper presented by Zichao Yung et al[4], a technique for document classification based on Attention Networks has been proposed. This model has two features that distinguishes it from other works:

- It presents a hierarchical structure resembling the hierarchical structure present in documents
- The levels of attention are two-fold: Applied at word-level and at sentence-level, enabling it to attend differentially to more or less important content when constructing document representation.

Attention refers to the procedure of extracting weights corresponding to each word vector using its own shallow neural network. The architecture of Hierarchical Attention Network is shown

below: It uses a word sequence encoder, a word-level attention layer, a sentence encoder and a sentence-level attention layer. The Network's different parts can be described as follows:

6.1.1 Word Encoder

The words in any given sentence are embedded to vectors through an embedding matrix. Annotations of these words are obtained using a Bi-GRU, from both directions, thereby incorporating contextual information in the annotation.

6.1.1 Word Attention

An attention mechanism is used to extract the contributions of each word to the sequence meaning. The representation of those informative words are then aggregated to form a sentence vector.

6.1.1 Sentence Attention

An attention mechanism, again is used to extract contribution of each sentence to the document meaning and rewards those highly that have a strong correspondence with the document meaning.

6.2 Preprocessing and Setup

The preprocessing is similar to the previous approaches. However, for this task, the GloVe 840B 300 Dimensional dataset is used.

6.3 Model

The Model in this network has two distinctive parts as seen below. Of particular interest here is the use of the Spatial Dropout Layer which is different from the regular dropout layer in use. Spatial dropout takes into account that the connections are a bit special when performing an encoding task. This seems to be in sync with the idea of our Attention based model. Another unique layer being used is the Time Distributed layer in the Sentence level Attention Model. Time Distributed Layer is a dense layer used in RNNs, including LSTMs to keep one-to-one correspondence on the input and output.

6.4 Improvements: Data Augmentation

On investigating the results from the approach discussed here, it is found that the model overfits at the end of the second epoch itself. This could be because Deep Sequential models are data hungry and need a lot of data to train well. This problem is addressed by providing similar contextual data by translating the original English data to another language and translating it back to English. This method is applied with three different languages, in accordance with Pavel Otsyakov's Machine Translation module based on TextBlob. The languages chosen are: Spanish, French and German [3]. The textBlob library uses Google Translate API. The usage of this

method helps train the model better and the training loss goes below validation loss at the end of the third epoch. Additionally, a highly improved training and validation accuracy is obtained

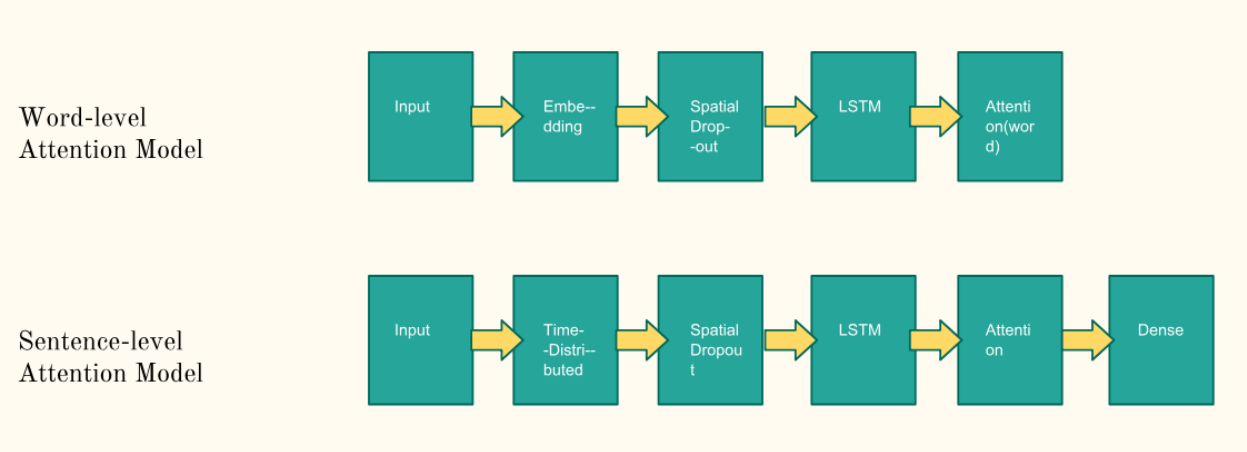


Fig 7: Model used in this task

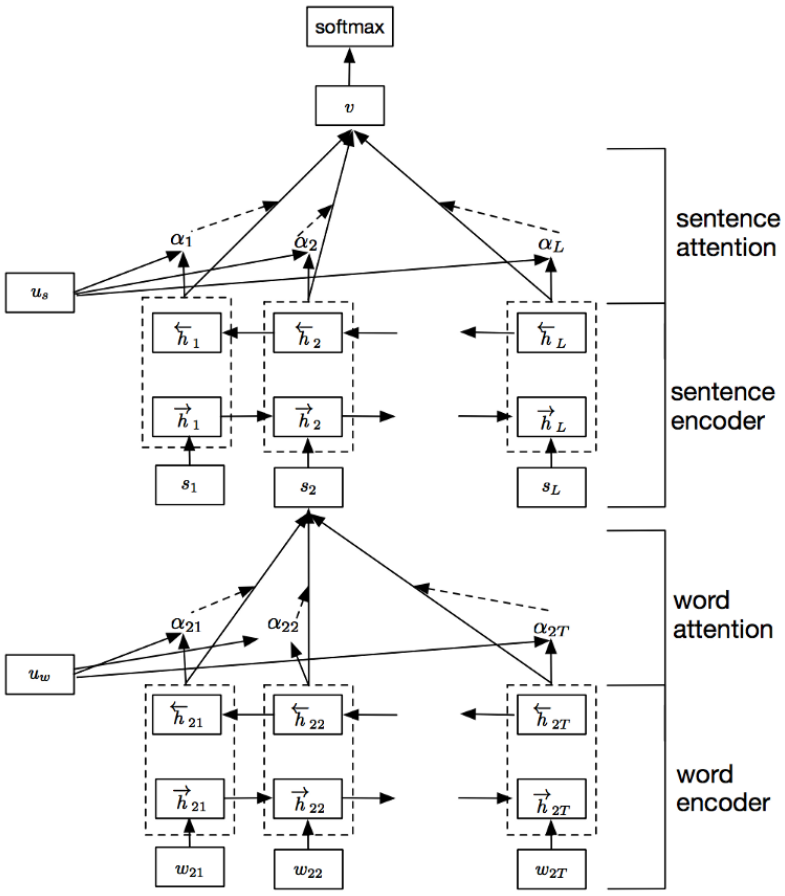


Fig 8: Hierarchical Attention Network

7 Code

The Code has been included as an appendix to this report. An updated version of the code can be found on my GitHub Profile: https://github.com/deepandas11

8 Results

The results have been summarized well in the following tables.

Method	**Metric: Kaggle Score**
Logisic Regression using TF-IDF features	0.9761

Method	**Tra_Loss**	**Val_Loss**	**Tra_Accuracy**	**Val_Accuracy**
Bi-LSTM+Random Embedding	0.0448	0.0489	97.01%	96.89%
Bi-LSTM+GloVe Embedding	0.0384	0.0455	98.50%	98.34%
Bi-LSTM+GloVe+Data Augmentation	0.0232	0.0298	99.11%	98.91%
Hierarchical Attention Network(HAN) + GloVe	0.0478	0.0546	98.28%	98.11%
HAN+Data Augmentation	0.0241	0.0340	99.16%	98.90%

The results can be shown graphically as well:

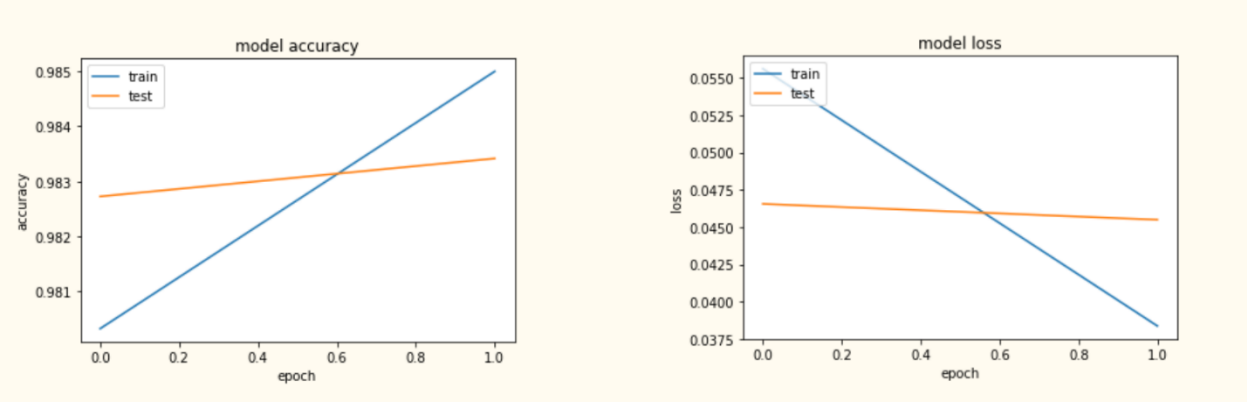


Fig9: Training curves pertaining to **BiLSTM+GloVe** model

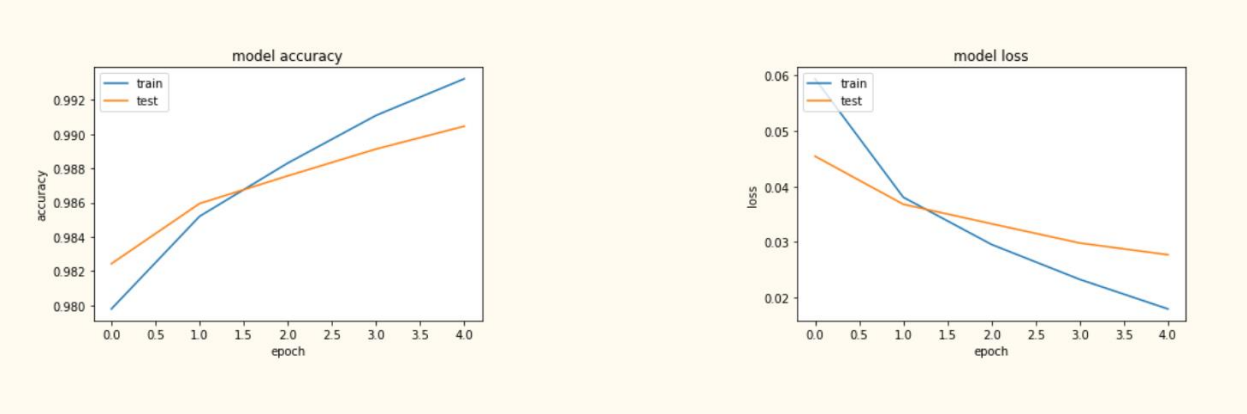


Fig10: Training curves pertaining to **BiLSTM+GloVe+Data Augmentation** model

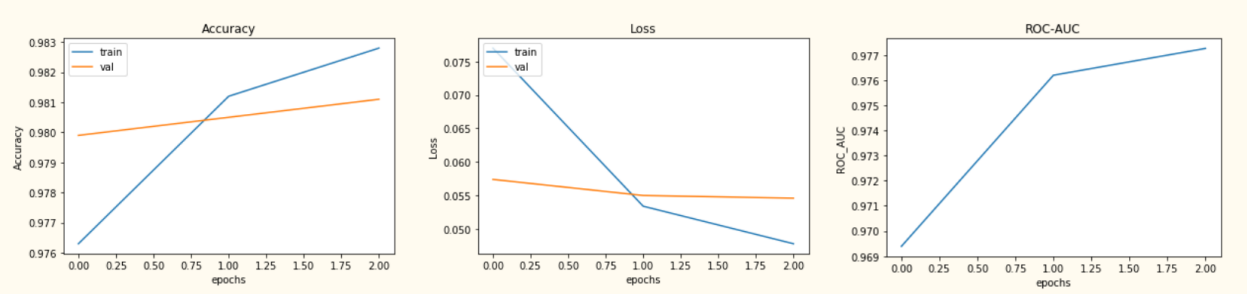


Fig11: Training curves pertaining to the **HAN+GloVe** model

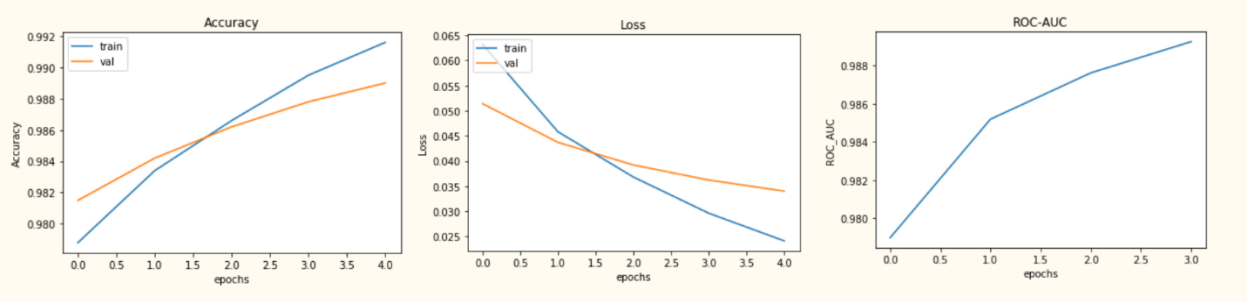


Fig12: Training curves pertaining to the **HAN+GloVe+Data Augmentation** model

9 Conclusion

We notice that Data Augmentation helps us train the network better. The model does not overfit as soon as when it is not being used. Based on observations made by the winning participants, a possible case could be the use of English to English translation, but it fails to effect considerable change. The use of GloVe Embedding has also proved to be advantageous as it clusters similar words together. This accomplishes the task of generating a fair bit of context. The use of a Hierarchical Attention Network seems logical as there are comments with more than one sentence, but such comments are not a lot in number, that may be the cause of no major

improvement in the performance of the HAN network for this problem. Therefore, some of the major conclusions to be drawn from this project could be the fundamental principle that Deep Learning models are really data-hungry. With the use of a data augmentation module, the accuracy jumps up by a lot and it goes to show, how even slightly different data can be useful in augmenting the existing dataset. Moreover, we can observe that model complexity doesn't really matter much for this classification task. The fact that the simple Bidirectional LSTM model with GloVe Embedding can dish out the best validation accuracy stands testimony to this fact. Lastly, the use of a suitable Embedding layer has also helped improve the accuracy by a huge margin. The use of Random Embedding was not justified and the advantages of using a well-clustered embedding initialization is a well-studied field and has been found to work better in a large number of applications and tools.

Acknowledgments

I would like to thank Prof. Yu Hen Hu for having provided us with an opportunity to undertake such a project independently. He was readily available to provide any kind of technical guidance and expertise whenever needed. Moreover, I would like to extend my gratefulness to the brilliant teams on Kaggle that came up with such wonderful solutions and let us use their ideas to build on them.

References

1. "Toxic Comment Classification Challenge | Kaggle." *Kaggle.com*. N. p., 2018. Web. 3 Dec. 2018.
2. Pennington, Jeffrey. "Glove: Global Vectors For Word Representation." Nlp.stanford.edu. N. p., 2018. Web. 3 Dec. 2018.
3. "A Simple Technique For Extending Dataset | Kaggle." Kaggle.com. N. p., 2018. Web. 3 Dec. 2018.
4. Yang, Zichao, et al. "Hierarchical attention networks for document classification." Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. 2016.
5. "Understanding LSTM Networks -- Colah's Blog." Colah.github.io. N. p., 2018. Web. 3 Dec. 2018.
6. Hochreiter, Sepp, et al. "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies." (2001).

Appendix

Code

1. Bi-LSTM + Data Augmentation

```
import sys,os, re, csv, codecs
import numpy as np
import pandas as pd
import keras

from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.layers import Dense, Input, LSTM, Embedding, Dropout, Activation
from keras.models import Model
from keras.layers import Bidirectional, GlobalMaxPool1D
from keras import initializers, regularizers, constraints, optimizers,layers
from time import time

train_file_en= '../input/jigsaw-toxic-comment-classification-challenge/train.csv'
train_file_de= '../input/toxic-comments-french-spanish-german-train/train_de.csv'
train_file_fr= '../input/toxic-comments-french-spanish-german-train/train_fr.csv'
train_file_es= '../input/toxic-comments-french-spanish-german-train/train_es.csv'

test_file= '../input/jigsaw-toxic-comment-classification-challenge/test.csv'
test_label_file = '../input/jigsaw'

embedding_file = '../input/glove6b200d/glove.6B.200d.txt'

train_en = pd.read_csv(train_file_en)
train_es = pd.read_csv(train_file_es)
train_fr = pd.read_csv(train_file_fr)
train_de = pd.read_csv(train_file_de)

train    =    train_en.append(train_es.append(train_de.append(train_fr,    ignore_index=    True),
ignore_index=True), ignore_index=True)
test = pd.read_csv(test_file)

embed_size = 200
max_features = 200000
maxlen = 100

list_sentences_train = train["comment_text"].fillna("_na_").values
list_sentences_test = test["comment_text"].fillna("_na_").values

list_classes = ["toxic","severe_toxic", "obscene", "threat", "insult", "identity_hate"]
y_t = train[list_classes].values

tokenizer = Tokenizer(num_words=max_features)

tokenizer.fit_on_texts(list(list_sentences_train))

list_tokenized_train = tokenizer.texts_to_sequences(list_sentences_train)
list_tokenized_test = tokenizer.texts_to_sequences(list_sentences_test)

X_t = pad_sequences(list_tokenized_train, maxlen=maxlen)
X_te = pad_sequences(list_tokenized_test, maxlen=maxlen)
```

```
word_index = tokenizer.word_index
print('Found %s unique tokens.' %len(word_index))

embeddings_index={}

f = open(embedding_file)
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:],dtype='float32')
    embeddings_index[word] = coefs

f.close()
print('Found %s word vectors.'%len(embeddings_index))

all_embeddings = np.stack(embeddings_index.values())

emb_mean= all_embeddings.mean()
emb_stddev = all_embeddings.std()

nb_words = min(max_features, len(word_index)+1)
embedding_matrix = np.random.normal(emb_mean, emb_stddev, (nb_words, embed_size))
for word, i in word_index.items():
    if i>=max_features: continue
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector

inp = Input(shape=(maxlen,))
x = Embedding(max_features, embed_size, weights = [embedding_matrix])(inp)
x = Bidirectional(LSTM(50, return_sequences=True, dropout=0.1, recurrent_dropout=0.1))(x)
x = GlobalMaxPool1D()(x)
x = Dense(50, activation="relu")(x)
x = Dropout(0.1)(x)
x = Dense(6, activation="sigmoid")(x)
model = Model(inputs=inp, outputs=x)
model.compile(loss='binary_crossentropy', optimizer = 'adam', metrics = ['accuracy'])

history = model.fit(X_t, y_t, batch_size=50, epochs=10, validation_split=0.3);

y_test = model.predict([X_te],batch_size = 1024, verbose=1)
sample_submission           =           pd.read_csv('../input/jigsaw-toxic-comment-classification-
challenge/sample_submission.csv')
sample_submission[list_classes] = y_test
sample_submission.to_csv('submission.csv',index = False)
```

2. Hierarchical Attention Network

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
np.random.seed(42)
# Input data files are available in the "../input/" directory.

import os
import sys
from keras import backend as K
```

```
from keras.layers import Dense,Input, LSTM, Bidirectional, Embedding, TimeDistributed,
SpatialDropout1D
from keras.preprocessing import text, sequence
from keras import initializers, regularizers, constraints, optimizers, layers, callbacks
from keras.callbacks import EarlyStopping, ModelCheckpoint, Callback
from keras.models import Model
from keras.optimizers import Adam
from keras import losses
from keras import initializers as initializers, regularizers, constraints
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score,roc_auc_score
import nltk
import re
from keras.engine.topology import Layer

import warnings
warnings.filterwarnings('ignore')

print(os.listdir("../input"))

# Any results you write to the current directory are saved as output.

EMBEDDING_FILE = '../input/glove840b300dtxt/glove.840B.300d.txt'
train = pd.read_csv('../input/jigsaw-toxic-comment-classification-challenge/train.csv')
test = pd.read_csv('../input/jigsaw-toxic-comment-classification-challenge/test.csv')

train["comment_text"].fillna("fillna")
test["comment_text"].fillna("fillna")
X_train = train["comment_text"].str.lower()
y_train = train[["toxic", "severe_toxic", "obscene", "threat", "insult", "identity_hate"]].values

X_test = test["comment_text"].str.lower()

X_train = list(X_train)
X_test = list(X_test)

def remove_noise(input_text):
    text = re.sub('\(talk\)(.*)\(utc\)','',input_text)
    text = text.split()
    text = [re.sub('[\d]+','',x) for x in text]
    return ' '.join(text)


for i in range(len(X_train)):
    X_train[i] = remove_noise(X_train[i])
for i in range(len(X_test)):
    X_test[i] = remove_noise(X_test[i])

def replace_word(X):
    repl = {
        "&lt;3": " good ",":d": " good ",":dd": " good ",":p": " good ","8)": " good ",":-)": "
good ", ":)": " good ",";)": " good ",
        "(-:": " good ","(:": " good ","yay!": " good ","yay": " good ","yaay": " good ","yaaay":
" good ","yaaaay": " good ",
        "yaaaaay": " good ",":/": " bad ",":&gt;": " sad ",":')": " sad ",":-(": " bad ",":(": "
bad ", ":s": " bad ",":-s": " bad ",
        "&lt;3": " heart ",":d": " smile ",":p": " smile ",":dd": " smile ","8)": " smile ", ":-
)": " smile ", ":)": " smile ",
        ";)": " smile ","(-:": " smile ","(:": " smile ",":/": " worry ",":&gt;": " angry ",
":')": " sad ",":-(": " sad ",":(": " sad ",
        ":s": " sad ", ":-s": " sad ",r"\br\b": "are",r"\bu\b": "you",r"\bhaha\b":
"ha",r"\bhahaha\b": "ha",r"\bdon't\b": "do not",
```

```
        r"\bdoesn't\b":     "does     not",r"\bdidn't\b":     "did     not",r"\bhasn't\b":     "has
not",r"\bhaven't\b": "have not",r"\bhadn't\b": "had not",
        r"\bwon't\b":     "will     not",r"\bwouldn't\b":     "would     not",r"\bcan't\b":     "can
not",r"\bcannot\b": "can not",r"\bi'm\b": "i am",
        "m": "am","r": "are","u": "you","haha": "ha","hahaha": "ha","don't": "do not","doesn't":
"does not","didn't": "did not",
        "hasn't":   "has   not","haven't":   "have   not","hadn't":   "had   not","won't":   "will
not","wouldn't": "would not","can't": "can not",
        "cannot": "can not","i'm": "i am","m": "am","i'll" : "i will","its" : "it is","it's" :
"it is","'s" : " is","that's" : "that is",
        "weren't" : "were not"
    }
    keys = repl.keys()
    new_X = []
    for i in X:
        arr = str(i).split()
        xx = ""
        for j in arr:
            j = str(j).lower()
            if j[:4] == 'http' or j[:3] == 'www':
                continue
            if j in keys:
                j = repl[j]
            xx += j + " "
        new_X.append(xx)
    return new_X

X_train = replace_word(X_train)
X_test = replace_word(X_test)

max_features=200000
max_senten_len=30
max_senten_num=10
embed_size=300

def filt_sent(X,max_senten_num):
    X_sent = []
    sent_tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')
    for paragraph in X:
        raw = sent_tokenizer.tokenize(paragraph)
        filt = []
        min_sent_len = 5 if len(raw) <= 10 else 10
        for sentence in raw:
            if len(sentence.split()) >= min_sent_len and len(filt) < max_senten_num:
                filt.append(sentence)
        while len(filt) < max_senten_num:
            filt.append('nosentence')
        X_sent.append(filt)
    return X_sent


X_train_sent = filt_sent(X_train ,max_senten_num)
X_test_sent = filt_sent(X_test, max_senten_num)
tok=text.Tokenizer(num_words=max_features,lower=True)
tok.fit_on_texts(list(X_train)+list(X_test))
for i in range(len(X_train_sent)):
        X_train_sent[i] = tok.texts_to_sequences(X_train_sent[i])
        X_train_sent[i] = sequence.pad_sequences(X_train_sent[i],maxlen=max_senten_len)
for i in range(len(X_test_sent)):
        X_test_sent[i] = tok.texts_to_sequences(X_test_sent[i])
        X_test_sent[i] = sequence.pad_sequences(X_test_sent[i],maxlen=max_senten_len)
embeddings_index = {}
```

```
with open(EMBEDDING_FILE,encoding='utf8') as f:
    for line in f:
        values = line.rstrip().rsplit(' ')
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs
word_index = tok.word_index
#prepare embedding matrix
num_words = min(max_features, len(word_index) + 1)
embedding_matrix = np.zeros((num_words, embed_size))
for word, i in word_index.items():
    if i >= max_features:
        continue
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector def dot_product(x, kernel):
    """
    Wrapper for dot product operation, in order to be compatible with both
    Theano and Tensorflow
    Args:
        x (): input
        kernel (): weights
    Returns:
    """
    if K.backend() == 'tensorflow':
        return K.squeeze(K.dot(x, K.expand_dims(kernel)), axis=-1)
    else:
        return K.dot(x, kernel)

class AttentionWithContext(Layer):
    """
    Attention operation, with a context/query vector, for temporal data.
    Supports Masking.
    Follows the work of Yang et al. [https://www.cs.cmu.edu/~diyiy/docs/naacl16.pdf]
    "Hierarchical Attention Networks for Document Classification"
    by using a context vector to assist the attention
    # Input shape
        3D tensor with shape: `(samples, steps, features)`.
    # Output shape
        2D tensor with shape: `(samples, features)`.
    How to use:
    Just put it on top of an RNN Layer (GRU/LSTM/SimpleRNN) with return_sequences=True.
    The dimensions are inferred based on the output shape of the RNN.
    Note: The layer has been tested with Keras 2.0.6
    Example:
        model.add(LSTM(64, return_sequences=True))
        model.add(AttentionWithContext())
        # next add a Dense layer (for classification/regression) or whatever...
    """

    def __init__(self,
                 W_regularizer=None, u_regularizer=None, b_regularizer=None,
                 W_constraint=None, u_constraint=None, b_constraint=None,
                 bias=True, **kwargs):

        self.supports_masking = True
        self.init = initializers.get('glorot_uniform')

        self.W_regularizer = regularizers.get(W_regularizer)
        self.u_regularizer = regularizers.get(u_regularizer)
        self.b_regularizer = regularizers.get(b_regularizer)
```

```
        self.W_constraint = constraints.get(W_constraint)
        self.u_constraint = constraints.get(u_constraint)
        self.b_constraint = constraints.get(b_constraint)

        self.bias = bias
        super(AttentionWithContext, self).__init__(**kwargs)

    def build(self, input_shape):
        assert len(input_shape) == 3

        self.W = self.add_weight((input_shape[-1], input_shape[-1],),
                                 initializer=self.init,
                                 name='{}_W'.format(self.name),
                                 regularizer=self.W_regularizer,
                                 constraint=self.W_constraint)
        if self.bias:
            self.b = self.add_weight((input_shape[-1],),
                                     initializer='zero',
                                     name='{}_b'.format(self.name),
                                     regularizer=self.b_regularizer,
                                     constraint=self.b_constraint)

        self.u = self.add_weight((input_shape[-1],),
                                 initializer=self.init,
                                 name='{}_u'.format(self.name),
                                 regularizer=self.u_regularizer,
                                 constraint=self.u_constraint)

        super(AttentionWithContext, self).build(input_shape)

    def compute_mask(self, input, input_mask=None):
        # do not pass the mask to the next layers
        return None

    def call(self, x, mask=None):
        uit = dot_product(x, self.W)

        if self.bias:
            uit += self.b

        uit = K.tanh(uit)
        ait = dot_product(uit, self.u)

        a = K.exp(ait)

        # apply mask after the exp. will be re-normalized next
        if mask is not None:
            # Cast the mask to floatX to avoid float64 upcasting in theano
            a *= K.cast(mask, K.floatx())

        # in some cases especially in the early stages of training the sum may be almost zero
        # and this results in NaN's. A workaround is to add a very small positive number ε to
the sum.
        # a /= K.cast(K.sum(a, axis=1, keepdims=True), K.floatx())
        a /= K.cast(K.sum(a, axis=1, keepdims=True) + K.epsilon(), K.floatx())

        a = K.expand_dims(a)
        weighted_input = x * a
        return K.sum(weighted_input, axis=1)
```

```
    def compute_output_shape(self, input_shape):
        return input_shape[0], input_shape[-1]

class RocAucEvaluation(Callback):
    def __init__(self, validation_data=(), interval=1):
        super(Callback, self).__init__()

        self.interval = interval
        self.X_val, self.y_val = validation_data

    def on_epoch_end(self, epoch, logs={}):
        if epoch % self.interval == 0:
            y_pred = self.model.predict(self.X_val, verbose=0)
            score = roc_auc_score(self.y_val, y_pred)
            print("\n ROC-AUC - epoch: %d - score: %.6f \n" % (epoch+1, score))
embedding_layer = Embedding(max_features,
                            embed_size,
                            input_length=max_senten_len,
                            weights=[embedding_matrix])

word_input = Input(shape=(max_senten_len,), dtype='int32')
word = embedding_layer(word_input)
word = SpatialDropout1D(0.2)(word)
word = Bidirectional(LSTM(128, return_sequences=True))(word)
word_out = AttentionWithContext()(word)
wordEncoder = Model(word_input, word_out)

sente_input = Input(shape=(max_senten_num, max_senten_len), dtype='int32')
sente = TimeDistributed(wordEncoder)(sente_input)
sente = SpatialDropout1D(0.2)(sente)
sente = Bidirectional(LSTM(128, return_sequences=True))(sente)
sente = AttentionWithContext()(sente)
preds = Dense(6, activation='sigmoid')(sente)
model = Model(sente_input, preds)
opt = Adam(clipnorm=5.0)

model.compile(loss='binary_crossentropy',
              optimizer=opt,
              metrics=['acc'])

X_train_sent = np.asarray(X_train_sent)
X_test_sent = np.asarray(X_test_sent)
print('Shape of data tensor:', X_train_sent.shape)

batch_size = 256
epochs = 3
X_tra,  X_val,  y_tra,  y_val  =  train_test_split(X_train_sent,  y_train,  train_size=0.95,
random_state=233)
RocAuc = RocAucEvaluation(validation_data=(X_val, y_val), interval=1)
filepath="weights_base.best.hdf5"
checkpoint  =  ModelCheckpoint(filepath,  monitor='val_acc',  verbose=1,  save_best_only=True,
mode='max', save_weights_only=True)
early = EarlyStopping(monitor="val_acc", mode="max", patience=5)
callbacks_list = [checkpoint, early, RocAuc]

model.fit(X_tra, y_tra, batch_size=batch_size,
y_pred  =  model.predict(x_test,batch_size=1024,verbose=1)epochs=epochs,  validation_data=(X_val,
y_val),callbacks = callbacks_list, verbose=1)
submission              =              pd.read_csv('../input/jigsaw-toxic-comment-classification-
challenge/sample_submission.csv')
submission[["toxic", "severe_toxic", "obscene", "threat", "insult", "identity_hate"]] = y_pred
submission.to_csv('submission.csv', index=False)
```