UNIVERSITY OF CALIFORNIA

Los Angeles

Application of Recurrent Neural Networks In Toxic Comment Classification

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Applied Statistics

by

Siyuan Li

2018

ABSTRACT OF THE THESIS

Application of Recurrent Neural Networks
In Toxic Comment Classification

by

Siyuan Li
Master of Applied Statistics
University of California, Los Angeles, 2018
Professor Yingnian Wu, Chair

Moderators of online discussion forums often struggle with controlling extremist comments on their platforms. To help provide an efficient and accurate tool to detect online toxicity, we apply word2vec's Skip-Gram embedding vectors, Recurrent Neural Network models like Bidirectional Long Short-term Memory to tackle a toxic comment classification problem with a labeled dataset from Wikipedia Talk Page. We explore different pre-trained embedding vectors from larger corpora. We also assess the class imbalance issues associated with the dataset by employing sampling techniques and penalizing loss. Models we applied yield high overall accuracy with relatively low cost.

The thesis of Siyuan Li is approved.

Nicolas Christou

Qing Zhou

Yingnian Wu, Committee Chair

University of California, Los Angeles

2018

TABLE OF CONTENTS

List of Figures

List of Tables

1 Introduction

Internet negativity has always been a hot topic. The anonymity and the sense of distance of people's internet presence have encouraged people to express themselves freely. This freedom can sometimes lead to extreme outtakes on others people or the particular topics. Extreme negativities has sometimes stopped people from expressing themselves or made them give up looking for different opinions online[1]. Issues like this happen almost all the time, across all platforms of discussion, and the modulators of these platforms have limited capabilities dealing with it. Needless to say the time, energy and effort these modulators have to put into controlling this negativity on their platform. People have been seeking help from various tools to analyze text-based information so that they can identify toxic expressions from a sea of information, both efficiently, and more importantly, accurately.

Natural language processing with deep Neural Networks is one of the most influential tools that enable researchers to extract, analyze, and classify essential features from text-based information. We see tremendous interests and development in utilizing deep learning in sentiment[2] and semantic analysis[3], text generation[4], machine translation[5], speech recognition[6] and so much more over the recent years.

In this thesis, we will be applying word embedding techniques and recurrent neural network to perform text classification on a multi-label text dataset to identify different forms of internet toxicity.

2 Problem Definition

The data we will be focusing on is a public dataset provided by the Conversation AI team; a research initiate co-founded by Jigsaw and Google. Jigsaw is a technology incubator created by Google, with the primary objective to "use technology to tackle the toughest geopolitical challenges, from countering violent extremism to thwarting online censorship to mitigating the threats associated with digital attacks."[7] Jigsaw and Google launched Perspective API in February 2017, a free tool that utilizes machine learning to identify toxic comments.[8] To improve the performance of the Perspective API, and with the belief that "collaborative problem-solving yields the best solutions"[7], the Conversation AI team hosted a "Wikipedia Talk Page Comments annotated with toxicity reasons" Kaggle competition[9]. We will be building our deep learning classification model and monitor its performance base on the dataset provided in this competition.

Currently, the model used by Conversation API performs quite well, able to provide a relatively accurate toxic score given text comments. However, the team mentioned that their model still makes errors[10]; it is unable to classify toxicity if the model has not seen the pattern before, and it may miss-classify texts that share similar patterns as toxic comments. Recurrent neural networks' ability to process sequences of documents and analyze contexts may prove useful in resolving the problems Conversation API currently encounters.

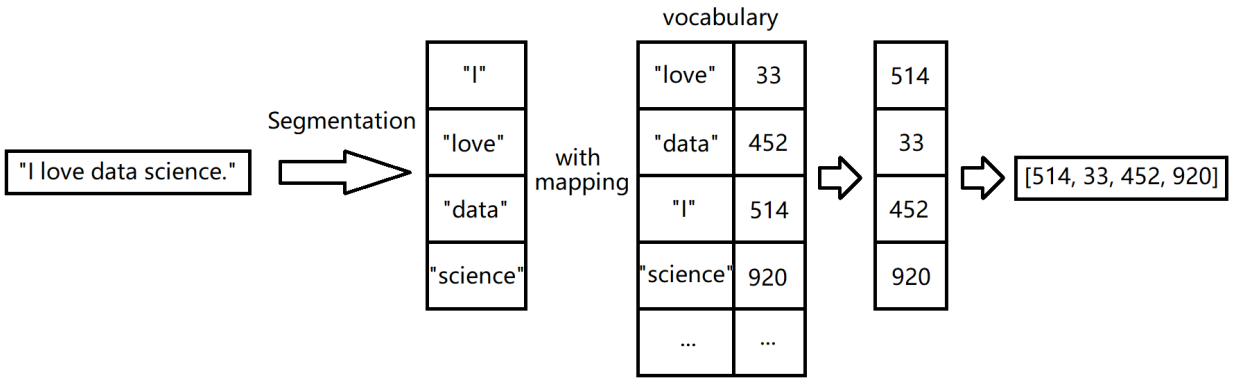
3 Methodologies

This section will emphasize on providing detailed explanations on the general methods that will be employed in later sections. Methodologies of our approach can be summarized in three section: word segmentation, embedding, and recurrent neural network. Model-specific components such as activation functions and optimizers will be elaborated in later sections.

3.1 Word Segmentation

Text data is a perfect example of unstructured data. To efficiently translate this unstructured data into machine-interpretable information, we separate chunks of continuous text data into a list of words, then encode them into numerical vectors. We then encode each unique word to its numerical representation. As shown in Figure 1, mapping this tokenization to segmented text data essentially returns the text as a list of numeric factors, which represent involved words in the vocabulary. This brief explanation serves to provide foundations to the "word2vec" method we will employ.

Figure 1: Tokenization Example



3.2 "word2vec"

Created by a team of researchers from Google in 2013, word2vec is a group of models to produce word embeddings that retains the context of words [11]. Word2vec models are shallow, two-layer neural networks constructed based on the idea that similar words would appear in similar positions in the context.

Representing this intuition is Cosine Similarity. Similarities of the given vectors are measured

in their inner product space by the cosine angle between them.

$$\cos(\theta) = \frac{u \cdot v}{||u||||v||}$$

Which in turn, means that larger dot product: $u \cdot v = ||u||||v|| \cos(\theta)$, indicates more similarity. Word2vec introduced two approaches in calculating the word embedding so that similar word vectors have higher dot product.

3.2.1 Skip-Gram

Skip-Gram model was first introduced by Mikolov et al.[11] in 2013. The skip-gram approach is to learn neighboring word vector representations based on a center word. Figure 2 [12] shows the sample extraction method of skip-gram. In this example, a "window" of size 2 is chosen, which means two word in the front of the center word and two words after the center word will be selected to pair up with the center word. The objective function of this skip-gram method is then defined to maximize the probability of any context word given the current center word:

$$L(\theta) = \prod_{t=1}^{T} \prod_{-m \leq j \leq m, j \neq 0} P(w_{t+j}|w_t; \theta)$$

Which translate to minimizing the negative log likelihood of any context word given the current center word:

$$Loss(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \leq j \leq m, j \neq 0} \log P(w_{t+j}|w_t; \theta)$$

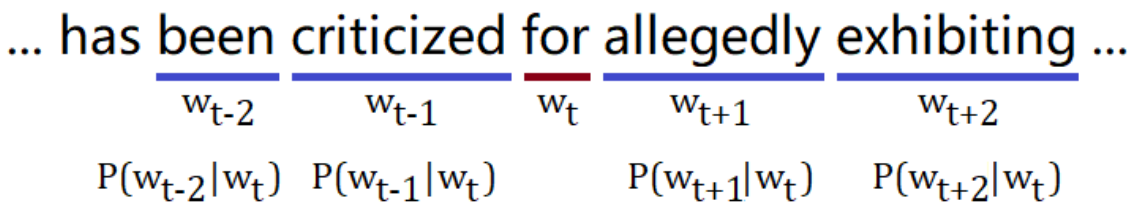
In these functions, t is the position of the center word, with a maximum of T; m is the defined window size; θ represents all the variables to be optimized in the function. The probability $P(w_{t+j}|w_t; \theta)$ is defined using a softmax function which involves the dot products of context

words and center word:

$$P(w_O|w_I) = \frac{exp(v'^T_{w_O} v_{w_I})}{\sum_{w=1}^{W} exp(v'^T_w v_{w_I})}$$

Where v' and v denotes the output and input vector representation of w; W is the number of words in the vocabulary.

Figure 2: Skip-gram Window Example[12]



3.3 Stochastic Gradient Descent

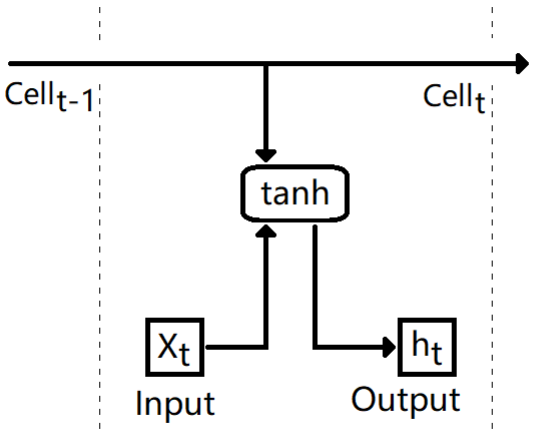
When training a neural network with back-propagation, we use gradient descent to update the parameters by finding the minimum of the loss functions. Suppose we have loss function $Loss(\theta)$ where θ is the parameters to be optimized using the whole training dataset. And $L(\theta) = \frac{1}{n}\sum_{i=1}^{n} Loss_i(\theta)$, which is the loss averaged over the whole training dataset. Gradient descent, is then defined as: $\theta_{t+1} = \theta_t - \eta L'(\theta_t)$.

However, this algorithm is not always efficient, since we have to sum over all the training samples. Instead, we substitute η with η_t, a mini-batch step at step t. η_t is called step size or learning rate. The gradient descent is then $\theta_{t+1} = \theta_t - \eta_t Loss'_i(\theta_t)$, which we call it stochastic gradient descent (SGD).

3.4 Recurrent Neural Network

Unlike a general feed-forward neural network, each node in a Recurrent Neural Network (RNN) has a memory component to process sequences of inputs. A recurrent neural net can be trained to compress previous inputs in low-dimensional space and are better at handling position invariance problems. This structure allows RNN to be highly efficient in processing text data where the order and structure of the inputs are essential.

Figure 3: Recurrent Neural Network



3.4.1 Long Short-term Memory

First introduced by Hochreiter et al.[13] in 1997, Long Short-term Memory (LSTM) has become one of the fundamental components of tools like language translations and voice assistants. LSTM models can avoid vanishing gradient problem, where long-term dependencies issues in RNN cause weights from the neural network cannot update its value because the gradient is becoming trivially small.

A general equation form of LSTM can be shown as:

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$

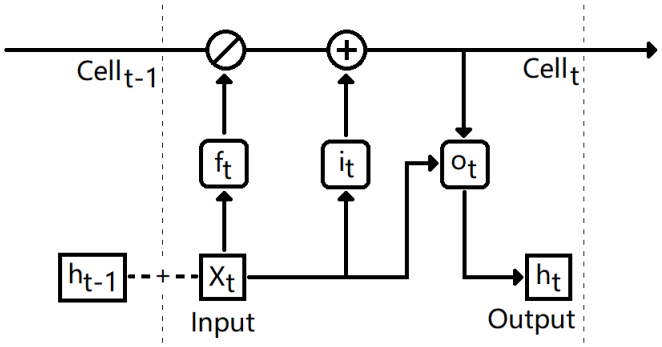
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$

$$C_t = f_t * C_{t-1} + i_t * \tanh(W_c x_t + U_c h_{t-1} + b_c)$$

$$h_t = o_t * \sigma_h(C_t)$$

In above equations, x_t is the input vector of the LSTM cell. i_t, f_t, o_t is the input gate, forget gate, and output gate activation vector respectively; they use σ_g, Sigmoid activation function. Cell state C_t is an important component in an LSTM cell; it allows LSTM to pass down processed historical states. Finally, h_t is the output vector of the LSTM cell unit. Note that W, U matrices, and b vector are weights to be learned in training of this cell. This b also stands for bias vector.

Figure 4: Detailed Structure of a LSTM Cell

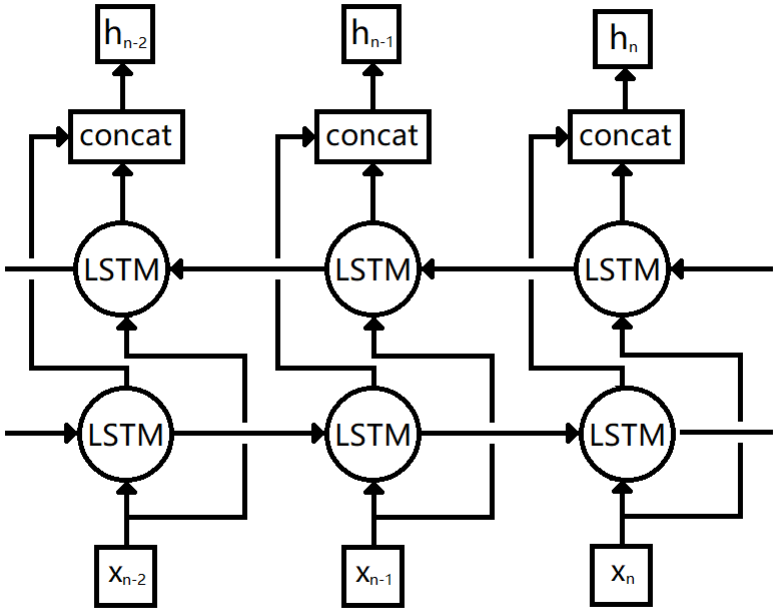


As input x_t and h_{t-1} was feed into the LSTM cell, first, forget gate f_t decides what values in cell state C_{t-1} to discard. Then x_t and h_t are passed to i_t to see what new values to store in current cell state C_t. After that, the output gate o_t computes what information to be outputted. During this process, weights W, U and b are continuously updated to minimize loss function.[14] Typical back-propagation method used in this trainig process is SGD.

3.4.2 Bidirectional Long Short-term Memory

Bidirectional Recurrent Neural Network was first introduced by Schuster et al.[15] in 1997. It increases the amount of input for a neural network. As the name suggests, the model not only takes in information in previous states, it processes data from both past and future states, which further enhances the neural network's ability to understand the context of the input. Comparing to a standard LSTM layer, a bidirectional LSTM adds another set of LSTM cells to process inputs in a reversed sequence. Outputs in both sets of cells are concatenated and feed to the next layer.

Figure 5: Bidirectional LSTM



4 Data Treatment

This section will go over the background, structure, and issues with the dataset we were set to investigate. In addition to brief exploratory analysis, data cleaning, text pre-processing will also be applied. Finally, we will build embedding vectors using Word2Vec algorithms discussed in the Methodology section.

4.1 Data Structure

"Wikipedia Talk Page Comments annotated with toxicity reasons" is a crowd-sourced dataset[10] including approximately 160,000 manually labeled comments from Wikipedia Talk Pages. Conversation AI team asked 5000 crowd-workers to read and label these comments based on the reasons why they think these comments would make others leave the discussion.

These labels were summarized into a total of six classes:

- Toxic: a general classification of the toxicity of the comment;
- Severe Toxic: extreme toxicity;
- Obscene: indecent language;
- Threat: statements with the intention to inflict hostile action;
- Insult: disrespect or verbal abuse;
- Identity Hate: sexism, racism, homophobic, etc..

As shown in Figure 6, these categories were individually labeled, meaning that comments were binary labeled in each class. And, one comment can have multiple labels attached to it.

Note that for further illustrations of the comment texts, offensive words will be blurred. Word-cloud plot is an excellent visualization for displaying word frequency. Due to the nature of our dataset, a word-cloud plot will not be shown.

Figure 6: Sample of the Data Structure

id	comment_text	toxic	severe_toxic	obscene	threat	insult	identity_hate
1082d5496191c27e	Blackpearll4 - calls herself evidently Mrs. Johnny Depp and Pirate lord-ess and such. She celebrates on her...	0	0	0	0	0	0
1083f46d76f93ffe	"Wow. Somebody bolded one of his words to add emphesis, so you told him to ""calm the [illegible] down"" and create ...	1	0	1	0	0	0
10849a3171faa726	Actually the format of titling appears to be DL# on the cover in the orange bar at the top only. On both the cover...	0	0	0	0	0	0
1085bc5da9d62871	Your swearing and personal attacks towards me doesn't put you in a very good position either, especially since...	0	0	0	0	0	0
108615e304dfa37a	Was Jesus a [illegible]? Yes, for sure ,	1	0	1	0	1	0
1087212c87361e32	" The size of Wim Crusio's Wiki page As of December 22nd, 2010 the Wim Crusio Wiki page constituted 28,725 ...	0	0	0	0	0	0

4.2 Issues

One of the problems the dataset exhibits is class imbalance. As shown in Table 1, the total number of comments that were not labeled (in other words, clean) take up around 90% of the dataset. Class imbalance issue has proven to be very impactful in the model training process because the model can be overwhelmed by majority class instances which can cause the model to ignore minority classes[16]. Class imbalance in our dataset is likely due to the fact that Wikipedia is an openly editable collaborative platform so that these toxic comments only take up a small portion of discussions[1]. As explained by Ying Liu et al.[17] in a similar situation, the high cost of efforts of human labeling can also cause this imbalance issue. Usually, with a neural network, class imbalance can be sorted out by the model given enough passes through training data (epochs). We can employ sampling techniques to the input dataset or even penalize on the loss function if the model makes a wrong prediction.

Although there are no missing values present in the dataset, the text section is slightly

Figure 7: Toxicity Type

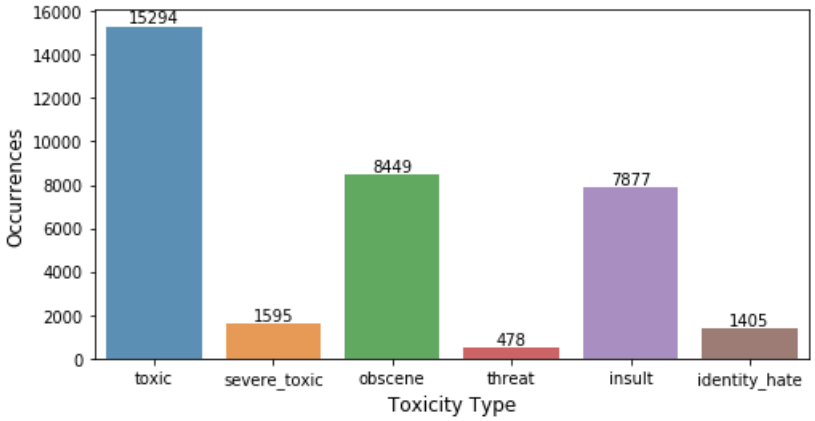


Table 1: Composition of Labeled and Unlabeled Rows

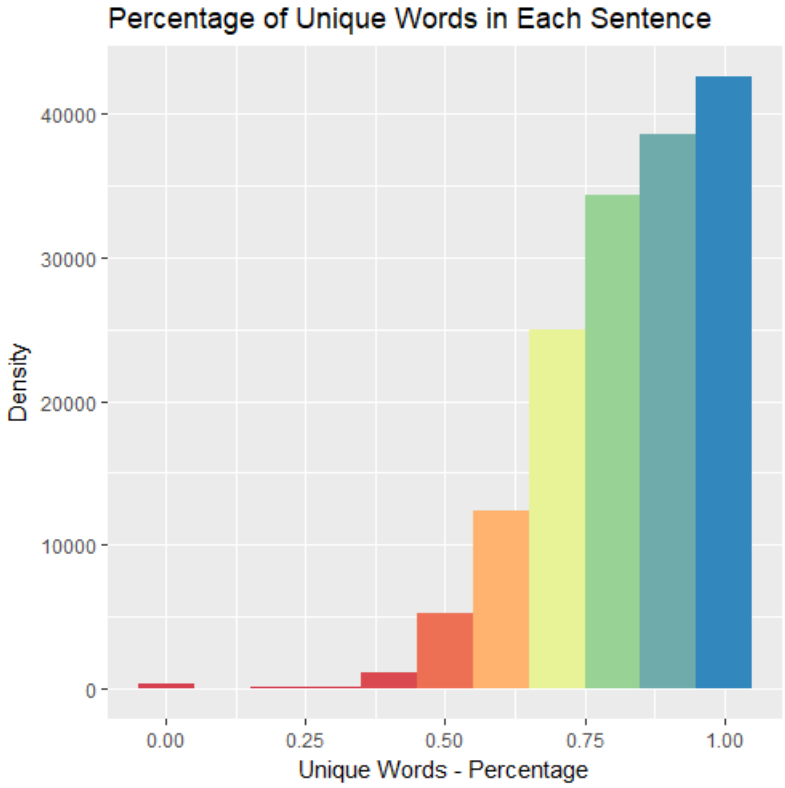
Clean	**Labeled**
143346	16225

polluted by spams, about 0.3% out of all the comments. Although their numbers are small, spams are harmful to both word embedding and model training. Firstly, with the increasing frequencies of words in the spams, embedding model is forced to take in words that are not meaningful. Secondly, spams fed into model training will cause unpredictable weight changes on these repeated sentences, potentially wasting training time and increase training cost. As shown in Figure 8, the majority of our comment texts has unique word percentage higher than 50 percent. After some exploration and testing, we will label comment texts with less than 30 percent unique words as spams. Treatments for these spam texts include removing all the repeated versions and keeps only the unique words.

4.3 Text Pre-processing and Segmentation

General text pre-processing steps are taken to clean the comment texts for embedding model training. All characters are converted to lower space. Numbers, punctuation and unnecessary

Figure 8: Percentage of Unique Words in Each Sentence



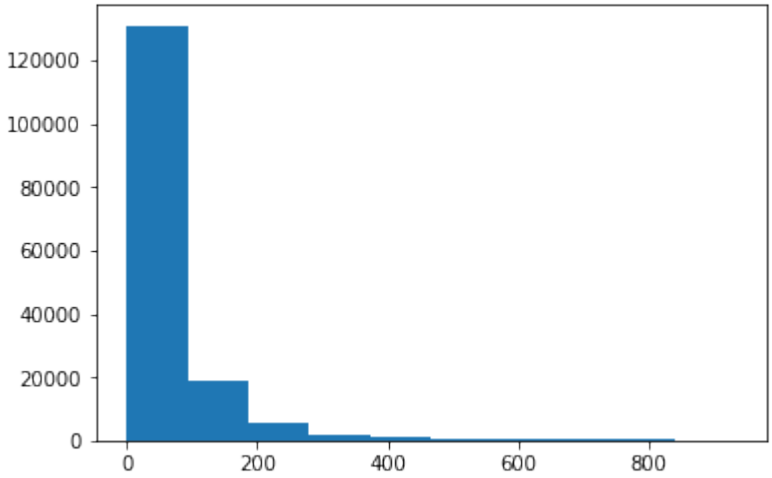
white spaces are removed since they provide no apparent meaning or context to our text. Stop words are removed from the comment text for the same reason. We us R to perform these text cleaning steps.

Segmentation is done using the tokenizer function in Keras framework with Tensorflow back-end in Python. We first define how many unique words to include in the vocabulary, then construct the tokenizer using the framework. We choose 20,000 words to keep the vocabulary small. The rule of selecting words to add to our vocabulary is to pick 20,000 words that have the highest frequencies in the comment text. Removing spam in our previous step helps to keep repeated spam words out of our vocabulary. Fitting the comment texts to the tokenizer returns a list of word vectors that consists of tokenized words in our comment text.

An input layer in a neural network takes in vectors of a set length. However, our lists of sequenced sentences vary in lengths. Exploratory analysis, as shown in Figure 9, indicates

that sentences in the comment text are generally less than 200 words; but 99% confidence interval shows that length of sentences falls in around 65. To give the training sample some headroom, we pad our sentence sequences to 100 words with trailing series of zeros. That is, we cut off sentences with longer than 100 words and put sequences of zeros after the shorter sentences. While the neural network will ignore the sequences of zeros, we may lose some, however negligible, information in our dataset.

Figure 9: Length of Segmented Sentences



4.4 Embedding

Word2Vec embedding is then employed using Gensim package in Python. Gensim uses Word2Vec algorithms written with C backend which are developed by Google. The Word2Vec model in Gensim takes in a list of lists of tokenized words as input, which we converted in the previous step. We also specify that we will only include words with a frequency above 10 in the lists of embedding vectors. We choose a common embedding size of 300 for the embedding vectors. We then train the Word2Vec embedding model using ten passes of our comment text data.

```
model_sg = gensim.models.Word2Vec(
  comment_token,window=2,min_count=10,size=300,workers=4,iter=10)
```

The returned embedding vectors show promising results. As shown in Figure 10, they can group words with similar meanings and functions with ease. Note that the similarity probability of each of these cases is not very high. It is likely because our corpus is very noisy, meaning we have a lot of different words with similar definitions and are used in a multitude of contexts, which is typical for comments on the internet. This noisiness can also indicate that our training samples and vocabulary are small. Using pre-trained embedding matrices with large corpora might prove beneficial.

Figure 10: Similarity Output of Skip-Gram Model

```
model_sg.wv.most_similar('wikipedia')
[('wp', 0.7156897783279419),
 ('wiki', 0.6534541845321655),
 ('wikepedia', 0.4830690324306488),
 ('wikis', 0.4307202696800232),
 ('wikipedias', 0.41485661268234253),
 ('encylopedia', 0.4096570909023285),
 ('site', 0.4041479229927063),
 ('encyclopaedia', 0.391428142786026),
 ('encyclopedia', 0.3877112865447998),
 ('wikipeida', 0.38550564646720886)]
```

(a) "Wikipedia"

```
model_sg.wv.most_similar('obscene')
[('indecent', 0.65315842628479),
 ('dismissive', 0.6378821134567261),
 ('namecalling', 0.6224454045295715),
 ('hurtful', 0.6220864057540894),
 ('inconsiderate', 0.6169877052307129),
 ('abusive', 0.607300877571106),
 ('vulgar', 0.5931051969528198),
 ('trollish', 0.590756893157959),
 ('flippant', 0.5901552438735962),
 ('disparaging', 0.5844916105270386)]
```

(b) "Obscene"

```
model_sg.wv.most_similar('[redacted]')
[('[redacted]', 0.58505779504776),
 ('[redacted]', 0.5793168544769287),
 ('[redacted]', 0.5734737515449524),
 ('[redacted]', 0.5372980237007141),
 ('[redacted]', 0.5339475870132446),
 ('[redacted]', 0.5327820181846619),
 ('[redacted]', 0.5280409455299377),
 ('[redacted]', 0.5256732702255249),
 ('[redacted]', 0.5230817198753357),
 ('[redacted]', 0.5220904350280762)]
```

(c) f-word

5 Model

This section will elaborate on the structure of our classification neural network model. We will also explain model-specific methodologies in this section.

5.1 ReLU and Sigmoid Activation Functions

Activation functions are used to transform the weighted sums of inputs in a fully connected layer to an output value. We will implement two activation functions in the recurrent neural network.

Defined as $ReLU(x) = max(0, x)$, ReLU activation function serves to counter gradient vanishing problems often associated with back-propagation in deep and complex neural network, and provide more efficient computations when training.

Another activation function we employ in the neural network is Sigmoid activation function. Sigmoid function exhibits a "s"-shaped curve and has a range of $[0, 1]$. It is defined as: $S(x) = \frac{1}{1+e^{-x}}$. It is a non-linear, smooth function with steep slope when x is close to zero. Which means it can show clear distinctions of response values even changes in predictors are small. Since we are dealing with binary classifications, it is important for the activation to side with either 0 or 1.

5.2 Binary Cross-entropy

Entropy measures the unpredictability of a state for a variable[18]. It calculates the optimal information needed to identify a state given the current distribution of information. Mathematically, $H(y) = -\sum_i y_i \log y_i$[19]. A cross entropy is then the optimal information needed to identify a state given a false distribution, substituting $\log y_i$ by $\log \hat{y}_i$, distribution

of wrong information. Simplifying for binary cases, we have the binary cross-entropy as $-y_i \log \hat{y}_i + (1-y) \log(1-\hat{y}_i)$. This binary cross entropy loss increases when the predicted probability diverges from actual values.

5.3 Adam Optimizer

We introduce optimizers to accelerate the minimization of loss functions by gradient descent. We already covered SGD, one of the optimizing algorithms in the methodology section. Although SGD is faster in helping the descent of gradient, its accuracy is not as desirable. Here, we introduce Adaptive Moment Estimation (Adam)[20], an optimizer that combines the concepts of Momentum and RMSprop.

Momentum uses an exponentially weighted average of gradients to solve the gradient descent oscillation problem. Sometimes, gradients take too many oscillations of descent to reach the local minimum, because when calculating gradients in SGD, we calculate in mini-batches, which not always leads to local minimum straight away. And increasing the learning rate of the gradient descent may cause the gradient to overshoot in their descent and even diverge. A mathematical formulation of momentum (v_t) can be shown:

$$v_t = \gamma v_{t-1} + \eta g_t$$

$$\theta = \theta_{t-1} - v_t$$

Where η is learning rate, and γ is a constant parameter that controls the weighted average, usually at 0.9, and g_t is the average of the gradient at batch t. By using this exponentially weighted average, we compound the "momentum" toward the local minimum of loss function as we calculate the gradient descent.

RMSprop divides the learning rate by an exponentially decaying average of squared gradi-

ents[21], which drastically reduces learning rate as gradients approaches the minimum.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{E[g^2]_{t-1} + \epsilon}} g_{t-1}$$

Adam computes an adaptive learning rate for gradient descent. It not only calculates exponentially decaying average of squared gradients like RMSprop, it also calculates exponentially decaying average of gradients like Momentum.

$$m_t = \gamma_1 m_{t-1} + (1-\gamma_1) g_t$$

$$v_t = \gamma_2 v_{t-1} + (1-\gamma_2) g_t^2$$

$$\theta_t = \theta_{t-1} - \eta \frac{m_{t-1}}{\sqrt{v_{t-1} + \epsilon}}$$

m_t and v_t are first and second moment of gradient respectively, which are essentially mean and variance of the gradient. Adam out performs regular SGD and RMSprop in terms of training cost.

5.4 Baseline Network Layers

Proceeding to construct a baseline recurrent neural network, we use Keras framework which uses Tensorflow backend with Graphics Processing Unit(GPU) support. The sheer amount of computations involved in training a neural network often throttles the CPU, that is the reason we use a Nvidia GTX 1060 GPU with 6 gigabytes of memory. It is tested that the GPU speed up our computation by more than three times, comparing to a typical four code eight thread CPU. On the other hand, Keras is a high level open-source neural network framework written in Python, which can utilize many machine learning backends, like Tensorflow. It

is user-friendly and modular without losing the functionality of a typical machine learning framework. We use its Sequential Model to build our recurrent neural network.

```
mode = Sequential()
```

The first layer of our model is the input/embedding layer. We will not do the embedding entirely again, but load the embedding matrices that was built using Skip-gram model. Since we have our 20,000-word vocabulary, padded our sentences to 100 words and each word has an embedding vector of length 300, this input/embedding layer has the size of $[20000 \times 300]$ for each input of size 100. One of the default hyperparameters in this layer is whether the embedding matrix is trainable. We default it to trainable so that the embedding vectors serve as initialization. Since we did not include all the available vocabularies in the embedding matrix with the minimum word count option in Skip-gram model, slight improvements can be made to the embedding vectors by this "trainable" option.

```
model.add(Embedding(20000,300,weights=[embedding_matrix],input_length=100))
```

The second layer of our model is the bidirectional LSTM layer. We set LSTM layer to have 100 cells. Thus the Bidirectional LSTM layer has 200 nodes in total. We also set it to have a 10% dropout rate and a 10% recurrent dropout rate. Referring to Figure 4, a regular dropout in this layer means that dropout is applied to the input gates, whereas a recurrent dropout means dropout is applied to hidden states across the recurrent units in the layer[22].

```
model.add(
  Bidirectional(LSTM(100,dropout=0.1,recurrent_dropout=0.1)))
```

We connect some fully connected layers with a dropout layer in between. The first fully-connected layer uses ReLU activation function, and the second one with Sigmoid. We use

Sigmoid in the second fully-connected layer because we are producing an output six binary classifications and we want results to be probabilities between 0 and 1.

```
model.add(Dense(50, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(6, activation='sigmoid'))
```

Finally, we compile the model using binary cross-entropy loss, Adam optimizer and evaluation metric of Area Under Curve (AUC). AUC is the area under the Receiver Operating Characteristics (ROC) curve, drawn by plotting true positive rate against false positive rate. AUC is essentially the probability that the classifier/model will rank a randomly drawn positive case higher than a randomly drawn negative case, all assuming "positive" ranks higher than "negative."[23]

```
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=[auc])
```

We fit the model with a mini-batch size of 256 and over ten epochs. We also take 10% of our training data out as a validation set. This validation set is drawn before model training. The reason behind this is that we want to control the splitting with a set random seed so that our result is somewhat reproducible. We also introduce early stopping on validation loss, so that training is stopped after the epoch that it detects the model is overfitting the training set.

```
X_t, X_v, Y_t, Y_v = train_test_split(X,Y,train_size=0.9,random_state=123)
early = EarlyStopping(monitor="val_loss", mode="min")
model.fit(X_t,Y_t,batch_size=256,epochs=10,
    validation_data=(X_v, Y_v),callbacks=[early])
```

6 Analysis of Results

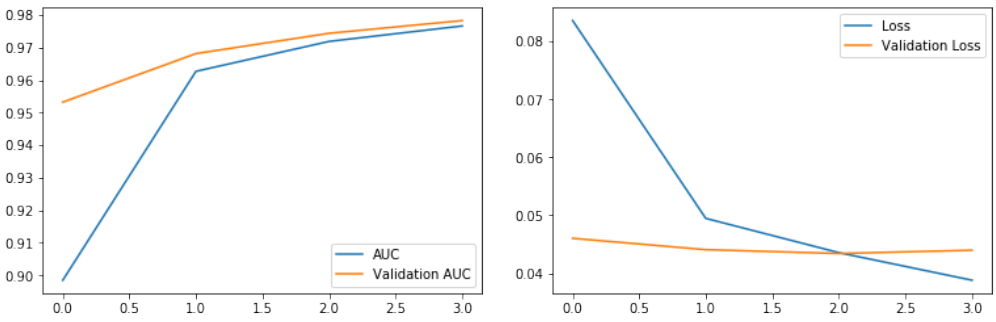
This section will illustrate the training cost and performance of previously constructed baseline model with Word2Vec embedding, bidirectional LSTM, fully connected layers with dropout, using binary cross entropy loss and Adam optimizer.

The approach of our analysis will be focusing on substituting components in our baseline model with proven alternatives instead of tuning. We will perform comparison studies that demonstrate models with Gated Recurrent Units, pre-trained word embedding vectors, penalizing loss, and undersampling.

6.1 Baseline Model

Baseline model took 706 seconds to complete training on 143613 samples and validating on 15958 samples. As shown in Figure 11 training stopped at epoch 4. We see that training loss and validation loss is extremely close at epoch 3. Training AUC at epoch 4 is 0.9782 and validation AUC is 0.9797.

Figure 11: AUC and Loss Over Epochs - Baseline Model



(a) "AUC" (b) "Loss"

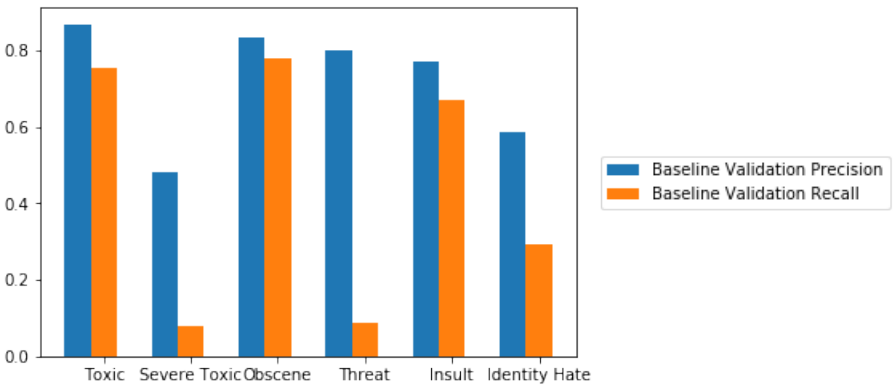
It is important to note that validation AUC is higher than training AUC. Explanation of this phenomena is that we use dropout vigorously in our training model to avoid overfitting,

and training and validation samples do not go through the same pipeline. When training data goes through the model and dropout layers, information is lost, resulting in harder training but less overfitting. However, when validation data goes through evaluation using the model, dropout is not applied to them, and validation samples retain all its information. This difference occurs similar to the model being underfitting. But in reality, it does not necessarily mean the model is underfitting.

However, using AUC as a single model performance metric does not give us a full picture, especially considering this is an imbalanced dataset. Here we introduce precision and recall scores to explore the model performance further.

Precision, in the Mathematical formulation, is defined as $\frac{TP}{TP+FP}$. TP stands for true positive rate, and FP is the false positive rate. In simple terms, it calculates the percentage of correct classification out of all the positive classifications. On the other hands, recall is defined as $\frac{TP}{TP+FN}$, where FN stands for the false negative rate. Also called "Sensitivity", recall shows the classifier's ability to classify the results as positive when the subjects are genuinely positive.

Figure 12: Baseline Precision and Recall



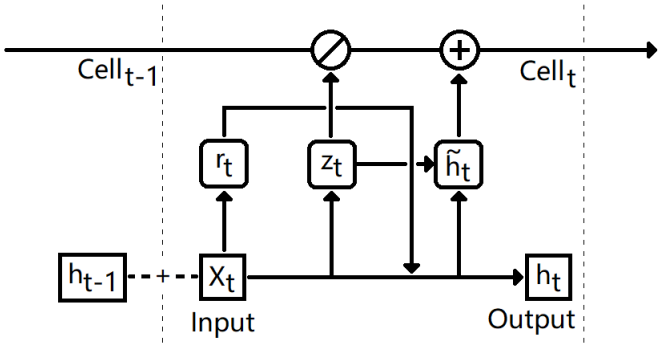
An indication for a descent classifier is that it shows not only high values in both precision and recall but also a balance between them. From Figure 12, we see that for relatively minor

classes like "Severe Toxic", "Threat", and "Identity Hate", the baseline model produces decent precision but low recall. These scores mean that while the model is somewhat accurate when it detects these minority classes, it still misses these comments quite often. It is also surprising to see that the baseline model performs worse at detecting "Severe Toxic" comments comparing to "Threat", since "Severe Toxic" comments have more occurrences in our training samples.

6.2 Comparison Study: Gated Recurrent Units

One of the alternatives to LSTM is Gated Recurrent Units (GRU). It has fewer parameters, as shown in Figure 13, which combines the input gate and forget gate as update gate, connects the hidden state with a reset gate and has no output gate. It has lower training cost and is shown to perform better on small datasets[24].

Figure 13: Detailed Structure of a GRU Cell



A general formulation of a GRU cell can be shown as:

$$z_t = \sigma_g(W_z x_t + U_z h_{t-1} + b_z)$$

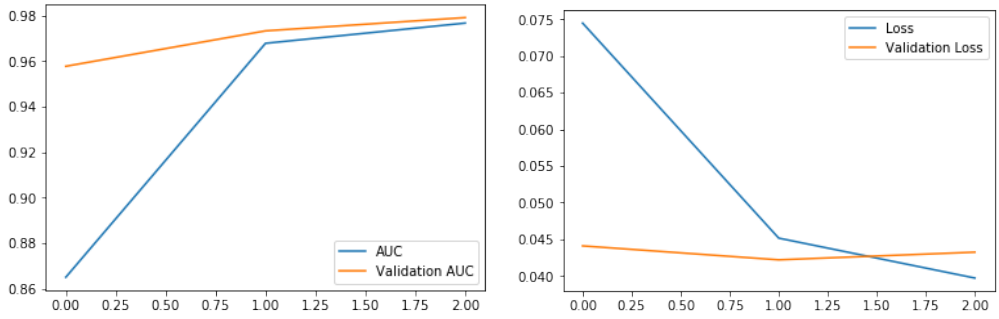
$$r_t = \sigma_g(W_r x_t + U_r h_{t-1} + b_r)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tanh(W_h x_t + U_h(r_t * h_{t-1}) + b_h)$$

Where z_t is the update gate, r_t is the reset gate, and h_t is the output vector. Note that from Figure 13, $\hat{h}_t$ is equivalent to $\tanh(W_h x_t + U_h(r_t * h_{t-1}) + b_h)$ in the above equation.

GRU model took 427 seconds to complete training on 143613 samples and validating on 15958 samples. As shown in Figure 14, training stopped at epoch 3. Training AUC at epoch 3 is 0.9761 and validation AUC is 0.9782. GRU model shows lower training cost, but it is faster to overfit.

Figure 14: AUC and Loss Over Epochs - GRU



(a) "AUC" (b) "Loss"

Shown in Figure 19 (at the end of Section 6), we see that the GRU model shows slightly lower precision and higher recall compared to the baseline model. Note that GRU model completely misclassifies comments that should be labeled "Threat". It performs somewhat worse overall and fails at detecting extremely minor classes.

6.3 Comparison Study: Pre-trained Embedding Vectors From Large Corpora

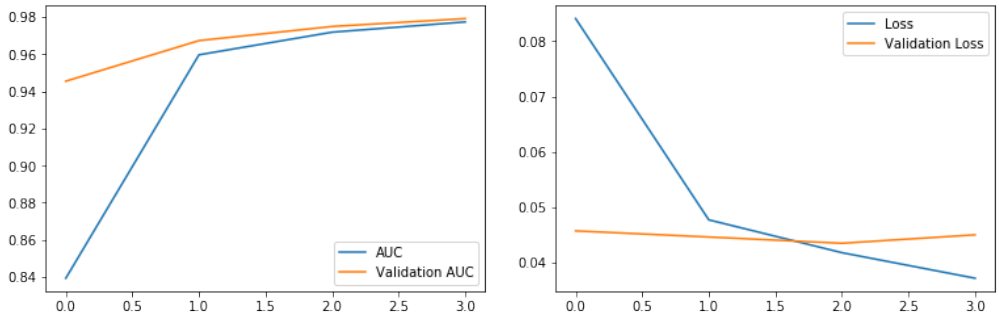
Word vectorization models like word2vec we implemented usually requires a large corpus to compute suitable word vector representations, which is evident in our observations on the

similarity output of the Skip-gram model on our comment texts. Thus, it is often suggested that we use pre-trained embedding vectors from large corpora to perform analysis on samples with a small corpus[25]. In this section, we introduce two pre-trained embedding vectors and use them as embedding initialization for our model.

First, we use embedding vectors trained by Google using 3 Million word vocabulary from Google Negative News.[26] These embedding vectors are prepared using the same word2vec method we employed in previous sections.

Same baseline model but with pre-trained "Google News Negative 3M" took 842 seconds to complete training on 143613 samples and validating on 15958 samples. As shown in Figure 15 training stopped at epoch 4. Training AUC at epoch 3 is 0.9719 and validation AUC is 0.9750. The model using pre-trained word2vec embedding shows worse overall performance and higher training cost comparing to baseline model.

Figure 15: AUC and Loss Over Epochs - Pre-trained Google News Negative 3M



(a) "AUC" (b) "Loss"

We also implement embedding vectors pre-trained using scrapped twitter data. These embedding vectors are trained using a different embedding method called Global Vector (GloVe) for word representation developed by Stanford[27]. It is an unsupervised learning model that obtains the embedding vectors by aggregating global word-word co-occurrence statistics in

a corpus. Its formulation is defined by:

$$w_i^T w_j + b_i + b_j = \log X_{ij}$$

$$Loss = \sum_{i=1}^{V} \sum_{j=1}^{V} f(X_{ij})(w_i^T w_j + b_i + b_j - \log X_{ij})^2$$

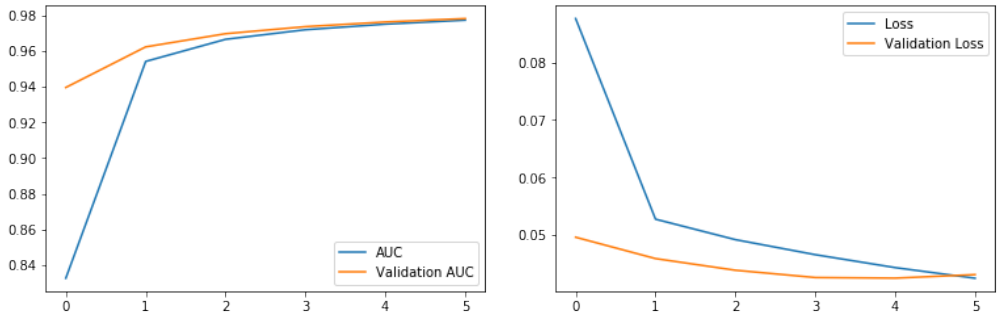
$$f(X_{ij}) = (\frac{X_{ij}}{x_{max}})^{\alpha} \text{ if } X_{ij} < XMAX \text{ or } 1, \text{otherwise.}$$

Where w_i and w_j are chosen word and context word, b_i and b_j are biases for the chosen and context word. X_{ij} is the occurrence matrix of word i in context word j. $f(X_{ij})$ denotes the weight function that reduces the weight on most common words[28].

This GloVe embedding matrix is trained using a 2 billion word twitter corpus, which shares more words with our vocabulary than the word2vec embedding vectors trained on Google News Dataset.

Same baseline model but with pre-trained "Twitter 2B" took 808 seconds to complete training on 143613 samples and validating on 15958 samples. As shown in Figure 16 training stopped at epoch 6. Training AUC at epoch 6 is 0.9771 and validation AUC is 0.9780.

Figure 16: AUC and Loss Over Epochs - Pre-trained Twitter 2B



(a) "AUC" (b) "Loss"

We see that model with GloVe embedding vectors takes more epochs to overfit, shows less validation loss at later epochs, and has relatively higher AUC score. However, it also in-

dicates that our baseline model with our own embedding performs as well as pre-trained embedding. The explanation behind this is that pre-trained embedding on larger corpus does not necessarily have the domain-specific vocabulary of our dataset, resulting in a lack-luster performance.

Regarding precision and recall, as shown in Figure 19, Google News word2vec and Twitter GloVe model both have slightly lower precision and higher recall comparing to our baseline model. Overall, they perform slightly worse than the baseline model. Moreover, we notice that they also completely missed "Threat" class from the validation set. Comparing Google News word2vec and Twitter GloVe models, GloVe has higher training cost and lower validation AUC but has higher precision across all labels.

6.4 Comparison Study: Penalizing Loss

One of the ways to combat class imbalance issue is to add weights on minority class prediction losses, which penalize against misclassification at the same time. These weights tell the model to pay more attention to the minority classes in the training samples.

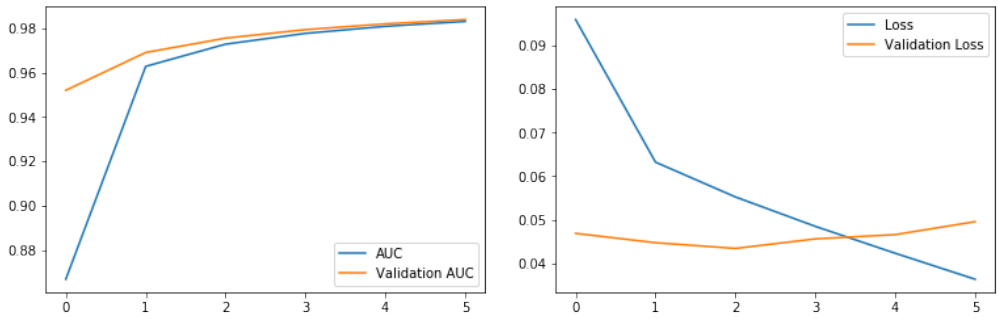
The Mathematical formulation of our weight, Inverse Frequency[29], is defined as

$$W_c = \log(\frac{N}{n_c}) \text{ if } W_c > 1, \text{ else } 1$$

Where N stands for total observations, n_c stands for the number of observations for the specific class, and they are logarithmically scaled. We also keep the weight of majority class samples (clean comments) at 1.0.

Our model with penalizing class weights took 1246 seconds to complete training on 143613 samples and validating on 15958 samples. It is less efficient in training than models we previously employed because penalizing weights on the loss function causes the model parameter

Figure 17: AUC and Loss Over Epochs - Penalizing Loss



(a) "AUC" (b) "Loss"

to fluctuate and converge very slowly. Training stopped at epoch 6 with training set AUC at 0.9830 and validation set AUC at 0.9838. According to Figure 19, penalizing class weights model shows slightly lower precision but higher recall overall.

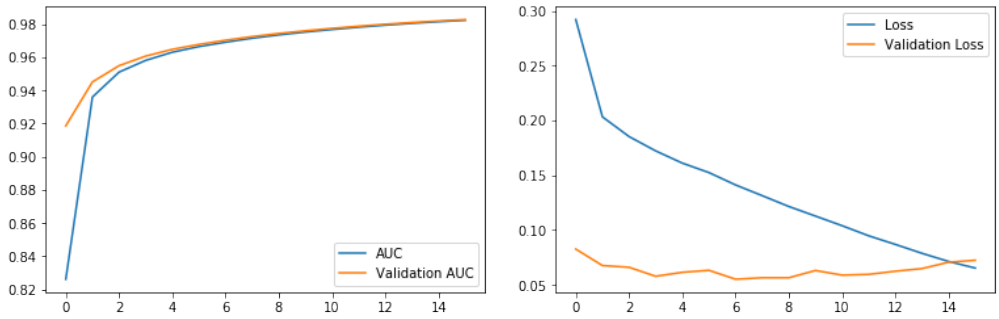
6.5 Comparison Study: Undersampling

We explore simple sampling techniques on our training step to see if they improve model performance. As an example, we apply undersampling to majority class training samples (clean comments) while retaining minority classes. This method essentially makes the model pay more attention to minority class samples.

We sample 10% the majority class training samples with replacement and combine them with all the minority class samples into a new training set. We do this on every new epoch and training our baseline model using the new training sets. Note that total amount of samples from majority class and minority class are about equal with this way of undersampling.

Model with undersampling majority class samples took 622 seconds to complete training. It stopped at epoch 16 with training AUC of 0.9824 and validation AUC 0.9828. It shows higher overall accuracy and relatively low cost. The model exhibits higher recall rate across the board with different classes, which means it is significantly better at picking out toxic

Figure 18: AUC and Loss Over Epochs - Undersampling



(a) "AUC" (b) "Loss"

comments. However, it has lower precision in general, meaning it is underfitting for classifying clean comments. The model still under-performs on minority class samples like "Severe Toxic" and "Threat".

Figure 19: Model Comparisons

		Baseline	GRU	News w2v	Twitter GloVe	Class Weight	Undersampling
Training Cost (seconds)		706	427	851	1564	1246	622
Validation AUC		**0.9797**	**0.9782**	**0.9778**	**0.9775**	**0.9838**	**0.9828**
Toxic	Precision	0.8672	0.8277	0.8052	0.8417	0.7862	0.6241
	Recall	0.7514	0.7707	0.7858	0.7762	0.7775	0.8485
Severe Toxic	Precision	0.48	0.4348	0.44	0.5231	0.4103	0.3714
	Recall	0.08	0.2667	0.22	0.2267	0.32	0.26
Obscene	Precision	0.8316	0.8252	0.827	0.8406	0.792	0.7327
	Recall	0.7783	0.8079	0.771	0.7734	0.8067	0.8067
Threat	Precision	0.8	0	0	0	0.5714	0.3913
	Recall	0.08889	0	0	0	0.1778	0.2
Insult	Precision	0.7704	0.742	0.7194	0.7434	0.7074	0.644
	Recall	0.6711	0.7141	0.7262	0.6805	0.7074	0.699
Identity Hate	Precision	0.5862	0.5556	0.6818	0.6667	0.4886	0.4615
	Recall	0.2906	0.4274	0.2564	0.2222	0.3675	0.5128

7 Conclusions

We successfully employed word2vec embedding and recurrent neural network in building a toxic comment classification model and achieved high accuracy with relatively low cost. Gated recurrent units layer proves to be more efficient at training but performs slightly worse than the baseline model with LSTM layer. Comparison studies show that pre-trained embedding vectors obtained from larger corpora do not necessarily improve the performance of our model because they require domain-specific vocabulary to perform accurately enough. Penalizing loss functions and using sampling techniques only marginally enhanced our model in detecting underrepresented classes. Like mentioned by [10], one of the significant challenges researchers in machine learning face is the limitation of "high quality" data. If we can have a significantly larger training sample, with more labeled texts and more balanced classes, we will likely achieve sound improvements to our model without relying heavily on the assistance of class imbalance solutions, since they require significant amount of tuning.

These observations also mean that further improvements can be made to improve our current model, which does not limit to a different overall network structure. We can also perform additional hyperparameter tuning on our model, which will most definitely prove beneficial. Nevertheless, the model's ability to process context of words proves efficient in identifying toxic texts from a large sample.

On a closing note, Conversation AI team's intention and effort in building an open source tool to monitor and control online toxicity is commendable. Researchers and discussion platform moderators have already found numerous ways to apply this tool in very creative manners[30]. We hope that with the collaborative help from the machine learning community, the team can continuously improve the performance of Perspective API and help maintain a toxic-free environment for our online discussions.

References

[1] CJ Adams and Lucas Dixon. *Better discussions with imperfect models*. URL: https://medium.com/the-false-positive/better-discussions-with-imperfect-models-91558235d442.

[2] Tetsuya Nasukawa and Jeonghee Yi. "Sentiment analysis: Capturing favorability using natural language processing". In: *Proceedings of the 2nd international conference on Knowledge capture*. ACM. 2003, pp. 70–77.

[3] Hassan Saif, Yulan He, and Harith Alani. "Semantic sentiment analysis of twitter". In: *International semantic web conference*. Springer. 2012, pp. 508–524.

[4] Heng Wang, Zengchang Qin, and Tao Wan. "Text Generation Based on Generative Adversarial Nets with Latent Variable". In: *arXiv preprint arXiv:1712.00170* (2017).

[5] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

[6] Geoffrey Hinton et al. "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 82–97.

[7] Eric Schmidt. *Google Ideas Becomes Jigsaw*. URL: https://medium.com/jigsaw/google-ideas-becomes-jigsaw-bcb5bd08c423.

[8] Daisuke Wakabayashi. *Google Cousin Develops Technology to Flag Toxic Online Comments*. URL: https://www.nytimes.com/2017/02/23/technology/google-jigsaw-monitor-toxic-online-comments.html.

[9] Conversation AI. *Toxic Comment Classification Challenge*. URL: https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge.

[10] Conversation AI. *Conversation AI by conversationai*. URL: https://conversationai.github.io/.

[11] Tomas Mikolov et al. "Efficient estimation of word representations in vector space". In: *ICLR Workshop* (2013). URL: https://arxiv.org/abs/1301.3781.

[12] Richard Socher. *CS224d Deep Learning for Natural Language Processing Lecture 2: Word Vectors*. URL: https://cs224d.stanford.edu/lectures/CS224d-Lecture2.pdf.

[13] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[14] Christopher Olah. *Understanding LSTM Networks*. 2015. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[15] Mike Schuster and Kuldip K. Paliwal. "Bidirectional recurrent neural networks." In: *Signal Processing, IEEE Transactions on 45.11* (1997).

[16] Nitesh V. Chawla, Nathalie Japkowicz, and Aleksander Koltcz. "Proceedings of the ICML'2003 workshop on learning from imbalanced data sets". In: (2003).

[17] Ying Liu, Han Tong Loh, and Aixin Sun. "Imbalanced text classification: A term weighting approach". In: *Expert Systems with Applications* (2009). URL: http://scholarbank.nus.edu.sg/handle/10635/60483.

[18] Claude Shannon and Warren Weaver. "The Mathematical Theory of Communication". In: *Univ of Illinois Press* (1949).

[19] Rob DiPietro. *A Friendly Introduction to Cross-Entropy Loss*. 2016. URL: https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/.

[20] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *Learning (cs.LG) arXiv:1412.6980* (2017).

[21] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016. URL: http://ruder.io/optimizing-gradient-descent/index.html#rmsprop.

[22] Yarin Gal and Zoubin Ghahramani. "A Theoretically Grounded Application of Dropout in Recurrent Neural Networks". In: *NIPS, arXiv:1512.05287* (2016).

[23] Tom Fawcett. "An introduction to ROC analysis". In: *Pattern Recognition Letters, 27, 861–874.* (2006).

[24] Junyoung Chung et al. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: *NIPS 2014 Deep Learning and Representation Learning Workshop* (2014).

[25] Seyed Mahdi Rezaeinia, Ali Ghodsi, and Rouhollah Rahmani. "Improving the Accuracy of Pre-trained Word Embeddings for Sentiment Analysis". In: *arXiv:1711.08609* (2017).

[26] Google. *Google Code Achieve, w2v.* URL: `https://code.google.com/archive/p/word2vec/`.

[27] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. *GloVe: Global Vectors for Word Representation.* URL: `https://nlp.stanford.edu/projects/glove/`.

[28] Dmitriy Selivanov. *GloVe Word Embeddings.* URL: `http://text2vec.org/glove.html`.

[29] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval.* first. Cambridge University Press, 2008.

[30] cjadams. *Perspective Hacks.* 2017. URL: `https://github.com/conversationai/perspectiveapi/wiki/perspective-hacks`.