COMP6208: Group Coursework - Toxic Comment Classification Challenge from Kaggle

Anton Okhotnikov
University Of Southampton
ao2u17@soton.ac.uk

Lakshmi Mukkawar
University Of Southampton
lpm1n17@soton.ac.uk

Nunzio Gatti
University Of Southampton
ng1e17@soton.ac.uk

Richa Ranjan
University Of Southampton
rr2n17@soton.ac.uk

Yeldos Balgabekov
University Of Southampton
yb2n17@soton.ac.uk

ABSTRACT

Supervised learning and Text classification are two of the most sought-after areas programmers work for. Natural Language Processing has seen some major breakthroughs in the last two decades. With the advent of digitization, an exponential increase in online textual information has been observed. Supervised classification comes into picture when these text data need to be grouped based on some pre-defined categories. Labeled data is fed to the Machine Learning algorithm, which further generates models to perform classification on the unseen data. This paper summarizes various experiments and observations that involved Multi-label classification of comments. In this project, we have tried to address a Kaggle competition based on comment classification.

1 INTRODUCTION

Kaggle, the well-known platform for predictive modeling and analytics defines the **Toxic comment classification** challenge as: building a model that classifies toxic comments into six different categories such as **threat**, **obscene**, **insult**, **toxic**, **identity hate** and **severe toxic**. Google and Jigsaw have worked together on improving online conversations and one of their focus is to identify negative comments, rude behavior or discourteous comments.

The original tagging was done via *crowdsourcing* which means that the dataset was rated by different people and the tagging might not be 100% accurate. Kaggle evaluates the submissions for this challenge by calculating the mean column-wise Area Under the Curve(AUC) for Receiver Operating Characteristics(ROC). So, the score for individual predicted column is recorded and then average of all six predicted columns is taken as the final evaluation.

2 EXPLORING AND PRE-PROCESSING THE DATA

The dataset provided by Kaggle is a collection of comments from the talk page edits on *Wikipedia*. Training dataset consists of around 159500 rows with columns as Id, Comment, and labels for each comment. There are some comments which have been classified into multiple toxicity types (e.g. a comment can be classified as both toxic and obscene). A closer look into the data showed that the data set is not clean. Comments contain special characters, punctuations, spelling mistakes, and some of the comments also contain IP addresses, numbers. Evidently, there are some words which do not come under English dictionary, as is the usual case with such comments and other online conversations. So, first step required was to clean the data.

Incorrect spelling seemed to be a common concern in the comments. To deal with this, we used the `pyenchant` library provided by python. The whole sentence was fed into `pyenchant`, it checked for correct words and suggested back the incorrect words. This gave us a list of words which have similar meaning to the incorrect word and we chose first word in that list as it is the most similar word.

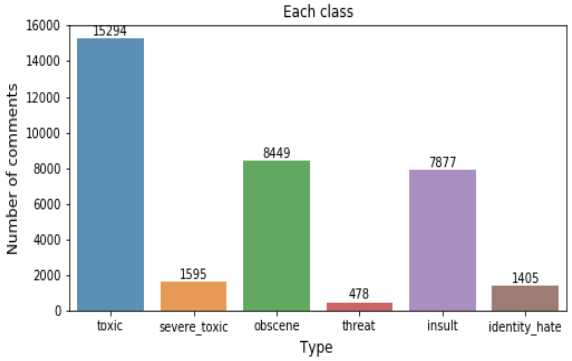

Figure 1: Distribution of comment labels across training dataset

In the training dataset, comments are not evenly distributed for each label. For example, There are 15294 comments which are Toxic whereas only 478 comments are of type Threat. Figure 1 shows the distribution of all comments type from

training data. Furthermore, There are 143346 comments which are neutral. Neutral comments are the ones which do not come under any of the six categories mentioned above. Also, there are some empty comments. This tells us that our training set is not balanced. As a sanity check, we applied a couple of classifiers (SVM and Random Forest) on the original data, and obtained an AUC of around 0.99. So it was evident that when applying any classifier on an unbalanced dataset, unrealistic results are obtained. Due to this reason, balancing the data is important.

2.1 Finding Correlations

After finding out the distribution of various comment classes, it was imperative to understand the relations between all the six labels used. The variables involved in this dataset are categorical values. The usual approach of finding Pearson correlation using the `corr` function is not applicable here, as we are dealing with categorical variables and Pearson correlation tends to normalize the data which is unfit for categorical data. As an alternative, a *Confusion Matrix* or *Crosstab* can be used to find the patterns between these variables. An example of a Confusion Matrix is shown in the figure 2. The table shown below is a confusion matrix of

	severe_toxic		obscene		threat		insult	
severe_toxic	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**
toxic								
0	144277	0	143754	523	144248	29	143744	533
1	13699	1595	7368	7926	14845	449	7950	7344

Figure 2: Confusion Matrix of Toxic label with other labels

Toxic class with other classes. In terms of observations, the following can be said:

- A *Severe Toxic* comment is always Toxic.
- Apart from a few exceptions, most of the other classes seem to be a subset of the Toxic set.

Understanding these patterns and relations between labels were helpful in further classification steps.

2.2 Balancing the Data

Unbalanced data is a major concern in real-world classification problems. If the metrics are not adjusted, we might end up optimizing for a meaningless metric and over-fitting. There are several options for handling unbalanced data. *Down-sampling* and *Up-sampling* are the commonly used techniques to make the dataset even. Observing accuracies of the classifiers on both these datasets would be an interesting insight. So, in this project we have applied both the techniques to balance our data.

In down-sampling method, we created a balanced set by matching number of samples in the minority class. We used different techniques to down-sample data:

- **Random Under-sampling** : We randomly ignored samples from the majority class. The main disadvantage of using this technique was the loss of some information from the corpus of sentences. To deal with this, we used the next technique, called the "near miss".
- **Near Miss** : We retained points from the majority class whose mean distance to the k-nearest points in the minority class was the lowest.

We observed a considerable amount of data loss on performing down-sampling. Because of less data, any training model could not learn all the features properly and hence we observed less AUC with this method. At this stage, up-sampling seemed to be a better approach.

Up-sampling method works by matching the number of samples with majority of class with the samples from minority classes. We performed this by transforming comments from different languages to English language. Mainly, we used three methods to up-sample our dataset :

- **Translation** : We applied a triple translation to our original comments(i.e. English -> Spanish -> German-> Italian-> English). In this way, we obtained comments with slightly different words keeping the same meaning.
- **Synonyms** : Conceptually similar to translation, but in this case we simply replaced words with their respective synonyms.
- **SMOTE** (Synthetic Minority Over-Sampling): Rather than just repeating the minority observations, it creates synthetic observations based on the existing minority observations. This algorithm takes into account two close instances, creates one synthetic example along the line, between the two instances considered, as shown in Figure 3.[1]

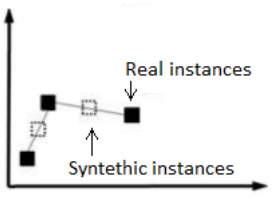

Figure 3: SMOTE analysis

3 FEATURE ENGINEERING

The first step in the Feature Engineering process was to convert text to numerical features, as text data cannot be directly classified. Thus, it was necessary to get a sense of *Direct Features* like *Word frequency* and *Vector distance mapping of words* .

3.1 Converting text to numerical features

The Count based direct features i.e. based on the frequency distribution of words were created from the text data. These methods described here first create a *vocabulary* of words, and then a *sparse matrix* of word counts. Vectorization techniques such as TF-IDF, Bag of words, word2Vec and Fast-text used in this project, have been described below:

3.1.1 TF-IDF. The *Term Frequency-Inverse Document Frequency* (**TF-IDF**) is the most famous term-weighting method. The `TFIDFVectorizer` library by `sklearn` does the feature extraction for us. The result is a matrix of TF-IDF features. This was applied on word and character levels in our dataset. Pre-processing of data is included in TF-IDF where we include stop words removal, changing words to lower case, *tokenization* and using regular expressions for deleting special characters.

3.1.2 Word2Vec, FastText and Google Word2Vec. A shallow word-embedding model, called **Word2Vec** provides embeddings for words from pre-trained two-layer Neural Network that keeps linguistic contexts of words (similar words are placed near each other in a multi-dimensional space)[4]. We used the `gensim`[1] package for Word2Vec method first, and created words representations in 100 and 300-dimensions spaces. One limitation of using Word2Vec was, if the word was not in the Word2Vec vocabulary, then there is no vector representation for that word (e.g. we couldn't find the vector representation for word "amortized").

For handling this problem, we moved to another approach similar to the Word2Vec model, which is called **FastText**[2] (an extension of Word2Vec). This model is able to reconstruct the word from the characters *n-grams* of different words that the model was trained on. So, even if the word is not in the model vocabulary, we still can have a vector representation for that word. For example, the word *sunlight* was reconstructed from parts of the words *sun-flower* and *moon-light*.

Another reasonable approach to FastText that we used was **Google Word2Vec**. This is a pre-trained model that has a vocabulary of 3 million words and their vector representations in 300-dimensional space. Such amount of words in a vocabulary reduces the probability that we won't find vector representation for a particular word. Originally this model was trained on roughly 100 billion words from a Google News dataset [3].

3.1.3 Bag Of Words. **Bag Of Words**(BOW), being one of the most commonly used feature engineering techniques, was applied on the comments for vectorizing. The approach is used in classification problems where the frequency of occurrence of each word is used as a feature for training a classifier. Each unique word will therefore correspond to a numerical feature. The BOW approach was applied on the up-sampled data and as a result, we had six different vectorizers, one corresponding to each label.

3.2 Fisher vector and GMM

Up to this stage, we had a vector representation for each word. But our comments are the sequence of words like $S =< w1, w2, w3...w_n > W = (w_i \epsilon D | i = 1....N)$. As these comments are different in length, the number of words in each comment varies. Therefore, a vector representation for the whole comment was required. To do this, we take the comments and vector representations of each word to find the *Fisher Vector* for each comment. We have applied a *Gaussian Mixture Model* (GMM) with the number of components as 32 and 64 on the sequence of vectors to obtain one *Fisher Vector* for a comment that can be used for a classification. Since we had 100 and 300-dimensional FastText word representation, the resulting Fisher Vector had about 6,000 to 60,000 features depending on the number of dimensions in the FastText model and number of components in GMM. In order to reduce the number of dimensions and remove irrelevant features we applied **PCA** (Principal Component Analysis) and saved only 1,000 main components. Before applying PCA, the data was normalized using *Min-Max* as well as *Standard Scaler* techniques. On experimentation, it was found that the Min-Max technique produced less outliers compared to Standard Scaler, and therefore, better AUCs. So we used Min-Max normalization going ahead.

4 MODELS

Once we have the vectorized representation of comments, there are a range of learning models we could use, and in order to check which provides the maximum AUC, we applied various models to our data. A general flow of the models/techniques used, is described in the figure 4.

4.1 Naive Bayes

Following the Bag-Of-Words approach for vectorization, the *Gaussian Naive Bayes* classifier was applied on it. With a train-test split of 80-20 % respectively, Naive Bayes was applied on all six comment labels. The AUCs for each of the

[1] https://radimrehurek.com/gensim/models/word2vec.html
[2] https://radimrehurek.com/gensim/models/fasttext.html
[3] http://mccormickml.com/2016/04/12/googles-pretrained-word2vec-model-in-python/

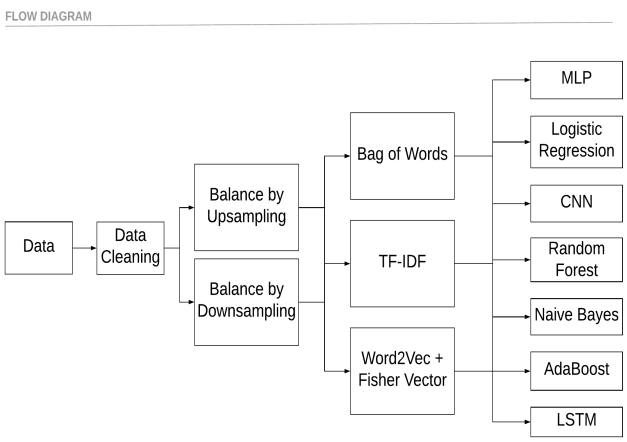

Figure 4: Flow Diagram

classifiers were calculated separately corresponding to each labels.

Observation: On an average, the AUC was about 0.50 for all classifiers. This being an undesirable figure, encouraged us to use the Word2Vec and Fisher Vector results for further classifiers.

4.2 Logistic Regression

Logistic Regression is applied on the down-sampled dataset using the TF-IDF method. This task includes classification in multiple classes such as toxic, severe-toxic, etc. We divided data into training set(80%) and validation set(20%). Our approach involved building six simple binary classifiers for each comment type. So, we have six trained models for each type of comment. For example, first model will classify if the given comment is toxic or not (0 or 1). This is how other classifiers will classify if the comment is severe-toxic, threat, insult, identity-hate, obscene or not. For checking the performance of these models, we used validation set and stored the predictions of each classifier. Then we calculated the AUC for individual classifier and took the average of all these 6 classifiers AUC.

Observation: The AUC for each comment type is given in the table 1. Overall AUC for this method is only 0.63. Logistic Regression on the down-sampled data provided poor performance. So we used 100 dimensions FastText embedding and the Fisher Vector (32 GMM components) data, the results obtained are summarized in table 2:

4.3 SVM

Support Vector Machines (SVM) can be used for classification and regression analysis. We used an SVM model as a starting point because we wanted to check classification performance with different models. SVM finds out the maximum margin

Comment type	AUC
Toxic	0.488
severe-toxic	0.555
Threat	0.565
Insult	0.504
Identity hate	0.5725
Obscene	0.489
Total	0.63

Table 1: AUC for each comment type by Logistic Regression on down-sampled data

Comment type	AUC	Time Taken
Toxic	0.986	39.89 s
Severe Toxic	0.973	3.02 s
Threat	0.997	0.805 s
Insult	0.985	16.95 s
Identity hate	0.969	3.42 s
Obscene	0.983	17.76 s

Table 2: Logistic Regression AUC for each comment type on GMM data

hyperplane where the distance from it to the nearest data point on each side is maximized. We used the SVM package from `sklearn` to learn data with validation set (80% training and 20% test data). By using up-sampled data with FastText embeddings and 32GMM Fisher vector representations of the input comments we trained our model.

Observation: We obtained an AUC of 0.959. Although this classifier gave a good accuracy, it was found to be slow to train, owing to the time complexity of SVM which is n^3.

4.4 Random Forest

Random forest is an ensemble learning method for classification, regression, etc.. This classifier works by constructing multiple decision trees and merging them to get good AUC, while training the data. We applied Random Forest model with TF-IDF, BOW and Fisher Vector representations.

First, with Bag of Words representation on the up-sampled data, we experimented with different values of maximum-depth (maximum depth of the tree), n-estimators (number of decision trees) and maximum-features (number of features to consider when looking into best split) attributes of the random forest classifier by `sklearn`. For cross-validation (80% for training and 20% for testing data) on training data, best AUC for different parameters was found.

We changed the n-estimators parameter from 10 till 100 with incrementing value by 10 each time. No drastic change in the AUC was observed. Around n-estimators = 50, we got an AUC of 0.5. For other values of n-estimators (below 50

and above 50) the AUC was around 0.49. Then, we checked AUC with 64, 100, 500 number for maximum-depth and for maximum-feature we set value as 100, 500 and 1000. With this approach we again observed an AUC of 0.5.

For the second attempt, we used TF-IDF method with both up-sampled and down-sampled data.

Observation: On down-sampled dataset, we got AUC of 0.77 whereas with the up-sampled dataset our AUC changed drastically to 0.9615 by considering number of estimators between 800 and 900, and features values between 35 and 40. We also trained Random Forest using the Fisher Vector with different representations of words(Google Word2Vec and Fast Text). The observations recorded are shown in table 3.

Type of data	Embedding used	AUC
Down-sampled	TF-IDF	0.77
Up-sampled	Bag of Words	0.50
Up-sampled	TF-IDF	0.93
Up-sampled	FisherVector & FastText	0.94
Up-sampled	FisherVector & Google Word2vec	0.96

Table 3: Random Forest AUC on different scenarios

4.5 AdaBoost

The *AdaBoost* algorithm is a classic approach to binary classification problems. In this case, learning is boosted by combining T weak learners together: $F_T(x) = \sum_{t=1}^{T} f_t(x)$ minimizing the error $E_t = \sum_i E[F_{t-1}(x_i) + \alpha_t h(x_i)]$

Observation: On testing this method on a small portion of our data set, we achieved AUC: 0.9702. However, the use of decision trees resulted in a slow performance and thus, we did not go ahead with this method for training the entire dataset, as this would result in a significantly slow training.

4.6 MLP

We tried different structures of the *Multi-Layer Perceptron* by changing number of hidden layers in the network and different activation functions, applying **dropout regularization** [5]. After a couple of experiments it was decided to use 4-layers network (1 input, 2 hidden and 1 output layers) with 1,000 inputs (as Fisher Vector dimensionality) and 50% dropout regularization between first and second hidden layers. This model was trained on the Fisher Vector representation of the comments to perform classification on each label separately.

Observation: The average AUC obtained by training on the up-sampled dataset using MLP was between 0.95 and 0.97.

4.7 CNN

Convolutional Neural Networks (CNNs) are the most commonly used models in Image classification tasks. However, we thought it would be an interesting attempt to implement CNN in a Natural Language based problem. Based on the model developed by Yoon Kim [2] , we implemented CNN on the comment text to identify the toxicity of the comments. We used two types of representations, Bag-Of-Words and Google Word2Vec. We used the Word2Vec pre-trained Google News corpus (3 billion running words) word vector model (3 million 300-dimension English word vectors), matching each index relative to word, with its representation in the Google Word2Vec. This was used to build the CNN structure. Furthermore, in BOW, after removing stop words, we assigned one index to each remaining word. Each sentence was a vector containing the index associated with the respective words. This representation was further used to train the CNN. For this case we used convolutions with 1 dimension because it takes care of neighboring words.Images have height and width, so we use `conv2d`, sentences are linear lists of words, so `conv1d`. The Model design parameters are as follows:

- filter used (dimension of sliding windows) = 3,4,5
- 2 hidden layers with 128 and 64 filters.
- we also used dropout to improve our generalization performance.

The results of CNN are provided in the following table:

Label	AUC	Dropout	Filters used
Insult	0.962	50%	128 , 64
Toxic	0.971	50%	128 , 64
Obscene	0.949	50%	128 , 64
Threat	0.96	50%	128 , 64
Identity Hate	0.97	50%	128 , 64
Severe Toxic	0.95	50%	128 , 64

Table 4: CNN AUC's on up-sampled data

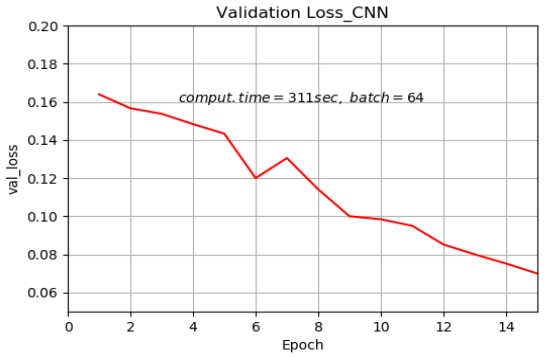

Figure 5: CNN Validation Loss on batch size 64

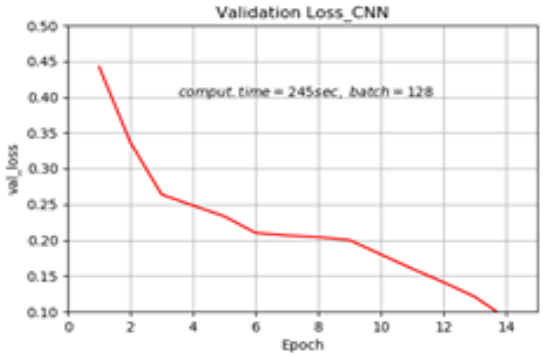

Figure 6: CNN Validation Loss on batch size 128

Observation: To obtain such high AUC we increased the number of epoch and due to the huge amount of data in input we used the parallel programming concept and GPU machines. The validation loss graphs with regard to one of the labels, "toxic", are shown below. The two graphs are based on different batch size considerations.

As shown in the graphs, increasing the batch size reduces the computational time, but the error rate increases slightly.

4.8 LSTM

Recurrent Neural Networks(RNNs) are the most widely used state-of-the-art algorithm for handling sequential data. As an extension to RNNs, the LSTMs (Long Short-Term Memory) are used as they help RNNs to preserve the error for *backpropagation through time*, thereby handling the vanishing/exploding gradient problems fairly well. The idea behind LSTM is to allow signals to propagate through time without changing them. [6] In this task, the original dataset was first tokenized and then indexed, in order to apply LSTM methodology, implemented using the `keras` [4] library. The input layer was a list of encoded sentences. In the LSTM layer, we chose a dimension of 60, and the whole unrolled sequence of sentences was returned. The output from this layer was a 3D tensor. *Global Max Pooling* layer was used to reshape it to a 2D tensor. Following this was the *Drop out* layer, the output of which was fed back to the *Dense* layer. This layer had the *ReLu* activation function. Later, the output was fed to the *Sigmoid* layer. To optimize the Loss function, the *Adam* [3] optimizer was used. With a default learning rate of 0.001 and a train-test split of 80-20% respectively, the AUCs obtained on two different epochs were 0.978 and 0.98. The results may be misleading in this context, but the reason for such high accuracies is the down-sampled data. Further classifiers were used on the up-sampled data.

[4]https://keras.io/layers/recurrent

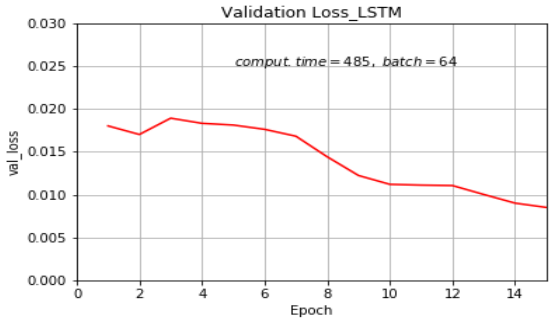

Figure 7: LSTM Validation Loss on batch size 64

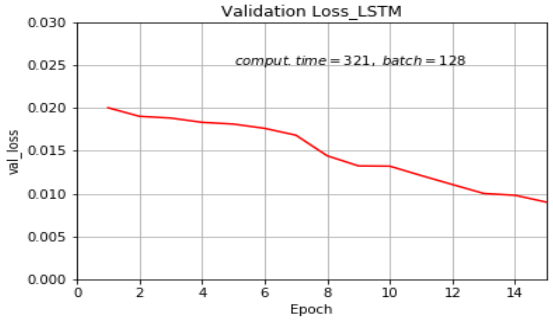

Figure 8: LSTM Validation Loss on batch size 128

Observation: The plots for validation loss on different batch sizes are shown below.

Also, the results from LSTM on up-sampled data, using mean-squared error and Adam optimization are shown in table 5.

Label	AUC
Insult	0.974
Toxic	0.971
Obscene	0.969
Threat	0.98
Identity Hate	0.95
Severe Toxic	0.96

Table 5: LSTM AUC's on up-sampled data

RESULTS: Table 6 shows the overall AUCs of each model, along with the type of data used. While there can be greater AUC values observed, it would be unfair to state that a particular model is the best. Various configurations, parameter tunings and minor tweaks in the algorithms can provide different values at each run.

Model	type of data	Embedding	AUC
Naive Bayes	Up-sampled	BOW	0.50
LR	Down-sampled	IF-IDF	0.63
LR	Up-sampled	FastText + FV	0.9821
SVM	Up-sampled	FastText + FV	0.9592
RF	Down-sampled	TF-IDF	0.77
RF	UP-sampled	BOW	0.50
RF	UP-sampled	TF-IDF	0.9615
RF	UP-sampled	FastText + FV	0.965
AdaBoost	UP-sampled	FastText + FV	0.97
MLP	UP-sampled	FastText + FV	0.95
CNN	Up-sampled	GoogleW2V+BOW	0.9604
LSTM	Up-sampled	BOW	0.9674

Table 6: AUCs on different models

5 CONCLUSION

The take-aways from this project are listed in this section. Models when trained on unbalanced data produced biased results. Thus, up-sampling/down-sampling the data set is necessary to avoid huge over-fitting scenarios. The best accuracies were observed on the up-sampled data, which was later translated to other languages. Though there can be several techniques to balance the data, we explored quite a few of them to see significant differences in results.

Appropriate pre-processing techniques are important for efficient training. With the growing popularity of Neural Network models like CNN, LSTM and MLPs, they are the preferred models for multi-label classification tasks. However, we obtained remarkable accuracies with simpler models like Logistic Regression and Random Forest. This implies that if the data is pre-processed in a meaningful way, training on Neural Networks is not the only option. In our experiment, the Fisher Vector representation when applied to Logistic Regression, produced a frequency of about 0.98 (Table 2) on the up-sampled data. The same data settings when applied to an LSTM network, produces an accuracy of about 96%.

In terms of models, there cannot be one specific name that can be mentioned as the best approach. We did experience greater accuracies with LSTMs, but it has its own complexities like GPU's requirement. Alternatively, CNNs showed good accuracies too, but they are complicated to adapt for text data, as they are designed mainly for images. On the other hand, training LSTMs was easier. Additionally, Logistic Regression provided accuracies up to 98%. Thus we can infer that if the pre-processing techniques (cleaning, normalization, vectorization and dimensionality reduction) are applied accurately, less complex models at times yield better results. Also, with different configurations, different results can be achieved, which establishes the fact that training machines is an empirical process.

6 CHALLENGES

As this is a Multi-label classification problems, we did face quite a few challenges. Some of them are:

- Unbalanced data
- A number of comments contained characters like IP addresses(e.g. 62.158.73.165), user names (e.g. ARK-JEDI10) and some other similar characters. These *Leaky Features* can cause huge over-fitting.
- For many pre-processing operations (like spell checking, translation, removing stop words) took the entire dataset as input and was annoyingly slow. To handle this, we parallelized these functions to accelerate the process. To do so, we used the `ipyparallel` [5] Python module. Also, for accelerating the training and other computations involved, we used a GPU machine which significantly reduced the training time.
- Using *Word2Vec* model for words embedding was not an appropriate choice because we can't make any representation for words that we didn't have in its vocabulary. It led us to using an alternative library called Fasttext.

7 FUTURE EHNACEMENTS

As an improvement, *Model Blending* can be attempted. All the above models can be stacked together and the sequential results can be voted on. Also, bi-directional *Gated Recurrent Units* (GRUs) can be used as a variation of LSTMs. This could handle the vanishing gradient problem in a different way, or probably provide better results.

REFERENCES

[1] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[2] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).

[3] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[4] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[5] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[6] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

[5] https://ipyparallel.readthedocs.io/en/latest/