Final Report: Classification of Probable Disaster Tweets

NEERAJ ADHIKARI AND ARCHANA CHITTOOR

ABSTRACT

Twitter has become an important communication channel in times of emergency, perhaps more so than any other major social media platform. The ubiquity of smartphones enables people to announce an emergency they observe in real-time. Because of this, more and more agencies are interested in programatically monitoring Twitter (i.e. disaster relief organizations and news agencies) so that early detection and response to disasters of all kinds is possible. However, its not always clear whether a tweet is talking about an actual disaster. For instance, if someone tweets âĂIJTHE SKY IS ABLAZEâĂİ the overwhelming chance is that it is meant just metaphorically. This is clear to a human right away, especially with visual aids like images and the extraordinary ability to detect subtle cues from the style and manner of writing. But for a machine, such a statement is ambiguous. Perfectly detecting every time whether a statement is meant in the literal sense or figurative falls squarely in the realm of general AI.
This project attempts to determine if a machine learning algorithm can discriminate tweets that are talking about real disasters vs tweets that are not with an acceptable level of reliability. The objective of the project is to build a deep learning model that classifies tweets into one of two classes - disaster tweets and non-disaster tweets.

ACM Reference Format:
NEERAJ ADHIKARI and ARCHANA CHITTOOR. 2020. Final Report: Classification of Probable Disaster Tweets. 1, 1 (May 2020), **??** pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The purpose of the project is to classify tweets into one of two categories — disaster tweets and non-disaster tweets. Disaster tweets are tweets that refer to or talk about an actual disaster (natural or artificial) such as an earthquake, volcano, wildfire, railway crash, etc. Non-disaster tweets may be talking about anything else. In a lot of cases it is hard to distinguish these two classes apart using simple methods because often both types of tweets use the same words. Non-disaster tweets may contain sarcasm, irony, metaphors and figurative manners of speech which has to be distinguished from tweets that seriously refer to disasters.
We have applied various Deep Learning techniques and evaluated each to determine which one gives the best accuracy and throughput. We also investigated different combinations of all these models and came up with an ensemble which provides the optimum accuracy while giving a reasonable performance.

Author's address: NEERAJ ADHIKARI and ARCHANA CHITTOOR.

XXXX-XXXX/2020/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 RELATED WORK

There has been a lot of related work done as this is a Kaggle project [**?**]. We have explained about a couple of solutions that we found were interesting and provided good models to compare our work with.

(1) Using Machine Learning Algorithms [2] In this work, various Machine Learning techniques such as Logistic Regression, SVC, Decision Trees, KNN and Random Forest have been used to perform the disaster tweet classification. The methods have been applied preprocessing and vectorization of the tweet texts. Out of all the ML techniques, Logistic Regression provided the best accuracy which is 79%.

(2) Using BERT Transformer [3] This work involves the usage of BERT which is the encoder part of the transformer. In this project, first tokenization is performed, and sentences are transformed into a pre-dictated format. The inputs are then fed into the BERT model and then to a Logistic Regression model. The complete model produces around 83% accuracy.

3 METHODOLOGY

In this section, we describe each of the models that we implemented and the processes involved therein.

3.1 Preprocessing

3.1.1 Data Cleaning. The first thing we performed was data cleaning. The data is very 'unclean' in the sense that it contains the raw text of tweets, along with hashtags, URLs, username mentions, grammatical and typographical errors and colloquial forms of writing. It also contains a lot of mis-encoded characters and non-printable characters. Cleaning of the data involved removing URLs, username mentions, mis-encoded characters and splitting condensed hashtags into its constituent words using a library called wordninja [**?**].

3.1.2 Tokenization and Lemmatization. After the cleaning of data, we performed tokenization of the text. To perform tokenization we used the SpaCy[**?**] python library. Tokenization enabled us to convert words from their encountered forms to their lemmatized forms (i.e. their root forms). It also enabled us to eliminate stop-words (words that appear too often in English to carry significant information) from the tweets. After cleaning, tokenization, lemmatization and elimination of stop-words, the tweets became much more amenable to classification methods.

3.1.3 Feature Extraction. Up until this point, even after a lot of processing on the input data, we still have a list of words for every data instance instead of having a vector or series of vectors. Since almost all the machine learning models need input data in a numerical form, we used pre-trained word embeddings to convert our words to word vectors. The specific emebeddings we used was GLoVe[**?**] with 300-dimensional vectors trained on the Common Crawl English corpus. Obviously our sentences were variable-length, but we took the length of the longest sentence and used that as the length of all sentences, padding the others with zero vectors. This gave us a nice 3-dimensional vector of our entire training set, which could be easily fed into our machine learning models.

3.2 Machine Learning Methods

3.2.1 Multilayer Perceptron. The first machine learning model we attempted to use was Multilayer Perceptron (MLP). After computing a single vector of each tweet by summing its word vectors, we trained a Multilayer Perceptron with two hidden layers of sizes 500 and 50 on them. The scikit-learn library was used for the training and inference, with all the default settings (ReLU activations, Adam optimizer, initial learning rate of 0.001 etc.) for the MLP classifier except the hidden layer sizes.

With 5-fold cross validation, we were able to obtain an average F1 score of 68.37%. F1 score was computed instead of other metrics because the Kaggle Competition uses it as the scoring metric. Because of the relatively low F1 score, we dropped this model and did not use it in our final implemyentation.

3.2.2 K Nearest Neighbors. We used a simple K-Nearest Neighbors model to establish a baseline of performance for our Deep Learning models. We used Scikit-learn for the implementation of this model. The feature vector for KNN was computed by summing the word vectors of all the words in a sentence. The value of k was set to 7 as that seemed to provide reasonably good results.

3.2.3 Convolutional Neural Network. We then tried a Convolutional Neural Network Model (CNN) implemented in PyTorch. Unlike the CNNs used in computer vision tasks, where there are a lot of 2-dimensional convolutional layers, our network had 1-dmensional convolutions. That was because our data had no concept of height and width, instead we only had a length of the sentence.

Here we treated the 300-dimensional word vector as input activations in 300 different channels. Through trial and error we reached the following layer composition of our CNN model.

(1) Size-3 convolutional layer with 300 input channels, 4 output channels and ReLU activation.
(2) Size-5 convolutional layer with 4 input channels, 8 output channels and ReLU activation.
(3) A fully connected layer with input as all channels of previous layer, 40 outputs and ReLU activation.
(4) A fully connected layer with 40 inputs, 2 outputs (corresponding to two classes) and softmax activation.

It was found that changing the number of layers (both convolution and fully connected), the sizes of the layers and use of pooling didn't make significant differences toward the final F1 score. We trained the network with the adam optimizer and used the entire trainig data instead of using mini-batches or Stochastic Gradient Descent because the network was small and could be trained in a short time.

3.2.4 Long Short-Term Memory Network. To take our classification to the next level, we used a Long Short-Term memory network with the expectation that retaining the word history instead of just looking at a sliding window of words would be better for classification. Like Convolutional Neural Network we again used PyTorch for our implementation here. Since training of LSTM is computationally intensive and thus time-consuming, we only used a single LSTM cell (single layer). The hidden state (and cell state) size was 300, same as the size of our word vector. We also used a two fully connected layers after the LSTM layer. Our layer architecture is as follows:

(1) (Bidirectional) LSTM cell with hidden state of size 300 and cell state of same size.
(2) Fully connected layer with input of size 300 (input would be sum of final hidden states in both directions), output of size 30 and ReLU activation.
(3) Fully connected layer with input of size 30, output of size 2 (corresponding to the number of classes) and softmax activation.

It was found that changing the hidden state and cell state dimensions didn't significantly alter the performance of our model. We decided against using multiple layers of LSTMs because that would increase the already large training time.

3.2.5 Ensemble Model. To make use of our models simultaneously and improve performance by a notch, we decide to use our last three models (KNN, CNN and LSTM) in an ensemble. We use a majority voting classifier with all three votes equally weighted. The final predictions for submission into the Kaggle competetion were generated using the ensemble model.

4 EXPERIMENTS, RESULTS AND ANALYSIS

4.1 Data Description

Our data comes from a Kaggle competition (*Real or Not? NLP with Disaster Tweets*[**?**]). The competition was started by Kaggle itself and the dataset was compiled by a company called *figure-eight*. The dataset contains about 10000 tweets and they are hand labeled. Given data set contains 3 files:

(1) train.csv - the training set
(2) test.csv - the test set
(3) sample_submission.csv - a sample submission file in the correct format

Each sample in the train and test set has the following information:

- The text of a tweet
- A keyword from that tweet (can be blank)
- The location the tweet was sent from (can be blank)

Features in the train data file:

- id - a unique identifier for each tweet
- text - the text of the tweet
- location - the location the tweet was sent from (may be blank)
- keyword - a particular keyword from the tweet (may be blank)
- target - in train.csv only, this denotes whether a tweet is about a real disaster (1) or not (0)

Even though we have 'location' and 'keyword' fields in the CSV files, in the dataset those fields are non-empty only in a very small proportion of the instances. So we made the natural decision to not use location and keyword fields but only use the 'text' field for classification, which already contains more than enough information for a human to make correct labelings.

As mentioned before, the data is very noisy and needs significant processing before word vectors can be extracted from them. As an example, here are a few samples of the text field of the instances.

```
#??? #?? #??? #??? MH370: Aircraft debris found on La Reunion is from
missing Malaysia Airlines ... http://t.co/5B7qT2YxdA
```

```
Barbados #Bridgetown JAMAICA ÂĽÃŽÃŠ Two cars set ablaze: SANTA CRUZ
ÂĽÃŽÃŞ Head of the St Elizabeth Police Superintende... http://t.co/wDUEaj8Q4J
```

The above two are examples of tweets that are referring to real disasters (positives). In contrast, the two below are not talking of disasters (negatives).

```
Cramer: IgerÂĽÃŽÂłs 3 words that wrecked DisneyÂĽÃŽÂłs stock ÂĽÃŽÃŠÃěÃŁCNBC
http://t.co/PnlucERp0x,
```

```
@WBCShirl2 Yes God doessnt change he says not to rejoice over the fall
of people or calamities like wild fires ect you wanna be punished?
```

-	1	3	5	7
1	0.434	0.498	0.734	0.539
3	0.726	0.627	0.722	0.721
5	0.729	**0.749**	0.725	0.738
7	0.708	0.742	0.749	0.723

Table 1. F1 scores for different values of layer 1 filter size (rows) and layer 2 filter size (columns).

4.2 Experiments

In the process of tuning various hyperparameters of the different models, we performed a vareity of experiments. Most of the experiments were conducted on the CNN model as it was the fastest to train and had plenty of tuneable hyperparameters. All of the following experiments were conducted on a Lenovo IdeaPad 520s device with an Intel Core i7-8550U running at 1.80GHz and with an Nvidia GeForce 940MX GPU. The training and inference for LSTM and CNN was performed on the GPU and the KNN inference was performed on the CPU.

4.2.1 Selection of filter sizes on CNN. We performed 3-fold cross validation across the entire training dataset and evaluated the target score (F1 score) for a range of filter sizes from 1 to 7. To reduce complexity of model and combinatorial explosion, we didn't tune the number of layers except in the initial trial and error phase.

In Table **??**, we tabulate the F1 scores observed for different values of the filter sizes. Admittedly there is not a large variation in scores, but some filter sizes are clearly bad (like a filter size of 1 for the first filter). The best combination was found to be 5 for the first filter and 3 for the second filter.

4.2.2 Selection of channel sizes on CNN. Again, to select the channel sizes, We performed 3-fold cross validation across the entire training dataset and evaluated the target score (F1 score) for a range of filter output channel sizes from 2 to 24. In this stage of the experiemnt we used the best values from the previous experiment (best filter sizes).

In Table **??**, we tabulate the F1 scores observed for different values of the channel sizes. Despite trying a large number of values we don't see a large variation in the scores. There aren't even clear candidates for bad channel sizes like we had with filter sizes. Our conjecture for why this might be happening is that even a small number of channels are already enough to capture all the complexity picked up by the small filters. As we can see in the table, the best combination was found to be 4 for the first filter and 8 for the second filter.

-	2	4	6	8	10	12
2	0.722	0.521	0.700	0.619	0.470	0.600
4	0.704	0.739	0.736	**0.752**	0.746	0.747
6	0.526	0.742	0.728	0.729	0.742	0.748
8	0.723	0.743	0.705	0.729	0.749	0.728
10	0.720	0.725	0.730	0.731	0.741	0.741
12	0.733	0.723	0.740	0.747	0.719	0.749

Table 2. F1 scores for different values of layer 1 channel size (rows) and layer 2 channel size (columns).

4.2.3 Selection of layer count and hidden state sizes on LSTM. To select the optimum hidden state size and layer count for our LSTM model, again performed a similar search as for CNNs,

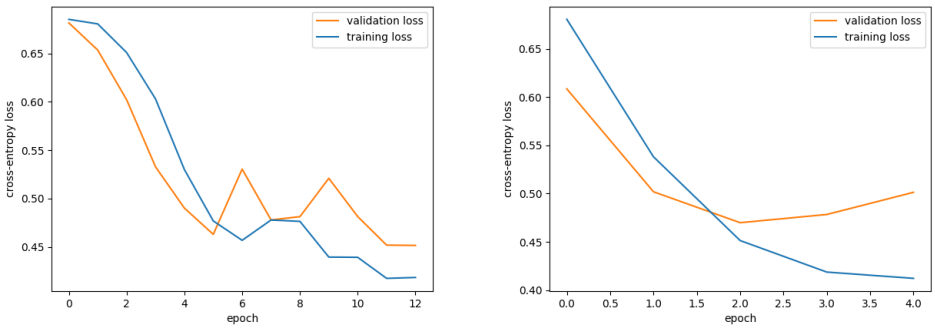
tabulating the cross-validation performance across different combination of the hyperparameters. We tried only a small range of the layer counts (1 to 3) because both inference and training on LSTMs is much more computationally expensive than CNNs and increasing layer count makes this complication worse.

In Table **??**, we tabulate the F1 scores observed for different values of layer count and hidden state dimension. As we can see in the table, the best combination was found to be 2 layers and a hidden state dimension of 300. It is interesting to note that we found the best values to be the maximums of our search range. But due to the computational resources and time required to search for more optimum values, we decided to stop at these hyperparameters.

-	20	40	60	90	150	300
1	0.699	0.648	0.704	0.691	0.704	0.717
2	0.669	0.683	0.666	0.696	0.724	**0.735**

Table 3. F1 scores for different values of layer count (rows) and and hidden state dimension (columns).

4.2.4 Training the models. After tuning the hyperparameters we trained the models on our full data set, and observed the trends of training loss and validation loss. Both losses here are cross-entropy loss. As we can see in figure **??**, the validation loss of CNN stops decreasing after the 6th epoch and begins increasing due to overfitting. Similarly, in figure **??**, we can see that the validation loss of LSTM starts increasing right after the 3rd epoch, indicating that it overfits even faster.



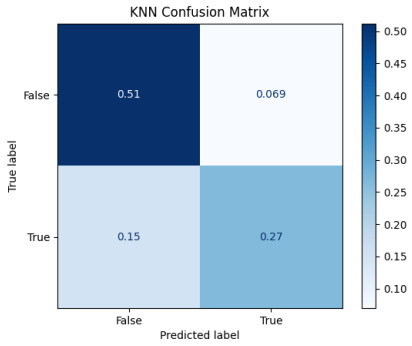
CNN Training Curve LSTM Training Curve

Fig. 1. Testing and Validation losses for CNN and LSTM Training.

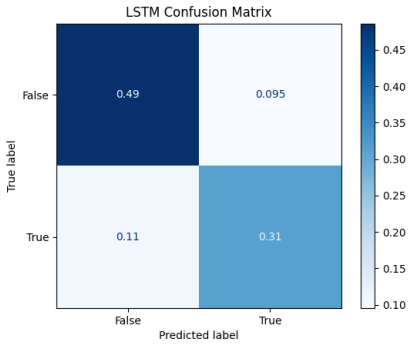
4.2.5 Observing the class distribution of the predictions. After training the model, we performed a hold-out cross validation to observe the class distribution of the predictions, i.e. to observe what mistakes the predictors were making. For this experiment we used a randomly chosen 75% subset of the training data for training and the rest for validation.

As we can see from the confusion matrices presented in Figure **??**, the different models vary slightly in their predictions. In general, all the classifiers seem to be reasonably good at labeling the negative samples as negative, except for maybe the CNN classifier. Here we analyze the observed confusions of different classifiers.

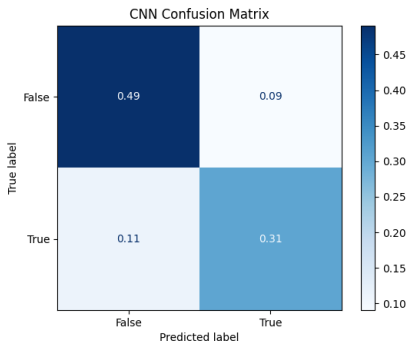
(1) **KNN classifier:** The KNN classifier apparently finds it hard to correctly classify positive instances, as there are a lot of false negatives. Its performance on the negative instances is unexpectedly high.
(2) **LSTM classifier:** The LSTM classifier makes very good predictions compared to other models, which is to be expected. It makes a roughly equal amount of Type-I and Type-II errors.
(3) **CNN classifier:** The CNN classifier for some reason seems to be more perplexed than even the KNN on classifiying negative samples. On the other hand, it performs the best among all classifiers in classifying positive samples. We couldn't find a good explanation for this phenomenon.
(4) **Ensemble classifier:** As we expected, a majority vote-based ensemble gives us a very balanced and 'best of all' classifier, as its mistakes are more or less even across the two types.



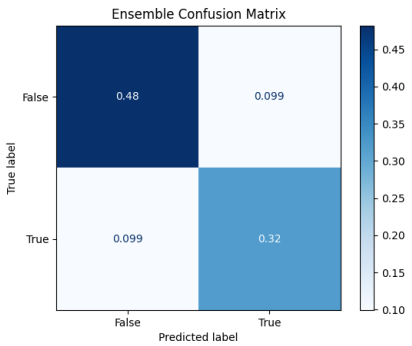
(a) KNN confusion matrix



(b) LSTM confusion matrix



(c) CNN confusion matrix



(d) Ensemble model confusion matrix

Fig. 2. Confusion matrices for different predictors

4.3 Kaggle Score

In the end, we submitted a final set of predictions from our ensemble model to Kaggle. Our submission obtained a score of 0.79447. This is not a great score in the grand scheme of things, but it

is certainly very good considering our cross-validation scores were generally 4-5 percentage points lower than that. Also, since this competition scores submissions based on the F1 score (harmonic mean of precision and recall), and given the fact that F1 scores are generally lower than accuracies, our accuracy is probably higher.

5 CONCLUSION