Classification of Disaster Tweets using NLP

Team: Manasa Hari, Amit Kamboj, Feiyu Cai, Tushar Sharma

**Abstract** — Twitter has been the most important part of digital communication these days. It has been the channel of communication across the world for daily news starting from television gossips to case-of-emergency news. But due to large amounts of data flooding into the twitter base every minute, sometimes, a highly important piece of news such as a disaster occurrence can be submerged amidst not at all important tweets. Due to this, it may take days or even weeks before the disaster news reaches to people, meanwhile, a huge irrevocable loss can happen already. This project aims to highlight the disaster emergency news so that users can be constantly notified on the emergency issues out of millions of other tweets. We take the challenge to build a machine learning model that classifies between tweets about real disasters and the rest. The key challenge is to distinguish metaphorical usage of tragedy vocabulary and the real intended usage of disaster terms. For example, a user tweets ‘Thoughts are a storm, unexpected’. This is clearly a metaphorical statement. Even though it is obvious for humans to interpret that this tweet is not about a real disaster, but it is less clear to a machine. This problem is also an actively ongoing Kaggle competition. We want to explore possible predictors and conclude on the right predictors to solve this classification problem. The dataset has 10,000 tweets that were hand classified[1]. We will explore Naïve Bayes, LSTM and CNN classifiers to solve this problem.

1. **Introduction**

This report describes the step by step process of building a recurrent neural network (RNN) that classifies tweets to one of two potential classes: Disaster related tweet or Not Disaster related tweet. A basic implementation of a Long Short-Term Memory Network (LSTM) along with a specialized embedding matrix is used in this method. We also explored the variation of test score with input batch size and optimization iterations in this process. We also used a 10-fold cross validation logistic regression with bag of words model which was modified to classify the tweets. This attributes to the inherent ’features’ that the models emphasize.

1. **Literature review/related work**

In this project, our goal is to make a classification that predicts is the tweet disaster or not. We chose to use neural networks over other classifiers for this project regarding our data input is going to be English words that are way more complicated than just numbers and word of characteristics. Different sentence structures and context can make the same word have different meanings. According to Shamina and Banerjee state in the article Improved Speech Inversion Using General Regression Neural Network, “It was shown that a three-layered feedforward neural network (NN) can perform better speech inversion than trajectory mixture density models, support vector regression, autoregressive NN, and distal supervised learning.” For this reason, we decided to use the neural network, which is very effective for high dimensionality problems, able to deal with complex relations between variables, non-exhaustive category sets, and complex functions relating input to output variables. Here is an example given by Finch and Schneider in Classification accuracy of neural networks vs. discriminant analysis, logistic regression, and classification and regression trees: Three- and five-group cases when comparing NN with common regression “If, for example, two variables interact and none of the others play a role, then the hidden layer would be represented by large weights for each of the two and near 0 weights for the others. On the other hand, a hidden layer could be thought of as the combination of several of the predictors with some contributing slightly more and, thus, having slightly larger weight values.” The strengths of NN is creating a hidden layer that contained nodes as weighted products of variables while expressing the interactions among the predictor variables.

With all the neural network methods, we chose the Recurrent Neural Network (RNN) with Long Short-Term Memory (LSTM) for this project. According to Hsu, Zhang, and Glass state in the paper A prioritized grid long short-term memory RNN for speech recognition that “Recurrent neural networks (RNNs) are naturally suitable for speech recognition because of their ability of utilizing dynamically changing temporal information.” First of all, RNN is a generalization of feedforwarding neural network that has an internal memory, which means RNN performs the same function for every input of data with the output of previous outputs. After every computation, the output is copied and sent back into the RNN function internal state (memory). For this reason, this characteristic makes it better for speech recognition while all inputs are related to each other.

A screenshot of a cell phone

Description automatically generated

Figure 1: RNN block model

However, the internal state cannot last forever. RNN has gradient vanishing while the process takes too long. Gradients are values used to update neural network weights, which will shrink during back-propagation. When the gradient becomes too small, it doesn't contribute to the learning process. So, we use LSTM to resolve this problem. LSTM is a modified version of RNN, which improved in remembering past data in memory. According to Hsu, Zhang, and Glass state in the paper A prioritized grid long short-term memory RNN for speech recognition that “…more importantly, a gated linear dependence is introduced between memory cell states across two consecutive time steps which allows memories to be preserved.” LSTM uses back-propagation to train the model within the LSTM's cell which is better when classify, process, and predict time series given time lags of unknown duration. In each LSTM cell model, 3 gates presented to help to decide what information to keep or throw away:

* Forget gate: used to decide what detail can be discarded from the block
* Input gate: used to decide which input should be used to modify the memory
* Output gate: used to calculate the cell state

A picture containing clock

Description automatically generated

Figure 2: LSTM block model

1. **Project Architecture**

Supervised learning – classification problem:

This problem is clearly a classification algorithm. It has only two classes (Class 1 – represents that is a disaster identified tweet and Class 0 – represents that is not a disaster tweeted about). Since, we are provided with the known labels in the training set, this is considered as a supervised problem. So, in total, we call this problem as a supervised classification problem.

We want to build a model of these examples, then use that model to make predictions. Hence, this is model-based learning.

We are going to follow six major steps in this project:

1. Study the data
2. Data pre-processing
3. Develop a model for the data
4. Train the model on the training data
5. Finally, apply the model to make predictions on new cases (inference)
6. In this process, we strive for better accuracy and hope that this model will generalize well. We will fine-tune the model, while studying the data.

**Step 0: Setting up the environment and libraries:**

We used Jupyter notebook for this project.

A screenshot of a cell phone

Description automatically generated

**Step 1: Study the data**

We got the dataset from Kaggle.com. <https://www.kaggle.com/c/nlp-getting-started/data>

The given data comprises of train.csv and test.csv. We are supposed to study the data and discover a model using. Below are the steps to load the data.

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

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

Here, the given datasets are 1) train.csv and 2) test.csv files.

Train.csv:-

A screenshot of a social media post

Description automatically generated

Test.csv:-

A screenshot of a social media post

Description automatically generated

As seen above, there are some hyperlinks in the data. So, there is a necessity of removing the hyperlinks to clean up the data. In this data cleaning process,

**Step 2: Data Pre-processing**

A screenshot of a social media post

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a social media post

Description automatically generated

A screenshot of a cell phone

Description automatically generated

Important notes:

Normally, we perform the following operations too in a typical machine learning problem.

1. Feature scaling
2. Data visualization to gain insights
3. Regularization

Since this is a tweet classification problem using NLP, and given are string features, we would not do these above specified three operations.

**Step 2.5: One hot encoding**

ML algorithms usually assume that two nearby numbers have more similarity than two distant values. This can be okay in some cases like text labelled like: good, bad, worst, excellent categories. But it does not make sense in 0,1,2,3,4 categories. A simple fix to this problem is to create a binary attribute per category.

That is, when this attribute is equal to 1, it belongs to a particular category. If it does not belong that category, the attribute is set to 0.

This is called **One-hot encoding**. In this type of encoding, only one attribute will be 1 (hot), while the others will be 0 (cold).

We used to\_categorical from keras.utils.np\_utils library to do this operation.

A screenshot of a social media post

Description automatically generated

We could see that now, two *dummy* attributes created. Only one of them will be representing the real category. Here, the number of *dummy* columns created is equal to the number of classes represented by the target column.

As an added step, we also built a reverse encoder to later convert the one-hot encoded values to the original required target values.

**Step 2.6: Reverse Encoder**

A screenshot of a cell phone

Description automatically generated

**Step 2.8: Tokenizer – Hash encoding**

We did research on different encoding techniques provided by Keras.

Why we need to encode a document:

Raw data cannot be fed directly into machine learning models. Text data must be encoded as numbers to use as input or output in a machine learning model.

In this project, we explored the path of encoding text using Keras library.

As a first step, we split the text into words using text\_to\_word\_sequence() function.

By default, this function does 3 things as below: [4]

1. Splits words by space (split=” “).
2. Filters out punctuation (filters=’!”#$%&()\*+,-./:;<=>?@[\\]^\_`{|}~\t\n’).
3. Converts text to lowercase (lower=True).

**Step 2.7: Tokenizer - Text pre-processing with Keras**

One of the most sophisticated methods provided by Keras API is the Tokenizer class.

It not only is used to prepare the text to fit, but also can be reused to prepare multiple other text documents. This API also supports large datasets.

We first need to construct this Tokenizer class and then use it to fit our raw text.

We create the constructor of this class using the following code snippet

Code: t = Tokenizer()

A screenshot of a cell phone

Description automatically generated

Once we created the constructors, we fit the documents as below.

A close up of a logo

Description automatically generated

In the following, we can observe what was learnt about the data using Tokenizer.

Tokenizer class’s fit provides the below four attributes: [4]

1. word\_counts: A dictionary of words and their counts.
2. word\_docs: A dictionary of words and how many documents each appeared in.
3. word\_index: A dictionary of words and their uniquely assigned integers.
4. document\_count:An integer count of the total number of documents that were used to fit the Tokenizer.

A screenshot of a newspaper

Description automatically generated

A close up of a newspaper

Description automatically generated

We used variable length tokenization here. We used Keras library for creating a vector of works in every tweet. The vectors are padded up to 50 (here, as we defined), to be the limit of number of possible words in a 140-characters length sized tweet.

We considered approximately, 150/50 = 3 characters per word (which includes a space).

Below is how we performed integer encoding to both the training and testing documents.

A screenshot of a social media post

Description automatically generated

Now, we are completed with the data pre-processing step. We will now look into building the model for this data.

**Step 3: Develop a model for the data**

We explored different models for this problem initially. After a long exploration, we found RNN (Recurrent Neural Networks) work better. RNNs are a class of neural networks that can predict based on input sequences of arbitrary lengths, rather than fixed sized inputs, unlike other nets. That means, they can take sentences, documents etc. as inputs to predict the targets. And that is why, they are extremely useful for natural language processing (NLP).

1. Testing methodology/Accuracy testing (5 pts)

**Predictions:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 | 0 | 0 |
|  | 0 | 0.1505156 | 0.8494844 |
|  | 1 | 0.09050537 | 0.9094946 |
|  | 2 | 0.015072056 | 0.984928 |
|  | 3 | 0.0039795944 | 0.9960204 |
|  | 4 | 0.01810527 | 0.9818948 |
|  | 5 | 0.34434786 | 0.65565217 |
|  | 6 | 0.91536677 | 0.084633164 |
|  | 7 | 0.88879395 | 0.11120602 |
|  | 8 | 0.7927213 | 0.20727876 |
|  | 9 | 0.86122763 | 0.13877234 |
|  | 10 | 0.8790129 | 0.12098707 |
|  | 11 | 0.8526885 | 0.14731154 |
|  | 12 | 0.85276455 | 0.14723547 |
|  | 13 | 0.8664383 | 0.13356169 |
|  | 14 | 0.86926454 | 0.13073544 |
|  | 15 | 0.91398925 | 0.08601075 |
|  | 16 | 0.6973401 | 0.30265993 |
|  | 17 | 0.48740608 | 0.51259387 |
|  | 18 | 0.90057504 | 0.099424995 |
|  | 19 | 0.8963903 | 0.10360971 |
|  | 20 | 0.7162792 | 0.28372073 |
|  | 21 | 0.74825865 | 0.25174132 |
|  | 22 | 0.88079435 | 0.11920561 |
|  | 23 | 0.6161239 | 0.38387606 |
|  | 24 | 0.791695 | 0.208305 |
|  | 25 | 0.8442739 | 0.15572602 |
|  | 26 | 0.90055424 | 0.099445716 |
|  | 27 | 0.7927861 | 0.20721382 |
|  | 28 | 0.8518568 | 0.14814316 |
|  | 29 | 0.027026983 | 0.97297305 |
|  | 30 | 0.84411454 | 0.15588543 |
|  | 31 | 0.78474003 | 0.21526 |
|  | 32 | 0.86785036 | 0.13214958 |
|  | 33 | 0.38570628 | 0.6142937 |
|  | 34 | 0.0017570416 | 0.998243 |
|  | 35 | 0.9125669 | 0.08743304 |
|  | 36 | 0.46324474 | 0.53675526 |
|  | 37 | 0.75315607 | 0.24684389 |
|  | 38 | 0.8243213 | 0.17567864 |
|  | 39 | 0.014520138 | 0.9854799 |
|  | 40 | 0.85757434 | 0.14242563 |
|  | 41 | 0.15367855 | 0.84632146 |
|  | 42 | 0.61276424 | 0.38723567 |
|  | 43 | 0.4745252 | 0.5254747 |
|  | 44 | 0.8907204 | 0.109279625 |
|  | 45 | 0.3176081 | 0.6823919 |
|  | 46 | 0.9421273 | 0.05787269 |
|  | 47 | 0.9461451 | 0.053854868 |

**Results:**

|  |  |  |
| --- | --- | --- |
|  | id | target |
|  | 0 | 1 |
|  | 2 | 1 |
|  | 3 | 1 |
|  | 9 | 1 |
|  | 11 | 1 |
|  | 12 | 1 |
|  | 21 | 0 |
|  | 22 | 0 |
|  | 27 | 0 |
|  | 29 | 0 |
|  | 30 | 0 |
|  | 35 | 0 |
|  | 42 | 0 |
|  | 43 | 0 |
|  | 45 | 0 |
|  | 46 | 0 |
|  | 47 | 0 |
|  | 51 | 1 |
|  | 58 | 0 |
|  | 60 | 0 |
|  | 69 | 0 |
|  | 70 | 0 |
|  | 72 | 0 |
|  | 75 | 0 |
|  | 84 | 0 |
|  | 87 | 0 |
|  | 88 | 0 |
|  | 90 | 0 |
|  | 94 | 0 |
|  | 99 | 1 |
|  | 101 | 0 |
|  | 103 | 0 |
|  | 106 | 0 |
|  | 108 | 1 |
|  | 111 | 1 |
|  | 115 | 0 |
|  | 116 | 1 |
|  | 122 | 0 |
|  | 123 | 0 |
|  | 124 | 1 |
|  | 125 | 0 |
|  | 127 | 1 |
|  | 140 | 0 |
|  | 142 | 1 |
|  | 147 | 0 |
|  | 148 | 1 |
|  | 150 | 0 |
|  | 152 | 0 |

**Probabilities:**

For iter 10

Accuracy 0.8046875

Loss 0.50922424

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 20

Accuracy 0.796875

Loss 0.4614792

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 30

Accuracy 0.8203125

Loss 0.40784174

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 40

Accuracy 0.7734375

Loss 0.4762222

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 50

Accuracy 0.8046875

Loss 0.42260525

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 60

Accuracy 0.7890625

Loss 0.44487348

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 70

Accuracy 0.8125

Loss 0.45680463

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 80

Accuracy 0.8203125

Loss 0.4178124

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 90

Accuracy 0.765625

Loss 0.47575513

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 100

Accuracy 0.8515625

Loss 0.33627287

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 110

Accuracy 0.78125

Loss 0.5046478

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 120

Accuracy 0.796875

Loss 0.49587548

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 130

Accuracy 0.84375

Loss 0.35237136

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 140

Accuracy 0.875

Loss 0.37459505

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 150

Accuracy 0.859375

Loss 0.34797686

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 160

Accuracy 0.8515625

Loss 0.40814596

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 170

Accuracy 0.828125

Loss 0.353389

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 180

Accuracy 0.8125

Loss 0.48322877

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 190

Accuracy 0.8359375

Loss 0.4301808

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 200

Accuracy 0.8125

Loss 0.42958814

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 210

Accuracy 0.8671875

Loss 0.3825365

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 220

Accuracy 0.8828125

Loss 0.3277074

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 230

Accuracy 0.859375

Loss 0.35490602

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 240

Accuracy 0.8359375

Loss 0.40459144

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 250

Accuracy 0.8203125

Loss 0.4415267

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 260

Accuracy 0.8125

Loss 0.41012534

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 270

Accuracy 0.8671875

Loss 0.32890195

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 280

Accuracy 0.8203125

Loss 0.43306082

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 290

Accuracy 0.8671875

Loss 0.3210754

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 300

Accuracy 0.8203125

Loss 0.42362547

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 310

Accuracy 0.8125

Loss 0.39847904

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 320

Accuracy 0.8671875

Loss 0.34221205

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 330

Accuracy 0.828125

Loss 0.40324652

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 340

Accuracy 0.859375

Loss 0.36731666

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 350

Accuracy 0.7421875

Loss 0.5234597

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 360

Accuracy 0.8125

Loss 0.40690982

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 370

Accuracy 0.8125

Loss 0.428889

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 380

Accuracy 0.8671875

Loss 0.35530594

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 390

Accuracy 0.875

Loss 0.31927136

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 400

Accuracy 0.8046875

Loss 0.43344277

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 410

Accuracy 0.8671875

Loss 0.31188732

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 420

Accuracy 0.8125

Loss 0.38951474

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 430

Accuracy 0.828125

Loss 0.40016627

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 440

Accuracy 0.78125

Loss 0.46216962

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 450

Accuracy 0.859375

Loss 0.35073376

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 460

Accuracy 0.8359375

Loss 0.413001

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 470

Accuracy 0.8125

Loss 0.38684937

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 480

Accuracy 0.7734375

Loss 0.48281783

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 490

Accuracy 0.8046875

Loss 0.47355783

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 500

Accuracy 0.828125

Loss 0.34480453

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 510

Accuracy 0.8046875

Loss 0.44361097

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 520

Accuracy 0.84375

Loss 0.35121757

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 530

Accuracy 0.8125

Loss 0.38925427

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 540

Accuracy 0.90625

Loss 0.34675968

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 550

Accuracy 0.8125

Loss 0.41969198

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 560

Accuracy 0.84375

Loss 0.3851005

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 570

Accuracy 0.859375

Loss 0.35212684

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 580

Accuracy 0.8046875

Loss 0.41215682

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 590

Accuracy 0.8359375

Loss 0.34320468

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 600

Accuracy 0.7890625

Loss 0.4188121

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 610

Accuracy 0.84375

Loss 0.38491297

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 620

Accuracy 0.8515625

Loss 0.36662343

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 630

Accuracy 0.90625

Loss 0.31803682

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 640

Accuracy 0.8203125

Loss 0.40634412

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 650

Accuracy 0.8359375

Loss 0.3846455

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 660

Accuracy 0.8359375

Loss 0.34824216

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 670

Accuracy 0.859375

Loss 0.31586075

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 680

Accuracy 0.8359375

Loss 0.42279512

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 690

Accuracy 0.8671875

Loss 0.3291077

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 700

Accuracy 0.8671875

Loss 0.3270688

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 710

Accuracy 0.859375

Loss 0.3034977

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 720

Accuracy 0.828125

Loss 0.3483694

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 730

Accuracy 0.78125

Loss 0.43891728

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 740

Accuracy 0.875

Loss 0.36716694

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 750

Accuracy 0.8671875

Loss 0.34643334

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 760

Accuracy 0.8515625

Loss 0.36824173

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For iter 770

Accuracy 0.7734375

Loss 0.4055602

1. Result, conclusion, future work detailed description (10 pts)

This project enabled us to explore the basic static LSTM and feed forward neural networks. We could classify a dataset of tweets into 'disaster related' and 'non-disaster related' classes. We used GloVe word vector based on Twitter corpus, in order to optimize the process. After that, we experimented by varying weight optimization iterations and LSTM input batch sizes to the test set scores. We compared the LSTM using a 10-fold cross validation model.

CMPE-257 Project Developed in Spring 2020

—————————— ◆ ——————————

**References**

1. Data and Problem source: <https://www.kaggle.com/c/nlp-getting-started/data>.
2. Mittal, Aditi. “Understanding RNN and LSTM”. Oct 12, 2019. <https://towardsdatascience.com/understanding-rnn-and-lstm-f7cdf6dfc14e>
3. Phi, Michael. “Illustrated Guide to LSTM’s and GRU’s: A step by step explanation”. Sep 24, 2018. <https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21>
4. <https://machinelearningmastery.com/prepare-text-data-deep-learning-keras/>
5. https://nlp.stanford.edu/projects/glove/

**Dataset and Description**

Link: <https://www.kaggle.com/c/nlp-getting-started/data>

1. Data format: Each sample in the train and test set has the following information:
2. The text of a tweet
3. A keyword from that tweet (although this may be blank!)
4. The location the tweet was sent from (may also be blank)
5. Files we have:
6. train.csv - the training set
7. test.csv - the test set
8. sample\_submission.csv - a sample submission file in the correct format
9. Columns we use:
10. id - a unique identifier for each tweet
11. text - the text of the tweet
12. location - the location the tweet was sent from (may be blank)
13. keyword - a particular keyword from the tweet (may be blank)
14. target - in train.csv only, this denotes whether a tweet is about a real disaster (1) or not (0)
15. Output: We will predict whether a given tweet is about a real disaster or not. If so, predict a 1. If not, predict a 0.

**Technologies used:** Python, Keras, Tensorflow.