[[1]](#footnote-1)

NLP WITH DISASTER TWEETS

1st Akash Indani, 2nd Nikhil Khaladkar

# MOTIVATION

Nowadays, social networking sites hold more information than any other source. People immediately share their views, opinions, and information on different social networking website. This makes these websites an important data source to implement data mining. In our project, we are classifying whether tweets are disaster related or not, using NLP techniques, focusing on different word/sentence embeddings to increase accuracy of classification. After the classification, we extract entities such as country, number of people and location, to get an insight about the tweets.

To classify any text, we need to convert them to vectors, and we have used different methodologies t do so. First, the most important and fundamental word embedding technique that we are using is Word2vec, used to perform the vectorization of words [1]. We have used another method known as Doc2vec, which focuses on the sentence vector for prediction [2]. Further, we have also built a recursive neural network, LSTM, which gives a state-of-the-art result [3]. Just to compare the results we have obtained, we have tried different pre-trained models by Google, GloVe and Spacy to see the classification results. Once we have the word/sentence embeddings from any of these methods, we use various classification algorithms like Logistic regression, Linear support vector machine and KNN, to classify the tweets.

# RELATED CONCEPTS & APPROACH

In this section we discuss the various word embedding techniques that we used and the approaches that we took.

1. ***Word2vec***

In 2013 Mikolov introduced word2vec with two model architectures, Continuous Bag of Words model and Skip-gram model [1]. In the CBOW model, the distributed representations of context (or surrounding words) are combined to predict the word in the middle, while in the Skip-gram model, the distributed representation of the input word is used to predict the context. As Mikolov quoted, Skip-gram model functions slower than CBOW model, but it works

well with small amount of training data and well represents even rare words or phrases. This is why the skip-gram model was selected, keeping in mind the small data set.

Skip-gram Model:

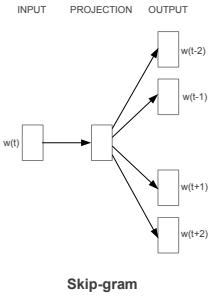
The word2vec model tries to optimize a supervised classification problem. Specifically, given a “context word”, we intend to train the model in such a way that, the model can predict a “target word” which appears within a predefined window size from the context word.

E.g. “After the deduction of the costs of **investing**, beating the stock market is a loser’s game”

The underlined phrases indicate a window size of 5 and the word in bold is the context word which is ‘investing’.

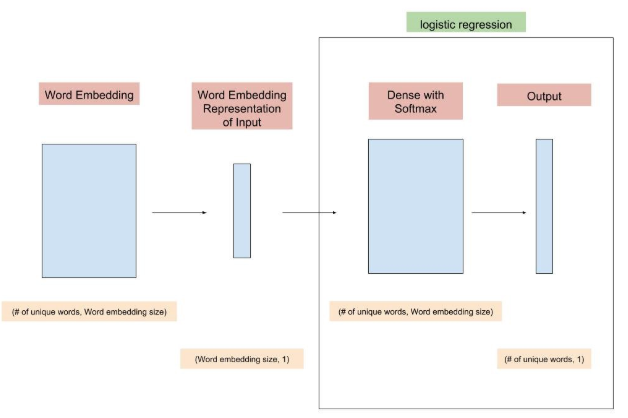
For the given context word, we intend to find the underlying word from – ‘deduction, of, the, costs, beating, the, stock, market, is’

Below is the original diagram from the Mikolov’s paper which suggests the same that we have explained in the above example [1]:



*Fig 1: Word2Vec SkipGram Model [1]*

A closer look at the whole process:



*Fig 3: The Word2Vec classification Process*

The word-embedding layer is essentially a matrix with a shape = (# of unique words in the corpus, word embedding size). Each row of the matrix represents a word in the corpus. Word-embedding size is a hyper-parameter to be decided and can be thought of as the number features or dimensions that we would like to use, to represent each word. The latter part of the model is simply a logistic regression in a neural network form.

Steps that were followed: -

1. Creating the training data:
   1. Create a word list and tokenize them
   2. We then created the word pair based on index size, for e.g. consider the previous instances that we created - (investing, deduction), (investing, of), (investing, the), (investing, costs), (investing, of), (investing, beating), (investing, the), (investing, stock), (investing, market), (investing, is)
   3. For simplicity, we assigned the integer IDs to each word and made pairs of IDs.

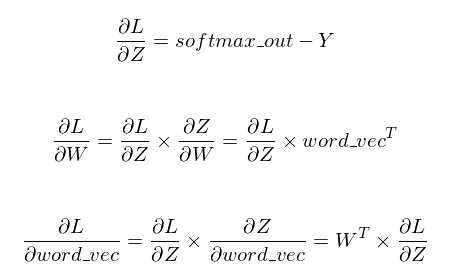
Next, a few important steps that are needed to be performed are: initializing weights (parameters that we want to train), propagating forward, calculating the cost, propagating backward, and updating the weights. The whole process will be repeated for several iterations based on how many epochs we want to train.

1. Initialization of parameters:
   1. Word-embedding layer: We initialize the matrix as (vocab\_size, emb\_size), which represents all the vocabularies with (vocab\_size \* emb\_size) matrix. Here, each row of the matrix represents a word from vocabulary.
   2. Dense layer: Input would be- (emb\_size, no of training examples), consider one training example and as one-hot encoder.
2. Forward pass:

There are three steps in the forward propagation, obtaining input word’s vector representation from word-embedding; passing the vector to the dense layer; and applying softmax function to the output of the dense layer. Here the input is the one-hot encoder.

Here, the dense layer is calculating for each word in the vocab, after applying the softmax function to each word appearing near the given input word.

1. Backward propagation:
   1. After the forward pass, we have to calculate the gradients of the weights with respect to loss function and seeing the difference update the weights with its associated gradients. It follows a chain rule:
   2. We calculate the gradients for the weights in the dense layer as we want to train them as



* 1. And we update the weights as

1. Training of model:

To train the model, we repeat the process of forward propagation, backward propagation and weight updating. During the training, the cost after each epoch should have a decreasing trend.

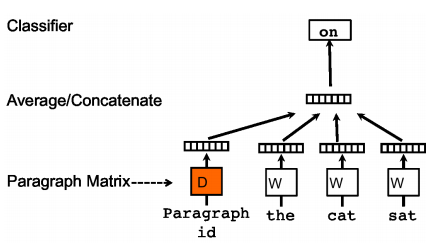
1. **Doc2vec**

Despite the popularity of word2vec models, they have two major weaknesses: they lose the order of the words and they also ignore semantics of the words. In the paper, Quoc Le and Tomas Mikolov introduced an easy, yet clever method to overcome these limitations [2]. They proposed to have a paragraph vector, an unsupervised algorithm that learns fixed-length feature representations from variable-length pieces of texts, such as sentences, paragraphs, and documents.

We used “Paragraph Vector: A distributed memory model” (PV-DM model), which suggests creating another paragraph vector as shown in figure 2. In this framework, every paragraph (sentence in our case) is mapped to a unique column-paragraph vector D and each word is mapped to a unique column-word vector W. The paragraph vector is shared across the same paragraph, but not across different paragraphs. However, the word vectors are shared across all paragraphs, for e.g. “investing” has same word vector across all the paragraphs.

The paragraph vector and word vectors are trained using gradient descent via back propagation as shown previously, in word2vec. As done in word2vec, we will calculate error in each step and update the parameters in our model. For predicting the new paragraph vector, we use the gradient descent, keeping rest of the model as it is.

We have used gensim library for building the doc2vec model



*Fig 4: This is the original image of PV-DM model taken from the paper [2]*

1. **Long Short-Term Memory Networks (LSTM)**

In the word2vec and other word embedding based approach we used the word averaging technique along with the BOW model where we averaged the weights of all the words in each sequence to represent the sequence in the vector space.

This approach however can be less accurate as it does not consider the sequence of the words in the sentence. For example, let us consider these two sentences:

*I will study and not play.*

*I will play and not study.*

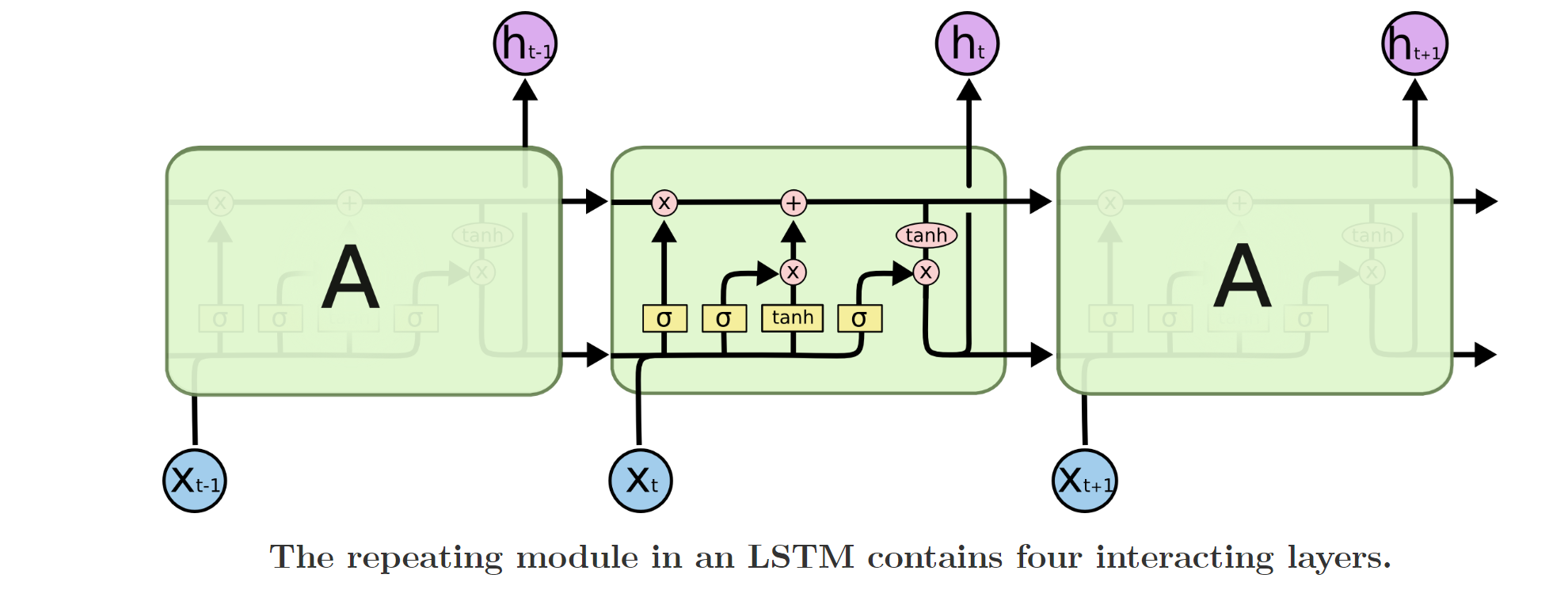
If we give these sentences to the averaging function and a word embedding model, then the averaging function will return the same averaged out weight for both these sentences as both the sentences have same set of words. Here, the sequence of the words is important to understand their meaning. By simply changing the sequence of the same words in the first sentence the second sentences become significantly different from the first sentence and hence should be distinct from each other when represented in a vector space.

Recurrent Neural Networks now come into picture. Recurrent Neural Networks or RNN’s are type of neural networks which can use current state output in the calculation of a future state. So, in simple words the RNN’s make use of historical data learned from previous nodes to calculate the state or output of the current node. However, RNN’s have the following issue:

First since the inputs to the RNN’s are in temporal order the output of an RNN node mostly depend on the previous context and cannot consider future context during output generation [3]. Also, they have difficulties to learn time-dependencies more than a few time footsteps [3]. This is called the vanishing gradient issue where after a few passes through the neural network the previous context is lost. These shortcomings are overcome by the LSTM or Long Short-Term Memory Networks.

LSTM’s architecture has blocks that are set of recurrently connected units. Each LSTM block consists of one or more self-connected memory cells along with input, forget, and output multiplicative gates. The gates allow the memory cells to store and access information for longer time periods to improve performance [4].

Here we summarize our understanding of the working of a LSTM [4,5]:



*Fig 5: LSTM Representation [5]*

1. In the first step the node decides what information to discard away from the cell state which has the historical information. This is done by passing the cell state information through an activation function of sigmoid. An activation function is a function which takes in a single or multiple input and maps it to a non-linear output [6].

The sigmoid function maps the given input between the range of 0 to 1.

The formula for the sigmoid function is:

So the forget gate the function takes which the output from previous steps given by the cell state and which is input to the current state and passes them through the sigmoid function and outputs all the numbers in the cell state between 0 and 1. A 0 represents to completely forget the information and 1 represents to completely remember the information.

So, the equation now becomes for the forget state:

1. In the second step the update gate or the input gate is activated which decides what new information should be updated to the cell state based on the current input of the current node. This is done in two steps. First a sigmoid function decided which values to update and then a tanh activation function creates candidate values that represent new information to be added to the cell state.

The tanh activation is a function which when given a set of inputs outputs them between the range -1 to 1.

So, the input gate or the update gate produces the following equations:

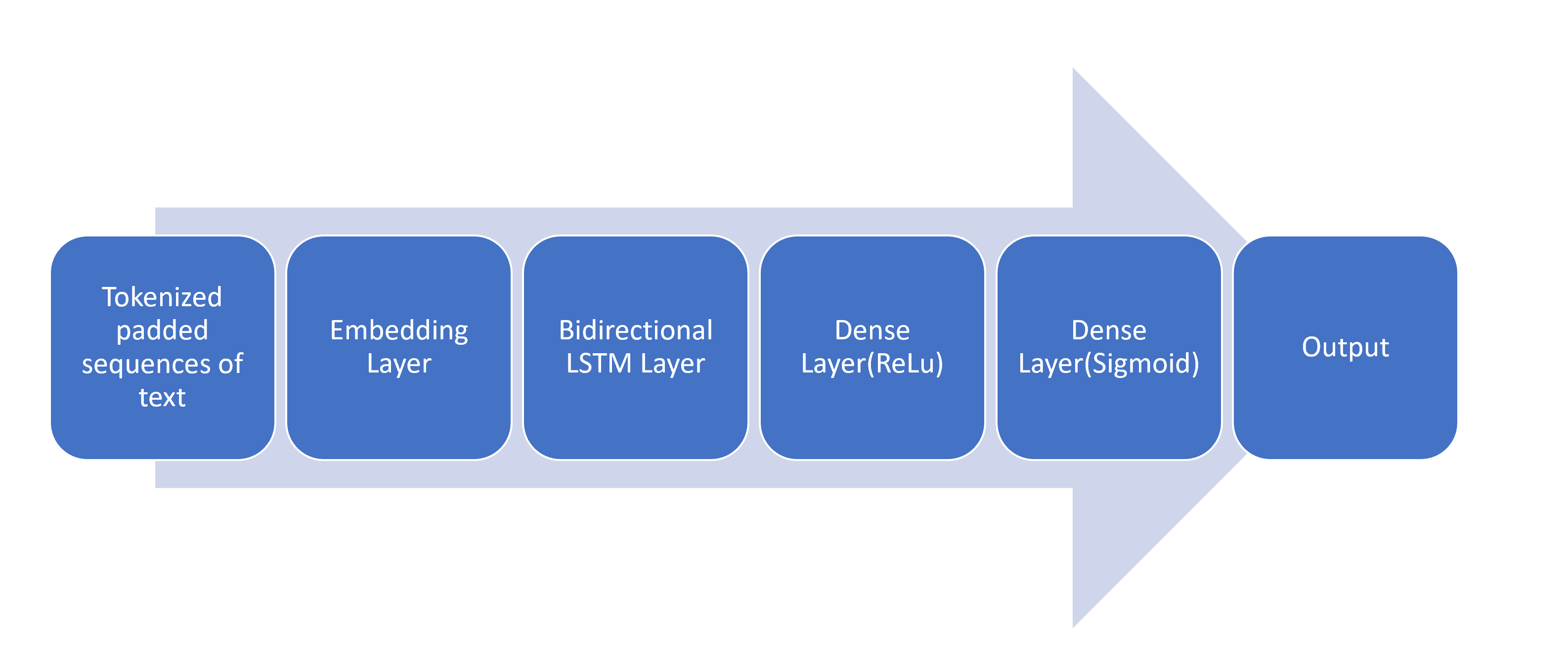
1. The next step is where the new cell state is created. In this stage we multiply the old cell state with so we forget the things we decided to discard. Then add the new update which is the product of and . So now our new cell state becomes:
2. Then the node processes two outputs. One is the updated cell state and other is the output that is part of the cell state which the node decides to pass on as its output.

So, the equations for the output become:

1. The is then pass along to the next node and the next node does the steps 1-4 where the is equal to this it receives.

With this understanding of the LSTM network we then used the Keras deep learning library for our implementation of the binary classification model LTSM of text tweets as disaster(1) or non-disaster(0) .

The following diagram will depict our architecture:



*Fig 6: LSTM Model Process*

Following are the steps that we followed:

1. Based on the number of unique words we had in the training data, we kept the vocabulary size to be 13082. We had the maximum sequence length to be 31 as the sequence length is the length of the sentences in the tweets and maximum of this length came out to be 31 in case for the training dataset.
2. We then split the training data into train and test data for the model to train and test on.

Here we split the data into 80% train data and 20% test data.

1. The text then needs to be given to the embedding layer as sequences of fixed length. This is done by using the Tokenizer class of the Keras which cleans out the text and converts them into sequences. We then pad these sequences to be of length 31. This makes sure that if the sequences are of shorter length additional tokens are added to make them of length 31 and if they are bigger than 31 then they are truncated to the same size.
2. Now the embedding layer takes these sequences and trains on them to represent the text in a vector space. It creates initial weights of each word in the sequences and the similar words are clustered together.
3. Then these sequences along with their initial weights are passed to the bidirectional LSTM layer.
4. Bidirectional LSTM are proved to be significantly more effective than the unidirectional ones [3]. Hence, we decided to have a bidirectional LSTM layer than the unidirectional one. This was achieved by using the keras.layers.Bidirectional function.

A bidirectional LSTM in simple words are two copies of the unidirectional LSTM. The input sequence is given as it is to one layer and its reversed sequence is given to the other layer. Both of these LSTM layers are connected to the same output and this makes sure that for any given point (word in our case) in the sequence the neural network has the contextual information of the points before and after it[3].

So, we created a bidirectional LSTM of 100 neurons. We experimented with the number of neurons like 50, 200 but found that we had better results with the 100 neurons LSTM.

1. Later we also added the dropouts to the LSTM. Dropouts are functions which randomly choose the nodes in the neural nets and drop them out of the network. Which means that the outputs of the randomly selected nodes will not be considered while calculating the neural networks output for a given pass through the network.
2. As tanh functions tend to saturate between the values -1 to 1[7], instead of adding tanh as the activation function we added ReLu as the activation function for the dense layer. For any given input x, ReLu has the following output:

If x>0 🡪 x

If x<0 🡪 0

We experimented with the units of this dense layer to be 24,32,64,100 and equal to the embedding dimension. We got better results with the units of the dense layer equal to 24.

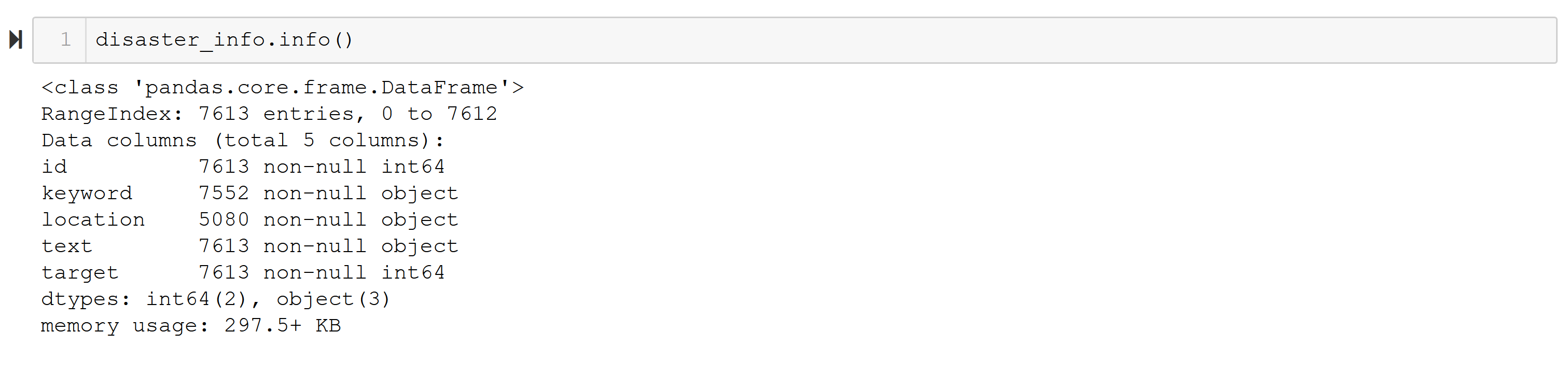
1. We then added the sigmod function dense layer which then maps the output of the sequence from the inputs given by the previous layer’s input.
2. We fit the model on the padded training data acquired in step 2 and pass the test data for validation. While evaluating the model we make use of binary\_crossentropy as the loss function and adam algorithm as the optimizer. We choose binary\_crossentropy as it is a binary classification problem and we used the output layer as the sigmoid dense layer[8]
3. We then make predictions using the model on the unknown test data.

# EXPERIMENTATION AND RESULTS

1. Data Exploration and Preprocessing
2. Data Exploration

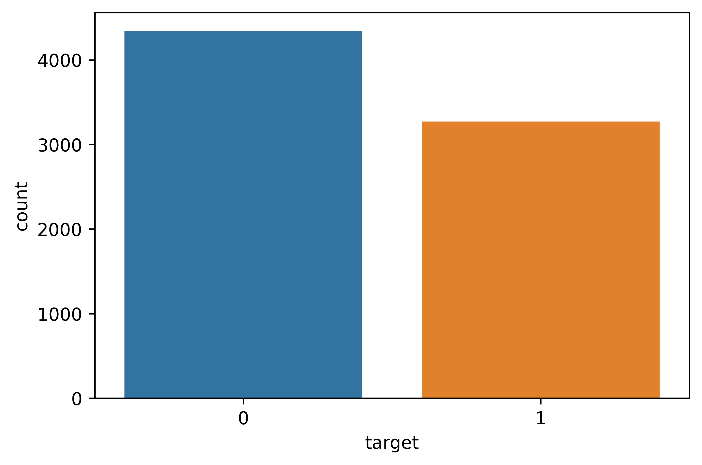
In the Data Exploration phase we read the train\_csv file provided from Kaggle Databases [9].

In the exploration phase we found that there are 4 columns in the data. Following displays the details of the data:

*Fig 7 : Data Information*

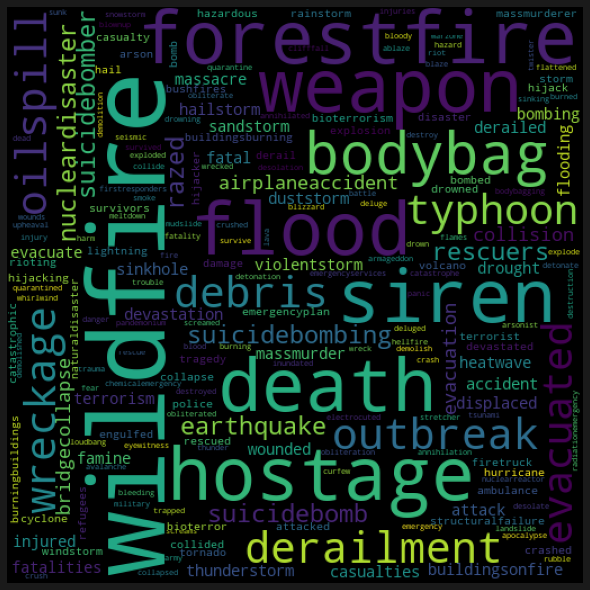
We then explored how is the class distribution in the training samples and we found that there was a certain imbalance between the number of non-disaster tweets (target=0) and disaster tweets (target=1).

Here is the graph of denoting the imbalance:

*Fig 8: Class Imbalance*

Even though there was an imbalance between the data we chose to ignore it for sake of simplicity during the experimentation as the difference was not huge.

After that we explored the disaster related keywords to understand what type of disaster related tweets are present in the training set. Following is the word cloud of the keywords for the tweets which are classified as disaster related:

*Fig 9: Word Cloud of Disaster Keywords*

As we can see the training data has disaster

tweets related with keywords like hostage,

derailment, wreckage, oil spills, earthquakes

etc.

1. Data Preprocessing

Below are the data cleaning steps we have performed:

1. Removing punctuations: characters like !"#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~

2. Removing Usernames:

Twitter usually have usernames associated with @ character. So, we remove the string after @ until space.

3. Removing URLs: Removing words starting with ‘http’, ‘https’, ‘www’ or ending with ‘.com’

4. Removing accented characters as this can lead to wrong concept of words e.g. café

5. Remove extra space: Just removing the extra space from start and end of the sentence. This is to tokenize words correctly

6. Remove numbers: Removing numbers as these are not words and does not make any sense towards understanding the context.

7. Lemmatization: Converting words to their base form e.g. eating, ate, eat are all converted to eat. This is very useful in reducing the unnecessary vocabulary which can sometimes lead to bad vectors.

Then we tokenized the text to list of lists such that each list inside the big outer list are the words in the sentence.

Following is the screenshot of the few of the tokens in the list:

# *Figure 10: Sentence Tokens*

1. Self-Trained Word Embeddings and Classification Results

In this section we present the experimentation results of the self-trained word embedding over the training data and their classification results.

Following steps were followed for the same:

1. Pre-process and tokenize the data.
2. Train the word embeddings over the tokenized data and form the word embeddings. We use genism library functions for both Word2Vec and Doc2Vec word embedding creation.
3. For Word2Vec we have kept the following parameters while training:

size=300, window=5, min\_count=1, workers=5, sg=1. (For skip-gram sg=1a and for CBOW sg=0)

We create models for skip-gram and CBOW model with the above configuration.

1. Then we pass the trained models and tokenized sequences and generate the average weights of sequences by averaging the L2 normalized weights of each word in the sequence.
2. We then run the following set of machine learning classifiers and pass the above generated vectors as input to it and calculate the accuracy of each model. We use the sklearn classification library functions.
3. Following table shows the accuracy for the Word2Vec model with skip-gram model with different classifiers:

|  |  |  |
| --- | --- | --- |
| Sr No | Classification Model | Accuracy(%) |
| 1 | Logistic Regression | 73.86 |
| 2 | Random Forest | 72.76 |
| 3 | Linear SVM | 71.01 |
| 4 | SVM | 57.70 |
| 5 | KNN | 70.27 |

*Table 1: Word2Vec SkipGram Model Approach Accuracy*

1. Following table shows the accuracy for the Word2Vec model with CBOW model with different classifiers:

|  |  |  |
| --- | --- | --- |
| Sr No | Classification Model | Accuracy(%) |
| 1 | Logistic Regression | 65.80 |
| 2 | Random Forest | 66.76 |
| 3 | Linear SVM | 57.70 |
| 4 | SVM | 57.70 |
| 5 | KNN | 65.93 |

*Table 2: Word2Vec CBOW Model Approach Accuracy*

1. Now for the Doc2Vec implementation we first tag the sequences with tags ‘Train’ and ‘Test’. After that we combine both the test and train data as all\_data. Then we use the genism Doc2Vec library function to train create our Doc2Vec model and pass all\_data to it for training. Following are the parameters that we have set for training the Doc2Vec model:

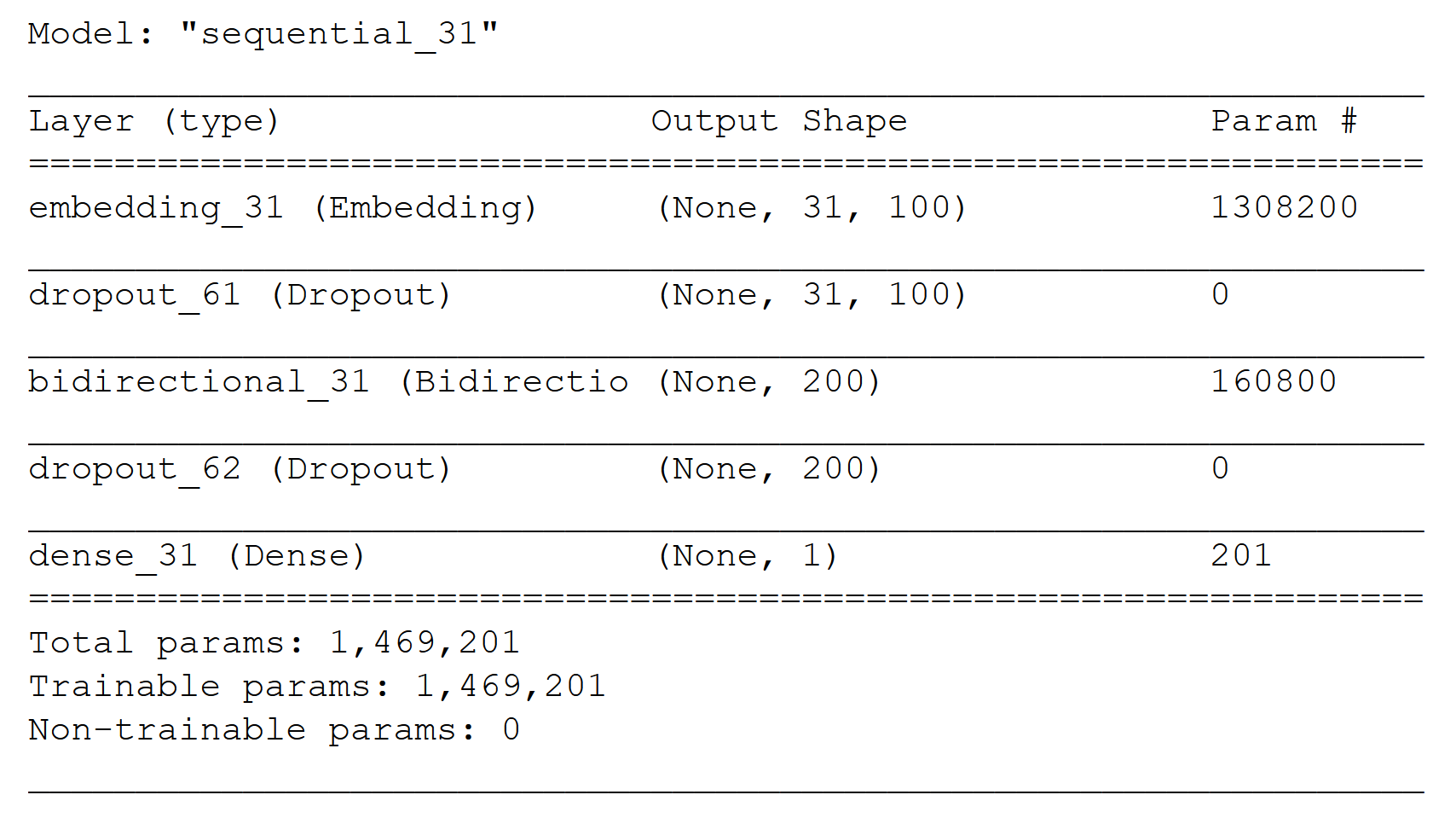
dm=0, vector\_size=300, negative=5, min\_count=1, alpha=0.065, min\_alpha=0.065.

The following table shows the accuracy of the Doc2Vec model with the different classifiers:

|  |  |  |
| --- | --- | --- |
| Sr No | Classification Model | Accuracy(%) |
| 1 | Logistic Regression | 76.57 |
| 2 | Random Forest | 70.79 |
| 3 | Linear SVM | 75.61 |
| 4 | SVM | 72.85 |
| 5 | KNN | 69.70 |

*Table 3: Doc2Vec Model Approach Accuracy*

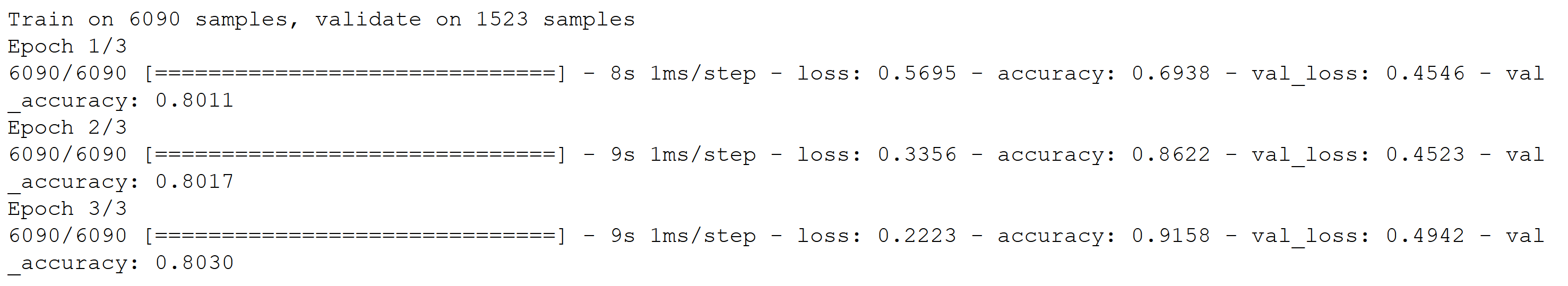
1. Now we built the LSTM model and here is the screenshot of the LSTM model built:



*Fig 11: LSTM Model Summary*

Now while compile we have set the loss function as binary\_crossentropy and the optimizer as adam. We train the model over the fixed length sequences of the train data that we get from the Tokenizer. We run the model for 3 epochs and with the batch\_size = 64.

Here is the screen shot of the running of the model:



*Fig 12: LSTM Execution*

Here as we can see we get accuracy of **80.30%.**

1. Pre-Trained Word Embeddings and Classification Results

In this section we present the experimentation results of the pretrained word embeddings and their accuracy achieved after using them with the different classifiers.

1. Google Word2Vec

This model contains 300-dimensional vectors for 3 million words and phrases.The phrases were obtained using a simple data-driven approach using n-grams. The model can be downloaded from the google archive [10].We follow the following steps for classification using pre-trained word embeddings:

1. Clean and process the training data and form sequences of tokens using preprocessing techniques.
2. Pass the these sequences and the pre-trained word embedding models to the averaging function. The averaging function uses the word embedding weights of each word in the sequence and then averages it to return the average weight of the sequence.
3. Then we pass these into the classification models and train the data. The following table summarizes the experimentation results by using pre-trained Google Word2Vec word embeddings and different classifier models:

|  |  |  |
| --- | --- | --- |
| Sr No | Classification Model | Accuracy(%) |
| 1 | Logistic Regression | 77.71 |
| 2 | Random Forest | 77.01 |
| 3 | Linear SVM | 77.88 |
| 4 | SVM | 77.10 |
| 5 | KNN | 75.35 |

*Table 4: Google Word2Vec Model Approach Accuracy*

1. GloVe Model:

The glove-twitter-25 consists a vocabulary of size 1.2M and 25 dimensional vectors. The GloVe model is different than the Word2Vec model in a sense that it calculates the weights of the words in the sequence based on a global context using the co-occurrence matrix techniques. This forms more relevant relationships between words in the vector space and the model has shown better results compared to Word2Vec.

We follow the same steps that we mentioned in the previous section for Word2Vec and calculate the accuracies of the different classifiers. The pre-trained GloVe model can be downloaded from the Stanford site [11].

Following is summarization of the experimental results by using GloVe pre-trained word embeddings along with different classifiers:

|  |  |  |
| --- | --- | --- |
| Sr No | Classification Model | Accuracy(%) |
| 1 | Logistic Regression | 78.80 |
| 2 | Random Forest | 77.40 |
| 3 | Linear SVM | 78.50 |
| 4 | SVM | 77.01 |
| 5 | KNN | 75.35 |

*Table 5: GloVe Embedding Model Approach Accuracy*

1. Spacy Model

The glove-twitter-25 model consists a vocabulary of size 1.2M and 25 dimensional vectors.

The Spacy Model can be downloaded from the spacy archive [12].

We follow the same steps as we did for the Word2Vec model and calculate the accuracy of the different classifiers used with the Spacy pre-trained word embedding. Following table summarizes our results:

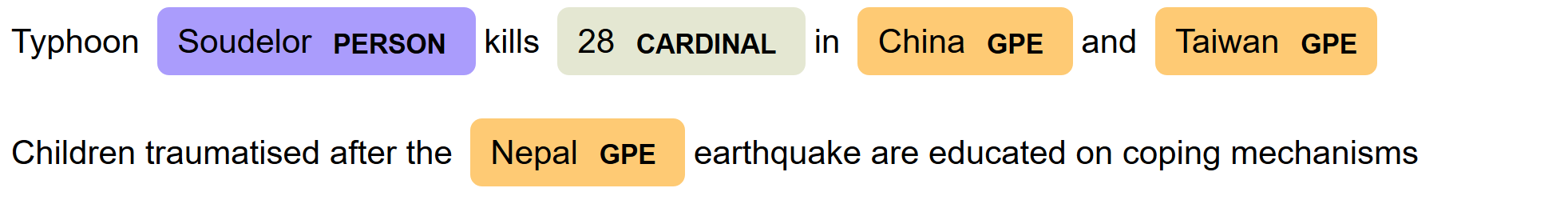
|  |  |  |
| --- | --- | --- |
| Sr No | Classification Model | Accuracy (%) |
| 1 | Logistic Regression | 80.21 |
| 2 | Random Forest | 74.91 |
| 3 | Linear SVM | 81.34 |
| 4 | SVM | 58.53 |
| 5 | KNN | 76.26 |

*Table 6: Spacy Embedding Model Approach Accuracy*

1. Mining Entities from the Tweets

Once we have identified the disaster-related tweets, we will mine knowledge from them. We have used the spacy library to find the relevant entities in the tweets. This is a very good tool where we can find out the different entities like the names, countries, cities, numbers, organizations, etc.

Also, we can display them in a very good fashion by using one more functionality of spacy wiz is displacy as below:



*Fig 13: Entity Mining*

As we can see in the above example using the library we can find the following details from the tweet:

1. The person related to the disaster tweet.
2. Number of causalities that are related to the disaster in the tweet.
3. The disaster location.

All this information can be mined from the other tweets that are disaster related and can be used by the concerned authorities for relevant action.

# CONCLUSION

So, in this report we discuss the various word embedding approaches like word2Vec and Doc2Vec and LSTM. We discuss in detail their working as well as sketch out the different approaches that were undertaken as part of the experiment. From the various experimentation approaches, we found that in the approach for the self-trained word embedding over the training data we got the best accuracy of 80.30% with the LSTM model. This states that the long short term memory networks or LSTM networks which are neural networks perform better in comparison with the other approach of the averaging the word weights in a sequence and providing it to the classifiers. This also reiterates the observation that model considering sequence of the words in a sequence while building vector representation works better than the models which do not consider the sequence of words. The other approach where we used the pre-trained word embeddings performed better with highest accuracy of 81.34%. This observation can be attributed to the fact that these pre-trained word embeddings have been trained over large corpuses making the model more accurate.

We also were able to mine entities from tweets like the person related to the disaster, the place of the disaster etc.

# FUTURE WORK

As part of the experimentation we built basic word embeddings over the training data provided and built basic embeddings from them. We will like to extend the experimentation to try different permutation and combination of the various parameters to train these word embeddings and observe the accuracy of the classifiers using those.

We also will like to extend the study of recurrent neural networks and experiment with different neural networks like Gated Recurring Units (GRU) etc and observe the accuracy of those models.

# REFERENCES

[1] SpaceTomas Mikolov Google Inc., Kai Chen Google Inc., Greg Corrado Google Inc., Jeffrey Dean Google Inc, (2013) Efficient Estimation of Word Representations in Vector <https://arxiv.org/pdf/1301.3781.pdf>

[2] Quoc Le and Tomas Mikolov,(2014) Distributed Representations of Sentences and Documents by <https://arxiv.org/pdf/1405.4053.pdf>

[3] Alex Graves,J¨urgen Schmidhuber, 2005 Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures,

<https://doi.org/10.1016/j.neunet.2005.06.042>

[4] Farzad, Amir, Mashayekhi, Hoda, Hassanpour, Hamid, 2019 A comparative performance analysis of different activation functions in LSTM networks for classification, Neural Computing and Applications, <https://doi.org/10.1007/s00521-017-3210-6>

[5] Christopher Olah, 2015, Understanding LSTM Networks,

[https://colah.github.io/posts/2015-08-Understanding- LSTMs/](https://colah.github.io/posts/2015-08-Understanding-%20%20LSTMs/)

[6] Sagar Sharma, 2017,Activation Functions in Neural Networks, <https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>

[7] Jason Brownlee, 2019, A Gentle Introduction to the Rectified Linear Unit (ReLU), Deep Learning Performance, <https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/>

[8] Jason Brownlee, 2019, Loss and Loss Functions for Training Deep Learning Neural Networks, Deep Learning Performance, <https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks/>

[9] 2020, Real or Not? NLP with Disaster Tweets, <https://www.kaggle.com/c/nlp-getting-started/data>

[10] Google Pre-trained Model, [**GoogleNews-vectors-negative300.bin.gz**](https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=sharing)

[11] Jeffrey Pennington, Richard Socher, Christopher D. Manning, [**https://nlp.stanford.edu/projects/glove/**](https://nlp.stanford.edu/projects/glove/)

[12]Spacy Pre-trained Model, <https://spacy.io/models/en#en_core_web_md>

1. [↑](#footnote-ref-1)