Machine Learning Engineer Nanodegree

Capstone Project

Eva Yang Long
25TH September, 2018

I. Definition

Project Overview

Sentiment analysis of the Natural Language processing (NLP) field has been evolved quickly in the recent years. It helps us to understand the human attitudes and emotions towards a particular topic or product.

The traditional method for sentiment analysis on texts may involve applying word embedding techniques such as bag of words, Vector Space model to extract data and data structure, and then adopting a supervised machine learning models such as Logistics Regression and SVM. More recently, academic researchers have been exploring word embeddings techniques in combination with neural network. For instance, Researchers from Google developed word2vec which inputs corpus of documents and generates a vector space of hundred dimensions. (Mikolov, Tomas; et al., 2013) The new techniques are much faster to train and are able to identify words that have similar meanings and are of the same context. Researchers also have shown that RNN (Recurrent Neural Network) and LSTM (Long Short Term Memory) performs better than other similar Neural network and classic supervised training models, as a recurrent structure learns from semantic of previous texts and can better capture contextual information. (Siwei Lai et al., 2015)

In this project, I explored the use of word vectors and neural networks on predicting sentiments of Twitter Tweets. I tried various scoring based and vector based embedding techniques in combination with traditional as well as neural network based classification techniques.

Problem Statement

The goal of the project is to create a neural network based model that would optimize the performance of the sentiment prediction for tweets. There are two predicted outcomes: Negative and Positive, so it will be a binary classification problem.

My proposed model is Pre-trained GloVe embedding and a LSTM/ neural network classifier, to a traditional method with vector space model and a supervised machine learning model.

The tasks are as follows:

- Data Pre-processing and Exploration: Import data, perform data cleansing and check data quality/distribution.
- Feature Extraction: convert the texts to vectors and in the correct shape to be consumed by the models. Split the data set into train, validation and test sets.
- Train and Validate on the data with the chosen models.
- Predict data on the test sets and compare using the chosen performance metrics: Accuracy and F1 score.

The options for word embeddings and neural networks are:

Word level and Document level vectorisation: Pre-trained GloVe embeddings, Doc2vec

Neural Network: LSTM and simple layered NN

Metrics

Accuracy

$$\frac{(True\ Positive + True\ Negative)}{(True\ Positive + False\ Positive + True\ Negative + False\ Negative)}$$

Accuracy is a ratio of correctly predicted observation to the total observations. It is the most intuitive measure of model performance, and also very straightforward for a binary classification. However, it could be averaging out the class outcomes where there may be large difference between False Positive and False Negative data. We might want to detect negativity in tweets and monitor users with lots of negative posts, so we are interested more in False Positive than False Negative, but accuracy will not be able to tell us enough information on that.

Hence I also used F1 to measure the model performance. It takes account into False Positives and False Negatives.

F1

F1 Score= $\frac{2*(Precision*Recall)}{(Precision+Recall)}$

Where

Precision = $\frac{True\ Positive}{(True\ Positive+False\ Positive)}$ Recall = $\frac{True\ Positive}{(True\ Positive+False\ Negative)}$

Train/Validation Loss for Neural Network

This is the error between computed outputs and desired target outputs produced by the loss function, it is a way to detect overfitting and underfitting.

1. Underfitting – Validation and training error high
2. Overfitting – Validation error is high, training error low
3. Good fit – Converging errors, both low

II. Analysis

Data Exploration and Visualization

Dataset used in the analysis is the Sentiment 140 Twitter Data. It is obtained from Open source via the following website. http://help.sentiment140.com/for-students

The dataset contains 1.6 million twitter tweets collected in 2009. The data has six fields:

0 - the polarity of the tweet (0 = negative, 4 = positive)
1 - the id of the tweet (2087)
2 - the date of the tweet (Sat May 16 23:58:44 UTC 2009)
3 - the query (lyx). If there is no query, then this value is NO_QUERY.
4 - the user that tweeted (robotickilldozr)
5 - the text of the tweet (Lyx is cool)

According to the data creators, the training data was automatically created, as opposed to having humans manual annotate tweets. They have assume that any tweet with positive emoticons, like :), were positive, and tweets with negative emoticons, like :(, were negative.1 There are two caveats of this approach.

1) Neutral sentiments are not considered at all. Though this has become a more simple problem, this approach may not able to reflect neutral sentiments and polarize them to 'positive' or 'negative'.
2) Some users may use the emoticons that reflects the opposite of their sentiments as an act of sarcasm. However, I believe that this should not be a common practice and therefore comfortable with their straightforward definition. For any future analysis on Twitter data, it would be interesting to explore this area, e.g. sarcasm detection.

The data is shown as in figure 1 after converting to pandas dataframe.

	tag	id	date	query	user	text
0	0	1467810369	Mon Apr 06 22:19:45 PDT 2009	NO_QUERY	_TheSpecialOne_	@switchfoot http://twitpic.com/2y1zl - Awww, that's a bummer. You shoulda got David Carr of Third Day to do it. ;D
1	0	1467810672	Mon Apr 06 22:19:49 PDT 2009	NO_QUERY	scotthamilton	is upset that he can't update his Facebook by texting it... and might cry as a result School today also. Blah!
2	0	1467810917	Mon Apr 06 22:19:53 PDT 2009	NO_QUERY	mattycus	@Kenichan I dived many times for the ball. Managed to save 50% The rest go out of bounds
3	0	1467811184	Mon Apr 06 22:19:57 PDT 2009	NO_QUERY	ElleCTF	my whole body feels itchy and like its on fire
4	0	1467811193	Mon Apr 06 22:19:57 PDT 2009	NO_QUERY	Karoli	@nationwideclass no, it's not behaving at all. i'm mad. why am i here? because I can't see you all over there.

Figure 1

The dataset is quite balanced with 800 thousands 'Negative' and 'Positive' labels.

As we see from the sample data there are potentially quite a lot of information that can be redundant, such as the @users, weblink, and punctuations. We may, however, want to keep the punctuations that form expressions, such as the ';D'. Data cleansing is a key step in this analysis and I will discuss in details in section III. Methodology – Data Processing.

[1] The detailed description is in the paper
https://cs.stanford.edu/people/alecmgo/papers/TwitterDistantSupervision09.pdf

Checking the character count helps to understand the anomalies in data. Twitter only allows maximum 140 characters in tweet, but as figure 2 below shows, there are a lot of outliers that the characters far exceeded 140. After investigation, I found that non-English characters failed to encode and it was shown in forms like \x80\x89. They were removed in the data cleansing step.

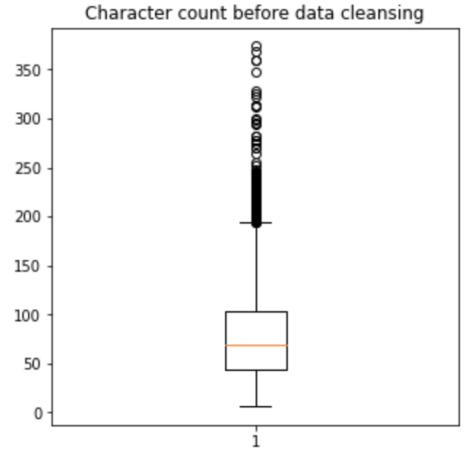


Figure 2

Word count distribution

For the embedding layer of the neural network, the input data should be padded and so they have the same length that made of sequences of integers. Since our tweets are of different length, I plotted the word count distribution of the tweets in order to select the sequence length. Figure 3 shows that most 75% tweets are between 8 to 20 words long, and there are quite a few outliers even after the above steps of data cleansing. (More than 35 words) Choosing big sequence length may not be helpful to the model as we will be padding zeros to most of the tweets, though we can mask them, it could take a lot of time to train. I selected 40 as my maximum sequence length after a few experiments.

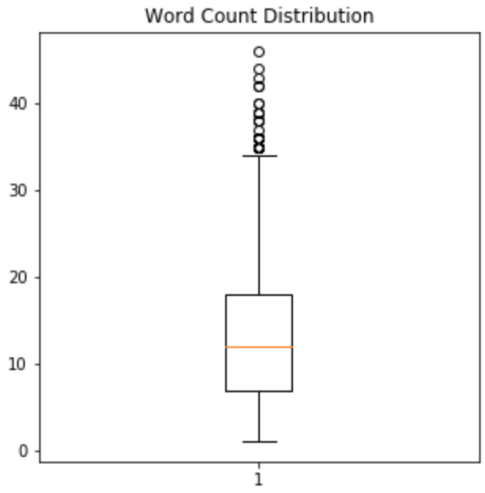


Figure 3

Algorithms and Techniques

I went for two types of word vectorisation techniques: pre-trained GloVe Twitter 200-dimension and genism Doc2Vec to train my own corpus. Similar to Word2Vec, Glove is an unsupervised models for generating word vectors, but it learns by constructing a co-occurrence matrix that counts how frequently a word appears in a context, while Word2Vec predicts context given word. Doc2Vec uses the same technique as Word2Vec but applies to the entire document. The experiment was to compare the performance between the two. As I discussed in previous section, Twitter data contains quite special use of language, and so I decided to try training own Doc2Vec instead of using pre-trained embeddings. As for GloVe, it already has twitter data embedding so it's convenient to use.

For the sentiment classification, I used Long-Short-Term Memory Neural Network for the GloVe embedding. LSTM is a recurrent network that learns from semantic of previous texts and can better capture contextual information. It stores memory of past learnt information. A Recurrent network could experience exploding or vanishing gradients during updates of the model weights as the information keeps feeding into the loop, and LSTM addresses this issue by controlling memory process using a tanh and sigmoid layer.2

I also used a simple neural network structure for Doc2Vec embedding, as Doc2Vec is sentence level vector and it should hopefully take account into some level of contextual information during the training.

The structure for LSTM network is as follows:

- Embedding layer: The input is tokenized data with same sequence length, padded
- LSTM network: Input is 3D sensor of shape (samples, sequence_length, embedding_dim).
- Batch Normalization: Normalize gradients and accelerate learning process
- Dense: Connect and activate the network, 2 dimensional output

For simple neural network the structure will be

- Activation, Batch Normalization, Dropout, we may add more layers.

Also we need to add a compilation argument to configure the learning process. It receives three arguments: Optimizer, Loss function and metrics.

The following parameters can be tuned to optimise the classifier:

- Input Parameters: Sequence Length, Vector Dimension, Embedding Weights (Depend on the word embedding model)
- Model Parameters: Output dimension, Dropout Rate, zero-value masking, activation function and loss function
- Training Parameters: Training length (number of epochs), Batch Size, Learning Rate

[2] Details in the Paper Long Short-Term Memory by Hochreiter, Schmidhuber (1997)

Benchmark

The benchmark model consists of a bag-of-words word vectorization model and a logistic regression as classifier. Pipeline as follows:

Feature extraction - Feature selection - Classification

I used sklearn library to build the pipeline, tuned the parameters and got the model output.

For feature extraction, I used Term Frequency – Inverse Document frequency to transform texts to numeric data.

$$TF(t,d) = \frac{\text{number of times term appears in document(d)}}{\text{total number of terms in document(d)}}$$

$$IDF(t,D) = \log\frac{\text{total number of documents(D)}}{\text{number of documents with the term t in it}}$$

$$TFIDF(t,d,D) = TF(t,d) * IDF(t,D)$$

This is a common technique to infer how important a word or phrases (a term) is to a document in the corpus, and it would help indicate which term is important to determine the sentiment. I used trigram for the 'term', similar analyses on twitter data suggest that using phrases boost the model performance. [3]

The classifier is a Logistics Regression model which is also commonly used for text classification. Model parameters and feature selection method were selected using sklearn's GridSearch package.

The best parameters are shown below in figure 4, the feature selection technique chosen is Truncated SVD with 200 features. The best validation accuracy score is 0.74 and the F1 score for test data is 0.74.

```
print('Optimal alpha parameter: {:}'.format(grid.best_params_))
print('Best score (on validation data): {:0.2f}'.format(grid.best_score_))
```

```
Optimal alpha parameter: {'reduce_dim': TruncatedSVD(algorithm='randomized', n_components=200, n_iter=5,
       random_state=None, tol=0.0), 'reduce_dim__n_components': 200}
Best score (on validation data): 0.74
```

```
y_pred=grid.predict(x_test_tfidf)
print(classification_report(y_true=y_test, y_pred=y_pred))
```

```
             precision    recall  f1-score   support

          0       0.74      0.72      0.73     79473
          4       0.73      0.75      0.74     80167

avg / total       0.74      0.74      0.74    159640
```

Figure 4

[3] The detailed comparison of n-gram vectorization is described in this article https://towardsdatascience.com/another-twitter-sentiment-analysis-with-python-part-5-50b4e87d9bdd

III. Methodology

Data Pre-processing

One important step in text analysis is to clean up the words and extract useful information for the model to train. In addition, Twitter data may contain very special vocabularies and use of words that are different to the normal style of written language. It is more casual and so it contains lots of slangs or use of punctuations as expressions. After sampling some tweets, the following steps are taken to get the cleaned corpus.

1. Remove Hyperlinks, unrecognised encodings
2. Remove '...' and '..'
3. Remove hashtag signs
4. Remove UTF-8 BOM (Byte Order Mark) – This is to remove foreign language words and unrecognised symbols
5. Tokenize, strip html encoding, Remove @user using Nltk's TweetTokenizer function
6. Remove punctuations (but not smiley faces etc.)
7. Remove characters that are failed to encode: [\x80-\xFF] these are non-English characters

After cleansing, I plotted the distribution of tweet length again, which looked much better.

Plot distribution of words

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(5, 5))
plt.boxplot(df_train_clean.len)
plt.show()
```

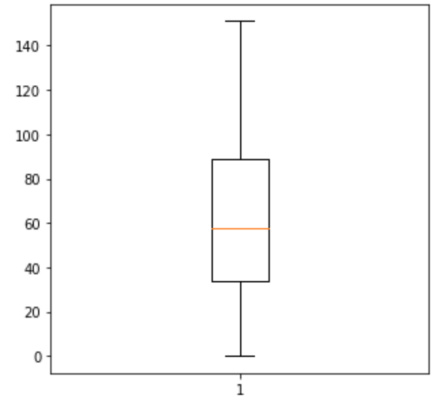


Figure 5

The final step is to remove any empty tweets as a result of the data cleansing, so

8. Remove Empty Tweets

After the last step of data cleansing, I removed 3600 empty tweets and there are **1596400** trainable tweets.

Implementation

I used Keras library for word-level vectorisation and neural network training exercise, Genism library for training Doc2Vec embedding.

The first step of the implementation is vectorisation.

For word-level vectorisation:

1. Data was tokenised using Kera's Tokenizer package. This creates a word library for the corpus and index each unique word.
2. Reconstruct the sentence using tokenised words, the output is a sequence of numbers, and pad the sentence with equal length of 40 tokens. (See example output below in figure 6) 282800 unique tokens were found.
3. The data is and split into train, validation and test. (80%, 10%, 10%).
4. Load pre-trained GloVe 200 dimension embeddings. About 44% of the tokens were being vectorised, the number seemed slightly low but in the training later it actually produced decent result.
5. Get the embedding vectors for each token in the corpus and form an embedding matrix to be fed into the embedding layer of the neural network.

```
padded_data[:10]
```

```
array([[   0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,   56,    1, 1098,  673,  156,  116,
           8,   25,   52,  302, 1099,  163,  136, 1100,   78,    7,   39,
        1101,   92,  164,  130,    2,  226,  674],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,  137,    7,   61,  505,   32],
```

Figure 6

For Sentence-level vectorisation:

1. Build corpus. Use genism library doc2vec.TaggedDocument function to split the words in tweets and label each tweet. I used the whole dataset because it is a completely unsupervised training.
2. Build vocabulary with the corpus and Train doc2vec model. Model parameters are shown in the code snippet below. DBOW (Distributed Bag of words) was used training algorithm to produce the sentence level vector. This is the Doc2Vec model analogous to Skip-gram model in Word2Vec. The vector size is 200, negative sampling is used to prevent overfitting of the data. [4] The vocabulary size considered useful after the training is reduced to 104193.

```
embedding_model1=Doc2Vec(dm=0, vector_size=200, negative=5, min_count=2, epochs=20,workers=cores)
embedding_model1.build_vocab(corpus)
embedding_model1.train(corpus,total_examples=embedding_model1.corpus_count,epochs=embedding_model1.epochs)
```

Figure 7

[4] From genism doc2vec documentation: The integer for negative specifies how many "noise words" should be drawn (usually between 5-20)

3. Get the sentence-level vector from trained model. Split the data into train, validation and test set.

I picked a few random tweets and check the most similar and least similar tweets, and in figure 7 it shows one of the results which look satisfying.

```
import random

doc_id = np.random.randint(embedding_model1.docvecs.count)  # pick random doc, re-run cell for more examples
sims = embedding_model1.docvecs.most_similar(doc_id, topn=embedding_model1.docvecs.count)  # get *all* similar document
print(u'TARGET (%d): «%s»\n' % (doc_id, ''.join(all_x[doc_id])))
```

```
TARGET (1260713): «awww that's a cute pic»
```

```
print(u'SIMILAR/DISSIMILAR DOCS PER MODEL %s:\n' % embedding_model1)
for label, index in [('MOST', 0), ('MEDIAN', len(sims)//2), ('LEAST', len(sims) - 1)]:
    print(u'%s %s: «%s»\n' % (label, sims[index], ''.join(all_x[sims[index][0]])))
```

```
SIMILAR/DISSIMILAR DOCS PER MODEL Doc2Vec(dbow,d200,n5,mc2,s0.001,t4):

MOST (909279, 0.9406556487083435): «awww that's cute»

MEDIAN (1083885, 0.38701751828193665): «is above and beyond life condition strengthened»

LEAST (1059007, -0.23264192044734955): «7»
```

Figure 8

Note that the labels were consists of an integer 0 or 4, and I had to transform them to Keras readable format which is an array of 2, labelled 0 or 1.
The second step is to have a simple check on the performance of the two vectorisation techniques. I tried with an untuned Logistics regression model and the result shows the pre-trained GloVe actually achieved slightly better result.

Pre-Trained Glove

	precision	recall	f1-score	support
0	0.76	0.76	0.76	80040
1	0.76	0.76	0.76	79600
avg / total	0.76	0.76	0.76	159640

Figure 9

Trained Doc2Vec

```
0.745107742420446
```

	precision	recall	f1-score	support
0	0.74	0.75	0.74	78472
4	0.75	0.74	0.75	81168
avg / total	0.75	0.75	0.75	159640

Figure 10

The third step is to feed the data to our neural networks.

Keras is used to implement the model architecture defined in section II.
The graph illustrates the implementation of the LSTM Neural Network model.

Diagram 1: Training process. Loss function is Binary Cross Entropy that produces probability of the two output labels. Adam optimizer is used as it's computationally efficient and require less tuning. (Figure 11)

Figure 11

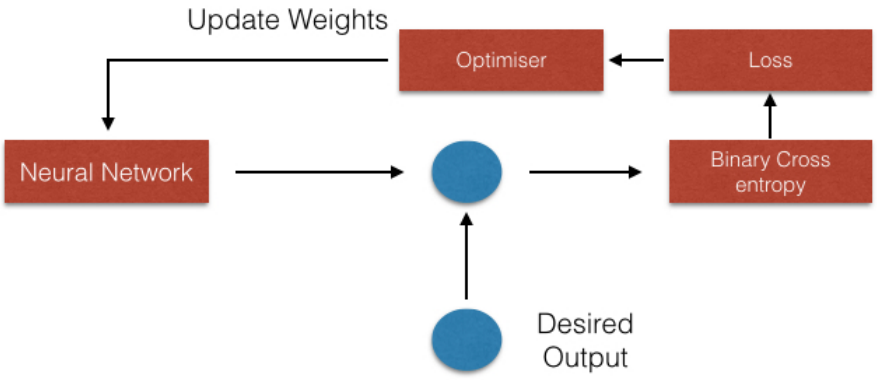


Diagram 2: Detailed architecture of LSTM network. (Figure 12)[5]

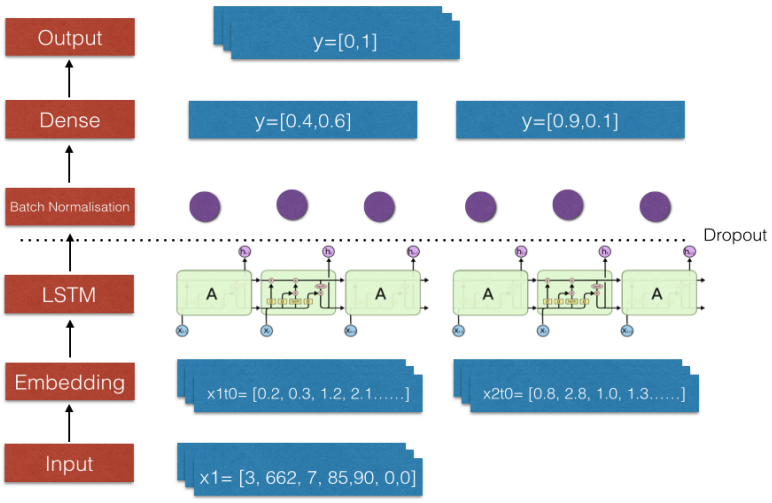


Figure 12

Diagram 3: Detailed architecture of 2-layer neural network. (Figure 13)

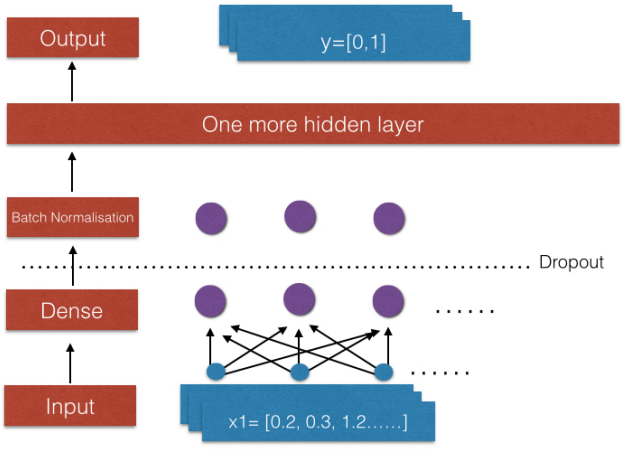


Figure 13

[5] The LSTM layer illustration is taken from source: http://adventuresinmachinelearning.com/keras-lstm-tutorial/

I fitted the train data to the model for 20 epochs, provided validation data for checking overfitting/underfitting of the data. I used Kera's callbacks.ModelCheckpoint to monitor the validation loss and set up early stopping to prevent overfitting. A patience level is for the number of epochs with no improvement after the which training will be stopped. Accuracy is computed after finishing one epoch of learning.

Model history is saved to plot the training/validation loss for evaluation.

Refinement

The initial LSTM model trained very slowly and had tendency to overfit by looking at the train/validation loss graph: the validation loss surged after epoch 3 and the training stopped at episode 4. The prediction has F1 score around 0.79 but the precision and recall for the two labels show quite different performances so it's not balanced. Several changes had been made to refine the model:

Learning Rate/Batch Size: Increased the learning rate from 0.001 to 0.01 and batch size from 32 to 64 as larger batch size should give more confidence in gradient descend and we can try larger learning rate.

Paddings: Masked the zero paddings

The tuned model showed improvement in the precision and recall balance though F1 score stayed around 0.79. In addition, the train and validation loss shows convergence instead of divergence. (In Figure 14, Green lines are tuned result, yellow was initial result)

I tried averaging the GloVe word vectors to sentence-level vector and fed into the LSTM network, and it was also performing well and trained relatively quickly. (red plot in Figure 14 below)

As for the Doc2vec model, initially I tried using a convolutional network but the validation loss showed volatility, and it was predicting a significantly more number of 'Negative' output over 'Positive', even though The F1 score is 0.77. It was quite meaningless as the input is sentence-level vectors and there was not much useful information when the filter convolving over one set of data. (The input shape would be (data length, 1, dimension)).

I switched to simple network, added 1 more hidden layer and batch normalization layer, and the result showed balanced output with F1 score equal to 0.78.

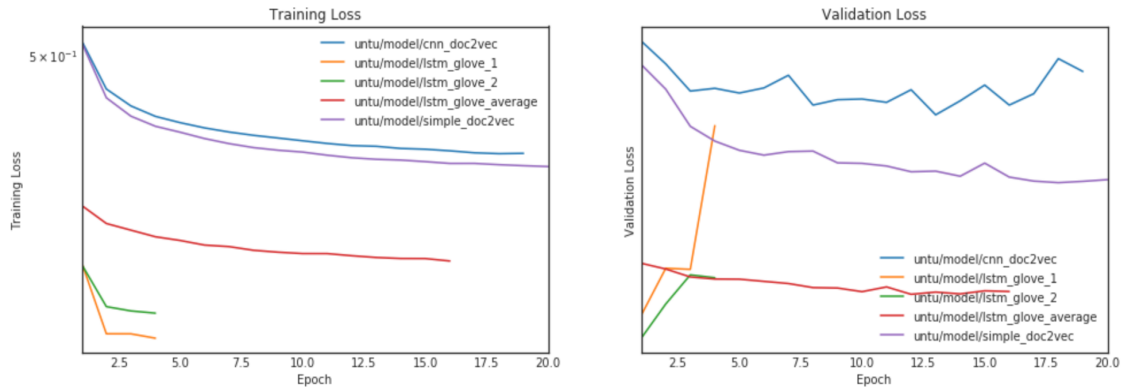


Figure 14

IV. Results

Model Evaluation and Validation

For each model, training data was shuffled and validated using the validation data and predicted on unseen test dataset. The final model is derived by tuning and considering a bunch of metrics that indicate the model performance: Precision/Recall/F1 as well as the training/validation loss values.

The model chosen is the LSTM with pre-trained GloVe – showing low training/validation loss and best F1 score.

The chosen parameters and final architecture is shown in the code snippets in Figure 15 and 16.

```
embed_dim=200
#max_feature=len(embedding_model1.wv.vocab)+1
max_feature=len(word_index)+1
lstm_units=128

def lstm(max_features, lstm_units, output_dim=2):
    model = Sequential()
    model.add(Embedding(max_features, embed_dim, weights=[embedding_matrix], mask_zero=True))
    model.add(LSTM(lstm_units, input_shape=(None,200), return_sequences=False, dropout=0.2))
    model.add(BatchNormalization(axis=-1,
                                 momentum=0.99,
                                 epsilon=0.001,
                                 center=True,
                                 scale=True,
                                 beta_initializer='zeros',
                                 gamma_initializer='ones',
                                 moving_mean_initializer='zeros',
                                 moving_variance_initializer='ones',
                                 beta_regularizer=None,
                                 gamma_regularizer=None,
                                 beta_constraint=None,
                                 gamma_constraint=None))
    model.add(Dense(output_dim, activation='sigmoid'))
    optimizer=Adam(lr=0.01)
    model.compile(loss="binary_crossentropy", optimizer=optimizer, metrics=['accuracy'])
    print(model.summary())
    return model
```

Figure 15

```
batch_size = 64
learning_rate=.01

fname = '/home/ubuntu/model/keras-lstm-word_2.h5'
cbks = [callbacks.ModelCheckpoint(filepath=fname, monitor='val_loss', save_best_only=True),
    callbacks.EarlyStopping(monitor='val_loss', patience=2)]

hist = model3.fit(x=train_x, y=train_y,
    batch_size=batch_size, epochs=20,
    verbose=1,
    callbacks=cbks,
#     validation_split=0.8,
    validation_data=(val_x, val_y),
    shuffle=True,
    class_weight=None,
    sample_weight=None)
```

Figure 16

Pre-trained Glove + LSTM result shown below in Figure 17:

```
Epoch 1/20
1277120/1277120 [==============================] - 4728s - loss: 0.4385 - acc: 0.7982 - val_loss: 0.4436 - val_acc:
0.7946
Epoch 3/20
1277120/1277120 [==============================] - 4717s - loss: 0.4375 - acc: 0.7993 - val_loss: 0.4481 - val_acc:
0.7923
Epoch 4/20
1277120/1277120 [==============================] - 4706s - loss: 0.4370 - acc: 0.8001 - val_loss: 0.4477 - val_acc:
0.7934
             precision    recall  f1-score   support

          0       0.81      0.78      0.79     82460
          1       0.77      0.80      0.79     77180

avg / total       0.79      0.79      0.79    159640
```

Figure 17

Doc2Vec + 2 hidden neural network result shown in Figure 18:

```
             precision    recall  f1-score   support

          0       0.80      0.76      0.78     83157
          1       0.76      0.79      0.77     76483

avg / total       0.78      0.78      0.78    159640
```

Figure 18

Justification

Both neural network models have shown better performance to the benchmark model by using the F1 and accuracy metrics, where the LSTM model slightly outperformed the rest. Pre-trained Glove+LSTM is about 5.5% better for both metrics comparing to benchmark, and Doc2Vec+Simple NN is about 4% better.

In terms of Losses, the LSTM model had lower training and validation losses than the other, suggesting good fit of the model.

For the LSTM model the training speed is much slower due to the processing of each word vector of the same dimensionality as sentence vector. The whole training of Trained Doc2vec + Simple neural network took about 45 minutes in total (around 138s per epoch), and each LSTM network took over an hour per epoch (4700s), total 4 epochs. Training Doc2vec took around 30 minutes and still it seemed more efficient than the LSTM model.

Due to the limitation in computing efficiency, I believe it would be more convincing if the LSTM model had shown a more significant improvement in performance, but I think it has the potential – There are still room for model adjustment both for the embedding and the neural network parameters. This is detailed in the Reflection section.

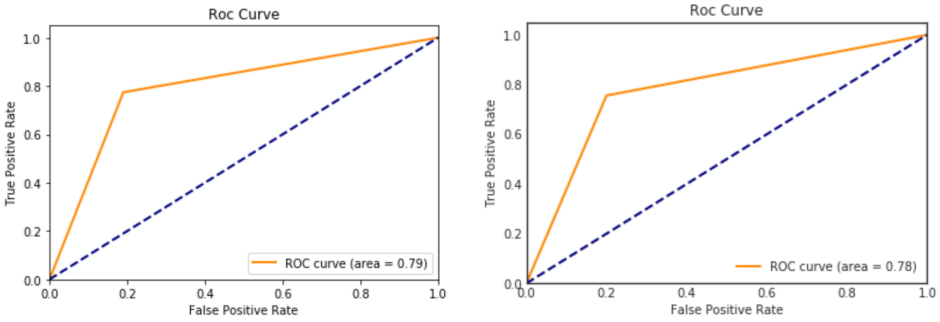
V. Conclusion

Free-Form Visualization

Most of the visualization outputs were being discussed thoroughly in the previous section.

One additional plot to visualise the performance is the ROC curve (Receiver Operating Characteristic) and Area Under the Curve (AUC) that represents the degree of separability. It tells how much model is capable of distinguishing between classes. Higher the AUC (curve further to the diagonal line), better the model is at predicting 0s as 0s and 1s as 1s. It plots False Positive Rate defined as False Positive/(True Negative + False Positive) against True Positive (Recall) as True Positive/(False Negative +True Positive).

In Figure 19 it shows the plot for the two neural network models, as the performances are quite similar so they look almost the same. curve is kinked as I had [1,0] binary classes instead of probability outcome, hence there is a cut-off.



Pre-trained Glove + LSTM Doc2Vec + Simple NN

Figure 19

Reflection

The solution for the problem can be summarised in the following process:

1. Research on sentiment analysis, find data and relevant empirical analyses
2. Define problem and propose solutions and benchmark technique based on the research
3. Set up environment (I used Amazon EC2 Instance for deep learning) and get data
4. Data exploration and preparation
5. Special processing for text document (tokenization and vectorization)
6. Establish, evaluate and refine neural network models, get result
7. Run benchmark analysis and get result
8. Compare results

I think step 5 and 6 are the most challenging part. There are a lot of choices of word embeddings (Word2Vec, Glove, Doc2vec, trained, pre-trained embeddings etc.) and deep learning models with many parameters/layers. I had to do extensive amount of research and experiments to understand the differences and suitability of the choices, eventually reaching a desirable result.

However I feel I have understood much more about natural language processing and neural network in general after doing this project. It also gave me ideas of how this can be applied in projects at work.

Improvement

I believe the following can be implemented as further improvements for this analysis.

- Explore the use of FastText as a pre-trained embedding may help improve the coverage of embedded words. In this model, only half of the tokens were embedded, I believe it is due to Twitter has a casual written style and not every word is matched to the pre-trained vector.
- Another solution would to train individual word level vector using Genism word2vec and feed into the LSTM network
- A more comparable set of models will be Pre-trained word-level embedding VS trained word-level embedding for LSTM network, then the same to a simple neural network, plus a sentence-level embedding for a simple neural network. In this proposed approach, each model will have 1 variable that's changed hence it's more robust.

References:

https://nlp.stanford.edu/projects/glove/
https://towardsdatascience.com/another-twitter-sentiment-analysis-with-python-part-10-neural-network-with-a6441269aa3c
http://adventuresinmachinelearning.com/keras-lstm-tutorial/
https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html
https://cs.stanford.edu/~quocle/paragraph_vector.pdf
https://github.com/RaRe-Technologies/gensim/blob/develop/docs/notebooks/doc2vec-IMDB.ipynb