EE P 596 Mini Project 2 Report

Ryan Maroney
Blake Downey

3/17/2022

Introduction

For this mini project we were tasked with classifying tweets as having positive or negative sentiment. We were required to train at least one non-deep learning baseline model and at least one deep learning model. There is a Kaggle competition associated with this project for us to submit predictions for unlabeled data.

For our non-deep learning models we trained Logistic Regression (LR) and Stochastic Gradient Descent (SGD) models. For our deep learning models we trained using BERT (Bidirectional Encoder Representation from Transformers) and Bi-LSTM models. In this report we will discuss our results and insights from using each of these models.

Data Preprocessing

The data is prepared similarly for both the non-deep learning LR / SGD and the deep learning BERT / Bi-LSTM models. Twitter is a place where people use a lot of slang and not everyone uses the same slang, so this problem is a very complex one. We figured that if the tweets were all in the same sort of baseline vocab set, it would help improve accuracy.

So in an attempt to translate the twitter slang to regular english words, we built a series of translation functions to be our all in one translation system for twitter to english words. The operation to translate tweets is very simple by design, use the apply method of pandas to apply the translate function to the tweets column. This first enters into the translate function which operates tweet by tweet. It removes mentions (@name) and also removes urls from the tweet. It then handles special cases like ‘b/c’ which should be interpreted as because but fails to translate that way after tokenizing due to the split on ‘/’. This special case function is run on every tweet and is very short. The tweet is then tokenized using a Twitter Tokenizer from nltk library. Each token or word in the tweet is then processed through a set of if else statements to translate each word to regular English if possible. The punctuation is removed and words are lowered here as well. After multi word translations are made (translation eg. ‘imy’ -> ‘i miss you’), single word translations are made (translation eg. ‘u’ -> ‘you’), and the tweet is then joined back into a sentence and returned. Note we did not remove stop words as we were able to achieve 2% higher results with the stopwords.

The LR / SGD methods used the translate function and then used a count vectorizer with a 100k max feature count and n_gram range [1,4]. We found that when including more n_grams we were able to see a ~3% improvement in accuracy.

The deep learning Bi-LSTM model also used the translate function, but instead of a count vectorizer we built a token and pad function which uses a keras preprocessing

text Tokenizer. We capped the tokenizer to have 20k words and return a maximum padded sequence of length 50.

The deep learning BERT model also used the custom translate function to begin the preprocessing in an effort to provide text that more closely resembles standard english. Keras provides an additional preprocessing layer that can be downloaded and used to transform the raw text input into a fixed-length encoded sequence for the BERT encoder. This sequence consists of tokenized input words represented with numerical ids.

Non-Deep Learning Models

For the non-deep learning baselines we simply used sklearn's LogisticRegression and SGDClassifier methods. For LR, we used an L2 norm penalty with a damping factor of 0.1. For SGD we also used the L2 norm penalty. For metrics we used Train vs Validation Accuracy and F1-Score.

Bi-LSTM

For the Bidirectional LSTM we played around with the architecture for a while until we eventually found a suitable layer count and feature count that regularized overfitting while giving good validation scores. We ended with a model that used an embedding layer to go from the 20000 word count to the embedded 32-d space, 500 feature output 1D-convolutional layer, a 1D max pool layer, a Bidirectional LSTM layer with 128 as the unit count, a 500-d dense layer, a heavy dropout layer with rate 0.58 and finally a 2-d dense layer with softmax to predict the sentiment.

We tested with a few different loss functions and optimizers, but found that we got the best results when using the binary cross entropy loss coupled with the rmsprop optimizer.

To track the best model we used keras callbacks method ModelCheckpoint. With this we were able to automatically track the validation loss during the epochs of training and only save the models that had lower loss than previously saved models. For metrics we decided to use accuracy, precision and recall originally, but found that the information was redundant as each epoch's results were the same number across the 3 metrics for both train and validation, so the metrics were reduced to one: accuracy.

We then used a batch size of 1024 to help speed up training as the training process can take a very long time even on GPU hardware. When training with a smaller batch size we saw slight improvements in validation accuracy over the epochs but the

improvement was not big enough to justify the increase in training time. We also tried using multiple Bidirectional layers, but found that adding even one layer increased the training time by up to 10 times.

BERT

We looked at a few different articles for inspiration on incorporating BERT into our model (reference 1, reference 2, reference 3). The third reference proved to be the most useful and easy to follow. This model consisted of essentially four high level layers. The first being the Keras hub BERT preprocessing layer to transform the raw text input into tokenized sequences to be trained on. The second layer is the primary training layer which is the BERT encoder layer that was also downloaded from Keras hub. The final two layers were general neural network layers. One being a 10% dropout layer to help prevent overfitting, and the final layer being a single node dense layer with a sigmoid activation function for binary classification. The model was compiled with the adam optimizer and binary cross entropy loss function.

Training with BERT proved to be a challenge due to its size. The BERT model consists of 110 million parameters, and being limited to Google Colab's GPU resources, training even just one epoch could take around 6 hours. Unfortunately when initially beginning this training we ran into Colab's resource limits for GPU and had lost the training that had been done. To resolve this we divided the data into chunks and trained the model on smaller chunks at a time, saving the model between each chunk. This meant that we were able to maintain our training progress, but resulted in taking even more time due to the iterative nature of dividing into chunks and not being able to run it all at one time.

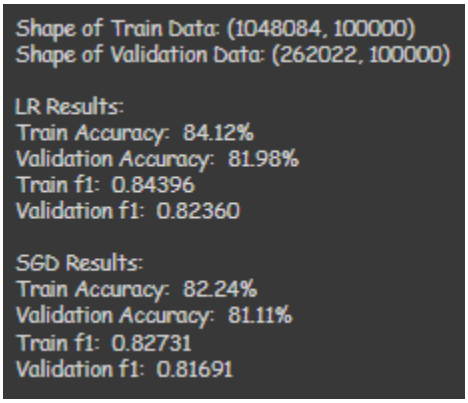
Because of the above stated challenges and limitations we were only able to fully train a single epoch on the full set of training data, and were unable to iterate over different model variations. Given more time we would have attempted training with different variations of preprocessing, and different optimizers, similar to how we did for the other models.

Unfortunately we did not realize this earlier, but we found that there is a smaller version of BERT that is also pretrained. This would have been able to save us some time and allow us to complete more iterations, but we had already devoted more of our time to fine tuning the Bi-LSTM model.

Results

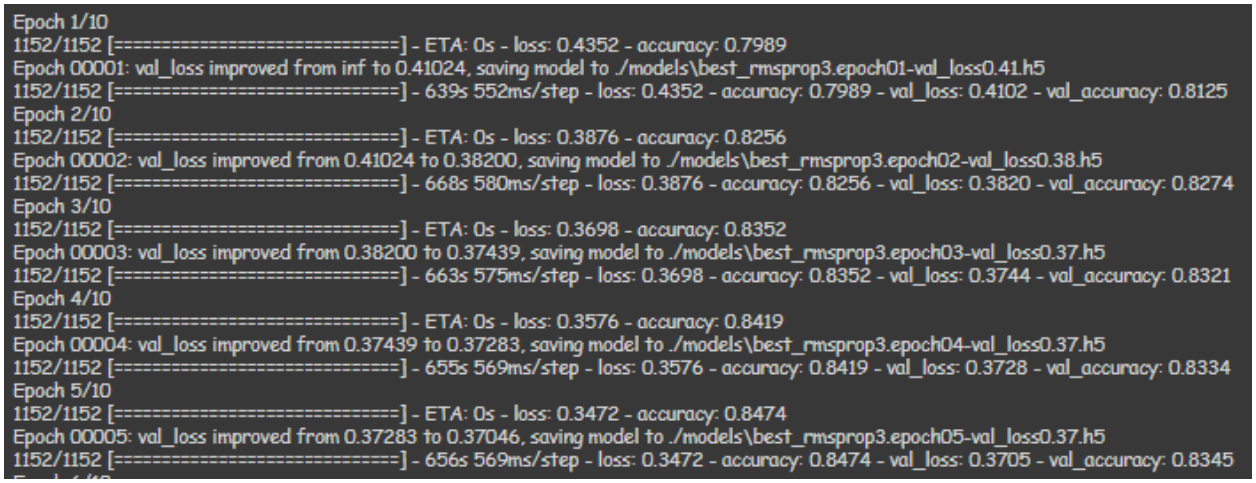
With non deep learning we were able to achieve an f1-score of 0.82360 on the validation data and a kaggle score of 0.80833 on the evaluation set. This was using the LR set up with a screen grab of the results pasted below. These results, we believe, are

a result of the preprocessing that was done to translate the slang to English words, including the removal of punctuations, word lowering, mention and url removal, the keeping of stop words, and the count vectorizer. As shown below the LR and SGD results were close but the LR results always showed slightly higher accuracy and f1 scores than the SGDClassifier.

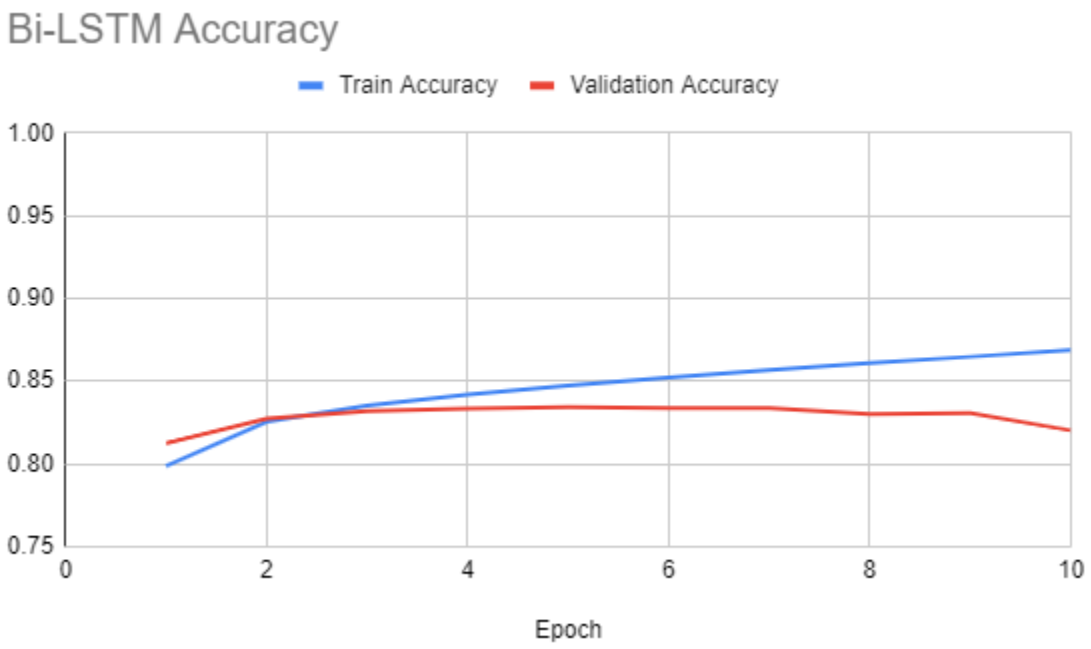


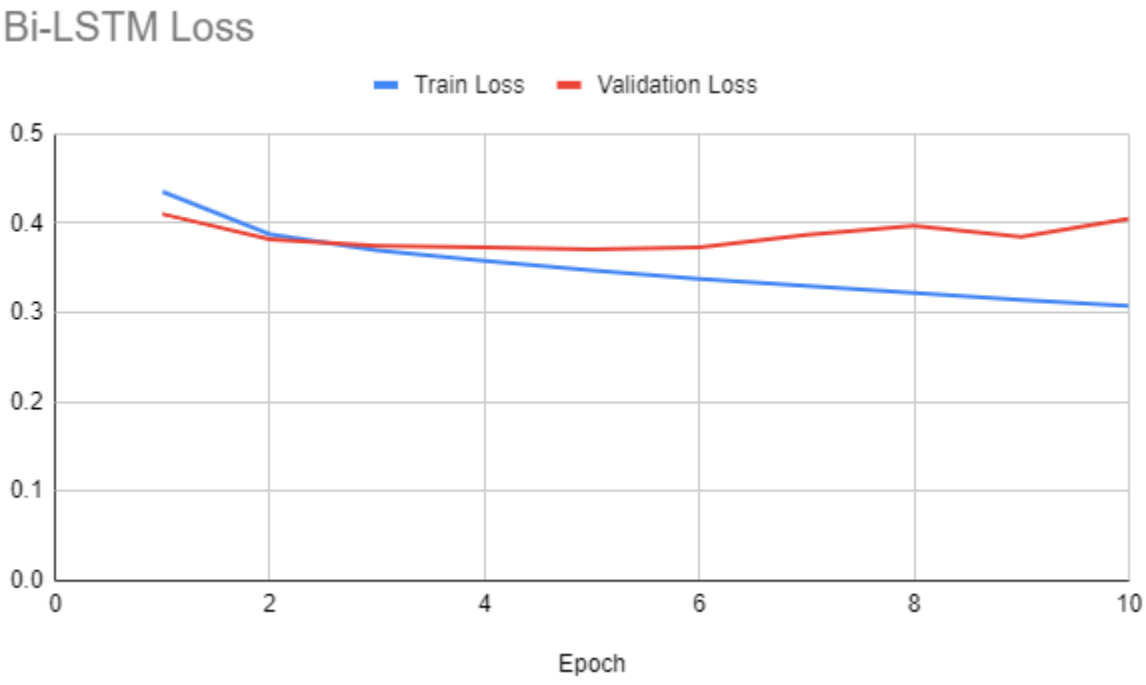
LR and SGD Results on Validation Data

With a deep learning based Bi-LSTM model, we were able to achieve a validation accuracy of 83.45%. We found that when the lstm model achieved higher and higher training accuracy (accuracies 85-95%), we saw a depreciation of accuracy for the validation set. This was a prime example of overfitting and because of this we added the large rate to the dropout layer. Our best trained model occurred on the 5th epoch of this training cycle with a validation accuracy of 83.45% and we decided this was sufficient. We then ran this model on the kaggle evaluation set and achieved a score of 0.83294! The screen grab of the training process is below to show train vs validation accuracy over time.



Bi-LSTM Train and Validation Accuracy





With the BERT based deep learning model we were only able to obtain a 0.73708 F-1 score on the Kaggle test data set after training for 1 epoch. The loss is still above 0.5 after one full epoch though so there appears to still be room for improvement, but as mentioned earlier, this model is very inefficient. From the image below it took 6156 seconds to train on 5% of the training data, and detect the accuracy for all of the

validation data. Making the predictions for the test data can also take upwards of an hour.

```
1638/1638 [==============================] - 6156s 4s/step - loss: 0.5391 - accuracy: 0.7298 - val_loss: 0.5093 - val_accuracy: 0.7560
<keras.callbacks.History at 0x7f4823367110>
```

Insights

An interesting takeaway that we were not anticipating in this homework assignment, was that we were able to achieve only 0.02461 better results on the evaluation kaggle data with our final Bi-LSTM deep network than with our Logistic Regression Baseline attempt.

As mentioned before, one main insight from the BERT model is that the model is very large and takes a long time to train, even if only a small portion of the parameters are marked as trainable. While we were not able to extensively train and iterate over the BERT model, it was interesting that it performed worse than the baseline models. One thing I am curious to understand more is that since the BERT model is primarily trained on wikipedia articles, does this make it more difficult to understand tweets where the language is much less formal and the text is more abrupt.

NLP is a very complicated problem space that has so many different variations of what preprocessing can be, how to build a particular architecture and countless other factors that lead to high variance in results. For example, we found that in contrast to the email classification problem, when we remove stop words for tweets, we actually performed worse than with the stop words. We also experimented with punctuation removal, and translating vs not translating the twitter slang. After some rounds of empirical results we found that training with all the data (minus 10% for validation) was the most optimal and we were able to get our deep learning results higher than our LR results.

As for scalability to a million data points, we first began training with a small amount of the data, simply by subsampling the data into 3rds or 4ths. After we achieved some initial results we removed the subsampling factor and ran the baseline and Bi-LSTM models on the entire ~1.3 million training points. We found that the computational overhead was not significantly different with the LR method and increased the Bi-LSTM training time by roughly 3 times. In terms of explainability to a non technical person, we would opt to use LR since it is a linear model and does not include the complex architecture of neural networks. Like the explainability of LR, we would also opt to use LR as it has a lower computational cost then BERT or the Bi-LSTM if computational cost is a limiting factor..

Contributions

Blake: Translate functions, baseline non deep learning methods and results, bi-lstm architecture and results.

Ryan: BERT preprocessing/training/results. Assisting with fine tuning other models. Analyzing results.