Disaster Tweets NLP

**EDA**

1. Remove ‘id’ and ‘location’ columns because 2533 out of 7613 rows does not have a location.
2. Fill NaN in ‘keyword’ column with ‘unknown’
3. Feature engineering:
   1. Total word count
   2. Unique word count
   3. Character count
   4. Hashtag count
   5. Mentions count
   6. Stopwords count
   7. Punctuation count
4. Append keyword to text column and remove keyword column
5. Expand contractions
6. Use autocorrect.Speller() to spell check.
7. Expand hashtags and usernames, abbreviations, and acronyms
8. Remove URLs
9. Make all text lowercase, remove text in square brackets, remove words containing numbers
10. Remove \n new line characters and similar special characters
11. Remove stopwords
12. Stemming
13. Train-test-split
14. Text was tokenized
15. Meta-features scaled with StandardScaler()

Steps 4 through 12 was added on after a couple of preliminary models that did not perform well, and boosted the F1 score of a simple RNN model from 0.6 to above 0.7.

Train-test-split was done here through sklearn instead of using the validation\_split argument when fitting the Keras model, which causes a bigger overfit.

**Model Development**

Models were generally laid out as:

1. Tokenized text input
2. Embedding
3. RNN layer(s)
4. Concatenate RNN output with meta-features
5. Dense layer(s)
6. Output Dense(1) with sigmoid activation

I use dropout layers to reduce overfitting, which was a significant issue in the first few model iterations. Reducing layer units seemed to reduce this issue, as well as using a learning rate reduction callback. The recurrent\_dropout argument in the LSTM layers also reduced some overfitting, but I excluded it from tuning because it does not allow Tensorflow to use cuDNN, which slows down models significantly. All models were also tuned with Optuna, which has a built-in Keras pruning callback that speeds up the process. Parameters tuned: RNN dropout, spatial dropout, dropouts, learning rate, RMSprop vs Adam optimizer, number of units at each hidden layer, and embedding output dimension.

Models ran:

1. Single SimpleRNN layer
2. 2 SimpleRNN layers
3. LSTM
4. Bidirectional LSTM
5. Dual bidirectional LSTM
6. GloVe embedded bidirectional LSTM
7. GloVe embedded dual bidirectional LSTM
8. BERT

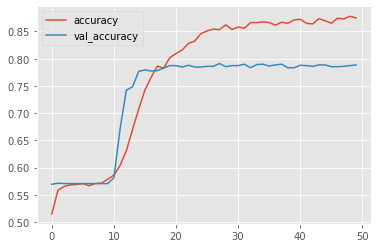
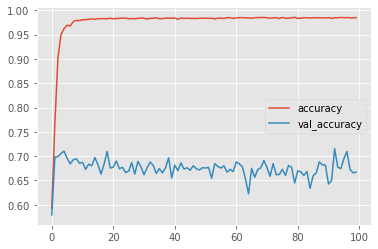
GloVe embeddings: I tested a simple bidirectional LSTM model on both the twitter and Wikipedia-derived sets with the different dimensions. The Wikipedia set of 200d word vectors performed the best.

I also tried out a simple implementation with base BERT without any tuning. The model has way too many parameters, causing an OOM on my local GPU. I used Kaggle’s GPU-accelerated notebook to run 10 epochs.

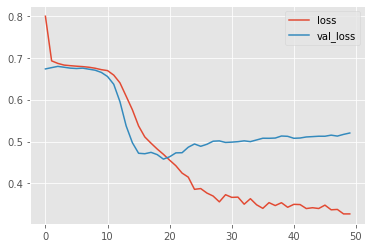
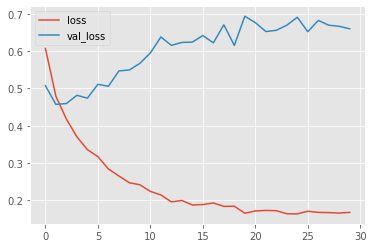
**Model Evaluation**

For each model, I track loss and accuracy per epoch to determine overfit/underfit. I also use Tensorflow Addons to track the F1 score, since Kaggle uses it as the metric. The outputs are probabilities, which I convert to 1/0 for Kaggle scoring and other metrics. I also output the classification report and confusion matrix to observe the results.

Accuracy/epoch plots for an obviously extremely overfit model vs a better model.



Loss plots are more indicative of overfitting:



**Kaggle Results:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Model | Train loss | Val. loss | Train Acc | Val. Acc | Val F1 | Kaggle F1 |
| SimpleRNN(40) | 0.4695 | 0.4580 | 0.8020 | 0.7873 | 0.7750 | 0.78731 |
| 2x SimpleRNN(40) | 0.3821 | 0.4598 | 0.8325 | 0.7886 | 0.7831 | 0.78180 |
| LSTM(46) | 0.4484 | 0.4509 | 0.8199 | 0.7938 | 0.7810 | 0.78302 |
| Bidirectional(LSTM(100)) | 0.4790 | 0.4572 | 0.7860 | 0.7912 | 0.7765 | 0.78792 |
| 2x Bidirectional(LSTM(100)) | 0.4262 | 0.4480 | 0.8144 | 0.7945 | 0.7750 | 0.78516 |
| GloVe Bidirectional(LSTM(100)) | 0.4664 | 0.4228 | 0.7977 | 0.8116 | 0.7869 | 0.79129 |
| GloVe 2x Bidirectional(LSTM(120)) | 0.3965 | 0.4065 | 0.8279 | 0.8162 | 0.8016 | 0.80294 |
| BERT | 0.3314 | 0.4113 | 0.8824 | 0.8289 | 0.8320 | 0.84155 |

**Insights**

Similar to the previous week’s work, deep pretrained layers defeat even the most tuned basic models. GloVe did not do as well as I thought it would. Originally, its results were much better than the models without pretrained word vectors (0.77 and below), but tuning the output vectors of Keras’ embedding layer worked to close the gap.

EDA was also quite critical. The GloVe models occasionally worked better when certain data processing steps were excluded, because it contained its own vectors for punctuations, for example. The BERT model didn’t require any EDA at all, and adding my meta-features actually worsened the model.

Using 2 RNN layers generally improved the final score, but shows more overfitting as well.

I did not have enough time this week, but I would have like to try a GRU-based model as well.