Felix Chen, felixchenyijun@gmail.com

For my model, I decided to do a classification model because I think that positive, negative, and neutral tweets should be fundamentally different. My hypothesis is that there's not a spectrum of how some tweets can be more positive or negative than other tweets. Positive and negative tweets are fundamentally different. Some vocabularies found in positive tweets can't be found in negative tweets, and some negative tweets can't be found in positive tweets.

For preprocessing, I did word vectorization with GloVe. GloVe has a huge library of word to vector conversion generated by going through tweets online. Turning words into vectors is more effective than one hot encoding in this scenario because the value of vectors of each word can show relationships between the words. For example, the vector of "king" is roughly equal to the vector of "queen" - "female" + "male". The GloVe vectors have 200 indexes each. My implementation of this word vectorization is a bit messy because I experimented with other methods before choosing to go with GloVe word vectorization. I first used Tokenizer from Keras to turn tweets(strings) into a list of indexes(kind of similar concept to one hot encoding) and I saved a dictionary of all words and their corresponding index. Then I padded each list so that they'd all have the same length (118 parameters). In order to change the list of indexes into a list of vectors, I first created a new dictionary that has the key as an index that a word corresponds to, and value as the word. Then I iterated through the entire data structure and made another list of list of vectors by using the GloVe dictionary that has the word as the key and the 200 dimension vector as the value. However, I didn't use the entire dataset with 1.6 million tweets, I only used 1% of the dataset because it took my computer too long to run when I used the entire dataset.

For postprocessing, since we are using classification, I used one hot encoding for the three different results. Negative(valence=0) corresponded to [1, 0, 0]; neutral(valence=2) corresponded to [0, 1, 0]; positive(valence=2) corresponded to [0, 0, 1]. The neural network would also output a 3-dimensional vector, each index corresponding to the possibility that the tweet is negative, neutral, or positive.

Sentiment
[1, 0, 0] = negative; [0, 1, 0] = neutral; [0, 0, 1] = positive

For the model, I used LSTM. I used LSTM because interpreting the sentiment of a tweet requires remembering what the previous words in the tweet were. For the architecture of my model, I had:

LSTM(output = 64, activation = 'relu') -> Dropout(rate = 0.2) -> LSTM(output = 32, activation = 'relu') -> Dropout(rate = 0.2) -> Dense(output = 3, activation = 'tanh')

I chose to have 2 layers of LSTM layers and each with their perspective number of output nodes after trying different combinations and trying to balance between model complexity and overfitting. I added the two dropout layers, each with a rate of 0.2 to avoid overfitting, and I had a final dense layer to condense all the options down into the final three choices, negative,

neutral, or positive. I chose ‘relu’ as the activation for the two LSTM layers to avoid any potential problems with the vanishing gradient that a sigmoid activation function could have. And I chose ‘tanh’ as the activation for the final dense layer because the outputs should be within the range of 0 and 1 in each index.

I chose the loss function to be ‘binary_crossentropy’ and the optimizer to be ‘RMSprop’ because I heard those are good hyperparameters for recurrent neural networks, which LSTM is.

I split the data such that 70% of the data is training and 30% of the data is testing data. I chose this instead of 50-50 so that my model would have a bit more data to train with, since I ended up only using 1% of the data set.

My evaluation metric was the accuracy in predicting the sentiment of the tweet. My model ended up having around 83% accuracy.