capstone

November 12, 2019

1 Machine Learning Engineer Nanodegree

1.1 Capstone Project

Brian Long
November 10th, 2019

1.2 I. Definition

1.2.1 Project Overview

Twitter is a micro-blogging platform where users can post short messages referred to as "tweets". Tweets used to be limited to 140 characters but this limit was increased to 280 characters in 2017. This enforced conciseness made Twitter a popular platform for people to quickly voice their opinion about a topic without needing to publish a multi-page article. Millions of people write multiple tweets every day to express their thoughts or opinions, both formal and informal. Twitter has become a place for friends to share pictures of their lunch, as well as a place for professionals to discuss emerging practices and technologies.

The thing that interests me the most about Twitter as a social media platform is the frequency and openness with which people post. In many ways, a person's twitter feed is like a journal of their day to day life. There is a wealth of information that seems almost purposefully structured for data science. For example, the use of hashtags to label tweets allows them to be categorized by topic without breaking up the platform into multiple smaller communities that are defined by that topic. Using the public Twitter API, developers can access data about tweets that can then be aggregated and used to drive business decisions across a variety of fields.

Twitter stands out as a tool for sentiment analysis because the text differs heavily from the usual data that comes from sources such as movie reviews, product reviews, or news articles. These other sources are typically large bodies of text with proper grammar, whereas tweets are have a strict character limit which keeps them short while also having a very informal tone that doesn't follow grammatical convention. Twitter has been studied as a source of sentiment analysis in the following papers:

Go, A., Bhayani, R. and Huang, L., 2009. Twitter sentiment classification using distant supervision. CS224N Project Report, Stanford, 1(12). https://cs.stanford.edu/people/alecmgo/papers/TwitterDistantSupervision09.pdf

Agarwal, A., Xie, B., Vovsha, I., Rambow, O., Passonneau, R., Sentiment Analysis of Twitter Data. Columbia. http://www.cs.columbia.edu/~julia/papers/Agarwaletal11.pdf

1.2.2 Problem Statement

The information I would most like to extract from these tweets is the general sentiment in terms of whether it has a positive or negative tone. Sentiment analysis is a popular problem in the field of Natural Language Processing. The intricacies of language make it difficult to algorithmically extract meaning from a given body of text. While true sentiment contains much more complexity than can be expressed by the broad definitions of having a positive or negative point of view, this is still valuable information as it can be used to quantify the general public's feelings towards a company, product, or idea.

I will be attempting to create a model that can determine whether a tweet expresses a positive or negative sentiment. I plan to approach this task by using a variant of the Recurrent Neural Network architecture, known as a Long Short-Term Memory network, to better understand text as a sequence rather than a random collection of words. This should allow the model to take ordering of words into account, and draw meaning from the overall sequence as well as the individual words. Before feeding the training data into the model, I will be preprocessing the tweet content in an attempt to normalize the text. This should help to reduce the feature dimensionality of the data.

This model could eventually be used to build a tool that uses the Twitter API to have the sentiment evaluated for new tweets. This could further be used to show trends in how the general public views a given topic and leveraged to make business decisions.

1.2.3 Metrics

The main metric I will be using to measure the benchmark and solution models will be accuracy. I choose to focus on accuracy over precision or recall because mislabeling a tweet as positive or negative has the same impact. In this instance we are not specifically trying to identify all positive tweets or avoid incorrectly labeling a negative tweet as positive, but to correctly identify as many tweets as possible, regardless of whether they are positive or negative. Since the target class distribution is perfectly balanced between positive and negative tweets, using accuracy should not be affected by simply favoring one choice over the other. I will also be looking at F1-score as a secondary metric so that precision and recall are not entirely discounted, although my main focus will be on improving accuracy.

1.3 II. Analysis

1.3.1 Data Exploration

I will be using a subset of the Sentiment140 dataset to build my Twitter sentiment analysis model. This dataset is a collection of 1.6 million tweets that were gathered from the Twitter API and are labeled as positive or negative. I have cut down the dataset to 700k tweets and added column headers to the csv file included in this project. Since this dataset was intended for use in sentiment analysis, it was set up such that there is a perfect class balance with exactly 50% of the training data being positive and 50% being negative. The tweets were labeled based on their inclusion of emoticons, with variations of smiling emoticons being labeled positive, and variations of frowning emoticons being labeled negative. The emoticons were also removed from the dataset after the sentiment labels were added. This dataset was put together using distant supervised learning as a way to showcase its viability in labeling data this way. The data was tested against Naive Bayes, Maximum Entropy, and Support Vector Machine models, all of which scored over 80% accuracy. The complete Sentiment140 dataset is available at the following link:

http://help.sentiment140.com/for-students

Go, A., Bhayani, R. and Huang, L., 2009. Twitter sentiment classification using distant supervision. CS224N Project Report, Stanford, 1(12). https://cs.stanford.edu/people/alecmgo/papers/TwitterDistantSupervision09.pdf

Below, I will begin by importing the dataset and viewing a sample of the entries.

The dataset contains 6 columns: - sentiment: can be either 0 or 4. 0 represents a negative sentiment and 4 represents a positive sentiment. - tweet_id: the id of the tweet from the Twitter API. - date: the date that the tweet was posted to Twitter. - query: the search term used by to find the tweet. - user: the username of the person who posted the tweet. - tweet: the content of the tweet that was posted to Twitter.

The 'sentiment' column represents the label we will be attempting to predict.

```
In [1]: import pandas as pd
        import numpy as np

        pd.options.display.max_colwidth = 1000
        df = pd.read_csv("twitter_data_700k.csv")
```

```
In [2]: df.head()
```

```
Out[2]:    sentiment    tweet_id                          date     query  \
        0          0  1467810369  Mon Apr 06 22:19:45 PDT 2009  NO_QUERY
        1          0  1467810672  Mon Apr 06 22:19:49 PDT 2009  NO_QUERY
        2          0  1467810917  Mon Apr 06 22:19:53 PDT 2009  NO_QUERY
        3          0  1467811184  Mon Apr 06 22:19:57 PDT 2009  NO_QUERY
        4          0  1467811193  Mon Apr 06 22:19:57 PDT 2009  NO_QUERY

                      user  \
        0  _TheSpecialOne_
        1    scotthamilton
        2         mattycus
        3          ElleCTF
        4           Karoli


        0  @switchfoot http://twitpic.com/2y1zl - Awww, that's a bummer.  You shoulda got David
        1      is upset that he can't update his Facebook by texting it... and might cry as a re
        2                           @Kenichan I dived many times for the ball. Managed to save
        3                                                                  my whole body fee
        4      @nationwideclass no, it's not behaving at all. i'm mad. why am i here? because I
```

In the 'sentiment' column above, all of the values appear to be 0, indicating that the sentiment is negative. This is because the data is organized in such that the first half of the dataset is negative and the second half is positive. This can be seen further by examining the tail of the dataset.

```
In [3]: df.tail()
```

```
Out[3]:         sentiment    tweet_id                          date     query  \
        699995          4  2193601966  Tue Jun 16 08:40:49 PDT 2009  NO_QUERY
```

```
699996          4  2193601969  Tue Jun 16 08:40:49 PDT 2009  NO_QUERY
699997          4  2193601991  Tue Jun 16 08:40:49 PDT 2009  NO_QUERY
699998          4  2193602064  Tue Jun 16 08:40:49 PDT 2009  NO_QUERY
699999          4  2193602129  Tue Jun 16 08:40:50 PDT 2009  NO_QUERY

                    user  \
699995  AmandaMarie1028
699996      TheWDBoards
699997           bpbabe
699998     tinydiamondz
699999    RyanTrevMorris

                                                                                tweet
699995                        Just woke up. Having no school is the best feeling ever
699996  TheWDB.com - Very cool to hear old Walt interviews!   http://blip.fm/~8bmta
699997                       Are you ready for your MoJo Makeover? Ask me for details
699998              Happy 38th Birthday to my boo of alll time!!! Tupac Amaru Shakur
699999                 happy #charitytuesday @theNSPCC @SparksCharity @SpeakingUpH4H
```

Since the data is organized this way, I will want to shuffle it before training the model. A better representation of the data distribution can be seen by calling sample() on our dataset. The columns 'tweet_id', 'date', 'query', and 'user' were included in this dataset, but aren't particularly useful in determining the sentiment of the tweet. The 'tweet' column is the only feature I will be focusing on. I dropped the unnecessary features and sample the dataset below.

```
In [4]: df.drop(["tweet_id", "date", "query", "user"], axis=1, inplace=True)
        df.sample(10)

Out[4]:         sentiment  \
        44155           0
        262390          0
        265848          0
        74424           0
        603928          4
        426997          4
        70731           0
        641516          4
        154279          0
        370671          4

                                                                        tweet
        44155                                                    rip @TroyLLF h
        262390                                                          so sad
        265848             @tractorqueen its not fair  mourning for south east asian blockheads
        74424                                                     @lugowski oh
        603928                                  @Sweetnote I know but it does me good to
        426997  Motorola unveiled the first consumer DOCSIS3 cable modem today...  100mbit just
        70731                                    It would appear that the neighbours ha
```

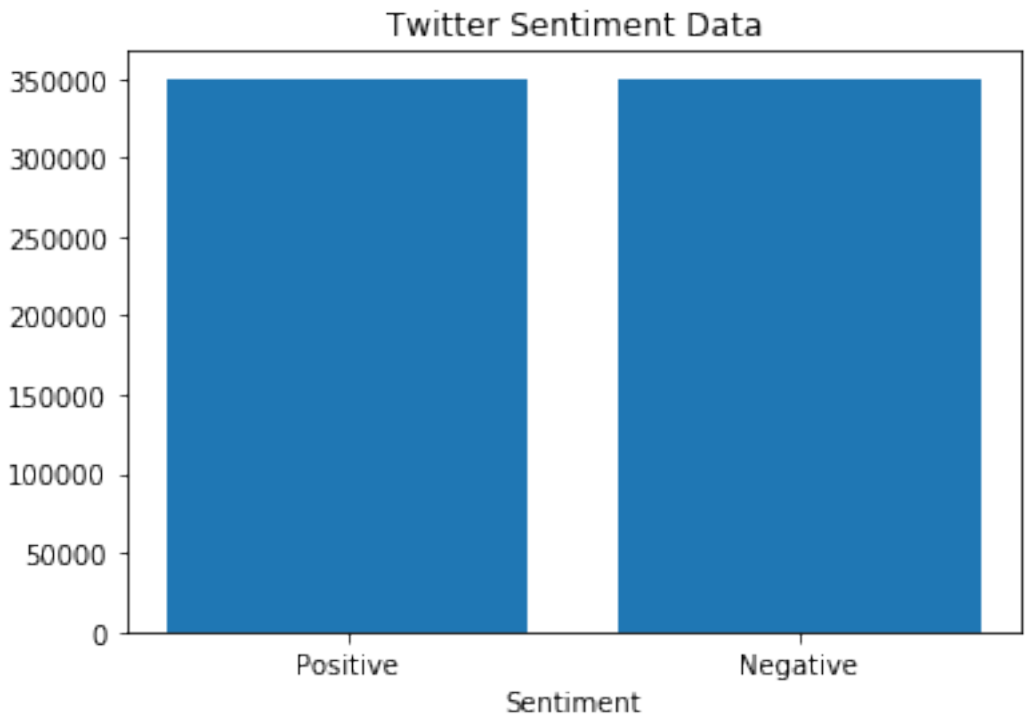
```
641516                          @u2gal I take it I'll be seeing you in Memphis, then?
154279                                                  @gothunts That sounds gha
370671    @jazziebabycakes oh yes, i looooove hitchhiker's guide to the galaxy! i'm rewa
```

As there are only 2 values representing sentiment, I should change that label to be either a 0 or 1 rather than using 4 for the positive sentiment. The tweet content also contains a lot of information that likely won't be useful in determining sentiment, such as usernames and urls. Another challenge will be the informality of speech used. Words are often exaggerated by repeating letters, words are misspelled, and sarcasm is common.

1.3.2 Exploratory Visualization

The bar graph below shows the perfect class balance of the dataset with the positive and negative tweets each having 350,000 instances.

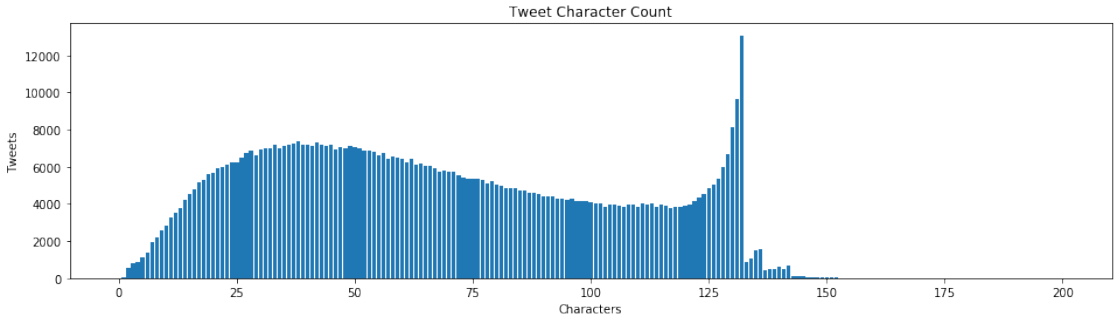
```
In [5]: import matplotlib.pyplot as plt
        %matplotlib inline

        pos = len(df.loc[df["sentiment"] == 4])
        neg = len(df.loc[df["sentiment"] == 0])

        labels = ["Positive", "Negative"]
        values = [pos, neg]

        index = np.arange(len(labels))

        plt.title("Twitter Sentiment Data")
        plt.xlabel("Sentiment")
        plt.xticks(index, labels)
        plt.bar(index, values)
        plt.show()

        print(f"Positive Tweets: {pos}")
        print(f"Negative Tweets: {neg}")
```




```
Positive Tweets: 350000
Negative Tweets: 350000
```

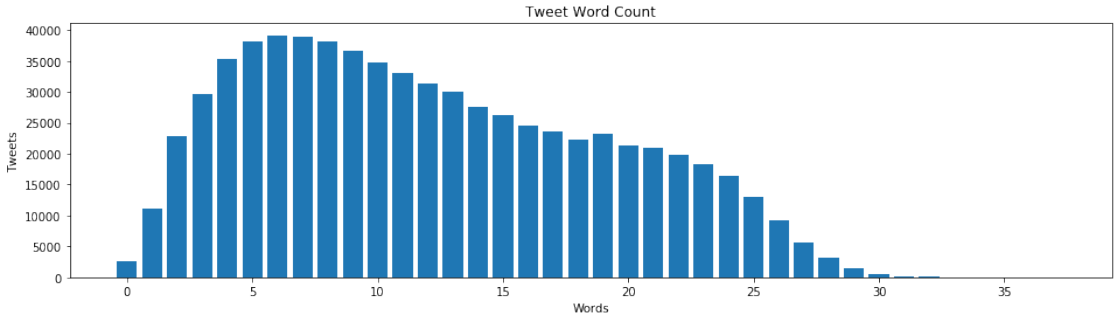
Below I show the character counts across the tweet data. The original 140 character limit is shown clearly here, with a few instances using a couple more characters which must be after the update to allow 280 characters. It seems to be a somewhat level distribution after about 15 characters, with a large number of tweets being "short and sweet" at around 25-50 characters, followed but a slight dip until a spike in tweets right near the character limit.

```
In [6]: char_count = df["tweet"].apply(lambda x: len(x))
        char_dict = char_count.groupby(char_count).count().to_dict()

        index = np.arange(len(char_dict))

        plt.figure(figsize=(16,4))
        plt.title("Tweet Character Count")
        plt.xlabel("Characters")
        plt.ylabel("Tweets")
        plt.bar(index, char_dict.values())
        plt.show()
```



Next, I will look at the word count in these tweets. While this does relate closely to the character count, we get additional information here which can be useful in future steps when we will need to tokenize and pad the tweets to all be the same length. It appears that most tweets contain around 5 to 10 words. The number of tweets then trail off and don't seem to extend beyond 35 words.

```
In [7]: word_count = df["tweet"].apply(lambda x: len(x.split()))
        word_dict = char_count.groupby(word_count).count().to_dict()

        index = np.arange(len(word_dict))

        plt.figure(figsize=(16,4))
        plt.title("Tweet Word Count")
        plt.xlabel("Words")
        plt.ylabel("Tweets")
        plt.bar(index, word_dict.values())
        plt.show()
```



1.3.3 Algorithms and Techniques

I intend to use a Recurrent Neural Network (RNN) as a classifier for the sentiment of the tweets. RNN's, and in particular Long Short-Term Memory networks (LSTMs), have gained popularity as a tool for natural language processing tasks because they have the ability to understand a sentence as a sequence of words rather than a random jumbling of words.

RNN's do this by looping through steps in a sequence and maintaining the state of its previous outputs to use as input in the next step. RNN's in their most basic form have issues with vanishing and exploding gradients. Vanishing gradient is when early layers in a model are unable to adjust their weights enough for significant learning to occur. This is because backpropagation calculates the gradient of a layer as a product of the derivatives of the following layers from the output layer backwards. If these gradients are smaller than 1, multiplying them together can quickly bring the number close to 0. This leads to an RNN only being able to remember very recent steps when processing a sequence. Exploding gradient is the opposite, when values larger than 1 are amplified by backpropagation. Exploding gradients can be avoided with gradient clipping, which basically sets an upper bound on a gradient value. Vanishing gradients are trickier to prevent.

LSTMs are a variation of the RNN architecture that solves the vanishing gradient problem by maintaining a cell state and selectively remembering or forgetting information at each step. A forget gate is responsible deciding which information should be removed from or stay in the cell state, an input gate determines what new information should be added, and an output gate decides which information from the call state should be used as input in the following step. These gate functions have their own weights which are adjusted during learning, much like a neural network node. These gates regulate the cell state and allow the LSTM to have a better understanding of long term dependencies in sequences.

This leads to a sophistication of understanding that other "bag-of-words" models can't achieve. For example, given the text "My team did not win", a bag-of-words model might see the word "win" and predict that the text is positive, where a RNN might notice the negation in "did not win" and be able to predict that the text is negative.

I plan to do some preprocessing on the tweet content to try to normalize the text, as the informal and concise nature of Twitter leads to some very creative spelling and grammar. This will mostly involve removing text that doesn't seem useful in determining the sentiment of the tweet.

After the text content has gone through preprocessing. I will train an LSTM network on the data using an embedding layer as the input. The embedding layer is used to create word vector representations and is more efficient than using one-hot encoding as they can lead to a huge amount of features that are mostly 0's.

1.3.4 Benchmark

I will be using a Naive Bayes model with a Bag of Words approach as my benchmark model. Naive Bayes is often used as a benchmark in Natural Language Processing problems since the implementation is simple but fairly effective. This model will then be tested for accuracy in terms of how many tweets were correctly identified as positive or negative.

Before testing the benchmark model, I first separate the data into labels and features. The labels will be the sentiment, which I will also run through a mapping function to convert all of the 4's into 1's. The feature set will consist of only the tweet content. I then split the dataset into training and testing subsets using scikit-learn's train_test_split.

```
In [8]: from sklearn.model_selection import train_test_split

        y = df['sentiment'].map({0: 0, 4: 1}) # here I am mapping the labels to be 0 and 1 rathe
        X = df.drop(['sentiment'], axis=1)
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.25, random_state=42
```

I will tokenize the tweet content using a CountVectorizer. This will keep track of the number

of times each feature/word appears in a given instance. A Naive Bayes model is then fit to the training data before predicting the sentiment class of the test data.

```
In [9]: from sklearn.feature_extraction.text import CountVectorizer

        count_vector = CountVectorizer()

        train_vector = count_vector.fit_transform(X_train['tweet'])
        test_vector = count_vector.transform(X_test['tweet'])

In [10]: from sklearn.naive_bayes import MultinomialNB
         naive_bayes = MultinomialNB()
         naive_bayes.fit(train_vector, y_train)

Out[10]: MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)

In [11]: predictions = naive_bayes.predict(test_vector)

In [12]: from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

         print(confusion_matrix(y_test,predictions))
         print(classification_report(y_test,predictions))
         print(accuracy_score(y_test, predictions))

[[72530 14882]
 [23765 63823]]
              precision    recall  f1-score   support

           0       0.75      0.83      0.79     87412
           1       0.81      0.73      0.77     87588

   micro avg       0.78      0.78      0.78    175000
   macro avg       0.78      0.78      0.78    175000
weighted avg       0.78      0.78      0.78    175000

0.77916
```

The accuracy of the benchmark Naive Bayes model is about 78%.

1.4 III. Methodology

1.4.1 Data Preprocessing

The labels were already altered in the benchmarking step to have positive tweets labeled with a 1 rather than a 4, but the tweet content will still need a good amount of preprocessing due to the informal nature of the tweets. I will apply a series text substitutions using regular expressions to make the tweets a little more uniform and to remove unnecessary information to reduce the dimensionality of the feature set.

First, all characters will be converted lowercase. Usernames int the format @user and hashtags in the format #tag will be removed as these can often refer to a subject but do not a sentiment. Urls are removed as they may link to relevant information, but don't carry that information in url form. Repeated letters are reduced down to only repeating the letter twice so that words like "hellooooo" and "hellllooooo" would both map to "helloo". While this is not ideal, as they should map to "hello", some repeated letters are valid, such as the "ll" in "hello", so reducing all repeated characters to a single character would have its own problems. Punctuation that is used to break up a thought, such as a comma, period, or exclamation point, is replaced with a space since it is common to forget to add that space and simply removing it could combine the two words around it. Other punctuation, such as an apostrophe, is simply removed to combine the letters in contractions rather than treating them as separate words, for example "don't" will be "dont" instead of "don" and "t". Finally all words with less than 3 characters are removed to get rid of filler words that dont add much to a sentence like "as" or "of".

This cleans up the tweet content by a significant amount, but the informal speech will still have some issues that are harder to detect and fix such as sarcasm.

```
In [13]: import re

         def processTweet(tweet):
             tweet = tweet.lower() # lowercase
             tweet = re.sub("[@|#]\w+\S","", tweet) # remove @usernames and #hashtags
             tweet = re.sub("http[s]?://[\S]+", '', tweet) # remove urls
             tweet = re.sub(r"(.)\1\1+",r"\1\1", tweet) # remove letters that repeat more than 2
             tweet = re.sub('[!&()+,-./:;<=>?[\\]_{|}~]', ' ',tweet) # replace certain punctuati
             tweet = re.sub('[!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~]', '',tweet) # remove remaining
             tweet = re.sub(r'\b\w{1,2}\b', '', tweet) # remove words that are less than 3 chara
             return tweet

In [14]: df['tweet'] = df['tweet'].map(lambda tweet: processTweet(tweet))

         df.head()

Out[14]:    sentiment  \
         0          0
         1          0
         2          0
         3          0
         4          0


         0                                    aww  thats  bummer   you shoulda got david carr  t
         1   upset that  cant update his facebook  texting    and might cry   result  school tod
         2                                dived many times for the ball  managed  save   the res
         3                                                        whole body feels itchy and l
         4                    its not behaving  all   mad  why   here  because  cant see you al
```

1.4.2 Implementation

I first run the data through a tokenizer so that the sentences are broken down into sequences. A maximum vocabulary size is used so that less important features are discarded. After the tweets are tokenized, I add padding to the shorter sequences to ensure that they will all be the same length to be fed into the neural network. I decided that the maximum length should be 35 based on the word count graph in the Exploratory Visualization section.

```
In [15]: from keras.preprocessing import sequence, text
         from keras.utils import to_categorical

         from sklearn.feature_extraction.text import TfidfVectorizer

         vocabulary_size = 20000

         tokenizer = text.Tokenizer(num_words=vocabulary_size)
         tokenizer.fit_on_texts(X_train['tweet'])

         train_token = tokenizer.texts_to_sequences(X_train['tweet'])
         test_token = tokenizer.texts_to_sequences(X_test['tweet'])

         max_words = 35
         train_padded = sequence.pad_sequences(train_token, maxlen=max_words)
         test_padded = sequence.pad_sequences(test_token, maxlen=max_words)
```

```
Using TensorFlow backend.
```

After the input data is prepared, I define the architecture for the neural network. I will use an embedding layer layer as input, with the input dimensions being equal to the vocabulary size. The input length of each sequence will be max_words which is the length of all of the sequences after padding. The output dimension of the embedding layer will be 32, which is a number I found through trial and error. This will then feed into an LSTM layer, which will have a dropout rate of 0.5 to prevent overfitting. The output of the LSTM layer will go into a Fully-Connected layer which will determine the final prediction using a sigmoid activation function. The summary can be seen below.

```
In [16]: from keras import Sequential
         from keras.layers import Embedding, LSTM, Dense, Dropout, Activation

         embedding_size=32

         model = Sequential()
         model.add(Embedding(vocabulary_size, embedding_size, input_length=max_words))
         model.add(LSTM(32, dropout=0.5))
         model.add(Dense(1, activation='sigmoid'))

         print(model.summary())
```

```
WARNING: Logging before flag parsing goes to stderr.
W1110 17:10:46.975821 140190382028544 deprecation_wrapper.py:119] From /home/blong/anaconda3/lib

W1110 17:10:46.991589 140190382028544 deprecation_wrapper.py:119] From /home/blong/anaconda3/lib

W1110 17:10:46.994090 140190382028544 deprecation_wrapper.py:119] From /home/blong/anaconda3/lib

W1110 17:10:47.061294 140190382028544 deprecation_wrapper.py:119] From /home/blong/anaconda3/lib

W1110 17:10:47.067534 140190382028544 deprecation.py:506] From /home/blong/anaconda3/lib/python3
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate = 1 - keep_prob`.


_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 35, 32)            640000
_________________________________________________________________
lstm_1 (LSTM)                (None, 32)                8320
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 33
=================================================================
Total params: 648,353
Trainable params: 648,353
Non-trainable params: 0
_________________________________________________________________
None
```

```
In [17]: loss_func = 'binary_crossentropy'

         model.compile(loss=loss_func,
                       optimizer='adam',
                       metrics=['accuracy'])
```

```
W1110 17:10:47.222622 140190382028544 deprecation_wrapper.py:119] From /home/blong/anaconda3/lib

W1110 17:10:47.236896 140190382028544 deprecation_wrapper.py:119] From /home/blong/anaconda3/lib

W1110 17:10:47.242471 140190382028544 deprecation.py:323] From /home/blong/anaconda3/lib/python3
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
```

Since there are only two classes as output, positive (1) and negative (0), binary crossentropy will be used as the loss function when compiling the model.

When training the model, I will use two callbacks, ModelCheckpoint for saving the best performing model during training, and EarlyStopping to prevent the training from continuing if the

model isn't improving anymore. Both the ModelCheckpoint and the EarlyStopper will be using validation accuracy as the metric to monitor. This allows me to set the epochs at a higher number than I expect to be beneficial and allow the model to stop itself when it isn't learning anymore. The data is then shuffled and split into 80% training and 20% validation sets and the model is trained on it.

```
In [18]: from keras.callbacks import ModelCheckpoint, EarlyStopping

         saved_model_path = 'model.weights.best.hdf5'

         checkpointer = ModelCheckpoint(filepath=saved_model_path,
                                        monitor='val_acc',
                                        save_best_only=True,
                                        verbose=1)

         early_stop = EarlyStopping(monitor='val_acc',
                                    patience=3,
                                    verbose=1)

         batch_size = 32
         num_epochs = 20
         model.fit(train_padded,
                   y_train,
                   validation_split=0.2,
                   batch_size=batch_size,
                   epochs=num_epochs,
                   shuffle=True,
                   callbacks=[checkpointer, early_stop])
```

```
Train on 420000 samples, validate on 105000 samples
Epoch 1/20
420000/420000 [==============================] - 483s 1ms/step - loss: 0.4494 - acc: 0.7895 - va

Epoch 00001: val_acc improved from -inf to 0.80579, saving model to model.weights.best.hdf5
Epoch 2/20
420000/420000 [==============================] - 479s 1ms/step - loss: 0.4016 - acc: 0.8170 - va

Epoch 00002: val_acc improved from 0.80579 to 0.81350, saving model to model.weights.best.hdf5
Epoch 3/20
420000/420000 [==============================] - 485s 1ms/step - loss: 0.3824 - acc: 0.8276 - va

Epoch 00003: val_acc improved from 0.81350 to 0.81642, saving model to model.weights.best.hdf5
Epoch 4/20
420000/420000 [==============================] - 490s 1ms/step - loss: 0.3690 - acc: 0.8353 - va

Epoch 00004: val_acc did not improve from 0.81642
Epoch 5/20
420000/420000 [==============================] - 489s 1ms/step - loss: 0.3582 - acc: 0.8408 - va
```

```
Epoch 00005: val_acc improved from 0.81642 to 0.81734, saving model to model.weights.best.hdf5
Epoch 6/20
420000/420000 [==============================] - 484s 1ms/step - loss: 0.3501 - acc: 0.8456 - va

Epoch 00006: val_acc did not improve from 0.81734
Epoch 7/20
420000/420000 [==============================] - 484s 1ms/step - loss: 0.3427 - acc: 0.8493 - va

Epoch 00007: val_acc did not improve from 0.81734
Epoch 8/20
420000/420000 [==============================] - 486s 1ms/step - loss: 0.3365 - acc: 0.8525 - va

Epoch 00008: val_acc did not improve from 0.81734
Epoch 00008: early stopping


Out[18]: <keras.callbacks.History at 0x7f7ff9b25cf8>
```

After the training has completed, I will load the best weights into the model. Then I will use the model to predict the sentiment of the test data.

```
In [19]: model.load_weights(saved_model_path)

In [20]: rnn_pred = model.predict_classes(test_padded)

In [21]: print(confusion_matrix(y_test, rnn_pred))
         print(classification_report(y_test, rnn_pred))
         print(accuracy_score(y_test, rnn_pred))

[[72197 15215]
 [16438 71150]]
              precision    recall  f1-score   support

           0       0.81      0.83      0.82     87412
           1       0.82      0.81      0.82     87588

   micro avg       0.82      0.82      0.82    175000
   macro avg       0.82      0.82      0.82    175000
weighted avg       0.82      0.82      0.82    175000

0.8191257142857142
```

The final RNN model has an accuracy of about 82%. This is a slight improvement from the benchmark accuracy of about 78%.

1.4.3 Refinement

I went through a lot of variations of model parameters and neural network architecture before landing on the final model. During this time I sampled the full dataframe to only use a subset of the data for quicker iteration. I experimented with different numbers of words in the vocabulary, I saw an increase in performance as the number went up, but also significant slow down in the training process. I found a good compromise at 20,000 words in the vocabulary. I found that large batch sizes performed slightly poorer than smaller batch sizes. I reduced the batch size until the improvements seemed to stagnate and ended up with a batch size of 32.

Originally I was training the model for a fixed number of epochs. When I noticed the quality of the model was degrading in the later epochs, I decided to use a ModelCheckpointer to save the top performing model. I later added the EarlyStopping callback as well, to prevent the model from continuing to train when performance isn't increasing anymore. I then set the epochs to 20, which is more than I expect the improvements to last, and let the model decide when to stop training. The addition of these callbacks was where I found the greatest improvements in both performance and iteration speed.

For the neural network architecture, I ended up with a pretty simple model with one Embedding layer, one LSTM layer, and one Fully-Connected layer. I tried adding more Fully Connected layers, more LSTM layers, putting Dropout layers in between these layers, etc. but with every additional thing I added, I saw the performance drop. So I ended up going back to the simple architecture and then tweaked the dropout until I found the best results at 0.5. I also tried a few values for the output dimensions of the embedding layer as low as 10 and as high as 128 and found 32 to be the best fit.

1.5 IV. Results

1.5.1 Model Evaluation and Validation

Below I will test the model on some additional data to examine the results.

In [42]:
```
pos1 = "I had a great time at the concert tonight!"
pos2 = "This is the best blueberry muffin in the world."
pos3 = "Gotta love ice cream on a hot day"

neg1 = "I hate going to the dentist!"
neg2 = "My team did not win the game today"
neg3 = "This rainy weather is really getting me down"

ntrl1 = "The sky is blue"
ntrl2 = "The car is red"
ntrl3 = "The current month is November"

seq = tokenizer.texts_to_sequences([pos1, pos2, pos3, neg1, neg2, neg3, ntrl1, ntrl2, n
pad = sequence.pad_sequences(seq, maxlen=max_words)

pred_class = model.predict_classes(pad)
pred_prob = model.predict(pad)

np.concatenate([pred_class, pred_prob], axis=1)
```

```
Out[42]: array([[1.        , 0.98263985],
                [1.        , 0.98761964],
                [1.        , 0.89375877],
                [0.        , 0.00770876],
                [0.        , 0.01853481],
                [0.        , 0.05545729],
                [1.        , 0.71028316],
                [0.        , 0.28143016],
                [1.        , 0.64149153]])
```

I ran 3 positive, 3 negative, and 3 neutral texts through the model. The positive and negative texts were all identified correctly, even in the tricky case of neg3 where negation is used with a positive word in "did not win". However, neutral text is where this model starts to fall apart. This model is trained to output positive or negative and will always pick one, despite its level of certainty. In reality, a lot of text doesn't really express positive or negative sentiment. In the array shown above, "The sky is blue" is predicted to be a positive sentiment while "The car is red" is predicted to be negative. These are essentially the same sentences, just switching out the subject and the color. No sentiment is actually expressed in either case, but the model had to pick a classification. It is also interesting to see the confidence of the predictions in each case, as shown in the right column of the array. In the positive examples, the numbers are all very close to 1, and in the negative examples they are all very close to 0. In the neutral cases, you can see that the values are getting closer to the middle at 0.5, so it is possible that had there been a third class for neutral, that these texts could have been appropriately classified.

1.5.2 Justification

The final result of the RNN model was an accuracy of about 82% which is an improvement over the benchmark accuracy of 78%. I believe that in this instance the benchmark model performed pretty well, so the RNN model doesn't initially seem too impressive being only 4% more accurate. However, had the benchmark been a random guess of positive or negative, the benchmark accuracy would be about 50% and the improvement would seem huge at a 32% difference. With a base accuracy of 78%, there is only 22% room for improvement, so 4% is actually fairly significant.

I believe that the final solution did solve the problem, since being able to determine sentiment in a short amount of text with an 82% accuracy rate can be very useful as a tool. The output should not be taken as ultimate truth, but it will most likely be correct. It becomes more useful if you take the confidence values into account and not just the classification. This can help to alleviate the problem of neutral text. If you only consider the sentiment classification valid if positive is above a certain threshold such as 0.85 or negative is below 0.15 you can weed out a lot of the neutral texts and be more confident in the output of the model. The output of this model can then be used along with other data for decision making or as a feature to be used in another machine learning model.

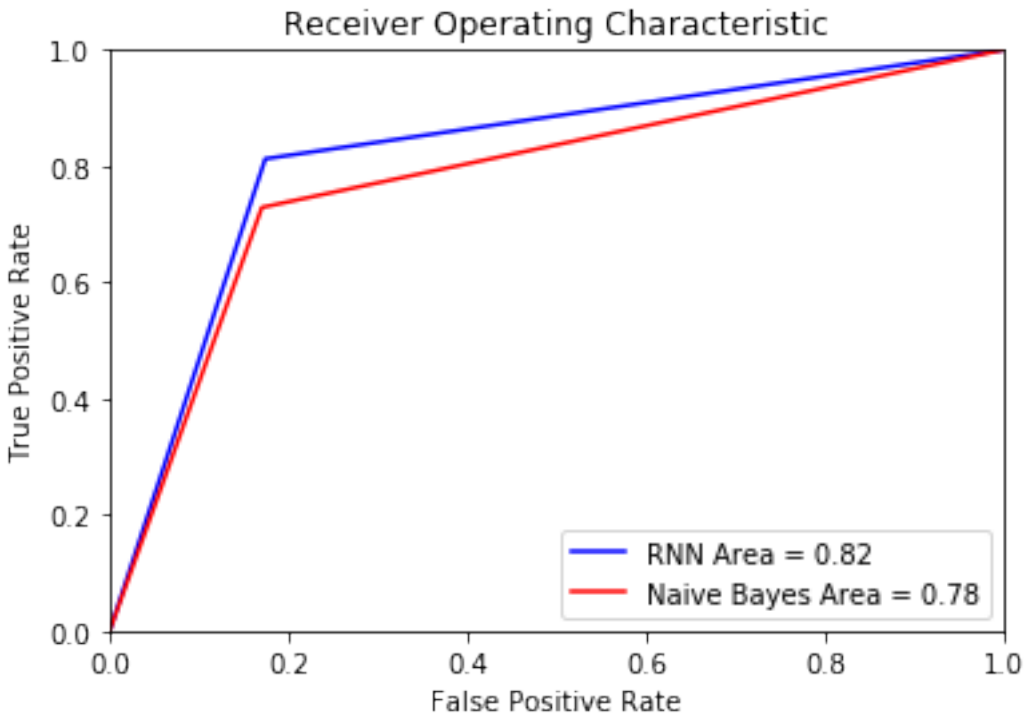
1.6 V. Conclusion

1.6.1 Free-Form Visualization

In the graph below, the Receiver Operating Characteristic (ROC) is shown for both the benchmark and the final model. Both models take up a majority of the graph, but the final model has a slightly bigger area under the curve and performs better at classification than the benchmark model. A

model with no ability to classify better than a random guess would have an area of 0.5 and look like a diagonal line from (0, 0) to (1, 1). The benchmark model had an area of 0.78 and the final model has an area of 0.82, both of which are much better at classification than a random guess.

```
In [22]: import sklearn.metrics as metrics

         naive_fpr, naive_tpr, naive_threshold = metrics.roc_curve(y_test, predictions)
         naive_roc_auc = metrics.auc(naive_fpr, naive_tpr)

         rnn_fpr, rnn_tpr, rnn_threshold = metrics.roc_curve(y_test, rnn_pred)
         rnn_roc_auc = metrics.auc(rnn_fpr, rnn_tpr)

         plt.title('Receiver Operating Characteristic')
         plt.plot(rnn_fpr, rnn_tpr, 'b', label = 'RNN Area = %0.2f' % rnn_roc_auc)
         plt.plot(naive_fpr, naive_tpr, 'r', label = 'Naive Bayes Area = %0.2f' % naive_roc_auc)
         plt.legend(loc = 'lower right')
         plt.xlim([0, 1])
         plt.ylim([0, 1])
         plt.ylabel('True Positive Rate')
         plt.xlabel('False Positive Rate')
         plt.show()
```



1.6.2 Reflection

The steps I took in solving the problem in this project are as follows: - Put together a subset of the Sentiment140 dataset with column headers - Import the dataset into a dataframe. - Drop all unnecessary columns from the dataframe. - Split the data into test and training sets. - Test the performance of a benchmark Naive Bayes model. - Preprocess the tweet data in an attempt to normalize the speech and shrink the dimensionality of features - Tokenize the tweet data and add padding to make all of the tweets the same length. - Build an Recurrent Neural Network consisting of Embedding, LSTM, and Dense layers. - Train the RNN on the twitter data using checkpoints and early stopping. - Test the accuracy of the RNN model against the benchmark model.

One of the interesting things I ran into during this project was coming up with the right regular expressions to use in the data preprocessing. I wanted to remove abnormalities in the text without affecting the overall message. In cases such as punctuation, it was easy to remove all punctuation or to replace all punctuation with a space but I found that it was better to pick and choose for each case based on how that punctuation is commonly used. For example, apostrophes should simply be removed so that "don't" becomes "dont" and not "don" and "t". However, periods should be replaced with a space since it is common for people to forget to add space following it even thought the ideas are separate. If the text was something like "okay...no thank you.", removing the periods would leave us with "okayno thank you", while replacing periods with a space would give us "okay no thank you". The extra space looks weird at first, but this allows the tokenizer to properly separate "okay" and "no".

Coming up with the right neural network architecture proved to be somewhat difficult mainly because of the time it takes to iterate and try a new architecture. NLP tasks require a lot of data to perform well and take a long time to train. I tried to mitigate this by working on a subset of the training data when testing out different architectures and tuning parameters. Using a smaller set of data gave me lower accuracy, but I was able to compare that accuracy against the other architectures that I was testing at the same number of samples to determine what was performing better. Then applying the full data in the final run gave me the higher accuracy that I was hoping for.

Overall, the model did meet my expectations for this project. After seeing the benchmark accuracy of about 78%, I was just hoping to break 80%, and ended up with an accuracy of about 82%. I think that this model can be useful to determine sentiment of short texts such as tweets of article headlines in a fairly reliable manner. The model becomes more useful when the prediction values are used alongside the predicted class. This can allow you to see exactly how confident the model is in its prediction and can help you choose to trust or ignore the output. This way, text that isn't strongly positive or negative can be filtered out or ignored.

1.6.3 Improvement

I believe that this implementation can be improved in a number of ways. One path I didn't get a chance to explore was using transfer learning in my model. There are some popular pre-trained embeddings that have been fit to extremely large datasets such as Google's Word2Vec and Stanford's GloVe. These could potentially improve the accuracy of the model as it would have a solid starting point before training begins. I also would have liked to try out more advanced NLP techniques such as stemming or lemmatization in the preprocessing step. This could help further reduce the dimensionality of the feature set by converting words into their base form. I also think that using a cleaner dataset would help improve the accuracy. The data was compiled in a semi-automated manner, by treating smiling emoticons as positive and negative emoticons as

negative. For most cases, this works out fine, but tweets commonly use incorrect emoticons to express sarcasm, so it is likely that some of the data was mislabeled. While a manually labeled dataset would be ideal, I wouldn't likely find as large a dataset. I think the sarcasm could also be mitigated by increasing the size of the dataset to a point where any sarcasm that was mislabeled becomes insignificant to the overall training.

In [ ]: