Twitter Sentiment Analysis

Problem

The mission of this challenge is to build a model that predicts if a tweet's writer’s opinion is positive, neutral, or negative about the company and the services it provides.

The Tweets available were scraped from Twitter in February 2015 about each major US airline. Contributors then classified each tweet as either “positive”, “neutral”, or “negative” and cited the reason for a negative classification as well as a confidence score for the assigned label.

There are 14,640 rows and 15 columns. The included features are tweet id, sentiment, sentiment confidence score, negative reason, negative reason confidence, airline, sentiment gold, name, retweet count, tweet text, tweet coordinates, time of tweet, date of tweet, tweet location, and user time zone.

Preliminary Exploratory Analysis

Exploratory analysis of our data revealed that negative tweets are much more than neutral or positive ones.

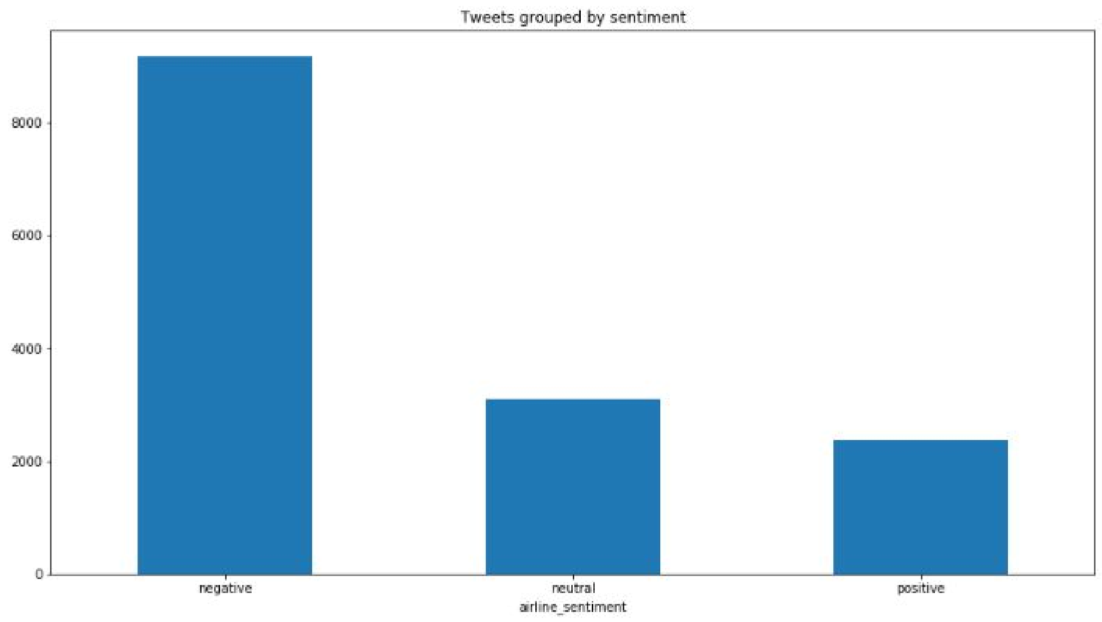


Figure 1 – Tweets grouped by sentiment

The distribution of the tweets per airline company can be found in the next Figure.

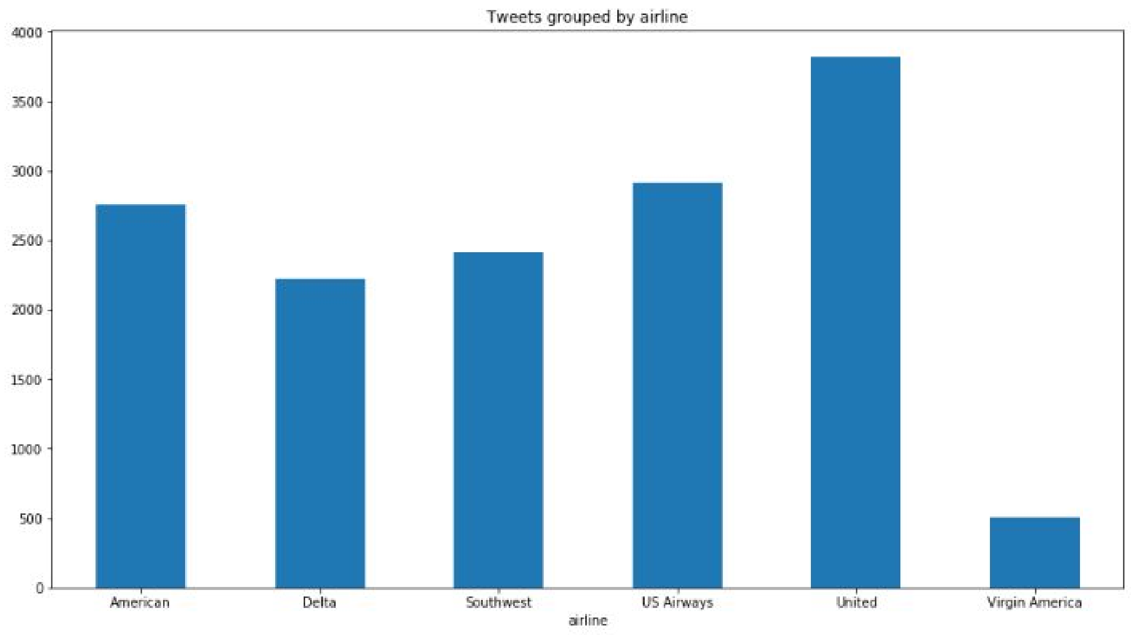


Figure 2 – Tweets grouped by airline company

If we combine these two bar-plots we created together, we can see the segmentation of tweets to positive, neutral and negative, per airline company.

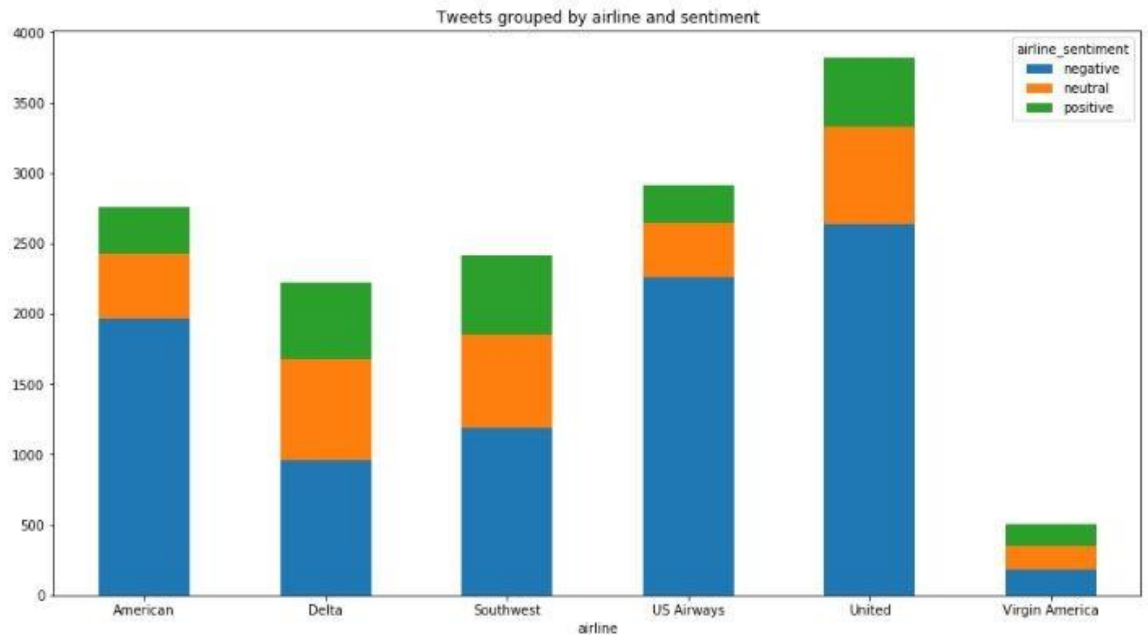
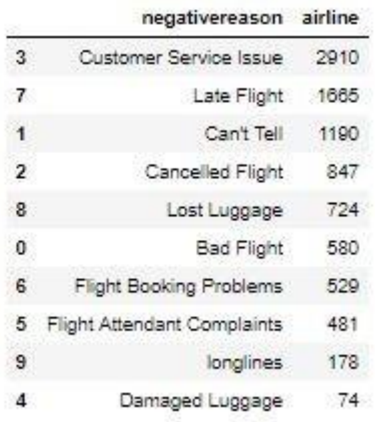


Figure 3 – Tweets grouped by airline and sentiment

We can easily see from the stacked bar-plot above that, United Airlines Company except having most of the comments in total is the one who gathered most of the negative comments when comparing to others. Virgin America Company, on the other hand, is the less popular airline company, since it gathered the least number of comments in total. Also, we can see that this company has an almost equal number of positive negative and neutral tweets. Most positive comments were gathered for Southwest Airline Company, Delta Airline Company and United Airline Company, as we can also see in the plot.

Finally, in the following table, we can see the reasons that lie behind the negative tweets. “Customer Service Issues”, flight delays (“Late Flight”) and other reasons, that aren’t defined explicitly (“Can’t Tell”), have gained the most negative tweets by customers. Also, Flight Cancellations (“Cancelled Flight”), “Lost Luggage” and “Bad Flight” experiences contribute also to negative tweets.



	negativereason	airline
3	Customer Service Issue	2910
7	Late Flight	1665
1	Can't Tell	1190
2	Cancelled Flight	847
8	Lost Luggage	724
0	Bad Flight	580
6	Flight Booking Problems	529
5	Flight Attendant Complaints	481
9	longlines	178
4	Damaged Luggage	74

Figure 4 – Reasons of negative tweets

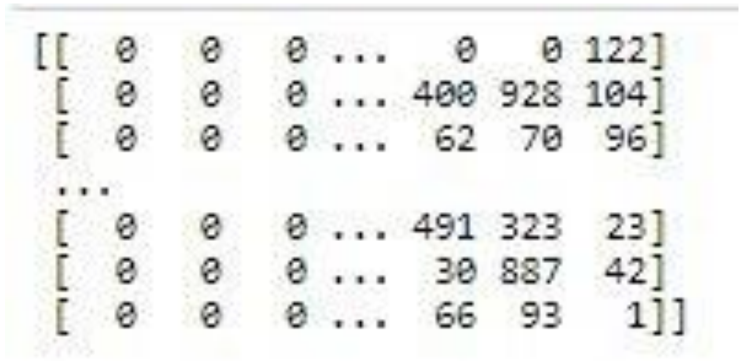
Data Processing and Representation

All the data was preprocessed in order to bring them to a certain state where they would give us the best results. The stages of this preprocessing were the following:

- Every tweet started with a reference to a specific airline company (i.e. “@AmericanAir”). I removed this as it has no semantic meaning.
- I also used regular expressions in order to remove any URL from the tweets.
- The final removals with the aid of regular expressions were all kinds of symbols, as well as numbers – letters only.
- I converted all tweets to lowercase.
- I removed all common words that do not have any significant semantic meaning such as “of”, “over”, “than” etc.
- Finally, by using Wordnet - the publicly available lexicon database for the English language - I conducted lemmatization, which is the transformation of

each word into a lemma. For example, transforming a word from plural to singular is one of the forms that lemmatization can take.

The **representation** of the data that I actually fed with the model were sequences of integers, each one representing one word. The sequences were padded with zeros on the left, up to the length of the largest one. They can be seen below:



```
[[  0   0   0 ...   0   0 122]
 [  0   0   0 ... 400 928 104]
 [  0   0   0 ...  62  70  96]
 ...
 [  0   0   0 ... 491 323  23]
 [  0   0   0 ...  30 887  42]
 [  0   0   0 ...  66  93   1]]
```

Figure 5 – The final sequences fed into our RNN model

My Approach and Why

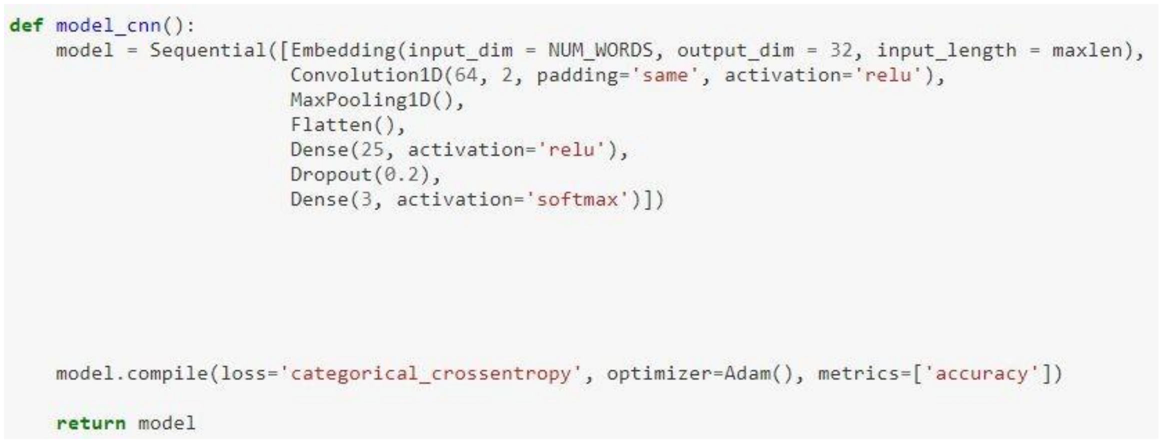
The combination of both machine learning and lexicon-based approaches can improve remarkably the sentiment classification performance (hybrid technique). The main advantage of the hybrid approach using the combination of the lexicon/learning approaches is to attain the best of both worlds - readability from a carefully designed lexicon, but also high accuracy from a powerful supervised learning algorithm. This system uses a sentiment lexicon constructed using public resources for initial sentiment detection.

Neural networks have been traditionally used for sentiment analysis with the use of loops within the network architecture to model language dependencies. Firstly, Convolutional Neural Networks (CNNs) learn to capture features regardless of where these might be. It makes sense to choose a CNN for classification tasks like sentiment classification since sentiment is usually determined by some key phrases. At the same time, Recurrent Neural Networks (RNNs) are popular models that have shown great promise in many NLP projects. A Recurrent Neural Network (RNN) creates loops between each node in the neural network. This makes it a very good candidate for sequential data, such as text. It can process sequences and create a state which contains information about what the network has seen so far. This is why RNNs are useful for natural language processing because sentences are decoded word-by-word while keeping the memory of the words that came beforehand to give better context for understanding. An RNN allows information from a previous output to be fed as input into the current state. Simply put, we can use previous information to help make a current decision.

So, the approach I decided to follow was to use a deep learning technique by utilizing Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNN) in order to create a statistical model.

First Model: Convolutional Neural Networks (CNNs) - Explanation

The model type that I used is Sequential. It allows us to build a model layer by layer. It is depicted in Figure 6.



```
def model_cnn():
    model = Sequential([Embedding(input_dim = NUM_WORDS, output_dim = 32, input_length = maxlen),
                        Convolution1D(64, 2, padding='same', activation='relu'),
                        MaxPooling1D(),
                        Flatten(),
                        Dense(25, activation='relu'),
                        Dropout(0.2),
                        Dense(3, activation='softmax')])




    model.compile(loss='categorical_crossentropy', optimizer=Adam(), metrics=['accuracy'])

    return model
```

Figure 6 - Parameters of CNN model

Firstly, the CNN model consists of an initial convolution layer that receives word embeddings for each token in the tweet as inputs. The embedding layer learns to map word vectors into a lower-dimensional vector space where distances between words correspond to how related they are. In our case, the word embedding learned from scratch.

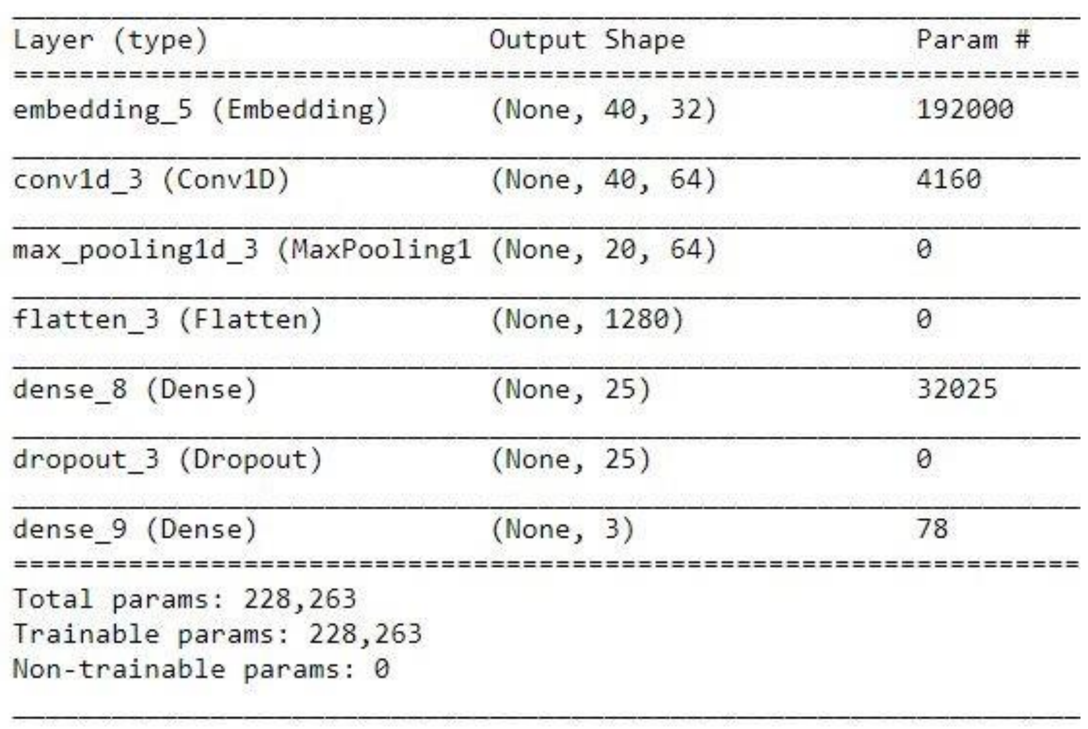
In this layer, I passed the top 6000 most common tokens, defined the output dimension equals to 32 and maximum length of a tweet equals to 40 tokens. So each tweet is **represented** by a 45x32 matrix in the end. It is important to say that the maximum number of words in a sentence within the data is 33. Thus, I determined the maximum length to be a bit longer than this, like 40 (maxlen=40).

The next layer in the network performs convolutions over the ordered embedded word vectors in a tweet using multiple filter sizes. This is the equivalent of looking at all 3-grams, 4-grams and 5-grams in a sentence and will allow us to understand how words contribute to sentiment in the context of those around them.

So, I applied 1D convolutions that are useful for the text classification problem. The reason is that they can learn patterns and then recognize them at different positions in sequences. It could capture a negative phrase such as "don't like" regardless of where it happens in the tweet. The convolutions have width 32 and different height in order to learn patterns of different word lengths.

Firstly, I applied 64 filters with kernel size of 2, meaning we have a 2x32 filter matrix and “RELU” function for nonlinear transformation.

Looking at the summary of the model we can check the dimensions of each output layer. The output width reflects the number of filters we apply, so we will have 40X64 dimension output. The results are shown in Figure 7



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_5 (Embedding)      (None, 40, 32)            192000
_________________________________________________________________
conv1d_3 (Conv1D)            (None, 40, 64)            4160
_________________________________________________________________
max_pooling1d_3 (MaxPooling1 (None, 20, 64)            0
_________________________________________________________________
flatten_3 (Flatten)          (None, 1280)              0
_________________________________________________________________
dense_8 (Dense)              (None, 25)                32025
_________________________________________________________________
dropout_3 (Dropout)          (None, 25)                0
_________________________________________________________________
dense_9 (Dense)              (None, 3)                 78
=================================================================
Total params: 228,263
Trainable params: 228,263
Non-trainable params: 0
_________________________________________________________________
```

Figure 7 - Output Dimensions

Then, I added Max Pooling layer. The pooling layer will extract the maximum value from each one of 64 filters, and the output dimension will be a just 1-dimensional vector with length equals to 64.

Next, I add dropout regularization, which will randomly disable a fraction of neurons in the layer to ensure that the model does not overfit. This prevents neurons from co-adapting and forces them to learn individually useful features. In our case set it equal to 20%.

The output from the global max pooling layer fed to a fully connected layer. At this layer there are 25 neurons and “relu” functions to verify the nonlinearity of the model.

Finally, classified the resulting output of this layer using the “softmax” function. The output layer consists of 3 neurons and yields a result among 0 (negative sentiment), 1(neutral sentiment) and 2(positive sentiment).

Compiling the model

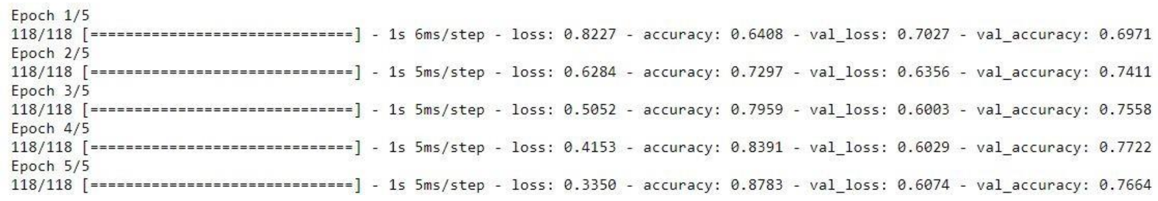
Compiling the model takes three parameters: optimizer, loss and metrics.

The optimizer controls the learning rate. I used ‘adam’ as the optmizer. Adam is generally a good optimizer to use for many cases. The adam optimizer adjusts the learning rate throughout the training. The learning rate determines how fast the optimal weights for the model are calculated. A smaller learning rate may lead to more accurate weights (up to a certain point), but the time it takes to compute the weights will be longer.

I also used ‘categorical_crossentropy’ for our loss function. This is the most common choice for classification. A lower score indicates that the model is performing better. To make things even easier to interpret, we used the ‘accuracy’ metric to see the accuracy score on the validation set when we train the model.

Training the model

Now we will train our model. To train, we will use the ‘fit()’ function on our model with the following parameters: training data (X_train), target data (encoded_Y), validation data, and the number of epochs. The results are shown in Figure 8.



```
Epoch 1/5
118/118 [==============================] - 1s 6ms/step - loss: 0.8227 - accuracy: 0.6408 - val_loss: 0.7027 - val_accuracy: 0.6971
Epoch 2/5
118/118 [==============================] - 1s 5ms/step - loss: 0.6284 - accuracy: 0.7297 - val_loss: 0.6356 - val_accuracy: 0.7411
Epoch 3/5
118/118 [==============================] - 1s 5ms/step - loss: 0.5052 - accuracy: 0.7959 - val_loss: 0.6003 - val_accuracy: 0.7558
Epoch 4/5
118/118 [==============================] - 1s 5ms/step - loss: 0.4153 - accuracy: 0.8391 - val_loss: 0.6029 - val_accuracy: 0.7722
Epoch 5/5
118/118 [==============================] - 1s 5ms/step - loss: 0.3350 - accuracy: 0.8783 - val_loss: 0.6074 - val_accuracy: 0.7664
```

Figure 8 - Validation accuracy and loss

For our validation data, we will use the test set provided to us in our dataset, which we have split into X_test and encoded_Y_test. The number of epochs is the number of times the model will cycle through the data. As we can see from the results above, the best validation accuracy is 77.22% and epochs equal to 3. So, we can reduce the epochs to 3 in order to save calculation time. Finally, the batch size is equal to 100. It means that tweets pass through network in smaller species of 100 tweets each time.

Loss and Accuracy

As we can see from the plots below, Figure 9 and 10 the loss function at the validation dataset increases slightly after epoch 3. At the same time, after the epoch 3 the accuracy remains almost the same. So, we can conclude that epoch equal to 3 is a good value for the model.

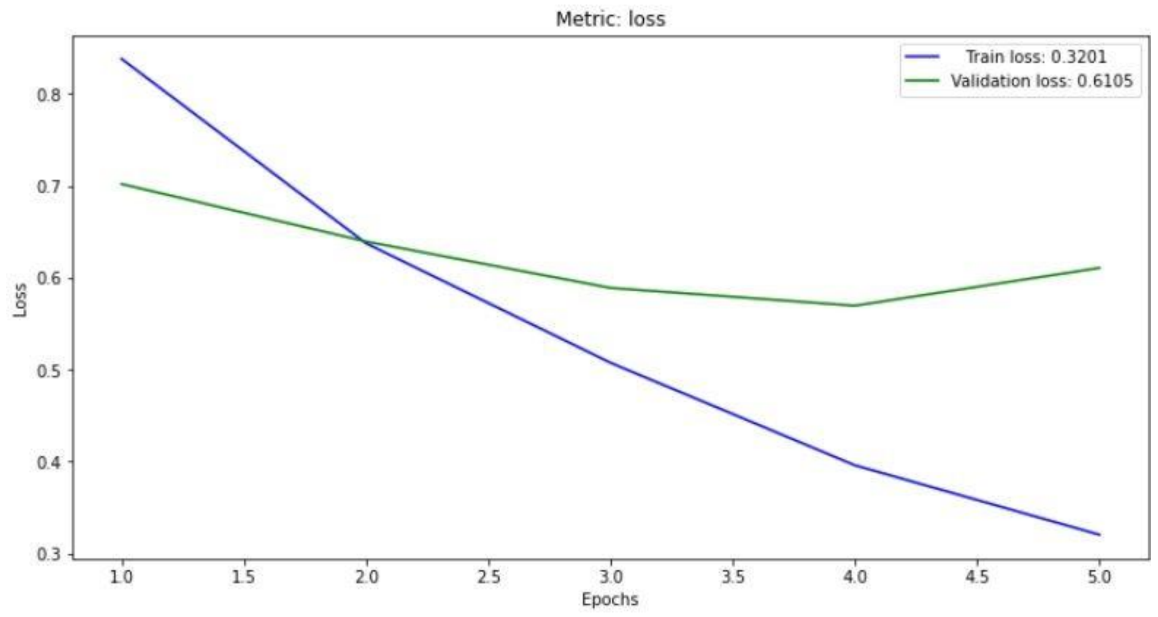


Figure 9 - Loss function for CNN model

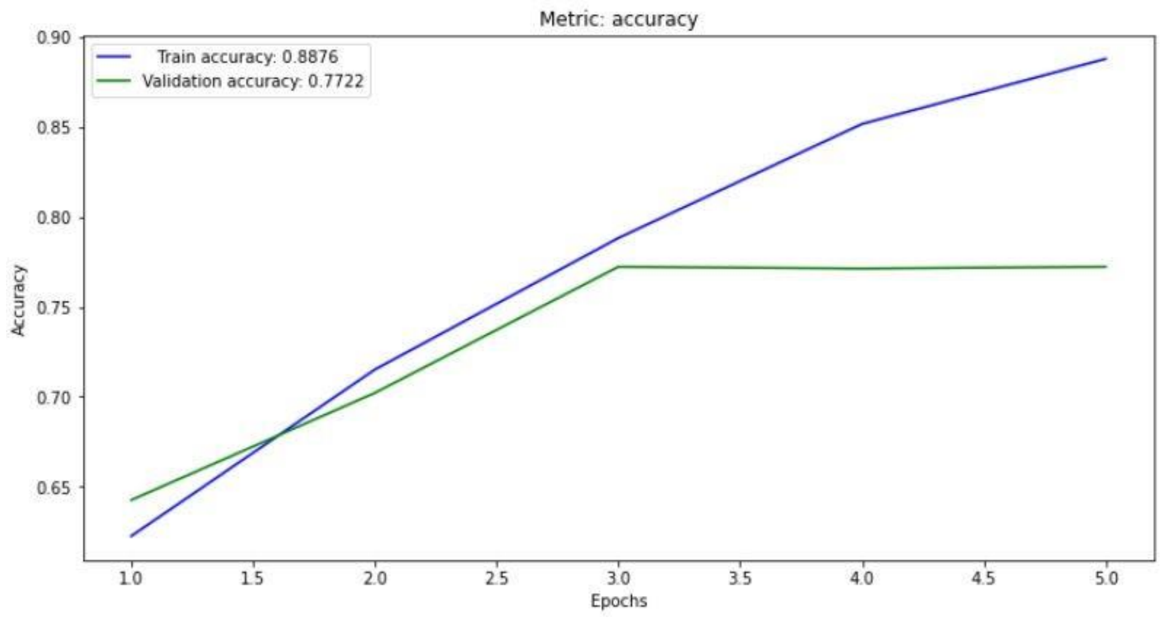


Figure 10 - Accuracy for CNN

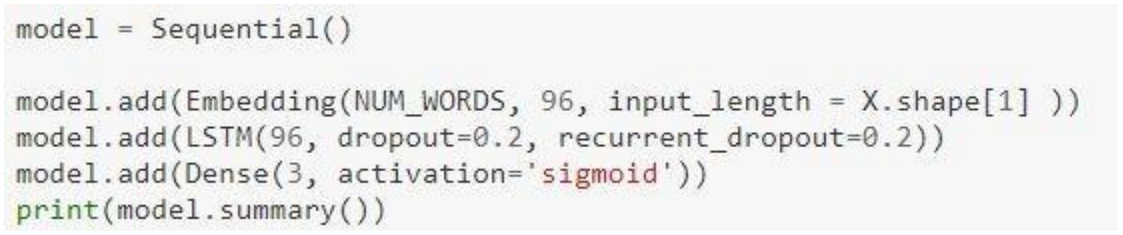
In conclusion, the accuracy of CNN model is equal to 77.2%. It is a good value, but I decided to try for a better value.

Improving CNN

Taking into account the above insights I created a new CNN model with the same number of inputs, output dimension and maximum length. But I applied 128 filters of kernel size 3 and run it for 3 epochs. Eventually, the accuracy on validation dataset is smaller in this case (74.8%). So, I remained with the initial CNN model.

Second Model: Recurrent Neural Networks (RNNs) - Explanation

RNNs are designed in such way that they take under consideration the previous observations, unlike for example conventional feed-forward neural networks. Along with the improvement provided by LSTM networks which offer a mechanism to remember or forget things, they are the state of art solution in handling sentiment analysis problems. The first layer that I used is an embedding layer. This is a transformation layer used to turn the provided indexes into dense vectors of fixed size. Word embeddings allow us to **represent**(embed) words in a continuous vector space where semantically similar words are mapped to nearby points. The embedding space I chose had 96 dimensions, because, as can be seen by the diagrams below, this value gave me better results. There is not any rule for choosing some specific dimension value and this value does not appear to have a significant effect on the accuracy of the model.



```
model = Sequential()

model.add(Embedding(NUM_WORDS, 96, input_length = X.shape[1] ))
model.add(LSTM(96, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(3, activation='sigmoid'))
print(model.summary())
```

Figure 12: RNN Archcitecture

The next layer is an LSTM (Long Short-Term Memory) layer with 96 memory units and some dropout values to avoid overfitting. I used LSTM as a special form of RNNs, as they are especially powerful when it comes to finding the right features when the chain of input-chunks becomes longer.

RNNs are powerful models but overfit quickly. In order to avoid overfitting I should accompany our LSTM layer by a Dropout layer. This will ignore randomly selected neurons during the training phase, therefore reducing the sensitivity to specific weights.. After many trials I decided to apply a dropout of 20% to the inputs and connections and not on the outputs which would require adding one more output layer. Finally, I added a

Dense layer which we used to change the dimensions of our vector from 96 to just 3, as many as our label categories are.

Model Input

In order to bring the data into a proper form to feed it in the model, I passed them through a tokenizing procedure. This consists of choosing the 2000 most used words and building an internal dictionary that contains all of them by assigning to each one an index. Next, I replaced each word with its index. So, we have now transformed the sequences of the words into sequences of integers. Next, as the sequences need to have the same length I added some zeros at the left of each one as padding and the length of all of them becomes the length of the largest one.

Loss, Evaluation, Optimizers

This is a multi-class classification problem, and cross entropy is the default loss function to use for multi-class classification problems which I eventually used. Finally, in order to choose the optimizer, I tried most of the available optimizers in Keras and from the results, which can be seen below, I chose the one with the best results. Adam optimizer was the one that gave us accuracy in the validation dataset right just below 80%.

Results

We can see below the learning curves of our model. I used 10 epochs but posed a trigger when two consecutive validation accuracy values are below the maximum, up to that point, value, then the procedure stops. The maximum value I got on the validation set is 78.6%. We can see it more clearly below:

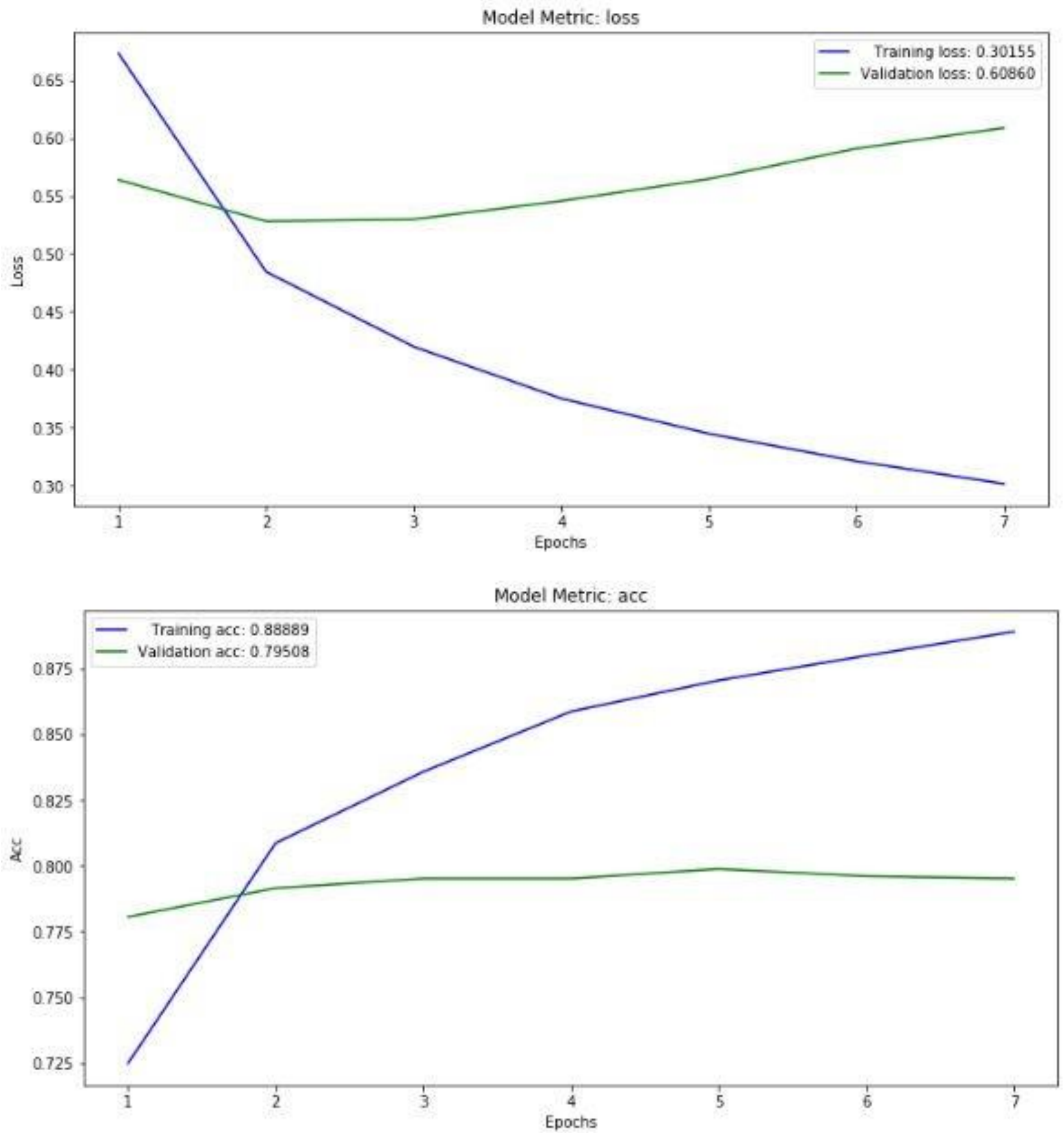


Figure 13 - Learning Curves of the RNN model

From epoch 5 onward I had the problem of overfitting which gives us lower correct predictions for our validation set in each consecutive epoch.

Limitations/Improvements/Future Work

If we decide that our model need further improvements, then the next step would be to manually review a sample of our validation losses and sort them according to what kinds of errors the model seems to be making. This should help us determine where the model has the most room for improvement. Then, by estimating how much effort it would take to improve each issue, we could make the best use of our time in achieving better results.
The overall accuracy of the best model is 78.6%. It’s not bad, but for sure there’s room for way more improvements:

- Using a high level NLP Library like Spacy could help a lot to perform feature extraction, because it has a huge English dictionary and corpus for stop words, words contractions, vocabulary and lemmatization.
- The distribution of positive/negative class categories of tweets is imbalanced...there are more negative tweets than positive ones. I should have done undersampling or oversampling.
- I would like to also try other state of the art techniques related to word embeddings like BERT and Glove.
- LSTM networks are prone to overfitting. Therefore, it requires a lot of work on model tuning. So, maybe it’s better to try another architecture.
- Try also to apply ensemble models.

Sell the Model to Business and Monitor the Progress

This solution is based on Keras and Tensorflow to use as the backend in order to implement a sentiment analysis neural network from scratch. So, it will lack most of the fine polishing of all the existing commercially available solutions.

After providing the model to the client, the company can create a client that will consume in the form of a stream, all the tweets that refer to it or its opponents, through the Twitter API. After getting the tweets by using the Api’s, the company can build a pipeline that will receive it, perform some ETL process, feed it through our model and finally perform some action according to the sentiment category the tweet has fallen into.

It’s very difficult to know if our model is good enough to go into production. I think that we would have to agree with the stakeholders on what metrics to use to assess the performance of the model and a proper threshold. The model typically goes through 5 stages and its performance will expectedly decrease at each stage: Training, Validation, Testing, Deployment and Production.

So, in production it is important to monitor the model’s performance continuously and have a threshold of cut off for when we need to train again and replace the model with the new one. Also, we can implement online learning and, in that case, we have to schedule a batch retraining and learn continuously with new data.

So, the key point is that we need to define what is good enough at each stage and continuously re-evaluate. It can be by having a clear agreement on KPIs required - accuracy, f1, response latency. We can also have different test datasets, like one for model development; another for model testing; and another for testing.

BONUS: Brief description of an approach to identify the reasons behind the tweet’s sentiments

This problem is kinda similar to what was done already before. So, to solve this, I would pretty much do the same data cleaning, processing and text representation. There are 10 different negative reasons labelled in the dataset. I would probably combine similar reasons into fewer categories like:

- Late & cancelled fight or;
- Lost & damaged luggage

Then I would split the negative tweets into training data (70%) and test data (30%). Proceed to train a multi-class logistic regression model to predict the reason for the negative tweets. Finally, apply this model to the test data: