Assignment 3 Report-Team 5

Introduction:

This assignment is going to use 3 types of models: BOW model, Word embeddings (GLOVE), and RNN to detect the sentiment of text datasets. Our team focusing on the earning call transcript of Microsoft in Q4 2018.

Experiment 1:

1. Mix all the paragraphs and split the data to a 80-20 split

We parsed the transcript in JSON format and manually input the "sentiment" for each sentence as "positive", "negative" or "neutral":


```
"text": {
       "1": "Greetings and welcome to the Microsoft Fiscal Year 2018
Third Quarter Earnings Conference Call. As a reminder, this
conference is being recorded. It is now my pleasure to introduce your
host, Mike Spencer, General Manager of Investor Relations. Thank you.
You may begin.",
       "2": "Good afternoon and thank you for joining us today. On
the call with me are Satya Nadella, Chief Executive Officer; Amy
Hood, Chief Financial Officer; Frank Brod, Chief Accounting Officer;
and Carolyn Frantz, Deputy General Counsel and Corporate
Secretary."},
"sentiment": {
       "1": "neutral",
       "2": "neutral",
}
```


We categorized the label as positive - 1, negative - 0, neutral - 2:

```
In [2]: json_file = open('Team5_Microsoft.json')
        json_str = json_file.read()
        data = json.loads(json_str)

        texts = list(data['text'].values())
        labels = list(data['sentiment'].values())
        labels = keras.utils.to_categorical(labels, 3)
        print(labels)
```

```
[[ 1.  0.  0.]
 [ 1.  0.  0.]
 [ 1.  0.  0.]
 [ 1.  0.  0.]
 [ 1.  0.  0.]
 [ 1.  0.  0.]
 [ 1.  0.  0.]
 [ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  1.  0.]
 [ 0.  1.  0.]
 [ 0.  1.  0.]
 [ 0.  1.  0.]
 [ 0.  1.  0.]
 [ 0.  1.  0.]
 [ 0.  1.  0.]
 [ 0.  1.  0.]
 [ 0.  1.  0.]
 [ 0.  1.  0.]
 [ 0.  1.  0.]
```

And then, tokenized the input data:

```
maxlen = 100  # We will cut reviews after 100 words
training_samples = 200  # We will be training on 200 samples
validation_samples = 10000  # We will be validating on 10000 samples
max_words = 10000  # We will only consider the top 10,000 words in the dataset

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
```

```
Found 1689 unique tokens.
```

```
data = pad_sequences(sequences, maxlen=maxlen)
label = np.asarray(labels)
print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', label.shape)
```

```
Shape of data tensor: (178, 100)
Shape of label tensor: (178, 3)
```

Finally, split our data to a 80-20 split as training set and test set:

```
X_train, X_test, y_train, y_test = train_test_split(data, label, test_size = 0.2)
```

```
print('Shape of X_train: ', X_train.shape)
print('Shape of X_test: ', X_test.shape)
print('Shape of y_train: ', y_train.shape)
print('Shape of y_test: ', y_test.shape)
```

```
Shape of X_train:  (142, 100)
Shape of X_test:  (36, 100)
Shape of y_train:  (142, 3)
Shape of y_test:  (36, 3)
```

2. Build the 3 models listed earlier and compute the confusion matrix for the training and testing datasets.

1) BOW model

Rather than a learning model, the team interpretes this model as a pre-processing approach. Before getting started, the team built a program to walk it through.

```
def word_extraction(sentence):
    ignore = ['a', "the", "is"]
    words = re.sub("[^\w]", " ",  sentence).split()
    cleaned_text = [w.lower() for w in words if w not in ignore]
    return cleaned_text
```

```
def tokenize(sentences):
    words = []
    for sentence in sentences:
        w = word_extraction(sentence)
        words.extend(w)

    words = sorted(list(set(words)))
    return words
```

```
import numpy as np
def generate_bow(allsentences):
    vocab = tokenize(allsentences)
    print("Word List for Document \n{0} \n".format(vocab));

    for sentence in allsentences:
        words = word_extraction(sentence)
        bag_vector = np.zeros(len(vocab))
        for w in words:
            for i,word in enumerate(vocab):
                if word == w:
                    bag_vector[i] += 1
        print("{0}\n{1}\n".format(sentence,np.array(bag_vector)))

    return vocab
```

Three steps it takes, whom are, extracting the words from sentences, tokenizing every one of them, and finally generate a unique vocabulary to the text to be analyzed.
By these steps, the function gives us a bunch of vector with the same length. In those vectors each element strands for a frequency that each word may appear.

```
Yeah. And I would say, Mark, while I apprec:
[0. 0. 0. ... 1. 0. 2.]

If you think about gross margins in particul
[0. 0. 0. ... 0. 0. 0.]
```

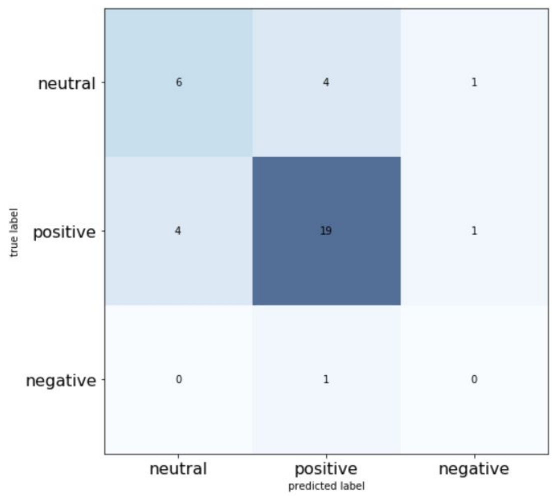
Fortunately sklearn has provided a built in method doing BOW algorithms. By calling:

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()
X = vectorizer.fit_transform(X_raw)

print(X.toarray())
```

That can we easily access this model.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 256)               421632
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 771
=================================================================
Total params: 422,403
Trainable params: 422,403
```

```
Non-trainable params: 0
```
Running this neural network after a procedure of BOW, it gives a matrix like



2) Word embeddings (GLOVE)

In this experiment we used glove word embeddings from 2014 English Wikipedia.

a. Preprocess the text

We load the JSON file and decode the data into two lists: text and label. The maximum length of each paragraph is set to 200 so the extra words need to be cut and those less than 200 should be added 0 to 200. In this case we consider top 2000 words in dataset.

b. Build embedding

We choose the 100D file and load the embedding words into vector. So the train data will be compressed into 100 dimensions vector.

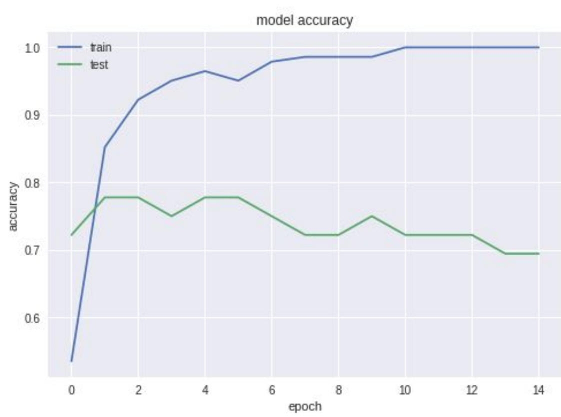
```
Found 400000 word vectors.
```

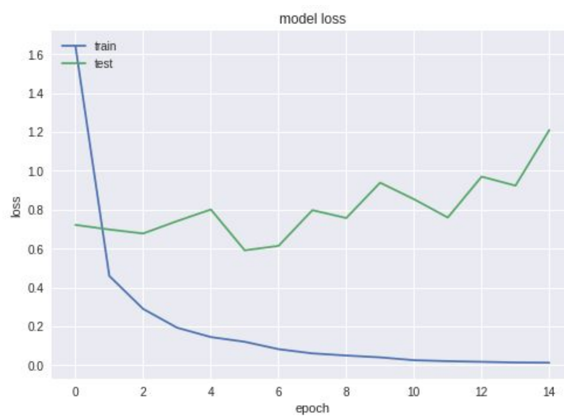
Then build the embedding matrix.

c. Train models

We trained two models in this part. This is the first one:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_5 (Embedding)      (None, 200, 100)          1000000
_________________________________________________________________
flatten_5 (Flatten)          (None, 20000)             0
_________________________________________________________________
dense_9 (Dense)              (None, 32)                640032
_________________________________________________________________
dense_10 (Dense)             (None, 3)                 99
=================================================================
Total params: 1,640,131
Trainable params: 1,640,131
Non-trainable params: 0
_________________________________________________________________
```

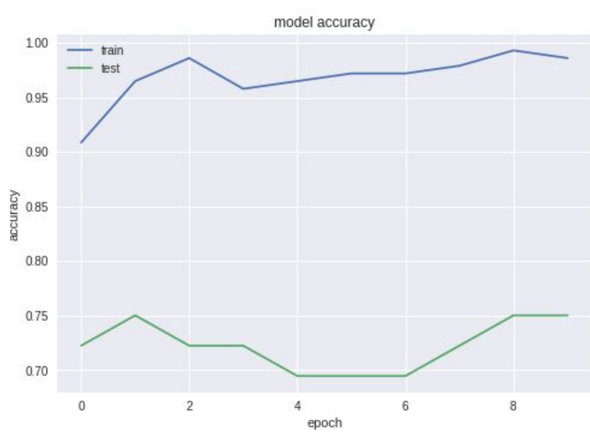


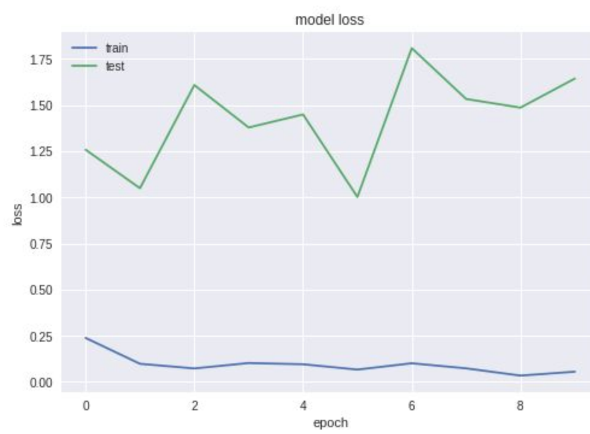


Test loss: 1.2100759877098932
Test accuracy: 0.6944444444444444

Though the training accuracy is almost 1, the testing result keeps going down. In the second one we add a dropout layer:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_8 (Embedding)      (None, 200, 100)          1000000
_________________________________________________________________
flatten_8 (Flatten)          (None, 20000)             0
_________________________________________________________________
dense_14 (Dense)             (None, 32)                640032
_________________________________________________________________
dropout_2 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_15 (Dense)             (None, 3)                 99
=================================================================
Total params: 1,640,131
Trainable params: 1,640,131
Non-trainable params: 0
_________________________________________________________________
```





Test loss: 1.6436499423450894
Test accuracy: 0.75

From the result dropout works a little better.

3) RNN
We use SimpleRNN from Keras to build our base model as follow:

```
max_features = 10000
model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(32))
model.add(Dense(3, activation='sigmoid'))
model.summary()
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_1 (Embedding)      (None, None, 32)          320000    
_________________________________________________________________
simple_rnn_1 (SimpleRNN)     (None, 32)                2080      
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 99        
=================================================================
Total params: 322,179
Trainable params: 322,179
Non-trainable params: 0
_________________________________________________________________
```

As the result:

```
history = model.fit(X_train, y_train,
                    epochs=30,
                    batch_size=32)
```

```
Epoch 21/30
142/142 [==============================] - 0s 1ms/step - loss: 0.1279 - acc: 0.9906
Epoch 22/30
142/142 [==============================] - 0s 1ms/step - loss: 0.1340 - acc: 0.9836
Epoch 23/30
142/142 [==============================] - 0s 1ms/step - loss: 0.1024 - acc: 0.9953
Epoch 24/30
142/142 [==============================] - 0s 1ms/step - loss: 0.1117 - acc: 0.9812
Epoch 25/30
142/142 [==============================] - 0s 1ms/step - loss: 0.0981 - acc: 0.9906
Epoch 26/30
142/142 [==============================] - 0s 1ms/step - loss: 0.0860 - acc: 0.9977
Epoch 27/30
142/142 [==============================] - 0s 1ms/step - loss: 0.0802 - acc: 1.0000
Epoch 28/30
142/142 [==============================] - 0s 1ms/step - loss: 0.0742 - acc: 0.9930
Epoch 29/30
142/142 [==============================] - 0s 1ms/step - loss: 0.0911 - acc: 0.9883
Epoch 30/30
142/142 [==============================] - 0s 1ms/step - loss: 0.0659 - acc: 1.0000
```


```
test_loss, test_score = model.evaluate(X_test, y_test, batch_size=32)
print("Loss on test set: ", test_loss)
print("Accuracy on test set: ", test_score)
```

```
36/36 [==============================] - 0s 2ms/step
Loss on test set:  0.587709943453
Accuracy on test set:  0.694444444444
```

As the result showing above, we have a 100% accuracy for training set, and nearly 70% accuracy for testing. And the confusion matrix also shows a high performance of the RNN model. However, because the negative sentiment is too few to test, we are not sure about the accuracy of negative sentiment.

Experiment 2: Transfer Learning

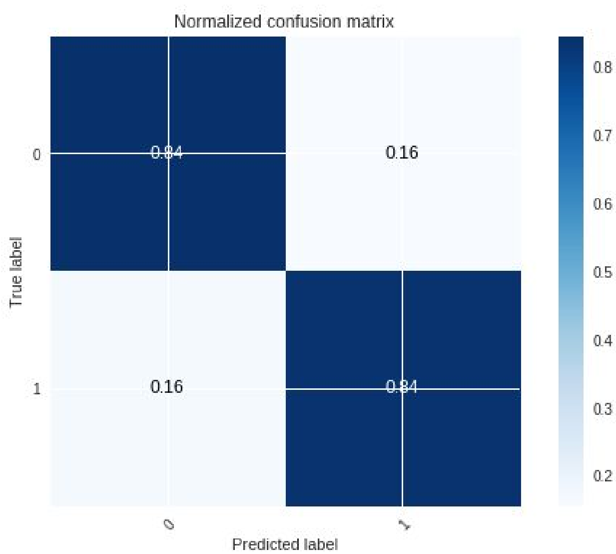
1. BOW model

We can still transfer the knowledge of vocabulary that BOW model learned from other scences to a similar context. To expand the vocabulary, we introduced IMBD dataset.

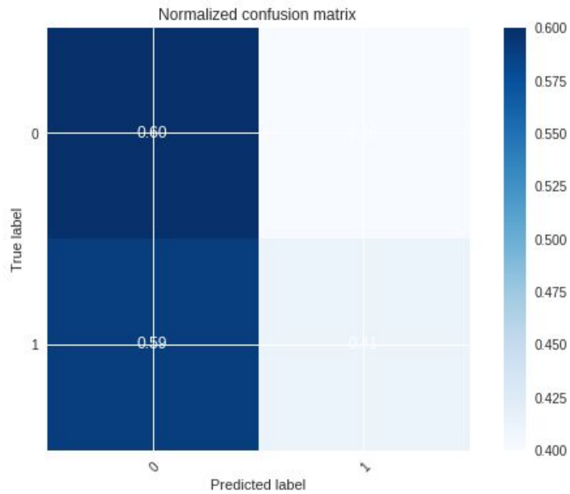
```
import keras
word_to_id = keras.datasets.imdb.get_word_index()
```

```
import collections
def swap_dictionary(original_dict):
    temp_dict = {}
    dict_list = original_dict.items()
    for i in dict_list:
        temp_dict[i[1]] = i[0]
        temp_dict[0] = 'unbekanntwort'
    return temp_dict

swapped = swap_dictionary(word_to_id)
print(swapped)
```

The training on imdb dataset gives:



And after transferring this model, it yields:



Which is not as good a performance as it was on the original dataset, but clearly there is a progress in comparison with the small vocabulary.

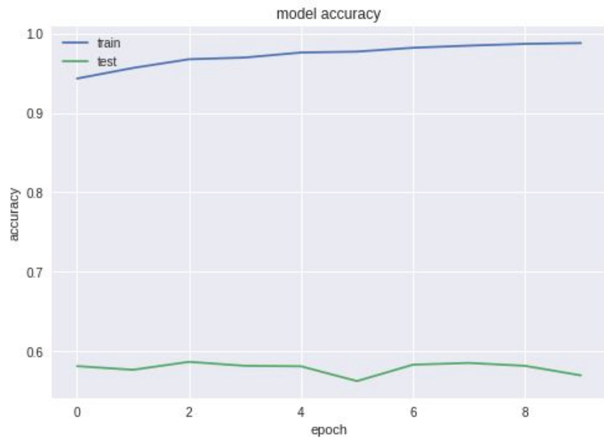
2. Word embeddings (GLOVE)

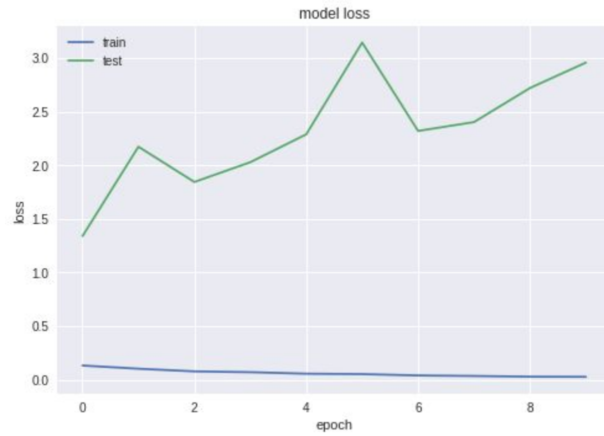
Use the same way to implement glove word embeddings to IMBD reviews raw data.

```
Found 88582 unique tokens.
Shape of data tensor: (25000, 200)
Shape of label tensor: (25000,)
(20000, 200) (5000, 200)
```

Use the first model which was used for our own data before to train IMBD:

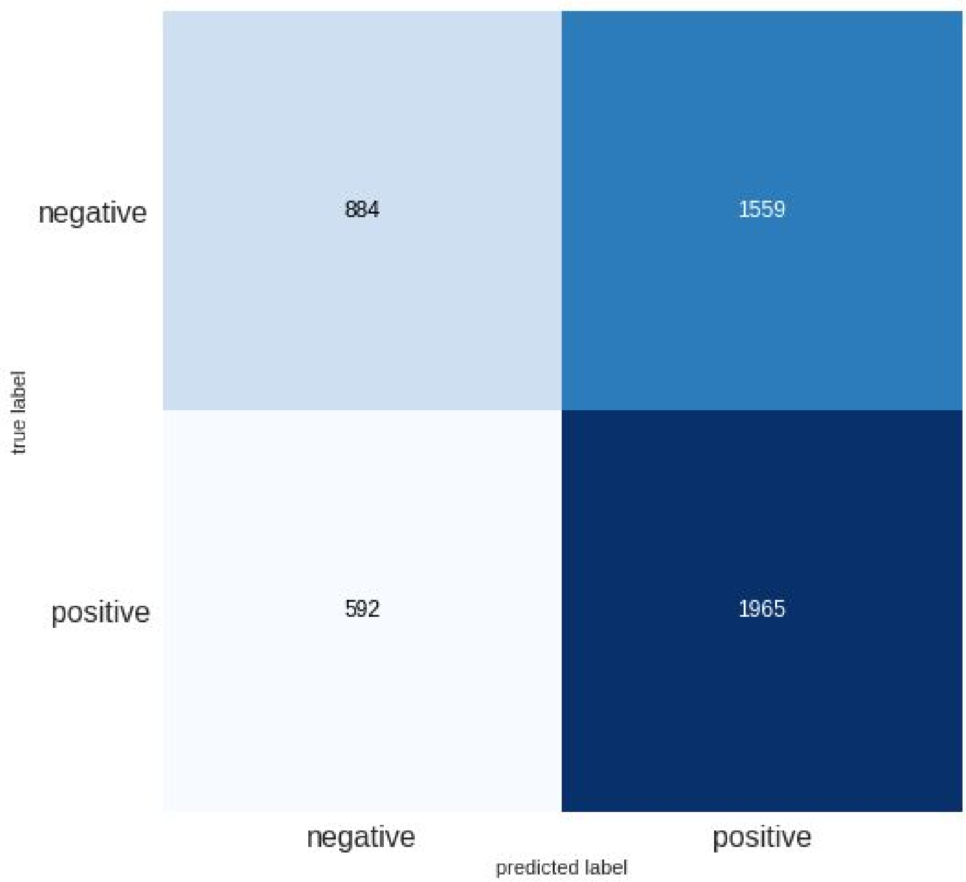
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_4 (Embedding)      (None, 200, 100)          1000000
_________________________________________________________________
flatten_4 (Flatten)          (None, 20000)             0
_________________________________________________________________
dense_7 (Dense)              (None, 32)                640032
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 33
=================================================================
Total params: 1,640,065
Trainable params: 1,640,065
Non-trainable params: 0
_________________________________________________________________
```



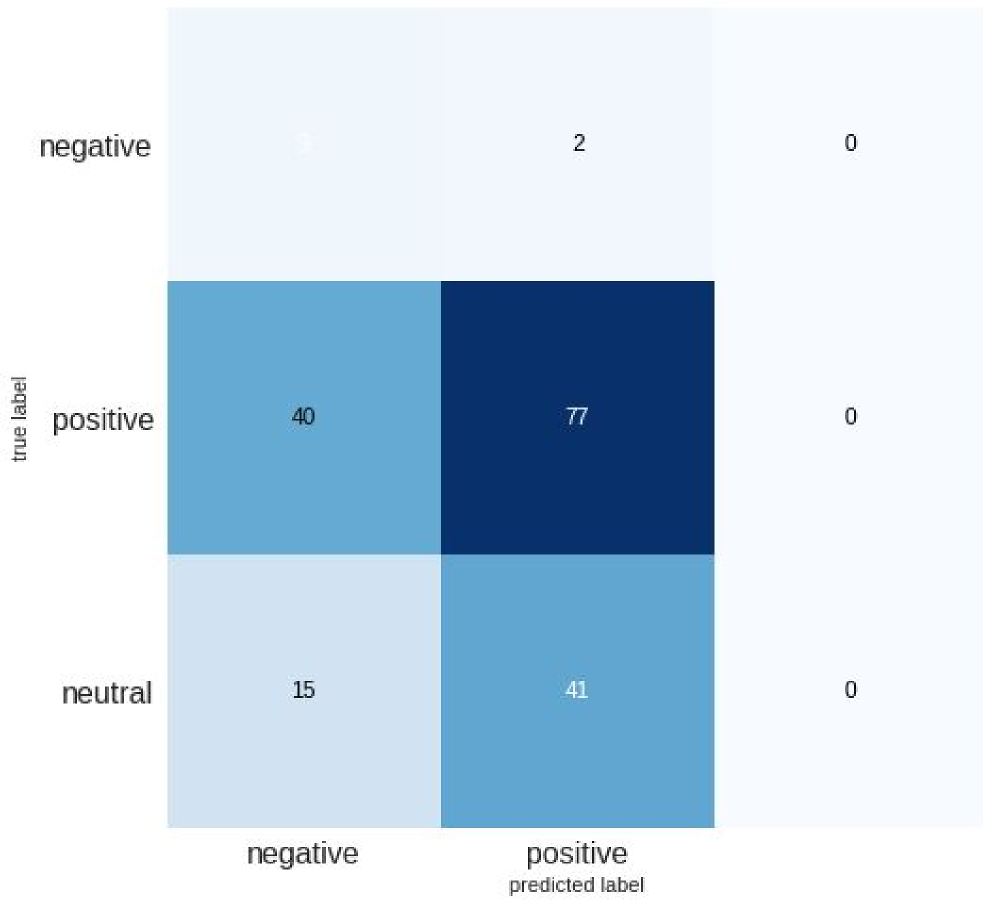


Test loss: 2.9573881818771364
Test accuracy: 0.5698

The confusion matrix of IMBD:



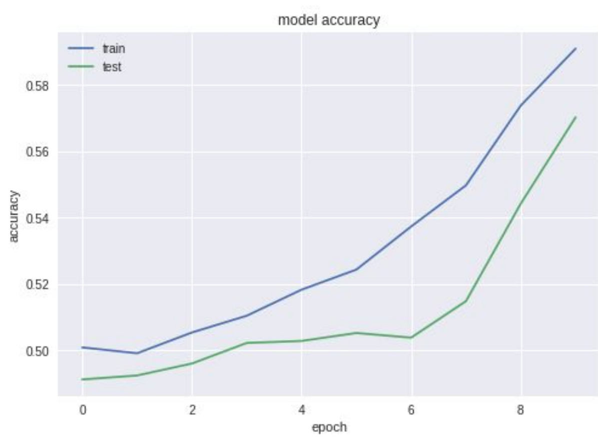
Then predict our financial data with this pretrained model, we get the confusion matrix as following:

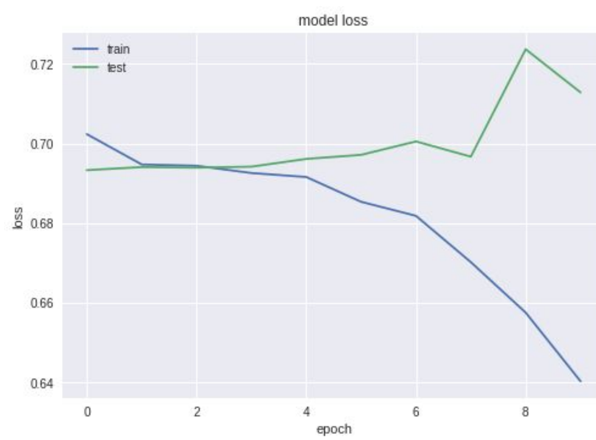


The result is acceptable, and it can be seen that most of the neutral data is considered as positive by transfer learning.

Train the second model:

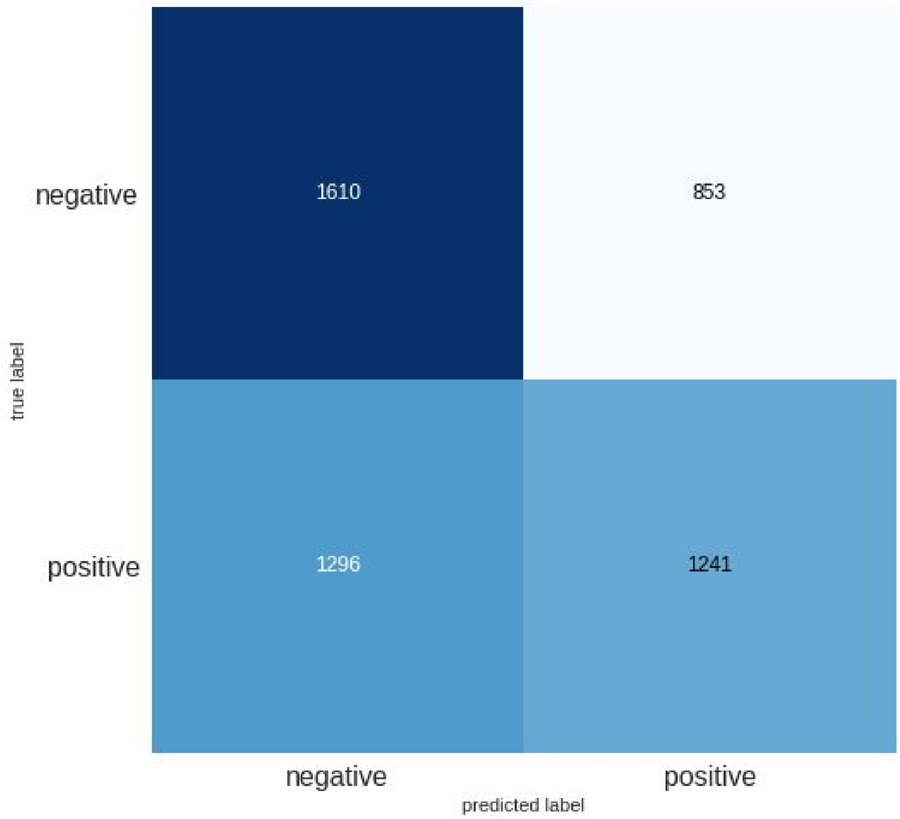
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_10 (Embedding)     (None, 200, 100)          1000000   
_________________________________________________________________
flatten_10 (Flatten)         (None, 20000)             0         
_________________________________________________________________
dense_19 (Dense)             (None, 16)                320016    
_________________________________________________________________
dropout_5 (Dropout)          (None, 16)                0         
_________________________________________________________________
dense_20 (Dense)             (None, 1)                 17        
=================================================================
Total params: 1,320,033
Trainable params: 1,320,033
Non-trainable params: 0
_________________________________________________________________
```

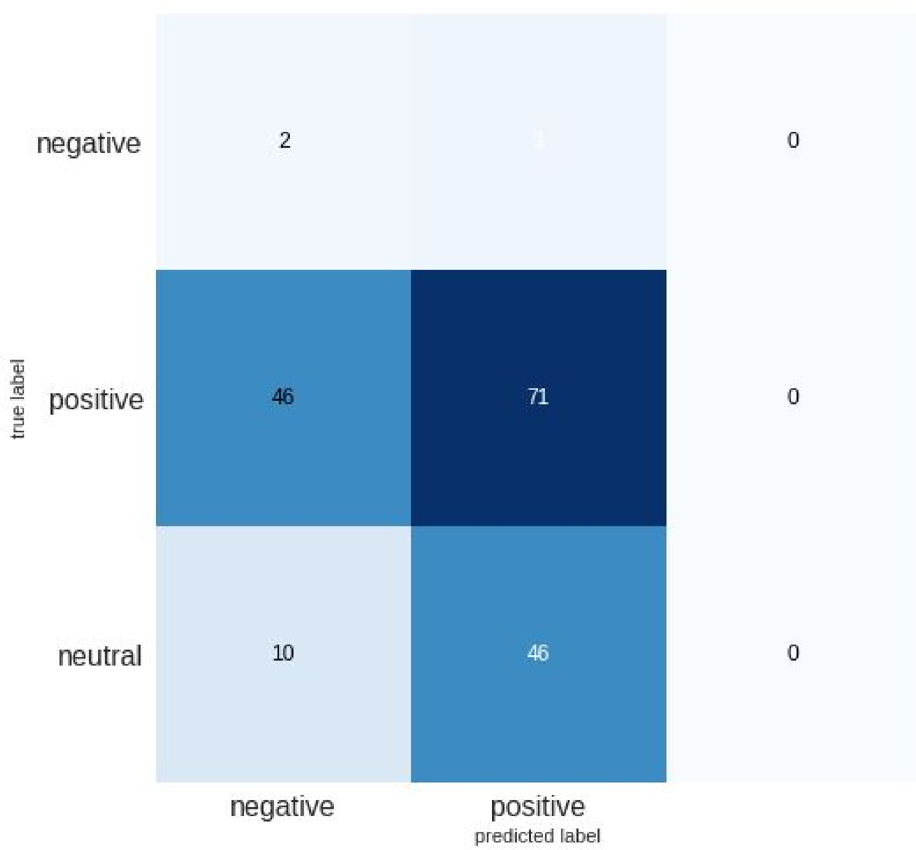




Test loss: 0.7128520246505737
Test accuracy: 0.5702

For IMBD dataset, the training result with adding dropout is not better than the simple one which is contrary to experiment 1. Confusion matrices are showed below:


negative
positive
true label
1610
853
1296
1241
negative
positive
predicted label


negative
positive
neutral
true label
2
0
46
71
0
10
46
0
negative
positive
predicted label

3. RNN

1. Build the 3 models listed earlier using the Movie review dataset [6] or from Keras.

Compute the confusion matrix for each model

Retrieving the IMDB review dataset by:

```
max_features = 10000  # number of words to consider as features
maxlen = 500  # cut texts after this number of words (among top max_features most common words)
batch_size = 32

print('Loading data...')
(input_train, y_train), (input_test, y_test) = imdb.load_data(num_words=max_features)
print(len(input_train), 'train sequences')
print(len(input_test), 'test sequences')

print('Pad sequences (samples x time)')
input_train = sequence.pad_sequences(input_train, maxlen=maxlen)
input_test = sequence.pad_sequences(input_test, maxlen=maxlen)
print('input_train shape:', input_train.shape)
print('input_test shape:', input_test.shape)
```

```
Loading data...
25000 train sequences
25000 test sequences
Pad sequences (samples x time)
input_train shape: (25000, 500)
input_test shape: (25000, 500)
```

Deployed the same RNN model as experiment 1:

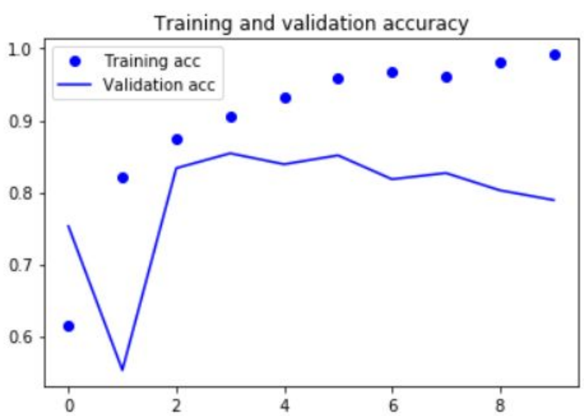
```
max_features = 10000
model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(32))
model.add(Dense(3, activation='sigmoid'))
model.summary()
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
```

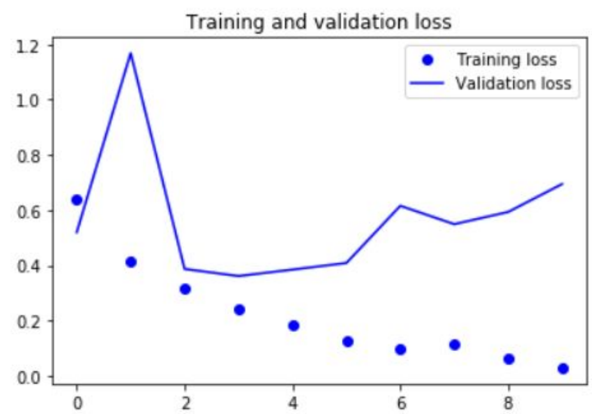
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, None, 32)          320000
_________________________________________________________________
simple_rnn_1 (SimpleRNN)     (None, 32)                2080
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 99
=================================================================
Total params: 322,179
Trainable params: 322,179
Non-trainable params: 0

_________________________________________________________________
```

As result

```
history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

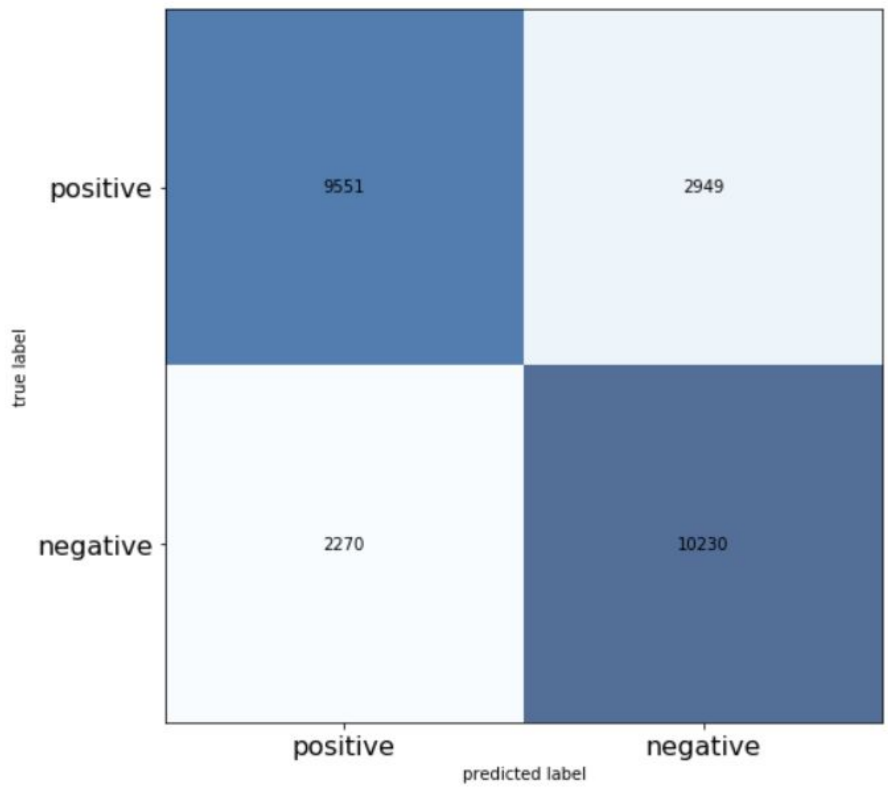
```
Epoch 1/10
20000/20000 [==============================] - 28s 1ms/step - loss: 0.6409 - acc: 0.6154 - val_loss: 0.5197 - val_acc: 0.7528
Epoch 2/10
20000/20000 [==============================] - 27s 1ms/step - loss: 0.4168 - acc: 0.8216 - val_loss: 1.1688 - val_acc: 0.5528
Epoch 3/10
20000/20000 [==============================] - 28s 1ms/step - loss: 0.3144 - acc: 0.8736 - val_loss: 0.3865 - val_acc: 0.8336
Epoch 4/10
20000/20000 [==============================] - 28s 1ms/step - loss: 0.2408 - acc: 0.9061 - val_loss: 0.3609 - val_acc: 0.8542
Epoch 5/10
20000/20000 [==============================] - 28s 1ms/step - loss: 0.1812 - acc: 0.9324 - val_loss: 0.3842 - val_acc: 0.8390
Epoch 6/10
20000/20000 [==============================] - 29s 1ms/step - loss: 0.1236 - acc: 0.9579 - val_loss: 0.4081 - val_acc: 0.8514
Epoch 7/10
20000/20000 [==============================] - 30s 1ms/step - loss: 0.0944 - acc: 0.9673 - val_loss: 0.6154 - val_acc: 0.8182
Epoch 8/10
20000/20000 [==============================] - 30s 1ms/step - loss: 0.1149 - acc: 0.9601 - val_loss: 0.5489 - val_acc: 0.8268
Epoch 9/10
20000/20000 [==============================] - 30s 1ms/step - loss: 0.0631 - acc: 0.9807 - val_loss: 0.5931 - val_acc: 0.8028
Epoch 10/10
20000/20000 [==============================] - 29s 1ms/step - loss: 0.0284 - acc: 0.9916 - val_loss: 0.6944 - val_acc: 0.7892
```





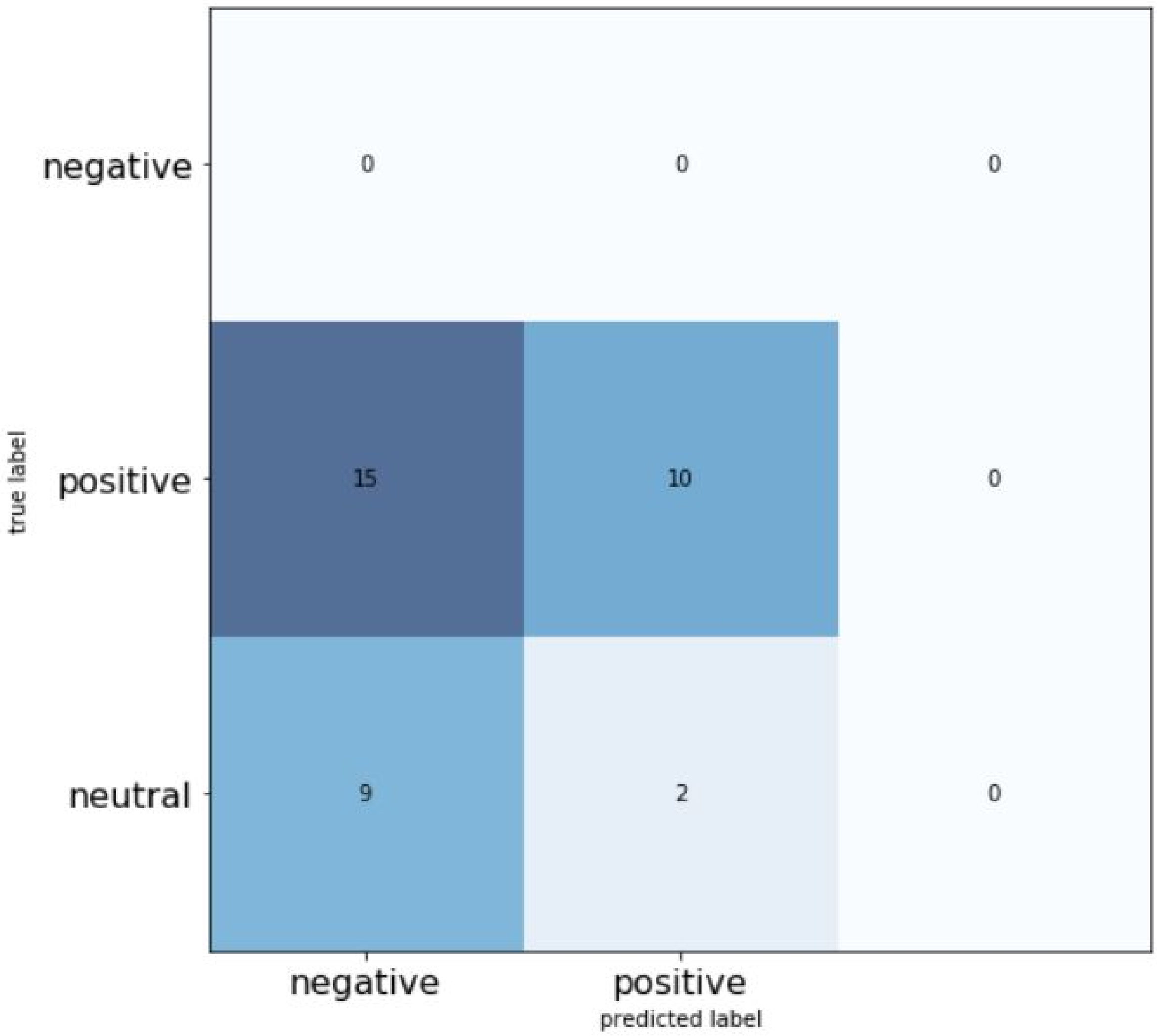

```
test_loss, test_score = model.evaluate(input_test, y_test, batch_size=32)
print("Loss on test set: ", test_loss)
print("Accuracy on test set: ", test_score)
```

```
25000/25000 [==============================] - 28s 1ms/step
Loss on test set:  0.675347436123
Accuracy on test set:  0.79124
```



The result looks great for IMDB dataset, we got a higher test accuracy as 79.12%. However, there is a little bit overfitting and the validation accuracy and loss are not as good as training data.

2. Use the 3 models to predict the sentiments for the financial dataset. And compute the confusion matrices for each model.



Because the IMDB dataset only has positive and negative output so we redesigned our confusion matrix, we determined the neutral sentiment in our dataset to fit as positive or negative. Therefore, the performance of this model is not good as experiment 1.

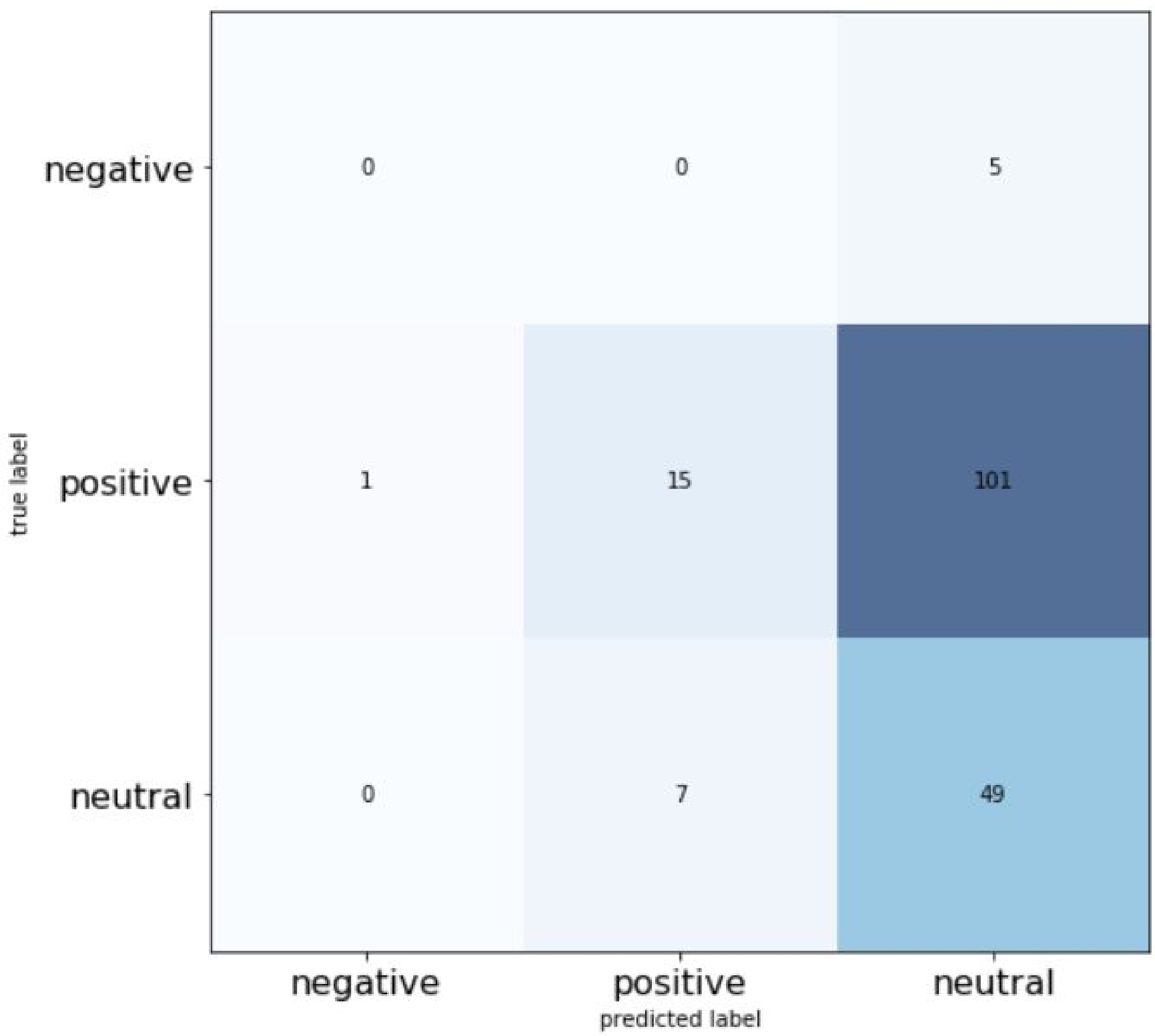
Experiment 3: Using APIs
1. Using the Amazon, Google, Microsoft and Watson APIs [2,3,4,5], obtain the sentiment scores for your entire dataset.
2. Normalize the scores and take the average normalized score to determine the sentiments
3. Compute the confusion matrix wrt the original dataset and discuss your results.

1) Amazon Comprehend:
Using Amazon Comprehend to analyze sentiment with “one document per line”, and gives us the output in JSON as shown below:


```
{
        "File": "AWS_Microsoft.json",
```

```
"Line": 0,
"Sentiment": "NEUTRAL",
"SentimentScore": {
    "Mixed": 0.010904182679951191,
    "Negative": 0.03218216076493263,
    "Neutral": 0.6769425272941589,
    "Positive": 0.279971182346344
}
```


There are 4 scores in “SentimentScore”: “Mixed”, “Negative”, “Neutral”, and “Positive”, which means there will be 4 kinds of sentiment outputs. The Amazon Comprehend gives a final result as “Sentiment” part by choosing the highest score of these 4.

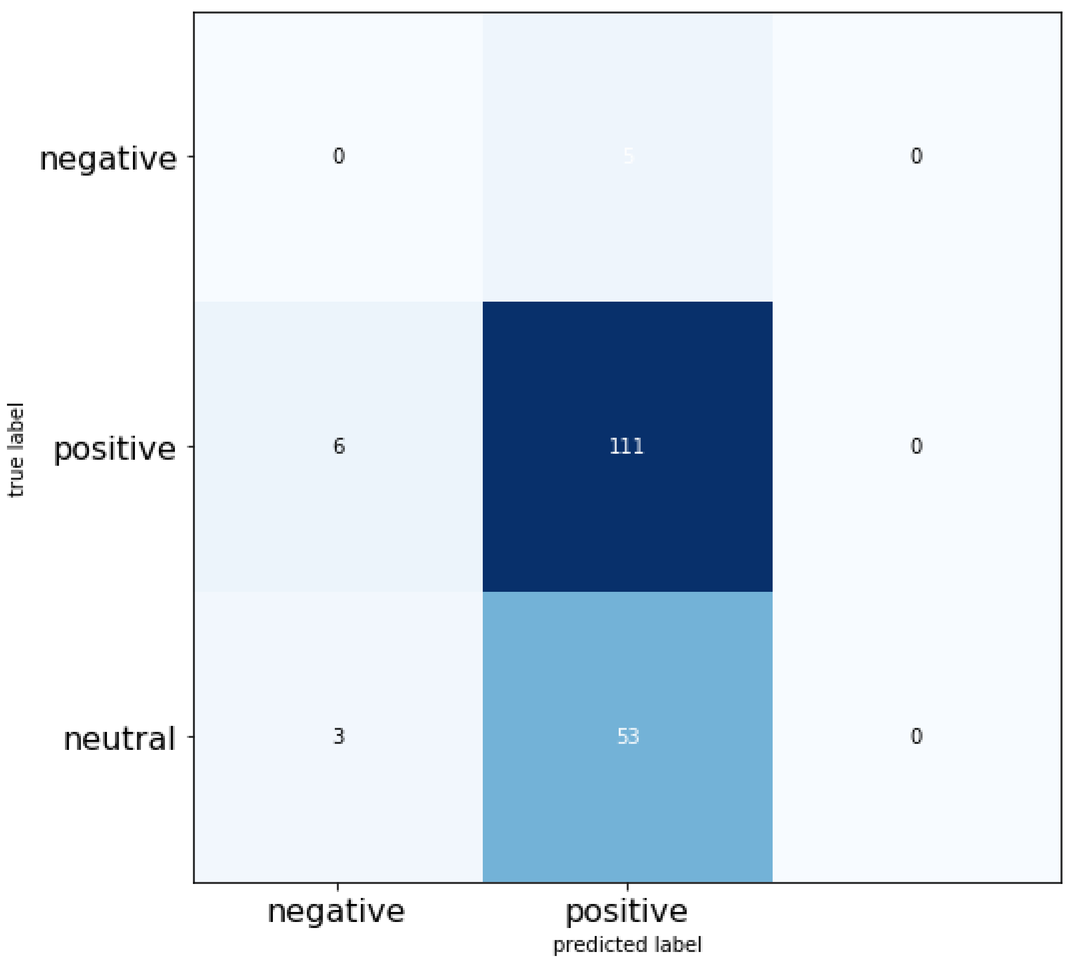


The result is not as good as our RNN model since most of positive sentiments are treated as neutral in Amazon Comprehend. However, this might partly due to our own classification error by reading and feeling. Also, Amazon Comprehend chooses the highest score of 4 output types as the final result which might be better if we normalized the score for 3 output types.

2) Google cloud natural language API

Google API accepts string input and returns sentiment object with sentiment score and magnitude. We use 0 as threshold in this case and get an pretty accurate result:

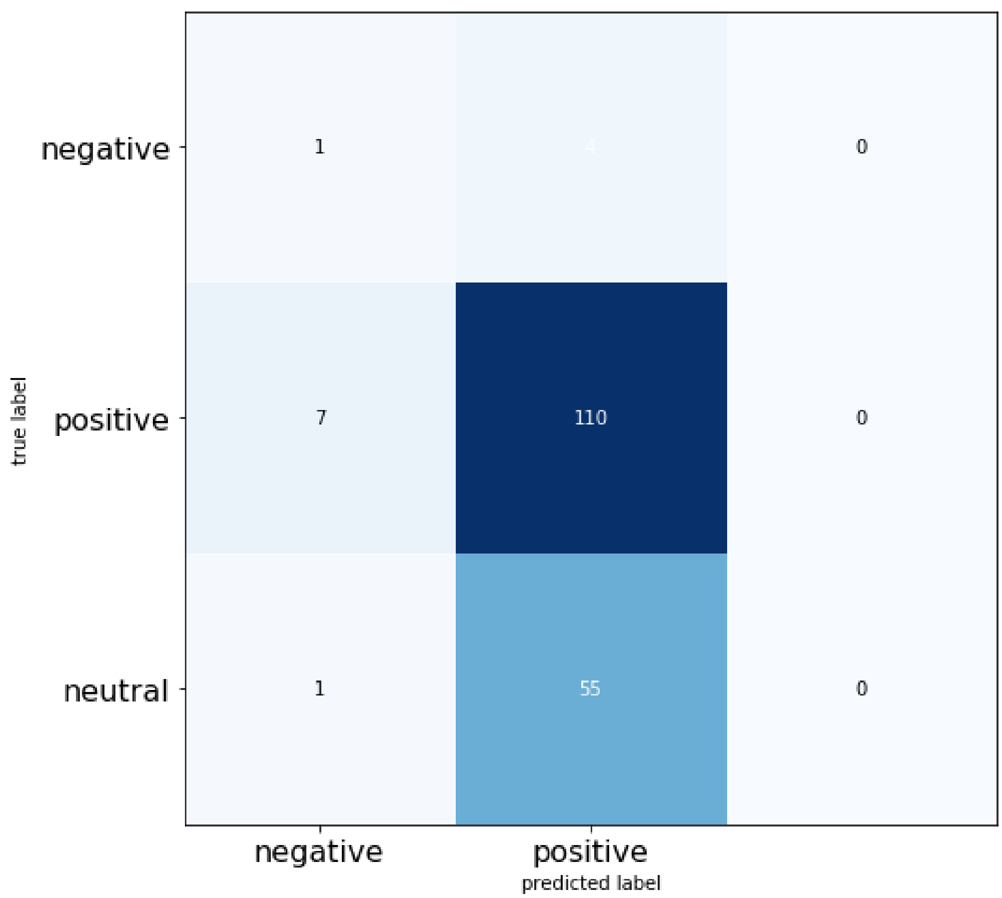
```
[magnitude: 1.2000000476837158
score: 0.20000000298023224
, magnitude: 0.800000011920929
score: 0.30000001192092896
, magnitude: 0.30000001192092896
score: 0.30000001192092896
, magnitude: 0.30000001192092896
, magnitude: 1.0
score: 0.20000000298023224
, magnitude: 0.6000000238418579
, , magnitude: 0.20000000298023224
score: -0.10000000149011612
, magnitude: 2.0
```



3) Microsoft text analytics API

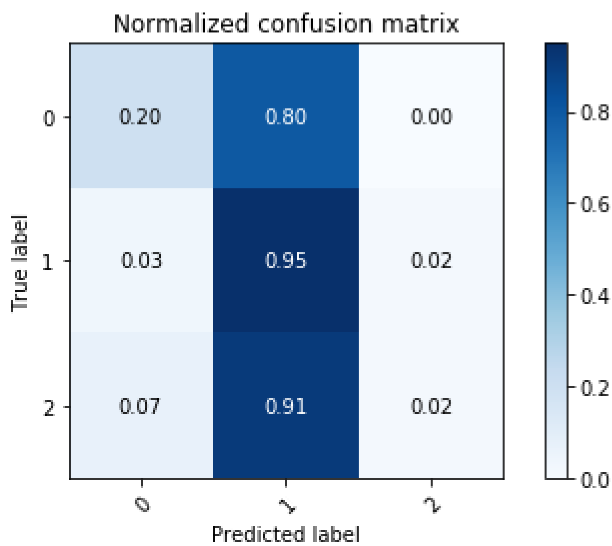
The microsoft web API accepts a special format json text and returns a json sentiment score. From the result it performs as good as google API.


```
{'documents': [{'id': '1', 'score': 0.985567569732666},
               {'id': '2', 'score': 0.863558232784271 2},
               {'id': '3', 'score': 0.5},
               {'id': '4', 'score': 0.5},
               {'id': '5', 'score': 0.7688121795654297},
               {'id': '6', 'score': 0.8619561791419983},
               {'id': '7', 'score': 0.5},
               {'id': '8', 'score': 0.5},
               {'id': '9', 'score': 0.9329003095626831},
               {'id': '10', 'score': 0.9553788900375366},
               {'id': '11', 'score': 0.9167450666427612},
               {'id': '12', 'score': 0.5},
               {'id': '13', 'score': 0.5},
               {'id': '14', 'score': 0.8877207040786743},
               {'id': '15', 'score': 0.5},
```

4) Watson API from IBM
This API provides a free approach for language comprehending, with a limit of 50000 characters per response. And only when the scoring system succeed an exact zero value, it gives a neutral response, which makes the results seems a little bit unbalanced.

As for its performance…



Experiemnt 4: Ensemble learning using AutoML
1. Instead of simple averaging, you wonder if you could build a model that can map the raw(not normalized) results from the 4 APIs to the outputs you labeled. i.e.
a. Inputs: Amazon, Google, IBM, Microsoft scores
b. Outputs: Sentiment scores you labeled
2. Use TPOT, AutoSKLearn, H2O.ai’s APIs and choose the best model.
3. Discuss if this model was better than the metrics you got from experiments 3 (simple averaging)

1) H2OAutoML
Use H2OAutoML API to train several models as below. The running time is set to 5 minutes.

model_id	mean_per_class_error	logloss	rmse	mse
GBM_grid_1_AutoML_20190322_143401_model_2	0.517348	0.633213	0.460432	0.211998
GBM_grid_1_AutoML_20190322_143401_model_13	0.521775	0.646734	0.448842	0.20146
StackedEnsemble_BestOfFamily_AutoML_20190322_143401	0.532102	0.636645	0.441232	0.194685
GBM_grid_1_AutoML_20190322_143401_model_26	0.533679	0.621452	0.438389	0.192185
GBM_3_AutoML_20190322_143401	0.533934	0.63786	0.455318	0.207314
GBM_grid_1_AutoML_20190322_143401_model_16	0.535511	0.752672	0.469799	0.220711
GBM_grid_1_AutoML_20190322_143401_model_20	0.537291	0.603439	0.444596	0.197666
GBM_grid_1_AutoML_20190322_143401_model_27	0.537546	0.616314	0.449239	0.201816
DeepLearning_grid_1_AutoML_20190322_143401_model_2	0.539886	1.14747	0.466202	0.217345
DeepLearning_grid_1_AutoML_20190322_143401_model_1	0.541972	0.873071	0.483193	0.233476

```
MSE: 0.21199769675014563
RMSE: 0.46043207615254783
LogLoss: 0.6332131131502787
Mean Per-Class Error: 0.5173483923483925
Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
```

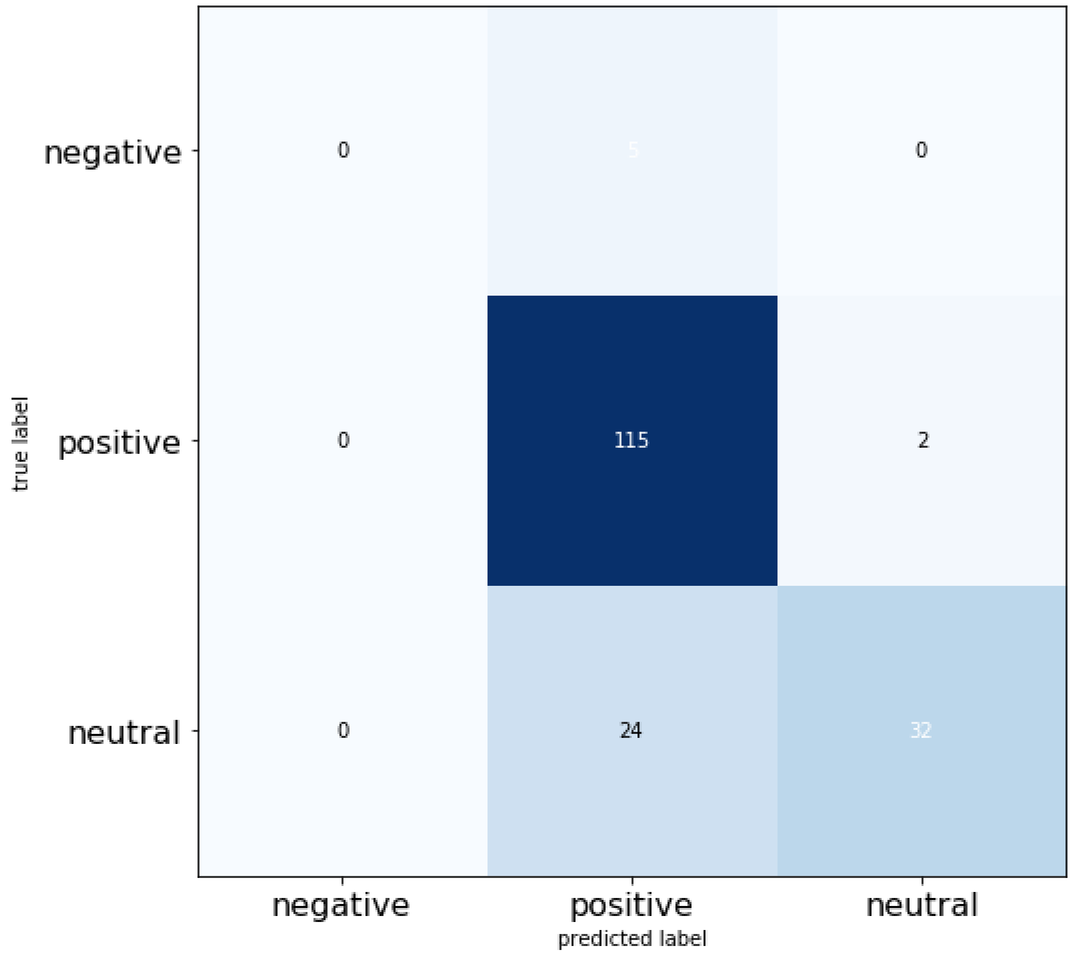
0	1	2	Error	Rate
0.0	5.0	0.0	1.0	5 / 5
0.0	113.0	4.0	0.0341880	4 / 117
0.0	29.0	27.0	0.5178571	29 / 56
0.0	147.0	31.0	0.2134831	38 / 178

Cross-Validation Metrics Summary:

	mean	sd	cv_1_valid	cv_2_valid	cv_3_valid	cv_4_valid	cv_5_valid
accuracy	0.7866667	0.0146144	0.7777778	0.8055556	0.75	0.8	0.8
err	0.2133333	0.0146144	0.2222222	0.1944444	0.25	0.2	0.2
err_count	7.6	0.5656855	8.0	7.0	9.0	7.0	7.0
logloss	0.6329293	0.0369553	0.6195024	0.5995686	0.7302462	0.6360946	0.5792344
max_per_class_error	0.8	0.1743939	1.0	1.0	1.0	0.4545455	0.5454546
mean_per_class_accuracy	0.6141919	0.1170883	0.4722222	0.540404	0.4333333	0.8207071	0.8042929
mean_per_class_error	0.3858081	0.1170883	0.5277778	0.4595959	0.5666667	0.1792929	0.1957071
mse	0.2118568	0.0174259	0.2088474	0.1942207	0.2575866	0.2117789	0.1868504
r2	0.1680170	0.0871463	0.2199822	0.3769555	0.0928472	0.0173139	0.1329860
rmse	0.4595378	0.0184629	0.4569982	0.4407047	0.5075299	0.4601944	0.4322619

Variable Importances:

variable	relative_importance	scaled_importance	percentage
google	104.9519196	1.0	0.3198447
ibm	90.4254150	0.8615890	0.2755746
amazon	71.4222641	0.6805237	0.2176619
microsoft	61.3344307	0.5844050	0.1869188



2) TPOT

```
Optimizati...  100% 120/120 [01:03<00:00, 1.77pipeline/s]
Generation 1 - Current best internal CV score: 0.7551815362160189
Generation 2 - Current best internal CV score: 0.7554278416347382
Generation 3 - Current best internal CV score: 0.7554278416347382
Generation 4 - Current best internal CV score: 0.7554278416347382
Generation 5 - Current best internal CV score: 0.7618482028826857

Best pipeline: GradientBoostingClassifier(input_matrix, learning_rate
0.0
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/classification
  score = y_true == y_pred
```

As an autoML tool, TPOT suggested a GBM classifier for this simple problem, with a better score than previous algorithms.

3) AutoSKLearn

```
pred = automl.predict(x_test)
print("Accuracy score", sklearn.metrics.accuracy_score(y_test, pred))
```

```
Accuracy score 0.75
```

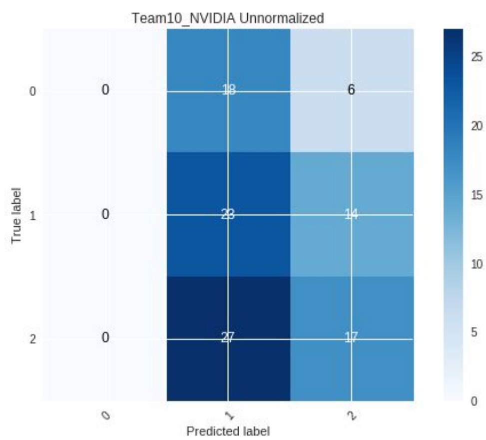
Final model:

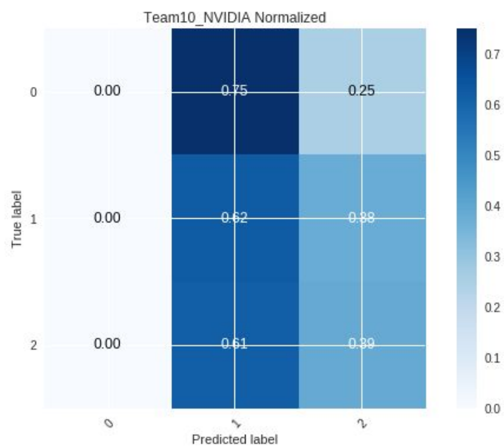
As a result, no model stands a better performance than the one using RNN.

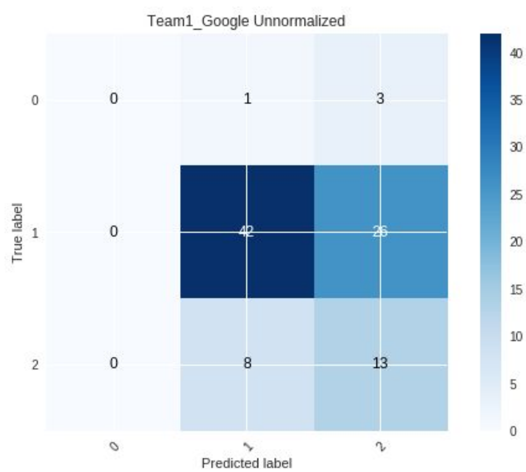
Final model:

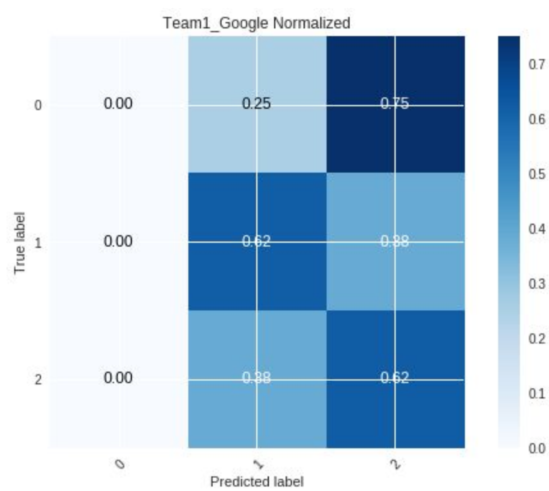
As a result, no model stands a better performance than the one using RNN.

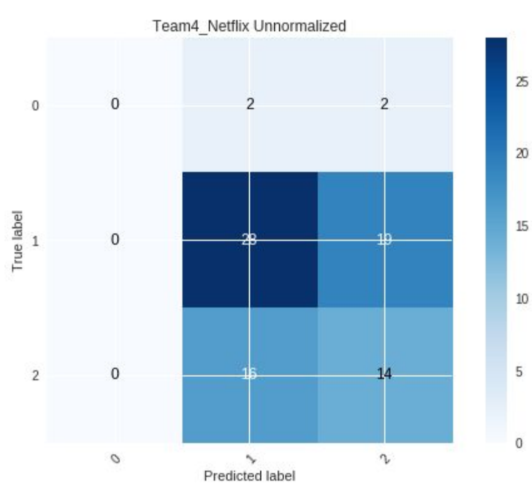
Here are some of the results transferred from the the pre-trained model..

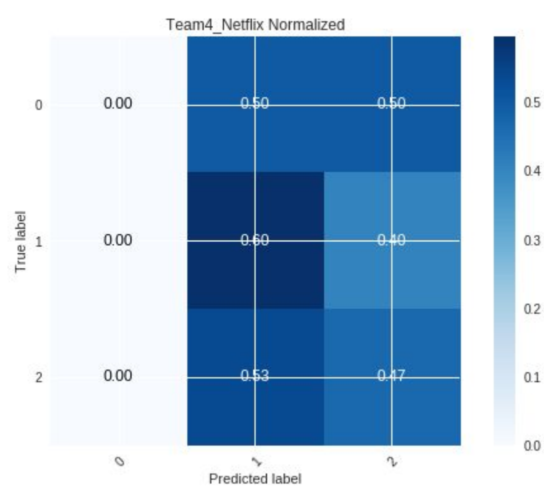




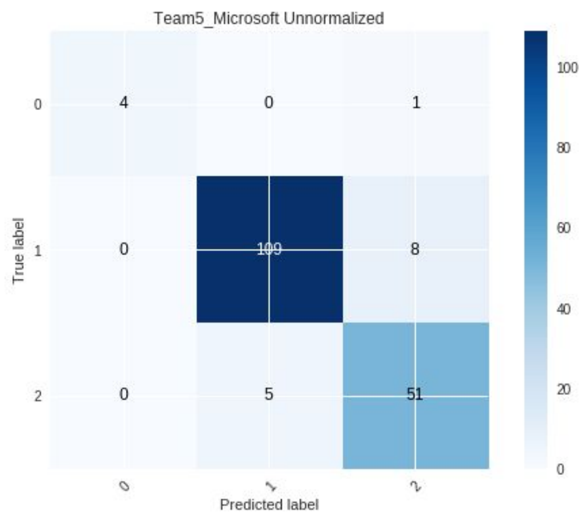


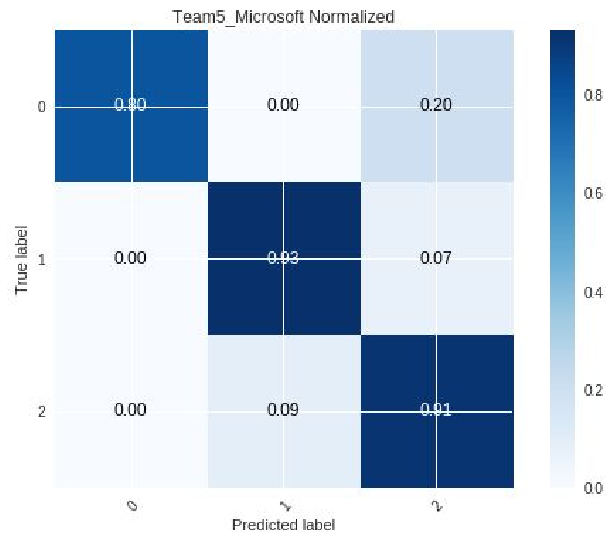






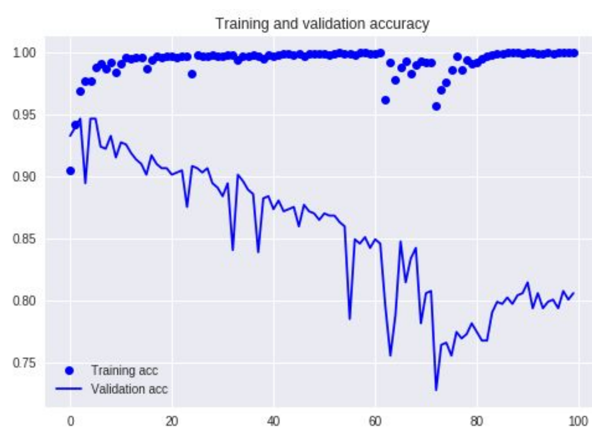
There are ups and downs in positive and negative predictions, but no neutral predictions are even made. In comparison with the original matrix…..





This is too small a dataset the model weighted on, thus further training may needed.

The team decided to concatenate all data available and analyze them altogether. And here is the learning curve of this baby RNN model.



Performance can be as good as:

```
240/240 [==============================] - 0s 424us/step
Loss on test set:  0.7727760295073192
Accuracy on test set:  0.7847222447395324
```