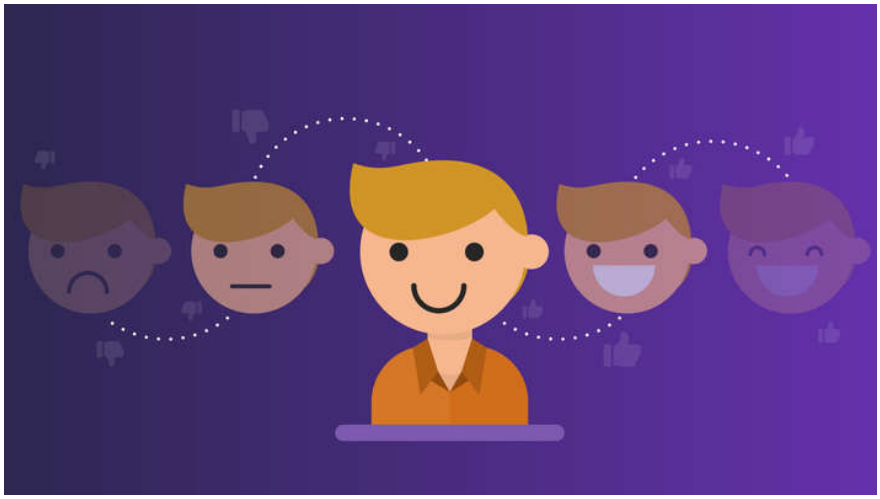
https://www.datasciencecentral.com/profiles/blogs/sentiment-analysis-with-naive-bayes-and-lstm

datasciencecentral.com

Sentiment Analysis with Naive Bayes and LSTM

Posted by Igor Bobriakov on February 4, 2020 at 11:00am Send Message View Blog

18-22 minutes



In this notebook, we try to predict the positive (label 1) or negative (label 0) sentiment of the sentence. We use the UCI Sentiment Labelled Sentences Data Set.

Sentiment analysis is very useful in many areas. For example, it can be used for internet conversations moderation. Also, it is possible to predict ratings that users can assign to a certain product (food, household appliances, hotels, films, etc) based on the reviews.

In this notebook we are using two families of machine learning algorithms: Naive Bayes (NB) and long short term memory (LSTM) neural networks.

- AYLIEN
- Deeplearning4j
- Understanding LSTM Networks
- Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling
- The Unreasonable Effectiveness of Recurrent Neural Networks

Import libraries

We will use pandas, numpy for data manipulation, nltk for natural language processing, matplotlib, seaborn and plotly for data visualization, sklearn and keras for learning the models.

```
import pandas as pdimport numpy as npimport
string, reimport itertools
import nltk

import plotly.offline as py

import plotly.graph_objs as go

import matplotlib.pyplot as plt

import seaborn as sns

from wordcloud import WordCloud,STOPWORDS

from sklearn.model_selection import
train_test_split

from sklearn.metrics import confusion_matrix

from sklearn.feature_extraction.text import
TfidfVectorizer

from sklearn.naive_bayes import MultinomialNB

from sklearn.metrics import accuracy_score

from keras.preprocessing.text import Tokenizer

from keras.preprocessing.sequence import
pad_sequences

from keras.models import Sequential

from keras.layers import Dense, Embedding, LSTM

from keras.callbacks import EarlyStopping
```

```
py.init_notebook_mode(connected=True)

%matplotlib inline
```

Load the dataset

First, we need to load the dataset from 3 separate files and concatenate them into 1 dataframe. You can find film reviews using the IMDB service, reviews about different local services using Yelp, and reviews about different goods using Amazon. The text of the reviews we insert in the Sentence column and the label with positive or negative sentiment in the Sentiment column.

```
imdb = pd.read_table('../input/imdb_labelled.txt',
header=None,                        names=
['Sentence', 'Sentiment'])yelp =
pd.read_table('../input/yelp_labelled.txt',
header=None,

                    names=['Sentence',
'Sentiment'])

amazon = pd.read_table('../input
/amazon_cells_labelled.txt',

                    header=None,

                    names=['Sentence',
'Sentiment'])


df = pd.concat([imdb, yelp, amazon])

df.head()
```

Figure 1: Preview of the dataset.

Source: Screenshot by Author

	Sentence	Sentiment
0	A very, very, very slow-moving, aimless movie ...	0
1	Not sure who was more lost - the flat characte...	0
2	Attempting artiness with black & white and cle...	0
3	Very little music or anything to speak of.	0
4	The best scene in the movie was when Gerardo i...	1

Exploratory Data Analysis

In the next 2 cells, we examine the shape of our dataset and check if there are some missing values. We can see that there aren't missing values because the shape of the dataset after removing missing values equals the shape of the dataset before this procedure.

```
print(df.shape)print(imdb.shape)print(yelp.shape)print(amazon.shap
print(df.dropna().shape)
```

Figure 2: Output.

Source: Screenshot by Author

```
(2748, 2)
(748, 2)
(1000, 2)
(1000, 2)
(2748, 2)
```

Next, we look at the length of all reviews in the Sentence column and measure some interesting statistics.

```
lens = df['Sentence'].str.len()print(lens.mean(),
lens.std(), lens.min(), lens.max())
```

Figure 3: Output.

Source: Screenshot by Author

```
71.52838427947599 201.98726634331325 7 7944
```

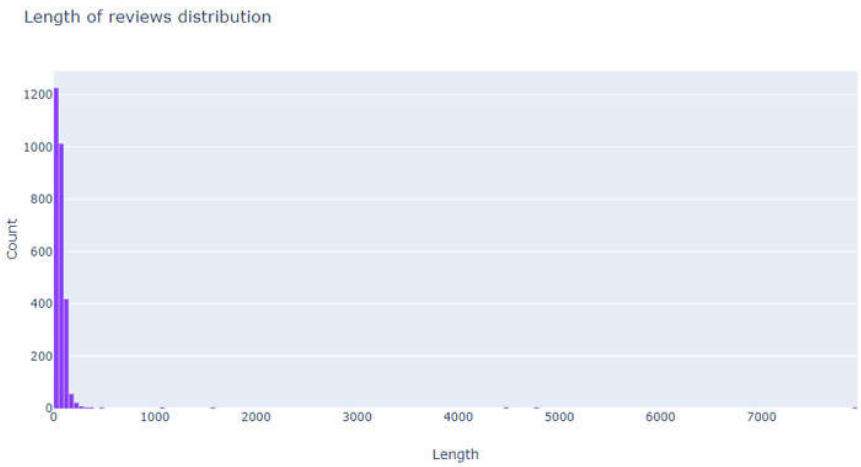
Note, that the average length of the review is 72 symbol, but very often it can reach 201 symbols. Also, we have some reviews with more than 7900 symbols. It seems, that there are some very short reviews because the shortest sentence has only 7 symbols and also the standard deviation and the mean points on this.

Let's also build a histogram with the distribution of the length of the reviews. We can see that most of the reviews have a length between 1 and 50 symbols. There are many reviews in the 50-100 symbols range. Almost all reviews have a length of fewer than 300 symbols.

```
data = [go.Histogram(x=lens, xbins=dict(start=0,
end=8000, size=50),
marker=dict(color='#8c42f4'))]layout = go.Layout(
title='Length of reviews distribution',
xaxis=dict(title='Length'),
yaxis=dict(title='Count'),

                    bargap=0.1)


fig = go.Figure(data=data, layout=layout)

py.iplot(fig, filename='length histogram')
```

Figure 4: Distribution of review length.

Source: Screenshot by Author

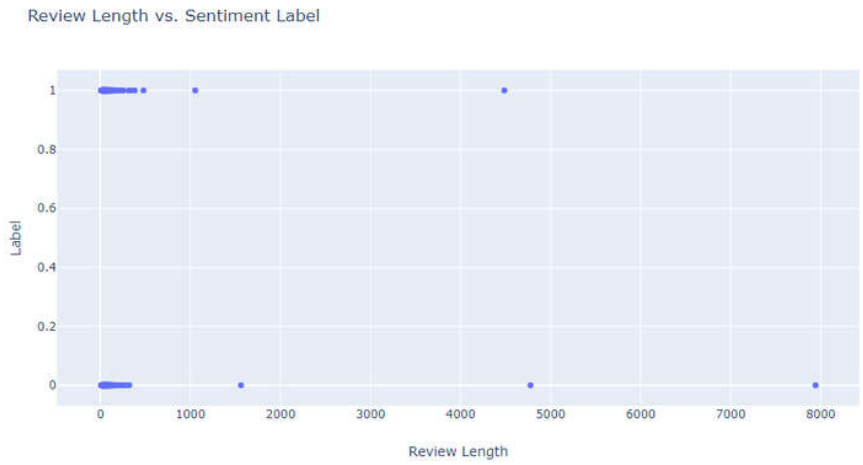


We can check whether there is any correlation between the length of the review and the sentiment label. On the 2 graphs below we demonstrate that there are no significant correlations between these variables.

```
df['senLen'] = df['Sentence'].apply(lambda x:
len(x))data = df.sort_values(by='senLen')plot =
go.Scatter(x = data['senLen'], y =
data['Sentiment'], mode='markers')lyt =
go.Layout(title="Review Length vs. Sentiment
Label", xaxis=dict(title='Review
Length'),yaxis=dict(title='Label'))
fig = go.Figure(data=[plot], layout=lyt)

py.iplot(fig)
```

Figure 5: Distribution of review length by the label.

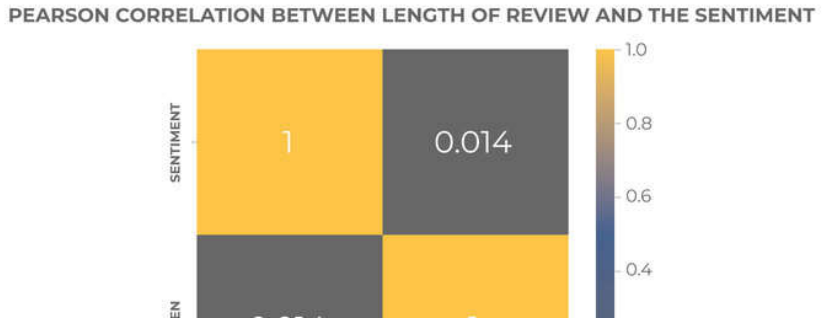
Source: Screenshot by Author




```
colormap = plt.cm.magmaplt.title('Pearson
correlation between length of review\
and the sentiment', y=1.05,
size=14)sns.heatmap(data.drop(['Sentence'],
axis=1).astype(float).corr(),
linewidths=0.01,

            vmax=1.0,

            square=True,

            cmap=colormap,

            linecolor='white',

            annot=True)

plt.show()
```

Figure 6: Heatmap of Pearson correlation.

Source: Screenshot by Author



Let's also visualize the wordclouds for sentences with positive and negative sentiment. You can see that for positive sentiment there are such words as "good", "well", "nice", "better", "best", "excellent", "wonderful" and so on. For the negative sentiment we can see words "bad", "disappointed", "worst", "poor" etc.

```
df_pos = df[ df['Sentiment'] == 1]df_pos =
df_pos['Sentence']df_neg = df[ df['Sentiment'] ==
0]df_neg = df_neg['Sentence']

wordcloud1 = WordCloud(stopwords=STOPWORDS,

                       background_color='white',

                       width=2500,

                       height=2000

                       ).generate(" ".join(df_pos))

plt.figure(1,figsize=(13, 13))

plt.imshow(wordcloud1)

plt.axis('off')

plt.show()
```

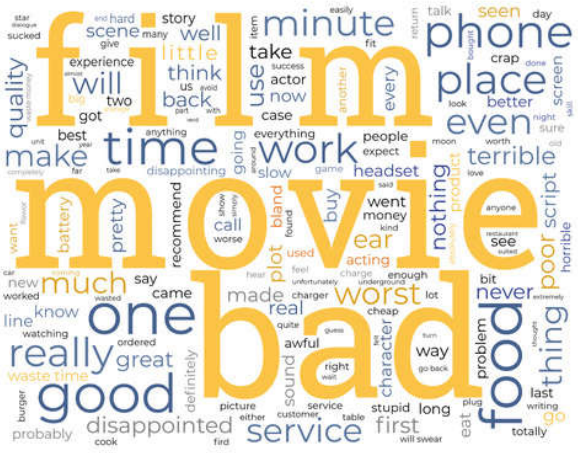
Figure 7: Wordcloud built with positive reviews.

Source: Screenshot by Author


```
wordcloud2 = WordCloud(stopwords=STOPWORDS,
background_color='white',
width=2500,                          height=2000
).generate(" ".join(df_neg))

plt.imshow(wordcloud2)

plt.axis('off')

plt.show()
```

Figure 8: Wordcloud built with negative reviews.

Source: Screenshot by Author



Distribution analysis

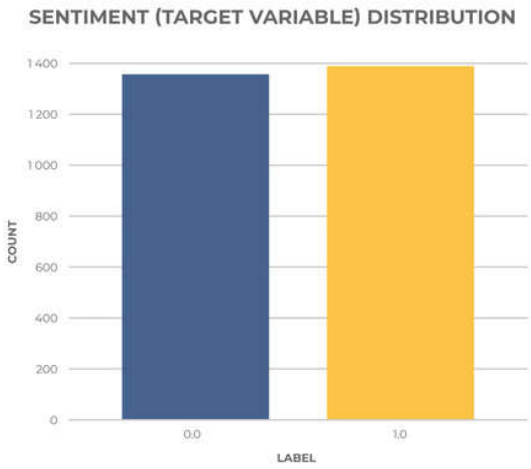
Now we would like to examine the distributions of sentiments. From the histogram below we can see that the dataset is balanced, which is very good in terms of algorithms training. There are approximately 1400 examples in both "0" and "1" categories.

```
sns.set(style="darkgrid")b =
sns.countplot(x='Sentiment',                    data
= df.drop(['Sentence'],
axis=1).astype(float))b.axes.set_title('Sentiment
(target variable) distribution')
b.set_xlabel("Label")

b.set_ylabel("Count")
```

```
plt.show()
```

Figure 9: Distribution of sentiments in the dataset.

Source: Screenshot by Author



Text preprocessing

Before we feed our data to the learning algorithms we need to preprocess it. For this dataset, we remove all punctuation signs and transform all letters to the lowercase.

```
def remove_punct(x):    comp = re.compile("[%s\d]"
% re.escape(string.punctuation))    return "
".join(comp.sub(" ", str(x)).split()).lower()
df['Sentence'] =
df['Sentence'].apply(remove_punct)
```

Now we split the dataset into features (x) and target variable (y). In the second cell, we split the dataset into train and test set. The size of the test set is 10%.

```
X = df['Sentence']y = df['Sentiment']X_train,
X_test, y_train, y_test = train_test_split(X, y,
test_size=0.1)
```

Train the model

Now we can train our model. We will use TfidfVectorizer to transform words into features. Also, we have tried several different models using scikit-learn. We noticed the following things (we use fixed random state to be able to compare results):

1. The performance of the models trained after CountVectorizer was

generally lower than those trained after TfidfVectorizer. For example, with lemmatization, Tfidf scored 0.8436 accuracy and CountVectorizer only 0.8181.

2. POS tagging filtering impairs the performance significantly (to just 0.6 accuracy score).
3. SnowballStemmer doesn't downgrade the performance, while PorterStemmer reduced the accuracy to 0.8472 from 0.8509.
4. Removing of stop words reduced the accuracy to 0.8.
5. Trying bigrams and 3-grams impairs the performance significantly.
6. Gaussian Naive Bayes produces only 0.669 accuracy score, so Multinomial Naive Bayes is better for this dataset.

   We use parameter min_df = 2 to filter out words that occur only once in the entire dataset.

```
vectorizer = TfidfVectorizer(min_df=2)train_term =
vectorizer.fit_transform(X_train)test_term =
vectorizer.transform(X_test)
```

You can look at words which will be used by our model as features:

```
vectorizer.get_feature_names()[:20]
```

Figure 10: Output

Source: Screenshot by Author

```
['ability', 'able', 'about', 'above', 'absolutely', 'abysmal', 'accept', 'access', 'acknowledged', 'act',
 'acted', 'acting', 'action', 'actor', 'actors', 'actress', 'actresses', 'actually', 'adaptation', 'add']
```

In the next cell, we train the Multinomial Naive Bayes model. After we get the predictions of this model on the training and testing dataset and measure the accuracy. We can see that the accuracy on the test set is quite high (about 85%).

```
model = MultinomialNB()model.fit(train_term,
y_train)predictions_train =
model.predict(train_term)predictions_test =
model.predict(test_term)

print('Train Accuracy:', accuracy_score(y_train,
predictions_train))

print('Test Accuracy:', accuracy_score(y_test,
predictions_test))
```

Figure 11: Output.

Source: Screenshot by Author

```
Train Accuracy: 0.9223615042458553
Test Accuracy: 0.8690909090909091
```

Now we want to build the confusion matrix to visualize the results of our model. The confusion matrix allows understanding what type of mistakes the algorithm makes.

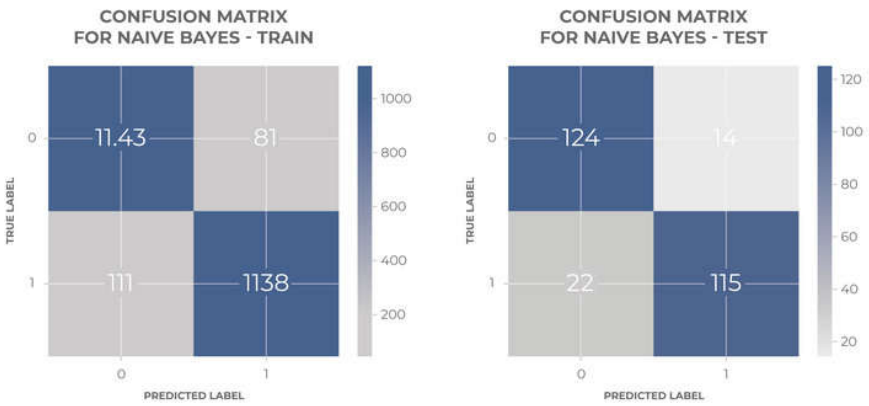
```
def plot_confusion_matrix(cm, classes,
normalize=False, title='Confusion matrix',
cmap=plt.cm.Blues):    """    This function prints
and plots the confusion matrix.    Normalization
can be applied by setting `normalize=True`.
"""

    if normalize:

        cm = cm.astype('float') /
cm.sum(axis=1)[:, np.newaxis]

        print("Normalized confusion matrix")

    else:

        print('Confusion matrix, without
normalization')

    print(cm)


    plt.imshow(cm, interpolation='nearest',
cmap=cmap)

    plt.title(title)

    plt.colorbar()

    tick_marks = np.arange(len(classes))

    plt.xticks(tick_marks, classes, rotation=45)
```

```
    plt.yticks(tick_marks, classes)


    fmt = '.2f' if normalize else 'd'

    thresh = cm.max() / 2.

    for i, j in
itertools.product(range(cm.shape[0]),
range(cm.shape[1])):

        plt.text(j, i, format(cm[i, j], fmt),

                 horizontalalignment="center",

                 color="white" if cm[i, j] >
thresh else "black")


    plt.tight_layout()

    plt.ylabel('True label')

    plt.xlabel('Predicted label')



cnf_matrix_train = confusion_matrix(y_train,
predictions_train)

cnf_matrix_test = confusion_matrix(y_test,
predictions_test)



plt.figure()

plot_confusion_matrix(cnf_matrix_train,

                      classes=[0,1],
```

```
                      title='Confusion matrix for
Naive Bayes - train')

plt.figure()

plot_confusion_matrix(cnf_matrix_test,

                      classes=[0,1],

                      title='Confusion matrix for
Naive Bayes - test')
```

Figure 12: Confusion matrices.

Source: Screenshot by Author



We can see that there are 18 test examples with "1" sentiment which model classified as "0" sentiment and 23 examples with "0" sentiment which model classified as "1" label.

Prediction with LSTM

Now we will try to use Long Short Term Memory neural network to improve the performance of our initial model. Firstly, we need to tokenize our sentences using Keras' Tokenizer. Also, we set num_words to 2000. This means that the tokenizer detects the 2000 most frequent words from the dataset and use them as features for further model building.

```
max_features = 2000tokenizer =
Tokenizer(num_words=max_features, split='
')tokenizer.fit_on_texts(df['Sentence'].values)X =
```

```
tokenizer.texts_to_sequences(df['Sentence'].values)
X = pad_sequences(X)
```

Here we build our model. First, we initialize the sequence model. Then, we add word embedding layer, lstm layer with 16 units and the fully connected (dense) layer with 1 (output) neuron with the sigmoid activation function. We use binary_crossentropy loss and Adam optimizer to train the model. Also, we set accuracy as the metric for measuring model's performance. You can see the summary of the model in the output of the next cell.

It is also worth to say that we tried to use Bidirectional LSTM model and 2-layers stacked LSTM model. Training time forthese models increases in several times, but there is almost no performance improvement.

```
embed_dim = 64lstm_out = 16model = Sequential()
model.add(Embedding(max_features, embed_dim,
input_length=X.shape[1]))

model.add(LSTM(lstm_out))

model.add(Dense(1,activation='sigmoid'))

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

print(model.summary())
```

Figure 13: Architecture of the LSTM model.

Source: Screenshot by Author

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 1191, 64)          128000
_________________________________________________________________
lstm_1 (LSTM)                (None, 16)                5184
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 17
=================================================================
Total params: 133,201
Trainable params: 133,201
Non-trainable params: 0
_________________________________________________________________
None
```

Get the target label in the separate variable:

```
Y = df['Sentiment'].values
```

Split the dataset into the train and test parts:

```
X_train, X_test, Y_train, Y_test =
train_test_split(X,Y, test_size =
0.1)print(X_train.shape,Y_train.shape)print(X_test.shape,Y_test.sha
```

Figure 14: Output.

Source: Screenshot by Author

```
(2473, 1191) (2473,)
(275, 1191) (275,)
```

Next, we trigger the training of LSTM neural network. We use batch_size=16 and train the network for 6 epochs. The performance of the model is measured on the validation dataset. Also, we use an early stopping callback if the result is not improved during 2 training iterations (epochs).

```
batch_size = 16model.fit(X_train,
Y_train,          epochs=6,
batch_size=batch_size,

          validation_data=(X_test, Y_test),

          callbacks =
[EarlyStopping(monitor='val_acc',

                        min_delta=0.001,

                        patience=2,

                        verbose=1)]

           )
```

Figure 15: Model training process.

Source: Screenshot by Author

```
Train on 2473 samples, validate on 275 samples
Epoch 1/6
2473/2473 [==============================] - 203s 82ms/step - loss: 0.6371 - acc: 0.6628 - val_loss: 0.5183 - val_acc: 0.8145
Epoch 2/6
2473/2473 [==============================] - 202s 82ms/step - loss: 0.3770 - acc: 0.8585 - val_loss: 0.3554 - val_acc: 0.8582
Epoch 3/6
2473/2473 [==============================] - 207s 84ms/step - loss: 0.2429 - acc: 0.9131 - val_loss: 0.3488 - val_acc: 0.8836
Epoch 4/6
2473/2473 [==============================] - 212s 86ms/step - loss: 0.1548 - acc: 0.9499 - val_loss: 0.3743 - val_acc: 0.8764
Epoch 5/6
2473/2473 [==============================] - 236s 95ms/step - loss: 0.1224 - acc: 0.9644 - val_loss: 0.3882 - val_acc: 0.8727
Epoch 00005: early stopping
```

Now we use our LSTM model to predict the labels for the train and test set. LSTM model produced the answers as probabilities of classes. So, we use the threshold 0.5 to transform probabilities of classes into class labels. You can see the accuracy of the LSTM

neural network in the third cell. After, we build the confusion matrices for train and test sets.

```
 predictions_nn_train =
model.predict(X_train)predictions_nn_test =
model.predict(X_test)
for i in range(len(predictions_nn_train)):

    if predictions_nn_train[i][0] < 0.5:

        predictions_nn_train[i][0] = 0

    else:

        predictions_nn_train[i][0] = 1


for i in range(len(predictions_nn_test)):

    if predictions_nn_test[i][0] < 0.5:

        predictions_nn_test[i][0] = 0

    else:

        predictions_nn_test[i][0] = 1



print('Train accuracy:', accuracy_score(Y_train,
predictions_nn_train))

print('Test accuracy', accuracy_score(Y_test,
predictions_nn_test))
```

Figure 16: Output.

Source: Screenshot by Author

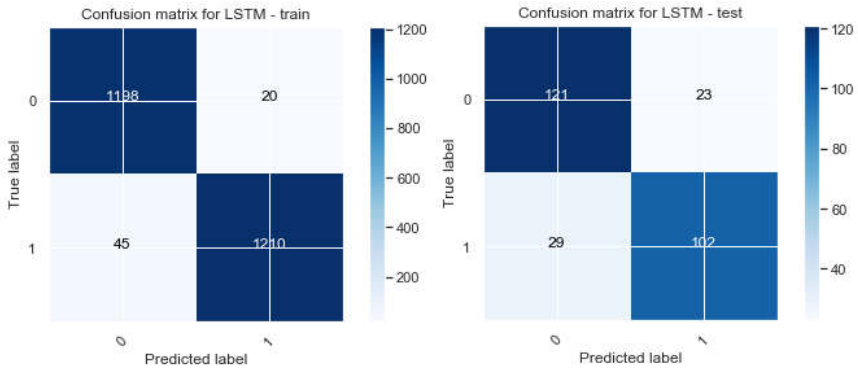
```
Train accuracy: 0.9789729073999192
Test accuracy 0.8727272727272727
```

```
predictions_nn_train =
model.predict(X_train)predictions_nn_test =
model.predicnf_matrix_train =
```

```
confusion_matrix(Y_train,
predictions_nn_train)cnf_matrix_test =
confusion_matrix(Y_test, predictions_nn_test)

plt.figure()

plot_confusion_matrix(cnf_matrix_train,

                      classes=[0,1],

                      title='Confusion matrix for
Naive Bayes - train')



plt.figure()

plot_confusion_matrix(cnf_matrix_test,

                      classes=[0,1],

                      title='Confusion matrix for
LSTM - test')
```

Figure 17: Confusion matrices of the LSTM model.

Source: Screenshot by Author



After looking at the confusion matrix we can assume that if we train several different models and then make an ensemble of these models we can improve the general prediction quality. This is possible because different models can make different mistakes and different true predictions and by ensembling we "average" the predictions (each model votes for a particular label for the target variable).

Let's make an error analysis. We will get the words for sentences where our model makes mistakes. Then, we compute the fraction of positive and negative sentences in which our model did mistakes.

```
 dictionary = tokenizer.word_indexsentences =
[]for j in range(len(X_test)):
sentence = []

    for i in range(len(X_test[j])):

        if X_test[j][i] == 0:

            continue

        else:

            for key, val in dictionary.items():

                if dictionary[key] ==
X_test[j][i]:

                    sentence.append(key)

    sentences.append(sentence)


err_analysis = pd.DataFrame({'Sentences':
sentences,

                             'y_true': Y_test,

                             'y_pred':
predictions_nn_test.reshape(275,)})

err_analysis.head(10)
```

Figure 18: Error analysis table.

Source: Screenshot by Author

	Sentences	y_pred	y_true
0	[i, would, highly, recommend, this]	1.0	1
1	[it, is, worth, the, drive]	0.0	1
2	[they, have, a, good, selection, of, food, inc...	1.0	1

3	[i, dont, think, i, will, be, back, for, a, ve...	0.0	0
4	[a, world, better, than, of, the, garbage, in,...	0.0	1
5	[works, like, a, charm, it, work, the, same, a...	1.0	1
6	[the, price, is, reasonable, and, the, service...	1.0	1
7	[the, feel, of, the, dining, room, was, more, ...	1.0	0
8	[and, the, beans, and, rice, were, mediocre, a...	0.0	0
9	[voice, quality, signal, dropped, calls]	0.0	0

Next, we can see that there are several possible reasons for mistakes. It is sometimes harder even for a human to detect the sentiment of some sentences. Some sentences have specific words that allow humans to detect sentiment, but the algorithm is not able to use these words. We think that this problem can be solved by using a larger dataset for training. Sometimes the model cannot detect sentiment where it is hidden by complex sentence structure. We can think about how to improve the performance by trying to help the model in these particular cases.

```
errors =
err_analysis.loc[err_analysis['y_pred']!=err_analysis['y_true']]er
```

Figure 19: Errors from error analysis table.

Source: Screenshot by Author

	Sentences	y_pred	y_true
1	[it, is, worth, the, drive]	0.0	1
4	[a, world, better, than, of, the, garbage, in,...	0.0	1
7	[the, feel, of, the, dining, room, was, more, ...	1.0	0
18	[it, is, perfect, for, a, sit, down, family, m...	0.0	1
33	[we, had, so, much, to, say, about, the, place...	1.0	0

Figure 20: Output.

Source: Screenshot by Author

```
['the',
 'feel',
 'of',
 'the',
 'dining',
 'room',
 'was',
 'more',
 'course',
 'than',
 'high',
 'class',
 'dining',
 'and',
 'the',
 'service',
 'was',
 'slow',
 'at',
 'best']
```

```
df_pos = df[ df['Sentiment'] == 1]df_neg = df[
```

```
df['Sentiment'] == 0]all_count_pos =
len(df_pos)all_count_neg = len(df_neg)
print('Count positives: ', all_count_pos)

print('Count negatives: ', all_count_neg)

err_count_pos = len(errors[ errors['y_true'] ==
1])

err_count_neg = len(errors[ errors['y_true'] ==
0])

print('Errors in true positive: ', err_count_pos)

print('Errors in true negative: ', err_count_neg)


print('Fraction of the errors with true
positive:', round(err_count_pos/all_count_pos, 4))

print('Fraction of the errors with true
negative:', round(err_count_neg/all_count_neg, 4))
```

Figure 21: Error analysis statistics.

Source: Screenshot by Author

```
Count positives:  1386
Count negatives:  1362
Errors in true positive:  23
Errors in true negative:  12
Fraction of the errors with true positive: 0.0166
Fraction of the errors with true negative: 0.0088
```

Conclusion

In this notebook, we trained the Long Short Term Memory Neural Network, as well as Multinomial Naive Bayes Classifier using UCI Sentiment Labelled Sentences Data Set. These models can be used to predict sentiment analysis of the users' reviews. The performance of the models is quite good. Both models give 85-87% accuracy on average.