Classification of Amazon Fine Food Reviews using LSTM

Prepared by

Büşra Erkoç

1.Introduction

Sentiment analysis aims to analyze the author's feelings and thoughts about a particular subject in a written text, while sentiment classification aims to classify the emotion as positive and negative. Social media has become an important repository of information, where people want to share their personal ideas, opinions and suggestions with other people and learn about others' views and suggestions on a topic. As a result of the work done in this information store, companies can learn their customers' opinions and opinions about their products and services and use the data obtained by classifying them to develop their products and services. Researchers uses these information to do an in-depth analysis of market trends and consumer opinions, which could potentially lead to a better prediction of the stock market. [2]

In this study, Amazon fine food reviews data set was used. [1] Amazon is one of the largest online vendor in the world. People often gaze over the products and reviews of the product before buying the product on amazon itself. The dataset includes the reviews from Oct 1999 to Oct 2012 and has a total of 568,454 reviews on 74,258 products.

Data preprocessing operations were performed on the reviews. Data preprocessing includes tasks such as removal of punctuation and numbers, normalization of uppercase and lowercase letters, removal of stop words.

The data was prepared for the embedding layer. Tokenization and padding operations were done. The resulting models were trained. 7 different models were used. These models are compared with the activation function, lot size, optimization algorithm, release layer and LSTM layer number, accuracy value.

The objective of this study is to classify the positive and negative reviews of customers on different products. The effects of hyperparameters in the model used for classification on model performance were compared.

The present study is the work on the Amazon fine food review classification based on the sentiment analysis using the machine learning approach. There are many ways that we can classify the review based on the emotion yet we are using the most advanced technique which is LSTM networks.

2.Material and Methods

a. Dataset Description

In this study, Amazon fine food reviews data set was used. The dataset is provided by Kaggle website [1] and was collected and published first by McAuley et al. (2013) in their research related to online reviews. The dataset consists of reviews on fine food posted on Amazon.com. It has a total of 568,454 reviews on 74,258 products. This dataset includes the reviews from Oct 1999 to Oct 2012.

Attribute Name	**Description**	**Variable Type**
ProductId	Unique identifier for the product	Categorical
UserId	Unqiue identifier for the use	Categorical
ProfileName	Profile name of the user	Text
Helpfulness Numerator	Number of users who found the review helpful	Numerical
Helpfulness Denominator	Number of users who indicated whether they found the review helpful or not	Numerical
Score	Rating between 1 and 5	Ordinal
Time	Timestamp for the review	Numerical
Summary	Brief summary of the review	Text
Text	Text of the review	Text

b. Theoretical details on the topic

Sentiment Analysis

Sentiment analysis, also called opinion mining, is the field of study that analyzes people's opinions, sentiments, evaluations, appraisals, attitudes, and emotions towards entities such as products, services, organizations, individuals, issues,

events, topics, and their attributes. [1] Sentiment classification, the best-known sentiment analysis task, aims to detect sentiments within a document, a sentence, or an aspect. This task can be divided into three steps: polarity detection (label the sentiment of the text as positive, negative, or neutral), aspect selection/extraction (obtain the features for structuring the text), and classification (apply machine learning or lexicon approaches to classify the text).

Sentiment analysis is performed on multiple linguistic levels.

- At the document level, the task is to classify whether a whole opinionated document has a positive, negative or neutral sentiment.
- At the sentence level, the task is to classify whether an individual sentence has a positive, negative or neutral sentiment.
- At the aspect level (the entity level), the task is to classify the sentiment of individual sentences or phrases intended towards certain entities or aspects.

Neural Networks

Deep learning is the application of artificial neural networks (neural networks for short) to learning tasks using networks of multiple layers. Inspired by the structure of the biological brain, neural networks consist of a large number of information processing units (called neurons) organized in layers, which work in unison. It can learn to perform tasks (e.g., classification) by adjusting the connection weights between neurons, resembling the learning process of a biological brain.[4]

Neural networks are a machine learning framework loosely based on the structure of the human brain. They are very commonly used to complete tasks that seem to require complex decision making, like speech recognition or image classification.

In traditional machine learning approaches, features are defined and extracted either manually or making by making use of feature use of feature selection selection methods. However, in deep learning models, features are learned extracted automatically, achieving better accuracy and performance.[5]

RNNs are one of the most commonly used neural network architectures today. Within the domain of Natural Language Processing, they are often used in speech generation and machine translation tasks. Additionally, they are often used to solve speech recognition and optical character recognition tasks.

Word Embedding

Word embedding is a technique for language modelling and feature learning, which transforms words in a vocabulary to vectors continuous real numbers.[2]

The applications of word embeddings include:

- entity recognition in chatbots
- sentiment analysis
- syntax parsing

Word embeddings are key to natural language processing. Each is a real number vector representation of a specific word. Contextual information about that word is encoded within the vector numbers.

Word2Vec is a statistical learning algorithm that creates embeddings using written text (a corpus). A common word embedding system is Word2Vec. This method was used in this study There are two different architectures of the corpus it can use:

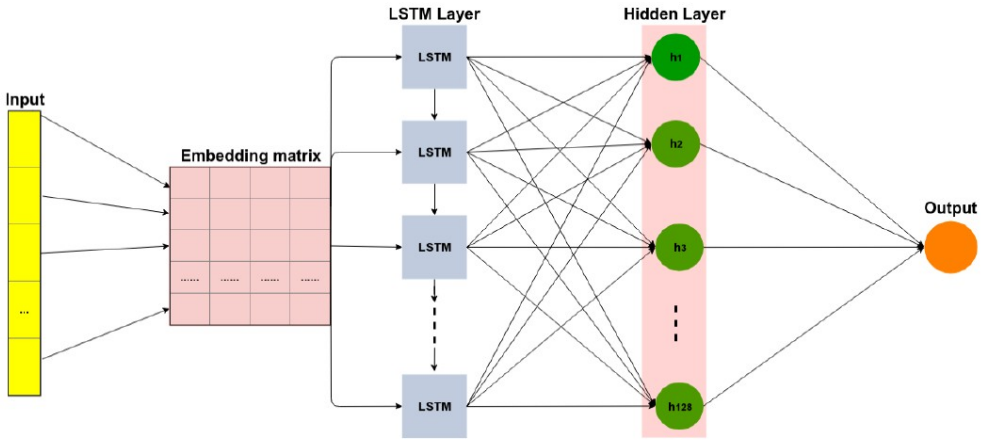
- Continuous Bag of Words: the algorithm goes through each word in the training corpus, in order, and predicts the word at each position based on applying bag-of-words to surrounding words. The order of the words does not matter!
- Continuous Skip-Grams: Look at sequences of words that are separated by some specified distance, as opposed to the common practice of looking at groups of n-consecutive words in a text (n-grams). The order of context is taken into consideration!

Another frequently used learning approach is **Global Vector (GloVe)**, which is trained on the non-zero entries of a global word-word co-occurrence matrix.[4]

In this study The max text length has been set to 100. Then, all the review text data has been padding to 200 word length. After that, each word is represented by word vector as the input of the neural network.

LSTM (Long Short Term Memory)

Long Short Term Memory(LSTM) is unit of Recurrent Neural Network(RNN). A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell. LSTM networks are well-suited to classifying, processing and making predictions based on time series data, since there can be lags of unknown duration between important events in a time series. [2]



Unlike RNN, LSTM contains special units called memory blocks in the recurrent hidden layer. The memory blocks contain memory cells with self-connections storing the temporal state of the network in addition to special multiplicative units called gates to control the flow of information. Each memory block in the original architecture contained three gate types which are namely:

- Input gate: the input gate controls the flow of input activations into the memory cell.
- Output gate: output gate controls the output flow of cell activations into the rest of the network.
- Forget gate: scales the internal state of the cell before adding it as input to the cell through the self-recurrent connection of the cell, therefore adaptively forgetting or resetting the cell's memory.[7]

In this project, we used the LSTM with 100 hidden units and then used a dense net with sigmoid as the activation function. The data has been trained for 5 epochs in experiments using LSTM. Adam optimizer has been used to optimize the parameters, the batch size is 64.

c. Python Codes

Sentiment Analysis with LSTM on Amazon Fine Food Reviews

In [5]:

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
import re
import string
import matplotlib.pyplot as plt
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers.embeddings import Embedding
from keras.preprocessing.text import Tokenizer
from keras.preprocessing import sequence
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
import seaborn as sns
from sklearn.model_selection import train_test_split
```

In [6]:

```
df = pd.read_csv('../input/amazon-fine-food-reviews/Reviews.csv')
df.shape
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 568454 entries, 0 to 568453
Data columns (total 10 columns):
 #   Column                  Non-Null Count   Dtype
---  ------                  --------------   -----
 0   Id                      568454 non-null  int64
 1   ProductId               568454 non-null  object
 2   UserId                  568454 non-null  object
 3   ProfileName             568438 non-null  object
 4   HelpfulnessNumerator    568454 non-null  int64
 5   HelpfulnessDenominator  568454 non-null  int64
 6   Score                   568454 non-null  int64
 7   Time                    568454 non-null  int64
 8   Summary                 568427 non-null  object
 9   Text                    568454 non-null  object
dtypes: int64(5), object(5)
memory usage: 43.4+ MB
```

Add labels to reviews according to score values

- Drop the reviews with score 3. (neutral)
- Those with a score value of 5 and 4 are labeled as positive, ie 1.
- Those with a score value of 1 and 2 are labeled negative, ie 0.

In [7]:

```
df=df[['Text','Score']]
df['review']=df['Text']
df['label']=df['Score']
df.drop(['Text','Score'],axis=1,inplace=True)
df.head()
df = df[df['label'] != 3]  # drop rating values 3 reviews, neutral reviews

def mark_sentiment(rating):
    if(rating<3):
        return 0 # negative
    else:
        return 1 # positive
df['label']=df['label'].apply(mark_sentiment)

# check missing value and duplicates
print(df['label'].isnull().sum())
df['review'].isnull().sum()
df.drop_duplicates(subset=['label','review'],keep='first',inplace=True)
```

```
0
```

Select balanced 20000 reviews

In [8]:

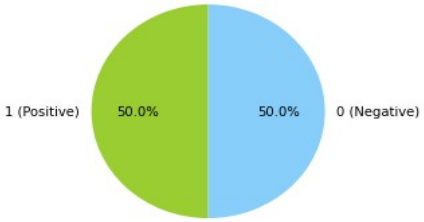
```
pos_df=df.loc[df.label==1,:][:10000]
neg_df=df.loc[df.label==0,:][:10000]
df=pd.concat([pos_df,neg_df],ignore_index=True)
df = df.sample(frac=1).reset_index(drop=True) #perform a random shuffle. It's a good practice to shuffle the data before splitting between a train and test set. That way the sentiment classes are equally distributed over the train and test sets. df = df.reindex(np.random.permutation(df.index))
df.head()
```

Out[8]:

	review	label
0	These are every bit as good as the other two r...	1
1	I ordered three of the trays (being a date nut...	0
2	Lana's Snowflake Black Tea with Coconut and Al...	1
3	Length:: 1:38 Mins <b...	1
4	I'd rank Toffifay as tied with Ferrero Rocher ...	1

Data Distribution

In [9]:
```
pie_label = ["1 (Positive)","0 (Negative)"]
pie_colors = ["yellowgreen", "lightskyblue"]
plt.pie(df.label.value_counts(), labels=pie_label, colors=pie_colors, startangle=90, autopct
='%.1f%%')
plt.show()
```



Data Preprocesing

In [10]:
```
df['review'] = df.review.apply(lambda x : x.lower())  # all reviews characters converts to lower
case
df['review'] = df.review.apply(lambda x: re.sub(r'\d+','', x))  # remove the numbers in reviews
df['review'] = df.review.apply(lambda x: re.sub(r'[^\w\s]', ' ',x)) # remove the punctuations in
reviews
df['review'] = df.review.apply(lambda x: x.strip()) # remove whitespaces
```

Preparing the Embedding Layer

Tokenizing the dataset

In [11]:
```
x = df['review']
y = df['label']
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.20, random_state=42)

tokenizer = Tokenizer(num_words=5000)  # Initialize the tokenizer with a 5000 word limit. This i
s the number of words we would like to encode.
tokenizer.fit_on_texts(x_train) # we call fit_on_texts to create associations of words and numbe
rs

# After having created the dictionary we can convert the text to a list of integer indexes.
# This is done with the text_to_sequences method of the Tokenizer.
# calling text_to_sequence replaces the words in a sentence with their respective associated numb
ers.
# This transforms each sentence into sequences of numbers.
x_train = tokenizer.texts_to_sequences(x_train)
x_test = tokenizer.texts_to_sequences(x_test)
```

Padding the dataset

In [12]:

```
# The reviews have different number of words, therefore, the length of the sequence of numbers wi
ll be different.
# Our model requires inputs to have equal lengths, so we will have to pad the sequence to have th
e chosen length of inputs.
# This is done by calling the pad_sequence method with a length of 200.
vocab_size = len(tokenizer.word_index) + 1
max_review_length = 200

x_train = sequence.pad_sequences(x_train, maxlen=max_review_length)
x_test = sequence.pad_sequences(x_test, maxlen=max_review_length)
print(x_train.shape)
```

```
(16000, 200)
```

Model 1

Embedding → LSTM → Dense(Output(Sigmoid))

In [41]:

```
# Build Model
embedding_vector_length = 32
model = Sequential()
model.add(Embedding(vocab_size, embedding_vector_length, input_length =max_review_length))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train Model
history = model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=128, epochs=
5)
result = model.evaluate(x_test, y_test, verbose=0)
#Results
print("Test Score:", result[0])
print("Test Accuracy:", result[1])
print("Accuracy: %.2f%%" % (result[1]*100))
```

```
Train on 16000 samples, validate on 4000 samples
Epoch 1/5
16000/16000 [==============================] - 47s 3ms/step - loss: 0.5106 - accuracy: 0.754
1 - val_loss: 0.3612 - val_accuracy: 0.8505
Epoch 2/5
16000/16000 [==============================] - 47s 3ms/step - loss: 0.2684 - accuracy: 0.891
5 - val_loss: 0.2854 - val_accuracy: 0.8895
Epoch 3/5
16000/16000 [==============================] - 47s 3ms/step - loss: 0.2038 - accuracy: 0.924
4 - val_loss: 0.2781 - val_accuracy: 0.8917
Epoch 4/5
```

Model 2

Embedding → LSTM → Dense(Output(Sigmoid))
Same with Model 1 except optimization algorithm. In this study used Adadelta optimizer. With this model, the effect of optimization algorithm is observed.

In [43]:

```
# Build Model
embedding_vector_length = 32
model = Sequential()
model.add(Embedding(vocab_size, embedding_vector_length, input_length =max_review_length))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='Adadelta', metrics=['accuracy'])

# Train Model
history = model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=128, epochs=
5)
result = model.evaluate(x_test, y_test, verbose=0)
#Results
print("Test Score:", result[0])
print("Test Accuracy:", result[1])
print("Accuracy: %.2f%%" % (result[1]*100))
```

```
Train on 16000 samples, validate on 4000 samples
Epoch 1/5
16000/16000 [==============================] - 47s 3ms/step - loss: 0.6917 - accuracy: 0.530
3 - val_loss: 0.6858 - val_accuracy: 0.5437
Epoch 2/5
16000/16000 [==============================] - 47s 3ms/step - loss: 0.5785 - accuracy: 0.696
8 - val_loss: 0.4429 - val_accuracy: 0.8037
Epoch 3/5
16000/16000 [==============================] - 47s 3ms/step - loss: 0.4190 - accuracy: 0.806
9 - val_loss: 0.3640 - val_accuracy: 0.8462
```

Model 3

Embedding → LSTM → Dense(Output(Sigmoid))
Same as Model 1, only batch_size different. With this model, the effect of batch size is observed.

In [45]:

```
# Build Model
embedding_vector_length = 32
model = Sequential()
model.add(Embedding(vocab_size, embedding_vector_length, input_length =max_review_length))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train Model
history = model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=256, epochs=
5)
result = model.evaluate(x_test, y_test, verbose=0)
#Results
print("Test Score:", result[0])
print("Test Accuracy:", result[1])
print("Accuracy: %.2f%%" % (result[1]*100))
```

```
Train on 16000 samples, validate on 4000 samples
Epoch 1/5
16000/16000 [==============================] - 41s 3ms/step - loss: 0.6486 - accuracy: 0.690
2 - val_loss: 0.5172 - val_accuracy: 0.7887
Epoch 2/5
16000/16000 [==============================] - 40s 3ms/step - loss: 0.4069 - accuracy: 0.827
1 - val_loss: 0.3598 - val_accuracy: 0.8462
Epoch 3/5
16000/16000 [==============================] - 40s 2ms/step - loss: 0.2722 - accuracy: 0.888
8 - val_loss: 0.2752 - val_accuracy: 0.8890
```

Model 4

Embedding → LSTM → LSTM → Dense(Output(Sigmoid))

In [16]:

```
# Build Model
embedding_vector_length = 32
model = Sequential()
model.add(Embedding(vocab_size, embedding_vector_length, input_length =max_review_length))
model.add(LSTM(100, return_sequences = True))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Train Model
history = model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=128, epochs=
5)
result = model.evaluate(x_test, y_test, verbose=0)
# Results
print("Test Score:", result[0])
print("Test Accuracy:", result[1])
print("Accuracy: %.2f%%" % (result[1]*100))
```

```
Train on 16000 samples, validate on 4000 samples
Epoch 1/5
16000/16000 [==============================] - 101s 6ms/step - loss: 0.4566 - accuracy: 0.77
32 - val_loss: 0.3384 - val_accuracy: 0.8600
Epoch 2/5
16000/16000 [==============================] - 99s 6ms/step - loss: 0.2822 - accuracy: 0.887
3 - val_loss: 0.2869 - val_accuracy: 0.8840
Epoch 3/5
16000/16000 [==============================] - 100s 6ms/step - loss: 0.2133 - accuracy: 0.91
87 - val_loss: 0.2872 - val_accuracy: 0.8845
Epoch 4/5
16000/16000 [==============================] - 99s 6ms/step - loss: 0.1729 - accuracy: 0.936
```

Model 5

Embedding → LSTM → Dropout → LSTM → Dropout → Dense(Output(Sigmoid))

In [19]:

```
from keras.layers import Dropout
# Build Model
embedding_vector_length = 32
model = Sequential()
model.add(Embedding(vocab_size, embedding_vector_length, input_length =max_review_length))
model.add(LSTM(100, return_sequences=True))
model.add(Dropout(0.5))
model.add(LSTM(100))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Train Model
history = model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=128, epochs=
5)
result = model.evaluate(x_test, y_test, verbose=0)
# Results
print("Test Score:", result[0])
print("Test Accuracy:", result[1])
print("Accuracy: %.2f%%" % (result[1]*100))
```

```
Train on 16000 samples, validate on 4000 samples
Epoch 1/5
16000/16000 [==============================] - 103s 6ms/step - loss: 0.4673 - accuracy: 0.76
20 - val_loss: 0.3294 - val_accuracy: 0.8625
Epoch 2/5
16000/16000 [==============================] - 103s 6ms/step - loss: 0.2675 - accuracy: 0.89
```

Model 6

Embedding → LSTM → Dense(activation:Relu) → Dense(Output(Sigmoid))
With this model, the effect of activation function is observed.

In [58]:

```
# Build Model
embedding_vector_length = 32
model = Sequential()
model.add(Embedding(vocab_size, embedding_vector_length, input_length =max_review_length))
model.add(LSTM(100))
model.add(Dense(64, kernel_initializer='uniform', activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Train Model
history = model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=128, epochs=
5)
result = model.evaluate(x_test, y_test, verbose=0)
# Results
print("Test Score:", result[0])
print("Test Accuracy:", result[1])
print("Accuracy: %.2f%%" % (result[1]*100))
```

```
Train on 16000 samples, validate on 4000 samples
Epoch 1/5
16000/16000 [==============================] - 47s 3ms/step - loss: 0.5237 - accuracy: 0.733
9 - val_loss: 0.3155 - val_accuracy: 0.8673
Epoch 2/5
16000/16000 [==============================] - 48s 3ms/step - loss: 0.2575 - accuracy: 0.899
8 - val_loss: 0.2882 - val_accuracy: 0.8832
```

Model 7

Embedding → LSTM → Dense(activation:Softmax) → Dense(Output(Sigmoid))
With this model, the effect of activation function is observed.

In [17]:

```
# Build Model
embedding_vector_length = 32
model = Sequential()
model.add(Embedding(vocab_size, embedding_vector_length, input_length =max_review_length))
model.add(LSTM(100))
model.add(Dense(64, kernel_initializer='uniform', activation='softmax'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Train Model
history = model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=128, epochs=
5)
result = model.evaluate(x_test, y_test, verbose=0)
# Results
print("Test Score:", result[0])
print("Test Accuracy:", result[1])
print("Accuracy: %.2f%%" % (result[1]*100))
```

```
Train on 16000 samples, validate on 4000 samples
Epoch 1/5
16000/16000 [==============================] - 47s 3ms/step - loss: 0.6920 - accuracy: 0.543
3 - val_loss: 0.6953 - val_accuracy: 0.5033
Epoch 2/5
16000/16000 [==============================] - 47s 3ms/step - loss: 0.6936 - accuracy: 0.500
6 - val_loss: 0.6928 - val_accuracy: 0.5048
```

main.py

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
import re
import string
import matplotlib.pyplot as plt
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers.embeddings import Embedding
from keras.preprocessing.text import Tokenizer
from keras.preprocessing import sequence
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
import seaborn as sns
from sklearn.model_selection import train_test_split
# Loading the Dataset
df = pd.read_csv('../input/amazon-fine-food-reviews/Reviews.csv')
df.shape
df.info()
```

```
df=df[['Text','Score']]

df['review']=df['Text']

df['label']=df['Score']

df.drop(['Text','Score'],axis=1,inplace=True)

df.head()

df = df[df['label'] != 3]  # drop rating values 3 reviews, neutral reviews

def mark_sentiment(rating):

    if(rating<3):

        return 0 # negative

    else:

        return 1 # positive

df['label']=df['label'].apply(mark_sentiment)

# check missing value and duplicates

print(df['label'].isnull().sum())

df['review'].isnull().sum()

df.drop_duplicates(subset=['label','review'],keep='first',inplace=True)

#Select balanced 20000 reviews

pos_df=df.loc[df.label==1,:][:10000]

neg_df=df.loc[df.label==0,:][:10000]

df=pd.concat([pos_df,neg_df],ignore_index=True)

df = df.sample(frac=1).reset_index(drop=True) #perform a random shuffle. It's a good practice to shuffle the data before splitting between a train and test set. That way the sentiment classes are equally distributed over the train and test sets. df = df.reindex(np.random.permutation(df.index))

df.head()
```

```
# Data Distribution

pie_label = ["1 (Positive)","0 (Negative)"]

pie_colors = ["yellowgreen", "lightskyblue"]

plt.pie(df.label.value_counts(), labels=pie_label, colors=pie_colors, startangle=90, autopct='%.1f%%')

plt.show()

# Data Preprocesing

df['review'] = df.review.apply(lambda x : x.lower())  # all reviews characters converts to lowercase

df['review'] = df.review.apply(lambda x: re.sub(r'\d+','', x))  # remove the numbers in reviews

df['review'] = df.review.apply(lambda x: re.sub(r'[^\w\s]', ' ',x)) # remove the punctuations in reviews

df['review'] = df.review.apply(lambda x: x.strip()) # remove whitespaces

# Preparing the Embedding Layer

# Tokenizing the dataset

x = df['review']

y = df['label']

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.20, random_state=42)

tokenizer = Tokenizer(num_words=5000)  # Initialize the tokenizer with a 5000 word limit. This is the number of words we would like to encode.

tokenizer.fit_on_texts(x_train) # we call fit_on_texts to create associations of words and numbers

x_train = tokenizer.texts_to_sequences(x_train)

x_test = tokenizer.texts_to_sequences(x_test)
```

```
# Padding the dataset

vocab_size = len(tokenizer.word_index) + 1

max_review_length = 200

x_train = sequence.pad_sequences(x_train, maxlen=max_review_length)

x_test = sequence.pad_sequences(x_test, maxlen=max_review_length)

print(x_train.shape)

# Model 1

# Embedding -> LSTM -> Dense(Output(Sigmoid))

# Build Model

embedding_vector_length = 32

model = Sequential()

model.add(Embedding(vocab_size, embedding_vector_length, input_length =max_review_length))

model.add(LSTM(100))

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

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

# Train Model

history = model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=128, epochs=5)

result = model.evaluate(x_test, y_test, verbose=0)

#Results

print("Test Score:", result[0])

print("Test Accuracy:", result[1])

print("Accuracy: %.2f%%" % (result[1]*100))

print(history.history.keys())
```

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
# Model 2
# Embedding -> LSTM -> Dense(Output(Sigmoid))
# Build Model
embedding_vector_length = 32
model = Sequential()
model.add(Embedding(vocab_size, embedding_vector_length, input_length =max_review_length))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='Adadelta', metrics=['accuracy'])
# Train Model
```

```
history = model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=128, epochs=5)

result = model.evaluate(x_test, y_test, verbose=0)

#Results

print("Test Score:", result[0])

print("Test Accuracy:", result[1])

print("Accuracy: %.2f%%" % (result[1]*100))

print(history.history.keys())

plt.plot(history.history['accuracy'])

plt.plot(history.history['val_accuracy'])

plt.title('Model accuracy')

plt.ylabel('Accuracy')

plt.xlabel('Epoch')

plt.legend(['Train', 'Test'], loc='upper left')

plt.show()

plt.plot(history.history['loss'])

plt.plot(history.history['val_loss'])

plt.title('Model loss')

plt.ylabel('Loss')

plt.xlabel('Epoch')

plt.legend(['Train', 'Test'], loc='upper left')

plt.show()

# Model 3

# Embedding -> LSTM -> Dense(Output(Sigmoid))

# Build Model
```

```
embedding_vector_length = 32

model = Sequential()

model.add(Embedding(vocab_size, embedding_vector_length, input_length =max_review_length))

model.add(LSTM(100))

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

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

# Train Model

history = model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=256, epochs=5)

result = model.evaluate(x_test, y_test, verbose=0)

#Results

print("Test Score:", result[0])

print("Test Accuracy:", result[1])

print("Accuracy: %.2f%%" % (result[1]*100))

print(history.history.keys())

plt.plot(history.history['accuracy'])

plt.plot(history.history['val_accuracy'])

plt.title('Model accuracy')

plt.ylabel('Accuracy')

plt.xlabel('Epoch')

plt.legend(['Train', 'Test'], loc='upper left')

plt.show()

plt.plot(history.history['loss'])

plt.plot(history.history['val_loss'])
```

```
plt.title('Model loss')

plt.ylabel('Loss')

plt.xlabel('Epoch')

plt.legend(['Train', 'Test'], loc='upper left')

plt.show()

# Model 4

# Embedding -> LSTM -> LSTM -> Dense(Output(Sigmoid))

# Build Model

embedding_vector_length = 32

model = Sequential()

model.add(Embedding(vocab_size, embedding_vector_length, input_length =max_review_length))

model.add(LSTM(100, return_sequences = True))

model.add(LSTM(100))

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

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

# Train Model

history = model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=128, epochs=5)

result = model.evaluate(x_test, y_test, verbose=0)

# Results

print("Test Score:", result[0])

print("Test Accuracy:", result[1])

print("Accuracy: %.2f%%" % (result[1]*100))

print(history.history.keys())
```

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
# Model 5
# Embedding -> LSTM -> Dropout -> LSTM -> Dropout -> Dense(Output(Sigmoid))
from keras.layers import Dropout
# Build Model
embedding_vector_length = 32
model = Sequential()
model.add(Embedding(vocab_size, embedding_vector_length, input_length =max_review_length))
model.add(LSTM(100, return_sequences=True))
model.add(Dropout(0.5))
model.add(LSTM(100))
```

```
model.add(Dropout(0.5))

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

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

# Train Model

history = model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=128, epochs=5)

result = model.evaluate(x_test, y_test, verbose=0)

# Results

print("Test Score:", result[0])

print("Test Accuracy:", result[1])

print("Accuracy: %.2f%%" % (result[1]*100))

print(history.history.keys())

plt.plot(history.history['accuracy'])

plt.plot(history.history['val_accuracy'])

plt.title('Model accuracy')

plt.ylabel('Accuracy')

plt.xlabel('Epoch')

plt.legend(['Train', 'Test'], loc='upper left')

plt.show()

plt.plot(history.history['loss'])

plt.plot(history.history['val_loss'])

plt.title('Model loss')

plt.ylabel('Loss')

plt.xlabel('Epoch')

plt.legend(['Train', 'Test'], loc='upper left')
```

```
plt.show()

# Model 6

# Embedding -> LSTM -> Dense(activation:Relu) -> Dense(Output(Sigmoid))

# Build Model

embedding_vector_length = 32

model = Sequential()

model.add(Embedding(vocab_size, embedding_vector_length, input_length =max_review_length))

model.add(LSTM(100))

model.add(Dense(64, kernel_initializer='uniform', activation='relu'))

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

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

# Train Model

history = model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=128, epochs=5)

result = model.evaluate(x_test, y_test, verbose=0)

# Results

print("Test Score:", result[0])

print("Test Accuracy:", result[1])

print("Accuracy: %.2f%%" % (result[1]*100))

print(history.history.keys())

plt.plot(history.history['accuracy'])

plt.plot(history.history['val_accuracy'])

plt.title('Model accuracy')

plt.ylabel('Accuracy')
```

```
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
# Model 7
# Embedding -> LSTM -> Dense(activation:Softmax) -> Dense(Output(Sigmoid))
# Build Model
embedding_vector_length = 32
model = Sequential()
model.add(Embedding(vocab_size, embedding_vector_length, input_length =max_review_length))
model.add(LSTM(100))
model.add(Dense(64, kernel_initializer='uniform', activation='softmax'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Train Model
history = model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=128, epochs=5)
result = model.evaluate(x_test, y_test, verbose=0)
```

```
# Results

print("Test Score:", result[0])

print("Test Accuracy:", result[1])

print("Accuracy: %.2f%%" % (result[1]*100))

print(history.history.keys())

plt.plot(history.history['accuracy'])

plt.plot(history.history['val_accuracy'])

plt.title('Model accuracy')

plt.ylabel('Accuracy')

plt.xlabel('Epoch')

plt.legend(['Train', 'Test'], loc='upper left')

plt.show()

plt.plot(history.history['loss'])

plt.plot(history.history['val_loss'])

plt.title('Model loss')

plt.ylabel('Loss')

plt.xlabel('Epoch')

plt.legend(['Train', 'Test'], loc='upper left')

plt.show()
```

3. Results and Discussion

Comparison of Optimization Algorithm:

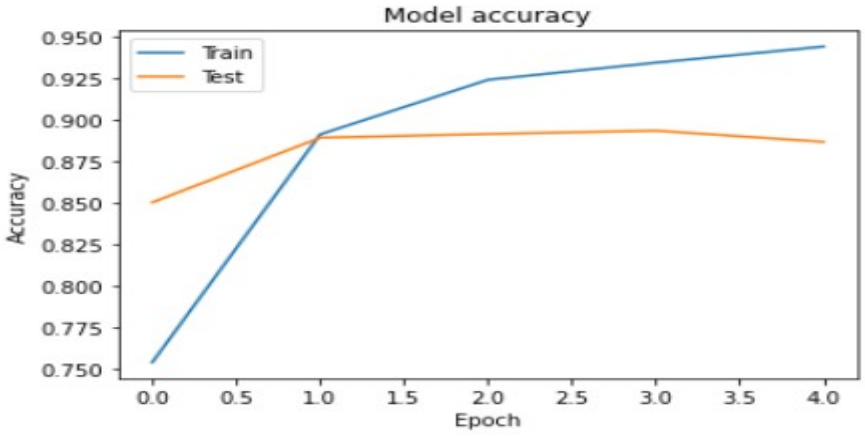
Optimization methods are used to find the optimum value for the solution of nonlinear problems. Optimization algorithms such as stochastic gradient descent, adagrad, adadelta, adam, adamax are widely used in deep learning applications. These algorithms differ in performance and speed. Model 1's optimization algorithm is Adam and Model 2's optimization algorithm is Adadelta.

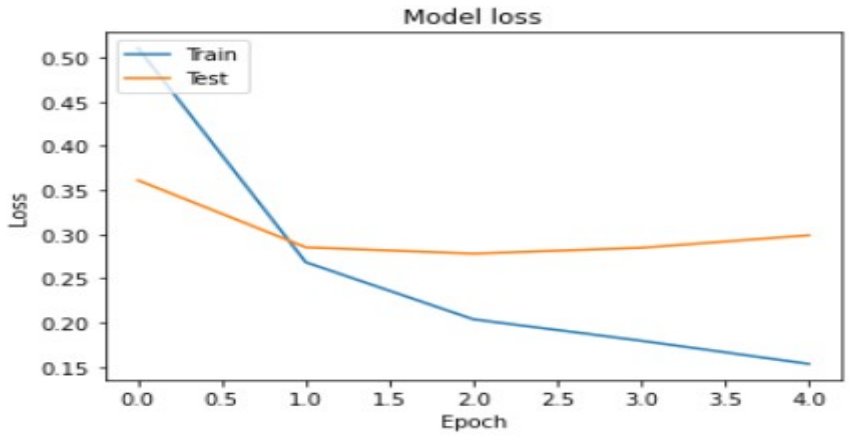
Result of Model 1:

Test Score: 0.2989017186164856

Test Accuracy: 0.8870000243186951

Accuracy: 88.70%



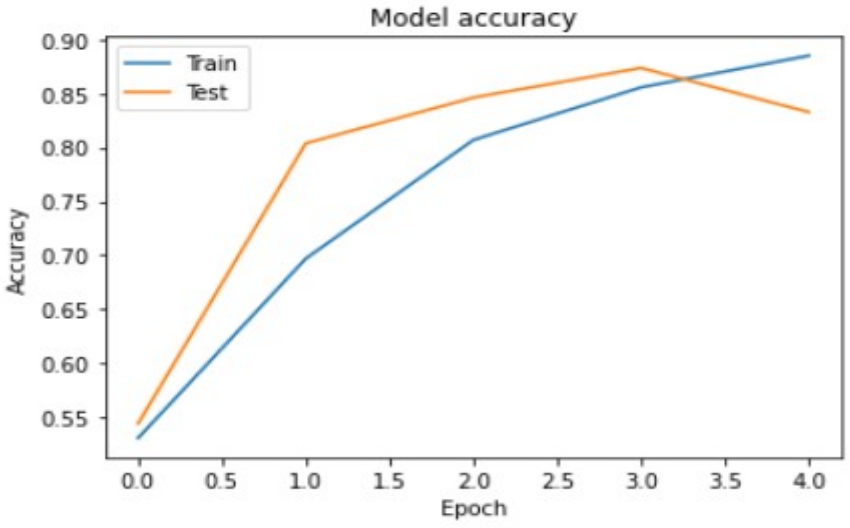


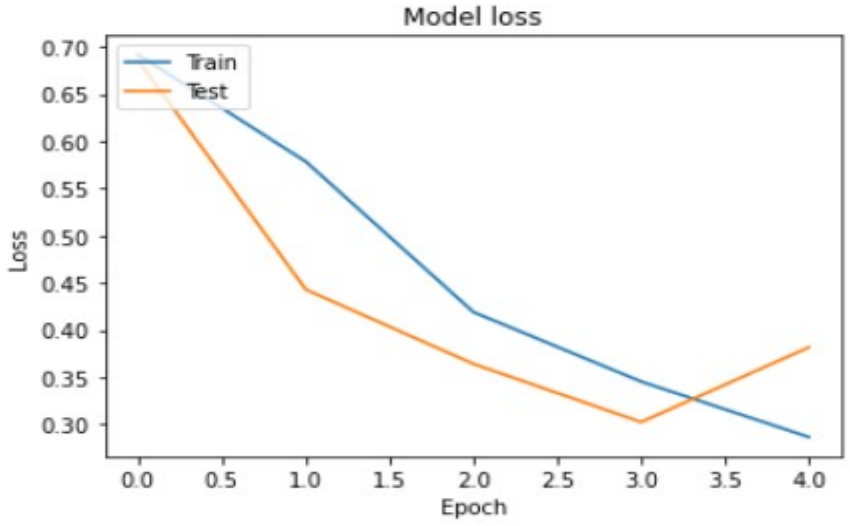
Result of Model 2 :

Test Score: 0.381622323513031

Test Accuracy: 0.8327500224113464

Accuracy: 83.28%





The final accuracy value of Model 1, which uses Adam as an optimization algorithm, is higher than the accuracy of the Model 2 which uses Adadelta as an optimization algorithm. An overfit model is one where performance on the train set is good and continues to improve, whereas performance on the validation set improves to a point and then begins to degrade.

This can be diagnosed from a plot where the train loss slopes down and the validation loss slopes down, hits an inflection point, and starts to slope up again. [6] Model 1 has progressed properly, but I observe that Model 2 is overfitting. As a result, the Adam optimization algorithm gave a better result.

Comparison of Batch Size :

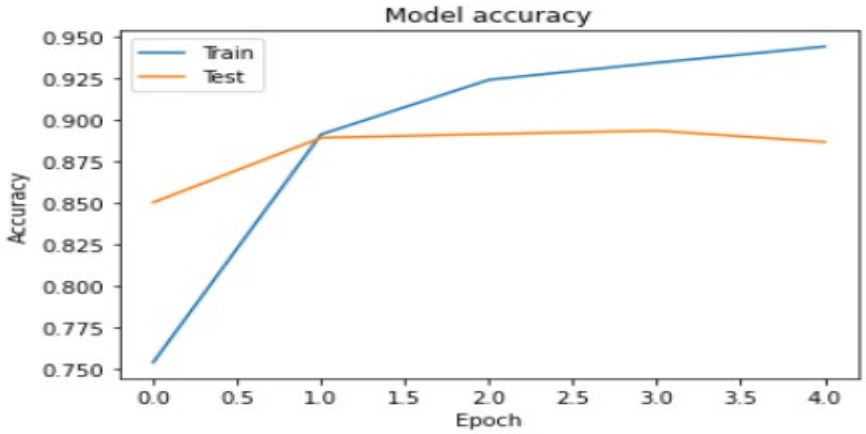
Batch size is a term used in machine learning and refers to the number of training examples utilized in one iteration. Model 1 and Model 3 are the same except batch_sizes. Model 1's batch size is 128 and Model 3's batch size is 256.

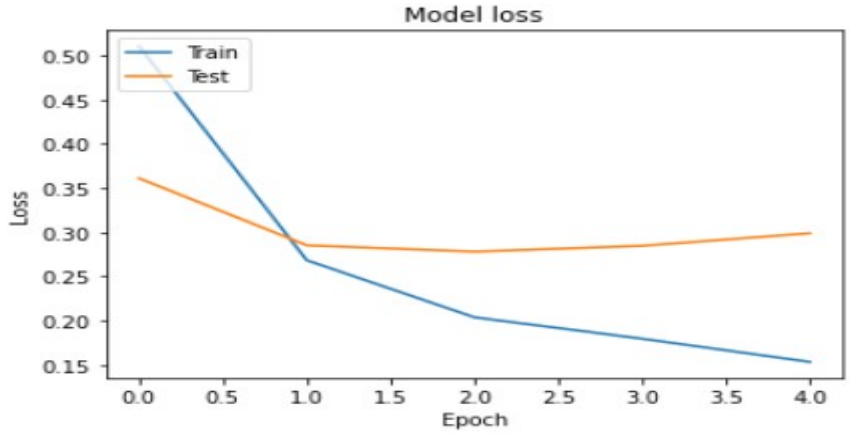
Result of Model 1:

Test Score: 0.2989017186164856

Test Accuracy: 0.8870000243186951

Accuracy: 88.70%



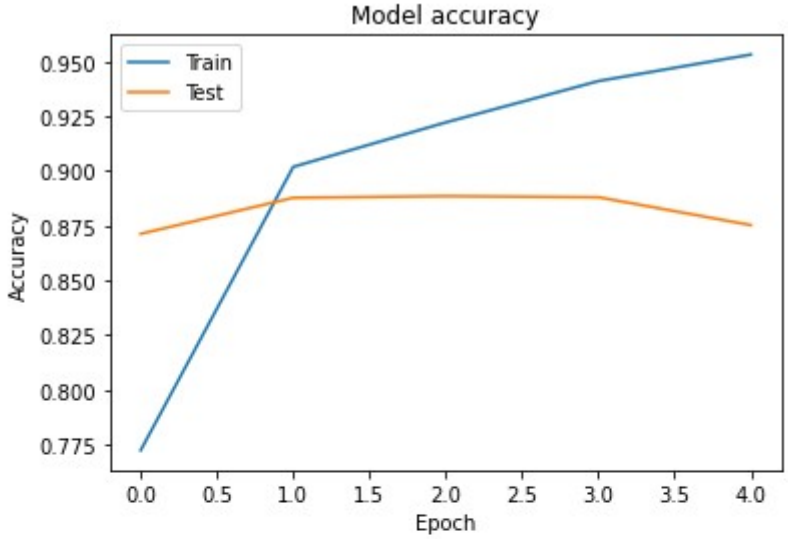


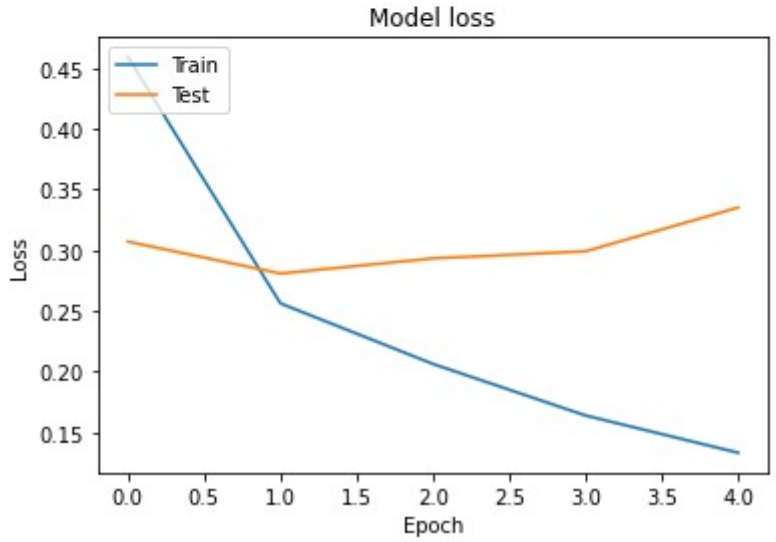
Results of Model 3:

Test Score: 0.2942531524002552

Test Accuracy: 0.8859999775886536

Accuracy: 88.60%





There is not a huge difference in accuracy value between them, but 128 batch size with the highest accuracy value is Model 1. For the test set, the loss value of Model 1 decreased continuously, but the loss value of Model 2 increased towards the end. It

is not a situation that is desired to increase loss value. Based on this comparison, small batch_sizes give better results.

Comparison of Single LSTM Layer and Multi LSTM Layer:

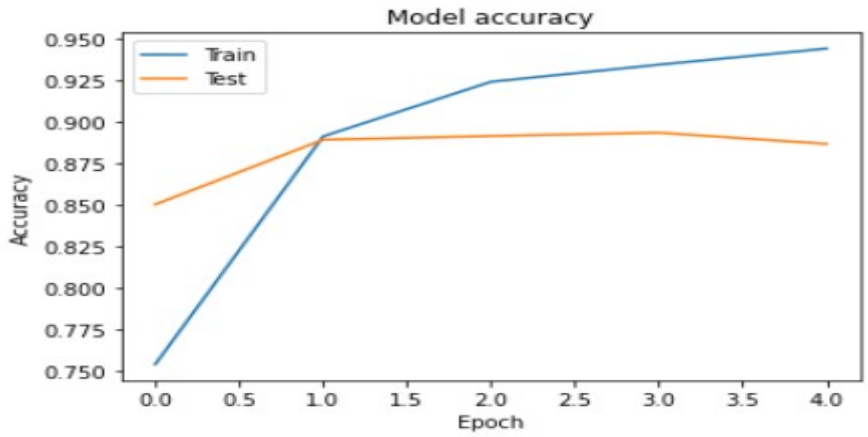
Model 1 has one LSTM layer, Model 4 has two LSTM layers.

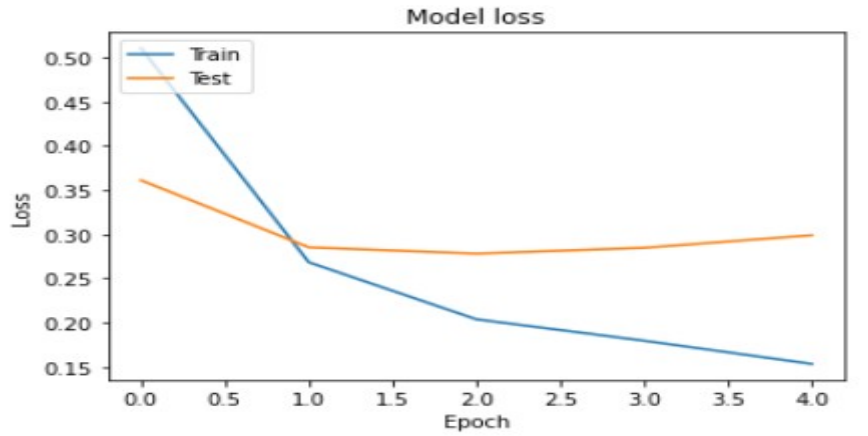
Result of Model 1:

Test Score: 0.2989017186164856

Test Accuracy: 0.8870000243186951

Accuracy: 88.70%



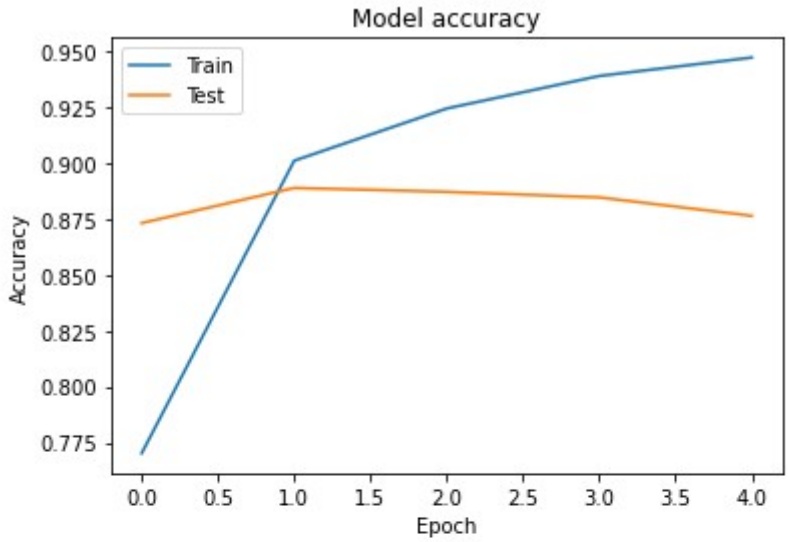


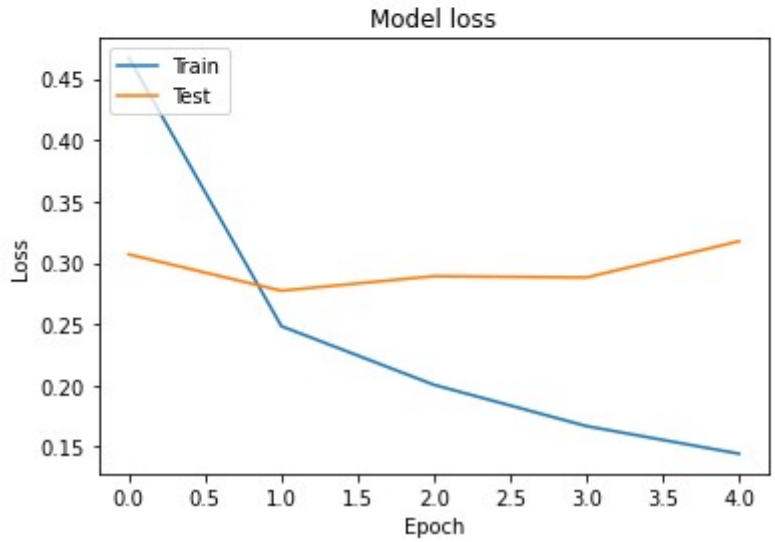
Results of Model 4:

Test Score: 0.31750876614451407

Test Accuracy: 0.8765000104904175

Accuracy: 87.65%





I would expect the accuracy value of the model using the Mutliple LSTM layer to be higher. In this study[7], multiple lstm layer shows better performance than single lstm layer. But my study was not like that either. The model using the single LSTM layer gave better results.

Effect of Dropout Layer:

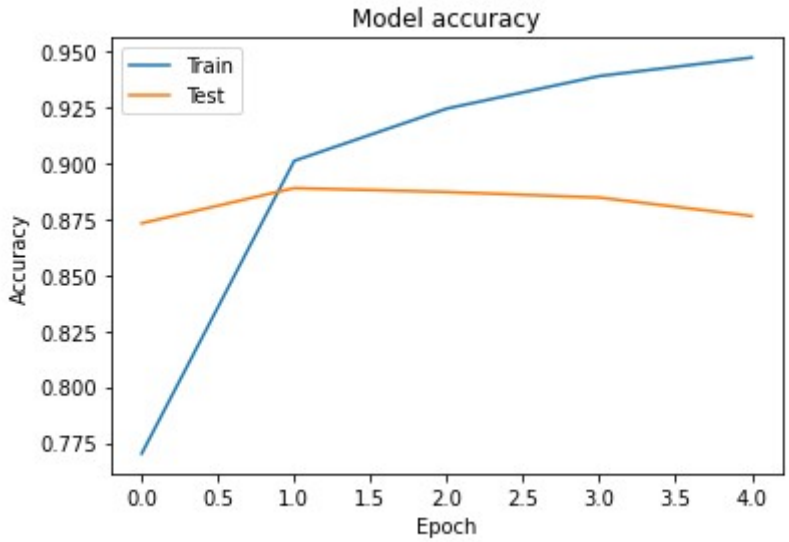
Model 4 consists of two LSTM layers and no dropout value or dropout layer. Model 5 consists of two LSTM layers and each dropout layer after each LSTM layer.

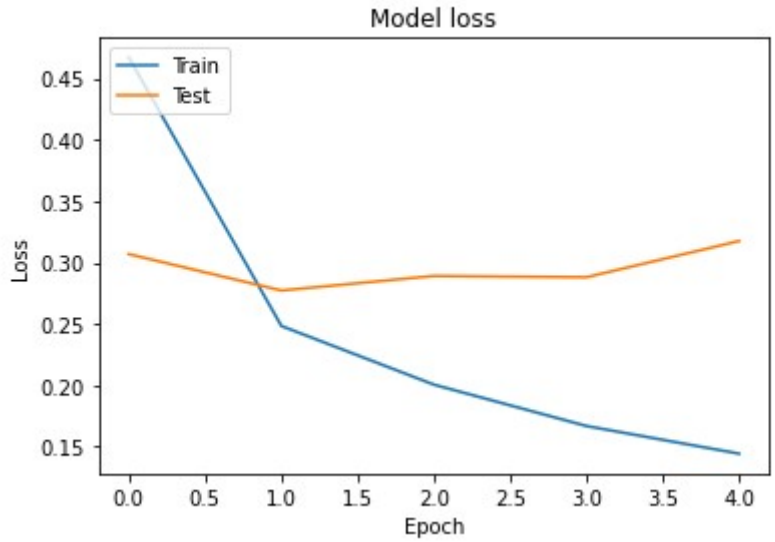
Results of Model 4:

Test Score: 0.31750876614451407

Test Accuracy: 0.8765000104904175

Accuracy: 87.65%



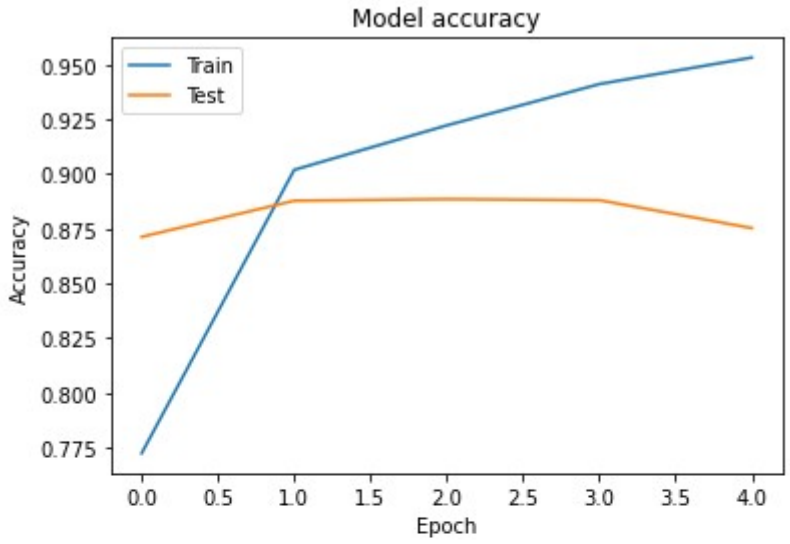


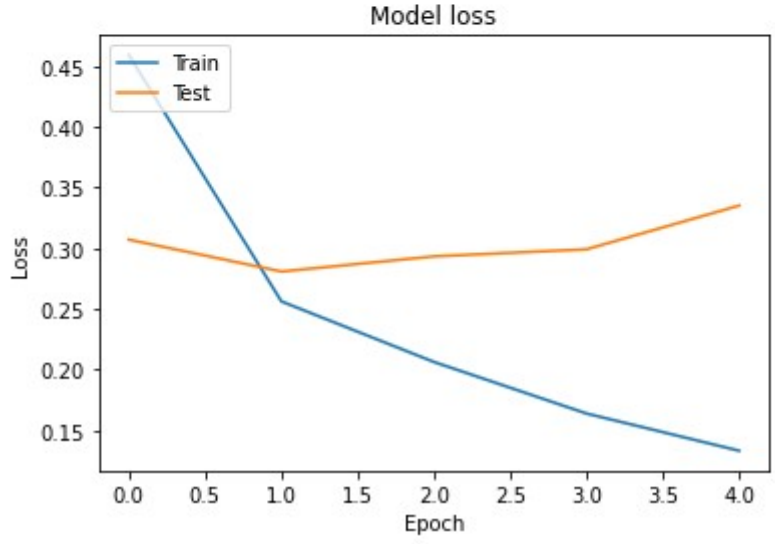
Results of Model 5:

Test Score: 0.33502410554885864

Test Accuracy: 0.875249981880188

Accuracy: 87.52%





Dropout layer prevents the model from overfitting. Since the model 1 that I did not use dropout in my study was not overfit, my dropout layer use had no effect.

Comparison of Activation Function :

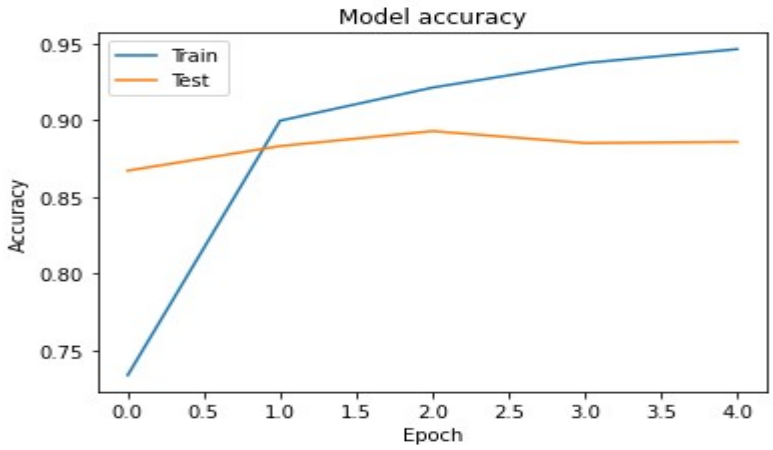
Model 6 used relu activation function and model 7 used softmax activation function. Generally, softmax is used in the final layer of neural networks for final classification in feedforward neural networks. The ReLU function has also become popular lately. Its activation is simply thresholded at zero when the input is less than 0. Compared with the sigmoid function and the tanh function, ReLU is easy to compute, fast to converge in training and yields equal or better performance in neural networks.[8]

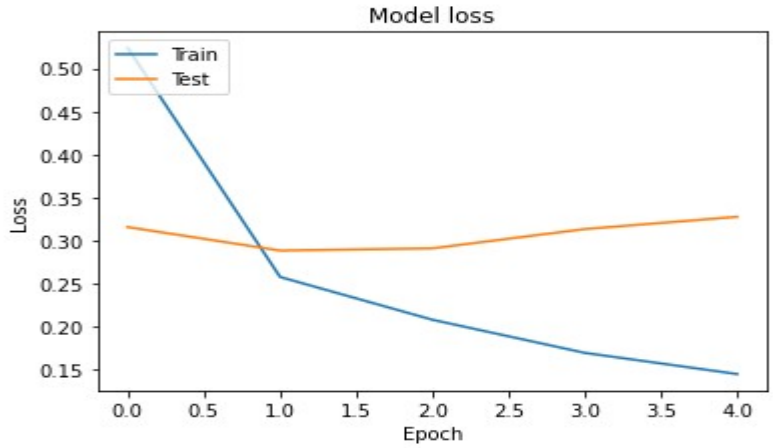
Result of Model 6:

Test Score: 0.32745230266451836

Test Accuracy: 0.8859999775886536

Accuracy: 88.60%



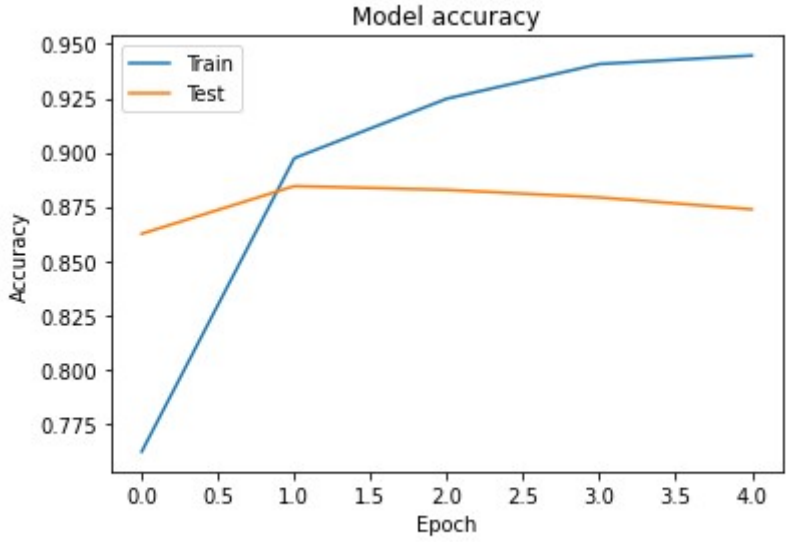


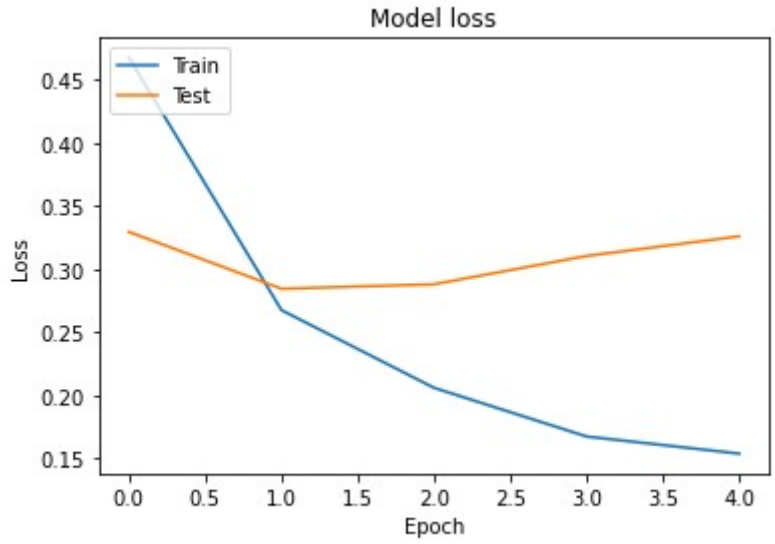
Result of Model 7:

Test Score: 0.541119369983673

Test Accuracy: 0.8352500200271606

Accuracy: 83.53%





When we created separate models with Relu and softmax activation functions, model 6 which have relu activation function gave higher accuracy value. Model 7 has

a high loss value, and this is not desired. Relu activation function showed better success than softmax.

As a result, in my study, effect of dropout layer, comparison of single lstm layer and multi lstm layer, comparion of batch size, comparison of optimization algorithm, comparison of activation function was made.

4.References

[1] Amazon Fine Food Data Set, https://www.kaggle.com/snap/amazon-fine-food-reviews

[2] Liu B. Sentiment analysis: mining opinions, sentiments, and emotions. The Cambridge University Press, 2015.

[3] Tan W., Wang X., Xu X., “Sentiment Analysis for Amazon Reviews”.

[4] Zhang, L, Wang, S, Liu, B. “Deep learning for sentiment analysis: A survey”. WIREs Data Mining Knowl Discov. 2018; 8:e1253.

[5] Dang, Cach & Moreno García, María & De La Prieta, Fernando. (2020). S:entiment Analysis Based on Deep Learning: A Comparative Study”. Electronics. 9. 483.

[6] https://machinelearningmastery.com/diagnose-overfitting-underfitting-lstm-models/

[7] Salman, A.G., Heryadi, Y., Abdurahman, E., & Suparta, W. (2018). Single Layer & Multi-layer Long Short-Term Memory (LSTM) Model with Intermediate Variables for Weather Forecasting. *Procedia Computer Science, 135*, 89-98.

[8] Glorot X, Bordes A, Bengio Y. Deep sparse rectifier neural networks. In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS 2011), 2011.