Generating Quotes from The Office show

- Name: Akshara Shukla
- Class: AI7

Introduction

For the first ANN exercise for this semester, I will be building an AI that will generate quotes using the hit The Office sitcom. The inspiration for this assignment comes after rewatching the show multiple times and it has been a project that I have been personally been waiting to try. In this notebook, the following steps will be described:

- Preparing the data,
- Analysing and visualising the data,
- Cleaning the data,
- Selecting features,
- Apply some callback functions
- Training your Machine learning algorithm,
- Applying the machine learning algorithm
- Evaluating its results
- and, generating new quotes

Firstly, lets import the required libraries.

In [1]:

```
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
from wordcloud import WordCloud
import re

from keras.preprocessing.text import Tokenizer
import keras.utils as ku
from keras import layers
from keras.preprocessing.sequence import pad_sequences
```

```
from keras.models import Sequential, Model
from keras.layers.embeddings import Embedding
from keras.models import model_from_json
from keras.layers import Input, Activation, Dense, Dropout
from keras.layers import LSTM, Bidirectional
from keras.callbacks import EarlyStopping


print('numpy version:', np.__version__)
print('pandas version:', pd.__version__)
print('matplotlib version:', sns.__version__)

%matplotlib inline
```

```
C:\Users\PC\anaconda3\lib\site-packages\numpy\_distributor_init.py:30: UserWarning: loaded more than 1 DLL from .libs:
C:\Users\PC\anaconda3\lib\site-packages\numpy\.libs\libopenblas.PYQHXLVVQ7VESDPUVUADXEVJOBGHJPAY.gfortran-win_amd64.dll
C:\Users\PC\anaconda3\lib\site-packages\numpy\.libs\libopenblas.WCDJNK7YVMPZQ2ME2ZZHJJRJ3JIKNDB7.gfortran-win_amd64.dll
  warnings.warn("loaded more than 1 DLL from .libs:\n%s" %
numpy version: 1.18.5
pandas version: 1.1.4
matplotlib version: 0.11.1
```

Preparing the data

For this, we will be using this dataset from kaggle. You can access it here. It includes all the lines spoken from each character in all the episodes throughout the course of the running of the show.

In [2]:

```
df = pd.read_csv('the-office_lines.csv')
```

In [3]:

```
df.head()
```

Out[3]:

	Unnamed: 0	Character	Line	Season	Episode_Number
0	0	Michael	All right Jim. Your quarterlies look very goo...	1	1
1	1	Jim	Oh, I told you. I couldn't close it. So...	1	1
2	2	Michael	So you've come to the master for guidance? Is...	1	1
3	3	Jim	Actually, you called me in here, but yeah.	1	1

	Unnamed: 0	Character	Line	Season	Episode_Number
4	4	Michael	All right. Well, let me show you how it's don...	1	1

Analysing the data

In [4]:
```
df = df.loc[:,('Character','Line')]
df.rename(columns = {'Line':'Quotes'}, inplace = True)
```

In [5]:
```
def exploratory_data_analysis(df):
    '''
    Function exploratory_data_analysis: This gives the data types, shape and unique characters that are present in the da
    Parameters:
    a) df: Name of the dataset
    '''
    info = df.info()
    shape = df.shape
    unique_characters = len(df['Character'].unique())
    print(f'There are a total of {unique_characters} characters present in the show who say something.')

    return info, shape, unique_characters
```

In [6]:
```
exploratory_data_analysis(df)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 58721 entries, 0 to 58720
Data columns (total 2 columns):
 #   Column     Non-Null Count  Dtype 
---  ------     --------------  ----- 
 0   Character  58721 non-null  object
 1   Quotes     58721 non-null  object
dtypes: object(2)
memory usage: 917.6+ KB
There are a total of 780 characters present in the show who say something.
```

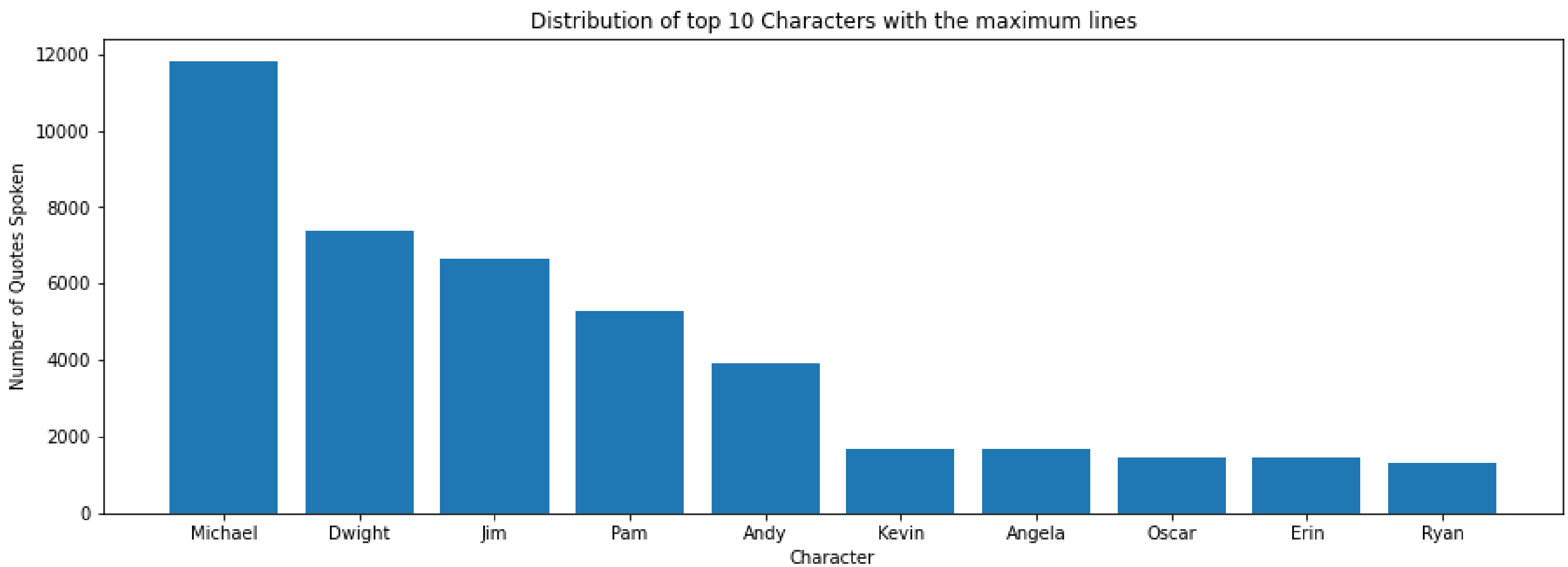
Out[6]:
```
(None, (58721, 2), 780)
```

In [7]:
```
max_lines_spoken = df.groupby('Character').agg({'Quotes':'count'}).reset_index().sort_values('Quotes',ascending=False)[:1
```

```
fig=plt.figure(figsize=(15, 5))
plt.bar(max_lines_spoken['Character'], max_lines_spoken['Quotes']);
plt.xlabel('Character')
plt.ylabel('Number of Quotes Spoken')
plt.title('Distribution of top 10 Characters with the maximum lines')

plt.plot()
```

Out[7]: []



In [8]:

```
#We are going to replace all the special characters with a ''.
#The ^0-9A-Za-z expression includes all the values except 0-9 and unnecesary alphabetic. Therefore, keeping only the alph
#initially.

df['Quotes']= df['Quotes'].str.replace(r'[^0-9A-Za-z ,\"\',]+', '')
df.head()
```

Out[8]:

	Character	Quotes
0	Michael	All right Jim Your quarterlies look very good...
1	Jim	Oh, I told you I couldnt close it So

	Character	Quotes
2	Michael	So youve come to the master for guidance Is t...
3	Jim	Actually, you called me in here, but yeah
4	Michael	All right Well, let me show you how its done

In [9]:

```
fig=plt.figure(figsize=(20, 6))

# Creating word_cloud with text as argument in .generate() method
text3 = ' '.join(df['Quotes'])
wordcloud = WordCloud(collocations = False, background_color = 'white').generate(text3)
# Display the generated Word Cloud
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.show()
```

EDA Analysis

From our EDA analysis, we can conclude that `Michael` was the character with the most lines spoken in the season. While, the top most said word are Im, Oh, Know, Michael, Thats and so on as they as visible in the wordcloud generated above. So far, I would agree with the results since these words are common but so is the character.

Preparing the Data

In order to prepare our text data to pass through the model, we need to perform tokenization. It is the process of extracting tokens such as words from a corpus. From our EDA results, we will be training the model for the character Michael as he has the most spoken words. Due to memory issues, I will be using a random sample of 3000 rows.

In [10]:

```
df.loc[df['Character'].isin(['Michael'])]
quotes = df['Quotes'].drop_duplicates()
quotes = quotes.sample(3000)
quotes = list(quotes)
quotes[:1]
```

Out[10]:

```
[' Mmhm Im a magical fairy who floated into your office to bring a little bit of magic into your lives, to give you all r
aises  ']
```

For tokenization, I will be using the keras tokenizer. This is because during my internship I have used the BERT tokenizer and now I want to analyse how this one works with textual data.

It first uses *.fit_on_texts()* to create a dictionary of the unique words in the data and stores its frequencies. Based on the index value, it assigns an integer value which is also used for padding the sentence.

Then, it uses *.texts_to_sequences()* which is passed through each line in the generated corpus and converts them to a sequences of integer values which can be used as the input to our model.

In [11]:

```
# Tokenization
tokenizer = Tokenizer()

# Function to create the sequences
def create_sequences(corpus):
    '''
    Function create_sequences: Takes in the quotes data and tokenizes them to make ready to be the input to our model.
    Parameters:
    a) corpus: This is cleaned quotes column from our dataset
    '''
```

```
    #tokenizing the individual texts
    tokenizer.fit_on_texts(corpus)
    total_words = len(tokenizer.word_index) + 1 #+1 because token ids start from 0
    print(f"Total unique words in our generated corpus: {total_words}")

    #now converting the corpus into a flat values that can be used as an input to our model
    input_sentences = []
    for line in corpus:
        gen_sequences = tokenizer.texts_to_sequences([line])[0]
        for i in range(1, len(gen_sequences)):
            ngram_seq = gen_sequences[:i+1]
            input_sentences.append(ngram_seq)

    return input_sentences, total_words

# Looking at the first sequence of the sentence
input_sentences, total_words = create_sequences(quotes)
input_sentences[:5]
```

```
Total unique words in our generated corpus: 4938
```

Out[11]:

```
[[1394, 17],
 [1394, 17, 5],
 [1394, 17, 5, 2106],
 [1394, 17, 5, 2106, 2107],
 [1394, 17, 5, 2106, 2107, 95]]
```

Now, that we have the data in unique integer values, it's required to pad them i.e., making each input_sequence with same lenth. I will pad both the sentences and it's predictor using the *pad_sequence* function from keras library.

''' Now, since we are working on text generation, we require the terms predictors and labels. To understand this, for instance we have a sentence: Sentence = 'Today we have a workshop' The predictors | The Label (next predicted work) Today | we Today we | have Today we have | a Today we have a | lecture '''

In [12]:

```
# Generating predictors and labels from the padded sequences
def input_data_values(input_sentences):
    '''
    Function input_data_values: Takes in the array of sentences and makes them all of one length. Also, extracts the pred
    Parameters:
    a) input_sentences: This is the output of the previous function i.e., the array of sentences.
    '''
    maxlen = max([len(x) for x in input_sentences])
    input_sentences = pad_sequences(input_sentences, maxlen=maxlen)

    #extracting the predictors and labels from our input sequences
    predictors, label = input_sentences[:, :-1], input_sentences[:, -1]
```

```
    #making the next predicted word in the sequence to a binary matrix
    #this is vital for assigning number for next work prediction
    label = ku.to_categorical(label, num_classes=total_words)
    return predictors, label, maxlen

predictors, label, maxlen = input_data_values(input_sentences)
predictors[:1], label[:1], maxlen
```

Out[12]:

```
(array([[   0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0, 1394]]),
 array([[0., 0., 0., ..., 0., 0., 0.]], dtype=float32),
 161)
```

Callbacks

I will be using the *earlystop* callback function. It will monitor the 'loss' value per each epoch and if the difference between the losses is 0 for next 4 epochs, it will stop the training. This helps in solving the issue of overfitting.

In [13]:

```
#EarlyStop Callback
earlystop = EarlyStopping(monitor = 'loss',
                          min_delta = 0,    #training process will be stopped if the absolute change of loss is less than
                          patience = 4,
                          verbose = 1,
                          restore_best_weights=True)
```

Fine-tuning the Hyperparameters

Since our quote generator is similar to the process of text generation, I will be using the Long - Short Term Memory model. It's a recurrent neural network model (RNN) that is a type of Artificial Neural Networks that works on the principle of gathering data and storing it as memory.

It's a type of ANN as it processes the data similar as to the human brain and processes the data in both ways.

The neurons or the activation ouputs are transferred firstly from the the inputs to outputs and then from outputs to inputs. Due to this double checking, it gives the model the ability to remember and store all the data points it has learned so far. This can be also associated as backpropagation. It's got three important layers:

1. Input Layer : Takes the sequence of words as input
2. LSTM Layer : Computes the output using LSTM units. We will be experimenting with this number in our hyper-parameter tuning.
3. Dropout Layer : A regularisation layer which randomly turns-off the activations of some neurons in the LSTM layer. It helps in preventing over fitting.
4. Output Layer : Computes the probability of the best possible next word as output

In [16]:

```
def model_tune(embedding_dim,neurons,dropout,optimizer):
'''
Function model_tune: This function takes in specific parameters which will be used to train our LSTM model.
Parameters:
The parameters are described below with the two values which will be used for training.
'''

    emdedding_dim = embedding_dim
    model = Sequential()

    #the input embedding layer
    model.add(layers.Embedding(total_words, embedding_dim, input_length = maxlen))

    #the LSTM and one hidden layer
    model.add(layers.LSTM(neurons, dropout = dropout))
    model.add(Dropout(0.2))

    #the output layer which shall give us the next word in the sentence
    model.add(layers.Dense(total_words,activation = 'softmax'))

    #compiling the model
    model.compile(loss = 'categorical_crossentropy', optimizer = optimizer)
    model.summary()
```

```
    #fitting and training
    model.fit(predictors, label,
              epochs=10,
              batch_size=64,
              callbacks = earlystop)

    #calculating the lost
    print(model.evaluate(predictors, label))

    return model
```

The function built above `model_tune`, trains our model with different values for the following hyperparameters. It gives us the loss value per epoch and the total average loss value for the next predicted word.

1. **Embedding_dim**: This parameter is the total or maximum number of feature dimensions that the input will be designed for. This is important because it makes sure that all the input sentences/word in our corpus is of the same length or dimension. To determine this number, the formula used is from this link
2. **Neurons**: The neurons are the number of LSTM units that will be learning from the data and transfering to the next layer. We will be experimentng with 100 and 128.
3. **Dropout**: This is the rate at which the model will randomly shut or ignored the output features. According to this link, the optimal value for the dropout rate is between 0.5 and 0.8. We will be experimenting with two values: 0.2 and 0.5
4. **Optimizer**: This is the value that is reponsible for the balanced learning of the model. It balances the overall weights and learning rate. Therefore, it helps in making sure the loss value decreases and improves the overall accuracy. According to think link, it suggests that *adam* is the most optimal optimizer but for the interest, I'll try it out with rmsprop as well.

Training the Models

In [17]:

```
#Adam(lr=0.001, decay=1e-6)
#The number 113 comes after applying the formula linked in the above chunck for the embedding_dim with the shape of 5066
model1 = model_tune(113,128,0.2,'adam')
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
embedding_1 (Embedding)     (None, 161, 113)          557994

_________________________________________________________________
lstm_1 (LSTM)               (None, 128)               123904

_________________________________________________________________
```

```
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 4938)              637002
=================================================================
Total params: 1,318,900
Trainable params: 1,318,900
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
WARNING:tensorflow:Model was constructed with shape (None, 161) for input KerasTensor(type_spec=TensorSpec(shape=(None, 1
61), dtype=tf.float32, name='embedding_1_input'), name='embedding_1_input', description="created by layer 'embedding_1_in
put'"), but it was called on an input with incompatible shape (None, 160).
WARNING:tensorflow:Model was constructed with shape (None, 161) for input KerasTensor(type_spec=TensorSpec(shape=(None, 1
61), dtype=tf.float32, name='embedding_1_input'), name='embedding_1_input', description="created by layer 'embedding_1_in
put'"), but it was called on an input with incompatible shape (None, 160).
525/525 [==============================] - 60s 110ms/step - loss: 7.0028
Epoch 2/10
525/525 [==============================] - 61s 116ms/step - loss: 6.3970
Epoch 3/10
525/525 [==============================] - 61s 116ms/step - loss: 6.2104
Epoch 4/10
525/525 [==============================] - 61s 116ms/step - loss: 5.8888
Epoch 5/10
525/525 [==============================] - 61s 117ms/step - loss: 5.6427
Epoch 6/10
525/525 [==============================] - 62s 118ms/step - loss: 5.4511
Epoch 7/10
525/525 [==============================] - 61s 116ms/step - loss: 5.2515
Epoch 8/10
525/525 [==============================] - 62s 117ms/step - loss: 5.0439
Epoch 9/10
525/525 [==============================] - 64s 121ms/step - loss: 4.8726
Epoch 10/10
525/525 [==============================] - 62s 118ms/step - loss: 4.7127
WARNING:tensorflow:Model was constructed with shape (None, 161) for input KerasTensor(type_spec=TensorSpec(shape=(None, 1
61), dtype=tf.float32, name='embedding_1_input'), name='embedding_1_input', description="created by layer 'embedding_1_in
put'"), but it was called on an input with incompatible shape (None, 160).
1049/1049 [==============================] - 26s 25ms/step - loss: 4.4021
4.402070999145508
```

In [18]:

```
#Here, I have used values of 100 just to counteract the values used above
model2 = model_tune(100,100,0.5,'RMSprop')
```

```
Model: "sequential_2"
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 161, 100)          493800
_________________________________________________________________
lstm_2 (LSTM)                (None, 100)               80400
_________________________________________________________________
dropout_2 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_2 (Dense)              (None, 4938)              498738
=================================================================
Total params: 1,072,938
Trainable params: 1,072,938
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
WARNING:tensorflow:Model was constructed with shape (None, 161) for input KerasTensor(type_spec=TensorSpec(shape=(None, 1
61), dtype=tf.float32, name='embedding_2_input'), name='embedding_2_input', description="created by layer 'embedding_2_in
put'"), but it was called on an input with incompatible shape (None, 160).
WARNING:tensorflow:Model was constructed with shape (None, 161) for input KerasTensor(type_spec=TensorSpec(shape=(None, 1
61), dtype=tf.float32, name='embedding_2_input'), name='embedding_2_input', description="created by layer 'embedding_2_in
put'"), but it was called on an input with incompatible shape (None, 160).
525/525 [==============================] - 57s 105ms/step - loss: 6.8640
Epoch 2/10
525/525 [==============================] - 58s 111ms/step - loss: 6.4284
Epoch 3/10
525/525 [==============================] - 58s 110ms/step - loss: 6.3613
Epoch 4/10
525/525 [==============================] - 55s 105ms/step - loss: 6.2484
Epoch 5/10
525/525 [==============================] - 54s 103ms/step - loss: 6.2123
Epoch 6/10
525/525 [==============================] - 54s 102ms/step - loss: 6.1458
Epoch 7/10
525/525 [==============================] - 54s 103ms/step - loss: 6.0971
Epoch 8/10
525/525 [==============================] - 55s 104ms/step - loss: 6.0237
Epoch 9/10
525/525 [==============================] - 55s 105ms/step - loss: 5.9907
Epoch 10/10
525/525 [==============================] - 54s 103ms/step - loss: 5.9602
WARNING:tensorflow:Model was constructed with shape (None, 161) for input KerasTensor(type_spec=TensorSpec(shape=(None, 1
61), dtype=tf.float32, name='embedding_2_input'), name='embedding_2_input', description="created by layer 'embedding_2_in
put'"), but it was called on an input with incompatible shape (None, 160).
```

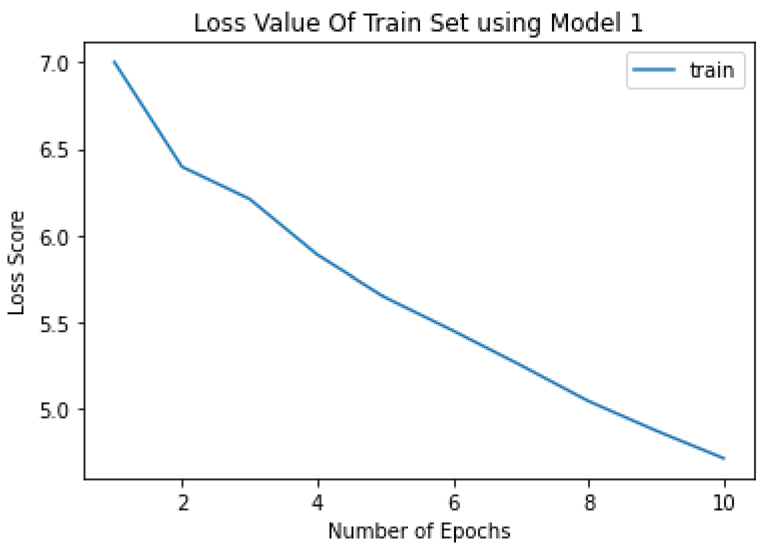
```
1049/1049 [==============================] - 19s 17ms/step - loss: 5.8077
5.807661056518555
```

Model Evaluation

For the model evaluation, according to the research link mentioned belo and specifically this, the optimal measure to evaluate a natural language generation model is human interpretibility. But, I will also be calculating the how the loss value varies over each epoch trained.

In [41]:

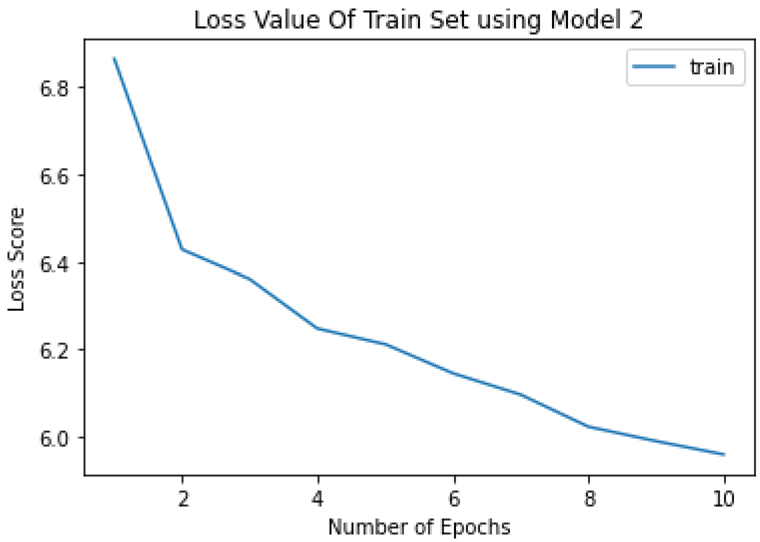
```
x = [1,2,3,4,5,6,7,8,9,10]
y = [7.0028,6.3970,6.2104,5.8888,5.6427,5.4511,5.2515,5.0439,4.8726,4.7127]
plt.plot(x,y)
plt.title('Loss Value Of Train Set using Model 1')
plt.xlabel('Number of Epochs')
plt.ylabel('Loss Score')
plt.legend(['train'], loc='upper right')
plt.show()
```



In [42]:

```
x = [1,2,3,4,5,6,7,8,9,10]
y= [6.8640,6.4284,6.3613,6.2484,6.2123,6.1458,6.0971,6.0237,5.9907,5.9602]
plt.plot(x,y)
plt.title('Loss Value Of Train Set using Model 2')
plt.xlabel('Number of Epochs')
plt.ylabel('Loss Score')
plt.legend(['train'], loc='upper right')
```

```
plt.show()
plt.show()
```



`Remark` : With the loss value graphs for both of the graphs, it can be concluded that the first model performed better with having loss value of 4.4. This value means, the model is prone to give about 4.4 in summary of errors when generating the next word in the quote. Therefore, this model will be used to generate quotes.

Generating new Quotes

In [54]:

```
def generate_text(first_word, next_word, modelname):
    '''
    Function generate_text: It takes the input which is the phrase giving by the user. Then, it generates the quote match

    Parameters:
    a) first_word (str): This is the phrase which is given as the input from the user.
    b) next_word (int): Total number of words that will be added to the first_word.
    c) modelname (var): This is the model which will generate the entire quote.

    '''

    #tokenizing and padding the predicted each next words by the model chosen
    #padding = 'pre' since I want the actual values that were remembers by the LSTM at the last layers.
    for _ in range(next_word):
        create_token = tokenizer.texts_to_sequences([first_word])[0]
```

```
        create_token = pad_sequences([create_token], maxlen= maxlen, padding = 'pre')

        #passing it through the modelname selected and getting the predictions for the next word in the quote generated
        output = modelname.predict_classes(create_token, verbose = 0)

        output_word = ""

        #create the output which is a combination of enterered word(s) and the output generated by the model
        #.word_index.items() finds the total length and stores the unique tokens
        # matches the index value with the output and predicts it
        for word, index in tokenizer.word_index.items():
            if index == output:
                output_word = word
                break
        first_word += " "+output_word
    return first_word.title()
```

In [55]:

```
generate_text("Follow",10,model1)
```

Out[55]:

```
'Follow The Phone Oh My God Oh My God Oh My'
```

In [49]:

```
generate_text("Follow", 15, model2, maxlen)
```

Out[49]:

```
'Follow You Know What I Dont Know What I Dont Know What I Dont Know What'
```

In [50]:

```
generate_text("This semester", 12, model1, maxlen)
```

Out[50]:

```
'This Semester Is A Lot Of The Office I Was A Little Bit Of'
```

In [52]:

```
generate_text("Today I am surprised because", 20, model1, maxlen)
```

Out[52]:

```
'Today I Am Surprised Because I Was Going To Be A Little Bit Of A 70S Theme Theme Like In The York Or A 70S'
```

Conclusion & Recommendations

To conclude, this is has been an interesting assignment as I was able to apply machine learning on my favorite show. By doing this assignment I have been able to understand how textual data can be passed through an artificial neural network. Secondly, knowing that the input length needs to be padded to make sure all the lengths are equal was an interesting aspect. Also, by conducting the hyper-parameter tuning of optimizers adam and rmsprop, it has been clear that *adam* is infact the better and optimal optimizer. The quotes or sentences that are generated are very close to what the character Michael would speak in a random episode. Therefore, I believe the model did a good job at capturing the textual data and finding the similarity in the entered phrases. For recommendations, I would like to try out different tokenizers such as BERT to experiment how it would vary the results.

Resources

- https://medium.com/@shivambansal36/language-modelling-text-generation-using-lstms-deep-learning-for-nlp-ed36b224b275
- https://machinelearningmastery.com/text-generation-lstm-recurrent-neural-networks-python-keras/
- http://karpathy.github.io/2015/05/21/rnn-effectiveness/
- https://www.analyticsvidhya.com/blog/2020/02/cnn-vs-rnn-vs-mlp-analyzing-3-types-of-neural-networks-in-deep-learning/
- https://www.ibm.com/cloud/learn/recurrent-neural-networks
- https://towardsdatascience.com/deep-learning-which-loss-and-activation-functions-should-i-use-ac02f1c56aa8
- https://towardsdatascience.com/evaluation-metrics-assessing-the-quality-of-nlg-outputs-39749a115ff3
- https://www.kaggle.com/code/shivamb/beginners-guide-to-text-generation-using-lstms