SENTENCE AUTOCOMPLETION USING LSTM MODEL

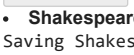
Importing the required libraries

```
import re
import numpy as np
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
import pickle
import warnings
warnings.filterwarnings('ignore')
```

Loading the dataset

```
from google.colab import files
uploaded = files.upload()
```



```
Saving Shakespeare_data.csv to Shakespeare_data.csv
```

```
data = pd.read_csv('Shakespeare_data.csv')
print(data.head())
```

```
   Dataline      Play  PlayerLinenumber ActSceneLine         Player  \
0         1  Henry IV               NaN          NaN            NaN
1         2  Henry IV               NaN          NaN            NaN
2         3  Henry IV               NaN          NaN            NaN
3         4  Henry IV               1.0        1.1.1  KING HENRY IV
4         5  Henry IV               1.0        1.1.2  KING HENRY IV

                                      PlayerLine
0                                          ACT I
1                       SCENE I. London. The palace.
2  Enter KING HENRY, LORD JOHN OF LANCASTER, the ...
3              So shaken as we are, so wan with care,
4         Find we a time for frighted peace to pant,
```

Extracting Text from data

```
# getting text from the data
text = []
for i in data['PlayerLine']:
    text.append(i)

# lets see how the text is looking
text[:5]
```

```
['ACT I',
 'SCENE I. London. The palace.',
 'Enter KING HENRY, LORD JOHN OF LANCASTER, the EARL of WESTMORELAND, SIR WALTER BLUNT, and others',
 'So shaken as we are, so wan with care,',
 'Find we a time for frighted peace to pant,']
```

Cleaning the text

```
# Text Cleaning
def clean_text(text):
    # removing special characters like @, #, $, etc
    pattern = re.compile('[^a-zA-z0-9\s]')
    text = re.sub(pattern,'',text)

    # removing digits
    pattern = re.compile('\d+')
    text = re.sub(pattern,'',text)

    # converting text to lower case
    text = text.lower()
    return text

texts = []
for t in text:
    new_text = clean_text(t)
    texts.append(new_text)

# cleaned text
texts[:5]
```

```
['act i',
 'scene i london the palace',
 'enter king henry lord john of lancaster the earl of westmoreland sir walter blunt and others',
 'so shaken as we are so wan with care',
 'find we a time for frighted peace to pant']
```

Text vectorization and One hot encoding

```
# lets take first 10000 words for the model training
texts = texts[:10000]

# using tensorflow tokenizer and
tokenizer = Tokenizer()
tokenizer.fit_on_texts(texts)

# generating text sequences, i.e. encoding the text
text_sequences = np.array(tokenizer.texts_to_sequences(texts))
print('Text -->>',texts[0])
print('Embedding -->>',text_sequences[0])

# padding the sequences
Max_Sequence_Len = max([len(x) for x in text_sequences])
text_sequences = pad_sequences(text_sequences,
                           maxlen = Max_Sequence_Len, padding='pre')

print('Maximum Sequence Length -->>',Max_Sequence_Len)
print('Text Sequence -->>\n',text_sequences[0])
print('Text Sequence Shape -->>',text_sequences.shape)
```

```
Text -->> act i
Embedding -->> [455, 4]
Maximum Sequence Length -->> 54
Text Sequence -->>
 [  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   0 455   4]
Text Sequence Shape -->> (10000, 54)
```

Splitting the dataset and One hot encoding:

```
# getting X and y from the data
X, y = text_sequences[:, :-1], text_sequences[:,-1]
print('First Input :',X[0])
print('First Target :',y[0])

word_index = tokenizer.word_index

# using one hot encoding on y
total_words = len(word_index) + 1
print('Total Number of Words:',total_words)

y = to_categorical(y, num_classes=total_words)

# printing X and y shapes
print('Input Shape :',X.shape)
print('Target Shape :',y.shape)
```

```
First Input : [  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   0 455]
First Target : 4
Total Number of Words: 7865
Input Shape : (10000, 53)
Target Shape : (10000, 7865)
```

Building the model

```
model = Sequential(name="LSTM_Model")

# adding embedding layer
model.add(Embedding(total_words,
                    Max_Sequence_Len-1,
                    input_length=Max_Sequence_Len-1))

# adding a LSTM layer
model.add(LSTM(512, return_sequences=False))
model.add(Dropout(0.5))

#adding the final output activation with activation function of softmax
model.add(Dense(total_words, activation='softmax'))

# printing model summary
print(model.summary())
```

```
Model: "LSTM_Model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 53, 53)            416845

 lstm (LSTM)                 (None, 512)               1159168

 dropout (Dropout)           (None, 512)               0

 dense (Dense)               (None, 7865)              4034745

=================================================================
Total params: 5610758 (21.40 MB)
Trainable params: 5610758 (21.40 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

Compiling and Training the Model

```
# Compiling the model
model.compile(
    loss="categorical_crossentropy",
    optimizer='adam',
    metrics=['accuracy']
)

# Training the LSTM model
history = model.fit(X, y,
                    epochs=50,
                    verbose=1)
```

```
Epoch 1/50
313/313 [==============================] - 200s 630ms/step - loss: 7.9328 - accuracy: 0.0114
Epoch 2/50
313/313 [==============================] - 196s 627ms/step - loss: 7.3773 - accuracy: 0.0164
Epoch 3/50
313/313 [==============================] - 194s 620ms/step - loss: 7.1245 - accuracy: 0.0257
Epoch 4/50
313/313 [==============================] - 195s 624ms/step - loss: 6.8606 - accuracy: 0.0287
Epoch 5/50
313/313 [==============================] - 193s 616ms/step - loss: 6.5705 - accuracy: 0.0329
Epoch 6/50
313/313 [==============================] - 194s 618ms/step - loss: 6.2691 - accuracy: 0.0405
Epoch 7/50
313/313 [==============================] - 195s 623ms/step - loss: 5.9083 - accuracy: 0.0489
Epoch 8/50
313/313 [==============================] - 194s 621ms/step - loss: 5.4743 - accuracy: 0.0656
Epoch 9/50
313/313 [==============================] - 198s 633ms/step - loss: 5.0169 - accuracy: 0.0913
Epoch 10/50
313/313 [==============================] - 194s 619ms/step - loss: 4.5170 - accuracy: 0.1325
Epoch 11/50
313/313 [==============================] - 196s 627ms/step - loss: 4.0038 - accuracy: 0.1921
Epoch 12/50
313/313 [==============================] - 198s 632ms/step - loss: 3.4521 - accuracy: 0.2722
Epoch 13/50
313/313 [==============================] - 201s 643ms/step - loss: 2.9668 - accuracy: 0.3582
Epoch 14/50
313/313 [==============================] - 196s 627ms/step - loss: 2.4998 - accuracy: 0.4534
Epoch 15/50
313/313 [==============================] - 193s 618ms/step - loss: 2.1037 - accuracy: 0.5448
Epoch 16/50
313/313 [==============================] - 192s 612ms/step - loss: 1.7391 - accuracy: 0.6296
Epoch 17/50
313/313 [==============================] - 195s 624ms/step - loss: 1.4615 - accuracy: 0.6934
Epoch 18/50
313/313 [==============================] - 197s 631ms/step - loss: 1.2454 - accuracy: 0.7415
Epoch 19/50
313/313 [==============================] - 201s 643ms/step - loss: 1.0582 - accuracy: 0.7813
Epoch 20/50
313/313 [==============================] - 202s 644ms/step - loss: 0.8887 - accuracy: 0.8185
Epoch 21/50
313/313 [==============================] - 201s 641ms/step - loss: 0.7907 - accuracy: 0.8411
Epoch 22/50
313/313 [==============================] - 202s 646ms/step - loss: 0.6946 - accuracy: 0.8638
Epoch 23/50
313/313 [==============================] - 202s 644ms/step - loss: 0.6059 - accuracy: 0.8794
Epoch 24/50
313/313 [==============================] - 200s 639ms/step - loss: 0.5266 - accuracy: 0.8992
Epoch 25/50
313/313 [==============================] - 201s 641ms/step - loss: 0.4926 - accuracy: 0.9035
Epoch 26/50
313/313 [==============================] - 197s 630ms/step - loss: 0.4486 - accuracy: 0.9119
Epoch 27/50
313/313 [==============================] - 197s 631ms/step - loss: 0.4189 - accuracy: 0.9173
Epoch 28/50
313/313 [==============================] - 200s 637ms/step - loss: 0.3918 - accuracy: 0.9196
Epoch 29/50
```

Sentence Autocomplete

```
def autoCompletations(text, model):
    # Tokenization and Text vectorization
    text_sequences = np.array(tokenizer.texts_to_sequences([texts]))
    # Pre-padding
    testing = pad_sequences(text_sequences, maxlen = Max_Sequence_Len-1, padding='pre')
    # Prediction
    y_pred_test = np.argmax(model.predict(testing,verbose=0))

    predicted_word = ''
    for word, index in tokenizer.word_index.items():
        if index == y_pred_test:
            predicted_word = word
            break
    text += " " + predicted_word + '.'
    return text

complete_sentence = autoCompletations('I have ',model)
complete_sentence
```