Name: Shrinika Telu

Enroll No: 700741742

Use Case Description:

1. Sentiment Analysis on the Twitter dataset

Programming elements:

1. Basics of LSTM

2. Types of RNN

3. Use case: Sentiment Analysis on the Twitter data set

In class programming:

1. Save the model and use the saved model to predict on new text data (ex, “A lot of good things are happening. We are respected again throughout the world, and that's a great thing.@realDonaldTrump”)

```python
import pandas as pd #Basic packages for creating dataframes and loading dataset
import numpy as np

import matplotlib.pyplot as plt #Package for visualization

import re #importing package for Regular expression operations

from sklearn.model_selection import train_test_split #Package for splitting the data

from sklearn.preprocessing import LabelEncoder #Package for conversion of categorical to Numerical

from keras.preprocessing.text import Tokenizer #Tokenization
from tensorflow.keras.preprocessing.sequence import pad_sequences #Add zeros or crop based on the length
from keras.models import Sequential #Sequential Neural Network
from keras.layers import Dense, Embedding, LSTM, SpatialDropout1D #For layers in Neural Network
from keras.utils.np_utils import to_categorical
```

[2]

```python
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

[3]

```python
import pandas as pd

# Load the dataset as a Pandas DataFrame
dataset = pd.read_csv('/content/gdrive/My Drive/Sentiment.csv')

# Select only the necessary columns 'text' and 'sentiment'
mask = dataset.columns.isin(['text', 'sentiment'])
data = dataset.loc[:, mask]

# Keeping only the necessary columns
data['text'] = data['text'].apply(lambda x: x.lower())
data['text'] = data['text'].apply((lambda x: re.sub('[^a-zA-z0-9\s]', '', x)))
```

```
<ipython-input-3-d0e745dc69e5>:11: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  data['text'] = data['text'].apply(lambda x: x.lower())
<ipython-input-3-d0e745dc69e5>:12: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  data['text'] = data['text'].apply((lambda x: re.sub('[^a-zA-z0-9\s]', '', x)))
```

1m

```python
for idx, row in data.iterrows():
    row[0] = row[0].replace('rt', ' ') #Removing Retweets
    max_fatures = 2000
tokenizer = Tokenizer(num_words=max_fatures, split=' ') #Maximum words is 2000 to tokenize sentence
tokenizer.fit_on_texts(data['text'].values)
X = tokenizer.texts_to_sequences(data['text'].values) #taking values to feature matrix
X = pad_sequences(X) #Padding the feature matrix

embed_dim = 128 #Dimension of the Embedded layer
lstm_out = 196 #Long short-term memory (LSTM) layer neurons
def createmodel():
    model = Sequential() #Sequential Neural Network
    model.add(Embedding(max_fatures, embed_dim,input_length = X.shape[1])) #input dimension 2000 Neurons, output dimension 128 Neurons
    model.add(LSTM(lstm_out, dropout=0.2, recurrent_dropout=0.2)) #Drop out 20%, 196 output Neurons, recurrent dropout 20%
    model.add(Dense(3,activation='softmax')) #3 output neurons[positive, Neutral, Negative], softmax as activation
    model.compile(loss = 'categorical_crossentropy', optimizer='adam',metrics = ['accuracy']) #Compiling the model
    return model
# print(model.summary())
labelencoder = LabelEncoder() #Applying label Encoding on the label matrix
integer_encoded = labelencoder.fit_transform(data['sentiment']) #fitting the model
y = to_categorical(integer_encoded)
X_train, X_test, Y_train, Y_test = train_test_split(X,y, test_size = 0.33, random_state = 42) #67% training data, 33% test data split
batch_size = 32 #Batch size 32
model = createmodel() #Function call to Sequential Neural Network
model.fit(X_train, Y_train, epochs = 1, batch_size=batch_size, verbose = 2) #verbose the higher, the more messages
score,acc = model.evaluate(X_test,Y_test,verbose=2,batch_size=batch_size) #evaluating the model
print(score)
print(acc)
```

```
291/291 - 57s - loss: 0.8159 - accuracy: 0.6480 - 57s/epoch - 198ms/step
144/144 - 4s - loss: 0.7643 - accuracy: 0.6671 - 4s/epoch - 29ms/step
0.7643396854400635
0.6671035289764404
```

0s [5]

```python
print(model.metrics_names) #metrics of the model
```

```
['loss', 'accuracy']
```

0s [6]

```python
#1. Save the model and use the saved model to predict on new text data (ex, “A lot of good things are happening. We are respected again throughout the world, and that's a great thing.@realDonaldTrump”)
model.save('sentimentAnalysis.h5') #Saving the model
```

0s [7]

```python
from keras.models import load_model #Importing the package for importing the saved model
model= load_model('sentimentAnalysis.h5') #loading the saved model
```

```
print(integer_encoded)
print(data['sentiment'])
```

```
[1 2 1 ... 2 0 2]
0         Neutral
1        Positive
2         Neutral
3        Positive
4        Positive
           ...
13866    Negative
13867    Positive
13868    Positive
13869    Negative
13870    Positive
Name: sentiment, Length: 13871, dtype: object
```

```
[9] # Predicting on the text data
    sentence = ['A lot of good things are happening. We are respected again throughout the world, and that is a great thing.@realDonaldTrump']
    sentence = tokenizer.texts_to_sequences(sentence) # Tokenizing the sentence
    sentence = pad_sequences(sentence, maxlen=28, dtype='int32', value=0) # Padding the sentence
    sentiment_probs = model.predict(sentence, batch_size=1, verbose=2)[0] # Predicting the sentence text
    sentiment = np.argmax(sentiment_probs)

    print(sentiment_probs)
    if sentiment == 0:
        print("Neutral")
    elif sentiment < 0:
        print("Negative")
    elif sentiment > 0:
        print("Positive")
    else:
        print("Cannot be determined")
```

```
1/1 - 0s - 329ms/epoch - 329ms/step
[0.42809132 0.12700436 0.4449044 ]
Positive
```

2. Apply GridSearchCV on the source code provided in the class

```
#2. Apply GridSearchCV on the source code provided in the class
```

```
from keras.wrappers.scikit_learn import KerasClassifier #importing Keras classifier
from sklearn.model_selection import GridSearchCV #importing Grid search CV

model = KerasClassifier(build_fn=createmodel,verbose=2) #initiating model to test performance by applying multiple hyper parameters
batch_size= [10, 20, 40] #hyper parameter batch_size
epochs = [1, 2] #hyper parameter no. of epochs
param_grid= {'batch_size':batch_size, 'epochs':epochs} #creating dictionary for batch size, no. of epochs
grid  = GridSearchCV(estimator=model, param_grid=param_grid) #Applying dictionary with hyper parameters
grid_result= grid.fit(X_train,Y_train) #Fitting the model
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_)) #best score, best hyper parameters
```

```
<ipython-input-11-6c99b49150f4>:4: DeprecationWarning: KerasClassifier is deprecated, use Sci-Keras (https://github.com/adriangb/scikeras) instead. See https://www.adriangb.com/scikeras/stable/migration.html for help migrating.
  model = KerasClassifier(build_fn=createmodel,verbose=2) #initiating model to test performance by applying multiple hyper parameters
744/744 - 116s - loss: 0.8266 - accuracy: 0.6485 - 116s/epoch - 156ms/step
186/186 - 3s - loss: 0.7356 - accuracy: 0.6789 - 3s/epoch - 16ms/step
744/744 - 116s - loss: 0.8192 - accuracy: 0.6488 - 116s/epoch - 155ms/step
186/186 - 4s - loss: 0.7641 - accuracy: 0.6676 - 4s/epoch - 20ms/step
744/744 - 114s - loss: 0.8288 - accuracy: 0.6395 - 114s/epoch - 153ms/step
186/186 - 4s - loss: 0.7630 - accuracy: 0.6826 - 4s/epoch - 22ms/step
744/744 - 115s - loss: 0.8252 - accuracy: 0.6469 - 115s/epoch - 155ms/step
186/186 - 3s - loss: 0.7441 - accuracy: 0.6771 - 3s/epoch - 16ms/step
744/744 - 113s - loss: 0.8196 - accuracy: 0.6437 - 113s/epoch - 152ms/step
186/186 - 5s - loss: 0.7814 - accuracy: 0.6615 - 5s/epoch - 25ms/step
Epoch 1/2
744/744 - 116s - loss: 0.8285 - accuracy: 0.6485 - 116s/epoch - 155ms/step
Epoch 2/2
744/744 - 110s - loss: 0.6814 - accuracy: 0.7163 - 110s/epoch - 148ms/step
186/186 - 3s - loss: 0.7521 - accuracy: 0.6681 - 3s/epoch - 17ms/step
Epoch 1/2
744/744 - 115s - loss: 0.8254 - accuracy: 0.6434 - 115s/epoch - 154ms/step
Epoch 2/2
744/744 - 109s - loss: 0.6875 - accuracy: 0.7066 - 109s/epoch - 147ms/step
186/186 - 3s - loss: 0.7322 - accuracy: 0.6772 - 3s/epoch - 16ms/step
Epoch 1/2
744/744 - 115s - loss: 0.8311 - accuracy: 0.6421 - 115s/epoch - 155ms/step
Epoch 2/2
744/744 - 110s - loss: 0.6770 - accuracy: 0.7150 - 110s/epoch - 148ms/step
186/186 - 3s - loss: 0.7515 - accuracy: 0.6891 - 3s/epoch - 17ms/step
Epoch 1/2
744/744 - 119s - loss: 0.8218 - accuracy: 0.6459 - 119s/epoch - 160ms/step
Epoch 2/2
744/744 - 115s - loss: 0.6737 - accuracy: 0.7139 - 115s/epoch - 154ms/step
```

```
372/372 - 62s - loss: 0.6676 - accuracy: 0.7189 - 62s/epoch - 166ms/step
93/93 - 2s - loss: 0.7810 - accuracy: 0.6685 - 2s/epoch - 22ms/step
186/186 - 41s - loss: 0.8520 - accuracy: 0.6341 - 41s/epoch - 222ms/step
47/47 - 1s - loss: 0.7750 - accuracy: 0.6649 - 1s/epoch - 32ms/step
186/186 - 44s - loss: 0.8350 - accuracy: 0.6418 - 44s/epoch - 237ms/step
47/47 - 2s - loss: 0.7696 - accuracy: 0.6708 - 2s/epoch - 35ms/step
186/186 - 43s - loss: 0.8443 - accuracy: 0.6347 - 43s/epoch - 233ms/step
47/47 - 2s - loss: 0.7667 - accuracy: 0.6719 - 2s/epoch - 32ms/step
186/186 - 43s - loss: 0.8509 - accuracy: 0.6346 - 43s/epoch - 230ms/step
47/47 - 2s - loss: 0.7729 - accuracy: 0.6566 - 2s/epoch - 33ms/step
186/186 - 45s - loss: 0.8444 - accuracy: 0.6343 - 45s/epoch - 241ms/step
47/47 - 2s - loss: 0.7802 - accuracy: 0.6647 - 2s/epoch - 34ms/step
Epoch 1/2
186/186 - 43s - loss: 0.8472 - accuracy: 0.6338 - 43s/epoch - 233ms/step
Epoch 2/2
186/186 - 40s - loss: 0.6986 - accuracy: 0.6979 - 40s/epoch - 217ms/step
47/47 - 2s - loss: 0.7312 - accuracy: 0.6842 - 2s/epoch - 33ms/step
Epoch 1/2
186/186 - 42s - loss: 0.8451 - accuracy: 0.6361 - 42s/epoch - 226ms/step
Epoch 2/2
186/186 - 39s - loss: 0.6906 - accuracy: 0.7000 - 39s/epoch - 211ms/step
47/47 - 2s - loss: 0.7559 - accuracy: 0.6842 - 2s/epoch - 39ms/step
Epoch 1/2
186/186 - 44s - loss: 0.8456 - accuracy: 0.6298 - 44s/epoch - 236ms/step
Epoch 2/2
186/186 - 40s - loss: 0.6921 - accuracy: 0.7057 - 40s/epoch - 214ms/step
47/47 - 2s - loss: 0.7897 - accuracy: 0.6638 - 2s/epoch - 35ms/step
Epoch 1/2
186/186 - 43s - loss: 0.8456 - accuracy: 0.6391 - 43s/epoch - 231ms/step
Epoch 2/2
186/186 - 40s - loss: 0.6864 - accuracy: 0.7037 - 40s/epoch - 217ms/step
47/47 - 2s - loss: 0.7489 - accuracy: 0.6857 - 2s/epoch - 46ms/step
Epoch 1/2
186/186 - 41s - loss: 0.8538 - accuracy: 0.6316 - 41s/epoch - 221ms/step
Epoch 2/2
186/186 - 39s - loss: 0.6858 - accuracy: 0.7072 - 39s/epoch - 209ms/step
47/47 - 2s - loss: 0.7689 - accuracy: 0.6749 - 2s/epoch - 33ms/step
Epoch 1/2
465/465 - 86s - loss: 0.8222 - accuracy: 0.6479 - 86s/epoch - 184ms/step
Epoch 2/2
465/465 - 82s - loss: 0.6737 - accuracy: 0.7127 - 82s/epoch - 177ms/step
Best: 0.682986 using {'batch_size': 20, 'epochs': 2}
```