**CREATE A CHATBOT IN PYTHON**

**PHASE 5:** Project Documentation & Submission

**PROJECT TITLE**:Documentation and Submission

**TOPIC:** In this part you will document your project and prepare it for submission.

**Documentation:**

**Problem Statement:** The project aims to create a chatbot integrated into a web application that assists users with natural language processing (NLP) techniques. The chatbot is designed to understand user queries, provide relevant responses, and interact with external APIs to fetch data, enhancing user experience and functionality.

**Design Thinking Process:**

1. **Empathize:** Understanding user needs and pain points related to information retrieval.
2. **Define:** Defining the problem statement and the goals of the chatbot.
3. **Ideate:** Brainstorming possible solutions and features for the chatbot.
4. **Prototype:** Creating a functional prototype of the chatbot and web application.
5. **Test:** Iteratively testing the chatbot with users to gather feedback and make improvements.

****

**Phases of Development:**

1. **Data Collection:** Gathering a diverse dataset for training the NLP models.
2. **Data Preprocessing:** Cleaning and preparing the dataset for training.
3. **NLP Model Development:** Building NLP models using libraries like NLTK, spaCy, or TensorFlow.
4. **Chatbot Implementation:** Integrating the NLP models into the chatbot interface.
5. **Web Application Development:** Designing and developing the web interface for users to interact with the chatbot.
6. **Integration:** Connecting the chatbot with external APIs for data retrieval.
7. **Testing and Optimization:** Rigorous testing, feedback collection, and optimizing the chatbot's performance.
8. **Deployment:** Deploying the chatbot and web application on a server for public access.

**Libraries Used and Integration of NLP Techniques:**

* **Libraries:** NLTK, spaCy, TensorFlow, Flask (for web application), requests (for API integration).
* **NLP Techniques:** Tokenization, entity recognition, intent classification, sentiment analysis.

**Chatbot Interaction:** The chatbot interacts with users through a chat interface on the web application. Users can input text queries, and the chatbot processes the input, analyzes the intent, and generates appropriate responses. If required, the chatbot can also fetch additional information from external APIs based on user requests.

**Innovative Techniques or Approaches:**

* **Dynamic Context Management:** The chatbot maintains context during the conversation, allowing for more coherent and relevant responses.
* **Multi-API Integration:** The chatbot integrates with multiple external APIs, providing diverse and comprehensive information to users.
* **Sentiment Analysis:** The chatbot employs sentiment analysis to understand user emotions and respond empathetically.

**Submission:**

1. **Code Files:**
   * Include all the code files, including chatbot implementation, NLP models, and web application code, in the submission.
2. **README File:**
   * Provide a well-structured README file explaining how to run the code.
   * Include details about dependencies, installation instructions, and any configuration settings.
   * Provide examples of input queries and expected output formats.
   * Clearly outline the steps to deploy the web application and access the chatbot.
3. **Dataset Source:**
   * Include information about the dataset source used for training the NLP models.
   * Provide a brief description of the dataset, including its format and contents.
4. **Sharing the Submission:**
   * Host the project on platforms like GitHub, GitLab, or Bitbucket for version control and easy access for others.
   * Provide a link to the repository in your resume or personal portfolio, enabling others to review and explore the project.

By following these guidelines, your project documentation and submission will be comprehensive, allowing others to understand, run, and evaluate your work effectively.

PROGRAM:

import warnings

warnings.filterwarnings('ignore')

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import tensorflow as tf

import keras

from tqdm import tqdm

from keras.layers import Dense

import json

import re

import string

from sklearn.feature\_extraction.text import TfidfVectorizer

import unicodedata

from sklearn.model\_selection import train\_test\_split

In [2]:

question =[]

answer = []

with open("../input/simple-dialogs-for-chatbot/dialogs.txt",'r') as f :

for line **in** f :

line = line.split('**\t**')

question.append(line[0])

answer.append(line[1])

print(len(question) == len(answer))

True

In [3]:

question[:5]

Out[3]:

['hi, how are you doing?',

"i'm fine. how about yourself?",

"i'm pretty good. thanks for asking.",

'no problem. so how have you been?',

"i've been great. what about you?"]

In [4]:

answer[:5]

Out[4]:

["i'm fine. how about yourself?\n",

"i'm pretty good. thanks for asking.\n",

'no problem. so how have you been?\n',

"i've been great. what about you?\n",

"i've been good. i'm in school right now.\n"]

In [5]:

answer = [ i.replace("**\n**","") for i **in** answer]

In [6]:

answer[:5]

Out[6]:

["i'm fine. how about yourself?",

"i'm pretty good. thanks for asking.",

'no problem. so how have you been?',

"i've been great. what about you?",

"i've been good. i'm in school right now."]

In [7]:

data = pd.DataFrame({"question" : question ,"answer":answer})

data.head()

Out[7]:

|  | question | answer |
| --- | --- | --- |
| 0 | hi, how are you doing? | i'm fine. how about yourself? |
| 1 | i'm fine. how about yourself? | i'm pretty good. thanks for asking. |
| 2 | i'm pretty good. thanks for asking. | no problem. so how have you been? |
| 3 | no problem. so how have you been? | i've been great. what about you? |
| 4 | i've been great. what about you? | i've been good. i'm in school right now. |

In [8]:

def unicode\_to\_ascii(s):

return ''.join(c for c **in** unicodedata.normalize('NFD', s)

if unicodedata.category(c) != 'Mn')

In [9]:

def clean\_text(text):

text = unicode\_to\_ascii(text.lower().strip())

text = re.sub(r"i'm", "i am", text)

text = re.sub(r"\r", "", text)

text = re.sub(r"he's", "he is", text)

text = re.sub(r"she's", "she is", text)

text = re.sub(r"it's", "it is", text)

text = re.sub(r"that's", "that is", text)

text = re.sub(r"what's", "that is", text)

text = re.sub(r"where's", "where is", text)

text = re.sub(r"how's", "how is", text)

text = re.sub(r"\'ll", " will", text)

text = re.sub(r"\'ve", " have", text)

text = re.sub(r"\'re", " are", text)

text = re.sub(r"\'d", " would", text)

text = re.sub(r"\'re", " are", text)

text = re.sub(r"won't", "will not", text)

text = re.sub(r"can't", "cannot", text)

text = re.sub(r"n't", " not", text)

text = re.sub(r"n'", "ng", text)

text = re.sub(r"'bout", "about", text)

text = re.sub(r"'til", "until", text)

text = re.sub(r"[-()**\"**#/@;:<>**{}**`+=~|.!?,]", "", text)

text = text.translate(str.maketrans('', '', string.punctuation))

text = re.sub("(**\\**W)"," ",text)

text = re.sub('\S\*\d\S\*\s\*','', text)

text = "<sos> " + text + " <eos>"

return text

In [10]:

data["question"][0]

Out[10]:

'hi, how are you doing?'

In [11]:

data["question"] = data.question.apply(clean\_text)

In [12]:

data["question"][0]

Out[12]:

'<sos> hi how are you doing <eos>'

In [13]:

data["answer"] = data.answer.apply(clean\_text)

In [14]:

question = data.question.values.tolist()

answer = data.answer.values.tolist()

In [15]:

def tokenize(lang):

lang\_tokenizer = tf.keras.preprocessing.text.Tokenizer(

filters='')

lang\_tokenizer.fit\_on\_texts(lang)

tensor = lang\_tokenizer.texts\_to\_sequences(lang)

tensor = tf.keras.preprocessing.sequence.pad\_sequences(tensor,

padding='post')

return tensor, lang\_tokenizer

In [16]:

input\_tensor , inp\_lang = tokenize(question)

In [17]:

target\_tensor , targ\_lang = tokenize(answer)

In [18]:

*#len(inp\_question) == len(inp\_answer)*

In [19]:

def remove\_tags(sentence):

return sentence.split("<start>")[-1].split("<end>")[0]

In [20]:

max\_length\_targ, max\_length\_inp = target\_tensor.shape[1], input\_tensor.shape[1]

In [21]:

*# Creating training and validation sets using an 80-20 split*

input\_tensor\_train, input\_tensor\_val, target\_tensor\_train, target\_tensor\_val = train\_test\_split(input\_tensor, target\_tensor, test\_size=0.2)

In [22]:

*#print(len(train\_inp) , len(val\_inp) , len(train\_target) , len(val\_target))*

In [23]:

BUFFER\_SIZE = len(input\_tensor\_train)

BATCH\_SIZE = 64

steps\_per\_epoch = len(input\_tensor\_train)//BATCH\_SIZE

embedding\_dim = 256

units = 1024

vocab\_inp\_size = len(inp\_lang.word\_index)+1

vocab\_tar\_size = len(targ\_lang.word\_index)+1

dataset = tf.data.Dataset.from\_tensor\_slices((input\_tensor\_train, target\_tensor\_train)).shuffle(BUFFER\_SIZE)

dataset = dataset.batch(BATCH\_SIZE, drop\_remainder=True)

example\_input\_batch, example\_target\_batch = next(iter(dataset))

example\_input\_batch.shape, example\_target\_batch.shape

Out[23]:

(TensorShape([64, 22]), TensorShape([64, 22]))

In [24]:

class **Encoder**(tf.keras.Model):

def \_\_init\_\_(self, vocab\_size, embedding\_dim, enc\_units, batch\_sz):

super(Encoder, self).\_\_init\_\_()

self.batch\_sz = batch\_sz

self.enc\_units = enc\_units

self.embedding = tf.keras.layers.Embedding(vocab\_size, embedding\_dim)

self.gru = tf.keras.layers.GRU(self.enc\_units,

return\_sequences=True,

return\_state=True,

recurrent\_initializer='glorot\_uniform')

def call(self, x,hidden):

x = self.embedding(x)

output, state = self.gru(x, initial\_state = hidden)

return output, state

def initialize\_hidden\_state(self):

return tf.zeros((self.batch\_sz, self.enc\_units))

In [25]:

encoder = Encoder(vocab\_inp\_size, embedding\_dim, units, BATCH\_SIZE)

*# sample input*

sample\_hidden = encoder.initialize\_hidden\_state()

sample\_output, sample\_hidden = encoder(example\_input\_batch, sample\_hidden)

print ('Encoder output shape: (batch size, sequence length, units) **{}**'.format(sample\_output.shape))

print ('Encoder Hidden state shape: (batch size, units) **{}**'.format(sample\_hidden.shape))

Encoder output shape: (batch size, sequence length, units) (64, 22, 1024)

Encoder Hidden state shape: (batch size, units) (64, 1024)

In [26]:

class **BahdanauAttention**(tf.keras.layers.Layer):

def \_\_init\_\_(self, units):

super(BahdanauAttention, self).\_\_init\_\_()

self.W1 = tf.keras.layers.Dense(units)

self.W2 = tf.keras.layers.Dense(units)

self.V = tf.keras.layers.Dense(1)

def call(self, query, values):

*# query hidden state shape == (batch\_size, hidden size)*

*# query\_with\_time\_axis shape == (batch\_size, 1, hidden size)*

*# values shape == (batch\_size, max\_len, hidden size)*

*# we are doing this to broadcast addition along the time axis to calculate the score*

query\_with\_time\_axis = tf.expand\_dims(query, 1)

*# score shape == (batch\_size, max\_length, 1)*

*# we get 1 at the last axis because we are applying score to self.V*

*# the shape of the tensor before applying self.V is (batch\_size, max\_length, units)*

score = self.V(tf.nn.tanh(

self.W1(query\_with\_time\_axis) + self.W2(values)))

*# attention\_weights shape == (batch\_size, max\_length, 1)*

attention\_weights = tf.nn.softmax(score, axis=1)

*# context\_vector shape after sum == (batch\_size, hidden\_size)*

context\_vector = attention\_weights \* values

context\_vector = tf.reduce\_sum(context\_vector, axis=1)

return context\_vector, attention\_weights

In [27]:

attention\_layer = BahdanauAttention(10)

attention\_result, attention\_weights = attention\_layer(sample\_hidden, sample\_output)

print("Attention result shape: (batch size, units) **{}**".format(attention\_result.shape))

print("Attention weights shape: (batch\_size, sequence\_length, 1) **{}**".format(attention\_weights.shape))

Attention result shape: (batch size, units) (64, 1024)

Attention weights shape: (batch\_size, sequence\_length, 1) (64, 22, 1)

In [28]:

class **Decoder**(tf.keras.Model):

def \_\_init\_\_(self, vocab\_size, embedding\_dim, dec\_units, batch\_sz):

super(Decoder, self).\_\_init\_\_()

self.batch\_sz = batch\_sz

self.dec\_units = dec\_units

self.embedding = tf.keras.layers.Embedding(vocab\_size, embedding\_dim)

self.gru = tf.keras.layers.GRU(self.dec\_units,

return\_sequences=True,

return\_state=True,

recurrent\_initializer='glorot\_uniform')

self.fc = tf.keras.layers.Dense(vocab\_size)

*# used for attention*

self.attention = BahdanauAttention(self.dec\_units)

def call(self, x, hidden, enc\_output):

*# enc\_output shape == (batch\_size, max\_length, hidden\_size)*

context\_vector, attention\_weights = self.attention(hidden, enc\_output)

*# x shape after passing through embedding == (batch\_size, 1, embedding\_dim)*

x = self.embedding(x)

*# x shape after concatenation == (batch\_size, 1, embedding\_dim + hidden\_size)*

x = tf.concat([tf.expand\_dims(context\_vector, 1), x], axis=-1)

*# passing the concatenated vector to the GRU*

output, state = self.gru(x)

*# output shape == (batch\_size \* 1, hidden\_size)*

output = tf.reshape(output, (-1, output.shape[2]))

*# output shape == (batch\_size, vocab)*

x = self.fc(output)

return x, state, attention\_weights

In [29]:

decoder = Decoder(vocab\_tar\_size, embedding\_dim, units, BATCH\_SIZE)

sample\_decoder\_output, \_, \_ = decoder(tf.random.uniform((BATCH\_SIZE, 1)),

sample\_hidden, sample\_output)

print ('Decoder output shape: (batch\_size, vocab size) **{}**'.format(sample\_decoder\_output.shape))

Decoder output shape: (batch\_size, vocab size) (64, 2347)

In [30]:

optimizer = tf.keras.optimizers.Adam()

loss\_object = tf.keras.losses.SparseCategoricalCrossentropy(

from\_logits=True, reduction='none')

def loss\_function(real, pred):

mask = tf.math.logical\_not(tf.math.equal(real, 0))

loss\_ = loss\_object(real, pred)

mask = tf.cast(mask, dtype=loss\_.dtype)

loss\_ \*= mask

return tf.reduce\_mean(loss\_)

In [31]:

@tf.function

def train\_step(inp, targ, enc\_hidden):

loss = 0

with tf.GradientTape() as tape:

enc\_output, enc\_hidden = encoder(inp, enc\_hidden)

dec\_hidden = enc\_hidden

dec\_input = tf.expand\_dims([targ\_lang.word\_index['<sos>']] \* BATCH\_SIZE, 1)

*# Teacher forcing - feeding the target as the next input*

for t **in** range(1, targ.shape[1]):

*# passing enc\_output to the decoder*

predictions, dec\_hidden, \_ = decoder(dec\_input, dec\_hidden, enc\_output)

loss += loss\_function(targ[:, t], predictions)

*# using teacher forcing*

dec\_input = tf.expand\_dims(targ[:, t], 1)

batch\_loss = (loss / int(targ.shape[1]))

variables = encoder.trainable\_variables + decoder.trainable\_variables

gradients = tape.gradient(loss, variables)

optimizer.apply\_gradients(zip(gradients, variables))

return batch\_loss

In [32]:

EPOCHS = 40

for epoch **in** tqdm(range(1, EPOCHS + 1), desc='Epochs', unit='epoch'):

enc\_hidden = encoder.initialize\_hidden\_state()

total\_loss = 0

for (batch, (inp, targ)) **in** enumerate(dataset.take(steps\_per\_epoch)):

batch\_loss = train\_step(inp, targ, enc\_hidden)

total\_loss += batch\_loss

if epoch % 4 == 0:

print('Epoch:**{:3d}** Loss:**{:.4f}**'.format(epoch, total\_loss / steps\_per\_epoch))

Epochs: 10%|█ | 4/40 [01:04<07:01, 11.72s/epoch]

Epoch: 4 Loss:1.5688

Epochs: 20%|██ | 8/40 [01:25<03:25, 6.43s/epoch]

Epoch: 8 Loss:1.3319

Epochs: 30%|███ | 12/40 [01:45<02:29, 5.33s/epoch]

Epoch: 12 Loss:1.1658

Epochs: 40%|████ | 16/40 [02:05<02:03, 5.15s/epoch]

Epoch: 16 Loss:1.0096

Epochs: 50%|█████ | 20/40 [02:25<01:39, 4.99s/epoch]

Epoch: 20 Loss:0.8384

Epochs: 60%|██████ | 24/40 [02:44<01:18, 4.91s/epoch]

Epoch: 24 Loss:0.6452

Epochs: 70%|███████ | 28/40 [03:03<00:57, 4.81s/epoch]

Epoch: 28 Loss:0.4549

Epochs: 80%|████████ | 32/40 [03:23<00:38, 4.83s/epoch]

Epoch: 32 Loss:0.2688

Epochs: 90%|█████████ | 36/40 [03:42<00:19, 4.84s/epoch]

Epoch: 36 Loss:0.1247

Epochs: 100%|██████████| 40/40 [04:01<00:00, 6.04s/epoch]

Epoch: 40 Loss:0.0602

In [33]:

def evaluate(sentence):

sentence = clean\_text(sentence)

inputs = [inp\_lang.word\_index[i] for i **in** sentence.split(' ')]

inputs = tf.keras.preprocessing.sequence.pad\_sequences([inputs],

maxlen=max\_length\_inp,

padding='post')

inputs = tf.convert\_to\_tensor(inputs)

result = ''

hidden = [tf.zeros((1, units))]

enc\_out, enc\_hidden = encoder(inputs, hidden)

dec\_hidden = enc\_hidden

dec\_input = tf.expand\_dims([targ\_lang.word\_index['<sos>']], 0)

for t **in** range(max\_length\_targ):

predictions, dec\_hidden, attention\_weights = decoder(dec\_input,

dec\_hidden,

enc\_out)

*# storing the attention weights to plot later on*

attention\_weights = tf.reshape(attention\_weights, (-1, ))

predicted\_id = tf.argmax(predictions[0]).numpy()

result += targ\_lang.index\_word[predicted\_id] + ' '

if targ\_lang.index\_word[predicted\_id] == '<eos>':

return remove\_tags(result), remove\_tags(sentence)

*# the predicted ID is fed back into the model*

dec\_input = tf.expand\_dims([predicted\_id], 0)

return remove\_tags(result), remove\_tags(sentence)

In [34]:

questions =[]

answers = []

with open("../input/simple-dialogs-for-chatbot/dialogs.txt",'r') as f :

for line **in** f :

line = line.split('**\t**')

questions.append(line[0])

answers.append(line[1])

print(len(question) == len(answer))

True

In [35]:

def ask(sentence):

result, sentence = evaluate(sentence)

print('Question: **%s**' % (sentence))

print('Predicted answer: **{}**'.format(result))

ask(questions[100])

Question: <sos> i believe so <eos>

Predicted answer: good i hope it does not cool off this weekend <eos>

In [36]:

ask(questions[20])

Question: <sos> it is not bad there are a lot of people there <eos>

Predicted answer: good luck with that <eos>

In [37]:

print(answers[20])

good luck with that.

In [38]:

ask(questions[10])

Question: <sos> good luck with school <eos>

Predicted answer: thank you very much <eos>

In [39]:

print(answers[10])

thank you very much.