**CHATBOT USING PYTHON**

TEAM MEMBER

963521106006: K.AKASH

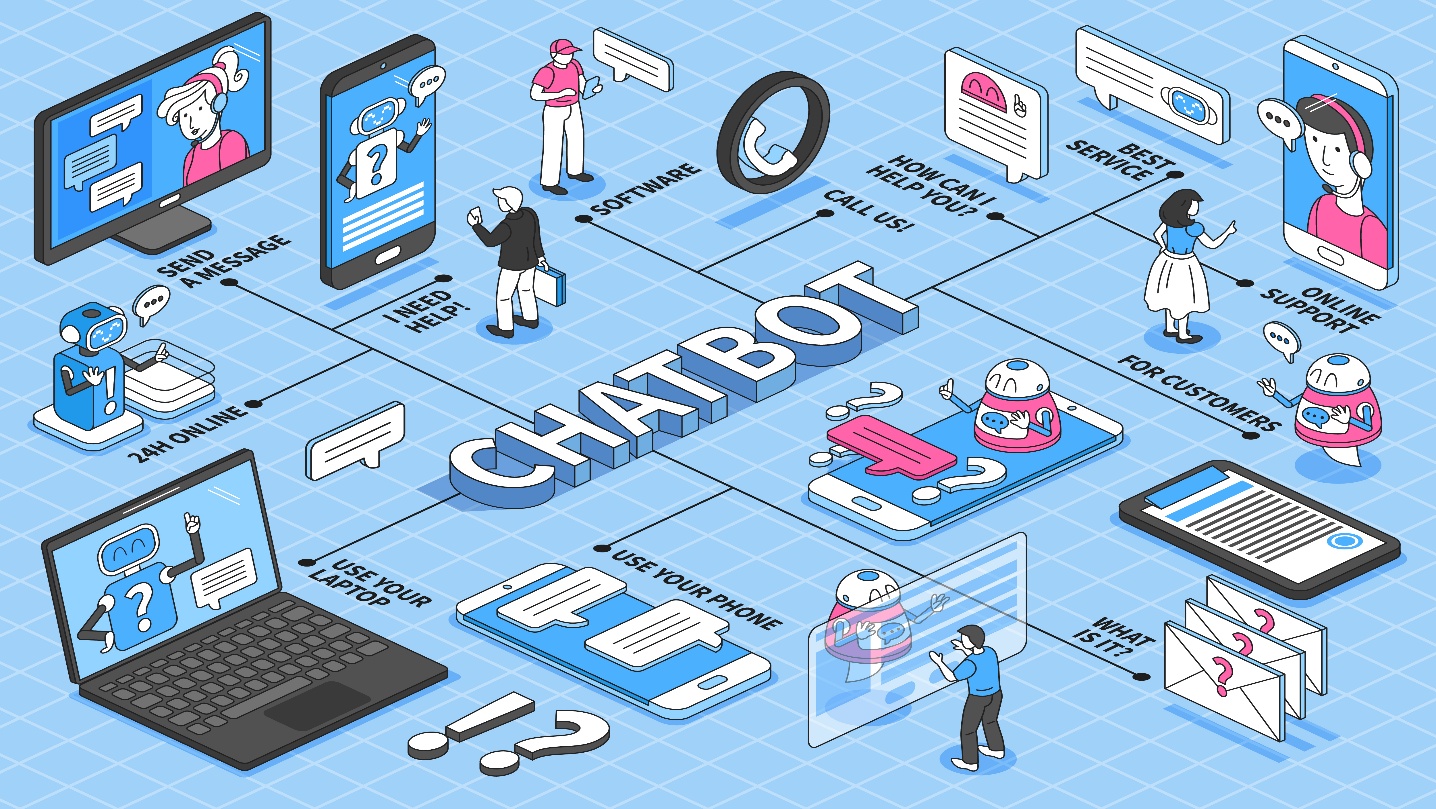
PHASE 4

PROJECT DEVELOPMENT PART 2

**TITLE: Create a Chatbot using Python**

**Abstract:**

Chatbots are computer programs that can simulate conversation with humans. They are becoming increasingly popular in a variety of applications, such as customer service, education, and entertainment. Python is a popular programming language for chatbot development, due to its flexibility and ease of use. Chatbots are computer programs that can simulate conversation with humans.

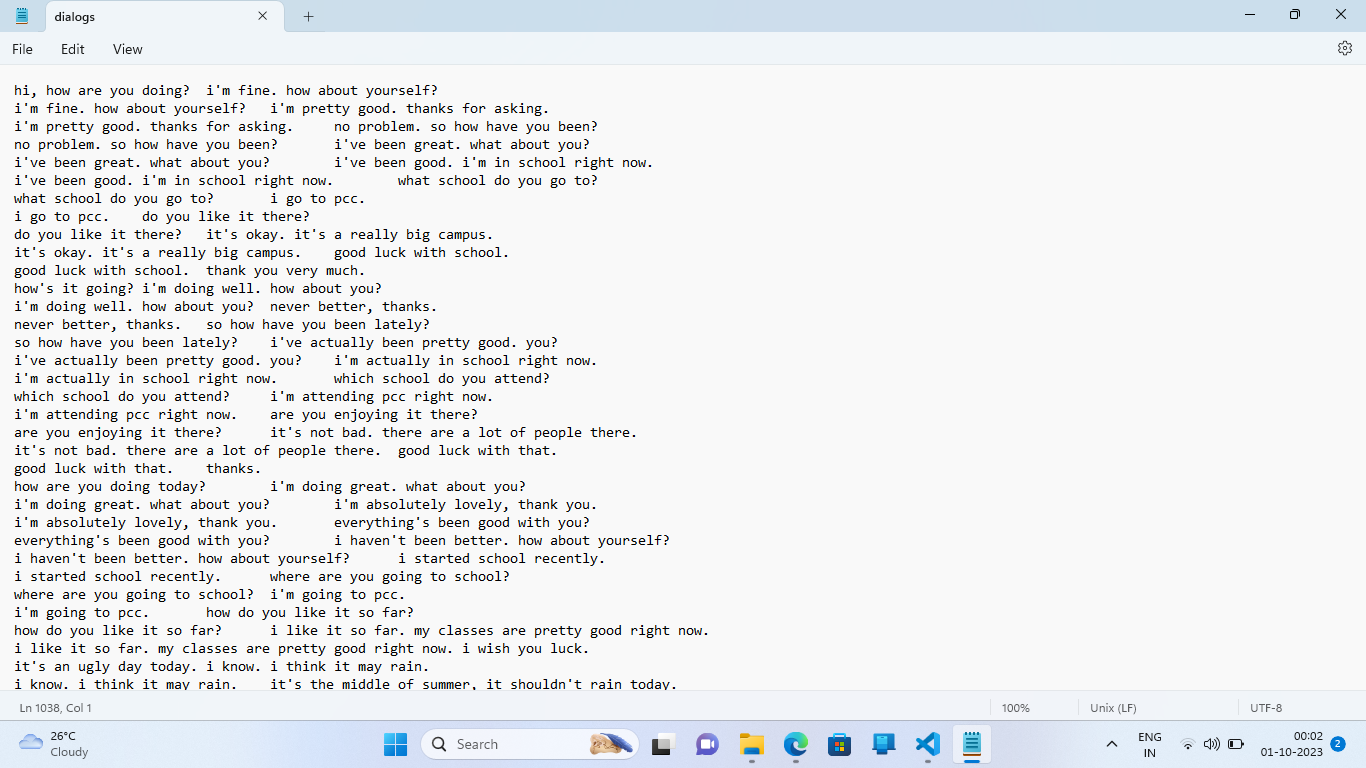


**Problem Definition:**

The challenge is to create a Chatbot in Python that provides exceptional customer service, answering user queries on a website or application. The objective is to deliver high-quality support to users, ensuring a positive user experience and customer satisfaction.

**Dataset:**

**Dataset Link:** [**https://www.kaggle.com/datasets/grafstor/simple-dialogs-for-chatbot**](https://www.kaggle.com/datasets/grafstor/simple-dialogs-for-chatbot)



**Creating Dataset:**

* **Identify the chatbot's purpose and capabilities.** What do you want your chatbot to be able to do? What kind of questions and requests will it need to be able to handle? Once you have a good understanding of the chatbot's purpose, you can start to identify the types of data that you will need to collect.
* **Collect relevant data.** You can collect data from a variety of sources, such as customer support tickets, chat logs, social media posts, and surveys. You can also use existing datasets, such as Wikipedia or the Corpus of Contemporary American English.
* **Clean and preprocess the data.** This may involve removing errors, inconsistencies, and outliers. You may also need to normalize the data by scaling it to a common range or encoding categorical variables.
* **Label the data.** This involves identifying the target variable that you want your chatbot to predict. For example, if you are building a chatbot to answer customer questions, you would need to label each piece of data with the corresponding question category.
* **Split the data into training and testing sets.** The training set is used to train the chatbot, and the testing set is used to evaluate the chatbot's performance on unseen data.

In [1]:

def load\_Dataset(data,size=None):  
   
 if(size!=None):  
 y,X=data[:size]  
 else:  
 y,X=data  
   
 X\_tokenizer=tokenize(X)  
 y\_tokenizer=tokenize(y)  
   
 X\_tensor=vectorization(X\_tokenizer,X)  
 y\_tensor=vectorization(y\_tokenizer,y)  
   
 return X\_tensor,X\_tokenizer, y\_tensor, y\_tokenizer

In [2]:

size=30000  
data=preprocessed\_answers,preprocessed\_questions\  
  
X\_tensor,X\_tokenizer, y\_tensor, y\_tokenizer=load\_Dataset(data,size)

In [3]:

*# Calculate max\_length of the target tensors*  
*max\_length\_y, max\_length\_X* = y\_tensor.shape[1], X\_tensor.shape[1]

**Splitting the data:**

Creating training and validation sets using an 80-20 split after the required preprocessing is applied to the whole data.

In[4]:

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X\_tensor, y\_tensor, test\_size=0.2)  
  
# Show length  
print(len(X\_train), len(y\_train), len(X\_val), len(y\_val))

2980 2980 745 745

## **Tensorflow Dataset**

In[5]:

BUFFER\_SIZE = len(X\_train)  
BATCH\_SIZE = 64  
steps\_per\_epoch = len(X\_train)//BATCH\_SIZE  
embedding\_dim = 256  
units = 1024  
vocab\_inp\_size = len(X\_tokenizer.word\_index)+1  
vocab\_tar\_size = len(y\_tokenizer.word\_index)+1  
  
dataset = tf.data.Dataset.from\_tensor\_slices((X\_train, y\_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

(TensorShape([64, 24]), TensorShape([64, 24]))

## **Building Model Architecture**

* **Choose a machine learning algorithm.** There are a variety of machine learning algorithms that can be used to train chatbots, such as support vector machines (SVMs), recurrent neural networks (RNNs), and transformers. The best algorithm to choose will depend on the specific requirements of your chatbot.
* **Design the model architecture.** The model architecture defines the structure and connections of the machine learning model. For a chatbot, the model architecture will typically consist of an input layer, an output layer, and one or more hidden layers.
* **Implement the model in Python**. There are a number of Python libraries that can be used to implement machine learning models, such as scikit-learn and TensorFlow.
* **Train the model.** Once the model is implemented, it needs to be trained on the dataset that you created. This involves feeding the dataset to the model and allowing it to learn the patterns in the data.
* **Evaluate the model.** Once the model is trained, it needs to be evaluated on a held-out test set. This will give you an estimate of how well the model will perform on unseen data.

#### **Encoder**

In[6]:

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[7]:

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, 24, 1024)  
Encoder Hidden state shape: (batch size, units) (64, 1024)

In[8]:

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[9]:

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, 24, 1)

#### **Decoder**

In[10]:

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[11]:

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, 2349)

## **Training Model**

* Pass the input through the encoder which return encoder output and the encoder hidden state.
* The encoder output, encoder hidden state and the decoder input (which is the start token) is passed to the decoder.
* The decoder returns the predictions and the decoder hidden state.
* The decoder hidden state is then passed back into the model and the predictions are used to calculate the loss.
* Use teacher forcing to decide the next input to the decoder.
* Teacher forcing is the technique where the target word is passed as the next input to the decoder.
* The final step is to calculate the gradients and apply it to the optimizer and backpropagate.

In[12]:

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[13]:

@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([y\_tokenizer.word\_index['<start>']] \* 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[14]:

EPOCHS = 40  
  
for epoch **in** range(1, EPOCHS + 1):  
 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))

Epoch: 4 Loss:1.5338  
Epoch: 8 Loss:1.2803  
Epoch: 12 Loss:1.0975  
Epoch: 16 Loss:0.9404  
Epoch: 20 Loss:0.7773  
Epoch: 24 Loss:0.6040  
Epoch: 28 Loss:0.4042  
Epoch: 32 Loss:0.2233  
Epoch: 36 Loss:0.0989  
Epoch: 40 Loss:0.0470

## **Model Evaluation**

In[15]:

def remove\_tags(sentence):  
 return sentence.split("<start>")[-1].split("<end>")[0]

In[16]:

def evaluate(sentence):  
 sentence = preprocessing(sentence)  
  
 inputs = [X\_tokenizer.word\_index[i] for i **in** sentence.split(' ')]  
 inputs = tf.keras.preprocessing.sequence.pad\_sequences([inputs],  
 maxlen=max\_length\_X,  
 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([y\_tokenizer.word\_index['<start>']], 0)  
  
 for t **in** range(max\_length\_y):  
 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 += y\_tokenizer.index\_word[predicted\_id] + ' '  
  
 if y\_tokenizer.index\_word[predicted\_id] == '<end>':  
 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[17]:

def ask(sentence):  
 result, sentence = evaluate(sentence)  
  
 print('Question: **%s**' % (sentence))  
 print('Predicted answer: **{}**'.format(result))

In[18]:

ask(questions[1])

Question: i m fine . how about yourself ?   
Predicted answer: i m pretty good . thanks for asking .

**Conclusion:**

Creating a chatbot using Python is a relatively straightforward process. By following the steps outlined in this guide, you can create a chatbot that can be used to answer customer questions, provide support, and automate tasks. Thus, a chatbot is to be created.