**CHATBOT USING PYTHON**

TEAM MEMBER

963521106006: K.AKASH

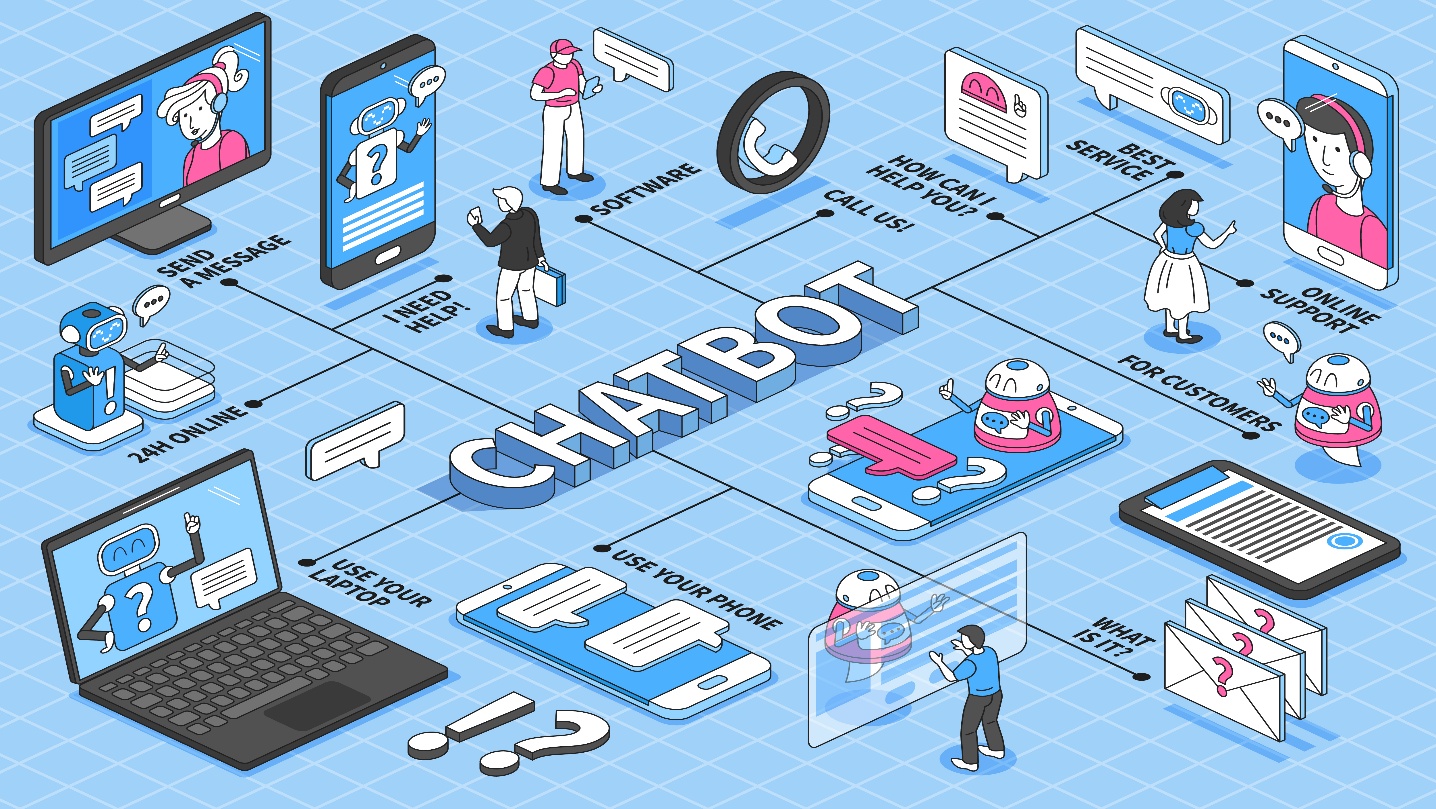
PHASE 5

PROJECT DOCUMENTATION AND SUBMISSION

**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.

**Design Thinking:**

**Functionality:**

* **Natural language processing (NLP):** Chatbots must be able to understand and respond to human language in a natural way. This requires the use of NLP techniques such as tokenization, part-of-speech tagging, named entity recognition, and semantic analysis.
* **Machine learning:** Chatbots can be trained using machine learning algorithms to improve their performance over time. For example, a chatbot can be trained to recognize and respond to different types of customer inquiries.
* **Dialog management:** Chatbots must be able to track the state of a conversation and generate appropriate responses based on the context. This requires the use of dialog management techniques such as slot filling and turn-taking.

Here are some examples of specific functionalities that can be implemented in a Python chatbot:

* **Customer service:** A chatbot can be used to provide customer service by answering customer questions, resolving issues, and providing support.
* **Education:** A chatbot can be used to teach students by providing them with information, answering their questions, and giving them feedback.
* **Entertainment:** A chatbot can be used to entertain users by telling stories, playing games, and cracking jokes.

Python is a popular language for chatbot development because it is flexible, easy to use, and has a large community of developers. There are a number of Python modules available for chatbot development, such as ChatterBot, spaCy, NLTK, rasa, and PyDialogflow. Here, we’ll create a chatbot to provide **Customer service**.

To **develop a chatbot in Python**, you will typically need to:

1. Choose a Python module for chatbot development.
2. Train the chatbot on some data. This data can be collected from a variety of sources, such as customer service transcripts, educational materials, or entertainment content.
3. Implement the desired chatbot functionalities. This may involve using NLP techniques, machine learning algorithms, and dialog management techniques.
4. Test the chatbot to ensure that it is working as expected.
5. Deploy the chatbot to make it available to users.

**User Interface:**

Chatbots can be deployed in a variety of ways, such as through web applications, mobile applications, and social media platforms. The user interface (UI) of a chatbot using Python can be implemented in a variety of ways. One common approach is to use a web-based UI. This involves creating a web page that contains the chatbot's interface. The chatbot can then be accessed by users by visiting the web page in a web browser. Another approach is to use a mobile-based UI. This involves developing a mobile app that contains the chatbot's interface. The chatbot can then be accessed by users by installing and running the mobile app on their mobile devices. Finally, it is also possible to implement a chatbot UI using a command-line interface (CLI). This involves developing a command-line tool that allows users to interact with the chatbot by typing commands. Here, we’ll use **web-based UI**. It can be done using Flash, Django, Kivy and Tkinter.

**Natural Language Processing (NLP):**

Once the chatbot UI has been implemented, it needs to be integrated with the chatbot backend. This involves connecting the UI to the chatbot's natural language processing (NLP) and machine learning (ML) components. This can be done using a variety of techniques, such as web APIs and sockets. Once the chatbot UI has been integrated with the chatbot backend, it is ready to be used by users. Users can then interact with the chatbot by typing or speaking to it. The chatbot will then process the user's input and generate a response.

Natural language processing (NLP) is a field of computer science that deals with the interaction between computers and human (natural) languages. It's used in a wide variety of applications, including chatbots. Chatbots use NLP to understand and respond to human language. This is done by breaking down the user's input into its constituent parts, such as words and phrases, and then interpreting the meaning of those parts. The chatbot can then use this information to generate a response that is relevant and informative. There are a number of different NLP techniques that can be used in chatbots. Some of the most common techniques include:

* **Tokenization:** This involves breaking down the user's input into individual words or phrases.
* **Part-of-speech tagging:** This involves assigning a part of speech (e.g., noun, verb, adjective) to each word or phrase in the user's input.
* **Named entity recognition:** This involves identifying named entities in the user's input, such as people, places, and organizations.
* **Semantic analysis:** This involves understanding the meaning of the user's input, including the relationships between the different words and phrases.

**Responses:**

To write the responses of a chatbot using Python, you can use a variety of techniques. One common approach is to use a rule-based system. This involves creating a set of rules that the chatbot can use to generate responses to different types of user input. Another approach is to use a machine learning-based system. This involves training the chatbot on a dataset of conversations. The chatbot can then use this dataset to learn how to generate responses to different types of user input.

**Integration:**

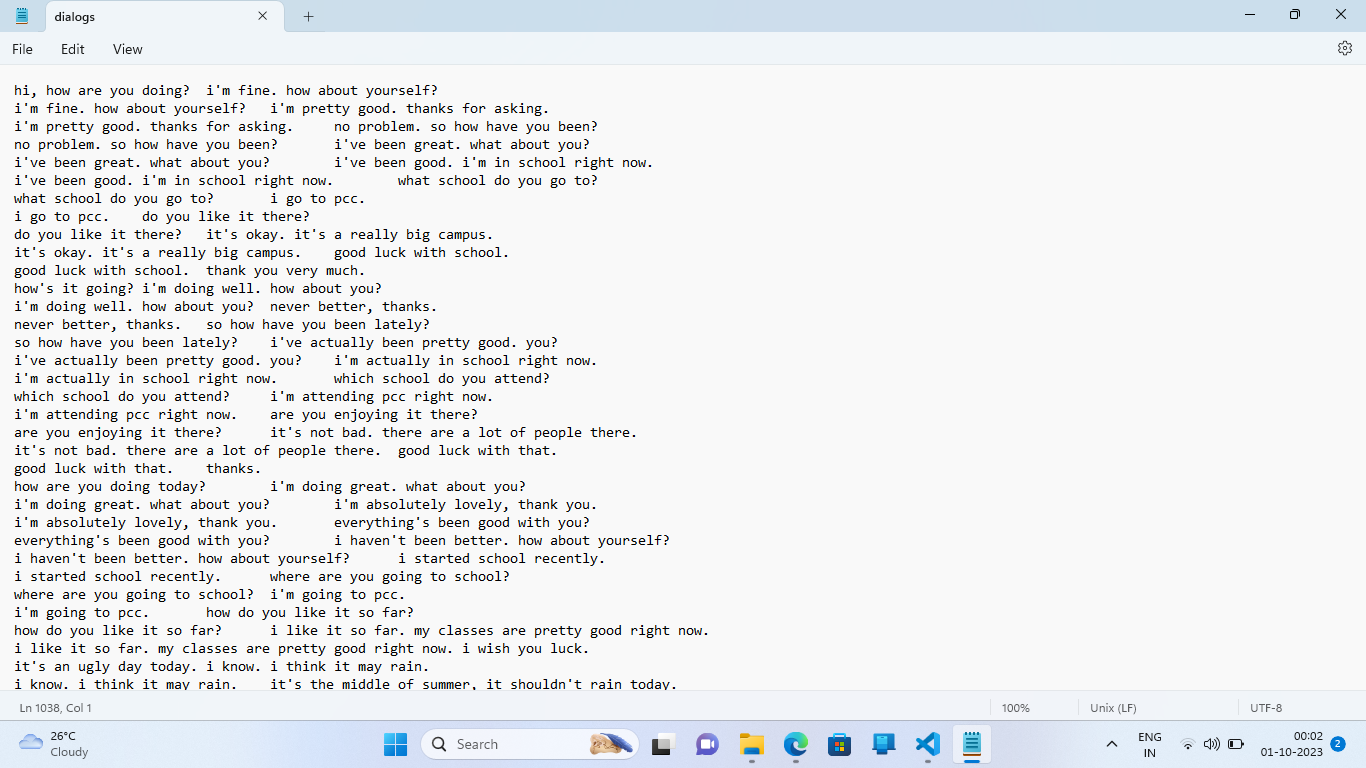
To integrate a chatbot in a website using Python, you can use a variety of approaches. One common approach is to use a web framework, such as Flask or Django. Another approach to integrating a chatbot in a website is to use a chatbot API. This involves using a third-party chatbot service, such as Dialog flow or Rasa.

**Testing and Improvement:**

To test a chatbot using Python, you can use a variety of techniques. One common approach is to use unit testing. This involves writing unit tests for each individual component of the chatbot, such as the natural language processing (NLP) component and the machine learning (ML) component. Another approach to testing a chatbot is to use integration testing. This involves testing the chatbot to ensure that all the components are working together correctly. Finally, you can also use user acceptance testing (UAT) to test the chatbot. This involves testing the chatbot with real users to get their feedback. To improve a chatbot using Python, you can use a variety of techniques. One common approach is to train the chatbot on more data. This will help the chatbot to learn how to respond to a wider range of user input and to generate more informative responses. Another approach to improving a chatbot is to use a different NLP or ML algorithm. This can be helpful if the chatbot is not performing as well as you would like. Finally, you can also improve a chatbot by getting feedback from users. This feedback can be used to identify areas where the chatbot can be improved.

**Dataset:**

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



**Innovation:**

We can use advanced techniques like using pre-trained language models such as ChatGPT and Google Bard to enhance the quality of responses.

Pre-trained program modules are pieces of code that have been trained on large datasets of text and code. These modules can be used to teach chatbots how to perform specific tasks, such as answering questions, translating languages, and generating text.

One advantage of using pre-trained program modules is that it can save time and effort in developing chatbots. Developers do not need to start from scratch when creating a new chatbot. Instead, they can use pre-trained program modules to provide the chatbot with the basic functionality it needs.

Another advantage of using pre-trained program modules is that it can make chatbots more adaptable. As new pre-trained program modules are developed, they can be easily integrated into existing chatbots. This allows chatbots to learn new skills and capabilities over time.

**Open AI ChatGPT 3.5:**

Using ChatGPT 3.5, we can access Chatbot using simple Python code provided with unique OpenAI API key.

**Python code**

import openai

import json

import time

import os

openai.api\_key=”YOUR OPENAI API KEY HERE.”

timestamp = time.strftime(“%Y\_%m\_%d-%H\_%M\_%S”, time.gmtime())

filename = timestamp + “.txt

if not os.path.exists(filename):

with open(filename, ‘w’) as f:

f.write(“User: Welcome to OpenAI chat!\n”)

discussions=[{“role”: “system”,

“content”: “You are a helpful assistant.”}]

while (True):

p=input(“Enter quit to quit, or enter your prompt: “)

if (p==”quit”):

break

discussions.append({“role”: “user”, “content”:p})

completion = openai.ChatCompletion.create(

model=”gpt-3.5-turbo”,

messages=discussions

)

x=json.loads(str(completion))

response = x[“choices”][0][“message”][“content”]

discussions.append({“role”: “assistant”, “content”: response})

with open(filename, ‘a’) as f:

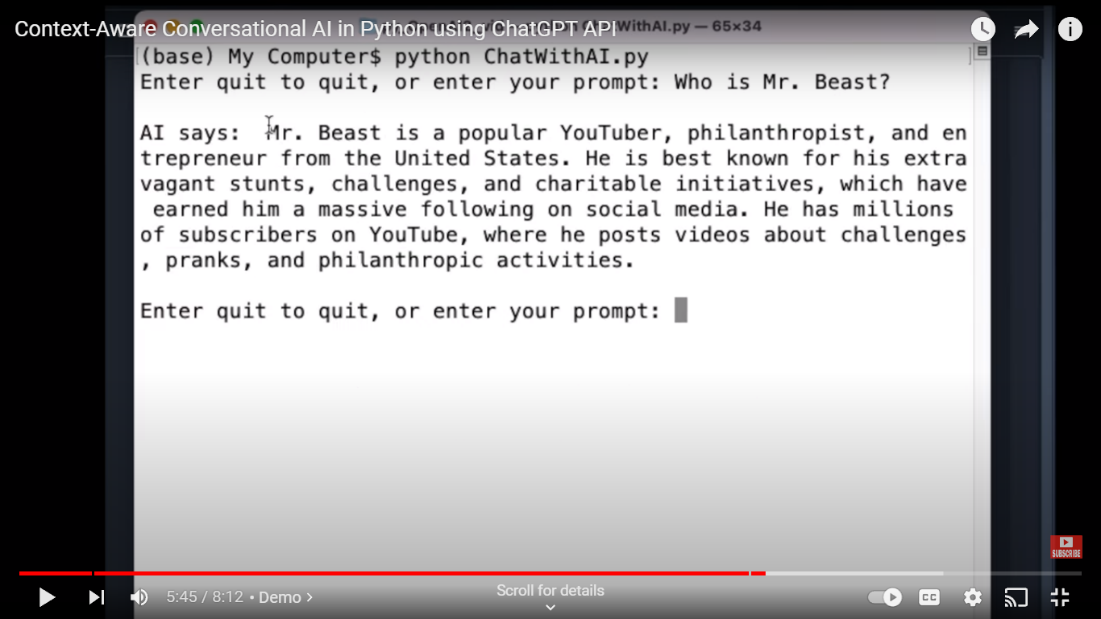
f.write(“User: “ + p + “\n”)

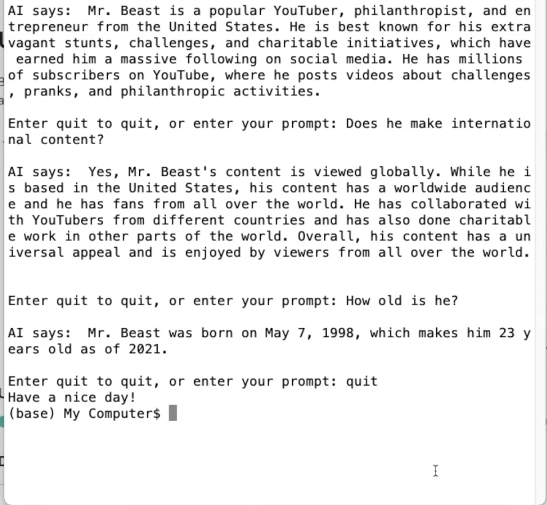
f.write(“AI: “ + response + “\n”)

print(“\nAI says: “, response, “\n”)

print(“Have a nice day!”)

**Output:**



****

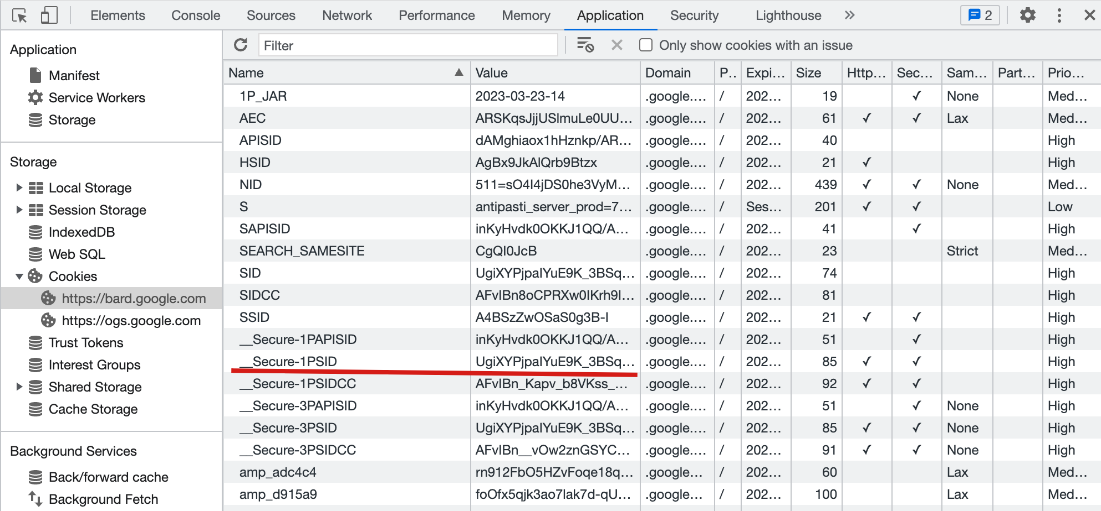
**Google Bard:**

Using Google Bard, we can access Chatbot using simple Python code provided with unique Google Bard’s API key.

## **Get the Session ID**

For security, we need to access Google Bard with Session ID or Token. It's a cookie that we can fetch:

The name is \_\_Secure-1PSID, and we need to copy the value.



## **Python Code**

**The init function is to set the headers and some parameters**.

def \_\_init\_\_(self, session\_id):

headers = {

"Host": "bard.google.com",

"X-Same-Domain": "1",

"User-Agent": "Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.114 Safari/537.36",

"Content-Type": "application/x-www-form-urlencoded;charset=UTF-8",

"Origin": "https://bard.google.com",

"Referer": "https://bard.google.com/",

}

self.\_reqid = int("".join(random.choices(string.digits, k=4)))

self.conversation\_id = ""

self.response\_id = ""

self.choice\_id = ""

self.session = requests.Session()

self.session.headers = headers

self.session.cookies.set("\_\_Secure-1PSID", session\_id)

self.SNlM0e = self.\_\_get\_snlm0e()

**The \_\_Secure-1PSID is the session id we get from browser.**

**After the init done, we need to prepare some data, and then make a POST request to the bard.google.com:**

resp = self.session.post(

"https://bard.google.com/\_/BardChatUi/data/assistant.lamda.BardFrontendService/StreamGenerate",

params=params,

data=data,

timeout=120,

)

chat\_data = json.loads(resp.content.splitlines()[3])[0][2]

if not chat\_data:

return {"content": f"Google Bard encountered an error: {resp.content}."}

json\_chat\_data = json.loads(chat\_data)

results = {

"content": json\_chat\_data[0][0],

"conversation\_id": json\_chat\_data[1][0],

"response\_id": json\_chat\_data[1][1],

"factualityQueries": json\_chat\_data[3],

"textQuery": json\_chat\_data[2][0] if json\_chat\_data[2] is not None else "",

"choices": [{"id": i[0], "content": i[1]} for i in json\_chat\_data[4]],

**}**

## **Run the code**

python Bard.py --session 'UgiXYPjpaIYuE9K\_3BSxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'

ChatGPT - A command-line interface to Google's Bard (https://bard.google.com/)

You:

What is terrible-two?

Google Bard:

Terrible-two is a term used to describe the behavior of a child around the age of 2. It is a normal developmental phase that all children go through. During this time, children

are learning to assert their independence and explore the world around them. This can lead to tantrums, defiance, and other challenging behavior.

There are a few things that parents can do to help manage the terrible twos:

• Be patient and understanding. Remember that this is a normal phase and that your child is not trying to be difficult.

• Set clear boundaries and expectations. Children need to know what is expected of them.

• Be consistent with your discipline. If you are not consistent, your child will not learn what is right and wrong.

• Offer positive reinforcement. Praise your child for good behavior and help them feel good about themselves.

• Talk to your child's doctor if you are concerned about their behavior. Your doctor can offer advice and support.

The terrible twos can be a challenging time for parents, but it is important to remember that this is a normal phase and that your child will eventually grow out of it.

You:

how far is the moon?

Google Bard:

The Moon is an average of 238,855 miles (384,400 km) away from Earth. This distance varies slightly because the Moon's orbit is not a perfect circle. The Moon is closest to Earth

at perigee, which is when it is about 225,623 miles (363,104 km) away. It is farthest from Earth at apogee, which is when it is about 252,088 miles (405,696 km) away.

You:

how far is the sun?

Google Bard:

The Sun is about 93 million miles (150 million kilometers) away from Earth. That's about 1 Astronomical Unit (AU).

The Sun is the center of our solar system. All of the planets and other objects in our solar system orbit around the Sun. The Sun is a very large star. It is about 109 times the

diameter of Earth and about 330,000 times more massive than Earth. The Sun is made up of mostly hydrogen and helium. It is so hot that the hydrogen and helium are constantly

fusing together to create heavier elements. This process is called nuclear fusion. Nuclear fusion releases a lot of energy, which is what powers the Sun.

You:

how to cure a flu?

Google Bard:

There is no cure for the flu, but there are treatments that can help relieve symptoms and prevent complications. The most effective way to prevent the flu is to get a flu vaccine

every year.

If you get the flu, there are a few things you can do to feel better:

• Rest. Get plenty of sleep and avoid strenuous activity.

• Drink plenty of fluids. Water, juice, and soup are all good choices.

• Take over-the-counter pain relievers like acetaminophen or ibuprofen.

• Use a humidifier. The moisture can help soothe your throat and make it easier to breathe.

• See a doctor if you have severe symptoms or are at high risk of complications.

Here are some tips to help you prevent the flu:

• Get a flu vaccine every year. The flu vaccine is the best way to prevent the flu.

• Wash your hands often with soap and water. This helps to prevent the spread of germs.

• Avoid close contact with people who are sick. If you must be around someone who is sick, wear a mask.

• Clean and disinfect surfaces that may be contaminated with germs.

• Stay home from work or school if you are sick. This will help to prevent the spread of the flu.

• Eat a healthy diet and exercise regularly. A healthy immune system is better able to fight off infection.

You:

Exiting...

**PROJECT**:

**LOADING THE DATASET:**

To load the dataset for creating a chatbot using Python, you can use the following steps:

1. **Identify the format of your dataset.** Your dataset can be in a variety of formats, such as CSV, JSON, or text file.
2. **Install the required libraries.** Depending on the format of your dataset, you may need to install additional Python libraries. For example, if your dataset is in CSV format, you will need to install the pandas library.
3. **Load the dataset into a Python object.** You can use the appropriate Python library function to load your dataset into a Python object. For example, to load a CSV file, you can use the pandas.read\_csv() function.

In [1]:

*#model*  
 *import* tensorflow as tf  
 from sklearn.model\_selection import train\_test\_split  
  
 #nlp processing  
 import unicodedata  
 import re  
 import numpy as np  
  
  
 import warnings   
 warnings.filterwarnings('ignore')

/opt/conda/lib/python3.10/site-packages/scipy/\_\_init\_\_.py:146: UserWarning: A NumPy version >=1.16.5 and <1.23.0 is required for this version of SciPy (detected version 1.23.5  
 warnings.warn(f"A NumPy version >={np\_minversion} and <{np\_maxversion}"

In [2]:

*#reading data*  
 *data*=open('/kaggle/input/simple-dialogs-for chatbot/dialogs.txt','r').read()

**PREPROCESSING THE DATASET:**

Preprocessing the dataset is an important step in chatbot training. It involves cleaning and converting the data into a format that can be used by the chatbot model.

## **Segmentation**

Segmentation is the process of dividing a dataset into different groups based on shared characteristics. This can be a useful technique for chatbot development, as it allows you to train different chatbot models for different user groups.

For example, you could segment your dataset by user demographics, such as age, gender, and location. You could also segment your dataset by user interests, such as hobbies and products. Once you have segmented your dataset, you can train different chatbot models for each group. This will allow you to provide more personalized and relevant responses to your users.

There are a number of different ways to segment a dataset for chatbot development. One common approach is to use natural language processing (NLP) techniques to extract features from the text data, such as keywords and phrases. You can then use these features to cluster the data into different groups.

Another approach is to use machine learning techniques to train a model to predict the user group for each data point. This model can then be used to segment the dataset into different groups.

Once you have segmented your dataset, you can train different chatbot models for each group. You can use any of the various Python chatbot libraries available, such as ChatterBot, Rasa, or TensorFlow.

Here are some of the benefits of using segmentation in creating chatbot using Python:

* **More personalized and relevant responses:** By training different chatbot models for different user groups, you can provide more personalized and relevant responses to your users.
* **Improved user satisfaction:** Users are more likely to be satisfied with a chatbot that provides personalized and relevant responses.
* **Increased user engagement:** Users are more likely to engage with a chatbot that provides them with the information and services they need.
* **Improved chatbot performance:** By training chatbot models on specific data sets, you can improve their performance.

In [3]:

*#paried list of question and corresponding answer*  
 *QA\_list*=[QA.split('**\t**') for QA **in** data.split('**\n**')]  
 print(QA\_list[:5])

[['hi, how are you doing?', "i'm fine. how about yourself?"], ["i'm fine. how about yourself?", "i'm pretty good. thanks for asking."], ["i'm pretty good. thanks for asking.", 'no problem. so how have you been?'], ['no problem. so how have you been?', "i've been great. what about you?"], ["i've been great. what about you?", "i've been good. i'm in school right now."]]

In [4]:

questions=[row[0] for row **in** QA\_list]  
 answers=[row[1] for row **in** QA\_list]

In [5]:

print(questions[0:5])  
 print(answers[0:5])

['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?"]  
["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."]

## **Normalization**

Normalization is the process of converting text into a consistent format. This is an important step in chatbot development, as it allows the chatbot model to better understand the text data.

There are a number of different normalization techniques that can be used for chatbot development. Some common techniques include:

* **Converting to lowercase:** Converting all words to lowercase.
* **Removing stop words:** Removing common words that do not add meaning to the sentence, such as "the", "is", and "of".
* **Lemmatizing:** Converting words to their base form.
* **Stemming:** Reducing words to their root form.
* **Punctuation removal:** Removing punctuation marks from the text.

The best normalization technique to use will depend on the specific chatbot application. For example, if you are developing a chatbot for customer service, you may want to remove punctuation marks from the text, as this can make it easier for the chatbot model to understand the user's query.

In [6]:

def remove\_diacritic(text):  
 return ''.join(char for char **in** unicodedata.normalize('NFD',text)  
 if unicodedata.category(char) !='Mn')

In [7]:

def preprocessing(text):  
   
 *#Case folding and removing extra whitespaces*

text=remove\_diacritic(text.lower().strip())  
   
 *#Ensuring punctuation marks to be treated as tokens*  
 text=re.sub(r"([?.!,¿])", r" \1 ", text)  
   
 *#Removing redundant spaces*  
 text= re.sub(r'[" "]+', " ", text)  
   
 *#Removing non alphabetic characters*  
 text=re.sub(r"[^a-zA-Z?.!,¿]+", " ", text)  
   
 text=text.strip()  
   
 *#Indicating the start and end of each sentence*  
 text='<start> ' + text + ' <end>'  
   
 return text

In [8]:

preprocessed\_questions=[preprocessing(sen) for sen **in** questions]  
 preprocessed\_answers=[preprocessing(sen) for sen **in** answers]  
  
 print(preprocessed\_questions[0])  
 print(preprocessed\_answers[0])

<start> hi , how are you doing ? <end>  
<start> i m fine . how about yourself ? <end>

## **Tokenization**

Tokenization is the process of splitting text into smaller units, called tokens. Tokens can be words, punctuation marks, or other symbols. Tokenization is an important step in chatbot development, as it allows the chatbot model to better understand the text data.

In [9]:

def tokenize(lang):  
 lang\_tokenizer = tf.keras.preprocessing.text.Tokenizer(  
 filters='')  
   
 *#build vocabulary on unique words*   
 lang\_tokenizer.fit\_on\_texts(lang)  
   
 return lang\_tokenizer

## **Word Embedding**

Word embedding is a technique for representing words as vectors of real numbers. Word embeddings are learned using machine learning algorithms, and they capture the semantic and syntactic relationships between words.

Word embeddings are useful for chatbot development because they allow chatbots to better understand the meaning of text. For example, if a user asks a chatbot "What is the meaning of life?", the chatbot can use word embeddings to understand that the words "meaning" and "life" are related, and that the user is asking a philosophical question.

In [10]:

def vectorization(lang\_tokenizer,lang):  
   
 *#word embedding for training the neural network*  
 tensor = lang\_tokenizer.texts\_to\_sequences(lang)  
  
 tensor = tf.keras.preprocessing.sequence.pad\_sequences(tensor,  
 padding='post')  
  
 return tensor

**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 [11]:

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 [12]:

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

In [13]:

*# 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[14]:

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[15]:

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[16]:

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[17]:

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[18]:

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[19]:

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[20]:

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[21]:

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[22]:

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[23]:

@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[24]:

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[25]:

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

In[26]:

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[27]:

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

In[28]:

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.