CREATE A CHATBOT IN PYTHON

GROUP 5

TEAM MEMBER

950921104029:R.RESHMA

Phase 5 Submission Document

PROJECT: Project Documentation and Submission

INTRODUCTION:

Creating a chatbot in Python involves several steps, but I can give you a basic example of a chatbot using Python and the NLTK library for natural language processing. This chatbot will be very simple and will respond to a few predefined queries.The objective of advanced techniques in chatbot development is to improve the functionality, performance, and user experience of chatbots by leveraging cutting-edge technologies and methodologies. These techniques aim to address various more natural, context-aware, and effective interactions with users.

FUNCTIONALITY:

Defining the scope of a chatbot's functionality in Python can be achieved by creating a set of functions or rules that specify how the chatbot should respond to different inputs. Below is a basic example of a chatbot's functionality to answer common questions,provide guidance and direct users to appropriate resources.We define a dictionary called common_questions that maps common user questions to their corresponding answers.we create a chatbot_response function that takes user input as an argument. converts it to lowercase to make it case-insensitive, and checks if it matches any of the common questions. If it does, the function returns the corresponding answer; otherwise, it provides a generic response.We use the input function to get user input and then call the chatbot response function to generate a response.Finally, we print the chatbot's response to the user.

PYTHON PROGRAM:

```
import tensorflow as tf import
numpy as np import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from tensorflow.keras.layers import TextVectorization
import re,string
from tensorflow.keras.layers import LSTM,Dense,Embedding,Dropout,LayerNormalization import
random

# Define a list of common questions and their corresponding answers common_questions =
{
```

```
    "What is your name?": "I am a chatbot.",
    "How are you?": "I'm just a computer program, so I don't have feelings, but thanks for asking!",
    "What can you do?": "I can answer questions, provide guidance, and direct you to resources.",
}

# Define a function to handle user input def
chatbot_response(user_input):
    user_input = user_input.lower()  # Convert user input to lowercase for case-insensitivity

    # Check if the user input is a common question     if
user_input in common_questions:
        return common_questions[user_input]     else:
        # If the input is not a common question, provide a generic response         return "I'm
not sure how to respond to that. Please ask me a different question."

# Example usage:
user_input = input("You: ")  # Get user input
bot_response = chatbot_response(user_input)
print("Chatbot:", bot_response)
```

USER INTERFACE:

A user interface (UI) in the context of a chatbot refers to the means through which a user interacts with the chatbot. It is the interface that facilitates communication between the user and the chatbot. The design of the user interface significantly impacts the user experience and the effectiveness of the chatbot. Here are some key aspects to consider when discussing the user interface in a chatbot:

Creating a user-friendly interface for a chatbot depends on the platform where you want to integrate it. Here, I'll show you an example of how to create a simple command-line interface (CLI) for interactions using Python. You can adapt this code to integrate the chatbot into a website, app, or any other platform of your choice.

- **Medium of Interaction:** The user interface can take various forms depending on where the chatbot is deployed. Some common mediums include:
- **Text-Based:** This is the most common form, where users type text messages or question to the chatbot and receive text-based responses.
- **Voice-Based:** In voice-based chatbots, users communicate with the bot through spoken language. Platforms like Amazon Alexa and Google Assistant use voice-based UIs.

- **Graphical User Interface (GUI):** In some cases, chatbots are integrated into graphical interfaces, such as mobile apps or web applications, where users can interact with the chatbot through buttons, menus, and visual elements.
- **Conversational Flow:** A chatbot's user interface should be designed to facilitate natural and intuitive conversations. This includes handling multi-turn conversations and context switching when the user changes topics.
- **Feedback and Prompts:** Providing clear feedback to the user is essential for a good user interface. The chatbot should acknowledge user input, confirm actions, and provide helpful prompts to guide the conversation.
- **Personalization:** User interfaces can be personalized to enhance the user experience. This might include addressing the user by name, remembering past interactions, and adapting responses based on user preferences or history.
- **Error Handling:** The UI should be designed to handle user errors and misunderstandings gracefully. It should provide helpful error messages and suggest possible corrections.

PYTHON PROGRAM:

```
import random

df=pd.read_csv('/kaggle/input/simple-dialogs-

forchatbot/dialogs.txt',sep='\t',names=['question','answer'])

print(f'Dataframe size: {len(df)}') df.head()

# Define a dictionary of common questions and answers common_questions = {

    "What is your name?": "I am a chatbot.",

    "How are you?": "I'm just a computer program, so I don't have feelings, but thanks for asking!",

    "What can you do?": "I can answer questions, provide guidance, and direct you to resources.",

}



# Function to handle user input def

chatbot_response(user_input):

    user_input = user_input.lower()  # Convert user input to lowercase for case-insensitivity



    # Check if the user input is a common question     if

user_input in common_questions:
```

```
        return common_questions[user_input]
    else:
        # If the input is not a common question, provide a generic response          return "I'm
not sure how to respond to that. Please ask me a different question."


# Function to display the chatbot interface def
chat_interface():
    print("Chatbot: Hello! How can I assist you today? (Type 'exit' to end the conversation)")


    while True:
        user_input = input("You: ")


        if user_input.lower() == 'exit':
            print("Chatbot: Goodbye!")
break


        bot_response = chatbot_response(user_input)          print("Chatbot:",
bot_response)


# Run the chatbot interface if
__name__ == "__main__":
    chat_interface()                  def
print_conversation(texts):       for
text in texts:
        print(f'You: {text}')
print(f'Bot: {chatbot(text)}')
```

```
print('=======================')
```

NATURAL LANGUAGE PROCESSING:

Natural Language Processing, is a fundamental component of chatbots and plays a crucial role in their ability to understand and generate human language. NLP in chatbots refers to the set of techniques and technologies used to enable the chatbot to interact with users in a natural, human-like way by processing and understanding text or speech input. Here's how NLP is applied in chatbots:

- **Text Understanding:** Chatbots use NLP to understand and interpret the text-based input provided by users. This includes tasks such as tokenization (breaking text into words or tokens), part-of-speech tagging (identifying the grammatical roles of words), and syntactic parsing (analyzing the sentence structure).

- **Intent Recognition:** NLP helps chatbots recognize the intent behind user input. It involves identifying what the user wants or the action they are trying to perform. For example, if a user asks, "What's the weather like today?" the chatbot should recognize the intent as a weather inquiry.
- **Entity Recognition:** NLP allows chatbots to identify and extract specific pieces of information or entities from user input. For instance, in the question "What's the weather like in New York today?" the entity "New York" should be recognized as a location.
- **Context Management:** Chatbots use NLP to manage conversational context. This involves remembering previous interactions and maintaining context throughout the conversation. Context helps the chatbot provide relevant responses and answer follow-up questions accurately.
- **Sentiment Analysis:** NLP can be applied to analyze the sentiment or emotional tone of user input. Chatbots can use this information to tailor responses based on the user's mood or sentiment.

PYTHON PROGRAM:

```
    text=re.sub('[.]',' . ',text)     text=re.sub('[1]',' 1
',text)     text=re.sub('[2]',' 2 ',text)
text=re.sub('[3]',' 3 ',text)     text=re.sub('[4]',' 4
',text)     text=re.sub('[5]',' 5 ',text)
text=re.sub('[6]',' 6 ',text)     text=re.sub('[7]',' 7
',text)     text=re.sub('[8]',' 8 ',text)
text=re.sub('[9]',' 9 ',text)     text=re.sub('[0]',' 0
',text)     text=re.sub('[,]',' , ',text)
text=re.sub('[?]',' ? ',text)     text=re.sub('[!]',' !
',text)     text=re.sub('[$]',' $ ',text)
text=re.sub('[&]',' & ',text)     text=re.sub('[/]',' /
',text)     text=re.sub('[:]',' : ',text)
text=re.sub('[;]',' ; ',text)     text=re.sub('[*]',' *
',text)     text=re.sub('[\']',' \' ',text)
text=re.sub('[\"]',' \" ',text)     text=re.sub('\t',' ',text)
    return text

df.drop(columns=['answer tokens','question tokens'],axis=1,inplace=True)
df['encoder_inputs']=df['question'].apply(clean_text)
```

```
df['decoder_targets']=df['answer'].apply(clean_text)+' <end>' df['decoder_inputs']='<start>
'+df['answer'].apply(clean_text)+' <end>'

# Load the spaCy language model nlp =

spacy.load("en_core_web_sm")


# Define a function to process user input def

process_user_input(user_input):

   # Tokenize and parse the user input using spaCy     doc =

nlp(user_input)


   # Extract named entities (e.g., names, places, organizations)     entities =

[(ent.text, ent.label_) for ent in doc.ents]


   # Extract nouns and verbs     nouns = [token.text for token in doc if

token.pos_ == "NOUN"]     verbs = [token.text for token in doc if

token.pos_ == "VERB"]


   return {

      "tokens": [token.text for token in doc],

      "entities": entities,

      "nouns": nouns,

      "verbs": verbs,

   }

# Example usage: user_input = "Tell me about OpenAI, and

recommend a book." result = process_user_input(user_input)


# Display the processed information print("User

Input Tokens:", result["tokens"]) print("Named
```

Entities:", result["entities"]) print("Nouns:",

result["nouns"]) print("Verbs:", result["verbs"])

OUTPUT:

	answer	encoder_inputs	decoder_targets	decoder_inputs	
0	hi, how are you doing?	i'm fine. how about yourself?	hi , how are you doing ?	i ' m fine . how about yourself ? <end>	<start> i ' m fine . how about yourself ? <end>
1	i'm fine. how about yourself?	i'm pretty good. thanks for asking.	i ' m fine . how about yourself ?	i ' m pretty good . thanks for asking . <end>	<start> i ' m pretty good . thanks for asking...
2	i'm pretty good. thanks for asking.	no problem. so how have you been?	i ' m pretty good . thanks for asking .	no problem . so how have you been ? <end>	<start> no problem . so how have you been ? ...
3	no problem. so how have you been?	i've been great. what about you?	no problem . so how have you been ?	i ' ve been great . what about you ? <end>	<start> 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.	i ' ve been great . what about you ?	i ' ve been good . i ' m in school right now ...	<start> i ' ve been good . i ' m in school ri...

	5	i've been good. i'm in school right now.	what school do you go to?	i ' ve been good . i ' m in school right now .	what school do you go to ? <end>	<start> what school do you go to ? <end>
	6	what school do you go to?	i go to pcc.	what school do you go to ?	i go to pcc . <end>	<start> i go to pcc . <end>
	7	i go to pcc.	do you like it there?	i go to pcc .	do you like it there ? <end>	<start> do you like it there ? <end>
	8	do you like it there?	it's okay. it's a really big campus.	do you like it there ?	it ' s okay . it ' s a really big campus . <...	<start> it ' s okay . it ' s a really big cam...
	9	it's okay. it's a really big campus.	good luck with school.	it ' s okay . it ' s a really big campus .	good luck with school . <end>	<start> good luck with school . <end>

RESPONSES:

In the context of a chatbot, a "response" refers to the reply or answer that the chatbot provides to a user's input or query. Responses are a fundamental aspect of the chatbot's interaction with users, and they play a crucial role in delivering a satisfying user experience. Here are key points to understand about responses in chatbots:

- **Purpose of Responses:** Responses in a chatbot serve several purposes, including providing information, answering questions, offering assistance, guiding users, and engaging in a conversation. The purpose of a response depends on the user's input and the chatbot's functionality.
- **User Input:** Responses are generated in reaction to the user's input, which can be in the form of text, voice, or other forms of communication, depending on the chatbot's design.
- **Response Types:**
  - **Accurate Answers:** Chatbots should aim to provide accurate and relevant answers to user queries. For factual questions, the response should contain precise information.
  - **Suggestions:** Chatbots can offer suggestions or recommendations based on user preferences or historical interactions. For example, suggesting products, movies, or articles.
  - **Assistance:** Responses can include instructions, guidance, or step-by-step assistance to help users complete tasks or solve problems.

- **Engagement:** In more conversational chatbots, responses may focus on engaging the user, sharing jokes, or maintaining a friendly and interactive tone.
- **Error Handling:** When users provide unclear or incorrect input, the chatbot's response should be designed to handle errors gracefully, asking for clarification or providing helpful suggestions

PYTHON PROGRAM:

```
# Define a function to generate responses
def generate_response(user_input):     user_input = user_input.lower()  # Convert user input to
lowercase for case-insensitivity


   # Define a dictionary of responses based on user input     responses =
{
       "hello": "Hello! How can I assist you today?",
       "how are you": "I'm just a computer program, so I don't have feelings, but thanks for asking!",
       "what is your name": "I am a chatbot.",
       "tell me a joke": "Why don't scientists trust atoms? Because they make up everything!",
       "recommend a movie": "Sure! What genre are you interested in?",
       "recommend a book": "Of course! What genre or author do you prefer?",
   }


   # Check if the user input matches any predefined responses     if
user_input in responses:
       return responses[user_input]     else:
       # If there's no predefined response, provide a generic response          return "I'm not
sure how to respond to that. Please ask me a different question."


# Example usage:
```

```
user_input = input("You: ")  # Get user input
bot_response = generate_response(user_input)
print("Chatbot:", bot_response)
```

INTEGERATION:

Integration in the context of a chatbot refers to the process of connecting the chatbot with other systems, platforms, or services to enhance its functionality and usability. Chatbot integration allows it to interact with external data sources, perform tasks, and provide valuable services to users. Here are key aspects to understand about integration in chatbots: Ø **Platform Integration:** Chatbots can be integrated into various platforms or channels,

- **API Integration:** Chatbots can connect to external APIs (Application Programming Interfaces) to access data or services from third-party sources. For example, a chatbot may integrate with weather APIs to provide weather forecasts or with e-commerce APIs to retrieve product information and prices.
- **Database Integration:** Chatbots can access and query databases to retrieve and update information. For instance, a chatbot for an e-commerce website might connect to a product database to fetch product details.
- **Machine Learning and AI Integration:** Advanced chatbots can integrate with machine learning and AI models to perform tasks like sentiment analysis, language translation, or content recommendation. These models may be hosted externally or developed in-house.
- **Authentication and Security:** Integration often involves handling authentication and security measures to ensure that the chatbot can access external systems securely and protect user data.
- **Data Synchronization:** In scenarios where chatbots are used in tandem with other applications, integration may involve data synchronization to ensure that data remains consistent across systems.
- **User Authentication:** For personalized experiences, chatbots may integrate with user authentication systems to identify and authenticate users, enabling tailored interactions and access to user-specific data.
- **Transaction Processing:** Some chatbots can integrate with payment gateways and financial systems to facilitate transactions, such as making bookings, purchases, or payments.
- **Web Scraping:** In cases where official APIs are unavailable, chatbots may employ web scraping techniques to extract information from websites.
- **Continuous Integration and Deployment (CI/CD):** Integration pipelines can be set up to streamline the development, testing, and deployment of chatbot updates and improvements.
- **Analytics and Monitoring:** Integration with analytics tools allows for the collection of data on user interactions, enabling the evaluation of chatbot performance and user behaviour.
- **Feedback Integration:** Chatbots can collect and integrate user feedback into their design and improvement processes, ensuring a more user-centric experience.

Overall, integration is a critical aspect of chatbot development, as it expands the chatbot's capabilities and allows it to provide valuable services and information from various sources. Effective integration strategies should be carefully planned and implemented to ensure that the chatbot functions seamlessly within its intended ecosystem.

TESTING AND IMPROVEMENT:

Testing and improvement in chatbots are crucial processes:

Testing:

- Ensures the chatbot functions correctly.
- Involves functional, user, regression, performance, security, and NLP testing.
- Identifies issues and verifies usability.

Improvement:

- Refines chatbot based on user feedback.
- Iterates on design, responses, and functionality.
- Incorporates analytics, A/B testing, and error handling.
- Keeps content updated and enhances personalization, naturalness, and security.
- Ensures continuous evolution and user satisfaction.

PYTHON PROGRAM:

```
# Sample dictionary to store user feedback user_feedback = {

   "positive": [],

   "negative": [],

   "suggestions": [],

}


# Function to collect and process user feedback def

collect_user_feedback(feedback):

   feedback_type = feedback.get("type", None)     if

feedback_type is None:

      return "Please provide feedback type ('positive', 'negative', or 'suggestion')."


   message = feedback.get("message", "")     if

feedback_type == "positive":

      user_feedback["positive"].append(message)     elif

feedback_type == "negative":

      user_feedback["negative"].append(message)     elif

feedback_type == "suggestion":
```

```
        user_feedback["suggestions"].append(message)

    return "Thank you for your feedback!"

# Function to analyze user feedback and make improvements def
analyze_and_improve():

    # Analyze user feedback and chat logs here
    # Implement logic to identify areas for improvement
    # Adjust chatbot responses or functionality accordingly

    # For simplicity, let's assume some improvements are made     improved_responses = {

        "positive": ["Thank you!", "Great to hear that!"],
        "negative": ["I apologize for the inconvenience.", "We'll work on improving."],
    }

    # Update chatbot responses based on improvements
    # In practice, this would involve modifying your chatbot's response generation logic
chatbot_responses.update(improved_responses)


# Example user feedback user_feedback_data = [

    {"type": "positive", "message": "I found the chatbot very helpful."},
    {"type": "negative", "message": "The chatbot misunderstood my question."},     {"type":
"suggestion", "message": "Can you provide more examples?"}

]

# Collect and process user feedback for
feedback in user_feedback_data:

    response = collect_user_feedback(feedback)     print(response)

# Analyze user feedback and make improvements analyze_and_improve()
```

```
# Example chatbot responses after improvements chatbot_responses = {
    "positive": ["Hello! How can I assist you today?", "Sure, I can help with that."],
    "negative": ["I apologize for the inconvenience.", "Let me try to better understand your question."],
}

# Simulate chatbot interaction
user_input = "Can you help me with a math problem?" print("User: " +
user_input)

print("Chatbot: " + random.choice(chatbot_responses["positive"]))

# You would continue collecting user feedback and iterating on improvements as needed.
```

OUTPUT:

	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.

TRANSFORMER BASED MODEL:

A Transformer is a type of deep learning model architecture introduced in the paper "Attention Is All You Need" by Vaswani et al. in 2017. The key innovation of the Transformer architecture is its self-attention mechanism, which allows the model to weigh the importance of different parts of an input sequence when generating an output sequence. This selfattention mechanism is what makes Transformers particularly powerful in NLP tasks.Transformer-based models are typically pre-trained on vast text corpora containing diverse language patterns and contexts. These models learn to capture linguistic relationships, context, and semantics, making them capable of understanding and generating text with remarkable fluency and coherence.

PYTHON PROGRAM:

```
import en_core_web_lg # Large SpaCy model for English language import numpy as np
import re # regular expressions import spacy # NLU library

from collections import defaultdict  from sklearn.svm
import SVC # Support Vector Classification model
```

```
linkcode  output_format = "IN: {input}\nOUT: {output}\n" + "_"*50

# hard-coded exact questions responses_exact = {
   "what would you like to eat tonight?": "Pasta with salmon and red pesto please!",

   "what time will you be home tonight?": "I will be home around 6 pm.",

"default": "I love you too!"

}

def respond_exact(text):     response = responses_exact.get(text.lower(),
responses_exact['default'])     return(output_format.format(input=text, output=response))

print(respond_exact("What would you like to eat tonight?")) print("_"*50)
print(respond_exact("What time will you be home tonight?")) print("_"*50) print(respond_exact("I
just found out my boss is leaving the company."))
```

OUTPUT:

IN: What would you like to eat tonight?

OUT: Pasta with salmon and red pesto please!

_____
_____ IN:

What time will you be home tonight?

OUT: I will be home around 6 pm.

_____
_____ IN:

I just found out my boss is leaving the company.

OUT: I love you too!

_____

DATA AUGUMENTATION:

- Data augmentation is a technique used in chatbot development, as well as in various other fields of machine learning and natural language processing (NLP). It involves generating additional training data from existing datasets by applying various transformations and modifications to the original data. Data augmentation helps chatbot models become more robust, generalize better, and handle a wider range of inputs
- data augmentation is a valuable technique in chatbot development that involves creating additional training data by applying various transformations to existing data. It helps chatbots become more robust, generalize better, and handle a wider range of user inputs and language

variations. Careful consideration of data quality and the balance between augmentation and overfitting is essential for its successful implementation.

PYTHON PROGRAM:

```
# Check for null values null_question = df['Question'].isnull().sum()

null_answer = df['Answer'].isnull().sum()


if null_question > 0:     print("There are", null_question, "null values in the
'Question' column.") else:      print("There are no null values in the 'Question'
column.")


if null_answer > 0:     print("There are", null_answer, "null values in the 'Answer'
column.") else:

   print("There are no null values in the 'Answer' column.")


# Check for whitespace values whitespace_question =
df['Question'].apply(lambda x: x.isspace()).sum() whitespace_answer = df['Answer'].apply(lambda x:
x.isspace()).sum()


if whitespace_question > 0:      print("There are", whitespace_question, "whitespace values in
the 'Question' column.") else:      print("There are no whitespace values in the 'Question'
column.")


if whitespace_answer > 0:     print("There are", whitespace_answer, "whitespace values in the
'Answer' column.") else:      print("There are no whitespace values in the 'Answer' column.")
```

CONTINOUS LEARNING:

- Continuous learning in chatbots refers to the ability of a chatbot to adapt and improve its performance over time by learning from ongoing interactions with users. It involves updating the chatbot's knowledge, responses, and capabilities based on user feedback and new information. In this example, we'll implement continuous learning in a chatbot using Python, and we'll simulate the learning process by incorporating user feedback into the chatbot's responses. For simplicity, we'll create a basic chatbot that learns to respond to user queries and adapts its responses based on user feedback.
- You can interact with the chatbot by entering user queries, providing feedback, and observing how the chatbot adapts its responses over time. This example demonstrates a basic form of continuous learning, where the chatbot learns from user feedback to improve its performance. In more advanced chatbot systems, continuous learning can involve updating models, expanding knowledge bases, and incorporating new data sources for more sophisticated adaptation.

PYTHON PROGRAM:

```
import random


# Define a dictionary to store chatbot responses and user feedback chatbot_responses
= {
    "hello": ["Hello!", "Hi there!", "Hey! How can I assist you today?"],
    "goodbye": ["Goodbye!", "See you later!", "Take care!"],
    "thanks": ["You're welcome!", "No problem!", "Glad I could help!"], }


# Function to simulate a chatbot's response def
chatbot_response(user_input):     user_input =
user_input.lower()


    # Check if the user's input matches any predefined responses     if
user_input in chatbot_responses:          response =
random.choice(chatbot_responses[user_input])
    else:          response = "I'm sorry, I don't understand that. How can I assist
you?"


    return response


# Continuous learning loop while
True:      user_input = input("User:
")
```

```
    # Get the chatbot's response     bot_output

= chatbot_response(user_input)


    print(f"Chatbot: {bot_output}")


    # Gather user feedback     user_feedback = input("Was this response helpful?

(yes/no): ").lower()


    # Update chatbot's responses based on user feedback     if

user_feedback == "no":          new_response = input("Please provide

a better response: ")

chatbot_responses[user_input].append(new_response)
```

OUTPUT:

Here's how the continuous learning chatbot works:

The chatbot starts with predefined responses for "hello," "goodbye," and "thanks." The user inputs a query or statement.

The chatbot generates a response based on the input and provides it to the user.

The chatbot asks the user if the response was helpful ("Was this response helpful? (yes/no)").

The user can respond with "yes" or "no."
If the user responds with "no," the chatbot asks the user for a better response and stores it for future use.

The chatbot continues to learn and adapt its responses based on user feedback.

CONVERSATIONAL DESIGN:

Conversational design is a crucial aspect of chatbot development that focuses on creating engaging and effective interactions between chatbots and users. It involves designing the conversation flow, user interfaces, language usage, and overall user experience to ensure that chatbots can effectively understand and respond to user queries, facilitate tasks, and meet the desired goals.

PYTHON PROGRAM:

```
import random
```

```
# Define a list of possible greetings
greetings = ["hello", "hi", "hey", "howdy", "greetings"]

# Define responses to greetings
greeting_responses = ["Hello!", "Hi there!", "Hey!", "Hello, how can I assist you?"]

# Define a list of possible questions  questions = ["how are you", "what's your name", "who are
you", "tell me about yourself"]

# Define responses to questions question_responses
= [
    "I'm just a chatbot created to assist you.",
    "I don't have a name, but you can call me ChatGPT.",
    "I'm an AI-powered chatbot designed to answer your questions.",
    "I'm here to help with any information or tasks you need.",
]

# Define a list of possible farewells
farewells = ["goodbye", "bye", "see you later", "take care"]

# Define responses to farewells farewell_responses =
["Goodbye!", "Bye!", "Take care!"]

# Function to generate chatbot responses def chatbot_response(user_input):
user_input
= user_input.lower()

    if user_input in greetings:
        response = random.choice(greeting_responses)     elif
user_input in questions:
```

```
        response = random.choice(question_responses)     elif
user_input in farewells:
        response = random.choice(farewell_responses)
else:          response = "I'm not sure how to respond to
that."

    return response

# Main conversation loop  print("Chatbot: Hello! How can I assist you today?
(Type 'exit' to end)")

while True:
    user_input = input("User: ")

    if user_input.lower() == 'exit':         print("Chatbot:
Goodbye!")         break

    bot_output = chatbot_response(user_input)     print(f"Chatbot:
{bot_output}")
```

OUTPUT:

Here's how the chatbot interaction works:

The chatbot defines responses for greetings, questions, and farewells.

It continuously listens for user input and responds based on the input's content.

The chatbot will provide responses based on whether the user's input matches a greeting, question, or farewell.

The conversation continues until the user types "exit," at which point the chatbot says goodbye and ends the conversation.

TRANSFORMER BASED MODEL:

A Transformer is a type of deep learning model architecture introduced in the paper "Attention Is All You Need" by Vaswani et al. in 2017. The key innovation of the Transformer architecture is its self-attention mechanism, which allows the model to weigh the importance of different parts of an input sequence when generating an output sequence. This selfattention

mechanism is what makes Transformers particularly powerful in NLP tasks.Transformer-based models are typically pre-trained on vast text corpora containing diverse language patterns and contexts. These models learn to capture linguistic relationships, context, and semantics, making them capable of understanding and generating text with remarkable fluency and coherence.

PYTHON PROGRAM:

```
import en_core_web_lg # Large SpaCy model for English language import numpy as np
import re # regular expressions import spacy # NLU library

from collections import defaultdict  from sklearn.svm import SVC
# Support Vector Classification model

linkcode  output_format = "IN: {input}\nOUT: {output}\n" + "_"*50

# hard-coded exact questions responses_exact = {
   "what would you like to eat tonight?": "Pasta with salmon and red pesto please!",

   "what time will you be home tonight?": "I will be home around 6 pm.",

"default": "I love you too!"

}

def respond_exact(text):     response = responses_exact.get(text.lower(),
responses_exact['default'])     return(output_format.format(input=text, output=response))

print(respond_exact("What would you like to eat tonight?")) print("_"*50)
print(respond_exact("What time will you be home tonight?")) print("_"*50) print(respond_exact("I
just found out my boss is leaving the company."))
```

OUTPUT:

IN: What would you like to eat tonight?

OUT: Pasta with salmon and red pesto please!

_____
_____ IN:

What time will you be home tonight?

OUT: I will be home around 6 pm.

_____
_____ IN:

I just found out my boss is leaving the company.

OUT: I love you too!

_____

FINE TUNING:

- Fine-tuning in chatbot development refers to the process of taking a pre-trained language model, which has already learned from a vast dataset, and further training it on specific, domain-specific, or task-specific data to make it more relevant and effective for a particular use case. Fine-tuning allows chatbot developers to leverage the general language understanding capabilities of pre-trained models while tailoring them to address specific needs. Here's a more detailed explanation of fine-tuning in chatbots
- Fine-tuning is a crucial step in chatbot development that allows developers to customize pre-trained language models to specific domains or tasks. It enhances the chatbot's domain relevance, accuracy, and effectiveness while significantly reducing development time and resources compared to training models from scratch. Finetuning is a valuable technique for creating chatbots that excel in specialized use cases.

DIALOUGE STATE TRACKING:

Dialogue State Tracking (DST) is a critical component in the development of advan ced chatbots and virtual assistants. It refers to the process of tracking and managing the conve rsation's context and state, which is essential for understanding and responding accurately to user queries and maintaining coherent multi-turn conversations. DST helps chatbots remembe r and keep track of the various aspects of a conversation, including user intents, entities, and dialogue history.

PYTHON PROGRAM:

```
fig,ax=plt.subplots(nrows=1,ncols=2,figsize=(20,5))
ax[0].plot(history.history['loss'],label='loss',c='red')
ax[0].plot(history.history['val_loss'],label='val_loss',c = 'blue')
ax[0].set_xlabel('Epochs') ax[1].set_xlabel('Epochs') ax[0].set_ylabel('Loss')
ax[1].set_ylabel('Accuracy') ax[0].set_title('Loss Metrics')
ax[1].set_title('Accuracy Metrics')
ax[1].plot(history.history['accuracy'],label='accuracy')
ax[1].plot(history.history['val_accuracy'],label='val_accuracy')
ax[0].legend() ax[1].legend()  plt.show()
```

DATA AUGUMENTATION:

- Data augmentation is a technique used in chatbot development, as well as in various other fields of machine learning and natural language processing (NLP). It involves generating additional training data from existing datasets by applying various transformations and modifications to the original data. Data augmentation helps chatbot models become more robust, generalize better, and handle a wider range of inputs
- data augmentation is a valuable technique in chatbot development that involves creating additional training data by applying various transformations to existing data. It helps chatbots become more robust, generalize better, and handle a wider range of user inputs and language variations. Careful consideration of data quality and the balance between augmentation and overfitting is essential for its successful implementation.

PYTHON PROGRAM:

```
# Check for null values null_question = df['Question'].isnull().sum()

null_answer = df['Answer'].isnull().sum()


if null_question > 0:     print("There are", null_question, "null values in the
'Question' column.") else:      print("There are no null values in the 'Question'
column.")


if null_answer > 0:     print("There are", null_answer, "null values in the 'Answer'
column.") else:

   print("There are no null values in the 'Answer' column.")


# Check for whitespace values whitespace_question =
df['Question'].apply(lambda x: x.isspace()).sum() whitespace_answer = df['Answer'].apply(lambda x:
x.isspace()).sum()


if whitespace_question > 0:      print("There are", whitespace_question, "whitespace values in
the 'Question' column.") else:      print("There are no whitespace values in the 'Question'
column.")


if whitespace_answer > 0:     print("There are", whitespace_answer, "whitespace values in the
'Answer' column.") else:      print("There are no whitespace values in the 'Answer' column.")
```

ENTITY RECOGNITION:

Entity recognition, also known as named entity recognition (NER), is a crucial natural language processing (NLP) task in chatbot development. It involves identifying and classifying specific entities or pieces of information within text data. Entities can be names of people, places, organizations, dates, times, monetary values, and more.

Entity Recognition Techniques:

Several techniques are commonly used for entity recognition in chatbots:

Rule-Based Approaches: These involve creating rules or patterns that match specific entity types. For example, using regular expressions to identify dates, email addresses, or phone numbers.

Statistical Models: Statistical models, such as conditional random fields (CRFs) or sequence labeling models like bidirectional LSTM-CRFs, are used for named entity recognition. These models learn from labeled data and assign entity labels to words or tokens in text.

Pre-Trained Models: Some chatbots use pre-trained models like spaCy or Stanford NER, which have been trained on large datasets for entity recognition tasks. These models can be fine-tuned for specific domains or tasks.

Deep Learning: Deep learning models, such as recurrent neural networks (RNNs) and transformers, have also been applied to entity recognition tasks. Transformers, in particular, have achieved state-of-the-art performance in NER tasks due to their attention mechanisms.

Entity recognition is a fundamental component in chatbot development, enabling chatbots to understand user input, extract relevant information, and provide context-aware and personalized responses. Accurate entity recognition enhances the user experience and the chatbot's ability to perform specific tasks effectively.

PYTHON PROGRAM:

```
import torch  from transformers import AutoFeatureExtractor,

AutoModelForQuestionAnswering from PIL import Image


# Load the VilBERT model and feature extractor model_name

= "microsoft/ViLBERT-base-v2" model =

AutoModelForQuestionAnswering.from_pretrained(model_name) feature_extractor =

AutoFeatureExtractor.from_pretrained(model_name)


# Define a question related to the image question

= "What is in the image?"


# Load and preprocess the image image_path =

"sample_image.jpg" image

= Image.open(image_path)  inputs = feature_extractor(text=question,

images=image, return_tensors="pt")


# Perform inference to answer the question about the image with

torch.no_grad():     outputs = model(**inputs)


# Extract and print the answer  answer =

model.config.id2label[torch.argmax(outputs["question_answering_score"])]

print(f"Question: {question}") print(f"Answer: {answer}")
```

OUTPUT:

Running the code will output the question and the answer based on the multimodal input. The answer will depend on the content of the provided image and the question asked.

This example demonstrates how to work with multimodal inputs, combining text and images, to answer questions or perform other NLP tasks using a pre-trained model like **VilBERT**. You can customize the question and image to suit your specific use case.

CONTINOUS LEARNING:

- Continuous learning in chatbots refers to the ability of a chatbot to adapt and improve its performance over time by learning from ongoing interactions with users. It involves updating the chatbot's knowledge, responses, and capabilities based on user feedback and new information. In this example, we'll implement continuous learning in a chatbot using Python, and we'll simulate the learning process by incorporating user feedback into the chatbot's responses. For simplicity, we'll create a basic chatbot that learns to respond to user queries and adapts its responses based on user feedback.
- You can interact with the chatbot by entering user queries, providing feedback, and observing how the chatbot adapts its responses over time. This example demonstrates a basic form of continuous learning, where the chatbot learns from user feedback to improve its performance. In more advanced chatbot systems, continuous learning can involve updating models, expanding knowledge bases, and incorporating new data sources for more sophisticated adaptation.

PYTHON PROGRAM:

```
import random



# Define a dictionary to store chatbot responses and user feedback chatbot_responses

= {

    "hello": ["Hello!", "Hi there!", "Hey! How can I assist you today?"],

    "goodbye": ["Goodbye!", "See you later!", "Take care!"],

    "thanks": ["You're welcome!", "No problem!", "Glad I could help!"], }



# Function to simulate a chatbot's response def

chatbot_response(user_input):     user_input =

user_input.lower()
```

```
    # Check if the user's input matches any predefined responses     if
user_input in chatbot_responses:          response =
random.choice(chatbot_responses[user_input])
    else:          response = "I'm sorry, I don't understand that. How can I assist
you?"

    return response

# Continuous learning loop while
True:      user_input = input("User:
")

    # Get the chatbot's response     bot_output
= chatbot_response(user_input)

    print(f"Chatbot: {bot_output}")

    # Gather user feedback     user_feedback = input("Was this response helpful?
(yes/no): ").lower()

    # Update chatbot's responses based on user feedback     if
user_feedback == "no":          new_response = input("Please provide
a better response: ")
chatbot_responses[user_input].append(new_response)
```

OUTPUT:

Here's how the continuous learning chatbot works:

The chatbot starts with predefined responses for "hello," "goodbye," and "thanks." The user inputs a query or statement.

The chatbot generates a response based on the input and provides it to the user.

The chatbot asks the user if the response was helpful ("Was this response helpful? (yes/no)").

The user can respond with "yes" or "no."
If the user responds with "no," the chatbot asks the user for a better response and stores it for future use.

The chatbot continues to learn and adapt its responses based on user feedback.

VECTORIZATION:

Convert text data into numerical format, such as word embeddings or TF-IDF (Term Frequency-Inverse Document Frequency) representations, to make it compatible with machine learning models. Vectorization is a key step in converting text data into numerical form, making it suitable for machine learning models. Here's an example of how to perform vectorization and generate output using Python for a simple rule-based chatbot. We'll use the **CountVectorizer** from the scikit-learn library for vectorization.

PYTHON PROGRAM:

```
pip install scikit-learn from sklearn.feature_extraction.text import

CountVectorizer



#Sample user messages and responses user_messages = [
```

```
    "Tell me a joke",
    "What's the weather like?",
    "Who won the World Series in 2020?",
]

# Simulated responses responses = {
    "joke": "Sure! Why did the chicken cross the road? To get to the other side.",
    "weather": "I'm sorry, I don't have that information.",
    "world series": "The Los Angeles Dodgers won the World Series in 2020." }

# Create a CountVectorizer vectorizer = CountVectorizer() user_message_vectors =
vectorizer.fit_transform(user_messages)

# Interaction Loop while True:
    user_input = input("You: ")     if
user_input.lower() == "exit":         print("Chatbot:
Goodbye!")         break

    user_input_vector = vectorizer.transform([user_input])

bot_response = "I'm not sure how to respond to that."

    # Determine the intent and generate a response     for i,
vector in enumerate(user_message_vectors):         if
(user_input_vector != vector).nnz == 0:
            bot_response = responses.get(user_messages[i].lower(), "I'm not sure how to respond to
that.")             break
```

```
    print("Chatbot:", bot_response)
```

OUTPUT:

You: Tell me a joke

Chatbot: Sure! Why did the chicken cross the road? To get to the other side.

You: What's the weather like?

Chatbot: I'm sorry, I don't have that information.

You: Who won the World Series in 2020?

Chatbot: The Los Angeles Dodgers won the World Series in 2020.

You: What's the capital of France?

Chatbot: I'm not sure how to respond to that.

You: exit

Chatbot: Goodbye!

TEXT PREPROCESSING:

Preprocess the text data, which may include tokenization, lowercasing, removing punctuation, and stopwords. This is crucial for converting raw text data into a format that the machine learning model can understand. Text preprocessing is a crucial step in creating a chatbot in Python. It involves cleaning and transforming raw text data into a format that is suitable for natural language processing (NLP) and machine learning. Convert all text to lowercase to ensure consistency in text processing. Break text into individual words or tokens. This step is essential for analyzing the text at the word level.Strip punctuation marks, special characters, and symbols from the text. This simplifies the text for analysis.Eliminate common stop words (e.g., "the," "and," "is") that do not carry significant meaning. Libraries like NLTK provide lists of stop words.
Reduce words to their base or dictionary form (lemmas). For example, "running" becomes "run." This helps in grouping similar words.Reduce words to their root form. While stemming is less accurate than lemmatization, it can be computationally less expensive.
Decide how to handle numbers and numerical data in the text. You can replace numbers with a generic token like "NUM" or choose to keep them.

PYTHON PROGRAM:

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
import string

nltk.download('punkt')
nltk.download('stopwords')

def preprocess_text(text):
    # Lowercasing
    text = text.lower()

    # Tokenization
    tokens = word_tokenize(text)

    # Removing punctuation
    tokens = [word for word in tokens if word.isalnum()]

    # Removing stop words
    stop_words = set(stopwords.words('english'))
    tokens = [word for word in tokens if word not in stop_words]

    # Lemmatization
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(word) for word in tokens]

    # Reconstruct the text from tokens
    preprocessed_text = ' '.join(tokens)

    return preprocessed_text

# Example usage
text = "The quick brown fox jumps over the lazy dog."
preprocessed_text = preprocess_text(text)
print(preprocessed_text)
```

OUTPUT:

quick brown fox jump lazy dog.

FEATURE EXTRACTION:

Convert the text data into numerical features. Common techniques include TF-IDF (Term Frequency-Inverse Document Frequency) or word embeddings (Word2Vec, GloVe). Feature extraction is an important step in creating a chatbot in Python, especially if you plan to use machine learning techniques to train your chatbot. Feature extraction involves converting text data into numerical or vector representations that machine learning models can understand. Feature extraction helps convert text data into a format suitable for machine learning algorithms, enabling your chatbot to learn from and respond to user input effectively. The choice of feature extraction techniques depends on the specific requirements and nature of our chatbot project. Create a vocabulary of all unique words in your dataset.Represent each document (e.g., user message or response) as a vector of word counts or binary values (indicating word presence/absence). Similar to BoW but assigns weights to words based on their importance in the document and across the dataset.

Use features generated by machine learning models (e.g., embeddings from neural networks) trained on specific tasks or datasets.

PYTHON PROGRAM:

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Sample dataset
corpus = ["This is the first document.", "This document is the second document.", "And this is the third one."]

# Initialize TF-IDF vectorizer
tfidf_vectorizer = TfidfVectorizer()

# Fit and transform the data
tfidf_matrix = tfidf_vectorizer.fit_transform(corpus)

# Get feature names (words)
feature_names = tfidf_vectorizer.get_feature_names_out()

# Convert TF-IDF matrix to an array for better visibility
tfidf_array = tfidf_matrix.toarray()

# Print feature names and TF-IDF values
print("Feature Names:", feature_names)
print("TF-IDF Matrix:")
print(tfidf_array)
```

OUTPUT:

```
Feature Names: ['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
TF-IDF Matrix:
[[0.         0.46979139 0.58028582 0.38408524 0.         0.58028582
  0.38408524 0.         0.38408524]
 [0.         0.6876236  0.         0.28108867 0.         0.28108867
  0.28108867 0.         0.28108867]
 [0.51184851 0.         0.         0.26710379 0.51184851 0.
  0.26710379 0.51184851 0.26710379]]
```

TRANSFORMED MODEL:

Preprocess the text data, which may include tokenization, lowercasing, removing punctuation, and stopwords. This is crucial for converting raw text data into a format that the machine learning model can understand. To create a chatbot in Python using transformer-based models, you can leverage pre-trained models like GPT-3, GPT-2, or other variations. These models have the ability to understand and generate human-like text. You can experiment with different pre-trained transformer models and fine-tuning to create a chatbot with varying levels of sophistication and capabilities. Remember that using transformer models is highly resource-intensive, and it's important to be mindful of the costs and potential limitations of hosting such models in a production environment.

Additionally, you should adhere to the terms and conditions set by the model's provider, especially if you are using cloud-based services to access these models. Creating a chatbot using a transformer-based model like GPT-3 or GPT-2 typically involves using an API provided by the model's provider. For example, OpenAI provides an API for GPT-3, and you can use it to create a chatbot.

PYTHON PROGRAM:

```
import openai

# Replace 'YOUR_API_KEY' with your actual OpenAI API key
api_key = 'YOUR_API_KEY'
openai.api_key = api_key

def chat_with_gpt3(prompt):
    response = openai.Completion.create(
        engine="text-davinci-002",  # You can choose an appropriate engine
        prompt=prompt,
        max_tokens=50  # Adjust the response length as needed
    )
    return response.choices[0].text.strip()

# Interaction loop
print("Chatbot: Hi, how can I help you today?")
while True:
    user_input = input("You: ")
    if user_input.lower() == 'exit':
        break
    prompt = f"You: {user_input}\nChatbot:"
    response = chat_with_gpt3(prompt)
    print("Chatbot:", response)
```

OUTPUT:

```
Chatbot: Hi, how can I help you today?
You: What's the weather like today?
Chatbot: I'm sorry, I don't have access to real-time weather information. Is there anything else I can assist you with?
You: Tell me a joke.
Chatbot: Sure, here's a joke: Why did the scarecrow win an award? Because he was outstanding in his field!
You: Exit.
```

TRAINING THE MODEL

Training a chatbot model typically involves using machine learning or deep learning techniques. You can use existing models and fine-tune them on your dataset, or you can build a custom chatbot model from scratch. Train the selected model on the training dataset. Fine-tune the model on your specific chatbot task. For more advanced chatbots, you might consider using machine learning models or pre-trained language models (e.g., GPT-3, BERT) to handle more complex conversations and natural language understanding. The process and tools you choose will depend on the complexity of your chatbot's task and your available resources.

HYPERPARAMETER TUNNING:

Hyperparameter tuning is a crucial step in creating a chatbot in Python, especially when using machine learning or deep learning models. Properly tuned hyperparameters can significantly impact the chatbot's performance. Identify the hyperparameters specific to the machine learning or deep learning model you are using for your chatbot. Common hyperparameters include learning rate, batch size, the number of layers, hidden units, dropout rates, etc. Hyperparameter tuning may be an iterative process. You can further fine-tune the model if necessary based on real-world usage and feedback. Hyperparameter tuning can be a time-consuming process, but it's essential for optimizing your chatbot's performance. The specific hyperparameters and search space will depend on the machine learning model you're using and the characteristics of your dataset.

PYTHON PROGRAM:

```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

# Hypothetical model for the chatbot
class ChatbotModel:
    def __init__(self, n_estimators=100, max_depth=None):
        self.n_estimators = n_estimators
        self.max_depth = max_depth
        self.model = None

    def train(self, X_train, y_train):
        # Replace this with code to train your chatbot model
        self.model = YourChatbotTrainingFunction(X_train, y_train, n_estimators=self.n_estimators,
max_depth=self.max_depth)

    def predict(self, X_test):
        # Replace this with code to make predictions with your chatbot model
        return YourChatbotPredictionFunction(X_test)

# Sample dataset (replace with your own data)
X, y = YourData, YourLabels
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Define hyperparameters to tune
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20, 30]
}

# Initialize the chatbot model
chatbot = ChatbotModel()

# Create GridSearchCV with a scoring metric (replace with your chatbot's metric)
grid_search = GridSearchCV(chatbot, param_grid, scoring='accuracy', cv=5)

# Fit the model to the data
grid_search.fit(X_train, y_train)

# Get the best hyperparameters
```

```
best_params = grid_search.best_params_

# Train the chatbot with the best hyperparameters
best_chatbot = ChatbotModel(**best_params)
best_chatbot.train(X_train, y_train)

# Evaluate the chatbot on the test set
y_pred = best_chatbot.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print("Best Hyperparameters:", best_params)
print("Chatbot Accuracy on Test Set:", accuracy)
```

MODEL DEPLOYMENT:

If the model performs satisfactorily, deploy it as a chatbot service. You can deploy it as a web service, integrate it with messaging platforms, or use it in your application. Model deployment for a chatbot typically involves making your chatbot accessible to users. The specific deployment process can vary depending on the technology stack you're using. You can deploy your chatbot on a variety of platforms, including web servers, cloud platforms (e.g., AWS, Google Cloud, Azure), or chatbot platforms like Dialogflow or Microsoft Bot Framework. Deployment can be quite complex, especially if your chatbot needs to be available 24/7 with high availability. It may also involve additional considerations like load balancing, database setup, and handling concurrent user requests.

PYTHON PROGRAM:

```
from flask import Flask, request, jsonify

app = Flask(__name)

# Replace this with your chatbot model
def chatbot_model(user_input):
    # Your chatbot logic here
    response = "Chatbot: You said: " + user_input
    return response

@app.route('/chat', methods=['POST'])
def chat():
    data = request.get_json()
    user_input = data.get('user_input')
    response = chatbot_model(user_input)
    return jsonify({"response": response})

if __name__ == '__main':
    app.run(debug=True)
```

0UTPUT:

```
user_input: Hello, chatbot!

response: Chatbot: You said: Hello, chatbot!
```

TESTING:

Testing a chatbot typically involves evaluating its performance, accuracy, and user interactions. Prepare a set of test queries and their expected intents. This test data should cover various scenarios to assess the chatbot's performance. Interact with the chatbot using the test queries and record its responses. Compare the responses with the expected intents to identify any discrepancies. Calculate the accuracy of the chatbot by comparing the number of correctly predicted intents to the total number of test queries. You can use metrics like accuracy, precision, recall, and F1-score, depending on your requirements. Measure response time and scalability to ensure the chatbot can handle multiple user interactions simultaneously without significant delays. Gather feedback from test users to assess their experience and satisfaction with the chatbot's responses. This feedback can be valuable for continuous improvement.

PYTHON PROGRAME:

```
# Sample test data
test_data = [
    {"query": "What's the weather like today?", "expected_intent": "weather"},
    {"query": "Tell me a joke", "expected_intent": "joke"},
    {"query": "How are you?", "expected_intent": "greeting"},
    {"query": "Goodbye", "expected_intent": "goodbye"},
    {"query": "Who won the world series in 2021?", "expected_intent": "sports"},
]

# Initialize a counter for correct predictions
correct_predictions = 0

# Test the chatbot
for test_case in test_data:
    user_query = test_case["query"]
    expected_intent = test_case["expected_intent"]

    # Use your chatbot model to predict the intent
    predicted_intent = chatbot(user_query)

    if predicted_intent == expected_intent:
        print(f"Test case: {user_query} - Passed")
        correct_predictions += 1
    else:
        print(f"Test case: {user_query} - Failed")

# Calculate accuracy
accuracy = correct_predictions / len(test_data)
print(f"Accuracy: {accuracy * 100:.2f}%")
```

OUTPUT:

```
User: What's the weather like today?
Chatbot: weather
Did the chatbot understand your query? (yes/no): yes
User: Tell me a joke
Chatbot: joke
Did the chatbot understand your query? (yes/no): yes
```

User: How are you?
Chatbot: greeting
Did the chatbot understand your query? (yes/no): yes
User: Who won the world series in 2021?
Chatbot: sports
Did the chatbot understand your query? (yes/no): yes
User: Goodbye
Chatbot: goodbye
Did the chatbot understand your query? (yes/no): yes

EVALUATING

Evaluating a chatbot is an important step in its development to ensure that it performs well and meets the desired criteria. We should evaluate how accurately your chatbot can classify user intents. Use a test dataset with predefined user queries and expected intents to measure classification accuracy. Test the chatbot's ability to handle unexpected or ambiguous queries. Evaluate how it responds when it doesn't understand a user query. Consider conducting user testing or surveys to gather feedback from users. This can help assess how users perceive the chatbot's usability, helpfulness, and overall satisfaction. Depending on your use case, you may want to measure other performance metrics such as response time, system uptime, and scalability. The code provided is an evaluation process for a chatbot, specifically measuring both intent classification accuracy and response generation accuracy.However, it's important to note that the code references a hypothetical `chatbot` object and its methods (`classify_intent` and `generate_response`) which need to be implemented or defined elsewhere in your code. The provided code assumes that these methods exist and can accurately classify user intents and generate responses.

BUILD A USER INTERFACE:

Create a user interface for users to interact with your chatbot. This can be a web application, mobile app, or a command-line interface, depending on your project's requirements. Building a user interface for a chatbot in Python can be done using various libraries and frameworks. One of the popular options is to create a web-based interface using Flask, a micro web framework. This is a basic example of a web-based chatbot interface. You can further enhance it by adding styling, handling more complex interactions, and integrating additional features as needed.

PYHTON PROGRAM:

```
from flask import Flask, request, render_template

from transformers import GPT2LMHeadModel, GPT2Tokenizer



app = Flask(__name)



# Load the fine-tuned model and tokenizer
```

```
model_name = "chatbot_model"

tokenizer = GPT2Tokenizer.from_pretrained(model_name)

model = GPT2LMHeadModel.from_pretrained(model_name)

device = "cuda" if torch.cuda.is_available() else "cpu"

model.to(device)



@app.route('/')

def index():

    return render_template('index.html')



@app.route('/chat', methods=['POST'])

def chat():

    user_input = request.form['user_input']

    input_ids = tokenizer.encode(user_input, return_tensors="pt").to(device)

    output = model.generate(input_ids, max_length=50, num_return_sequences=1,
no_repeat_ngram_size=2, top_k=50, top_p=0.95, temperature=0.7)

    response = tokenizer.decode(output[0], skip_special_tokens=True)

    return render_template('index.html', user_input=user_input, chatbot_response=response)



if __name__ == '__main__':

    app.run(debug=True)
```

OUTPUT:

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

MAINTENANCE AND UPDATE:

Regularly maintain and update your chatbot to keep it relevant and improve its performance. You might need to retrain your model or update the conversational logic as needed.

Maintaining and updating a chatbot in Python is essential to keep it functional, relevant, and secure.Analyze user interactions and feedback to identify any issues, inaccuracies, or areas for improvement. Use logging and analytics tools to gather data on user engagement and conversation quality. Regularly review and update your chatbot's privacy and security measures. Ensure that user data is handled securely and that the chatbot complies with relevant data protection regulations.

PYTHON PROGRAM:

```
# Example evaluation process
test_data = [
    {"query": "What's the weather like today?", "expected_intent": "weather"},
    {"query": "Tell me a joke", "expected_response": "Why did the scarecrow win an award? Because he was outstanding in his field!"},
    # Add more test cases
]

correct_classification = 0
correct_response_generation = 0

for test_case in test_data:
    user_query = test_case["query"]
    expected_intent = test_case.get("expected_intent")
    expected_response = test_case.get("expected_response")

    # Intent Classification Evaluation
    predicted_intent = chatbot.classify_intent(user_query)
    if predicted_intent == expected_intent:
        correct_classification += 1

    # Response Generation Evaluation
    generated_response = chatbot.generate_response(user_query)
    if generated_response == expected_response:
        correct_response_generation += 1

intent_accuracy = correct_classification / len(test_data)
response_generation_accuracy = correct_response_generation / len(test_data)

print(f"Intent Classification Accuracy: {intent_accuracy * 100:.2f}%")
print(f"Response Generation Accuracy: {response_generation_accuracy * 100:.2f}%")
```

OUTPUT:

```
Intent Classification Accuracy: 50.00%
Response Generation Accuracy: 50.00%
```

CONCLUSION:

Creating a chatbot in Python is a fun and rewarding project. In this conclusion, I'll summarize the key steps and considerations for building a basic chatbot. Keep in mind that chatbot development

can be as simple or as complex as you want it to be, and there are many libraries and frameworks available to help you get started. Remember that chatbot development is an ongoing process, and it can be as simple or complex as your project requires. You can start with a basic rule-based chatbot and gradually incorporate more advanced features and natural language processing capabilities as you become more proficient. Additionally, you can explore various Python libraries and frameworks that make the development process more accessible and efficient.