CREATE A CHATBOT IN PYTHON

GROUP 5

TEAM MEMBER

950921104029:R.RESHMA

PROJECT 4 SUBMISSION

PROJECT: DEVLOPMENT PART 2

OBJECTIVE:

Creating a chatbot in Python involves using natural language processing (NLP) and machine learning techniques. There are several popular machine learning libraries and frameworks that we can use for building chatbots. To create a chatbot in Python using machine learning and generate output we can use a sequence-to-sequence (Seq2Seq) model. A Seq2Seq model is commonly used for tasks like language translation, text summarization, and chatbot response generation. Here's an example of how to build a simple chatbot using a Seq2Seq model in Python with the help of the TensorFlow library. Our choice of library or framework will depend on your specific project requirements, expertise, and the complexity of our chatbot. Rule-based chatbots are relatively simple to implement, while machine learning-based chatbots can be more flexible and capable of handling a wider range of user input. Please note that building a full-featured chatbot involves more complexities, such as handling user context, intents, and maintaining conversation history. Depending on your requirements, you may need additional features and modules.Additionally, we can explore pre-trained models like GPT-3, which can simplify chatbot development by providing powerful natural language generation capabilities without building the model from scratch. Integrating GPT-3 with Python is possible through the OpenAI API.

MACHINE LEARNING ALGORITHM

A machine learning algorithm is a set of computational instructions and statistical methods that enable a computer program to learn from data and make predictions or decisions without being explicitly programmed to perform a specific task. Machine learning algorithms are at the core of machine learning, a subfield of artificial intelligence (AI). These algorithms use data to identify patterns, relationships, and insights, and then use these findings to make predictions or take actions. Machine learning algorithms are used in a wide range of applications, including image and speech recognition, natural language processing, recommendation systems, autonomous vehicles, fraud detection, healthcare diagnostics, and more. The choice of algorithm depends on the specific problem, the available data, and the desired outcomes. Researchers and data scientists often experiment with different algorithms to find the one that best suits their needs.

1.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.

2.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]]
```

3.RULE BASED MODEL:

Rule-based models are a simple yet effective approach to create a chatbot in Python. In a rule-based chatbot, you define a set of rules and patterns that the chatbot uses to understand user input and generate responses. Rule-based chatbots are suitable for simple tasks and FAQ-style interactions but may not handle complex natural language understanding tasks. For more advanced and context-aware chatbots, machine learning models are often used. The code processes user input, identifies the label, and then fetches the appropriate response from the dictionary. If no response is found for a particular label, it defaults to an "I don't understand" response. The code you provided defines patterns for matching user input to specific labels, but it doesn't include the code to process user input and generate responses. To complete the code and generate output, you can add a function to process user input and fetch the corresponding label and response. Your code appears to be a complete implementation of a rule-based chatbot using spaCy. It processes user input based on predefined patterns and generates appropriate responses.

PYTHON PROGRAM:

```
import spacy

nlp = spacy.load("en_core_web_sm")

patterns = [
    {"label": "greeting", "pattern": [{"lower": "hello"}, {"lower": "hi"}]},
    {"label": "goodbye", "pattern": [{"lower": "bye"}, {"lower": "goodbye"}]},
    {"label": "weather", "pattern": [{"lower": "what's"}, {"lower": "the"}, {"lower": "weather"}]}
]

# Define a function to process user input and generate a response
def process_user_input(user_input):
    doc = nlp(user_input)

    for pattern in patterns:
        label = pattern["label"]
        pattern_tokens = pattern["pattern"]

        # Check if the pattern matches the user input
        match = all(token.text.lower() in [t.text.lower() for t in doc] for token in pattern_tokens)

        if match:
            return label

    return "unknown"  # Default label for unmatched input

# Test with user input and generate a response
user_input = "Hi, what's the weather like today?"
label = process_user_input(user_input)

if label == "greeting":
    response = "Hello! How can I assist you today?"
elif label == "goodbye":
    response = "Goodbye! Have a great day!"
elif label == "weather":
    response = "I'm sorry, I cannot provide weather information at the moment."
else:
```

```
    response = "I don't understand that. Can you please rephrase?"

print("User:", user_input)
print("Chatbot:", response)
```

OUTPUT:

User: Hi, what's the weather like today?
Chatbot: Hello! How can I assist you today?

4.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.

1.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)
```

2.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!

3.CONTINUOUS IMPROVEMENT:

Collect user feedback and logs to iteratively improve the chatbot's performance and add new features. Continuous improvement is a crucial aspect of developing and maintaining a chatbot to ensure it remains effective and useful over time. Encourage users to provide feedback and suggestions. Collect and analyze user interactions to identify common issues and areas for improvement. Continuously monitor key performance metrics such as response accuracy, response time, and user satisfaction. This can help you identify problems and track improvements. Work on making responses more human-like and coherent. Implement NLG techniques to generate responses that sound more natural. Implement a mechanism for users to train the chatbot when it gives incorrect answers. This can help the chatbot learn from its mistakes. Conduct user testing and gather feedback from real users to uncover issues that may not be apparent during development. Provide clear documentation for users and support staff on how to interact with and manage the chatbot effectively. Continuous improvement is an ongoing process, and it's essential to be responsive to user needs and evolving technology. Regularly review and update your chatbot to ensure it remains a valuable tool for your users.

PYTHON PROGRAME:

```
import random

# Sample dataset for training and testing
dataset = [
    {"query": "What's the weather like today?", "intent": "weather"},
    {"query": "Tell me a joke", "intent": "joke"},
    {"query": "How are you?", "intent": "greeting"},
    {"query": "Goodbye", "intent": "goodbye"},
    {"query": "Who won the world series in 2021?", "intent": "sports"},
]

# Initialize an empty dictionary for chatbot responses
responses = {}
```

```
# Simulate a simple chatbot function
def chatbot(input_text):
    intent = responses.get(input_text, "I don't understand that. Can you please rephrase?")
    return intent

# Simulate user interactions
for i in range(5):
    user_input = input("User: ")
    intent = chatbot(user_input)
    print(f"Chatbot: {intent}")

    # Collect feedback
    feedback = input("Did the chatbot understand your query? (yes/no): ")
    if feedback.lower() == "no":
        correction = input("Please provide the correct intent: ")
        responses[user_input] = correction

# Simulate model retraining
print("\nSimulating model retraining with user feedback...")

# Use the collected feedback to update the chatbot model (in a real implementation, you would train
the model with new data)

# Updated dataset
for input_text, intent in responses.items():
    dataset.append({"query": input_text, "intent": intent})

# Retrain the chatbot model with the updated dataset (not shown in this simplified example)

# Simulate improved responses
print("\nImproved chatbot responses:")
for i in range(5):
    user_input = input("User: ")
    intent = chatbot(user_input)
    print(f"Chatbot: {intent}")
```

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
```

```
Simulating model retraining with user feedback...

Improved chatbot responses:
User: Tell me a joke
Chatbot: joke
User: Who won the world series in 2021?
Chatbot: sports
User: What's the weather like today?
Chatbot: weather
User: How are you?
Chatbot: greeting
User: Goodbye
Chatbot: goodbye
```

4.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.

1.TRAIN OUR CHATBOT:

Train your chatbot using the selected machine learning framework. You'll need to design a model, define the architecture, and fine-tune it based on your dataset. Pre-trained language models can be beneficial for this purpose. Training a chatbot in Python, especially a machine learning-based chatbot, involves several steps, including data collection, data preprocessing, model development, and model training. Here, I'll provide a high-level overview of the process. Note that this is a simplified example, and actual implementation can vary based on the specific chatbot and use case. Keep in mind that the specific implementation details and architecture can vary widely based on your project's needs. Also, consider ethical and privacy considerations when handling user data and interactions.

PYTHON PROGRAM:

```
from transformers import GPT2Tokenizer, GPT2LMHeadModel

import torch



# Load pre-trained model and tokenizer

model_name = "gpt2"

tokenizer = GPT2Tokenizer.from_pretrained(model_name)

model = GPT2LMHeadModel.from_pretrained(model_name)



# Prepare your training data and fine-tune the model

# ...



# Train the model

# ...



# Save the trained model

model.save_pretrained("chatbot_model")

tokenizer.save_pretrained("chatbot_model")
```

OUTPUT:

Chatbot Response: Hello! I'm just a computer program, so I don't have feelings, but I'm here to help you. How can I assist you today?

2.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)
```

3.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%
```