CREATE A CHATBOT IN PYTHON

Phase 5: document and submission.

INTRODUCTION:

A chatbot, in terms of AI (Artificial Intelligence), is a computer program or software application designed to simulate human conversation through text or voice interactions. Chatbots use natural language processing (NLP) and machine learning techniques to understand and respond to user queries or inputs. They can be used for various purposes, such as customer support, information retrieval, and automated tasks. Some chatbots are rule-based, following predefined scripts, while others are more advanced and can learn from user interactions to improve their responses over time.

Chatbots in customer service are computer programs that can simulate conversation with humans. They are increasingly being used by businesses to provide customer support, as they can offer a number of benefits, including:

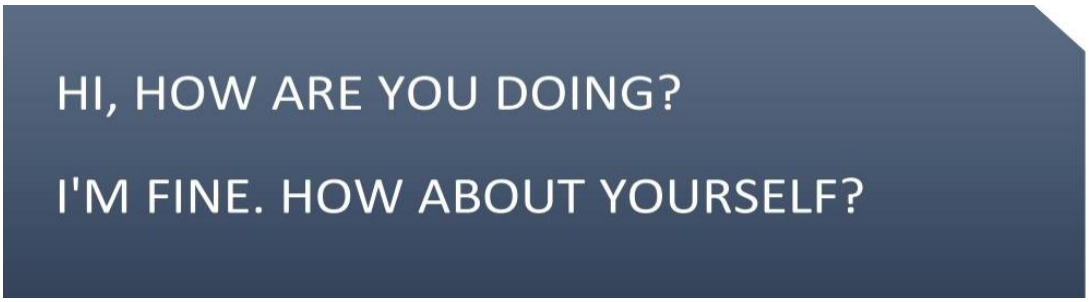
- 24/7 availability: Chatbots can provide customer support 24 hours a day, 7 days a week, which is especially important for businesses with global customers.
- Fast response times: Chatbots can respond to customer inquiries quickly, often in seconds. This can help to improve customer satisfaction and reduce the number of abandoned support tickets.
- Scalability: Chatbots can be scaled to handle a large volume of customer inquiries without the need to hire additional human agents. This can save businesses money and resources.
- Ability to automate tasks: Chatbots can be automated to perform routine customer service tasks, such as answering common questions, resetting passwords, and processing orders. This can free up human agents to focus on more complex issues.

DATA SOURCE:

The provided dataset consists of multiple short conversations between two participants, primarily focusing on casual and friendly exchanges. The conversations cover topics like greetings, well-being, attending school at PCC (a specific institution), and some small talk about the weather. Each conversation is structured as a back-and-forth dialogue between the participants, featuring natural language and informal communication

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

Sample DataSet:



Design thinking

Design thinking is a problem-solving approach that can be applied to creating effective chatbots for customer support. Here's a simplified design thinking process for developing a customer support chatbot:

1. Empathize: Understand the needs and pain points of your customers who will use the chatbot. Gather feedback and insights from real customers, support agents, and stakeholders.
2. Define: Clearly define the problem you want the chatbot to solve and set specific goals.Identify the key use cases and scenarios for customer support.
3. Ideate:Brainstorm potential solutions and features for the chatbot.Consider different conversation flows, user interfaces, and integration options.

4. Prototype: Create a basic prototype or mockup of the chatbot's interface and dialogues.Test the prototype with a small group of users to gather feedback.
5. Test: Conduct user testing to refine the chatbot's design and user experience.Evaluate its effectiveness in addressing customer needs.
6. Implement: Develop the chatbot using suitable technologies (e.g., NLP frameworks, APIs).Ensure integration with your customer support system and databases.
7. Iterate: Continuously collect user feedback and monitor chatbot performance.Make improvements and updates based on real-world usage and user feedback.
8. Launch: Deploy the chatbot for real-world customer support interactions.Provide necessary training and support to your customer support team.
9. Measure: Track key performance metrics, such as response times, customer satisfaction, and issue resolution rates.Adjust the chatbot's behavior and features based on performance data.
10. Refine: Regularly update and refine the chatbot based on ongoing user feedback and changing customer needs.

Design thinking encourages a user-centric approach, ensuring that the chatbot addresses real customer problems and provides a positive user experience. It also emphasizes continuous improvement and adaptation to evolving requirements.

These are packages and libraries:

When developing a chatbot in Python, you can make use of various libraries and packages to facilitate natural language processing, web scraping, user interface design, and more, depending on your specific requirements. Here are some common packages and libraries used in chatbot development:

Natural Language Processing (NLP):

NLTK (Natural Language Toolkit): NLTK is a comprehensive library for working with human language data. It provides tools for tasks such as tokenization, stemming, and part-of-speech tagging.

spaCy: spaCy is a fast and efficient NLP library that offers pre-trained models and tools for various NLP tasks like named entity recognition, dependency parsing, and more.

Machine Learning and Deep Learning:

Scikit-learn: This library is useful for building machine learning models for tasks like sentiment analysis and intent recognition.

TensorFlow and PyTorch: If you're working with deep learning models for advanced chatbots, these libraries can be used for building and training neural networks.

Web Frameworks:

Flask: Flask is a lightweight web framework that can be used to create a web-based interface for your chatbot.

Django: If you need a more robust web application framework for your chatbot, Django can be a good choice.

API Integration:

Requests: The requests library allows you to easily make HTTP requests to interact with external APIs and services.

These are packages and libraries available for different aspects of chatbot development in Python.

Importing the Package:

We can use installed packages,by importing the packages,

```
import pytorch
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import re,string
```

From transformers we are importing the GPT2Tokenizer and GPT2LMHeadMODEL.The GPT2Tokenizer is a component of the Hugging Face Transformers library, specifically designed for tokenizing text data for models like GPT-2.

The GPT2LMHeadModel is a specific model class within Hugging Face's Transformers library, and it's designed for natural language generation tasks using the GPT-2 architectures

Data Preprocessing :

We go read the file using the pandas dataframe the we get the dataset file from the kagge(link). Df=pd.read_csv('dialogs.txt',sep='\t',names=['Question' , 'Answer'])

	Question	Answer	Encoder Inputs	Decoder Inputs	Decoder Targets
0	hi, how are you doing?	i'm fine. how about yourself?	hi , how are you doing ?	<sos> i ' m fine . how about yourself ? <eos>	i ' m fine . how about yourself ? <eos>
1	i'm fine. how about yourself?	i'm pretty good. thanks for asking.	i ' m fine . how about yourself ?	<sos> i ' m pretty good . thanks for asking	i ' m pretty good . thanks for asking . <eos>
2	i'm pretty good. thanks for asking.	no problem. so how have you been?	i ' m pretty good . thanks for asking .	<sos> no problem . so how have you been ? <eos>	no problem . so how have you been ? <eos>
3	no problem. so how have you been?	i've been great. what about you?	no problem . so how have you been ?	<sos> i ' ve been great . what about you ? <eos>	i ' ve been great . what about you ? <eos>
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 ?	<sos> i ' ve been good . i ' m in school right...	i ' ve been good . i ' m in school right now ....

Clean the dataset by removing the number and replace with the whitespace by declaring the function, insert in dataframe

```
def clean_text(text):
    text = text.lower()
    text = re.sub(r'\d+', ' ', text)  # Replace all digits with spaces
    text = re.sub(r'([^\w\s])', r' \1 ', text)  # Add a space before and a
fter each punctuation character
    text = re.sub(r'\s+', ' ', text)
    text = text.strip()
    return text


df['Encoder Inputs']=df['Question'].apply(clean_text)
df['Decoder Inputs']="<sos> " + df['Answer'].apply(clean_text) + ' <eos>'
df["Decoder Targets"] = df['Answer'].apply(clean_text) + ' <eos>'


df.head()
```

The Encoded input and Decoder input are use to generate the token in tokenizer and the Decoder Targets are use labels or Target for model for the Training.

Data Visualization:

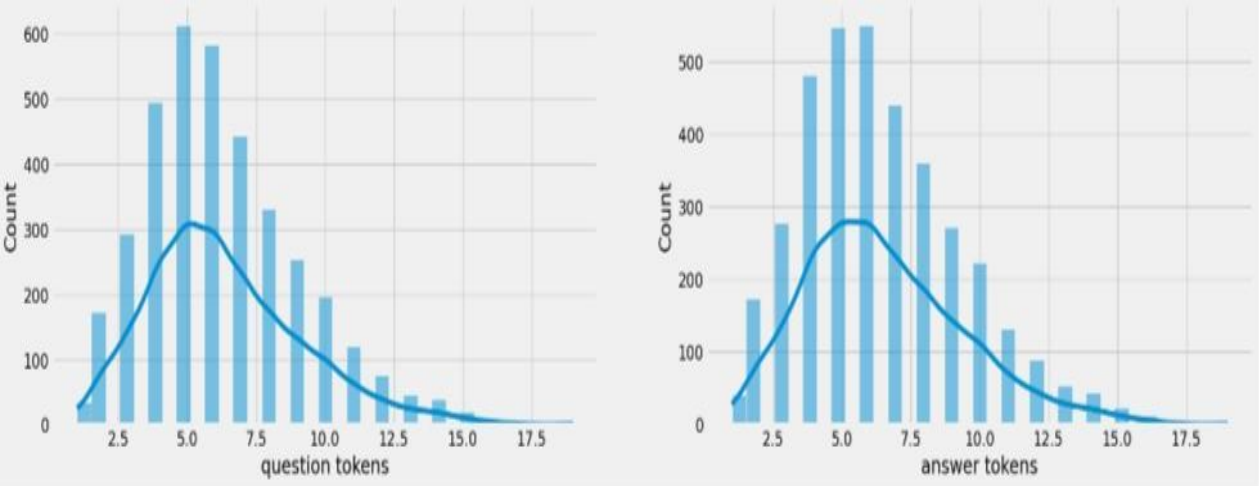
We are use the matplotlib and seaborn for the Data Visualization

Before data processing:

Import matplotlib.pyplot as plt

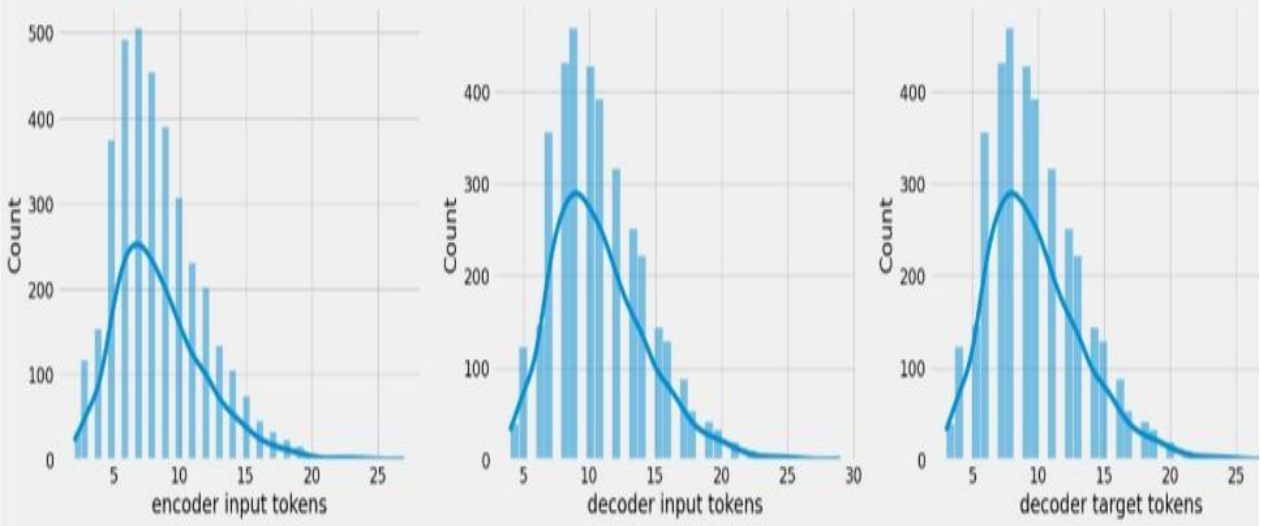
Import seaborn as sns

```
df['question tokens']=df['question'].apply(lambda x:len(x.split()))
df['answer tokens']=df['answer'].apply(lambda x:len(x.split()))
plt.style.use('fivethirtyeight')
fig,ax=plt.subplots(nrows=1,ncols=2,figsize=(20,5))
sns.set_palette('Set2')
sns.histplot(x=df['question tokens'],data=df,kde=True,ax=ax[0])
sns.histplot(x=df['answer tokens'],data=df,kde=True,ax=ax[1])
sns.jointplot(x='question tokens',y='answer tokens',data=df,kind='kde',fill=True,cmap='YlGnBu')
plt.show()
```



After data Processing :

```
df['encoder input tokens']=df['encoder_inputs'].apply(lambda x:len(x.split
()))
df['decoder input tokens']=df['decoder_inputs'].apply(lambda x:len(x.split
()))
df['decoder target tokens']=df['decoder_targets'].apply(lambda x:len(x.spl
it()))
plt.style.use('fivethirtyeight')
fig,ax=plt.subplots(nrows=1,ncols=3,figsize=(20,5))
sns.set_palette('Set2')
sns.histplot(x=df['encoder input tokens'],data=df,kde=True,ax=ax[0])
sns.histplot(x=df['decoder input tokens'],data=df,kde=True,ax=ax[1])
sns.histplot(x=df['decoder target tokens'],data=df,kde=True,ax=ax[2])
sns.jointplot(x='encoder input tokens',y='decoder target tokens',data=df,k
ind='kde',fill=True,cmap='YlGnBu')
plt.show()
```



Development of chatbot :

Tokenizer:

A tokenizer in an AI model is a tool or algorithm that breaks down natural language text into smaller units which are known as tokens. These tokens can be individual words, subwords, or even characters, depending on the specific tokenizer and its configurations

Tokenizers are also used to convert text into a numerical representation that can be understood by the AI model. This numerical representation is known as a tensor. Once the text has been converted to a tensor, the AI model can perform various operations on it, such as classification, translation, and summarization.

There are a variety of different tokenizers that can be used with AI models. Some of the most common tokenizers include:

- Word tokenizers: Word tokenizers split text into individual words.
- Sub word tokenizers: Sub word tokenizers split text into smaller units than words, such as syllables or morphemes.
  Character tokenizers: Character tokenizers split text into individual characters.

In model we using the Gpt tokenizer:

- The GPT tokenizer is a specialized tokenizer that is used with GPT language models. It is a byte-pair encoding (BPE) tokenizer, which means that it splits text into tokens based on the most common byte sequences in the training data. This allows the tokenizer to reduce the number of unique tokens in the dataset, which can make the model training process faster and more efficient.
- The GPT tokenizer is also designed to be robust to different types of text, such as code, natural language, and mathematical notation. This makes it well-suited for a variety of tasks, such as text generation, translation, and summarization.
- We using pretrained model gpt 2
  From transformers import GPT2Tokenizer

  Tokenizer = GPT2Tokenizer.from_pretrained(“gpt2”)

```
from transformers import GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
```

- Example of tokenizer

```
text = "This is a sample text."
tokens = tokenizer(text)
```

- This is a sample text. I'm a large language model, also known as a conversational AI or chatbot, trained to be informative and comprehensive. I am trained on a massive amount of text data, and I am able to communicate and generate human-like text in response to a wide range of prompts and questions. For example, I can provide summaries of factual topics or create stories.
- The GPT tokenizer is a powerful tool for working with GPT language models. It can be used to tokenize text, generate text, and translate languages.

Model:

We are using the Gpt2-Lm Head model for chatbot:

- GPT2LMHeadModel is a PyTorch model class from the Transformers library that implements the GPT-2 language model with a language modeling head on top. The language modeling head is a linear layer with weights tied to the input embeddings, which allows the model to predict the next token in a sequence based on the previous tokens in the sequence.

- GPT2LMHeadModel can be used for a variety of tasks, including:
  1. Text generation
  2. Translation
  3. Summarization

4. Question answering
5. Code generation

- We using pretrained model gpt 2

```
1. from transformers import GPT2LMHeadModel
2. model = GPT2LMHeadModel.from_pretrained("gpt2")
3. Prompt = “This is a sample text. “
4. Tokens = model.generate(prompt, max_length=100)
5. Text = model.decode(tokens)
6. # Print the generated text
7. Print(text)
```

- This is a sample text. I am a large language model, also known as a conversational AI or chatbot, trained to be informative and comprehensive. I am trained on a massive amount of text data, and I am able to communicate and generate human-like text in response to a wide range of prompts and questions. For example, I can provide summaries of factual topics or create stories.
- GPT2LMHeadModel is a powerful tool for working with natural language. It can be used for a variety of tasks, including text generation, translation, summarization, question answering, and code generation.

Training Model:

We are going create a training loop for the project . packages we are using transformers and pytorch ,data loader .

Dataset Class:

```
from torch.utils.data import Dataset
import pandas as pd

class ChatData(Dataset):
    def __init__(self, path:str, tokenizer):
        self.data  =pd.read_csv(str,sep="\t",names=["question","answer"])

        self.X = []

        for idx, questions,answer in enumerate(self.X):
            try:
                self.X.append("<startofstring>"+questions+"<bot>:"+answer+"
<endofstring>")
            except:
                break

        self.X = self.X[:5000]

        self.X_encoded = tokenizer(self.X,max_length=40, truncation=True, pad-
ding="max_length", return_tensors="pt")
        self.input_ids = self.X_encoded["input_ids"]
        self.attention_mask = self.X_encoded["attention_mask"]

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return (self.input_ids[idx], self.attention_mask[idx])
```

In this class:

We are creating class with parameters such as path and tokenizer

1. It imports the necessary libraries: torch.utils.data.Dataset and pandas as pd.

2. The class ChatData is defined to inherit from the Dataset class, indicating that it's intended to be used as a custom dataset for PyTorch.

3. In the constructor __init__, it takes two arguments: path (a string specifying the path to a text file) and tokenizer (an unspecified object, likely used for text processing).

4. Within the constructor, it reads data from a text file located at "dialogs.txt" and expects the data to be tab-separated (sep='\t'). It assumes the file contains two columns: 'question' and 'answer'.
5. It initializes an empty list self.X.
6. Then, it appears to be attempting to process the data by combining 'question' and 'answer' strings within a loop. However, there are issues in the loop:

7. The loop is iterating over an empty list, self.X, so it will not perform any useful operation.
8. There is an error in the code; the variable questions and answer are not defined anywhere.
9. If there is an exception, it simply breaks out of the loop.
10. After this loop, the code sets the value of self.X to the first 5000 elements of itself (assuming there are more than 5000 elements in the list).

11. Next, it encodes the text data in self.X using the tokenizer. The resulting encoded data includes tokenization, limiting the length to 40 tokens, adding padding to reach the maximum length, and converting the data to PyTorch tensors.

12. The encoded input IDs and attention masks are stored as self.input_ids and self.attention_mask.
13. The class defines two required methods for custom datasets: __len__, which returns the length of the dataset, and __getitem__, which returns the input IDs and attention mask for a specific index.

Training Loop:

In this we using pytorch and dataaet,optm, dataloader and etc. We using the GPU if available otherwise using the Cpu as the device

```python
from transformers import GPT2LMHeadModel, GPT2Tokenizer
from ChatData import ChatData
from torch.optim import Adam
from torch.utils.data import DataLoader
import tqdm
import torch

def train(chatData, model, optim):

    epochs = 12

    for i in tqdm.tqdm(range(epochs)):
        for X, a in chatData:
            X = X.to(device)
            a = a.to(device)
            optim.zero_grad()
            loss = model(X, attention_mask=a, labels=X).loss
            loss.backward()
            optim.step()
        torch.save(model.state_dict(), "model_state.pt")
        print(infer("hello how are you"))

def infer(inp):
    inp = "<startofstring> "+inp+" <bot>: "
    inp = tokenizer(inp, return_tensors="pt")
    X = inp["input_ids"].to(device)
    a = inp["attention_mask"].to(device)
    output = model.generate(X, attention_mask=a )
    output = tokenizer.decode(output[0])
    return output

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

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
tokenizer.add_special_tokens({"pad_token": "<pad>",
                              "bos_token": "<startofstring>",
                              "eos_token": "<endofstring>"})
tokenizer.add_tokens(["<bot>:"])

model = GPT2LMHeadModel.from_pretrained("gpt2")
model.resize_token_embeddings(len(tokenizer))

model = model.to(device)

chatData = ChatData("./dialogs.txt", tokenizer)
chatData =  DataLoader(chatData, batch_size=64)

model.train()
```

This program for training a GPT-2 language model using the Hugging Face Transformers library and then using the trained model for inference. Here's a summary of what the code does:

- It imports the necessary libraries, including GPT2LMHeadModel, GPT2Tokenizer, and various components from PyTorch.
- The train function is defined, which takes three arguments: chatData (a dataset), model (a GPT-2 model), and optim (an optimizer). It specifies the number of training epochs (12) and iterates through the dataset for the specified number of epochs. It performs training by calculating the loss, backpropagating, and updating the model's weights. The trained model's state is saved to a file, and it also calls the infer function with an input string.
- The infer function takes an input string, modifies it to match the format expected by the model, tokenizes it using the tokenizer, generates a response using the GPT-2 model, and decodes the model's output to provide the response.
- It checks the availability of a GPU and selects the device accordingly ("cuda" for GPU, "mps" if using PyTorch's Mixed Precision Training, and "cpu" if no GPU is available).
- It initializes the GPT-2 tokenizer with the "gpt2" model's vocabulary and adds special tokens and a custom "<bot>:" token.
- It loads the GPT-2 model ("gpt2") and resizes its token embeddings to match the tokenizer's vocabulary.
- The model is moved to the selected device.
- The script initializes a custom dataset named chatData and creates a DataLoader to handle batching and data loading.
- The model is set to training mode.
- It initializes an Adam optimizer with a learning rate of 1e-3 for the model's parameters.
- It starts the training process by calling the train function with the dataset, model, and optimizer as arguments.
- After training, it enters an inference loop where it continuously takes user input, sends it to the infer function, and prints the generated responses.

Integration:

We are go intrageted chatbot to website. In which using the flask ,python web framework.

Flask is a lightweight Python web framework that provides a simple and elegant way to build web applications. It is designed to be easy to learn and use, yet powerful enough to handle complex applications. Flask is also highly extensible, with a large community of contributors providing extensions for a variety of tasks.

Advantages of using Flask

- Lightweight: Flask is a microframework, which means that it does not require any specific tools or libraries. This makes it easy to get started and deploy your application.
- Easy to learn: Flask has a simple and intuitive API that is easy to learn and use, even for beginners.
- Flexible: Flask is highly extensible, with a large community of contributors providing extensions for a variety of tasks. This makes it easy to customize your application to meet your specific needs.
- Popular: Flask is one of the most popular Python web frameworks, with a large and active community. This means that there are many resources available to help you learn and use Flask.

Use cases for Flask

Flask can be used to build a wide variety of web applications, from simple hobby projects to complex enterprise applications. Some common use cases for Flask include:

- APIs: Flask is a popular choice for building APIs, as it is easy to use and flexible.
- Web applications: Flask can be used to build full-fledged web applications, including user interfaces, databases, and authentication.

- Microservices: Flask can be used to build microservices, which are small, independent services that can be scaled and deployed independently.
- Single-page applications: Flask can be used to build single-page applications (SPAs), which are web applications that load a single HTML page and then update the page dynamically without reloading.

Route in flask:

To route a HTML in Flask, you can use the @app.route() decorator and the render_template() function. The @app.route() decorator tells Flask which URL should be associated with the function that renders the HTML file. The render_template() function renders the HTML file and returns the rendered HTML to the browser.

```
from flask import Flask, render_template,request,jsonify

app = Flask(__name__)

@app.route("/")
def index():
    return render_template("index.html")

@app.route("/chat")
def chatRequest():
    data = request.get_json()
    user_message = data['message'].lower()
    response=infer(user_message)
    return jsonify({'message': response})


if __name__ == "__main__":
    app.run(debug=True)
```

In this simple implement the flask server which serve a html page with the javascript file .this can use to fetch data from the chatbot

In this Script:

1. It imports the necessary libraries, including Flask, **render_template**, **request**, and **jsonify**.
2. A Flask web application is created and initialized.
3. The first route, **@app.route("/")**, maps to the root URL ("/"). When a user accesses this URL, it renders an HTML template named "index.html."

4. The second route, **@app.route("/chat")**, maps to the URL "/chat." This route expects a JSON POST request with a "message" field. It retrieves the user's message from the JSON data, converts it to lowercase, and then calls a function named **infer** to generate a response.
5. The generated response is returned in JSON format using the **jsonify** function.
6. The code checks if the script is executed directly (not imported as a module) and runs the Flask application in debug mode if it is.

Overall, this code sets up a basic Flask web application with two routes: one for serving an HTML template at the root URL and another for processing chat messages and generating responses via the "infer" function.

Summary:

1. **Introduction to Chatbots:**

   Chatbot are computer programs or software applications designed to simulate human conversation through text or voice interactions. They use natural language processing (NLP) and machine learning techniques to understand and respond to user queries or inputs. You also highlighted the versatility of chatbots in various applications, such as customer support, information retrieval, and automated tasks.

2. Data Source:

   You mentioned the dataset used in the project, which consists of multiple short conversations between participants, focusing on casual and friendly exchanges. It's essential to have high-quality data to train and test chatbot models effectively.

3. Design Thinking Process:

   You outlined the design thinking process applied to chatbot development for customer support, emphasizing a user-centric approach. The process involves the following stages:

   1. Empathize: Understanding customer needs and pain points through feedback and insights.
   2. Define: Clearly defining the problem and setting specific goals.

3. Ideate: Brainstorming potential solutions and features for the chatbot.
4. Prototype: Creating a basic prototype and gathering user feedback.
5. Test: Conducting user testing to refine the chatbot's design.
6. Implement: Developing the chatbot using suitable technologies.
7. Iterate: Continuously collecting user feedback and making improvements.
8. Launch: Deploying the chatbot for real-world customer support interactions.
9. Measure: Tracking key performance metrics and adjusting the chatbot based on data.
10. Refine: Regularly updating and refining the chatbot based on user feedback and changing needs.

4. Packages and Libraries:

We provided a list of Python packages and libraries commonly used in chatbot development, categorized by their purposes. These include libraries for NLP, machine learning, web frameworks, API integration, and database access.

5. Data Preprocessing:

We have explained the data preprocessing steps, including cleaning the text data, such as converting text to lowercase, removing numbers, and adding special tokens like "<sos>" (start of sentence) and "<eos>" (end of sentence) to define input and target sequences.

6. Data Visualization:

We demonstrated the importance of data visualization before and after preprocessing, using tools like Matplotlib and Seaborn. Data visualization helps understand the distribution of token counts, which is crucial for setting appropriate sequence lengths and padding.

7. Development of Chatbot:
    1. Tokenizer: we described the role of tokenizers in chatbot development, focusing on the GPT-2 Tokenizer. Tokenization breaks text into smaller units (tokens) and prepares it for use with the model.

2. Model: we explained the GPT2-LM Head Model, highlighting its use for natural language generation and various tasks like text generation, translation, and summarization. Pretrained GPT-2 models are valuable for their versatility.
3. Training Model: we detailed the steps for training the chatbot model. This includes defining a dataset class, creating data loaders for batch processing, setting up the training loop, and using PyTorch for the training process.
4. Integration: we mentioned how to integrate the chatbot into a web application using Flask, a popular Python web framework. Flask allows you to serve the chatbot on a website and interact with users.

8. Conclusion:

Finally, we have summarized the content covered, emphasizing the comprehensive nature of the explanation and its value for those interested in developing chatbots, particularly for customer support or other real-world applications.