

Date:24.09.25

TASK:9

To Build an Intelligent **Chatbot system** with Python and Dialog-flow using Interactive Text Mining Framework for Exploration of Semantic Flows in Large Corpus of Text.

To Build an Intelligent Chatbot system with Python and Dialog-flow using Interactive Text Mining Framework for Exploration of Semantic Flows in Large Corpus of Text. **CO4 S3**

- To integrate with Google Cloud Speech-to-Text and third-party services such as Google Assistant, Amazon Alexa, and Facebook Messenger.
- Configure Dialogflow to manage your data across GCP services and let you optionally integrate Google Assistant.

Tools- Python, Dialog-flow Framework

TO BUILD AN INTELLIGENT CHATBOT SYSTEM WITH PYTHON AND DIALOG-FLOW USING INTERACTIVE TEXT MINING FRAMEWORK FOR EXPLORATION OF SEMANTIC FLOWS IN LARGE CORPUS OF TEXT

AIM:

To build an intelligent chatbox system with Python and dialog-flow using interactive text mining framework for exploration of semantic flow in large corpus of Text

ALGORITHM:

Steps to create an intelligent chatbot using OpenAI APIs:

1. Sign up for OpenAI API access at <https://beta.openai.com/signup/>. Once you sign up, you will receive your API key.
2. Choose the type of chatbot you want to create. For example, you can create an FAQ chatbot, a customer support chatbot, or a conversational chatbot.
3. Use OpenAI's GPT-3 language model to generate responses to user input. You can use the API to train the language model on your chatbot's intended use case/s.
4. Use Natural Language Processing (NLP) techniques to understand user input and provide relevant responses. You can use OpenAI's API to extract entities (such as dates and names) from user input.
5. Use Machine Learning to continually improve the chatbot's ability to understand and respond to user input.
6. Integrate the chatbot with your preferred messaging platform or channel (e.g., web chat, social media, etc.) using API connectors.
7. Test your chatbot frequently, and use user feedback to improve its performance and provide the best possible experience for your users.

A. SIMPLE CHATGPT USING GEMINI

CODE:

```
from langchain_google_genai import ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(

    model="gemini-2.5-flash", # Or "gemini-1.5-pro-latest" if available

    google_api_key="AIzaSyCp7RYEV2grZ3GkemVEGyqFQW_LXF9fUk4", # Keep this
    secure!

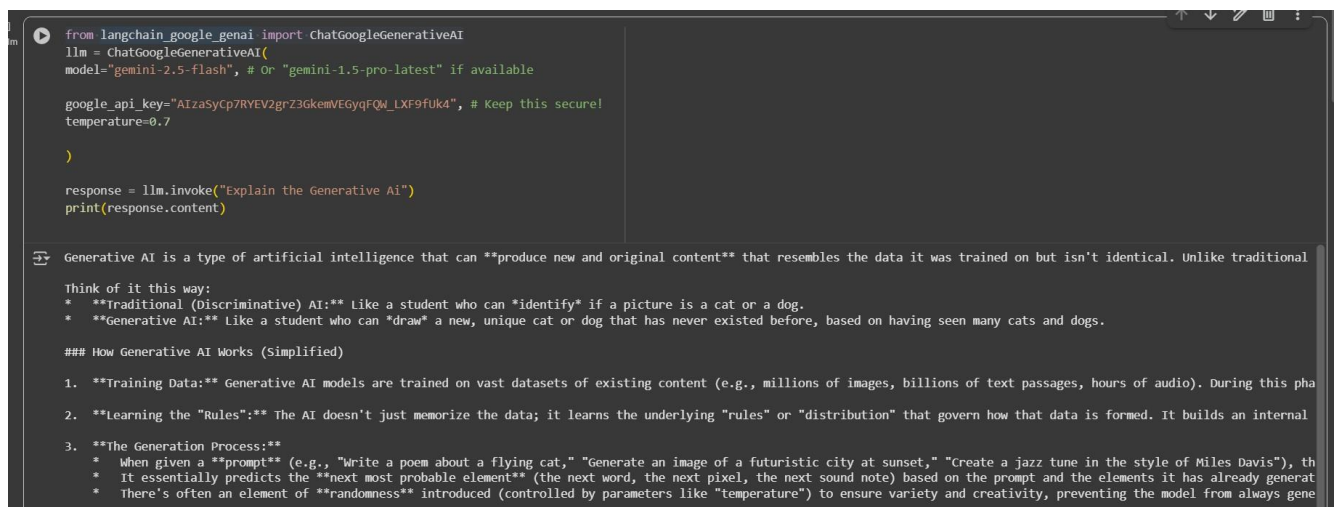
    temperature=0.7

)

response = llm.invoke("Explain quantum computing simply,breif in points")

print(response.content)
```

OUTPUT:



```
from langchain_google_genai import ChatGoogleGenerativeAI
llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash", # Or "gemini-1.5-pro-latest" if available

    google_api_key="AIzaSyCp7RYEV2grZ3GkemVEGyqFQW_LXF9fUk4", # Keep this secure!
    temperature=0.7

)

response = llm.invoke("Explain the Generative Ai")
print(response.content)
```

Generative AI is a type of artificial intelligence that can **produce new and original content** that resembles the data it was trained on but isn't identical. Unlike traditional

Think of it this way:

- * **Traditional (Discriminative) AI:** Like a student who can **identify** if a picture is a cat or a dog.
- * **Generative AI:** Like a student who can **draw** a new, unique cat or dog that has never existed before, based on having seen many cats and dogs.

How Generative AI Works (Simplified)

1. **Training Data:** Generative AI models are trained on vast datasets of existing content (e.g., millions of images, billions of text passages, hours of audio). During this phase, the model learns the patterns and structures within the data.
2. **Learning the "Rules":** The AI doesn't just memorize the data; it learns the underlying "rules" or "distribution" that govern how that data is formed. It builds an internal representation of the data's structure.
3. **The Generation Process:**
 - * When given a **prompt** (e.g., "Write a poem about a flying cat," "Generate an image of a futuristic city at sunset," "Create a jazz tune in the style of Miles Davis"), the model uses its learned rules to generate new content.
 - * It essentially predicts the **next most probable element** (the next word, the next pixel, the next sound note) based on the prompt and the elements it has already generated.
 - * There's often an element of **randomness** introduced (controlled by parameters like "temperature") to ensure variety and creativity, preventing the model from always generating the same output.

B. CHATGPT ASSISTANT USING GEMINI

CODE:

```
# gemini_chatbot.py

from flask import Flask, request, jsonify

import os

from google import genai

from google.genai import types

app = Flask(__name__)

GEMINI_API_KEY="AIzaSyCp7RYEV2grZ3GkemVEGyqFQW_LXF9fUk4"

# --- Configure API Key ---

# Using the hardcoded API key from above

api_key = GEMINI_API_KEY

# Initialize the client

client = genai.Client(api_key=api_key)

# Choose the Gemini model you want to use

MODEL = "gemini-2.5-flash" # or "gemini-2.5-pro" etc, depending on access

def generate_reply_from_gemini(prompt: str) -> str:

    """

    Send the user prompt to Gemini and return the response text.

    """
```

```

response = client.models.generate_content(

    model=MODEL,

    contents=prompt,

    # You can optionally provide a config, e.g. thinking_budget etc.

                                                                    #
    config=types.GenerateContentConfig(thinking_config=types.ThinkingConfig(thinking_budget
=0))

)

return response.text


@app.route("/")

def home():

    return app.send_static_file('index.html')


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

def chat():

    data = request.get_json()

    user_message = data.get("message", "")

    if not user_message:

        return jsonify({"error": "No message provided"}), 400

    try:

        reply = generate_reply_from_gemini(user_message)

        return jsonify({"reply": reply})

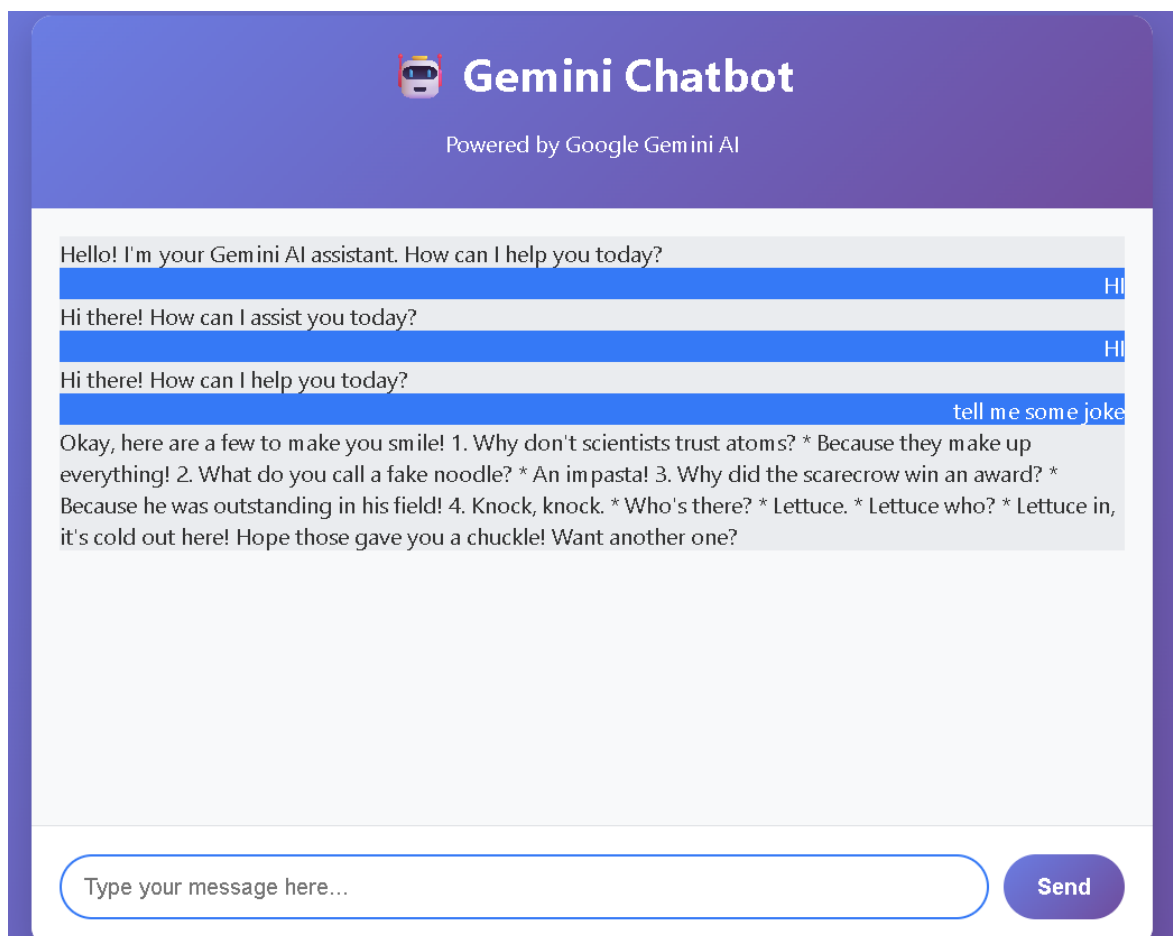
    except Exception as e:

        return jsonify({"error": str(e)}), 500

```

```
if __name__ == "__main__":  
  
    # Run in debug for development  
  
    app.run(host="0.0.0.0", port=5000, debug=True)
```

OUTPUT:



C. CHATBOT CHAT ASSISTANT WEBSITE

CODE:

```
import openai

import gradio

openai.api_key = "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"

messages = [{"role": "system", "content": "You are a financial experts that specializes in real estate investment and negotiation"}]

def CustomChatGPT(user_input):

    messages.append({"role": "user", "content": user_input})

    response = openai.ChatCompletion.create(

        model = "gpt-3.5-turbo",

        messages = messages

    )

    ChatGPT_reply = response["choices"][0]["message"]["content"]

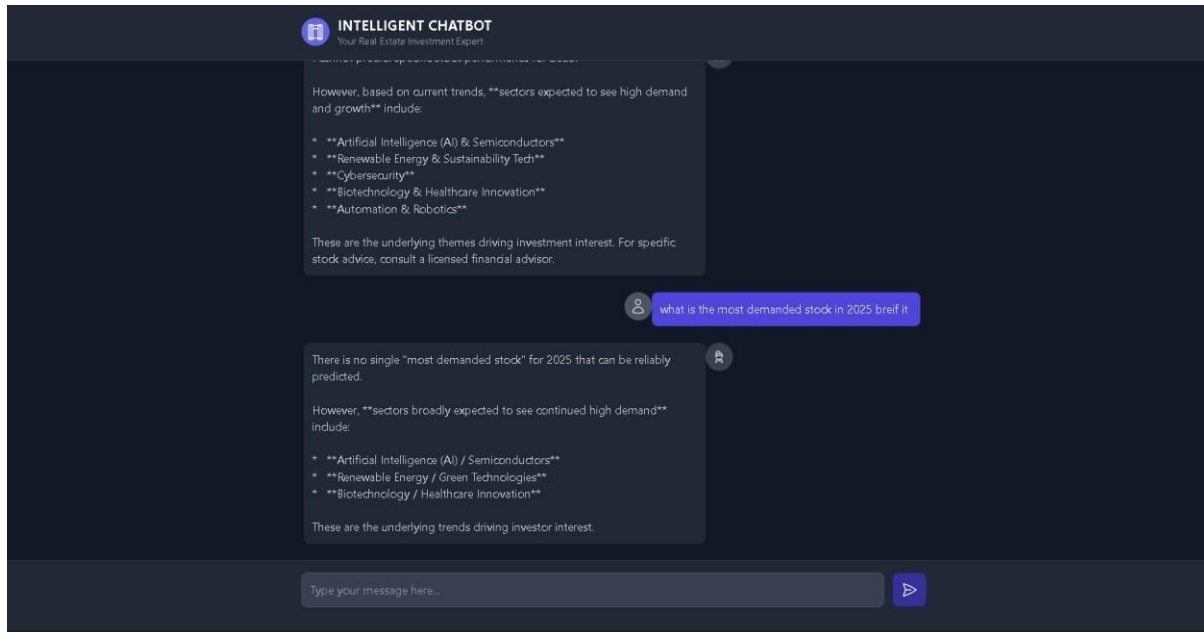
    messages.append({"role": "assistant", "content": ChatGPT_reply})

    return ChatGPT_reply

demo = gradio.Interface(fn=CustomChatGPT, inputs = "text", outputs = "text", title = "INTELLIGENT CHATBOT")

demo.launch(share=True)
```

OUTPUT:



RESULT:

Thus, to build an intelligent chatbox system with Python and dialogue flow was successfully completed and output was verified.