**RAG Integrated gen ai based chatbot project**

Here's a detailed step-by-step guide on what you likely did as a data scientist in the "Gen AI-based chatbot" project:

**1. Understanding the Project Requirements**

* **Action**: Collaborated with stakeholders to define the scope and purpose of the chatbot (e.g., customer service, education, or e-commerce support).
* **Deliverable**: Documented key use cases, personas, and expected outcomes for the chatbot.
* **Tools**: Requirement gathering templates, Miro/whiteboarding tools.

**2. Data Collection**

* **Action**: Sourced relevant datasets for training the generative model. This could include open-domain conversational data, domain-specific FAQs, or logs of previous customer interactions.
* **Deliverable**: A raw dataset stored in a structured format (e.g., JSON, CSV).
* **Tools**: Web scraping tools, APIs, SQL for databases.

**3. Data Preprocessing**

* **Action**: Cleaned and processed the data by removing noise (e.g., spelling errors, duplicates) and ensuring consistent formatting.
* **Deliverable**: A clean and tokenized dataset ready for model training.
* **Tools**: Python (pandas, NumPy, NLTK, or spaCy).

Steps involved:

* Removed irrelevant data (e.g., ads, unrelated queries).
* Tokenized text into words or subwords.
* Handled missing data or imbalanced classes.
* Annotated intent or entity tags, if applicable.

**4. Model Selection**

* **Action**: Selected a pre-trained generative AI model (e.g., GPT-3, T5, or open-source models like GPT-2, LLaMA) to fine-tune for your chatbot.
* **Deliverable**: A baseline model suitable for your task.
* **Tools**: Hugging Face Transformers library, OpenAI API, TensorFlow, PyTorch.

**Factors considered:**

* Language understanding capability.
* Licensing and cost constraints.
* Deployment feasibility.

**5. Fine-Tuning the Model**

* **Action**: Fine-tuned the pre-trained model using your clean dataset to make it domain-specific.
* **Deliverable**: A custom-trained chatbot model capable of answering specific queries.
* **Tools**:
  + Hugging Face Trainer or OpenAI Fine-tuning API.
  + GPUs for faster training.

**Steps involved:**

* Divided the dataset into training, validation, and testing subsets.
* Configured hyperparameters (e.g., learning rate, batch size).
* Performed training and evaluated metrics like perplexity.

**6. Designing and Testing Conversational Flows**

* **Action**: Developed a conversational framework (e.g., handling greetings, FAQs, fallbacks, and multi-turn interactions).
* **Deliverable**: A flow diagram or a script for the chatbot's expected behavior.
* **Tools**: Rasa, Botpress, or custom rule-based logic.

**Steps involved:**

* Implemented prompts for common queries.
* Tested user contexts and ensured the bot maintained continuity in multi-turn conversations.
* Incorporated fallback mechanisms for unknown queries.

**7. Evaluation and Refinement**

* **Action**: Evaluated the chatbot's responses against metrics (e.g., BLEU, perplexity) and real user feedback.
* **Deliverable**: Iteratively improved the chatbot’s performance based on evaluation results.
* **Tools**:
  + Metrics libraries like sacreBLEU, human validation interfaces, or A/B testing frameworks.

**Steps involved:**

* Conducted manual testing by simulating real conversations.
* Measured response relevance and user satisfaction scores.

**8. Deployment**

* **Action**: Deployed the chatbot on a platform (e.g., website, mobile app, or messaging services like WhatsApp or Slack).
* **Deliverable**: A live chatbot integrated with end-user systems.
* **Tools**: Docker, AWS/GCP/Azure, APIs for integration.

**Steps involved:**

* Packaged the trained model.
* Deployed a REST API or used a chatbot framework.
* Ensured latency and scalability through load testing.

**9. Post-Deployment Monitoring**

* **Action**: Monitored chatbot interactions to identify bugs, improve responses, and track performance metrics like average response time and engagement rate.
* **Deliverable**: Reports on chatbot performance and continuous updates.
* **Tools**: Kibana, Grafana, or custom analytics pipelines.

**Steps involved:**

* Logged interactions for analysis.
* Used feedback to retrain and update the model periodically.

**10. Continuous Improvement**

* **Action**: Periodically updated the chatbot by:
  + Retraining on new data.
  + Enhancing contextual understanding.
  + Adding new use cases or intents.
* **Deliverable**: Regular updates to keep the chatbot relevant and efficient.

**A simple "gen ai based chatbot Project" code end to end using gpt3.5 turbo**

**Step 1: Install Required Libraries**

bash

Copy code

pip install openai flask

**Step 2: Code for the Chatbot**

python

Copy code

import openai

from flask import Flask, request, jsonify

# Initialize Flask app

app = Flask(\_\_name\_\_)

# Set your OpenAI API key

openai.api\_key = "your\_openai\_api\_key\_here"

# Route for chatbot interaction

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

def chat():

try:

# Get user input from the request

user\_input = request.json.get('message', '')

if not user\_input:

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

# Call OpenAI API

response = openai.ChatCompletion.create(

model="gpt-3.5-turbo",

messages=[

{"role": "system", "content": "You are a helpful and friendly chatbot."},

{"role": "user", "content": user\_input}

],

max\_tokens=150,

temperature=0.7

)

# Extract the response text

chatbot\_reply = response['choices'][0]['message']['content']

# Return the chatbot's reply

return jsonify({"reply": chatbot\_reply})

except Exception as e:

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

# Run the app

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**Step 3: Test the Chatbot**

1. Save the code to a file, e.g., chatbot.py.
2. Run the script:

bash

Copy code

python chatbot.py

1. The chatbot will start a server locally, typically at http://127.0.0.1:5000.

**Step 4: Make Requests to the Chatbot**

You can interact with the chatbot using tools like **Postman** or via a Python client. Here's an example of a request using Python:

python

Copy code

import requests

url = "http://127.0.0.1:5000/chat"

message = {"message": "Hello, chatbot!"}

response = requests.post(url, json=message)

if response.status\_code == 200:

print("Chatbot Reply:", response.json()['reply'])

else:

print("Error:", response.json())

**End-to-end example of a RAG (Retrieval-Augmented Generation)** integrated generative AI chatbot. This approach uses a knowledge base (retrieval component) to provide contextually accurate responses and integrates GPT-3.5 for natural language generation.

**Overview**

1. **Knowledge Base**: A collection of documents (e.g., FAQs, articles) to retrieve context.
2. **Retriever**: Uses embeddings to fetch relevant documents from the knowledge base.
3. **Generator**: GPT-3.5 processes the user query along with retrieved documents to generate responses.

**Step 1: Install Required Libraries**

Install the necessary Python libraries:

bash

Copy code

pip install openai langchain faiss-cpu flask

**Step 2: Prepare the Knowledge Base**

Save your documents as plain text files in a folder called knowledge\_base.

Example directory structure:

Copy code

knowledge\_base/

doc1.txt

doc2.txt

doc3.txt

**Step 3: Code for the RAG-Based Chatbot**

python

Copy code

import os

from flask import Flask, request, jsonify

import openai

from langchain.vectorstores import FAISS

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.document\_loaders import DirectoryLoader

from langchain.text\_splitter import CharacterTextSplitter

# Initialize Flask app

app = Flask(\_\_name\_\_)

# Set OpenAI API key

openai.api\_key = "your\_openai\_api\_key\_here"

# Step 1: Load and Index the Knowledge Base

def load\_knowledge\_base():

# Load documents from the 'knowledge\_base' folder

loader = DirectoryLoader('knowledge\_base', glob="\*.txt")

documents = loader.load()

# Split large texts into smaller chunks

text\_splitter = CharacterTextSplitter(chunk\_size=500, chunk\_overlap=50)

docs = text\_splitter.split\_documents(documents)

# Generate embeddings and store in FAISS vector database

embeddings = OpenAIEmbeddings()

vector\_store = FAISS.from\_documents(docs, embeddings)

return vector\_store

# Load the vector store (knowledge base)

vector\_store = load\_knowledge\_base()

# Step 2: RAG Chatbot Endpoint

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

def chat():

try:

# Get user query from request

user\_query = request.json.get('message', '')

if not user\_query:

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

# Retrieve relevant documents from the knowledge base

relevant\_docs = vector\_store.similarity\_search(user\_query, k=3)

retrieved\_context = "\n\n".join([doc.page\_content for doc in relevant\_docs])

# Combine retrieved context with user query

system\_prompt = (

"You are a helpful assistant. Use the provided context to answer the user's question.\n\n"

f"Context:\n{retrieved\_context}\n\n"

"If the context is not sufficient, provide a helpful and general answer."

)

# Call OpenAI GPT-3.5 API

response = openai.ChatCompletion.create(

model="gpt-3.5-turbo",

messages=[

{"role": "system", "content": system\_prompt},

{"role": "user", "content": user\_query}

],

max\_tokens=300,

temperature=0.7

)

# Extract the chatbot's reply

chatbot\_reply = response['choices'][0]['message']['content']

# Return the chatbot's reply

return jsonify({"reply": chatbot\_reply})

except Exception as e:

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

# Run the app

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**Step 4: Run the Chatbot**

1. Save the script as rag\_chatbot.py.
2. Run the script:

bash

Copy code

python rag\_chatbot.py

1. Access the chatbot at http://127.0.0.1:5000/chat.

**Step 5: Test the Chatbot**

You can interact with the chatbot via a Python script or a tool like Postman.

**Example Test Script**

python

Copy code

import requests

url = "http://127.0.0.1:5000/chat"

user\_message = {"message": "What is the purpose of the documents in the knowledge base?"}

response = requests.post(url, json=user\_message)

if response.status\_code == 200:

print("Chatbot Reply:", response.json()['reply'])

else:

print("Error:", response.json())