## Code:-

```
import os
from dotenv import load_dotenv
from langgraph.graph import StateGraph, START, END
from typing import TypedDict, Annotated
from langchain_core.messages import HumanMessage, AIMessage
from langchain_groq import ChatGroq
from langgraph.graph.message import add_messages
from langgraph.checkpoint.memory import MemorySaver
from sentence_transformers import SentenceTransformer
from transformers import pipeline
import requests
import faiss
import numpy as np
import spacy

# Load environment variables
load_dotenv()
groq_key = os.getenv('GROQ_API_KEY')

# NLP pipelines and memory saver initialization
nlu_pipeline = pipeline("zero-shot-classification",
model="facebook/bart-large-mnli")
nlp = spacy.load("en_core_web_sm")
sentiment_pipeline = pipeline("sentiment-analysis")
memory = MemorySaver()

# Document Embeddings Model
embedding_model = SentenceTransformer('sentence-transformers/all-mpnet-base-v2')

# Sample Documents
documents = [
   "What is the capital of France?",
   "Who is the president of the United States?",
   "Explain quantum computing.",
   "Describe the theory of relativity.",
   "How does machine learning work?"
]

# Create embeddings for documents
embeddings = embedding_model.encode(documents)

# Create a FAISS index
d = embeddings.shape[1]  # Dimension of the embeddings
index = faiss.IndexFlatL2(d)
```

```
index.add(embeddings)

# Contextual memory class to store user-specific data
class ContextualMemory(MemorySaver):
    def __init__(self):
        super().__init__()
        self.user_context = {}

    def store_user_context(self, user_id, context):
        self.user_context[user_id] = context

    def get_user_context(self, user_id):
        return self.user_context.get(user_id, {})

contextual_memory = ContextualMemory()

# Feedback collector class
class FeedbackCollector:
    def __init__(self):
        self.feedback = []

    def collect_feedback(self, feedback):
        self.feedback.append(feedback)

feedback_collector = FeedbackCollector()

# External API call function (Optional for other API interactions)
def fetch_external_data(query):
    headers = {
        'Authorization': f'Bearer {os.getenv("EXTERNAL_API_KEY")}'
    }
    response = requests.post(
        "https://api.symbl.ai/v1/process/text",
        json={"messages": [{"text": query}]},
        headers=headers
    )
    if response.status_code == 200:
        return response.json()
    else:
        return {"error": "Failed to fetch data"}

# Define State class
class State(TypedDict):
    messages: Annotated[list, add_messages]
```

```
# Initialize the graph builder
graph_builder = StateGraph(State)

# Initialize LLM with Groq API
llm = ChatGroq(model_name="mixtral-8x7b-32768", groq_api_key=groq_key,
temperature=0.2)

# Document retrieval using FAISS
def retrieve_documents(query, top_k=3):
   query_embedding = embedding_model.encode([query])
   distances, indices = index.search(query_embedding, top_k)
   return [documents[i] for i in indices[0]]

# Augment LLM Input with Retrieved Documents
def generate_augmented_response(state, user_query):
   # Retrieve relevant documents
   retrieved_docs = retrieve_documents(user_query)

   # Format retrieved documents as input for LLM
   context = "\n".join(retrieved_docs)

   # Combine user query with retrieved context
   input_for_llm = f"Context: {context}\n\nQuestion: {user_query}"

   # Generate response using LLM
   response = llm.invoke([HumanMessage(content=input_for_llm)])
   state["messages"].append(AIMessage(content=response.content))
   return state

# Custom Nodes
def recognize_intent_node(state: State):
   user_input = state["messages"][-1].content
   candidate_labels = ["greeting", "request", "question", "feedback", "fetch"]
   intent = nlu_pipeline(user_input, candidate_labels)
   recognized_intent = intent["labels"][0] if intent["scores"][0] > 0.5 else
"unknown"
   state["messages"].append(AIMessage(content=f"Recognized Intent:
{recognized_intent}"))
   return state

def extract_entities_node(state: State):
   user_input = state["messages"][-1].content
   doc = nlp(user_input)
   entities = [(ent.text, ent.label_) for ent in doc.ents]
   entity_text = ", ".join([f"{ent[0]} ({ent[1]})" for ent in entities])
```

```
    state["messages"].append(AIMessage(content=f"Extracted Entities:
{entity_text}"))
    return state

def analyze_sentiment_node(state: State):
    user_input = state["messages"][-1].content
    sentiment_result = sentiment_pipeline(user_input)[0]
    sentiment_analysis = {"label": sentiment_result["label"], "score":
sentiment_result["score"]}
    state["messages"].append(AIMessage(content=f"Sentiment:
{sentiment_analysis['label']} with score {sentiment_analysis['score']}"))
    return state

# RAG-based generation node
def retrieve_and_generate_node(state: State):
    user_input = state["messages"][-1].content
    state = generate_augmented_response(state, user_input)
    return state

def chatbot_logic_node(state: State):
    user_input = state["messages"][-1].content
    # Handle feedback collection
    if "Was this helpful?" in user_input:
        feedback_collector.collect_feedback(user_input)
        state["messages"].append(AIMessage(content="Thank you for your feedback!"))
    else:
        # Ask for feedback after response
        state["messages"].append(AIMessage(content="I hope that was helpful. Was
this response useful?"))
    return state

# Build graph by adding all custom nodes and defining edges
graph_builder.add_node("intent_recognition", recognize_intent_node)
graph_builder.add_node("entity_extraction", extract_entities_node)
graph_builder.add_node("sentiment_analysis", analyze_sentiment_node)
graph_builder.add_node("retrieve_and_generate", retrieve_and_generate_node)
graph_builder.add_node("chatbot_logic", chatbot_logic_node)

# Define the flow of the nodes
graph_builder.add_edge(START, "intent_recognition")
graph_builder.add_edge("intent_recognition", "entity_extraction")
graph_builder.add_edge("entity_extraction", "sentiment_analysis")
graph_builder.add_edge("sentiment_analysis", "retrieve_and_generate")
graph_builder.add_edge("retrieve_and_generate", "chatbot_logic")
graph_builder.add_edge("chatbot_logic", END)
```

```
# Compile the graph
graph = graph_builder.compile(checkpointer=contextual_memory)

# Run the chatbot in a loop
while True:
   user_input = input("You: ")
   if user_input.lower() == "quit":
       break

   config = {"configurable": {"thread_id": "1"}}
   events = graph.stream({"messages": [HumanMessage(content=user_input)]}, config,
stream_mode="values")
   for event in events:
       print(event["messages"][-1].content)
```

This code exemplifies a Retrieve and Generate (RAG) approach for building a chatbot that combines document retrieval with natural language understanding (NLU), entity extraction, sentiment analysis, and generative AI response using a structured flow defined in LangGraph.

1. **Environment and Dependencies Initialization**
   - Load Variables: Environment variables are loaded using `dotenv`, crucial for securely storing API keys.
   - NLP and Memory Models: Initialize the NLU pipeline (`zero-shot-classification`), entity extraction (`spaCy`), sentiment analysis (`sentiment-analysis`), and memory saver (`MemorySaver`) for contextual storage.
   - Embeddings Model: A document embedding model (`SentenceTransformer`) converts documents into embeddings used for similarity-based retrieval.

2. **Document Indexing and Retrieval with FAISS**
   - Embeddings Creation: Given a set of sample documents, embeddings are created to represent the documents in vector space.
   - FAISS Indexing: Embeddings are indexed using FAISS for fast similarity-based retrieval. This index enables retrieval of the top `k` documents related to a given query.

3. **Define Chatbot Workflow Using LangGraph Nodes**

Nodes:

1. ***Intent Recognition Node***
   - Input: Last user message (`state["messages"][-1].content`).
   - Process: Applies the NLU pipeline to classify intent into predefined labels (e.g., greeting, request).
   - Output: Appends recognized intent to `state["messages"]` as an AI message.

2. ***Entity Extraction Node***
   - Input: User input from the last message.
   - Process: Uses `spaCy` for Named Entity Recognition (NER) to identify entities.
   - Output: Adds extracted entities to `state["messages"]`.

3. ***Sentiment Analysis Node***
   - Input: User input from the last message.
   - Process: Runs sentiment analysis on the user message, categorizing the sentiment and appending the result.
   - Output: Adds the sentiment label and score to `state["messages"]`.

4. ***Retrieve and Generate Node (RAG)***
   - Input: User query and `state` containing messages.
   - Process:
     1. Retrieve Documents: Uses FAISS to retrieve the most relevant documents based on the query.
     2. Generate Augmented Response: Combines retrieved documents and user query, then passes it to the language model (`llm.invoke()`).
   - Output: Generated response is appended to `state["messages"]`.

5. ***Chatbot Logic Node***
   - Input: Generated response or feedback prompt.
   - Process: Checks if user feedback is requested, logs feedback if present, otherwise asks for feedback.
   - Output: Appends an AI message for either feedback acknowledgment or a feedback prompt.

**Graph Construction and Flow Definition**
   - Graph Setup: Nodes are sequentially added and linked through edges to create a structured flow.
   - Edge Definition: Each node is connected to direct the flow from intent recognition through to chatbot logic, where interactions end with user feedback.
   - Compilation: The graph is compiled with `contextual_memory`, enabling retrieval of user-specific data across sessions.

4. **Execution Loop**
   - User Interaction: Continuously accepts user input until "quit" is entered.
   - State Update and Output: Processes each input through the graph, and responses are streamed back to the user.