AI-Powered Chat-bot

Udit Srivastava

Data Ingestion

Objective

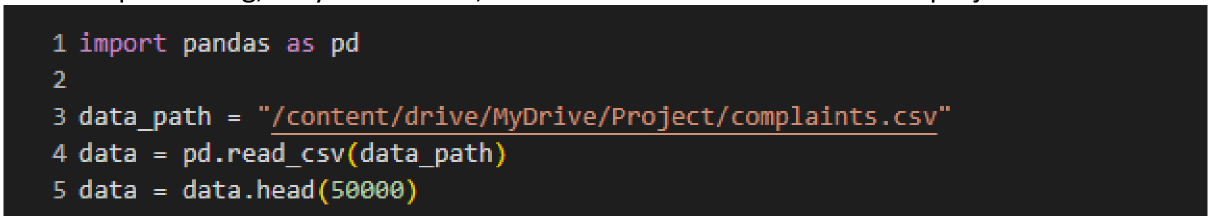
To utilize a domain-relevant dataset, preprocess it effectively, and structure it for subsequent tasks like retrieval, summarization, and sentiment analysis.

Dataset Overview

- **Source:** Bank Customer Complaint Analysis
- **Details:**
    - File: complaints.csv
    - Dataset contains information about customer complaints related to bank services.
    - Fields: Includes narratives describing complaints, along with other metadata.

Steps for Data Ingestion

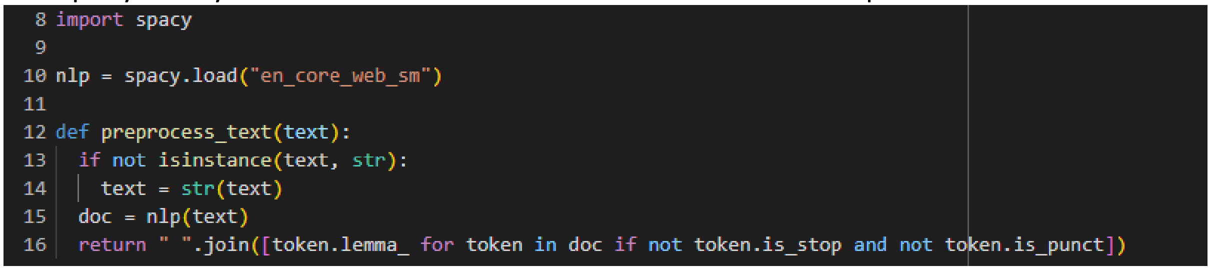
1. **Loading the Dataset:** The dataset was loaded using pandas for manipulation. To ensure efficient processing, only the first 50,000 records were selected for the project.



```python
import pandas as pd

data_path = "/content/drive/MyDrive/Project/complaints.csv"
data = pd.read_csv(data_path)
data = data.head(50000)
```

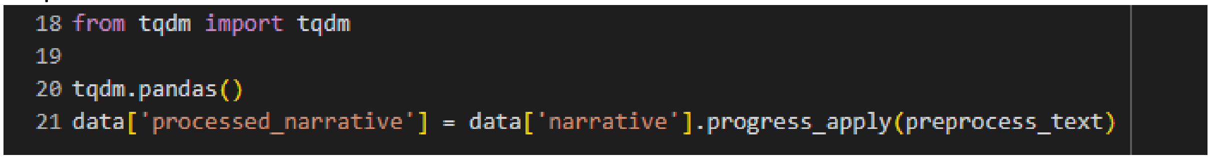
2. **Preprocessing the Text:**
   The preprocessing pipeline aimed to clean and normalize the complaint narratives.
   Key steps included:
   - Tokenization: Breaking down text into individual tokens (words).
   - Stop-word Removal: Removing common but irrelevant words like "the" and "is."
   - Punctuation Removal: Excluding punctuation marks to reduce noise.
   - Lemmatization: Converting words to their root forms (e.g., "complaining" → "complain").

   The spaCy library was used for these tasks due to its robust NLP capabilities.



```python
import spacy

nlp = spacy.load("en_core_web_sm")

def preprocess_text(text):
    if not isinstance(text, str):
        text = str(text)
    doc = nlp(text)
    return " ".join([token.lemma_ for token in doc if not token.is_stop and not token.is_punct])
```

3. **Applying Preprocessing to the Dataset**
   The narrative column, which contains the complaint details, was processed using the preprocess_text function. The tqdm library was used to monitor progress during this step.



```python
from tqdm import tqdm

tqdm.pandas()
data['processed_narrative'] = data['narrative'].progress_apply(preprocess_text)
```

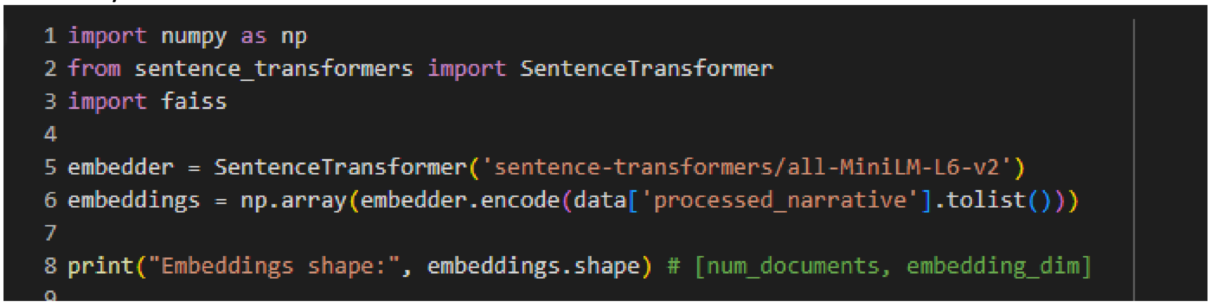
Search and Retrieval

Objective

Integrate a search backend using FAISS to retrieve relevant documents efficiently based on user queries.

Search Backend Implementation

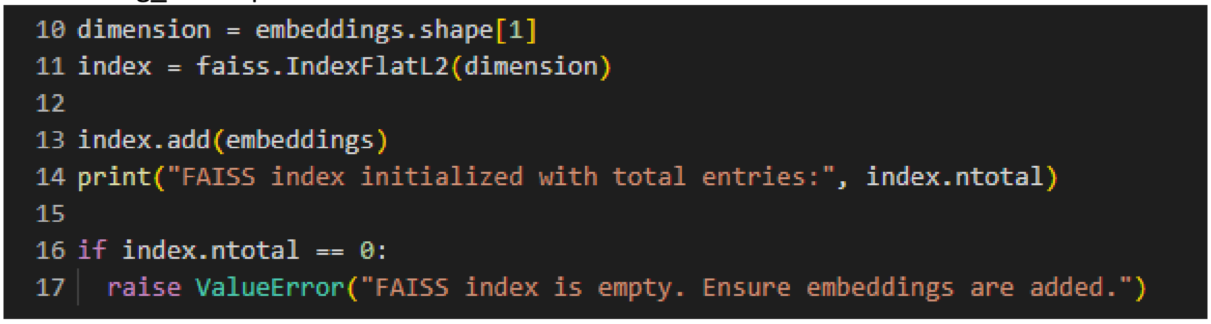
1. **Sentence Embedding Generation**
   - A pre-trained SentenceTransformer model (all-MiniLM-L6-v2) was utilized to generate dense embeddings for the preprocessed narratives.
   - These embeddings represent the semantic meaning of the text and are essential for similarity-based retrieval.



```python
import numpy as np
from sentence_transformers import SentenceTransformer
import faiss

embedder = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')
embeddings = np.array(embedder.encode(data['processed_narrative'].tolist()))

print("Embeddings shape:", embeddings.shape) # [num_documents, embedding_dim]
```

Outcome: The embeddings shape confirms the dimensionality of the generated vectors, where num_documents represents the number of documents in the dataset and embedding_dim represents the vector dimension.

2. **Index Creation with FAISS:**
   The embeddings shape confirms the dimensionality of the generated vectors, where num_documents represents the number of documents in the dataset and embedding_dim represents the vector dimension.

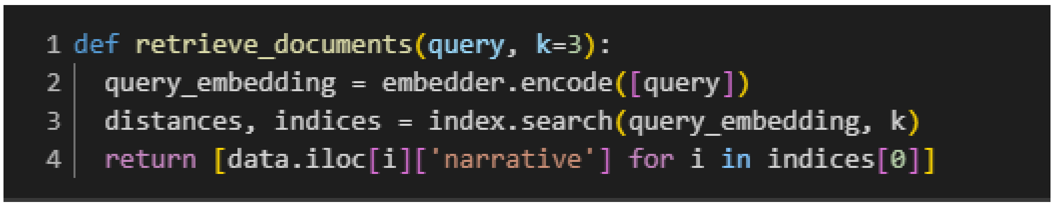


```python
dimension = embeddings.shape[1]
index = faiss.IndexFlatL2(dimension)

index.add(embeddings)
print("FAISS index initialized with total entries:", index.ntotal)

if index.ntotal == 0:
    raise ValueError("FAISS index is empty. Ensure embeddings are added.")
```

Outcome:

- The index was successfully populated with the embeddings.
- The index.ntotal value ensures all embeddings are added to the index.

3. **Query Processing and Document Retrieval**
   To enable real-time retrieval of relevant documents based on user queries, the following function was implemented:
   Document Retrieval Function:
   This function processes the user query, converts it into an embedding, and retrieves the top-k most relevant documents using the FAISS index.



```python
def retrieve_documents(query, k=3):
    query_embedding = embedder.encode([query])
    distances, indices = index.search(query_embedding, k)
    return [data.iloc[i]['narrative'] for i in indices[0]]
```

Parameters:

- query: The user's input query.
- k: The number of relevant documents to retrieve (default: 3).

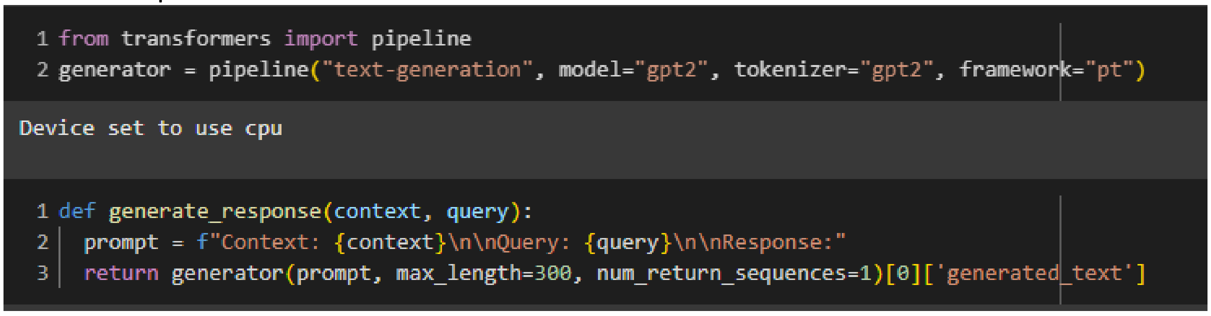
Output: A list of the top-k retrieved narratives.

4. **Response Generation**

   A response to the user query was generated by synthesizing the retrieved documents into a coherent reply. This was accomplished using a pre-trained text-generation model.

   Response Generation Function:

   This function combines the context (retrieved documents) and the query to generate a detailed response.



```python
from transformers import pipeline
generator = pipeline("text-generation", model="gpt2", tokenizer="gpt2", framework="pt")
```

```
Device set to use cpu
```

```python
def generate_response(context, query):
    prompt = f"Context: {context}\n\nQuery: {query}\n\nResponse:"
    return generator(prompt, max_length=300, num_return_sequences=1)[0]['generated_text']
```

Parameters:

- context: Concatenated retrieved documents.
- query: The user's input query.

Output: A generated response text based on the context and query.

Key Features of the Search System

- **Efficient Retrieval:** FAISS uses optimized algorithms for fast vector similarity search, making it suitable for large datasets.
- **Scalability:** The system can handle high-dimensional data while maintaining quick response times.

Summarization

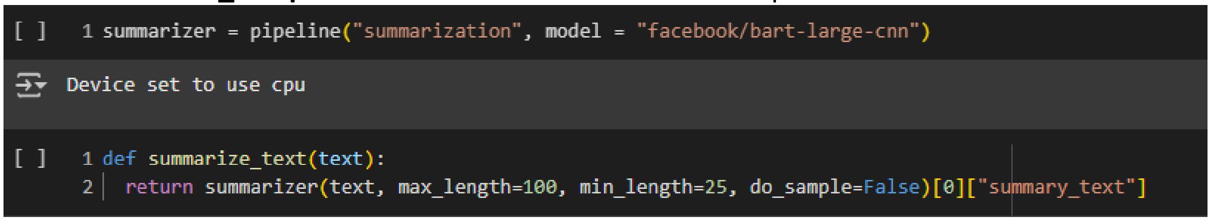
Objective

Summarize the retrieved documents before presenting them to the user, ensuring concise and informative outputs.

Summarization Approach

1. **Pre-trained Model**
   - A pre-trained BART (Bidirectional and Auto-Regressive Transformer) model, specifically facebook/bart-large-cnn, was used for summarization.
   - The model is designed for extractive summarization tasks, making it effective for condensing long texts while retaining key information.

2. **Implementation**
   - The summarizer was implemented using the Hugging Face pipeline API, which simplifies the application of state-of-the-art NLP models.
   - **Key parameters:**
     - **max_length:** Limits the length of the summary to ensure conciseness.
     - **min_length:** Ensures the summary captures enough detail.
     - **do_sample:** Set to False for deterministic outputs



```
summarizer = pipeline("summarization", model = "facebook/bart-large-cnn")
```

```
Device set to use cpu
```

```
def summarize_text(text):
  return summarizer(text, max_length=100, min_length=25, do_sample=False)[0]["summary_text"]
```

Workflow:

1. **Input:**
   Text retrieved from the Search and Retrieval module (e.g., user query results).
2. **Summarization Process:**
   a. Each document is passed to the summarize_text function.
   b. The function generates a concise summary by distilling the key points from the input text.
3. **Output:**
   A summary of each retrieved document is returned to the user, ensuring they receive the most relevant information without unnecessary details

Key Features

- **Conciseness:** The summaries are limited to a predefined length, making them easy to read.
- **Relevance:** By focusing on the main points, the summaries highlight the most critical information from the text.
- **Efficiency:** Leveraging a pre-trained model ensures high-quality summaries without extensive fine-tuning.

Potential Enhancements

- **Evaluation Metrics:**
  Use ROUGE (Recall-Oriented Understudy for Gisting Evaluation) to compare the model-generated summaries with human-written references, ensuring quality and relevance.
- **Custom Fine-Tuning:**
  Fine-tune the BART model on domain-specific data (e.g., customer complaints) to further enhance summarization accuracy.

Sentiment Analysis

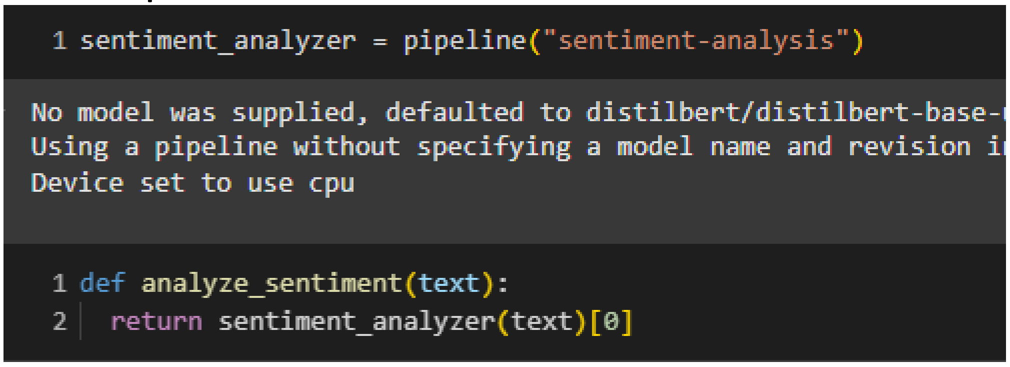
Objective

To analyze and provide sentiment insights based on the user’s query or the retrieved text, enabling a better understanding of the tone and emotions within the content.

Sentiment Analysis Approach

1. **Pre-trained Model**
   - **The Hugging Face Sentiment Analysis pipeline was used for this task.**
   - **This pipeline applies a pre-trained transformer model to classify text into sentiment categories, such as positive, negative, and neutral.**
2. **Implementation**
   - **The sentiment analysis was implemented using the Hugging Face pipeline API, which streamlines the integration of transformer-based models.**

   - **Key steps:**
     - **Pass the text input to the model.**
     - **Receive the predicted sentiment label and confidence score**



```
sentiment_analyzer = pipeline("sentiment-analysis")
```

```
No model was supplied, defaulted to distilbert/distilbert-base-
Using a pipeline without specifying a model name and revision i
Device set to use cpu
```

```
def analyze_sentiment(text):
  return sentiment_analyzer(text)[0]
```

Workflow

1. **Input:**
   Text from the user query or retrieved document.
2. **Sentiment Analysis Process:**
   - Each text snippet is passed to the analyze_sentiment function.
   - The model predicts the sentiment label along with a confidence score.
3. **Output:**
   The sentiment label (positive, negative, or neutral) and its corresponding confidence score are returned.

Applications in the Project

- **User Query Analysis:** Analyze the sentiment of user queries to better understand their concerns.
- **Document Insights:** Provide sentiment classifications for retrieved documents to highlight their emotional tone.

Potential Enhancements

- **Fine-Tuning:** Fine-tune the sentiment model on domain-specific data (e.g., financial complaints or product reviews) to improve performance.
- **Sentiment Distribution:** Visualize sentiment trends across the dataset to identify patterns or anomalies.

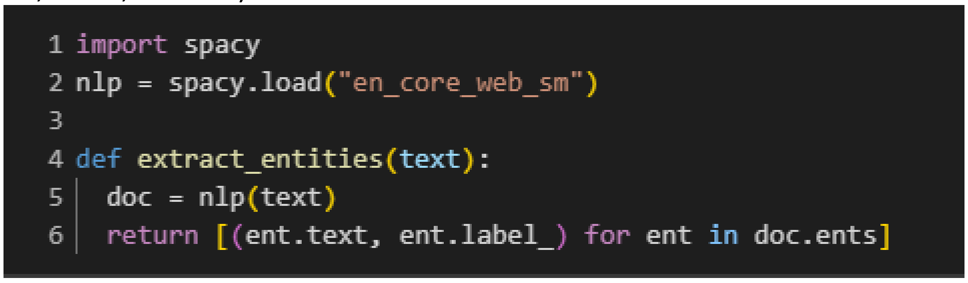
Entity Extraction

Objective

To extract and highlight key domain-specific entities from the chatbot's responses, enabling users to identify important information at a glance.

Entity Extraction Approach

1. **Pre-trained Model**
   - The spaCy NLP library was used for Named Entity Recognition (NER).
   - The model en_core_web_sm was chosen for its efficiency in identifying a wide range of entity types, such as dates, monetary amounts, and organizations.
2. **Implementation**
   - The text is processed using spaCy's nlp pipeline to identify entities.
   - Extracted entities include both the text and their corresponding labels (e.g., PERSON, ORG, DATE, MONEY).



```python
import spacy
nlp = spacy.load("en_core_web_sm")

def extract_entities(text):
    doc = nlp(text)
    return [(ent.text, ent.label_) for ent in doc.ents]
```

Workflow

1. **Input:**
   - Text from the chatbot's responses or the user query.
2. **Entity Extraction Process:**
   - The text is tokenized, and spaCy's NER module identifies entities.
   - Each entity is labeled based on its type.
3. **Output:**

- A list of tuples where each tuple contains:
  - The extracted entity.
  - Its corresponding label.

Key Features

- **Domain-Specific Insights:**
  Extracted entities like dates, monetary values, and organizations provide structured information for the chatbot's response.
- **Versatile Labeling:**
  The en_core_web_sm model recognizes a variety of entity types, including:
  - PERSON: Names of people.
  - ORG: Organizations.
  - MONEY: Monetary amounts.
  - DATE: Dates and time-related expressions.
  - GPE: Geographic locations.

Applications in the Project

- **Enhanced Responses:**
  - The chatbot can highlight extracted entities in its responses, making them more informative and actionable.
- **User Interaction:**
  - Entities like monetary values or dates can be emphasized, allowing users to focus on critical details.

Potential Enhancements

- **Custom Entity Recognition:**
  Train a domain-specific NER model to recognize entities not covered by the default spaCy model, such as:
  - Complaint IDs.
  - Product names.
  - Account numbers.
- **Visualization:**
  Use libraries like displacy to create visual representations of extracted entities within the text.

Interaction Logging

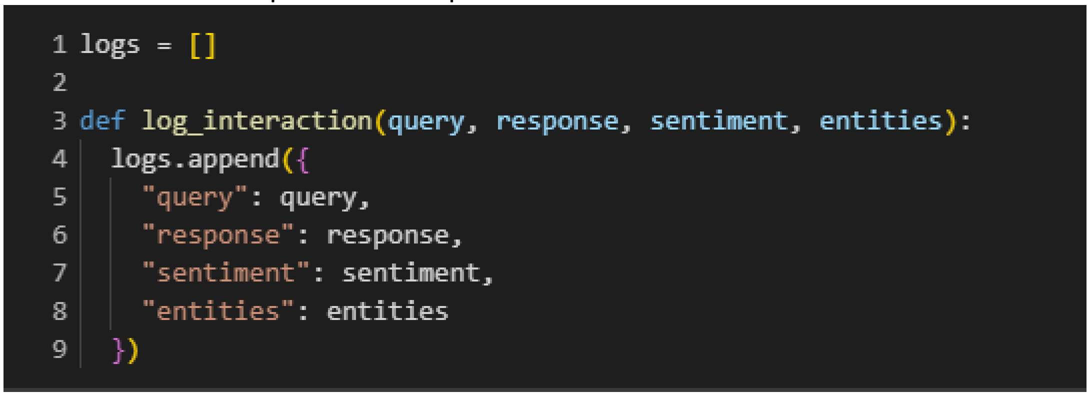
The Interaction Logging component of the chatbot system is designed to capture and store critical details from each interaction between the user and the chatbot. This enables post-interaction analysis and provides valuable insights into the chatbot's performance, user engagement, and areas for improvement. The key goal is to log all interactions for evaluation and analysis.

Purpose of Interaction Logging:

Track Queries and Responses: Capturing the queries and corresponding chatbot responses allows for the evaluation of the quality and relevance of the responses provided.

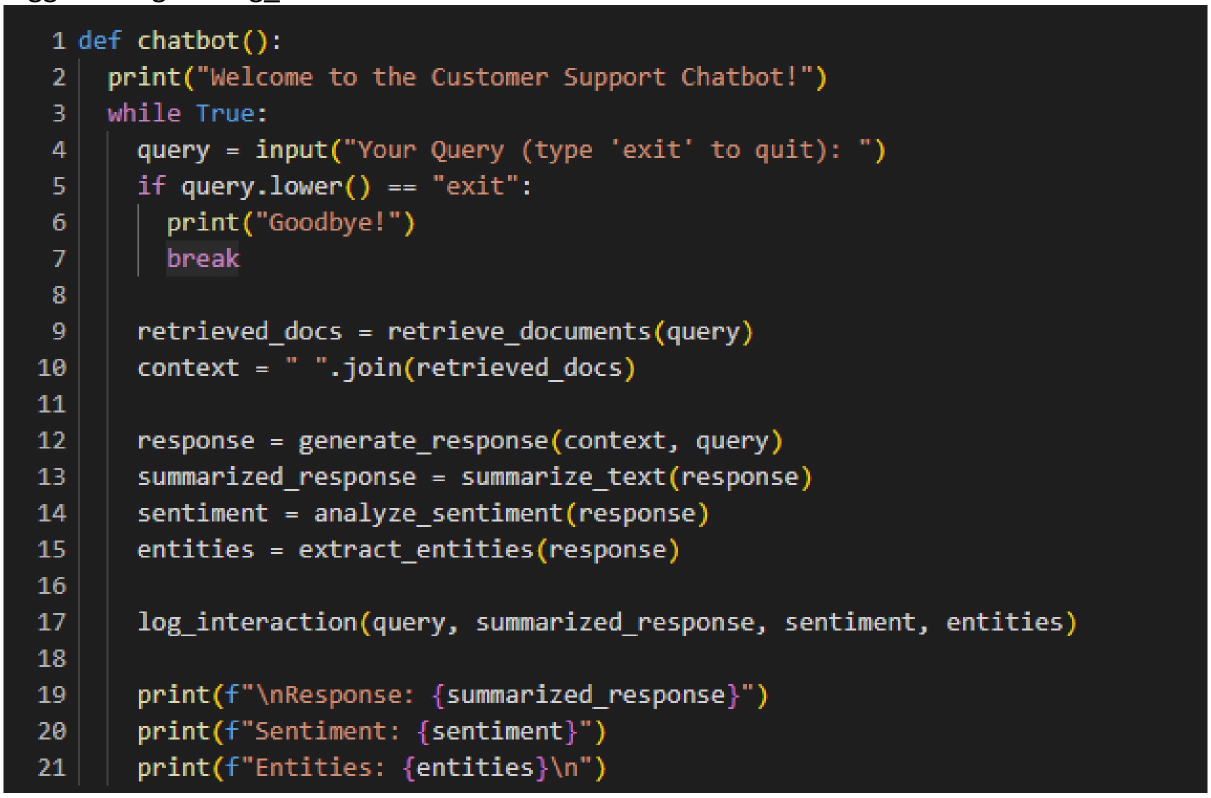
- **Sentiment Analysis:** Logging the sentiment of each response helps in understanding the user’s emotional tone, thereby allowing for the adjustment of the chatbot's tone and response style if necessary.
- **Entity Extraction:** Recording the extracted entities from the response enables the analysis of how well the chatbot identifies and responds to important concepts or keywords within the queries.
- **Performance Metrics:** By analyzing the logs, developers can identify patterns in frequent queries, response accuracy, sentiment trends, and entity recognition. This data can be used to continuously improve the chatbot’s performance.

Code Implementation for Interaction Logging:
The log_interaction function is used to log the query, chatbot’s response, sentiment, and extracted entities during every interaction. The logs are stored as dictionaries within the logs list. Here is the core part of the implementation:



```python
logs = []

def log_interaction(query, response, sentiment, entities):
  logs.append({
    "query": query,
    "response": response,
    "sentiment": sentiment,
    "entities": entities
  })
```

In the chatbot function, this logging mechanism is triggered after every interaction. The chatbot retrieves relevant documents based on the user's query, generates a response, summarizes it, performs sentiment analysis, and extracts entities. These details are then logged using the log_interaction function.



```python
def chatbot():
  print("Welcome to the Customer Support Chatbot!")
  while True:
    query = input("Your Query (type 'exit' to quit): ")
    if query.lower() == "exit":
      print("Goodbye!")
      break

    retrieved_docs = retrieve_documents(query)
    context = " ".join(retrieved_docs)

    response = generate_response(context, query)
    summarized_response = summarize_text(response)
    sentiment = analyze_sentiment(response)
    entities = extract_entities(response)

    log_interaction(query, summarized_response, sentiment, entities)

    print(f"\nResponse: {summarized_response}")
    print(f"Sentiment: {sentiment}")
    print(f"Entities: {entities}\n")
```

Benefits of Interaction Logging:
1. **Comprehensive Record of Interactions:** By logging every query and response, the system provides a detailed history of user interactions, which can be used to analyze the chatbot's ability to handle diverse queries and provide useful responses.
2. **Evaluation of Sentiment Trends:** Sentiment analysis helps in evaluating how users feel about the chatbot's responses, which could be used to fine-tune its emotional intelligence.

3. **Entity Tracking:** Capturing the entities helps in tracking the scope of topics the chatbot is interacting with and improving its capability to recognize and respond to different types of data effectively.
4. **Continuous Improvement:** Analysis of interaction logs allows the development team to identify common queries, chatbot limitations, and areas for optimization, such as improving response accuracy or enhancing sentiment detection.

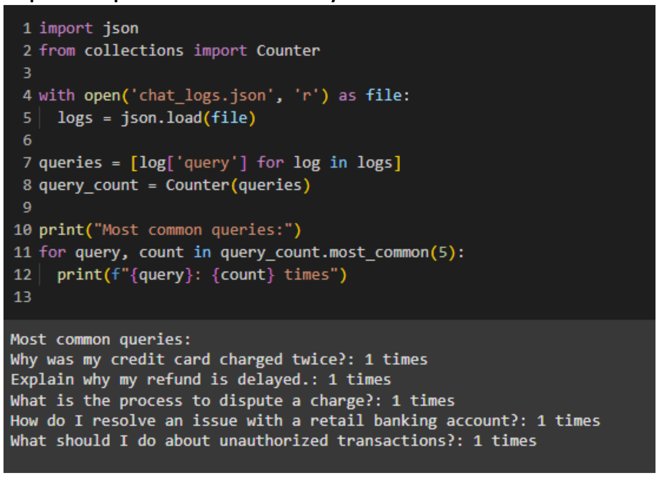
Conclusion:
The interaction logging mechanism in this chatbot system serves as an essential tool for ongoing evaluation and enhancement. By keeping track of user queries, responses, sentiment, and extracted entities, the chatbot's performance can be monitored and optimized over time, ensuring it becomes more accurate, user-friendly, and context-aware.

Explainability:

Explainability is crucial in AI systems, especially in the context of chatbots, to ensure that users and developers understand how and why the system generates responses, assigns sentiment labels, or retrieves certain documents. The goal of this component is to provide transparency into the chatbot's actions, making the decision-making process more interpretable and reliable.

Objective of Explainability:
- **Interpret Chatbot Responses:** Provide clear insights into why the chatbot gave a specific response.
- **Document Retrieval Explanations:** Explain why certain documents were retrieved in response to a query.
- **Sentiment Label Justification:** Offer reasoning behind the sentiment label assigned to user queries.

Code Implementation for Explainability:
1. Query Frequency Analysis:
The chatbot logs are analyzed to understand which queries are most common. This can help in identifying recurring user concerns, providing context to the chatbot’s responses, and ensuring it handles frequent queries effectively.



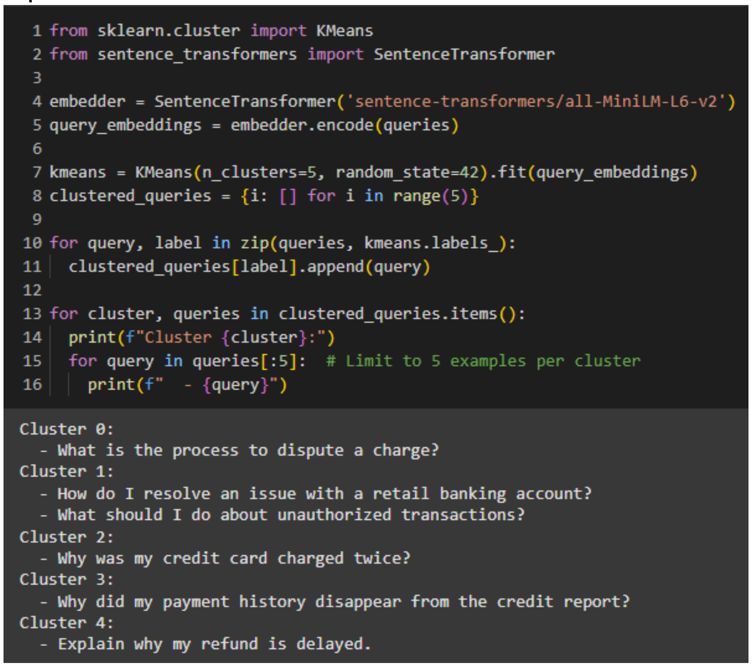
```python
import json
from collections import Counter

with open('chat_logs.json', 'r') as file:
    logs = json.load(file)

queries = [log['query'] for log in logs]
query_count = Counter(queries)

print("Most common queries:")
for query, count in query_count.most_common(5):
    print(f"{query}: {count} times")

```

```
Most common queries:
Why was my credit card charged twice?: 1 times
Explain why my refund is delayed.: 1 times
What is the process to dispute a charge?: 1 times
How do I resolve an issue with a retail banking account?: 1 times
What should I do about unauthorized transactions?: 1 times
```

This code calculates the most common queries by counting their occurrences in the logs, helping to highlight the most frequent user concerns.

2. Clustering of Similar Queries:

To identify patterns in user queries, KMeans clustering is used. This method groups similar queries together, which can assist in identifying specific themes or topics of interest that are shared across multiple user interactions.



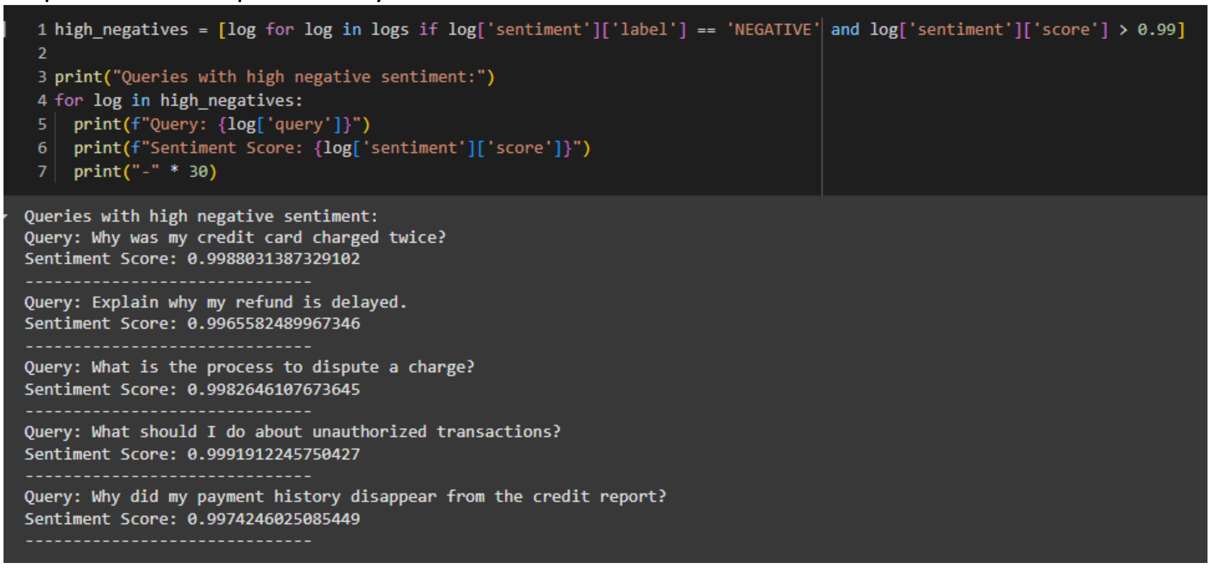
```python
from sklearn.cluster import KMeans
from sentence_transformers import SentenceTransformer

embedder = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')
query_embeddings = embedder.encode(queries)

kmeans = KMeans(n_clusters=5, random_state=42).fit(query_embeddings)
clustered_queries = {i: [] for i in range(5)}

for query, label in zip(queries, kmeans.labels_):
    clustered_queries[label].append(query)

for cluster, queries in clustered_queries.items():
    print(f"Cluster {cluster}:")
    for query in queries[:5]:  # Limit to 5 examples per cluster
        print(f"  - {query}")
```

```
Cluster 0:
  - What is the process to dispute a charge?
Cluster 1:
  - How do I resolve an issue with a retail banking account?
  - What should I do about unauthorized transactions?
Cluster 2:
  - Why was my credit card charged twice?
Cluster 3:
  - Why did my payment history disappear from the credit report?
Cluster 4:
  - Explain why my refund is delayed.
```

This clustering technique groups similar queries together, making it easier to identify recurring issues and providing an opportunity to tailor responses more effectively.

3. High Sentiment Analysis:

Identifying queries with extreme sentiment (e.g., highly negative sentiment) can help in understanding user frustration or dissatisfaction. The chatbot can then be designed to respond more empathetically or escalate the issue as needed.

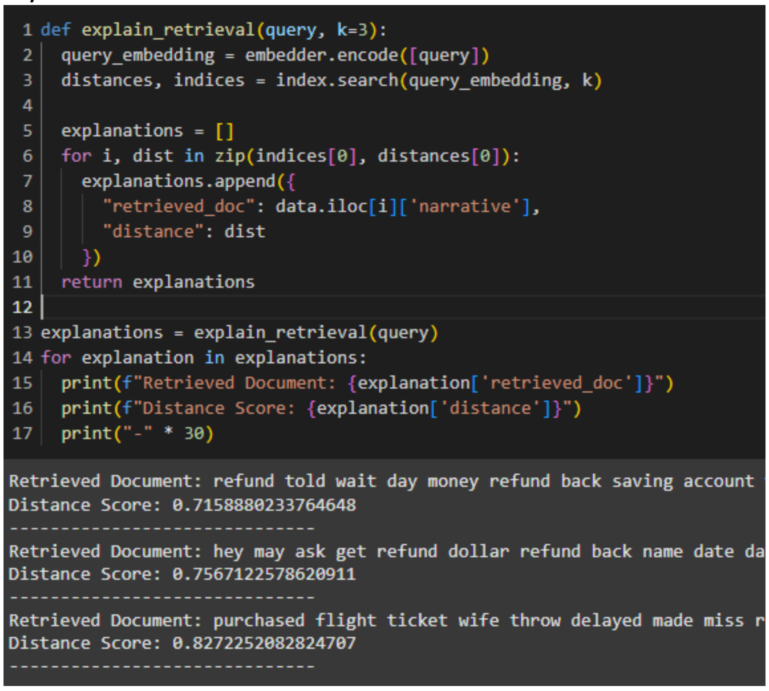


```python
high_negatives = [log for log in logs if log['sentiment']['label'] == 'NEGATIVE' and log['sentiment']['score'] > 0.99]

print("Queries with high negative sentiment:")
for log in high_negatives:
    print(f"Query: {log['query']}")
    print(f"Sentiment Score: {log['sentiment']['score']}")
    print("-" * 30)
```

```
Queries with high negative sentiment:
Query: Why was my credit card charged twice?
Sentiment Score: 0.9988031387329102
------------------------------
Query: Explain why my refund is delayed.
Sentiment Score: 0.9965582489967346
------------------------------
Query: What is the process to dispute a charge?
Sentiment Score: 0.9982646107673645
------------------------------
Query: What should I do about unauthorized transactions?
Sentiment Score: 0.9991912245750427
------------------------------
Query: Why did my payment history disappear from the credit report?
Sentiment Score: 0.9974246025085449
------------------------------
```

This approach focuses on queries that have very negative sentiment, helping to highlight situations where users may need additional assistance or where the chatbot's tone needs adjustment.

4. Document Retrieval Explanations:
The explain_retrieval function provides transparency into the document retrieval process. By using an embedding-based search (e.g., through the SentenceTransformer and k-nearest neighbor search), this function explains which documents were retrieved in response to a query and why they were chosen.



```python
def explain_retrieval(query, k=3):
    query_embedding = embedder.encode([query])
    distances, indices = index.search(query_embedding, k)

    explanations = []
    for i, dist in zip(indices[0], distances[0]):
        explanations.append({
            "retrieved_doc": data.iloc[i]['narrative'],
            "distance": dist
        })
    return explanations

explanations = explain_retrieval(query)
for explanation in explanations:
    print(f"Retrieved Document: {explanation['retrieved_doc']}")
    print(f"Distance Score: {explanation['distance']}")
    print("-" * 30)
```

```
Retrieved Document: refund told wait day money refund back saving account
Distance Score: 0.7158880233764648
------------------------------
Retrieved Document: hey may ask get refund dollar refund back name date da
Distance Score: 0.7567122578620911
------------------------------
Retrieved Document: purchased flight ticket wife throw delayed made miss r
Distance Score: 0.8272252082824707
------------------------------
```

This code snippet provides a breakdown of the documents retrieved for a given query. The "distance score" is used to explain the relevance of each document, helping users understand why specific information was chosen over others.

Benefits of Explainability:

1. **Transparency in Decision Making:** Users and developers can understand how the chatbot arrives at its responses, document retrieval choices, and sentiment assessments, leading to a more trustworthy system.
2. **Improved User Trust:** By providing reasons behind the actions of the chatbot (such as why a certain document was retrieved), users can better appreciate the logic of the system and feel more confident in its accuracy.
3. **Enhanced Debugging and Model Improvement:** The ability to review clustering results, sentiment analysis outcomes, and document retrieval choices makes it easier to identify potential issues, allowing for focused improvements in the chatbot's performance.

Conclusion:
Explainability is an essential component for making AI-driven systems like chatbots more interpretable and trustworthy. By leveraging techniques such as clustering, sentiment analysis, and document retrieval explanations, this system can provide transparency into the chatbot's actions, fostering greater user trust and enabling continuous improvement.