**Report on RAG-Based Chatbot for Therapists - Therapist Helper**

**1. Introduction-**

This report details the evaluation and optimization of a Retrieval-Augmented Generation (RAG) pipeline for a chatbot designed to assist in therapy sessions. The focus is on calculating key performance metrics and implementing improvements to enhance the system's accuracy, relevance, and robustness.

**2. Methodology-**

**2.1. Retrieval Metrics Calculation**

**1. Context Precision:**
- Measures the proportion of retrieved contexts that are relevant to the user's query.
- Calculation: Precision = Number of relevant contexts retrieved / Total number of contexts retrieved

**2. Context Recall:**
- Assesses the system's ability to retrieve all relevant contexts.
- Calculation: Recall = Number of relevant contexts retrieved / Total number of relevant contexts

**3. Context Relevance:**
- Evaluates the relevance of the retrieved contexts using cosine similarity between query and context embeddings.
- Calculation: Average cosine similarity score between query embedding and retrieved context embeddings.

**4. Noise Robustness:**
- Tests the system's performance under noisy inputs by adding random noise to the query.
- Calculation: Precision and recall metrics are recalculated after introducing noise to the query.

**2.2. Generation Metrics Calculation**

**1. Faithfulness:**
- Measures how accurately the generated answers reflect the reference answers.
- Calculation: Faithfulness = Number of correct answers / Total number of reference answers

**2. Answer Relevance:**
- Evaluates how relevant the generated answer is to the query.
- Calculation: Cosine similarity between the query and the generated answer embeddings.

**3. Information Integration:**

- Assesses the ability of the system to integrate and present information cohesively.
- Calculation: Cosine similarity between the combined embeddings of retrieved contexts and the generated answer.

**4. Counterfactual Robustness:**
- Measures the system's response to counterfactual or contradictory queries.
- Calculation: Proportion of answers that appropriately differ from counterfactual queries.

**5. Negative Rejection:**
- Tests the system's ability to reject or handle inappropriate queries.
- Calculation: Proportion of answers that appropriately handle or reject negative queries.

**6. Latency:**
- Measures the time taken from receiving a query to delivering an answer.
- Calculation: Time taken in seconds for retrieval and generation processes.

## 3. Results

### 3.1. Retrieval Metrics-

- Context Precision: 1.0
- Context Recall: 0.75
- Context Relevance: 0.687
- Noise Robustness Precision: 1.0
- Noise Robustness Recall: 0.75

### 3.2. Generation Metrics
- Faithfulness: 0.0
- Answer Relevance: 0.696
- Information Integration: 0.54
- Counterfactual Robustness: 0.25
- Negative Rejection: 0.0
- Latency: 1.476

## 4. Methods Proposed and Implemented for Improvement

1. Improving Context Recall:
   - Method: Enhanced the context retrieval mechanism by fine-tuning the similarity threshold and increasing the diversity of retrieved contexts.

2. Increasing Faithfulness:
   - Method: Implemented a post-processing step where the generated answers are cross-verified against reference answers to improve faithfulness.

3. Counterfactual Robustness and Negative Rejection:
- Method: Implemented stricter filters and validation checks to better handle counterfactual and negative queries.

**5. Comparative Analysis**

**- Before Improvements:**
- Context Recall was at 0.75, indicating that not all relevant contexts were being retrieved.
- Faithfulness was at 0.0, suggesting a lack of alignment between generated answers and reference answers.
- Counterfactual Robustness was at 0.25, showing limited ability to handle counterfactual scenarios.
- Negative Rejection was at 0.0, indicating a need for better handling of inappropriate queries.

**- After Improvements:**
- The enhancements led to a more comprehensive retrieval of relevant context.
- Faithfulness saw improvements through the implementation of a cross-verification mechanism.
- The system's robustness against counterfactual and negative queries showed some improvement.

**6. Challenges Faced and How They Were Addressed**

1. Handling Noisy Queries:
- Challenge: Ensuring the system remains robust when faced with noisy or irrelevant inputs.
- Solution: Implemented a noise-handling mechanism that maintained high precision and recall under noisy conditions.

2. Maintaining Token Limits:
- Challenge: Ensuring inputs do not exceed the maximum token limit of the model.
- *Solution*: Used token management techniques, including truncation and summarization, to keep inputs within the limit.

3. Evaluating Faithfulness:
- Challenge: Accurately measuring the faithfulness of generated answers.
- Solution: Introduced a cross-verification process to compare generated answers against a set of reference answers.

4. Improving Robustness:
-Challenge: Enhancing the system's robustness to handle counterfactual and negative queries effectively.
- Solution: Developed stricter validation and filtering mechanisms to improve the handling of such queries.

## 7. Steps for setting up the application -

## Setup

### Prerequisites

- Python 3.7 or higher
- Required Python libraries: OpenAI, scikit-learn, numpy, pandas, transformers, tiktoken

### Installation

1. Clone the repository:

```
git clone ithub.com/Mansi-Dabriwal/TherapistHelper.git
cd RAG-therapy-chatbot
```

2. Install the required libraries:

```
pip install -r requirements.txt
```

3. Set up OpenAI API key:
   - Create a `.env` file in the root directory with the following content:

```
OPENAI_API_KEY=your-openai-api-key
```

## 8. Steps to run the application -

## Usage

### Running the Evaluation

1. **Run the Application:**

```
python fastApi.py
```

2. **Run the Streamlit Application:**

```
streamlit run app.py
```

3. **Run the Evaluation:**

```
python evaluation.py
```

This script calculates the performance metrics and outputs the results.

## 9. GitHub Link -
[https://github.com/Mansi-Dabriwal/TherapistHelper](https://github.com/Mansi-Dabriwal/TherapistHelper)

## 10. You-Tube Link -
[https://www.youtube.com/watch?v=c3FsyAKIvtc](https://www.youtube.com/watch?v=c3FsyAKIvtc)

## 11. Code -

## evaluate.py

```python
from retrieval import retrieve_contexts, context_precision_recall, context_relevance, test_noise_robustness
from generation import generate_answer, faithfulness, answer_relevance, information_integration, counterfactual_robustness, negative_re
import numpy as np
import pandas as pd
import openai

openai.api_key = "add-your-key"
EMBEDDING_MODEL = "text-embedding-ada-002"
GPT_MODEL = "gpt-4"

# Load preprocessed data
transcripts = pd.read_csv('embedded_transcripts.csv')
embeddings = np.load('embeddings.npy')

# Example usage
query = "How is the patient handling anxiety?"
relevant_contexts = [
"Patient feels anxious due to a recent car accident.",
"Patient avoids driving and experiences sleepless nights.",
"Patient is using alcohol as a coping mechanism and expressing resilience.",
"Patient is experiencing a lot of stress and anxiety related to work deadlines and performance expectations"
]

counterfactual_queries = [
    "What if the patient's anxiety was not triggered by the car accident but by a recent job change?",
    "How would the patient's coping strategies change if they had received immediate support after the car accident?",
    "What if the patient had a different stressor, such as financial issues, affecting their sleep and concentration?",
    "How might the patient's daily routine and relationships be affected if their anxiety were managed more effectively through therapy
]

negative_queries = [
    "What if the therapist ignored the patient's anxiety and focused only on unrelated issues?",
    "How should the therapist respond if the patient provides misleading information about their symptoms?",
    "What if the therapist was unable to address the patient's needs due to personal biases?",
    "How should the therapist handle a situation where the patient's symptoms are exaggerated to avoid work responsibilities?"
]

```

```python
# Retrieval Metrics
precision, recall = context_precision_recall(query, embeddings, transcripts, relevant_contexts)
print(f"Context Precision: {precision}")
print(f"Context Recall: {recall}")

relevance = context_relevance(query, retrieve_contexts(query, embeddings, transcripts))
print(f"Context Relevance: {relevance}")

noise_precision, noise_recall = test_noise_robustness(query, embeddings, transcripts, relevant_contexts)
print(f"Noise Robustness Precision: {noise_precision}")
print(f"Noise Robustness Recall: {noise_recall}")

# Generation Metrics
reference_answers = [
    "Anxiety can be managed through techniques such as cognitive-behavioral therapy (CBT), mindfulness practices, and grounding exercis
    "You should try to establish a consistent bedtime routine, use relaxation techniques like progressive muscle relaxation or guided i
    "To handle work stress, consider discussing your workload with your supervisor and exploring temporary adjustments. Techniques to i
    "It's important to communicate openly with your loved ones about what you're experiencing. Managing irritability involves learning
]
generated_answers = [generate_answer(retrieve_contexts(query, embeddings, transcripts))]
faithfulness_score = faithfulness(reference_answers, generated_answers)
print(f"Faithfulness: {faithfulness_score}")

relevance_score = answer_relevance(query, generated_answers[0])
print(f"Answer Relevance: {relevance_score}")

integration_score = information_integration(retrieve_contexts(query, embeddings, transcripts), generated_answers[0])
print(f"Information Integration: {integration_score}")

robustness_score = counterfactual_robustness(counterfactual_queries, generated_answers)
print(f"Counterfactual Robustness: {robustness_score}")

rejection_score = negative_rejection(negative_queries, generated_answers)
print(f"Negative Rejection: {rejection_score}")

latency = measure_latency(query, retrieve_contexts, generate_answer)
print(f"Latency: {latency} seconds")
```

# generation.py -

```python
import openai
from transformers import GPT2TokenizerFast
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import TfidfVectorizer
import tiktoken
import numpy as np

# Initialize the GPT-2 tokenizer
tokenizer = GPT2TokenizerFast.from_pretrained("gpt2")

MAX_TOKENS = 8192  # Maximum number of tokens the model can handle
SAFE_BUFFER = 100  # Buffer to ensure we stay well within limits

def generate_answer(retrieved_contexts):

    # Join the retrieved contexts into a single string
    context = " ".join(retrieved_contexts)

    # Tokenize the context to count tokens accurately
    context_tokens = tokenizer.encode(context, return_tensors='pt').size(1)

    # If the context exceeds the maximum token limit, summarize or truncate it
    if context_tokens + SAFE_BUFFER > MAX_TOKENS:
        if 'summarize_text' in globals():
            context = summarize_text(context, model="gpt-4")
        else:
            context_tokens = tokenizer.encode(context, return_tensors='pt').size(1)
            # Truncate the context to fit within the limit, if summarization isn't defined
            while context_tokens + SAFE_BUFFER > MAX_TOKENS:
                context = context[:len(context) // 2]  # Reduce the length by half
                context_tokens = tokenizer.encode(context, return_tensors='pt').size(1)

    # Create the API request
    response = openai.ChatCompletion.create(
        model="gpt-4",
        messages=[{"role": "system", "content": context}]
    )
```

```python
    # Return the generated response
    return response['choices'][0]['message']['content']

encoding = tiktoken.encoding_for_model("gpt-3.5-turbo")

def count_tokens(text):
    """Estimate token count for a given text."""
    return len(encoding.encode(text))

def truncate_text(text, max_tokens):
    """Truncate text to fit within the maximum token limit."""
    tokens = encoding.encode(text)
    return encoding.decode(tokens[:max_tokens])

def embed_text(text, model="text-embedding-ada-002"):
    """Embed the text using OpenAI's embedding model."""
    # Truncate the text if it exceeds the maximum token limit
    token_count = count_tokens(text)
    if token_count > MAX_TOKENS:
        text = truncate_text(text, MAX_TOKENS)

    response = openai.Embedding.create(input=[text], model=model)
    return response['data'][0]['embedding']

def faithfulness(reference_answers, generated_answers):
    correct_answers = sum([1 for ref, gen in zip(reference_answers, generated_answers) if ref == gen])
    return correct_answers / len(reference_answers) if reference_answers else 0

def answer_relevance(query, generated_answer):
    query_embedding = embed_text(query)
    generated_answer_embedding = embed_text(generated_answer)
    return cosine_similarity([query_embedding], [generated_answer_embedding])[0][0]

def information_integration(retrieved_contexts, generated_answer):
    retrieved_info = ' '.join(retrieved_contexts)
    retrieved_info_embedding = embed_text(retrieved_info)
    generated_answer_embedding = embed_text(generated_answer)
```

```python
    correct_answers = sum([1 for ref, gen in zip(reference_answers, generated_answers) if ref == gen])
    return correct_answers / len(reference_answers) if reference_answers else 0

def answer_relevance(query, generated_answer):
    query_embedding = embed_text(query)
    generated_answer_embedding = embed_text(generated_answer)
    return cosine_similarity([query_embedding], [generated_answer_embedding])[0][0]

def information_integration(retrieved_contexts, generated_answer):
    retrieved_info = ' '.join(retrieved_contexts)
    retrieved_info_embedding = embed_text(retrieved_info)
    generated_answer_embedding = embed_text(generated_answer)
    return cosine_similarity([retrieved_info_embedding], [generated_answer_embedding])[0][0]

def counterfactual_robustness(counterfactual_queries, generated_answers):
    robust = sum([1 for query, answer in zip(counterfactual_queries, generated_answers) if query != answer])
    return robust / len(counterfactual_queries) if counterfactual_queries else 0

def negative_rejection(negative_queries, generated_answers):
    negative_handling = sum([1 for query, answer in zip(negative_queries, generated_answers) if "inappropriate" in answer])
    return negative_handling / len(negative_queries) if negative_queries else 0

def measure_latency(query, retrieval_function, generation_function):
    import time
    start_time = time.time()
    retrieved_contexts = retrieval_function(query)
    generated_answer = generation_function(retrieved_contexts)
    end_time = time.time()
    return end_time - start_time
```

**retrieval.py -**

```python
import numpy as np
import pandas as pd
from sklearn.metrics.pairwise import cosine_similarity
import openai
import tiktoken

# Load preprocessed data
transcripts = pd.read_csv('embedded_transcripts.csv')
embeddings = np.load('embeddings.npy')

# Define the maximum number of tokens for the embedding model
MAX_TOKENS = 4096

# Initialize the tokenizer for embedding
encoding = tiktoken.encoding_for_model("gpt-3.5-turbo")

def count_tokens(text):
    """Estimate token count for a given text."""
    return len(encoding.encode(text))

def truncate_text(text, max_tokens):
    """Truncate text to fit within the maximum token limit."""
    tokens = encoding.encode(text)
    return encoding.decode(tokens[:max_tokens])

def embed_text(text, model="text-embedding-ada-002"):
    """Embed the text using OpenAI's embedding model."""
    # Truncate the text if it exceeds the maximum token limit
    token_count = count_tokens(text)
    if token_count > MAX_TOKENS:
        text = truncate_text(text, MAX_TOKENS)

    response = openai.Embedding.create(input=[text], model=model)
    return response['data'][0]['embedding']

def retrieve_contexts(query, embeddings, transcripts, top_k=5):
    """Retrieve top_k contexts relevant to the query while managing token limits."""
```

```python
def retrieve_contexts(query, embeddings, transcripts, top_k=5):
    """Retrieve top_k contexts relevant to the query while managing token limits."""
    # Embed the query
    query_embedding = embed_text(query)

    # Compute similarity between the query and each context
    similarities = cosine_similarity([query_embedding], embeddings)

    # Get indices of top_k most similar contexts
    top_k_indices = np.argsort(similarities[0])[-top_k:]

    # Retrieve top_k contexts and manage their length
    top_contexts = []
    for idx in reversed(top_k_indices):  # Reverse to maintain descending order of similarity
        context = transcripts.iloc[idx]['Transcript']
        if count_tokens(context) > MAX_TOKENS:
            context = truncate_text(context, MAX_TOKENS)
        top_contexts.append(context)

    return top_contexts

def is_similar(text1, text2, threshold=0.5):
    """Check if two texts are similar based on cosine similarity."""
    # Embed both texts
    embedding1 = embed_text(text1)
    embedding2 = embed_text(text2)

    # Calculate cosine similarity
    similarity = cosine_similarity([embedding1], [embedding2])[0][0]
    return similarity >= threshold

def context_precision_recall(query, embeddings, transcripts, relevant_contexts, top_k=5, threshold=0.5):
    retrieved_contexts = retrieve_contexts(query, embeddings, transcripts, top_k)

    retrieved_relevant = []
    for context in retrieved_contexts:
```

```python
    retrieved_contexts = retrieve_contexts(query, embeddings, transcripts, top_k)

    retrieved_relevant = []
    for context in retrieved_contexts:
        for relevant_context in relevant_contexts:
            if is_similar(context, relevant_context, threshold=threshold):
                retrieved_relevant.append(context)
                break  # Stop after finding the first match

    precision = len(retrieved_relevant) / len(retrieved_contexts) if retrieved_contexts else 0
    recall = len(retrieved_relevant) / len(relevant_contexts) if relevant_contexts else 0

    return precision, recall

def context_relevance(query, retrieved_contexts):
    query_embedding = embed_text(query)
    context_embeddings = [embed_text(context) for context in retrieved_contexts]
    similarities = cosine_similarity([query_embedding], context_embeddings)
    return np.mean(similarities)

def add_noise(query, noise_level=0.1):
    words = query.split()
    num_noisy_words = int(len(words) * noise_level)
    noisy_indices = np.random.choice(len(words), num_noisy_words, replace=False)
    noisy_words = ["<noise>" for _ in noisy_indices]
    for idx in noisy_indices:
        words[idx] = noisy_words.pop()
    return ' '.join(words)

def test_noise_robustness(query, embeddings, transcripts, relevant_contexts, noise_level=0.1, top_k=5, threshold=0.5):
    noisy_query = add_noise(query, noise_level)
    precision, recall = context_precision_recall(noisy_query, embeddings, transcripts, relevant_contexts, top_k, threshold)
    return precision, recall
```

## 12. Conclusion

The evaluation and optimization of the RAG-based chatbot demonstrated the importance of both retrieval and generation metrics in assessing the performance of such systems. The implemented improvements showed potential in enhancing the system's accuracy and robustness, although further refinement and testing are needed. Future work will focus on enhancing counterfactual robustness and handling negative queries more effectively.