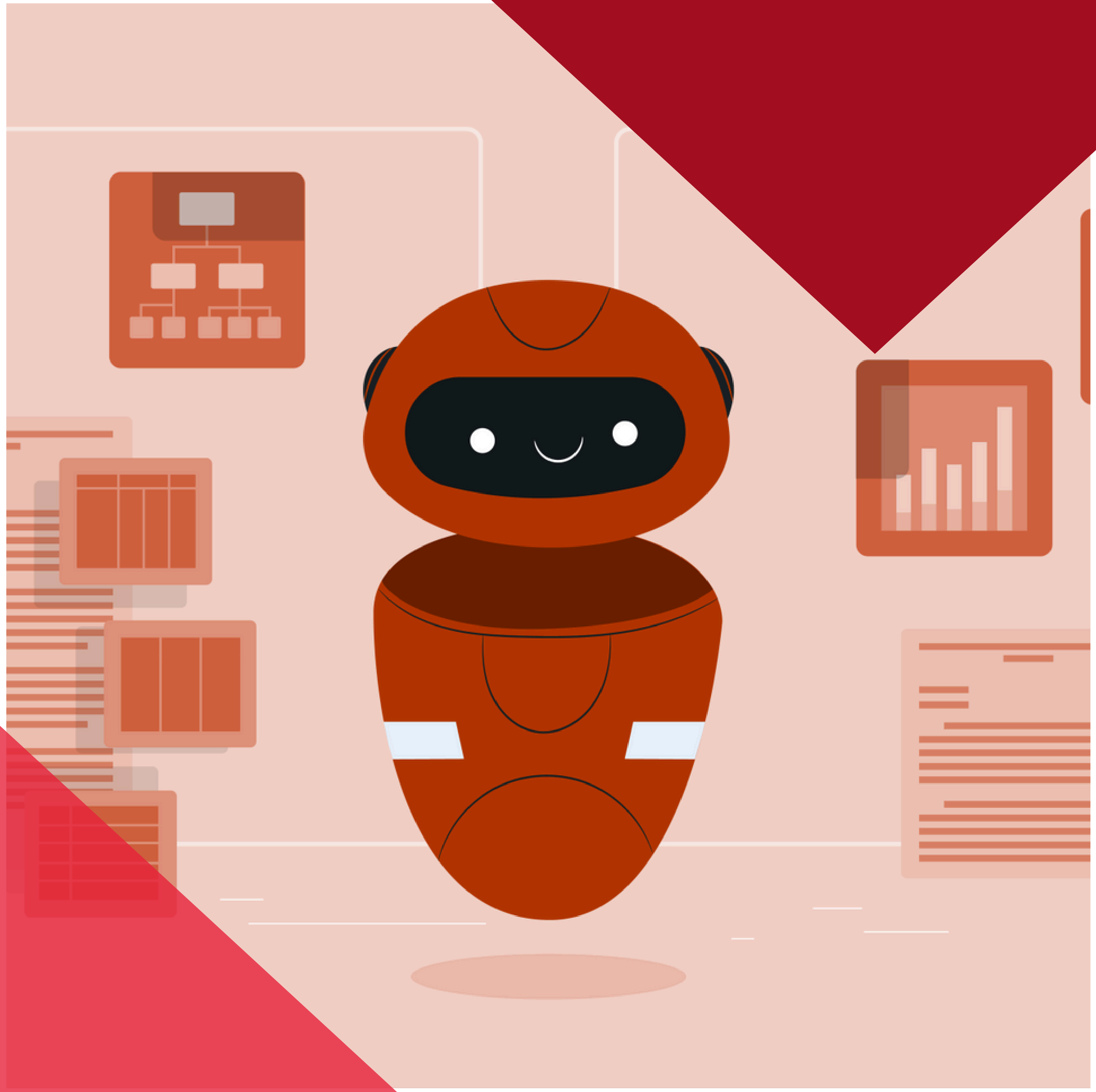
Metrics Evaluation of RAG Chatbot

Calculating and Reporting Metrics of RAG Pipeline



Like | Share | Subscribe

Agenda

What are Performance Metrics?

Initial Challenge

whether the output from the RAG system:

- Is of high-quality content, coherent and factually correct.
- Is relevant and complete
- Doesn’t have lot of noise
- Is not harmful, malicious and toxic
- Is fast in terms of performance

Performance metrics are quantitative measures used to evaluate the effectiveness, accuracy, and efficiency of a system, such as a Retrieval-Augmented Generation (RAG) chatbot.

They help in understanding how well the chatbot performs in various aspects, ensuring it meets the desired standards and provides a good user experience.

Retrieval Metrics

- **Context Precision:** Measure how accurately the retrieved context matches the user's query.

$$\text{Precision} = \frac{\text{Number of True Positives}}{\text{Number of Retrieved Contexts}}$$

- **Context Recall:** Evaluate the ability to retrieve all relevant contexts for the user's query.

$$\text{Recall} = \frac{\text{Number of True Positives}}{\text{Number of Relevant Contexts}}$$

- **Context Relevance:** Assess the relevance of the retrieved context to the user's query.

$$\text{Cosine Similarity} = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

- **Context Entity Recall:** Determine the ability to recall relevant entities within the context.

$$\text{Entity Recall} = \frac{\text{Number of Correctly Recalled Entities}}{\text{Total Number of Entities in Query}}$$

- **Noise Robustness:** Test the system's ability to handle noisy or irrelevant inputs.

$$\text{Noise Robustness} = \text{Relevance Score of Noisy Query}$$

Generation Metrics

- **Faithfulness:** Measure the accuracy and reliability of the generated answers.

$$\text{Faithfulness} = \text{Sigmoid Output of BERT}$$

- **Answer Relevance:** Evaluate the relevance of the generated answers to the user's query.

$$\text{BLEU Score} = \text{Geometric Mean of Precision for n-grams} \times \text{Brevity Penalty}$$

- **Information Integration:** Assess the ability to integrate and present information cohesively.

$$\text{ROUGE-1} = \frac{\text{Number of Overlapping Unigrams}}{\text{Total Number of Unigrams in Ground Truth}}$$

$$\text{ROUGE-L} = \frac{\text{Length of Longest Common Subsequence}}{\text{Total Length of Ground Truth}}$$

- **Counterfactual Robustness:** Test the robustness of the system against counterfactual or contradictory queries.

$$\text{Counterfactual Robustness} = \text{Generated Answer} \neq \text{Counterfactual Answer}$$

- **Negative Rejection:** Measure the system's ability to reject and handle negative or inappropriate queries.

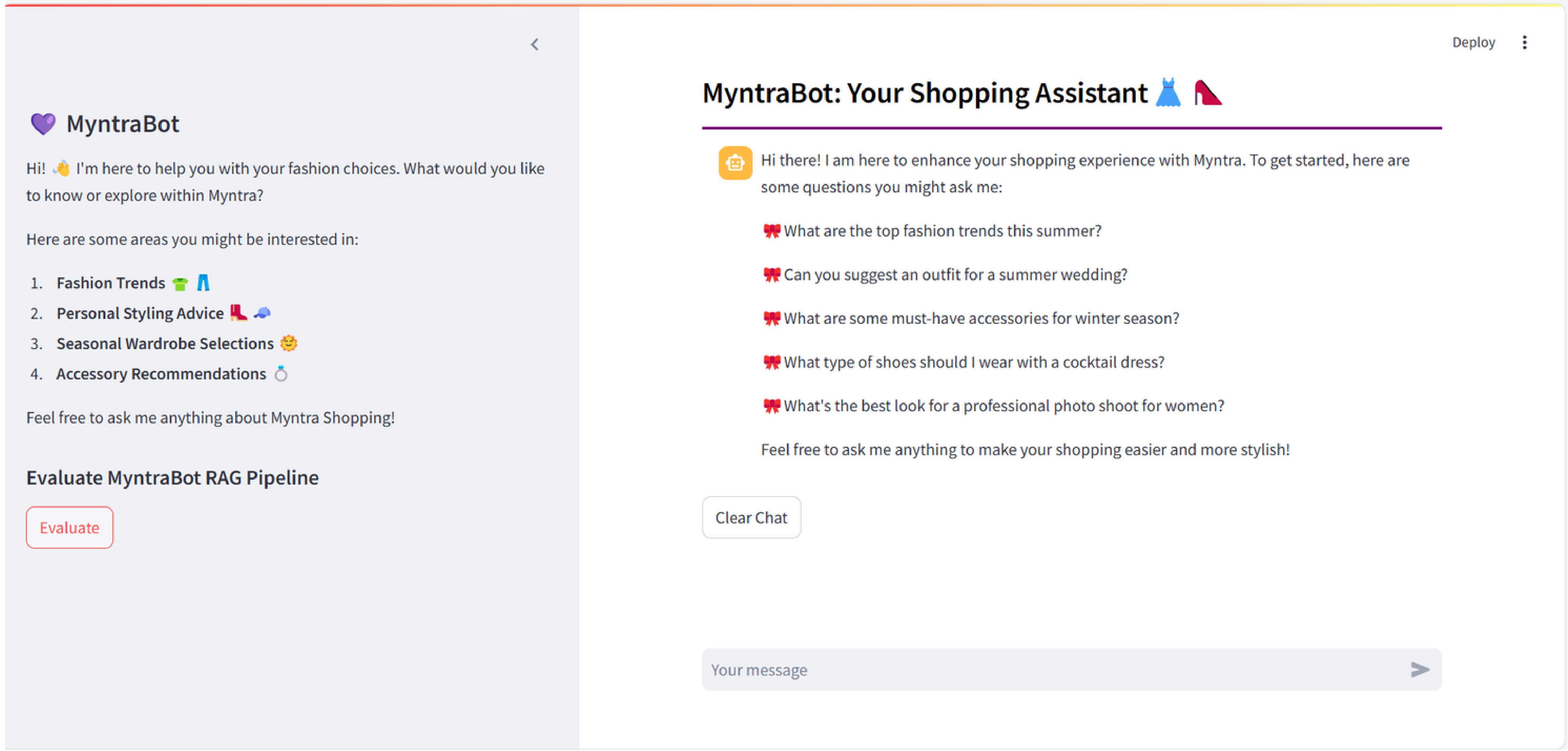
$$\text{Negative Rejection} = \text{Absence of Negative Keywords in Generated Answer}$$

Latency

- **Latency:** Measure the response time of the system from receiving a query to delivering an answer.

$$\text{Latency} = \text{End Time} - \text{Start Time}$$

Lets Implement Performance Metrics!



Import all required dependencies
Load NLP model
Define Bert Tokenizer and model

```python
import numpy as np
import time
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import TfidfVectorizer
from nltk.translate.bleu_score import sentence_bleu
from rouge_score import rouge_scorer
import spacy
import torch
from transformers import BertTokenizer, BertForSequenceClassification

# Load the English NLP model
nlp = spacy.load('en_core_web_sm')

# Define BERT tokenizer and model for later use
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertForSequenceClassification.from_pretrained('bert-base-uncased')
```

Query, Ground truth, Get_response and Vector_store

- **queries:** List of user queries to be answered by the chatbot.
- **ground_truths:** List of dictionaries containing the expected contexts and answers for the queries.
- **get_response:** Function that generates a response for a given query using the RAG pipeline.
- **vector_store:** Storage for vectorized text representations used for retrieving relevant contexts based on similarity searches.

```python
# Evaluation section
st.sidebar.header("Evaluate MyntraBot RAG Pipeline")

if st.sidebar.button('Evaluate'):

    queries = ["I want a sweatshirt for boy in yellow color"]
    ground_truths = [
        {
            # 'contexts':[product_data],
            'contexts':["ProductID: 1000894, ProductName: U.S. Polo Assn. Kids Boys Yellow Hood
            # 'answer': "Bubblegummers Boys Purple Printed Sports Sandals (ProductID: 10001209)
            'answer': '''Sure, I can help with that. Myntra offers a U.S. Polo Assn. Kids Boys
                        ProductID: 1000894
                        ProductName: U.S. Polo Assn. Kids Boys Yellow Hooded Sweatshirt
                        ProductBrand: U.S. Polo Assn. Kids
                        Gender: Boys
                        Price (INR): 899
                        Description: Yellow sweatshirt, has an attached hood with drawstring fa
                        PrimaryColor: Yellow
                        This sweatshirt might be a great fit for what you're looking for.'''
        }
    ]

    metrics = evaluate_metrics(queries, ground_truths, get_response, vector_store)
    st.sidebar.subheader("Evaluation Metrics")
    for metric, values in metrics.items():
        st.sidebar.write(f"{metric}: {np.mean(values)}")
```

Precision & Recall

```python
def convert_to_list(data):
    # Convert nested list of strings into a flat list of strings
    data_list = []
    for listItem in data:
        for item in listItem.split():
            data_list.append(item)
    return data_list
```

$$\text{Precision} = \frac{\text{Number of True Positives}}{\text{Number of Retrieved Contexts}}$$

$$\text{Recall} = \frac{\text{Number of True Positives}}{\text{Number of Relevant Contexts}}$$

```python
def precision_recall(retrieved, ground_truth):
    retrieved_set = set(convert_to_list(retrieved))# Convert lists to sets of strings
    ground_truth_set = set(convert_to_list(ground_truth))
    # Calculate precision and recall
    true_positives = retrieved_set.intersection(ground_truth_set)
    # Context Precision: Measure how accurately the retrieved context matches the user's query
    precision = len(true_positives) / len(retrieved_set) if retrieved_set else 0
    # Context Recall: Evaluate the ability to retrieve all relevant contexts for the user's query
    recall = len(true_positives) / len(ground_truth_set) if ground_truth_set else 0

    return precision, recall
```

Relevance

$$\text{Cosine Similarity} = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

```python
def relevance_score(retrieved_context, user_query):
    # Context Relevance: Assess the relevance of the retrieved context to the user's query
    vectorizer = TfidfVectorizer()
    vectors = vectorizer.fit_transform([retrieved_context, user_query])
    similarity_matrix = cosine_similarity(vectors)
    relevance = similarity_matrix[0, 1]
    return relevance
```

Entity Recall

$$\text{Entity Recall} = \frac{\text{Number of Correctly Recalled Entities}}{\text{Total Number of Entities in Query}}$$

```
def entity_recall(retrieved_context, user_query):
    retrieved_entities = set([ent.text for ent in nlp(retrieved_context).ents])
    query_entities = set([ent.text for ent in nlp(user_query).ents])
    true_positives = len(retrieved_entities & query_entities)
    recall = true_positives / len(query_entities) if len(query_entities) > 0 else 0
    return recall
```

Noise Robustness

$$\text{Noise Robustness} = \text{Relevance Score of Noisy Query}$$

```
def noise_robustness(query, noise_level=0.1):
    query = ' '.join(query)
    noisy_query = ''.join([char if np.random.rand() > noise_level else '' for char in query])
    return noisy_query
```

Faithfulness

$$\text{Faithfulness} = \text{Sigmoid Output of BERT}$$

```
def check_faithfulness(generated_answer, ground_truth):
    # Faithfulness: Measure the accuracy and reliability of the generated answers
    inputs = tokenizer(generated_answer, ground_truth, return_tensors='pt')
    labels = torch.tensor([1]).unsqueeze(0)  # Batch size 1
    outputs = model(**inputs, labels=labels)
    loss, logits = outputs[:2]
    faithfulness = torch.sigmoid(logits).mean().item()  # Get the mean value for faithfulness

    return faithfulness
```

Answer Relevance

$$\text{BLEU Score} = \text{Geometric Mean of Precision for n-grams} \times \text{Brevity Penalty}$$

```
def bleu_score(generated_answer, ground_truth):
    # Answer Relevance: Evaluate the relevance of the generated answers to the user's query using BLEU score
    reference = [ground_truth.split()]
    candidate = generated_answer.split()
    score = sentence_bleu(reference, candidate)
    return score
```

Information Integration

$$\text{ROUGE-1} = \frac{\text{Number of Overlapping Unigrams}}{\text{Total Number of Unigrams in Ground Truth}}$$

$$\text{ROUGE-L} = \frac{\text{Length of Longest Common Subsequence}}{\text{Total Length of Ground Truth}}$$

```python
def rouge_score(generated_answer, ground_truth):
    # Information Integration: Assess the ability to integrate and present information cohesively using ROUGE score
    scorer = rouge_scorer.RougeScorer(['rouge1', 'rougeL'], use_stemmer=True)
    scores = scorer.score(generated_answer, ground_truth)
    return scores['rouge1'].fmeasure, scores['rougeL'].fmeasure
```

Counterfactual Robustness

$$\text{Counterfactual Robustness} = \text{Generated Answer} \neq \text{Counterfactual Answer}$$

```
def counterfactual_robustness(generated_answer, counterfactual_answer):
    # Convert inputs to list
    generated_answer = ' '.join(generated_answer)
    counterfactual_answer = ' '.join(counterfactual_answer)
    # Counterfactual Robustness: Test the robustness of the system against counterfactual or contradictory queries
    return generated_answer != counterfactual_answer
```

This part will be define in Main function (dicussed in later slides)

```
# Calculate counterfactual robustness
counterfactual_query = 'not ' + query
counterfactual_generated_answer = get_response(counterfactual_query)
counterfactual_robustness_score = counterfactual_robustness(generated_answer, counterfactual_generated_answer)
metrics['counterfactual_robustness'].append(counterfactual_robustness_score)
```

Negative Rejection

$$\text{Negative Rejection} = \text{Absence of Negative Keywords in Generated Answer}$$

```
def negative_rejection(generated_answer):
    # Convert input to list
    generated_answer = ' '.join(convert_to_list([generated_answer]))
    # Negative Rejection: Measure the system's ability to reject and handle negative or inappropriate queries
    negative_keywords = ['no', 'not', 'none', 'nothing', 'never', 'out of']
    return any(negative in generated_answer.lower() for negative in negative_keywords)
```

Latency

$$\text{Latency} = \text{End Time} - \text{Start Time}$$

```
def measure_latency(query, rag_pipeline):
    # Latency: Measure the response time of the system from receiving a query to delivering an answer
    start_time = time.time()
    response = rag_pipeline(query)
    end_time = time.time()
    latency = end_time - start_time
    return latency
```

This part will be define in Main function (dicussed in later slides)

```
# Calculate latency
latency = measure_latency(query, get_response)
metrics['latency'].append(latency)
```

Main Function: evaluate_metrics()

```python
def evaluate_metrics(queries, ground_truths, get_response, vector_store):
    metrics = {
        'precision': [],
        'recall': [],
        'relevance': [],
        'entity_recall': [],
        'faithfulness': [],
        'bleu': [],
        'rouge1': [],
        'rougeL': [],
        'latency': [],
        'noise_robustness': [],
        'counterfactual_robustness': [],
        'negative_rejection': []
    }

    for query, ground_truth in zip(queries, ground_truths):
        # Perform retrieval and generation
        retrieved_contexts = [doc.page_content for doc in vector_store.search(query, search_type='similarity')]
        print("\n\n\n\n**********Retrieved Contexts from vector store: ************\n", retrieved_contexts)
        generated_answer = get_response(query)
        print("\n\n\n\n**********Generated Answer: ************\n", generated_answer)

        # Calculate retrieval metrics
        precision, recall = precision_recall(retrieved_contexts, ground_truth['contexts'])
        relevance = relevance_score(' '.join(retrieved_contexts), query)
        entity_recall_score = entity_recall(' '.join(convert_to_list(retrieved_contexts)), query)

        metrics['precision'].append(precision)
        metrics['recall'].append(recall)
        metrics['relevance'].append(relevance)
        metrics['entity_recall'].append(entity_recall_score)
```

```python
        # Calculate noise robustness
        noisy_query = noise_robustness(query)
        noisy_generated_answer = get_response(noisy_query)
        noise_robustness_score = relevance_score(noisy_generated_answer, query)
        metrics['noise_robustness'].append(noise_robustness_score)

        # Calculate generation metrics
        faithfulness = check_faithfulness(generated_answer, ground_truth['answer'])
        bleu = bleu_score(generated_answer, ground_truth['answer'])
        rouge1, rougeL = rouge_score(generated_answer, ground_truth['answer'])

        metrics['faithfulness'].append(faithfulness)
        metrics['bleu'].append(bleu)
        metrics['rouge1'].append(rouge1)
        metrics['rougeL'].append(rougeL)

        # Calculate counterfactual robustness
        counterfactual_query = 'not ' + query
        counterfactual_generated_answer = get_response(counterfactual_query)
        counterfactual_robustness_score = counterfactual_robustness(generated_answer, counterfactual_generated_answer)
        metrics['counterfactual_robustness'].append(counterfactual_robustness_score)

        # Calculate negative rejection
        negative_rejection_score = negative_rejection(generated_answer)
        metrics['negative_rejection'].append(negative_rejection_score)

        # Calculate latency
        latency = measure_latency(query, get_response)
        metrics['latency'].append(latency)
    return metrics
```

Calling evaluate_metrics()

Passing 4 main arguments:

queries

ground_truth

get_response

vector_store

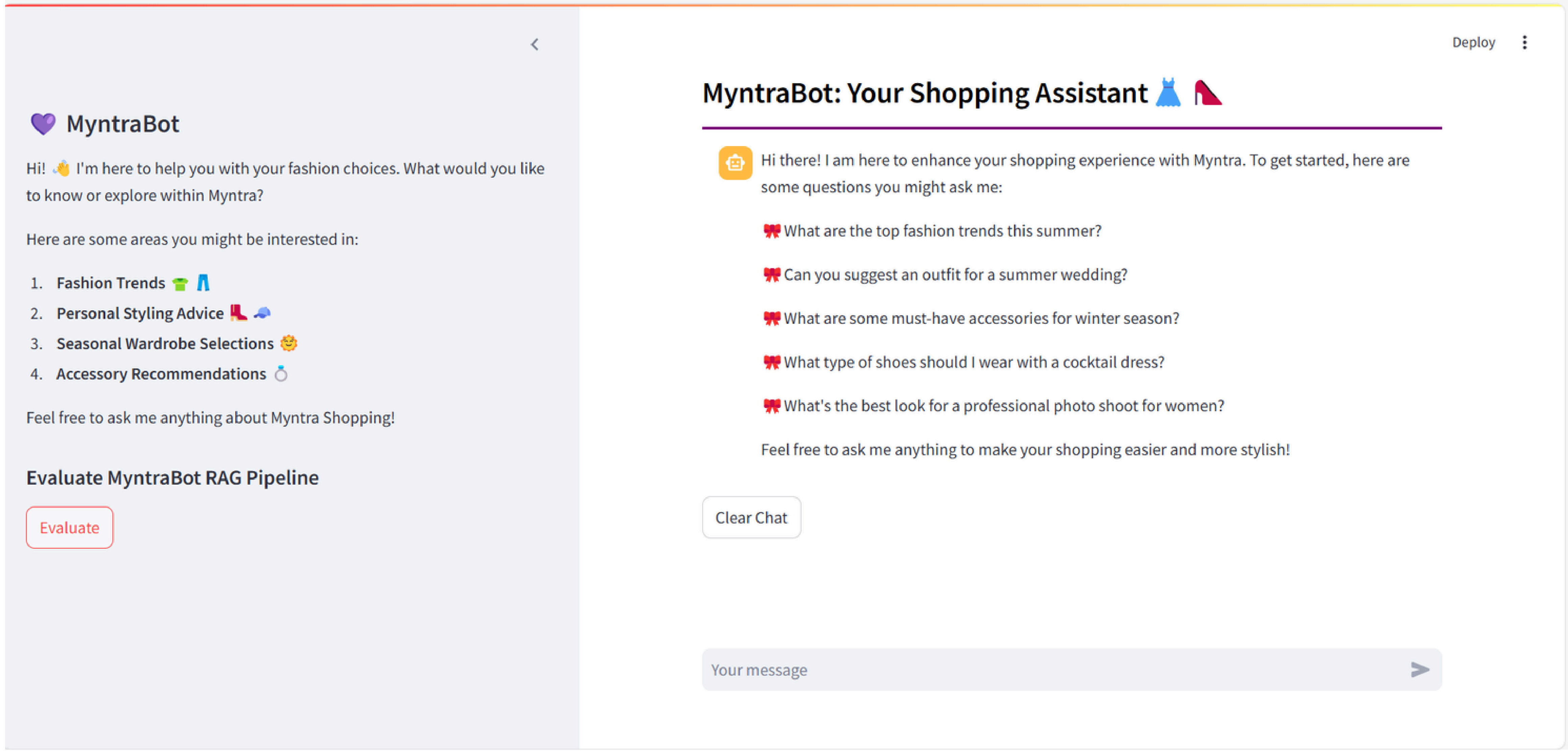
```
# Evaluation section
st.sidebar.header("Evaluate MyntraBot RAG Pipeline")

if st.sidebar.button('Evaluate'):

    queries = ["I want a sweatshirt for boy in yellow color"]
    ground_truths = [
        {
            # 'contexts':[product_data],
            'contexts':["ProductID: 1000894, ProductName: U.S. Polo Assn. Kids Boys Yellow Hooc
            # 'answer': "Bubblegummers Boys Purple Printed Sports Sandals (ProductID: 10001209)
            'answer': '''Sure, I can help with that. Myntra offers a U.S. Polo Assn. Kids Boys
                        ProductID: 1000894
                        ProductName: U.S. Polo Assn. Kids Boys Yellow Hooded Sweatshirt
                        ProductBrand: U.S. Polo Assn. Kids
                        Gender: Boys
                        Price (INR): 899
                        Description: Yellow sweatshirt, has an attached hood with drawstring fa
                        PrimaryColor: Yellow
                        This sweatshirt might be a great fit for what you're looking for.'''
        }
    ]

    metrics = evaluate_metrics(queries, ground_truths, get_response, vector_store)
    st.sidebar.subheader("Evaluation Metrics")
    for metric, values in metrics.items():
        st.sidebar.write(f"{metric}: {np.mean(values)}")
```

Lets Run our code to Evaluate Performance Metrics of RAG Chatbot!



Run and Evaluate

Step 1: To generate vector embeddings, run below command

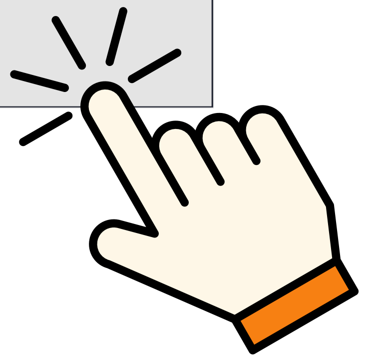
```
python Vector_Embeddings.py
```

Step 2: Run our chatbot using below command

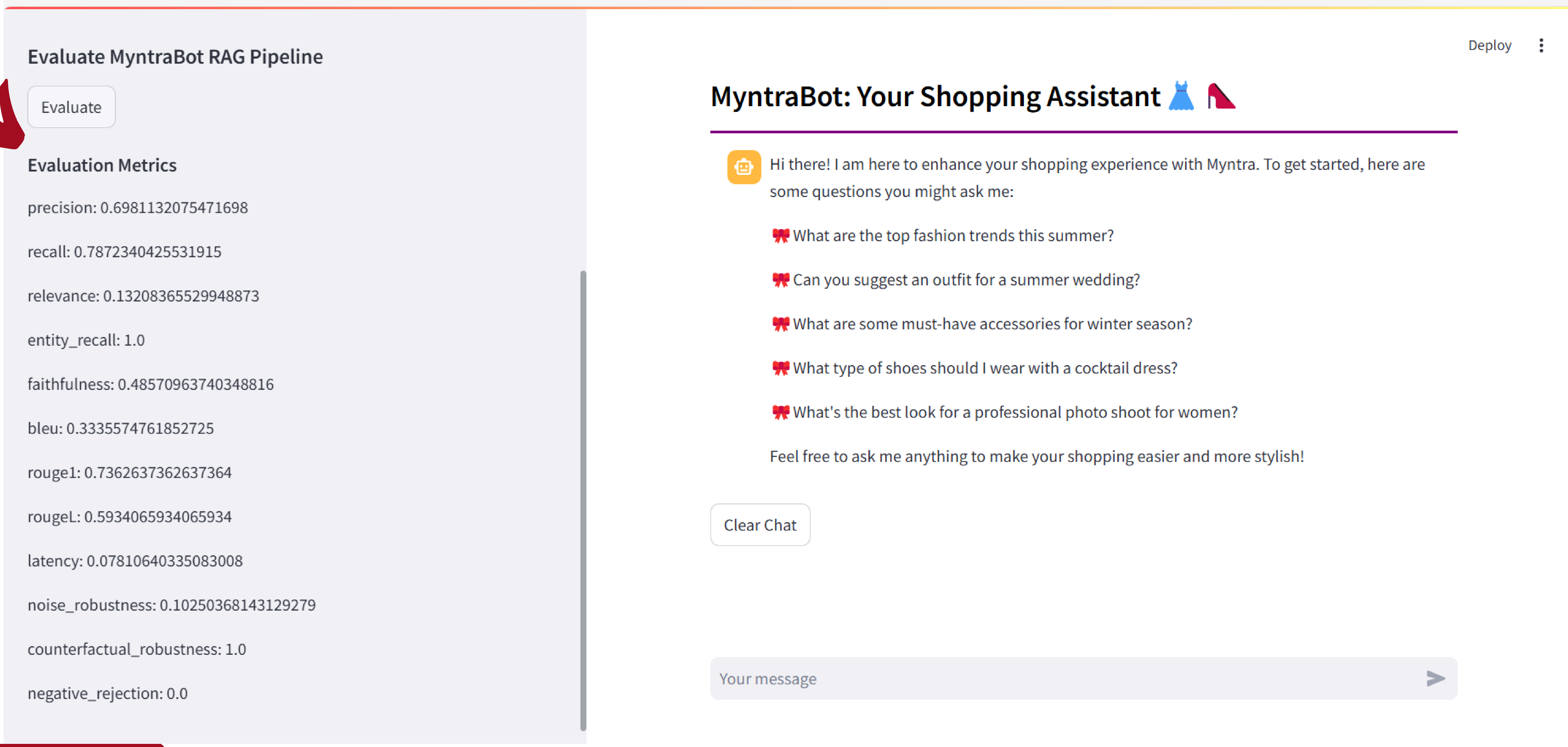
```
streamlit run MyntraBot.py
```

Step 3: Click on evaluate button to execute metrics evaluation of RAG pipeline

Evaluate



Results



Evaluate MyntraBot RAG Pipeline
Evaluate
Evaluation Metrics
precision: 0.6981132075471698
recall: 0.7872340425531915
relevance: 0.13208365529948873
entity_recall: 1.0
faithfulness: 0.48570963740348816
bleu: 0.3335574761852725
rouge1: 0.7362637362637364
rougeL: 0.5934065934065934
latency: 0.07810640335083008
noise_robustness: 0.10250368143129279
counterfactual_robustness: 1.0
negative_rejection: 0.0
Deploy
MyntraBot: Your Shopping Assistant 👗 👠
Hi there! I am here to enhance your shopping experience with Myntra. To get started, here are some questions you might ask me:
🎀 What are the top fashion trends this summer?
🎀 Can you suggest an outfit for a summer wedding?
🎀 What are some must-have accessories for winter season?
🎀 What type of shoes should I wear with a cocktail dress?
🎀 What's the best look for a professional photo shoot for women?
Feel free to ask me anything to make your shopping easier and more stylish!
Clear Chat
Your message

Metrics Improvement strategies

- **Enhanced Vector Representation**
  - Upgraded the embedding model to capture better semantic relationships, improving the quality of retrieved contexts. This enhancement can be expected to increase context relevance, precision, and recall, leading to more accurate and useful responses.

- **Context Filtering**
  - Implement an additional filtering layer post-retrieval using TF-IDF vectorization and cosine similarity to score and select the most relevant contexts.

- **Fine-tuning the LLM**
  - Fine-tuned the language model on a domain-specific dataset to improve the accuracy and reliability of generated answers.

- **Consistency Check**
  - Implement a consistency checking mechanism to compare the generated answer with retrieved contexts and ensure alignment.

Thank you!