Testing RAG Model

Aim:

To generate high-quality question-answer (QA) pairs using DeepSeek and a RAG chatbot, and evaluate their quality using NLP metrics.

Acceptance Criteria:

- The script should successfully extract meaningful paragraphs from Wikipedia history data.
- The DeepSeek API should generate valid QA pairs without errors.
- The RAG chatbot should generate answers based on relevant retrieved content.
- The evaluation script should calculate BLEU, METEOR, ROUGE, BERTScore, context relevance, and faithfulness.
- The results should be saved in structured JSON files for further analysis.

1 - Generating Question-Answer Pairs using DeepSeek

Introduction

This explains the process of generating high-quality question-answer (QA) pairs using the DeepSeek API. These QA pairs serve as ground truth data for evaluating a Retrieval-Augmented Generation (RAG) model.

The script processes Wikipedia history data, extracts meaningful paragraphs, and generates QA pairs using DeepSeek. The output is saved to a JSON file for further evaluation.

Overview of the Code

The script follows these key steps:

Reads historical text data from a JSON file.

Extracts relevant paragraphs from each entry.

Sends the extracted text to the DeepSeek API to generate QA pairs.

Stores the generated QA pairs in a file.

Code Breakdown

API Configuration

```
API_KEY = "your-api-key"
API_URL = "https://api.deepseek.com/v1/chat/completions"
```

The script uses **DeepSeek's chat API** to generate QA pairs.

The `API_KEY` is required for authentication.

The `API_URL` is the endpoint for sending requests.

Preparing API Headers

```
headers = {
    "Content-Type": "application/json",
    "Authorization": f"Bearer {API_KEY}"
}
```

The request must have a **Content-Type** of `application/json`.

The `Authorization` header includes the API key for authentication.

Generating a QA Pair

```
def generate_qa(title, paragraph):
    prompt = f"""You are an AI assistant that generates high-quality question-answer pairs for evaluating a Retrieval-Augmented Generation (RAG) model.

Given the following context:

**Title:** {title}
**Paragraph:** {paragraph}

Generate a well-formed question that can be answered from the paragraph and provide a precise answer.

Format your response as:
{{
  "question": "Generated question?",
  "answer": "The correct answer extracted from the paragraph."
}}
"""
```

This function creates a **structured prompt** containing:

The **title** of the Wikipedia article.

A **paragraph** extracted from the article.

```
    data = {
        "model": "deepseek-chat",
        "messages": [
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": prompt}
        ],
        "temperature": 0.3,
        "max_tokens": 1024
    }
```

System Role: Provides general guidance for DeepSeek.

User Input: The formatted prompt with title and paragraph.

Temperature: Controls randomness (0.3 keeps responses stable).

max_tokens: Limits the response length to 1024 tokens.

Sending Request to DeepSeek API

```
    try:
        response = requests.post(API_URL, headers=headers, json=data)
        response.raise_for_status()  # Will raise an error for HTTP errors
        result = response.json()
```

Sends a **POST request** to DeepSeek's API.

`raise_for_status()` ensures API errors are caught.

The response is converted to JSON format.

Extracting QA Pair from API Response

```
        output_text = result.get('choices', [{}])[0].get('message', {}).get('content', "").strip()
```

Breaking it Down

Code Section	What It Does
`result.get('choices', [{}])`	Gets the list of response choices; defaults to `[{}]` if missing.
`[0]`	Picks the first response.
`.get('message', {})`	Extracts the message object; defaults to `{}` if missing.
`.get('content', "")`	Retrieves the actual response text; defaults to `""` if missing.
`.strip()`	Removes unnecessary spaces.

Example API Response

```json
{
    "choices": [
        {
            "message": {
                "content": "What is the capital of France? Paris."
            }
        }
    ]
}
```

Extracted Output

```python
"What is the capital of France? Paris."
```

Reading Wikipedia Data

Loads **historical data** from a JSON file.

Extracting Paragraphs

```python
        # Select a meaningful paragraph (with more than 50 words)
        paragraphs = [p for p in content.split(". ") if len(p.split()) > 50]
        if not paragraphs:
            continue
```

Splits content into sentences.

Filters paragraphs that have more than **50 words** (to ensure meaningful context).

Saving QA Pairs One by One

```python
        selected_paragraph = random.choice(paragraphs)
        qa_pair = generate_qa(title, selected_paragraph)

        if qa_pair:
            # Append the QA pair to the list
            qa_pairs.append(qa_pair)

            # Write the QA pair to the output file
            outfile.write(json.dumps(qa_pair, ensure_ascii=False) + "\n")

        # Sleep a random interval to avoid hitting rate limits
        time.sleep(random.uniform(1, 3))
```

Writes QA pairs directly to the file to avoid data loss in case of errors.

Sleep time prevents API rate limit errors.

Final Output

The script generates a JSON file with entries like:

```json
[
    {
        "question": "What are the steps required for a land sale to be recognized by the state according to the Arthashastra?",
        "answer": "The price must be announced in front of witnesses, recorded, and taxes paid for the buy-sale arrangement to be deemed recognized by the state.
    },
    {
        "question": "What modern technological developments have influenced the study of Soanian tools?",
        "answer": "Modern technological developments have led to a focus on Soanian tools' occurrence as a complex behavioural system, involving careful surveyin
    },
    {
        "question": "Which key areas in Mewar were subjugated by Shahbaz Khan Kamboh under Mughal rule?",
        "answer": "Kumbhalgarh, Mandalgarh, Gogunda, and Central Mewar were subjugated by Shahbaz Khan Kamboh under Mughal rule."
    },
```

2 - Generating Question-Answer Pairs using RAG ChatBot

Execution Flow

1. Initialize the Gemini API client with the provided API key.
2. Load the BERT tokenizer and model for embedding generation.
3. Load the FAISS index and document metadata.
4. For each user query:
   - Generate an embedding.
   - Search the FAISS index for similar documents.
   - Prepare a prompt with the retrieved content.
   - Use the Gemini API to generate and return a response.
5. For batch processing:
   - Read questions from the input file.
   - Generate answers using the response generation method.
   - Save each question-answer pair to the output file, adhering to rate limits by introducing delays between API calls.

3 - Evaluation

This evaluates the quality of answers generated by a Retrieval-Augmented Generation (RAG) model. It compares the generated answers with ground truth answers using multiple NLP evaluation metrics.

Dependencies

The script requires the following Python libraries:

- `json` (for handling JSON files)
- `tqdm` (for progress tracking)
- `evaluate` (for loading NLP metrics)
- `transformers` (for NLI model inference)
- `statistics` (for calculating averages)
- `bert_score` (for BERT-based evaluation)

Evaluation Metrics

The script computes the following metrics:

1. **BLEU** – Measures word overlap between generated and reference answers.
2. **METEOR** – Evaluates meaning similarity with stemming and synonyms.
3. **ROUGE** – Measures unigram, bigram, and longest common subsequence overlap.
4. **BERTScore** – Uses contextual embeddings to compare answers.
5. **Context Relevance** – Checks if the generated answer is relevant to the ground truth.
6. **Faithfulness** – Evaluates whether the generated answer stays true to the reference.

How the Script Works

1. **Load Required Models and Metrics:**
   - `bleu`, `meteor`, `rouge`, and `bertscore` metrics are loaded.
   - A **Natural Language Inference (NLI) model** is initialized to compute **context relevance** and **faithfulness**.

```python
import json
from tqdm import tqdm
from evaluate import load
from transformers import pipeline
from statistics import mean

# Loading evaluation metrics
bleu = load("bleu")
meteor = load("meteor")
rouge = load("rouge")
```

2. **Define `compute_nli_score` Function:**
   - This function uses the NLI model to check how much the generated answer aligns with the reference answer.

```python

# Initializing NLI model for context relevance and faithfulness evaluation
nli_model = pipeline("text-classification", model="cross-encoder/nli-deberta-v3-base")

def compute_nli_score(reference, hypothesis):
    """
    Compute NLI-based score (used for both context relevance and faithfulness).
    Returns a score between 0 and 1 indicating relevance.
    """
    result = nli_model(f"{reference} [SEP] {hypothesis}")
    score = result[0]['score'] if result[0]['label'] == 'ENTAILMENT' else 1 - result[0]['score']
    return score
```

3. **Load Data:**
   - `generated_ques_ans.json` (Ground truth QA pairs)
   - `RAG_generated_qa_pairs.json` (Generated QA pairs)
   - The script ensures that both files have the same number of questions.

```python
# Load ground truth and RAG-generated QA pairs
with open("generated_ques_ans.json", "r", encoding="utf-8") as f:
    ground_truth_data = json.load(f)

with open("RAG_generated_qa_pairs.json", "r", encoding="utf-8") as f:
    rag_data = json.load(f)

assert len(ground_truth_data) == len(rag_data), "Mismatch in number of QA pairs between the two files."
```

4. **Evaluate Each QA Pair:**
   - For each question:
     - Compute BLEU, METEOR, ROUGE, and BERTScore.
     - Compute **Context Relevance** (how well the generated answer matches the reference).
     - Compute **Faithfulness** (how truthful the generated answer is compared to the reference).

```python
evaluation_results = []

# Lists to store individual metric scores
bleu_scores = []
meteor_scores = []
rouge1_scores = []
rouge2_scores = []
rougeL_scores = []
bertscore_f1_scores = []
context_relevance_scores = []
faithfulness_scores = []

for gt_pair, rag_pair in tqdm(zip(ground_truth_data, rag_data), total=len(ground_truth_data), desc="Evaluating QA Pairs"):
    question = gt_pair["question"]
    ground_truth_answer = gt_pair["answer"]
    generated_answer = rag_pair["answer"]

    # Compute BLEU score
    bleu_score = bleu.compute(predictions=[generated_answer], references=[ground_truth_answer])['bleu']
    bleu_scores.append(bleu_score)

    # Compute METEOR score
    meteor_score = meteor.compute(predictions=[generated_answer], references=[ground_truth_answer])['meteor']
    meteor_scores.append(meteor_score)

    # Compute ROUGE scores
    rouge_scores = rouge.compute(predictions=[generated_answer], references=[ground_truth_answer])
    rouge1_scores.append(rouge_scores['rouge1'])
    rouge2_scores.append(rouge_scores['rouge2'])
    rougeL_scores.append(rouge_scores['rougeL'])

    # Compute BERTScore F1
    bertscore_result = bertscore.compute(predictions=[generated_answer], references=[ground_truth_answer], lang="en")
    bertscore_f1 = bertscore_result['f1'][0]  # Extract F1 score
    bertscore_f1_scores.append(bertscore_f1)

    # Compute Context Relevance
    context_relevance_score = compute_nli_score(ground_truth_answer, generated_answer)
    context_relevance_scores.append(context_relevance_score)

    # Compute Faithfulness
    faithfulness_score = compute_nli_score(generated_answer, ground_truth_answer)  # Reverse order for faithfulness
    faithfulness_scores.append(faithfulness_score)

    # Append results
    evaluation_results.append({
        "Question": question,
        "Ground Truth": ground_truth_answer,
        "Generated Answer": generated_answer,
        "Metrics": {
            "BLEU Score": bleu_score,
            "METEOR Score": meteor_score,
            "ROUGE-1": rouge_scores['rouge1'],
            "ROUGE-2": rouge_scores['rouge2'],
            "ROUGE-L": rouge_scores['rougeL'],
            "BERTScore F1": bertscore_f1,
            "Context Relevance": context_relevance_score,
            "Faithfulness": faithfulness_score
        }
    })
```

5. **Store Results:**
    - The evaluation results are saved in `evaluation_results.json`.
    - The script calculates average scores for all metrics and displays them.

Output

- A JSON file (`evaluation_results.json`) containing individual scores for each question-answer pair.
- The script prints **average** evaluation scores at the end.

```
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Evaluating QA Pairs: 100%|██████████████████████████████| 1000/1000 [15:31<00:00,  1.07it
✅ Evaluation completed! Results saved to 'evaluation_results.json'.

Average Scores:
BLEU Score: 0.2531
METEOR Score: 0.5230
ROUGE-1: 0.5028
ROUGE-2: 0.3921
ROUGE-L: 0.4561
BERTScore F1: 0.8980
Context Relevance: 0.0295
Faithfulness: 0.0296
shtlp_0165@SHTLP0165:~/Desktop/Project-RAG-LLM$ 
```

Outcome:

- A dataset of QA pairs generated by DeepSeek and the RAG chatbot.
- An evaluation report comparing generated answers with ground truth.
- Insights into the performance of the RAG model using multiple NLP metrics.