Title: Question-Answering System

Background: Imagine you are building a question-answering (QA) system that leverages SentenceTransformers models for generating embeddings, ElasticSearch as a vector store, Docker for containerization and deployment, and Flask for the API. Your goal is to build a basic question-answering (QA) machine learning (ML) system that can take a user's question, search for relevant passages in a given corpus, and return them as answers. See this as a reference example of a query and answers.

There are two additional **optional** tasks:
(1) Use Generative AI (e.g., Falcon, LLaMa etc.) to additionally provide **one** direct, concise, and accurate answer to each question based on the relevant passages retrieved.
(2) Use Streamlit to create a UI to interact with the ML system.

Challenge Overview:
Task 0: Directory Structure

```
question_answering/
│
├── app/
│   ├── app.py              # Flask API implementation
│   ├── retrieval.py        # Passage retrieval logic
│   ├── model.py            # SentenceTransformers utilities
│   ├── parsing.py          # Document parsing logic
│   ├── indexing.py         # ElasticSearch indexing logic
│   ├── gui.py              # GUI interface implementation (OPTIONAL)
│   └── gen_ai.py           # Generative AI utilities (OPTIONAL)
│
├── docker/
│   ├── Dockerfile              # Docker configuration
│   └── docker-compose.yml      # Docker Compose configuration
│
├── docs/
```

```
│   ├── passage_metadata.csv              # Passages and metadata
│   ├── passage_metadata_emb.csv          # Passages, metadata, and embeddings
│   ├── question_answers.csv              #Questions and answers
│   ├── questions_answers_gen.csv  #Questions, answers, generative AI answer
│   ├── evaluation.csv                    #Evaluation data
│   ├── performance.csv          # Evaluation results
│   ├── technical.pdf            # Technical documentation
│
└── README.md                 # Description of implementation and setup
```

Task 1: Parsing

You are provided with a small dataset of legal files here containing (1) "Example Question and Answers" Google sheet which contains an example sheet with a query and answers, (2) "user_queries.txt" which contains user queries (having a list of 10 legal questions), and (2) "Corpus", a folder which contains files of case judgments and their corresponding metadata (each pair has the same filename with the only difference being the endings; *_Technical.txt* for judgments and *_Metadata.json* for metadata).

Your task is to write code to parse and extract the content of the judgment and metadata:

- Open one "*_Technical.txt" file.* Inspect to understand the structure of the contents
- Extract the passages within **child** *__paragraph__* markers under **parent** *__section__* markers
- Combine all the extracted passages into 1 passage, then, split them into chunks of 5-sentences to form a **passage** each
- For each judgment file, extract all its metadata from the accompanying metadata file.
- Now create pairings of each extracted passage and the corresponding metadata of the judgments for all the extracted passages. Output as a CSV with name ***passage_metadata.csv*** having columns *Passage* and *Metadata***.** This compilation of data will be used for indexing purposes.

Task 2: Passage Embeddings Generation

- Write code to use a SentenceTransformers model to create high-quality embeddings for the extracted passages for indexing. Choose a suitable model.
- Save the embeddings along with the corresponding passage and metadata as a CSV with the name ***passage_metadata_emb.csv*** with columns *Passage, Metadata,* and *Embedding.*

Task 3: ElasticSearch Integration

- Write code to set up an ElasticSearch index using this library to store the preprocessed and embedded data.
    - You need to design the index structure to efficiently store and retrieve the embeddings of the passages.
- Write code to index each row of the passage, metadata, and embedding data in your ***passage_metadata_emb.csv*** file into ElasticSearch.

Task 4: Document Retrieval

- Write code that implements a retrieval mechanism using ElasticSearch's search capabilities and the precomputed embeddings to identify relevant passages.
    - It should return a list of **3** passages (along with their metadata) that are most relevant to the question along with relevance scores (such as cosine similarity between embeddings).
- Save output as ***questions_answers.csv*** containing columns *Question, Passage 1, Relevance Score 1, Passage 1 Metadata, Passage 2, Relevance Score 2, Passage 2 Metadata, Passage 3, Relevance Score 3, Passage 3 Metadata*

Task 5: Containerization

- Create a Docker container for your application, including all necessary dependencies. The container should be easy to deploy, and the application should interact with the ElasticSearch instance for data retrieval.

Task 6: API Deployment with Flask

- Create a Flask application that exposes an API with an endpoint for:

    - receiving user questions
    - uploading documents to be indexed
- Implement features like logging, request/response validation, and handling concurrent requests.
- Create a configuration for different deployment environments (development, testing, production).

Task 7: Evaluation

Implement an evaluation of the performance of your QA system by using the 10 queries in our provided file: *user_queries.txt*, manually rating the top 3 answers, and automatically computing the accuracies of your system.

- Write code that generates an output file for the queries and the corresponding retrieved passages and metadata (n=3) in an ***evaluation.csv*** file that has the following columns: *Question, Passage 1, Relevance Score 1, Passage 1 Metadata, Is Passage 1 Relevant? (Yes/No), Passage 2, Relevance Score 2, Passage 2 Metadata, Is Passage 2 Relevant? (Yes/No), Passage 3, Relevance Score 3, Passage 3 Metadata, Is Passage 3 Relevant? (Yes/No)*
- Manually rate each of the 3 passages for all 10 queries by filling in the columns "*Is Passage # Relevant?"* with Yes/No. Rate as "Yes" if the passage contains any useful info that directly or could be used to answer the corresponding question. Save this new file containing the ratings as ***evaluation_rated.csv***
    - **NB:** We recognize that you do not have any legal background so perform the assessment using your lay knowledge. Don't stress about it.
- Write code that takes the ***evaluation_rated.csv*** and computes the following two metrics: top-1 and top-3 accuracy as follows:
    - $top\ 1\ accuracy\ =\ \frac{number\ of\ relevant\ passages}{1}\ x\ 100$
    - $top\ 3\ accuracy\ =\ \frac{number\ of\ relevant\ passages}{3}\ x\ 100$
- Save the two accuracy results computed for the 10 queries in a CSV file ***performance.csv*** that contains as columns *Top 1 Accuracy* and *Top 3 Accuracy*

Task 8: Demo Video

- Record a short demo video of your QA system working (no longer than 1 minute) that shows writing a query and getting the top 5 relevant passages along with their corresponding metadata and relevance scores. Save as **demo.mp4**

Bonus (Optional):

1) Direct Answers with Generative AI:

- Write code that uses an open-source, state-of-the-art generative LLM (e.g., LLaMa, Falcon, etc.) to additionally provide a quick, direct answer to the question based on the *relevant* passages among the retrieved ones.
    - **N.B:** Create a high-quality prompt and ensure that your prompt and parameters generate a near-deterministic response for the sake of reproducibility.
- Save output as ***questions_answers_gen.csv*** containing columns *Question, Passage 1, Relevance Score 1, Passage 1 Metadata, Passage 2, Relevance Score 2, Passage 2 Metadata, Passage 3, Relevance Score 3, Passage 3 Metadata, Generative AI Answer*

2) FrontEnd with Streamlit:

- Develop a basic front end using Streamlit to create a user-friendly interface for interacting with your question-answering system.
- The front end should provide input to upload documents for indexing and also text input for users to enter their questions, and display the relevant passages (with their relevance scores) and the generative AI answer (optional).

Challenge Requirements:

1. Use Python (minimum version 3.6) for all tasks.
2. Provide clear instructions on how to run your code.
3. Design the system with efficiency and scalability in mind.
4. Demonstrate error handling and proper API response formats.
5. Ensure that the Docker container runs the Flask application successfully.

6. Develop the system iteratively using best software development practices. Include all the Git commits you have made throughout the development. (i.e. commit as you're developing instead of pushing everything at the end).

Evaluation Criteria:

1. **Code Quality:** Is the code well-structured, well-commented, readable, and follows best practices?
2. **Functionality:** Does the system perform the question-answering tasks accurately?
3. **Efficiency:** Is the system designed for efficient data retrieval and processing?
4. **Error Handling:** Does the system handle errors gracefully and provide meaningful error messages?
5. **Scalability:** Is the system designed to handle larger datasets and increased user load?

Submission

Submit to this form (1) your demo video and (2) a link to a Git repository containing all your code, output files, a README with clear instructions on running the system (including all docker commands needed to set it up and use), and a technical PDF document that contains a brief explanation of the design choices made for each task. Submission deadline is **Saturday, 30th September, 11:59 PM GMT.**