Compu. J
FROM CLOUD TO GROUND

Take-home Technical Task: Smart Document Classifier Web Application

We'd like you to build a simple web application that allows users to upload documents, classify them into predefined categories using a pre-trained machine learning (ML) model, and view the classification results. This task is designed to assess your skills in API design, frontend development, data handling, and integration with ML models.

Core Functionality

- **Document Upload:** Users should be able to upload documents through a web interface. The application needs to extract the text content from these documents. Plain text files are sufficient for simplicity, but handling PDFs or other common document formats would be a bonus (*PyPDF2, python-docx*).
- **Document Classification:** The application should use a pre-trained machine learning model to perform zero-shot classification of the text content in the uploaded documents. We recommend exploring models from the Hugging Face Transformers library, such as **facebook/bart-large-mnli** being a reliable starting point, though you are strongly encouraged to explore other newer models to achieve faster and more accurate classification. If you're more comfortable with traditional methods and also lack the compute for transformers on your workstation, you could also consider a simpler approach using TF-IDF and Logistic Regression or similar as a fallback solution. The model should classify documents into these categories:
  - **Technical Documentation**
  - **Business Proposal**
  - **Legal Document**
  - **Academic Paper**
  - **General Article**
  - **Other**

  The application must display the predicted category and a confidence score for the classification.
- **Results Display and User Interface:** Create a clean and user-friendly web UI to:
  - Allow users to easily upload documents (drag-and drop would be great!)
  - Display the classification results clearly for each uploaded document, including the top predicted category, and confidence scores for other categories.
  - Present a list view of all previously uploaded and classified documents, showing key information like filename, predicted category, confidence, and upload time.
- **Data Storage:** Uploaded documents (or at least their essential metadata like filename, content, classification, confidence, and upload timestamp) should be stored in a database. You can use a simple database like SQLite or PostgreSQL. The document list view in the UI should retrieve data from this database.
- **API Design:** Design a clean and well-documented RESTful API for document upload and retrieval of classification results. Think about clear endpoints and appropriate data structures.
- **Basic Statistics (Bonus):** Consider adding a simple statistics dashboard to display insights like document distribution by type, upload trends over time, or confidence score distributions. Creativity is always encouraged!

Technical Requirements

For this task our recommended stack is as follows:

Backend:

- FastAPI or any alternative Python web framework
- SQLite/PostgreSQL with SQLAlchemy for ORM
- Pydantic for data validation
- Python 3.8+

Frontend:

- Any modern framework (React, Vue, Svelete, Angular, etc). Regular old HTML/CSS/JS is also not a problem at all; whatever works best for you while allowing you to fully demonstrate your design abilities!
- Communicate with backend exclusively via REST API

We would thoroughly recommend setting up a virtual environment, or use a docker container if you would prefer.

Differentiators: Handling the ML Pipeline and Error Cases

To really impress us, pay close attention to how you handle the ML pipeline and various error cases. This is a key differentiator for candidates. Consider these points:

- **ML Pipeline Robustness:** Think about how your application will handle long documents (potential need for chunking), situations, where the model has low confidence, or scenarios where no category seems to be a good fit.
- **Error Handling:** Implement robust error handling throughout your application, from document processing (invalid filetypes, corrupt files) and system issues (API failures, database errors). Provide informative error messages in both the API responses and the UI.
- **Model Choice Justification:** In your `README.md`, explain why you chose your specific ML model, and the alternatives you considered. Discuss the trade-offs you considered (e.g. speed vs accuracy). You will not be strictly evaluated on model quality, so please choose the best model that suits your system.

Evaluation Criteria

- **Functionality:** Does the application meet the core requirements?
- **Code Quality:** Is the code clean, well-structured, maintainable, and well-commented? Is error handling robust?
- **Backend Implementation:** Is the API well designed, RESTful, and documented? Is data validation implemented? Are error responses meaningful?
- **Going the Extra Mile:** Consideration of caching or batch processing, advanced UI features, demonstration of unit tests, and a well-written `README.md` documenting your architecture and setup will be viewed positively.

Deliverables:

- A link to a repository (e.g. GitHub) containing your code.
- A `README.md` file in the repository with clear setup and run instructions, a detailed overview of your architectural decisions, and any notes on potential future improvements or challenges you encountered.
- A working demo of your application is optional but appreciated.

Time Estimate: We anticipate this task will take approximately 4-8 hours to complete. Please focus on quality and demonstrating your core skills, alongside showing your problem-solving abilities and aptitude for learning. It's far more valuable to demonstrate a solid, well-structured implementation of the core features in your preferred stack, than to rush ahead and try to build bonus features.

We are excited to see your approach to this task! Good luck!