
Smart App Backend Explanation

In this project, I developed a backend system for a 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. Below is an overview of the architecture, design decisions, and implementation details, as well as the rationale behind the choices made throughout the process.

Backend Architecture

The backend is built using **FastAPI**, a modern Python web framework known for its speed and ease of use. It is ideal for building RESTful APIs, and its asynchronous support ensures scalability when handling large documents or multiple requests. The application is designed to handle document uploads, text extraction, classification, and storage of results in a **PostgreSQL** database.

Key Components:

1. **Database Layer**:
   - **PostgreSQL**: The choice of PostgreSQL is due to its reliability and scalability. It stores metadata for each uploaded document, including the filename, content, predicted category, confidence scores, and upload timestamp.
   - **SQLAlchemy ORM**: To interact with the database, SQLAlchemy is used with asynchronous support (asyncpg driver and AsyncSession), ensuring that database operations are non-blocking and scalable.
   - **Database Schema**: The documents table stores document metadata with fields for id, filename, content, predicted_category, confidence_scores, and upload_time.
2. **API Layer**:
   - **RESTful API**: The backend exposes two main endpoints:
     - POST /upload/: Handles document uploads, text extraction, and classification.
     - GET /documents/: Retrieves a list of all previously uploaded documents with their classification results.
   - **Data Validation**: Pydantic models (DocumentCreate and DocumentResponse) are used to ensure correct data formats and types for incoming and outgoing data.
3. **Machine Learning Pipeline**:
   - **Text Preprocessing**: Uploaded documents are preprocessed by removing special characters, stopwords, and unnecessary spaces. Lemmatization is applied to reduce words to their base forms.
   - **Chunking**: Long documents are split into smaller chunks (by sentences) to avoid overwhelming the model and to improve classification accuracy.

   - **Keyword Extraction**: TF-IDF is used to extract keywords from the text to provide additional context for classification.
   - **Zero-Shot Classification**: The facebook/bart-large-mnli model is employed for zero-shot classification. This model predicts the most likely category and provides confidence scores for each potential category.

4. **Error Handling**:
   - **Custom Exceptions**: The backend handles various error scenarios, such as invalid file types, low-confidence predictions, database errors, and model inference failures.
   - **Informative Error Messages**: Users are presented with clear and actionable error messages, both in the API responses and logs.
5. **Logging and Monitoring**:
   - **Logging**: Critical operations, including file uploads, database interactions, and model inferences, are logged for debugging and monitoring purposes.
   - **Request/Response Logging**: Middleware is implemented to log incoming requests and outgoing responses, providing visibility into API usage.

Why I Chose the facebook/bart-large-mnli Model

The choice of the **facebook/bart-large-mnli** model for zero-shot classification was driven by several key factors:

1. **Accuracy**:
   - It provided the highest accuracy compared to other models, such as MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli, google/flan-t5-base, and microsoft/deberta-v3-large-mnl. It effectively handled a variety of document types, producing reliable and consistent predictions.
2. **Speed**:
   - The model struck a good balance between inference speed and accuracy. It was faster than some alternatives like DeBERTa and T5, making it more suitable for real-time classification in a web application.
3. **Zero-Shot Classification**:
   - This model's zero-shot capabilities meant it could classify documents into predefined categories, even those it had never seen during training. This fit the project requirements perfectly, as the document categories were predefined and no fine-tuning was required.
4. **Ease of Integration**:
   - The model is available through the Hugging Face Transformers library, providing a simple and consistent API for integration, which made the implementation seamless.
5. **Confidence Scores**:

    - The model returns confidence scores for each category, which allows for the graceful handling of low-confidence predictions (e.g., classifying documents into an "Other" category).

ML Pipeline and Error Handling

Machine Learning Pipeline:

The document classification follows a structured pipeline:

- **Text Preprocessing**: Raw text is cleaned and preprocessed (removing stop words, special characters, lemmatization).
- **Text Chunking**: Long documents are broken into smaller chunks to optimize model performance.
- **Keyword Extraction**: TF-IDF is used to extract key terms from the document to provide additional context.
- **Zero-Shot Classification**: The processed text and extracted keywords are fed into the **BART model** to predict the document's category.
- **Confidence Filtering**: If the model's top prediction has a confidence score below 0.3, the document is classified as "Other," and a warning is logged.

Error Handling:

- **Invalid File Types**: Only .txt files are accepted. The backend rejects unsupported file types with an appropriate error message.
- **Corrupt or Empty Files**: Uploaded files are checked for validity, and users are notified of any issues.
- **Database Errors**: SQLAlchemy's error handling mechanisms are used to catch database-related issues.
- **Model Inference Errors**: Any issues during model inference (e.g., loading failure, inference errors) are logged, and users receive meaningful feedback.

Trade-offs and Alternatives

While facebook/bart-large-mnli provided the best results for this project, I also considered the following alternatives:

1. **Traditional ML Models (TF-IDF + Logistic Regression)**:
    - **Pros**: Faster inference and lower computational requirements.
    - **Cons**: Requires labeled training data and may not generalize well to unseen categories.
2. **Other Transformer Models**:
    - **Pros**: Some models (e.g., DeBERTa) offered slightly better accuracy in certain cases.

    - **Cons**: Slower inference times and higher memory usage made them less suitable for real-time applications.

Ultimately, the choice of facebook/bart-large-mnli was driven by its balance of accuracy, speed, and ease of use.

Frontend Integration

The frontend (React) interacts with the backend through a RESTful API. The frontend uses Axios for making HTTP requests, while React-Dropzone is used for handling document uploads. This setup ensures the user can easily upload a .txt file, view the classification result, and see a list of previously uploaded documents with their metadata.

The frontend communicates with the backend through these endpoints:

- POST /upload/: For uploading the document and retrieving classification results.
- GET /documents/: For retrieving a list of all previously uploaded documents with their metadata (filename, predicted category, confidence scores, upload time).

Database Design

The **PostgreSQL** database stores metadata for each uploaded document in a documents table with the following fields:

- **id**: A unique identifier for each document.
- **filename**: The name of the uploaded file.
- **content**: The content of the document.
- **predicted_category**: The category predicted by the ML model.
- **confidence_scores**: A JSON field that stores confidence scores for each category.
- **upload_time**: The timestamp when the document was uploaded.

This schema allows easy retrieval and display of documents in the frontend UI.

Conclusion

The backend of the Smart App project is designed to be robust, scalable, and user-friendly. By leveraging FastAPI, PostgreSQL, and the facebook/bart-large-mnli model, I created a system that efficiently handles document uploads, text classification, and result storage. The implementation demonstrates strong skills in API design, data handling, and ML integration, while also addressing key challenges such as error handling and pipeline robustness. This combination of tools and techniques allows for a powerful and flexible document classification system that meets the requirements of the task while providing a seamless user experience.