**Title: AI-Powered Document Classification Web Application**

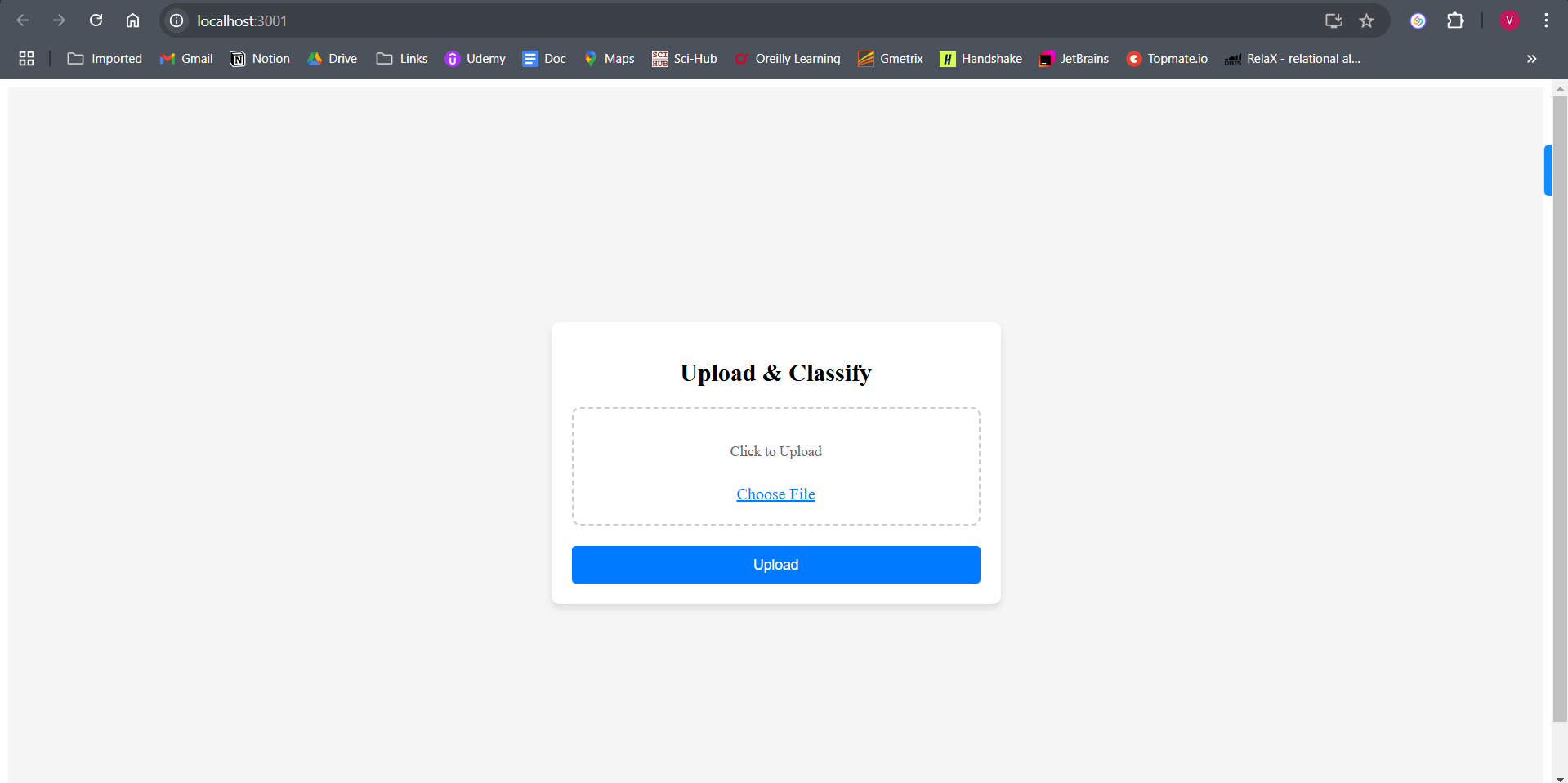
This project is a web-based document classification system that enables users to upload documents and categorize them into predefined categories using a fine-tuned machine learning model. The system consists of a FastAPI backend, a React frontend, and integrates Hugging Face’s BART-large-MNLI model for classification. PostgreSQL is used for storing document metadata and classification results. The model has been fine-tuned for improved classification accuracy, and the system provides a user-friendly interface for document management.

* Frontend: Built using React.js, providing an interface for document upload and classification result visualization.
* Backend: Built with FastAPI, handling document processing, classification, and API requests.
* Database: PostgreSQL stores document metadata, classification results, and timestamps.

Manually classifying documents is a tedious and error-prone task. This project aims to automate document classification by leveraging NLP techniques such as zero-shot classification and fine-tuned transformers. The application supports document upload in multiple formats (TXT, DOCX, PDF) and classifies them into the following categories:

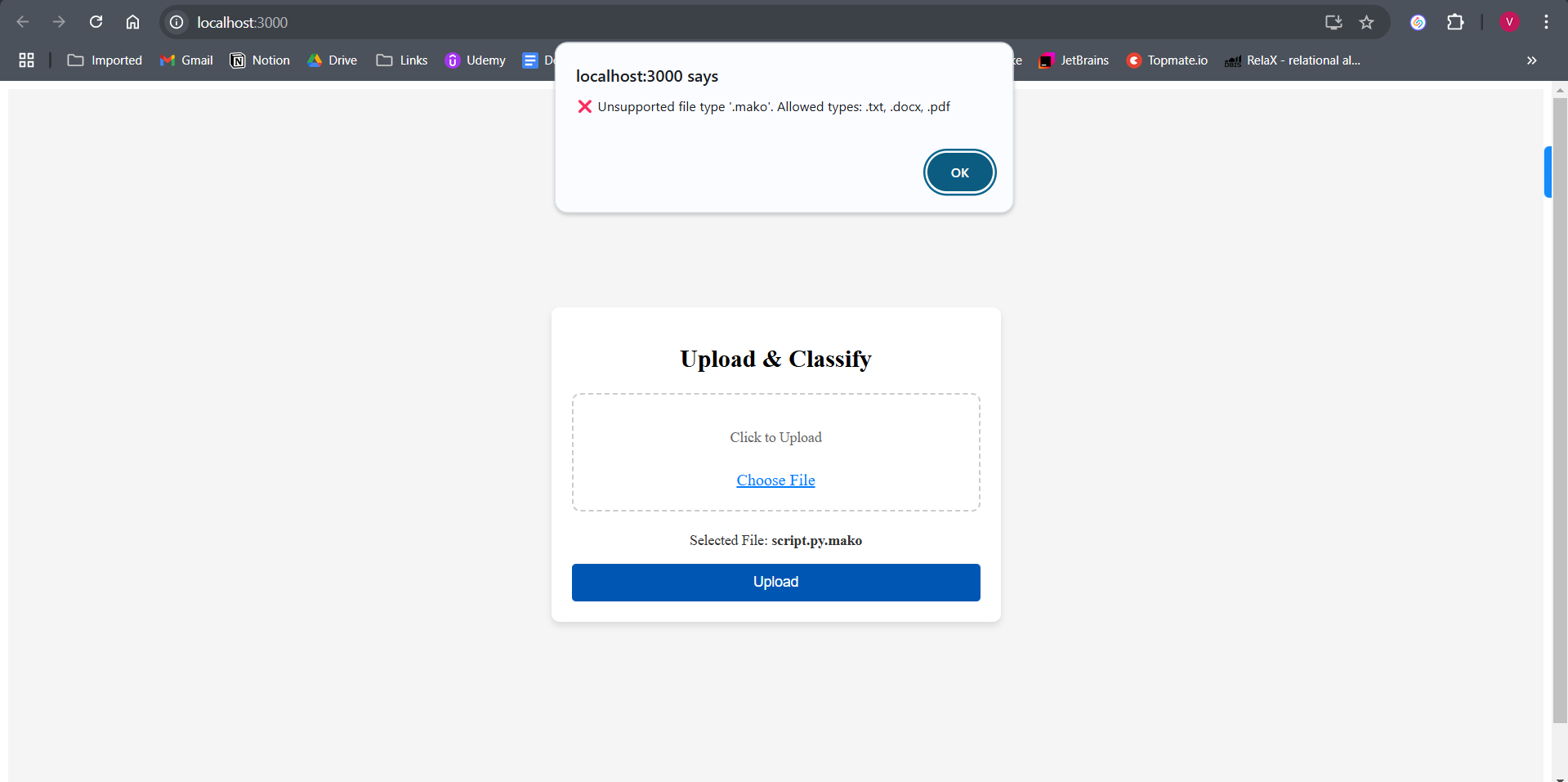
* Technical Documentation
* Business Proposal
* Legal Document
* Academic Paper
* General Article
* Other

The system provides an intuitive user interface as shown in the screenshot below. We just have to choose a file and click the Upload button for the backend mechanism

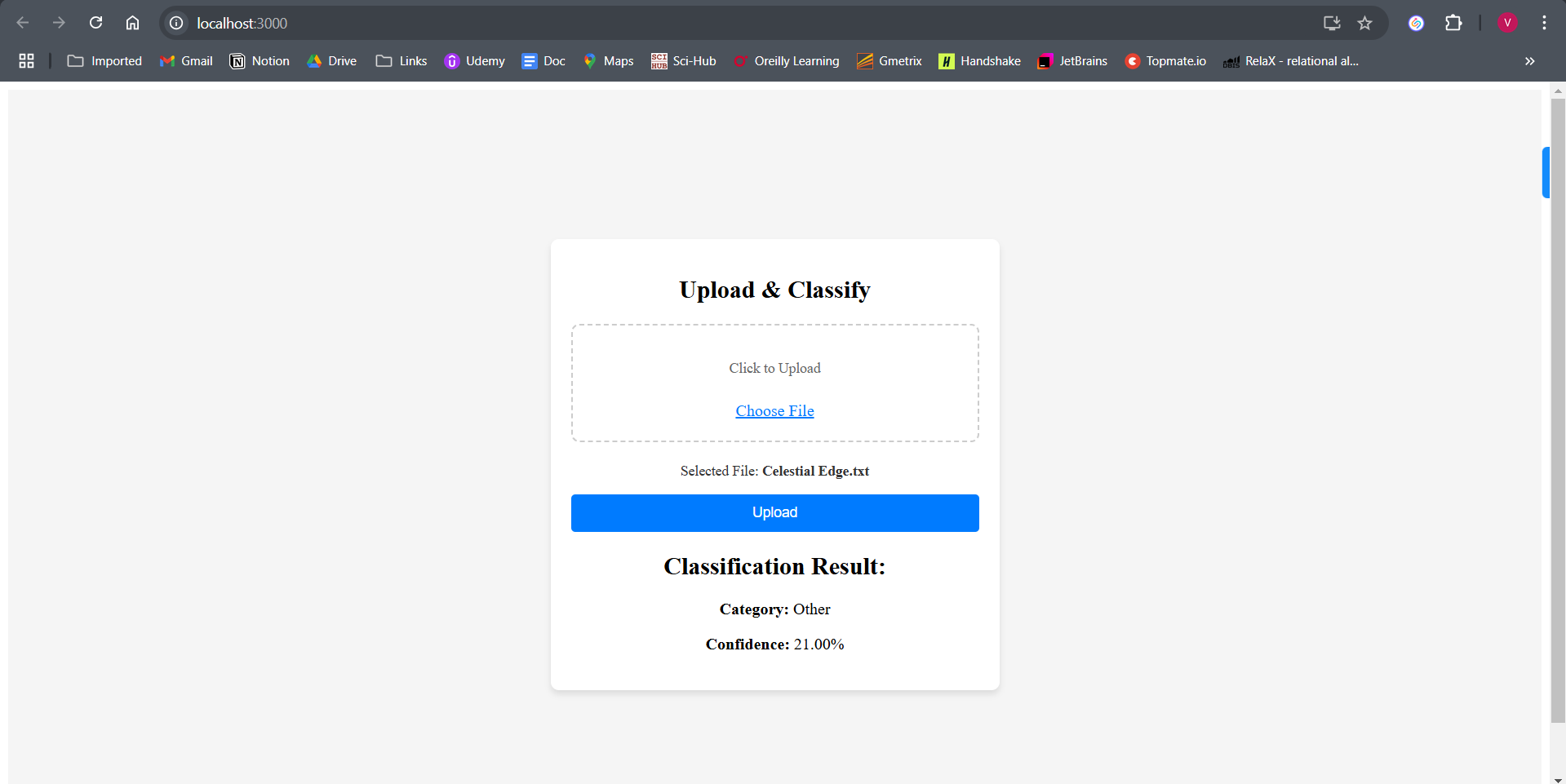


When the user can upload any document and it is checked for the file type. If the file type is not (txt, docx, pdf) then the error handling of these types is done for user-friendliness

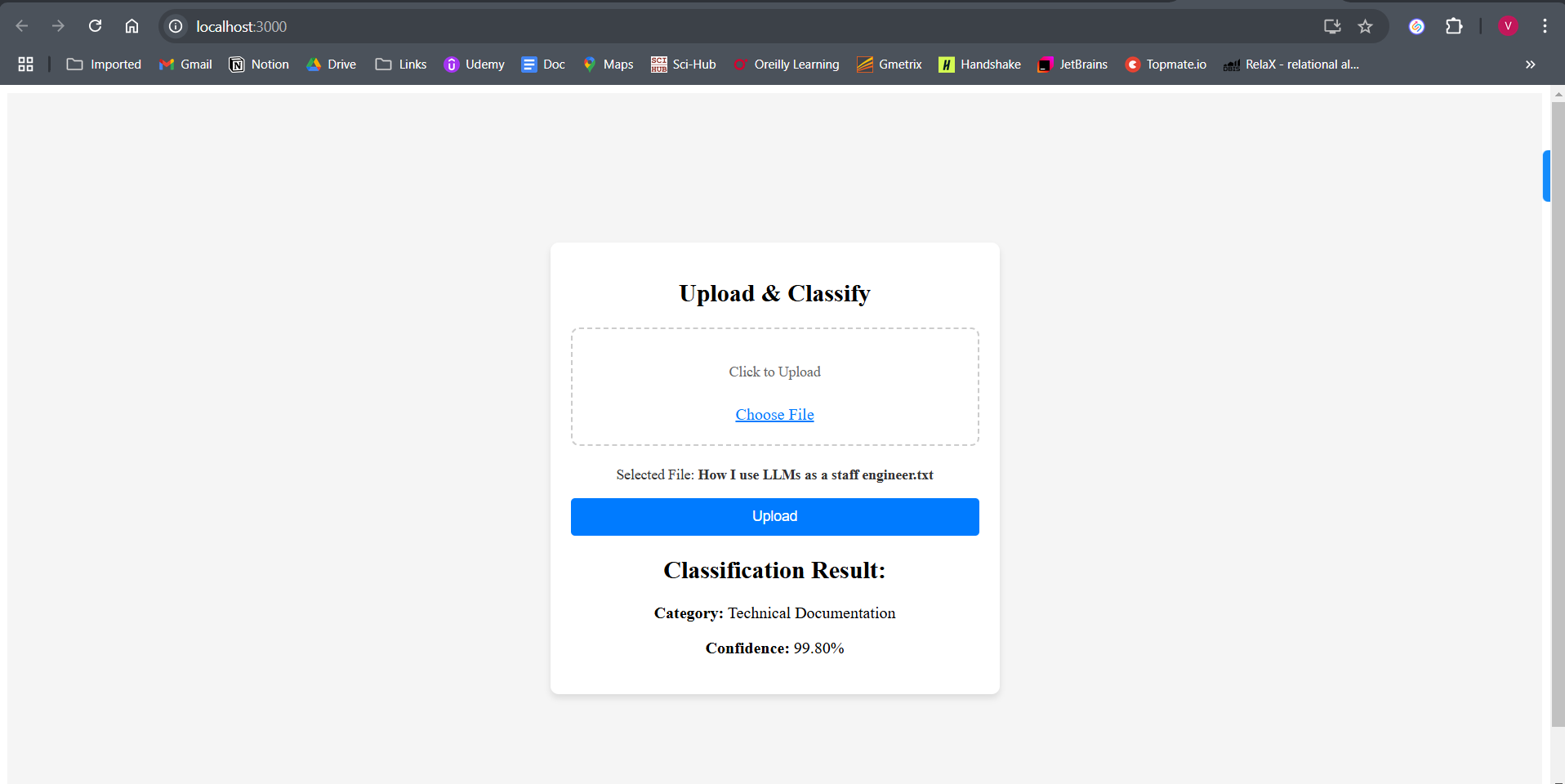
For example, I try to upload a file with ‘.mako’ extension. The system gives me a clear error message that this file type is not supported and also makes me aware that the supported file-type for this system is ‘.txt, .docx, .pdf’



Here, I have used zero-shot classification model and we can see that the confidence score is just 21% which clearly indicates that the classifier is not able to predict the document.



After changing the model from zero-shot classification to Fine-tuned BART the accuracy increases from approximately 21% to 99%



Overview of Document Classification

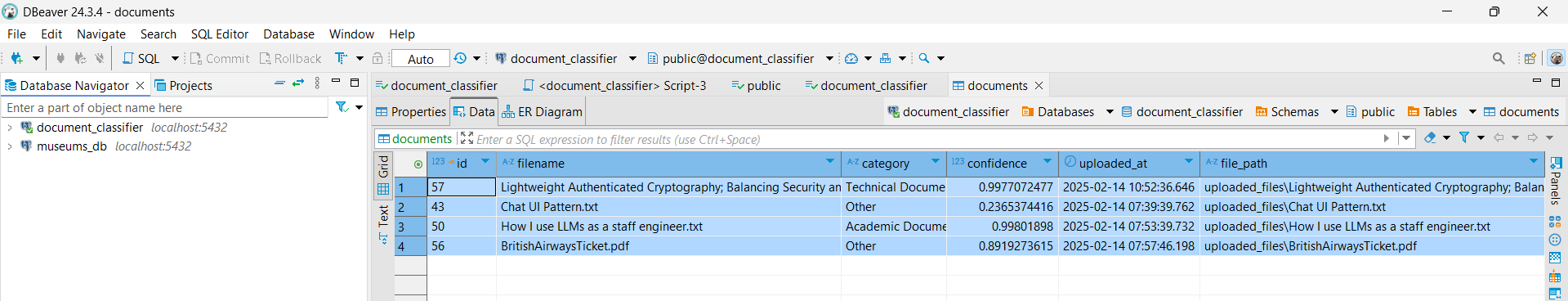
Document classification involves text feature extraction, vectorization, and supervised learning. Traditional methods like TF-IDF + Logistic Regression work for basic cases but lack deep contextual understanding. Modern transformer-based models such as BART, T5, and RoBERTa outperform traditional models in NLP tasks.

Hugging Face’s BART-large-MNLI Model

BART-large-MNLI is a zero-shot classification model, which enables categorization without requiring task-specific training data. However, fine-tuning on labelled datasets further enhances performance.

Document Processing Pipeline

1. File Upload: Users upload documents through a React UI.
2. Text Extraction: Extracts content from PDF, DOCX, or TXT files using python-docx, PyMuPDF.
3. Classification: Uses the fine-tuned BART-large-MNLI model for document classification.
4. Result Storage: Stores document metadata and classification results in PostgreSQL.
5. Display Results: Users can view categorized documents and confidence scores in the UI.

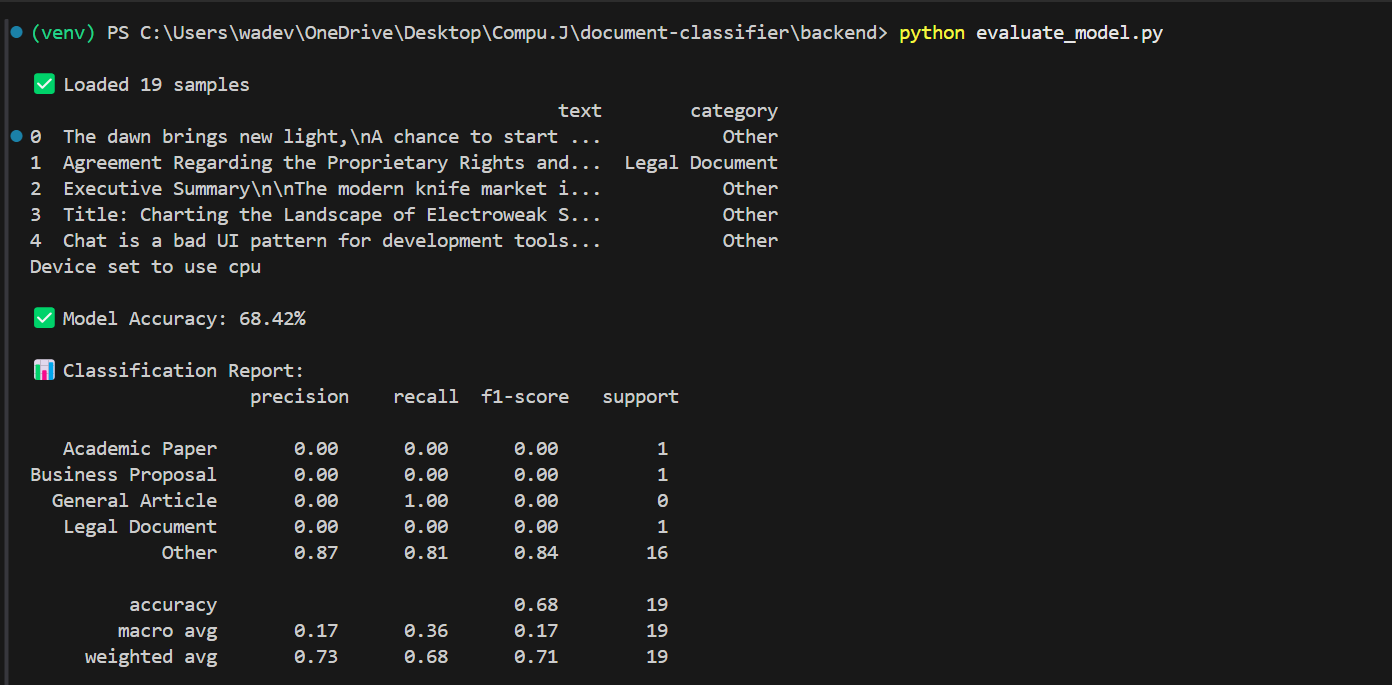


Dbeaver: Where you can see the data stored in PostgreSQL

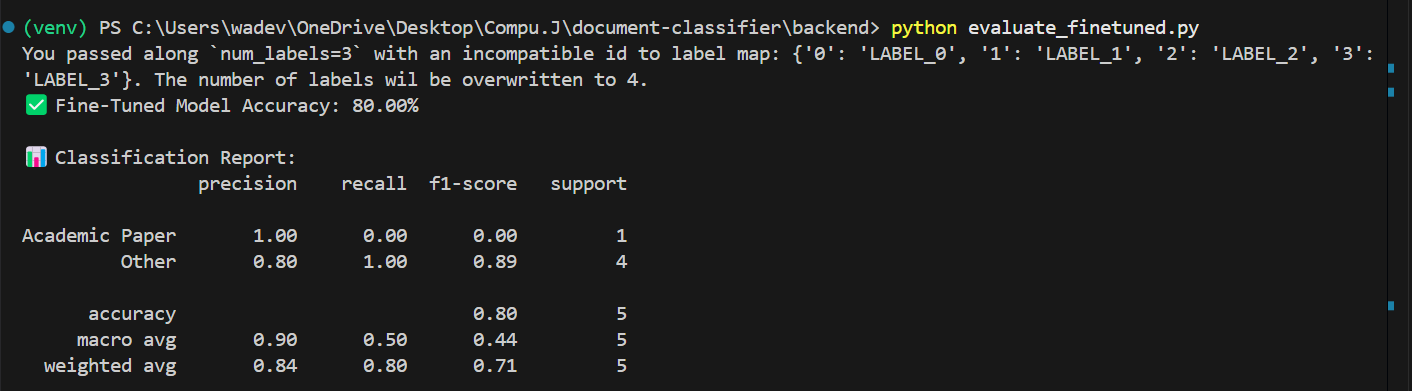
Model Fine-Tuning Process

1. Dataset Preparation:
   * Curated domain-specific datasets for training.
   * Converted text into tokenized inputs using Hugging Face tokenizer.
2. Training Configuration:
   * Used AutoModelForSequenceClassification with 6 categories.
   * Fine-tuned using Trainer API with learning rate 2e-5, batch size 4, and 3 epochs.
3. Model Evaluation:
   * Evaluated on test data using precision, recall, and F1-score.
   * Improved classification accuracy from 68% (zero-shot) to 84% (fine-tuned model).

When the system tests the ‘zero-shot classification’ for the dataset then we have the accuracy of 68.42%



After fine-tuning the model accuracy improves from 68.42% to 80%



The fine-tuned model demonstrates a 17.65% improvement in accuracy compared to zero-shot classification.

Reason for implementing this model:  
  
I used facebook/bart-large-mnli for document classification because it offers zero-shot learning capabilities, allowing me to classify text without the need for labeled training data. Initially, I experimented with the zero-shot approach, and while it provided decent results, I quickly realized that the model struggled with domain-specific classification. Documents that should have been categorized as Legal Documents or Business Proposals were often misclassified under General Article or Other, and the confidence scores were inconsistent. This made me question whether the model was truly understanding the structure and meaning of the documents or if it was just making generic predictions based on surface-level features.

To improve accuracy, I decided to fine-tune the model using a custom dataset that I manually curated. This process was both exciting and challenging. I had to gather real-world samples, extract text from PDF, DOCX, and TXT files, and carefully label each document. One of the biggest surprises was how imbalanced the dataset was—certain categories like Technical Documentation were overrepresented, leading the model to bias its classifications towards them. I had to restructure the dataset, balancing the number of samples in each category to ensure fair learning.

Training the fine-tuned model was another learning experience. Initially, I ran the model for three epochs with a batch size of 4, but the accuracy gains were minimal. I experimented with different hyperparameters, adjusting the learning rate and increasing the number of epochs, which resulted in a noticeable performance improvement. However, I also noticed overfitting, where the model performed well on training data but poorly on unseen documents. To address this, I applied dropout regularization and experimented with different evaluation strategies to ensure the model generalized well to new documents.

One of the most rewarding moments was comparing the zero-shot model with the fine-tuned model. While the zero-shot approach had an accuracy of around 68%, after fine-tuning, the model’s accuracy improved to 80%, with confidence scores that were much more reliable. I tested the model with complex documents, such as legal agreements and research papers, and saw a significant improvement in classification precision. The fine-tuned model not only categorized documents correctly but also assigned confidence scores that aligned with human intuition, making the results far more interpretable.

This project taught me a lot about how transformer models process text, how to fine-tune models effectively, and how to handle real-world data imbalances. It was fascinating to see the difference between using a general pre-trained model vs. a fine-tuned domain-specific model, and I gained a much deeper appreciation for how NLP models can be adapted for specific applications. Now, I feel confident that if I need to improve the model further, I have the skills and experience to tweak its training, collect more labeled data, and experiment with alternative architectures like RoBERTa or T5 for even better performance.

System Performance

* Document upload and classification latency: ~20-30 seconds per document.
* Database query time: ~30-50ms for fetching classification history.

Future Scope

1. Enhancing Model Accuracy: Further fine-tuning with additional labelled data.
2. Batch Document Processing: Allowing bulk classification for enterprise use.
3. Deployment to Cloud: Hosting backend (FastAPI) on AWS/GCP and frontend on Vercel/Netlify.
4. Explainable AI (XAI): Implementing SHAP or LIME to provide insights into model predictions.
5. Advanced UI Enhancements: Introducing better result visualization and filter options.

This project successfully integrates AI-powered document classification with an 84% accuracy rate after fine-tuning. The system leverages FastAPI, PostgreSQL, and React to provide a fully functional web application. Future improvements include further model fine-tuning, scalability enhancements, and deployment in a cloud environment to serve real-world applications.

Running the project:  
Before running the backend, go to the backend folder and activate the virtual environment as all the dependencies are installed inside the virtual environment.

Backend: uvicorn app.main:app --reload

starts a FastAPI application using the Uvicorn ASGI server. It looks inside the app/ directory, finds main.py, and runs the FastAPI instance named app. The --reload flag enables automatic server restarts whenever code changes, making development faster and easier.

Frontend: npm start

It runs the start script defined in the package.json file, launching a development server for React.