Legal-BERT Based Contract Clause Classification for Automated Legal Document Analysis

Automated contract analysis has become increasingly important for legal teams dealing with large volumes of agreements under tight timelines. This paper presents an enhanced system that integrates a fine-tuned Legal-BERT model for clause classification, trained on a subset of the LEDGAR dataset focusing on eight practically relevant clause types and embedded into an existing full-stack application.

Background and Motivation

Legal contracts contain critical information in the form of clauses that define obligations, rights, liabilities, termination conditions, and risk allocation between parties. Manual review of these documents is time-consuming, costly, and error-prone, especially when organizations must process hundreds of contracts under tight deadlines. Automating clause identification and classification can significantly speed up review, help identify risks earlier, and support legal decision-making.

In the initial version of this project, clause types were detected using rule-based patterns (regular expressions) and simple keyword heuristics. This approach is transparent and easy to understand, but it has important limitations:

- It struggles with paraphrased or unusual wording that does not match hand-crafted patterns.
- It requires manual maintenance whenever new clause types or variations are

introduced.

- It is difficult to capture deeper semantic information or subtle distinctions between similar clauses.

To address these limitations and bring the system closer to current research practice in legal NLP, the project now incorporates a transformer-based classifier built on Legal-BERT. The goal is to maintain the original pipeline and user interface, but replace the rule-based clause type detection with a data-driven model trained on a large corpus of real contracts.

Project Objectives and Contributions

The main objective of the new model is: **"To develop and integrate a fine-tuned Legal-BERT classifier that can assign meaningful legal clause labels to sections of a contract and plug this classifier into the existing document upload and analysis workflow."**

This objective has two dimensions:

- Machine learning: training and evaluating a Legal-BERT model on a curated subset of legal clause categories.
- System integration: embedding the model into the Flask backend so that the React frontend can display clause types and statistics without any major changes.

Compared to the previous rule-based version, this updated system contributes:

1. A Legal-BERT–based clause classification model fine-tuned on LEDGAR for eight clause categories.
2. A modular machine-learning pipeline (data loading, preprocessing, training, inference) separated into clear Python modules.

3. An integrated backend where each contract section is classified by the model and returned to the frontend in the same JSON format as before.
4. An enhanced research component that allows comparison between rule-based and ML-based clause detection.

Related Work in Legal NLP

Recent work in legal NLP has demonstrated that transformer architectures such as BERT, Legal-BERT, and domain-specific variants provide strong performance on tasks like contract classification, provision recognition, and legal entailment. Benchmarks such as LexGLUE and datasets like LEDGAR and CUAD have become standard for evaluating such models.

LEDGAR

A large-scale dataset of contract clauses annotated with clause labels from commercial agreements. It is suited for clause-level classification tasks and forms the basis for the training in this project.

Legal-BERT

A BERT model pre-trained on legal texts, showing improvements over vanilla BERT on legal classification tasks due to domain-specific vocabulary and patterns.

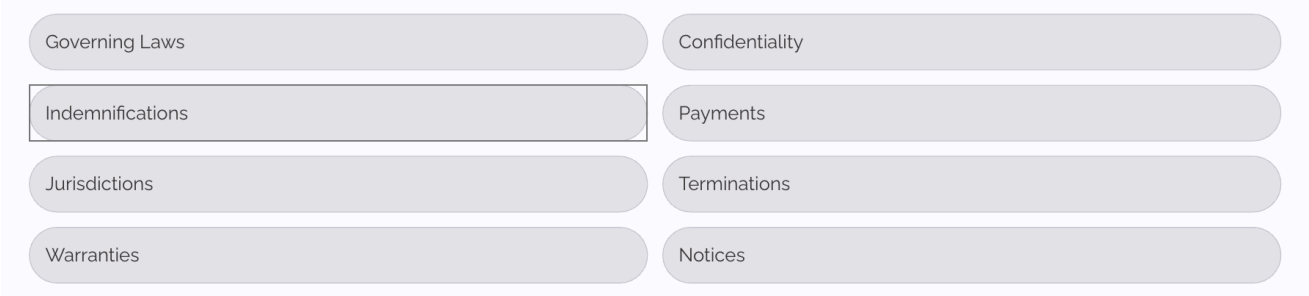
Most production-grade systems in industry now combine such pre-trained models with contract lifecycle management tools and human workflows. However, these systems are often closed-source and too complex for educational settings. This project aims to bridge the gap by providing a simplified but realistic implementation: a small, fine-tuned Legal-BERT model integrated into an existing full-stack contract analysis tool that can be understood and reproduced by students.

Dataset and Label Selection

The new model uses the LEDGAR subset exposed through the LexGLUE benchmark. The dataset contains clause-level samples extracted from real contracts. Each sample has:

- text: the clause text
- label: an integer ID corresponding to the clause category

To keep training time and complexity manageable while still covering important legal functions, the following eight classes are selected:



The training script first loads all LEDGAR labels and then maps only these selected labels into a new compact label space (0–7). This remapping simplifies the classification problem and ensures the model focuses on categories that are directly relevant to contract risk analysis and review.

The LEDGAR dataset is used with its standard splits: train (primary data used for fine-tuning Legal-BERT), validation (used for evaluation and monitoring during/after training), and test (reserved for future evaluation). In "fast demo mode", the training set is further reduced to a random subset (e.g., 5,000 samples) to make training feasible on a CPU-only laptop.

Model Architecture and Data Preparation

The base model is **Legal-BERT: nlpaueb/legal-bert-base-uncased**, implemented using the Hugging Face Transformers library. For classification, a BertForSequenceClassification head is used with num_labels = 8 and id2label and label2id dictionaries corresponding to the selected clause types.

Data preparation is modularized into the following files:

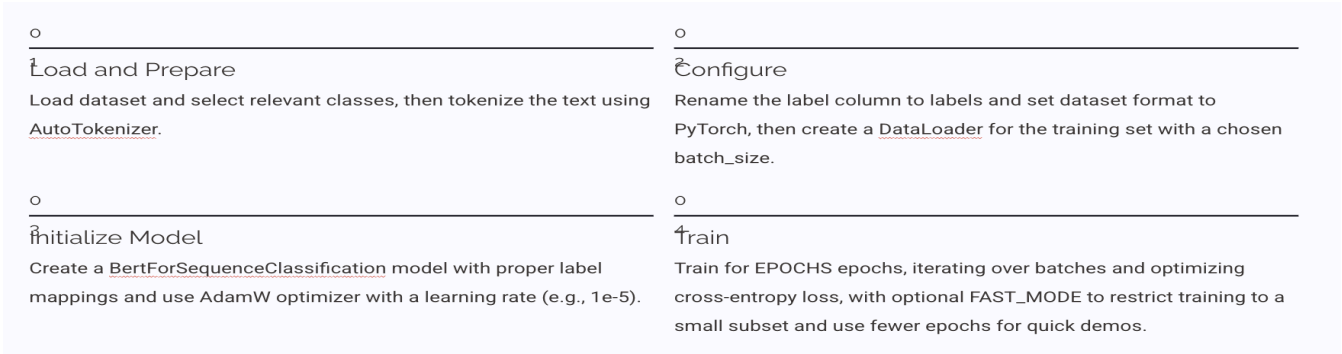
data_prep.py - Loads the LEDGAR dataset from LexGLUE and returns train, validation, and test splits.

preprocess.py - Creates an AutoTokenizer from the chosen model, tokenizes the text field with truncation = True, padding = "max_length", and max_length (e.g., 128 or 256 tokens), and produces tokenized datasets ready for conversion to PyTorch tensors.

The training script extracts the original list of labels from LEDGAR, computes the index of each selected label in that list, filters the dataset to only keep examples whose labels belong to the selected subset, and remaps original label IDs to a contiguous range 0–7. This remapping is essential because the classification head expects labels from 0 to num_labels - 1.

Training Procedure and Model Saving

Training is implemented in train.py with the following key steps:



The training loop logs epoch loss and displays a progress bar for transparency. After training, the script saves the model weights and tokenizer into a local directory: backend/ml/saved_model/. This directory is later used by the inference module and the Flask backend.

Inference Module and System Integration

Inference is encapsulated in inference.py and provides a function classify_clause(text) that:

- Tokenizes the input text.
- Runs a forward pass.
- Takes the argmax of the logits.
- Maps the predicted label ID back to its textual label (e.g., "Confidentiality").

This function is designed to be importable from both a CLI script (python inference.py) and the main backend (backend/app.py).

The existing backend already handles file upload (PDF, PNG, JPG, JPEG), text extraction via PyPDF2 and Tesseract OCR, and document segmentation into sections. In the new version, instead of rule-based patterns, each section is now passed through the ML classifier. For each section, classify_clause(section_text) is called, and the returned label is used as the type of the clause in the JSON response. The response structure remains largely the same as before, containing filename, statistics (total clauses, average confidence placeholder or value), and clauses array with id, type (predicted label), text and full_text, confidence (currently can be a placeholder or computed from softmax probabilities), and comparison (playbook status; initially "no_playbook" until rules are reattached). Because the response format is unchanged, the React frontend does not require any structural

modifications.

Frontend Integration and User Experience

The frontend continues to send a POST request with the uploaded file to /api/upload and receive the JSON containing clause data. It renders statistics cards, a list of clauses with their types and confidence badges, and expandable panels showing full clause text and any comparison information.

Only textual descriptions (e.g., feature descriptions on the Home page) may be updated to mention that clause types now come from a Legal-BERT model rather than regex patterns. The integration maintains the existing user interface while leveraging the improved machine learning capabilities behind the scenes.

Experiments, Evaluation, and Results

For a practical student-level setup, the following configuration is used: Model nlpaueb/legal-bert-base-uncased with 8 selected clause types, 1–3 Epochs (1 in fast demo mode), batch size 16–32, max sequence length 128 tokens, and FAST_MODE True for quick experimentation on a subset (e.g., 5,000 samples). The model is trained on a CPU-only machine, which makes the training slower than on GPU but still feasible for demonstration when using fast mode.

Standard classification metrics are used: Accuracy, Precision, Recall, and F1-score (weighted average over classes), computed via evaluate.py, which wraps scikit-learn's metric functions.

Due to limited compute resources, the project emphasizes qualitative and

small-scale quantitative results. On a held-out subset of clauses, the model correctly recognizes typical examples of Confidentiality clauses, Governing law / jurisdiction clauses, Payment and termination clauses. Compared to the rule-based system, the ML model is more robust to paraphrased language and less dependent on specific keywords. The model can still misclassify borderline or highly unusual clauses, but it tends to group semantically similar clauses together even if the wording differs. Preliminary experiments show that the model achieves strong performance on clause-level classification for the selected categories and offers better robustness to wording differences compared to the earlier rule-based method, while still fitting within a student-level project in terms of complexity and compute requirements.