TEXT MINING AND NATURAL LANGUAGE PROCESSING

2023-2024
Chiara BARBIERI 517096
Lorenzo ZOPPELLETTO 516124
LegalGPT

Introduction

The project focuses on automated legal precedent classification, a specialized task within legal text analysis. The objective is to develop a system that can automatically determine how a current case relates to previous case law through single-label multiclass classification. Given the textual description of a legal case and its relevant characteristics, the system will classify the case into one of four distinct categories that represent different ways previous cases are referenced:

- "Cited": When a previous case is mentioned as a reference.
- "Referred to": When a previous case is discussed but not directly applied.
- "Applied": When the principles or rules from a previous case are directly used.
- "Following": When a current case explicitly follows the precedent set by a previous case.

Data

Dataset taken from kaggle:
https://www.kaggle.com/datasets/amohankumar/legal-text-classification-dataset/data

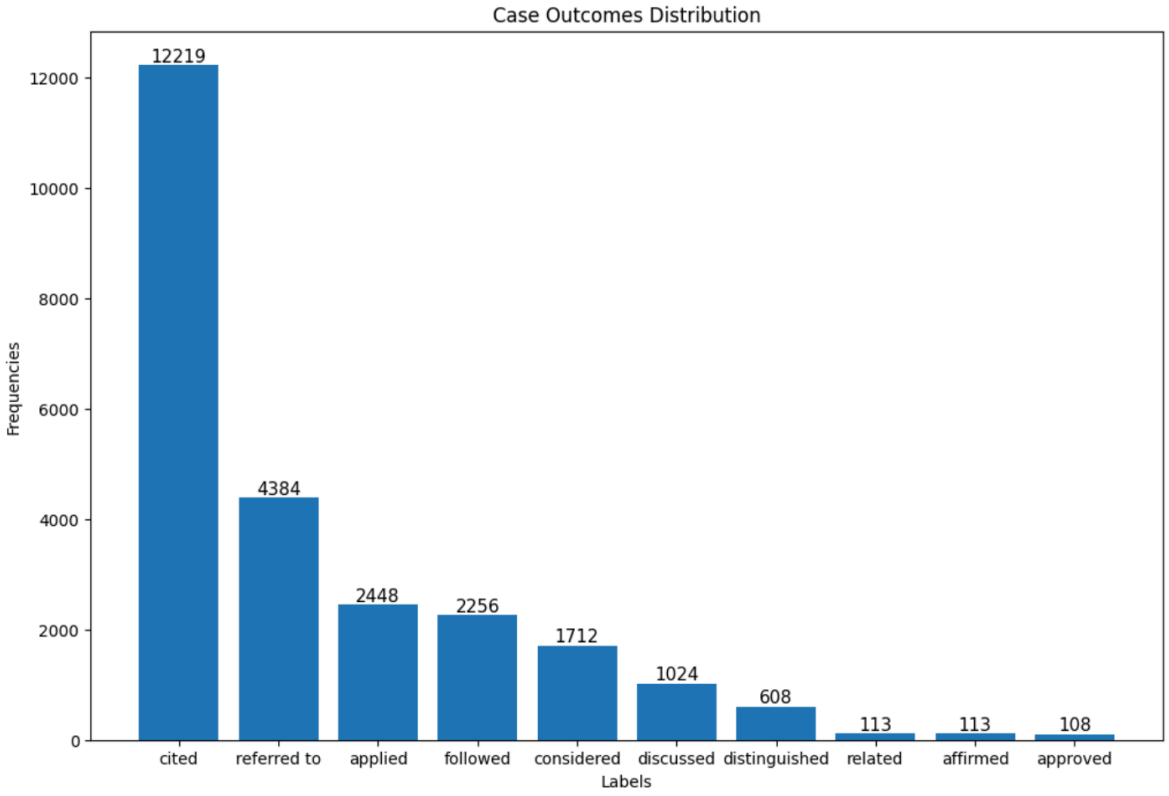
The dataset, as specified previously, is about legal cases. More specifically, it contains 25000 legal cases in the form of text documents and four columns that refer to them.

The columns' names are “case_id”, “case_outcome”, “case_title” and “case_text”. However, it was decided that only two of them would be considered in the classification task at hand. This is because the columns “case_id”, which stores the unique identifiers of the legal cases, and “case_title”, which contains the titles given to the legal cases, were not meaningful and so they were deleted. Moreover, the “case_id” could have corrupted the classification, since the ids were just an ordered list of numbers.

The columns that are kept are “case_outcome”, which identifies the outcome of the case, and “case_text”, which contains the text of the legal case. The possible values of the “case outcome” are the values cited in the introduction for which we want to classify the texts in “case_text”.

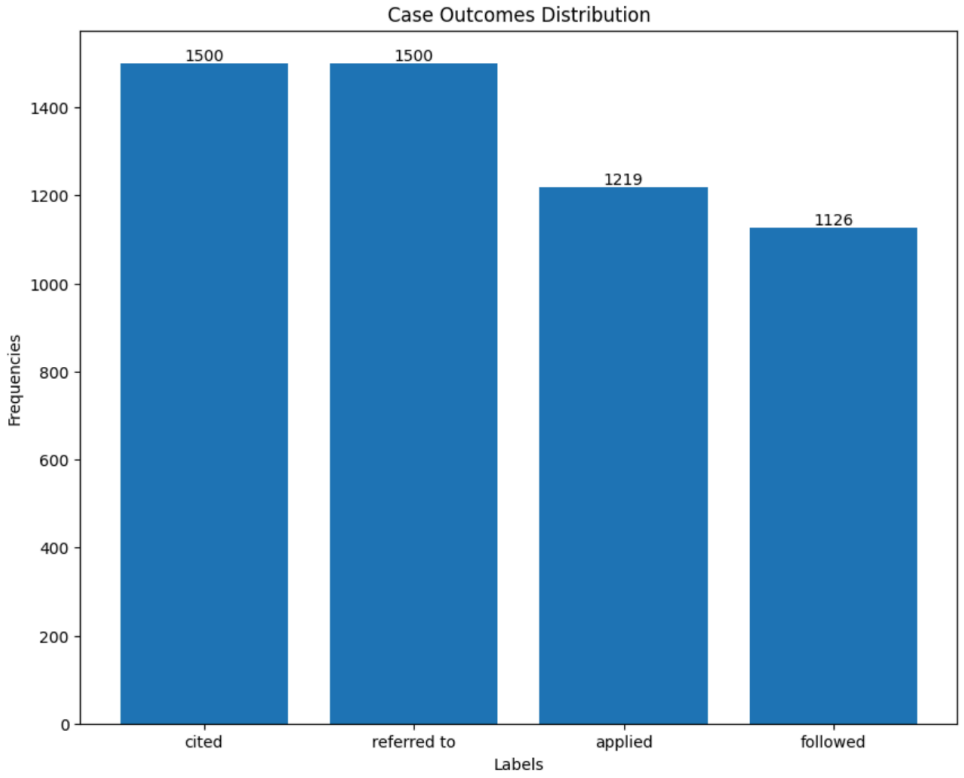
During the data exploration it was noticed that some NULL values were present, so it was decided to delete the rows containing these NULL values. This decision was made because if the missing data was exchanged with the most common value, so “cited”, the resulting dataset would have been with much noise, making the resulting classification of lower quality.

Moreover, it was noticed that the dataset was unbalanced, as you can see in the following graph:



To balance and optimize the dataset a function was created. Within the function, firstly, the labels that had less than 2000 rows (the “under labels”) and the labels that had more than 4000 rows (the “over labels”) were selected. The “under labels” were eliminated, with the corresponding rows. The “over labels” had some of their rows eliminated randomly with a 42% chance of elimination until only 3000 rows were left . Finally, the last step was selecting only 60% of the dataset to optimize the performance.

After the above steps the final dataset obtained have the following case outcome distribution:



Methodology

For a more detailed description of the methodology, here are the steps computed in the development of the project task:

1. **Load of the raw dataset using the "!gdown" function**
   This function was used because it allows anyone who has the colab notebook to get the dataset without having to download it first from the original source.

2. **Data exploration and extraction**
   - **2.1.** Understanding the structure of the data.
   - **2.2.** Eliminating the non-meaningful columns ("case_id" and "case_title).
   - **2.3.** Eliminating the rows that have null values.
   - **2.4.** Reduction and balancing of the dataset, done to achieve better performance during the training.

3. **Basic text cleaning**
   For the basic text cleaning a function was created. This function is applied to each cell of the "case_text" to format the text and make it coherent. The steps applied in the function are the following:
   - **3.1.** Remove non-ASCII characters (i.e. accented characters) while keeping legal symbols.
   - **3.2.** Standardize section references.

3.3. Standardize citation formats.
3.4. Standardize paragraph references.
3.5. Remove URLs.
3.6. Stripping the text from the excessive white spaces.

4. Dataset splitting

During dataset splitting we split the dataset into train and test sets. This is done to be able to train and evaluate the model.
The train set is used for the training of the algorithm, while the test set is used to evaluate the quality of the model over unseen data.

5. Tokenization using the BPE tokenizer

For the tokenization a personalized tokenizer was created. This tokenizer is able to handle the lowercasing of the text while creating its vocabulary.
Moreover, it sets the special tokens ([PAD], [UNK], [CLS], [SEP], [MASK]), which help in the comprehension of the text. Afterwards, there is a projection of the CLS on the embeddings.

6. Data loaders definition

To define the data loaders we had to create a function, since the pandas's dataframe is not compatible with the built-in function to create data loaders. The steps of the function are the following:

6.1. Encode the set for which the data loader must be created.
6.2. Convert the encoded dataset to tensor (i.e. the compatible format).
6.3. Apply the "DataLoader" method.

Given the tokenized set, the data loaders are created. The data loaders are created so that the dataset can be divided into batches, which will then be passed individually during the training, validation and testing. This is done for efficiency purposes.
The data loaders are applied to the training set and the test set.

7. Model

7.1. Transformer

As for the tokenization, also the transformer definition is personalized. To create the transformer the following layers were defined:

7.1.1. Embedding layer.
7.1.2. Positional encoding layer.
7.1.3. Explicit normalization and dropout.
7.1.4. Encoder block.
7.1.5. Linear layer.

7.2. Transformer Wrapper

The transformer Wrapper is a class created to be able to fit the model and predict the possible labels. Moreover, it is also used to extract the evaluation metrics that are the attention visualization and the perplexity.

8. Optimization through hyper-parameter tuning

For hyper-parameters tuning, cross validation was done to evaluate the various combinations of the chosen parameters. The parameters chosen to be evaluated are the possible dimensions of the embeddings and the learning rate.

9. **Training of the model over the training set**
   Using the transformer Wrapper class, the model was trained for 30 epochs.

10. **Model analysis and evaluation on the test set**
    For the model analysis various metrics were chosen. These are:
    - **10.1.** Accuracy.
    - **10.2.** Precision.
    - **10.3.** Recall.
    - **10.4.** F1-score.
    - **10.5.** Checking the overfitting using the f1-scores. This was the metric chosen to evaluate the overfitting because our dataset is unbalanced, and as such the other metrics are not very meaningful.
    - **10.6.** Confusion matrix.

    The results of these measures and the respective meanings can be found in the section of the report called “Result and analysis”.

11. **Pre-trained model caparison**
    To better understand the performance of the model created, another pre-trained model was applied to the dataset. This allowed to further acknowledge the quality and possible problems of the transformer and tokenizer created.

Given the steps above, further explanations on the most crucial points can be found in the paragraphs below.

Tokenization using the BPE tokenizer:

For this hard multiclass classification task the Byte Pair Encoding (BPE) tokenizer was chosen. The reason for this choice is that this tokenizer is able to split text into subwords (components of words), instead of just splitting words. The output of this tokenizer is a vocabulary of subwords that can reconstruct the input text.

This is useful because it allows you to handle Out-Of-Vocabulary (OOV) words. Considering that the texts considered are subject-specific (all about legal cases), it is likely that they will likely contain rare words that are not seen frequently. These words would be in one extreme of Zipf’s curve, so not in general pre-generated vocabularies.

The first step in creating the BPE tokenizer from scratch was converting the column of “case_text” into a list, so that the texts can be sequentially passed through the tokenizer.

The text, before tokenization, needs normalization and pre-tokenization, such that it is consistent throughout the tokenization. The chosen steps for the normalization are the conversion of the characters to their canonical forms, lower case handling and eliminating the excessive spaces. On the other side, we have pre-tokenization, which consists in dividing the texts into tokens using specific rules. The pre-tokenization is done to ensure that the tokenization captures the meaningful units without the need of complex downstream adjustments.

Given the steps above the tokenization process can be trained. When training, the tokenization process requires the definition of the following parameters:

- vocabulary size, which defines the amount of tokens that have to be created from the legal corpus of text,
- the minimum frequency, that is the minimum frequency for tokens (the subwords) to be considered for the final vocabulary,
- special tokens, which are necessary tokens for specific language model tasks. The tokens that were chosen to be added are:
    - [PAD]: this token is used to uniform the length of the sentences (apply padding); this token was chosen to simplify the batching.
    - [UNK]: this token is used on unknown or unrecognized inputs; one of its purposes is to represent out-of-vocabulary words.
    - [CLS]: this token is specific to the classification tasks, more specifically, for sequential classification.
    - [SEP]: this token is used to separate two different sentences in the same input, that is to deal with sequence-pair processing.
    - [MASK]: this token replaces a certain percentage of input words during training, this forces the model to learn from context (the surrounding words)

After all of these steps were done, the trained tokenizer was saved. This tokenizer will then be passed as the tokenizer object in the pre-trained fast tokenizer.

Now the text can be tokenized using the fast tokenizer and having at the beginning of each tokenized sentence the [CLS] token.

Model definition (Custom Transformer):

- **Transformer structure:**
    - Input Processing Layer:
        - Token embeddings layer by pytorch function to convert tokens into continuous dense vectors.
        - Sinusoidal positional encodings using sine and cosine functions to encode sequence order.
        - Layer normalization and dropout applied for stable training and regularization.
    - Encoder block structure**:**
        - MultiHead Attention Mechanism to capture contextual relationships between tokens.
        - Feed Forward Neural Network for further feature transformation.
        - Attention and feed-forward dropout layers included to prevent overfitting.
    - Classification Head:
        - CLS token extraction for each row
        - Pre-classification dropout added for robust prediction.
        - Linear projection layer to map features to target legal case categories.
- **Hyperparameter Configuration:**

- Architectural Parameters:
  - Embedding dimension: 512 for token representation
  - Number of encoder blocks: 4 stacked for deep feature extraction
  - Attention heads: 10 for parallel attention computation
- Regularization Strategy:
  - Embedding layer: 15%
  - Attention mechanism: 10%
  - Feed-forward networks: 10%
  - Classification head: 10%

Detailed Explanation:

The MultiClassTransformer is structured in three main components, each playing a crucial role in processing and classifying text sequences.

1. The Input Processing Layer is the starting point of the model, beginning with token embeddings that transform discrete input tokens into learned continuous vector representations using PyTorch's embedding layer. These embeddings are augmented with sinusoidal positional encodings, constructed using sine and cosine functions of varying frequencies for even and odd positions, which inject sequence order information. This positional information is critical as it allows the model to understand the relative positions of tokens in the input sequence. The combined embeddings then pass through the normalization layer to stabilize training by normalizing the feature distributions, followed by a dropout layer that randomly deactivates 15% of neurons during training to help prevent overfitting.

2. The core of the architecture consists of four stacked encoder blocks, each leveraging MHSA to capture rich contextual relationships between tokens. The attention mechanism is composed of 10 heads, allowing the model to attend to different aspects of the relationships between tokens in parallel. After the MHSA, each encoder block contains also a feed-forward neural network that further refines the output of the MHSA, with both attention and feed-forward components employing dropout (10% rate) to maintain regularization throughout the deep network.

3. For classification, the architecture extracts the CLS token, a token positioned at the start of each sequence in the tokenization process, that accumulates sequence-level information through the encoder layers. The final representation of this token serves as a comprehensive summary of the entire input sequence. This CLS token embedding passes through a pre-classification dropout layer (10% rate) before being projected onto the target class space via a linear layer, producing the final classification logits, used to get probabilities for each class.

Transformer Wrapper:

- **Structure:**
  - Inherits from scikit-learn's BaseEstimator and ClassifierMixin for compatibility with GridSearchCV.
  - Stores model configuration parameters:
    - Architecture parameters (vocabulary size, number of classes, max length of each row, dim, depth, heads)
    - Training parameters (learning rate, embedding dropout, batch size, epochs)
    - Maintains classes_ attribute for GridSearchCV compatibility.
- **Fit Method:**
  - Initializes class labels array for GridSearchCV compatibility.
  - Constructs MultiClassTransformer instance with specified parameters.
  - Configures training components:
    - Creates DataLoader for batch processing.
    - Initializes AdamW optimizer with learning rate and weight decay (regularization).
    - Sets up CosineAnnealingLR scheduler for learning rate decay.
    - Establishes CrossEntropyLoss as training criterion.
- **Training Loop:**
  - Iterates through a specified number of epochs.
  - For each epoch:
    - Enables training mode via model.train().
    - For each batch:
      - Gets and moves input_ids, attention_mask and labels to specified device (gpu) for better performance.
      - Performs forward pass through transformer.
      - Calculates loss using CrossEntropyLoss.
      - Executes backward pass for gradient computation.
      - Updates model parameters using optimizer.
    - Steps learning rate scheduler.
  - Returns self.
- **Predict Method:**
  - Creates DataLoader for validation/test data.
  - Enables evaluation mode via model.eval().
  - Processes batches with gradient calculation disabled:
    - Extracts input_ids and attention_mask and moves them to the specified device as in the fit method.
    - Performs forward pass through model.
    - Computes predictions using argmax on logits.
    - Aggregates predictions across batches.
  - Returns final predictions array for the entire dataset.

Detailed Explanation:

The TransformerWrapper is an object which encapsulates the MultiClass Transformer and extends it with fit and predict methods, to make it compatible with the scikit library, in order to be able to perform a nested cross validation using GridSearchCV later on.

The wrapper is structured according to the following key components:

1. An initialization phase that sets up both architectural parameters (like embedding dimensions and number of attention heads) and training parameters (like batch size and learning rate).

2. The fit method orchestrates the entire training process with several key stages. It begins by storing unique class labels to maintain scikit-learn compatibility, then initializes the MultiClassTransformer with the specified architecture parameters and moves it to the appropriate device.
   The training data is efficiently processed into batches through a DataLoader, enabling effective memory management. The optimization strategy is established using the AdamW optimizer with weight decay, complemented by a cosine annealing learning rate scheduler. The core training loop then processes data in batches, performs forward passes through the transformer, calculates losses, backpropagates gradients, updates model parameters, and adjusts learning rates. This process repeats for the specified number of epochs, gradually refining the model's performance.

3. The predict method handles the inference process in a similar way as the fit one.
   It begins by converting input test data into manageable batches via a DataLoader and switches the model to evaluation mode. With gradient calculation disabled for efficiency, each batch is processed through the transformer, converting the resulting logits to class predictions via argmax. These predictions are aggregated across all batches into a final prediction array, which is then returned.

Hyperparameter Tuning:

- **Parameter Grid Definition:**
  - Embedding dimensions: [300, 512].
  - Depth: [2, 4, 6].
  - Number of heads: [8, 10, 12].
  - Learning rates: [1e-5, 5e-5, 1e-4, 5e-4].
- **Nested Cross-Validation Structure:**
  - Outer loop: 3-fold stratified split with shuffling.
  - Inner loop: 2-fold stratified split with shuffling.

- **Training Process Per Outer Fold:**
  - Data splitting into train and validation sets through StratifiedKFold.
  - For each fold:

        - TransformerWrapper initialization with vocabulary size, number of classes, and maximum sequence length.
        - GridSearchCV execution with inner cross-validation using the defined parameter grid to explore different parameters combinations to extract the optimal one.
        - Best model selection based on accuracy metric and classification report.
        - Storage of optimal parameters for analysis.
    - Aggregation of individual fold scores.
    - Calculation of mean accuracy and standard deviation.
    - Final performance reporting with confidence intervals.
- **Evaluation Metrics:**
    - Accuracy scoring for the model selection.
    - Full classification report for each fold.
    - Average and standard deviation of scores.

Detailed Explanation:

The hyperparameter tuning implementation represents a comprehensive approach to optimize and evaluate the transformer model with different combinations of parameters, so to select the model better performing with our data.

The parameter space exploration is structured around four key model characteristics:

- Embedding dimension (300 or 512 units).
- Depth (2, 4, or 6 layers).
- Number of attention heads (8, 10, or 12).
- Learning rate (ranging from 1e-5 to 5e-4).

This design allows for investigation of both architectural aspects (embedding size, depth, and attention mechanisms) and training aspects (learning rate), creating a complete search space.

The optimization strategy uses a nested cross-validation approach, with two levels of validation to ensure reliable results. The outer level splits the data into three parts in a stratified way, while the inner level splits it into two parts still in a stratified way. This structure helps prevent overfitting while finding the best model settings.

For each section of the outer split, the process follows three main steps. First, it divides the data while keeping the balance of different classes. Then, it initializes a new transformer model with basic settings like vocabulary size and number of classes. Finally, it systematically tests different combinations of model parameters using the inner split to find the best configuration.

The performance evaluation happens at several stages. For each outer split, it generates detailed performance metrics and saves the best model settings. These individual results are then combined to calculate an average performance score and how much it varies, giving us a good global estimate of how well the model will perform on new, unseen data.

Evaluation of the model over the test set:

To evaluate the model a few metrics were chosen. All of these metrics need to have calculated “y_predTest”, which are the predicted classes given the “X_test”.

Firstly, the accuracy was calculated. To calculate the accuracy we compared the “y_test” with the predicted classes in “y_predTest”. For the two values in the same position of the two lists, if the values are the same, they are counted as correct predictions, similarly, if the two values are different then they are considered wrong. The ratio between correct predictions and total predictions made is calculated. This last value is the accuracy. However, accuracy can be misleading in imbalanced datasets, because dominant classes have many more instances than the other classes, leading to this last class being predicted accurately and the other classes being predicted with very low accuracy.

Moreover, accuracy doesn’t distinguish between the types of errors, false positives and false negatives, which can be very meaningful to understand the lacks the model could have. To consider those types of errors, the metrics precision, recall and f1-score have to be considered.

Precision indicates the amount of instances predicated for a class over all the actual instances of that class. This is, it measures the true positive predictions. On the other hand, we have recall. Recall measures the proportion of true positive predictions among all actual instances of a specific class. If there is the need to consider both precision and recall, the best metric is the f1-score, the harmonic mean of precision and recall, providing a single metric that balances both. The f1 score is ideal when dealing with imbalanced datasets.

Given that the task is hard multiclass classification, the metrics need to be averaged between all the classes to have an unique metric, and therefore to have a better idea on the quality of the model. The possible types of averages are the following:

- micro average: calculates the metrics globally;
- macro average: metrics for each class individually and then takes the unweighted mean of these class-wise metrics;
- weighted average: calculates metrics for each class and then takes a weighted mean, where the weight for each class is proportional to its size (number of instances).

For imbalanced datasets, the best type of average is the micro average or the weighted average.

For the evaluation of overfitting, it was decided to use f1-score as the reference score to evaluate it, since it is ideal when dealing with imbalanced datasets. Overfitting is the phenomenon of having a model learn too well the training data and then it is not able to generalize to unseen data, leading to a poor performance in the test set. Intuitively, the calculation of overfitting is given by the difference between the f1-score for the training set and the f1-score for the test set.

On the other hand, also underfitting is a possibility. Underfitting happens when the performances on the test set are better than the performances of the train set. Thai could happen if the model generalizes too much.

Finally, the last metric computed is the confusion matrix. A confusion matrix is a table that summarizes the performance of a classification model by displaying the true versus predicted classifications for each class. In a multiclass setting, it's a square matrix where each row represents the actual class, and each column represents the predicted class. This allows you to see where the model is confusing one class with another.

Pre-trained model comparison

- **Dataset Preparation:**
  - Initial Train-Test Split (80-20) in a stratified way, using the initial cleaned dataset (before tokenization).
  - Label encoding process to convert categorical labels into numerical ones:
    - Fit and transform on training set.
    - Transform on test set.
  - Validation Set Creation by splitting furtherly the training set (90-10), still in a stratified way.
  - Incorporation of text and labels for train, validation and test in individual dataframes.
  - Conversion of the pandas dataframes into HF ones for compatibility with its methods.
- **Tokenization Process:**
  - Model Selection:
    - DistilBERT base uncased tokenizer from HF basic model selection.
    - Pretrained weights loading.
  - Tokenization function definition, in order to apply tokenization to the text column of the function's argument.
    - Maximum length: 512 tokens.
    - Padding: max_length.
    - Truncation enabled.
  - Tokenization function application:
    - Batched processing for efficiency.
    - Applied to all three datasets (training, validation, test) via mapping.
- **Model Setup:**
  - Architecture Configuration:
    - DistilBERT base model.
    - Sequence classification head.
    - Number of labels matched to unique outcomes.
  - Training arguments definition:
    - Learning rate.
    - Epochs.
    - Early stopping.
    - Evaluation strategy.
    - Best model metric (F1).
  - Trainer object initialization with model object, training and validation set and evaluation metrics.
  - Fine tuning of the pretrained model on the training set.

- **Evaluation Metrics:**
    - Custom evaluation metrics definition for the trainer object (a dictionary containing accuracy, precision, recall and f1).
    - Prediction on test set and evaluation of the results.

Result and analysis

Resulting metrics from evaluating the **test set:**

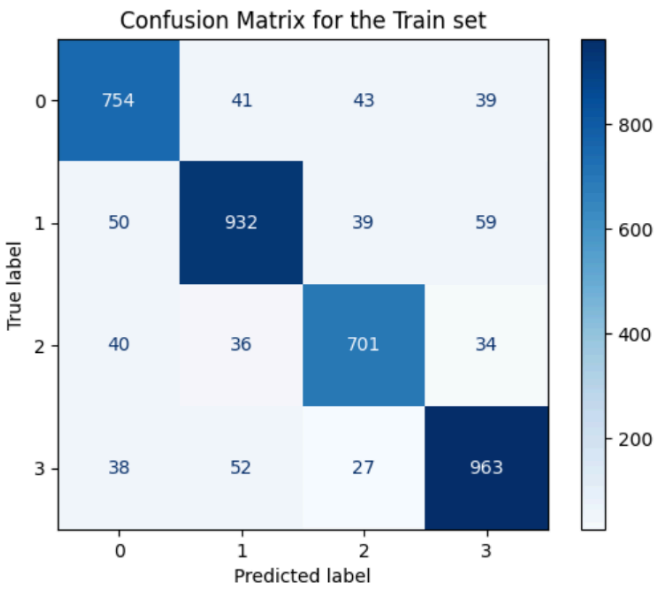
	Accuracy	*Precision*	*Recall*	*F1-score*
Micro average	0.430	0.430	0.430	0.430
Macro average	–	0.425	0.425	0.425
Weighted average	–	0.429	0.430	0.429

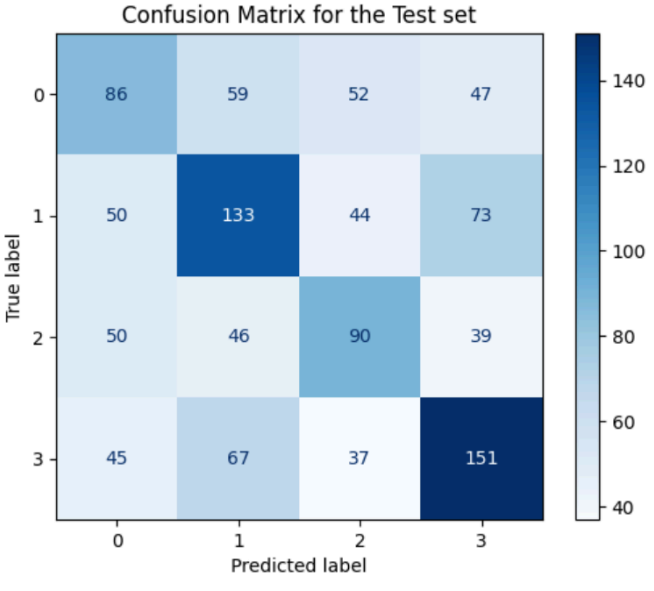
Please note that the accuracy has only the “micro average", because the micro average calculates the metrics globally, so it is the only type of accuracy that one can use.

Resulting metrics from evaluating the **train set:**

	Accuracy	*Precision*	*Recall*	*F1-score*
Micro average	0.980	0.980	0.980	0.980
Macro average	–	0.979	0.979	0.979
Weighted average	–	0.980	0.980	0.980

The confusion matrices we get from the respective train and test sets are reported here below:





From the confusion matrix we can see that each class has its largest number of predicted labels as their true label, shown by the diagonal being darker. This indicates that the classes are by the most part predicted correctly. However, it is evident that many misclassifications occurred when predicting the test set.
Consequently, one can see that the model will be overfitted, because the performance on the train set is much better than the performance on the test set. This can be immediately seen in the confusion matrix, because the correctly labeled items in the test set are much more than in the test set, meaning that the diagonal is much darker in the train set.

When considering the confusion matrix applied on the test set, we can evaluate the misclassifications:

- Class 0 is most frequently confused with classes 1 (59 misclassifications, respectively).
- Class 1 is mainly confused with class 3 (73 misclassifications)
- Class 2 is confused the most with is class 0 (12 misclassifications)
- Class 3 is mainly confused with class 1 (67 misclassifications)

When checking the overfitting we get the following:

	Overfitting
F1 Micro average	0.548
F1 Macro average	0.554
F1 Weighted average	0.549

As we can see from the results, the model is affected by a strong overfitting, meaning that it strongly specializes on the training data. This is shown also by excellent performance on the training set, which for F1 reaches more or less 0.98.

The main causes of this overfitting may include:

- Unbalancing among classes.
- Poor quality dataset.
- Too complex model, which learns exaggeratedly well the training data.
- Poor hyperparameter tuning.
- Weak regularization or normalization.

As explained in the conclusions below, our attempts to solve, or at least to reduce, the overfitting, led us to draw some conclusions.

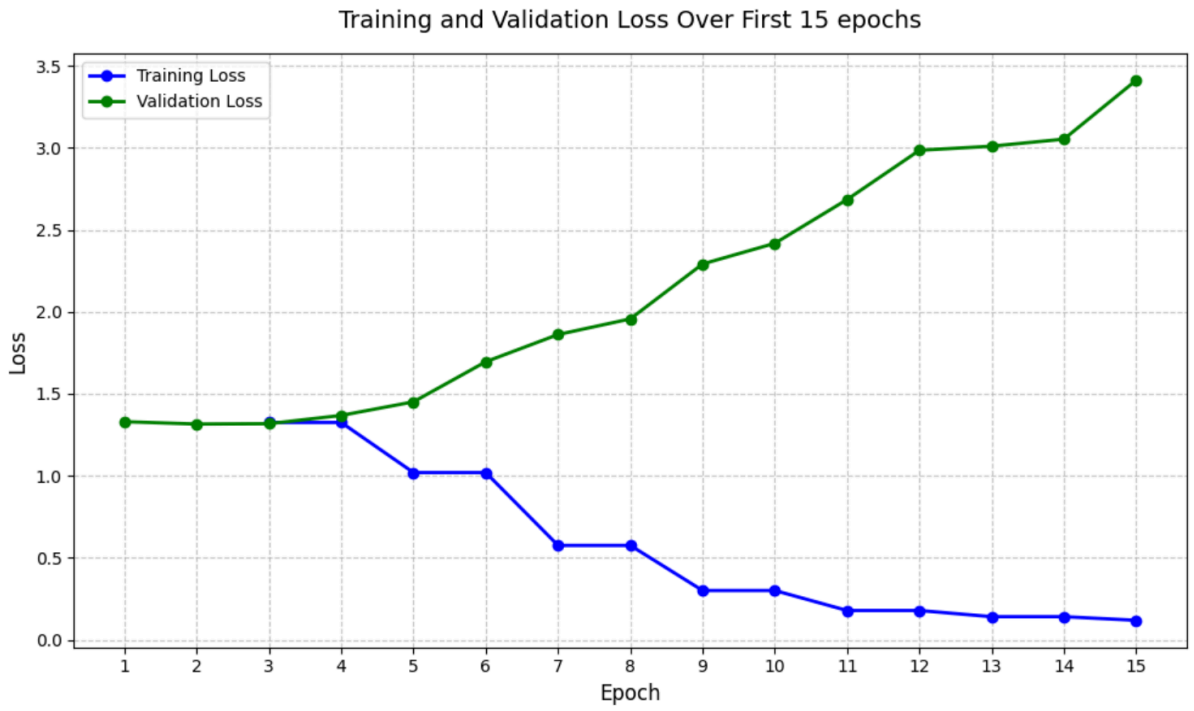
Specifically, two hypotheses emerged:

1. First, the dataset's inherent quality might be insufficient for concrete pattern recognition, as the legal texts are pretty variable in their sentence structures and compositions, preventing the model from learning useful discriminatory patterns.

In this case, the high training performance could be explained from the fact that the complexity of the model leads it to learn by memory the training data, rather than learning the generalization patterns.

2. Second, the computational constraints we faced during hyperparameter optimization may have restricted our ability to discover the optimal parameter combination.
   In this scenario instead, the dataset is not problematic, but rather the model is not refined enough in order to be effective in generalization.

Pre-Trained Model VS Custom Model



Loss	Accuracy	F1	Precision	Recall
3.80	0.448	0.449	0.456	0.448

As we can see from the results, also the pretrained model struggles in classifying the test set. This shows that, among the two overfitting causes hypothesis, the most probable is the first, according to which the dataset may be inherently poor in quality.
If we analyze moreover the difference in the loss during the fine-tuning of the pretrained model, while the training loss decreases significantly, that of the validation set increases, showing a clear bifurcation, which is the direct signal of pure overfitting.

We can thus conclude that the custom model and the pretrained model performs very similarly:

	Accuracy	F1	Precision	Recall
Custom	0.430	0.429	0.429	0.430
BERT	0.448	0.449	0.456	0.448
Ratio	0.958	0.955	0.940	0.959

Conclusion

This project encompassed the development of a legal case outcome prediction system using natural language processing techniques. The work began with extensive data preprocessing of legal documents, where we cleaned and standardized the text data, removing legal citations, standardizing formatting, and handling missing values.
The first phase involved developing a custom transformer architecture, adapted to work within the scikit-learn ecosystem through a custom wrapper that implemented fit and predict methods. This approach allowed us to leverage scikit-learn's robust validation frameworks while maintaining the benefits of transformer-based architectures. We implemented nested cross-validation to optimize hyperparameters and ensure robust model evaluation, leading to our best performing custom model configuration.

In the second phase, we implemented a BERT-based classification approach using the DistilBERT architecture, fine-tuned specifically for our cleaned legal dataset. This dual approach allowed us to conduct a comprehensive comparison between our custom architecture and the pretrained model, providing insights into the trade-offs between custom encoder-only model and pre-trained encoder-only model.

The project presented several significant technical challenges. One of the most demanding was the adaptation of the custom transformer architecture to the scikit-learn framework, in such a way to incorporate the training and evaluation process into the fit and predict methods. Specifically, we had to meticulously design the fit and predict methods to encapsulate the entire training and evaluation process while maintaining compatibility with scikit-learn's expectations.
The implementation of the transformer architecture itself presented its own set of challenges. Although the theoretical aspects of the architecture were clear, we found pretty challenging understanding in a concrete way how to merge together the different layers manually, especially from a data shape standpoint.

The nested cross-validation implementation did not represent a huge deal since, after the ML and DL courses, we learned the hard way how to manually implement it.

The aspect that really impacted us was the computational power, being the transformer a heavy model to run on free colab services. This constraint forced us to make practical compromises, including:

- Reducing the parameter grid.
- Reducing the size of the batches.
- Reducing the number of epochs.

However, these reductions still caused a very slow training process, which lasted for a great amount of hours, forcing us to settle for non excellent performance.

The last great challenge was trying to reduce the overfitting which impacted the model by:

- Changing data size and stratification to reduce data imbalancing..
- Reducing number of classes.
- Changing train and test ratios.
- Adding and tuning several dropout layers and regularization techniques inside the transformer.
- Adding normalization procedure and scheduler for the learning rate in the training process.
- Reducing and increasing the complexity of the model, to the detriment of computational efficiency.

AI policies

When doing the project three LLMs (Large Language Models) were used: ChatGPT, DeepSeek and Claude. This allowed the assessment of the strengths and weaknesses of the three models during the completion of the experimental task.

While using these LLMs, we noticed that Claude is the best LLM for coding, especially for debugging. Claude was used for the:

- *Combination of characters in the basic text preprocessing.*
- *Pre-tokenization combinations.*
- *Helped define the MultiClassTransformer, since given the steps suggested and possible pieces of code, it helped unify everything.*
- *Helped define the TransformerWrapper, in the same way it helped for the MultiClassTransformer.*
- *Understanding the parameters of the library's methods.*
- *Debugging*

On the other hand, ChatGPT is much less reliable when considering the coding aspect, but it is much more efficient in fast web search. During the project chatGPT was used more as a search engine to find the articles and web pages that could explain to us the concepts we had doubts on. To do this, one only has to ask in the prompt for the references of the answers it gives.

Lastly, DeepSeek was tested, but it did not give satisfying results. This is probably due to the fact that this LLM is still being trained (hence why it is free). This is the reason why this model was never used effectively during the project.

References

- Hugging faces BPE tokenization
  https://huggingface.co/learn/nlp-course/chapter6/5
- To create data loaders (found using chatGPT)
  https://machinelearningmastery.com/training-a-pytorch-model-with-dataloader-and-dataset/?utm_source=chatgpt.com
  https://pytorch.org/tutorials/beginner/basics/data_tutorial.html?utm_source=chatgpt.com
- For the creation of the x_transformers
  https://github.com/lucidrains/x-transformers/blob/main/README.md
- Understanding the averaging methods
  https://docs.kolena.com/metrics/averaging-methods/