Multi Label Classification

Models

Fine-tuned

https://drive.google.com/file/d/1-Gv3xTpz2wyL5LF9jIYUk28ghdksCpij/view?usp=drive_link

Pre-trained

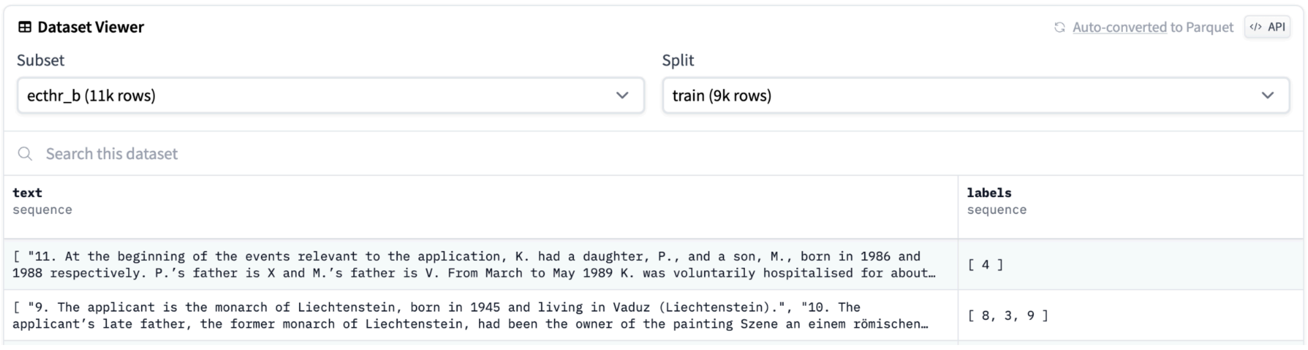
https://drive.google.com/file/d/1RuAil3kuWkbKuj7AQojycF_bf4D_PvzN/view?usp=sharing

Introduction

The goal of this project is to develop a multilabel legal text classification model for the European Court of Human Rights (ECHR) dataset. The dataset contains legal documents, and the task is to predict the articles of the European Convention on Human Rights allegedly violated by each case.

Dataset description

We are using ecthr_b dataset from the Lex GLUE collection, which is designed for the task of classifying legal documents tailored for the European Court of Human Rights (ECHR).



text sequence	labels sequence
["11. At the beginning of the events relevant to the application, K. had a daughter, P., and a son, M., born in 1986 and 1988 respectively. P.'s father is X and M.'s father is V. From March to May 1989 K. was voluntarily hospitalised for about…	[4]
["9. The applicant is the monarch of Liechtenstein, born in 1945 and living in Vaduz (Liechtenstein).", "10. The applicant's late father, the former monarch of Liechtenstein, had been the owner of the painting Szene an einem römischen…	[8, 3, 9]

Features & Labels

In the dataset specified above, we have just two features -

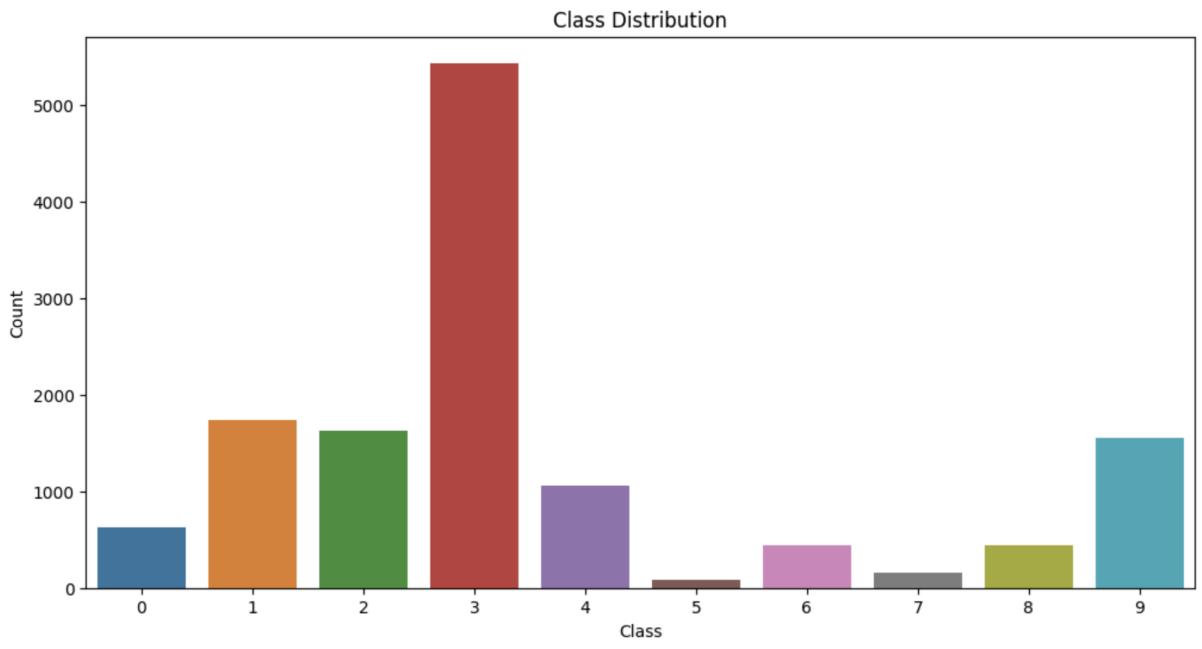
text:

This feature consists of textual content of the legal document, which is a list of factual paragraphs (facts) from the case description. Basically the legal document is divided into paragraphs and each string in the list represents a paragraph.

labels:
This feature is a list of numbers representing articles of ECHR that were allegedly violated (considered by the court) by the case specified in the document. For example, if the document violates article 3 and 4, labels will be [3, 4].

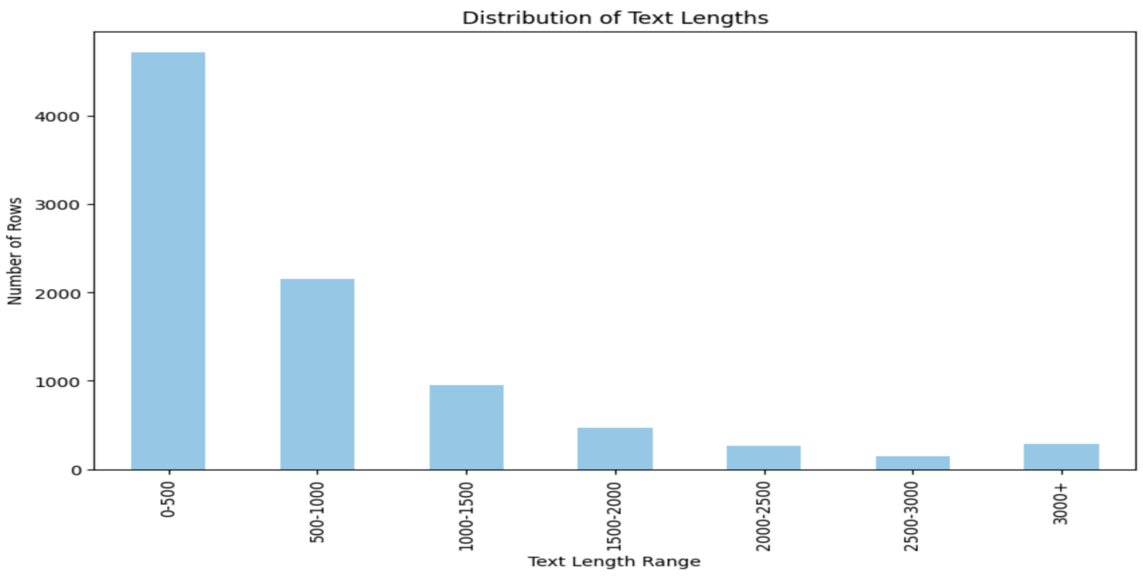
Data exploration

Class distribution



We can observe that in the data some classes occurred a lot more than others. For example class 3 occurred much more than any other.

Text length distribution



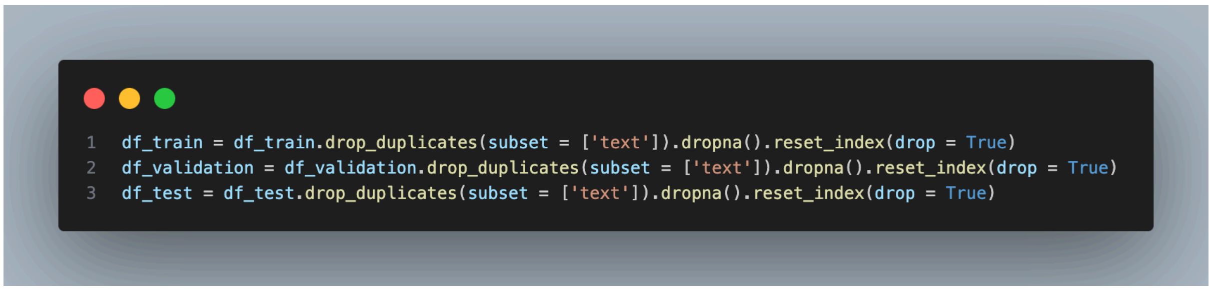
We have limited time and computation resources, so we can utilise this distribution to make decisions while deciding on models, sequence lengths etc.

Here, we can see most sentences are between length 0-1500, hence as a trade-off between compute power, time and accuracy, I decided to go with sequence length of 1500 while fine tuning the model.

Process/ Approach

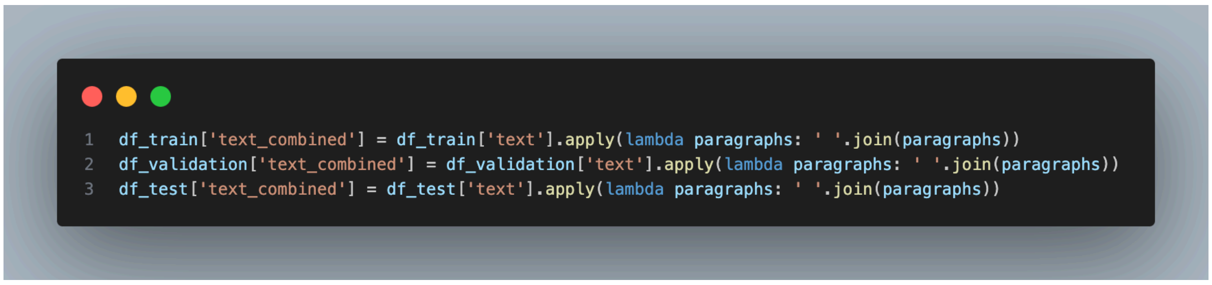
1. **Data Loading and cleaning**

   1.1 Removed duplicates and missing data to ensure data quality.
   1.2 Duplicates Removal: Remove duplicate entries in the dataset based on the 'text' feature.
   1.3 Missing Data Handling: Drop rows containing missing values after duplicate removal.
   1.4 Reset Index: Reset the indices after performing operations to maintain indexing.



```python
df_train = df_train.drop_duplicates(subset = ['text']).dropna().reset_index(drop = True)
df_validation = df_validation.drop_duplicates(subset = ['text']).dropna().reset_index(drop = True)
df_test = df_test.drop_duplicates(subset = ['text']).dropna().reset_index(drop = True)
```

   1.5 Combined paragraphs to get complete documents for each case.
   We concat all strings representing paragraphs of the document to form a complete document. We keep these combined text in column 'text_column'

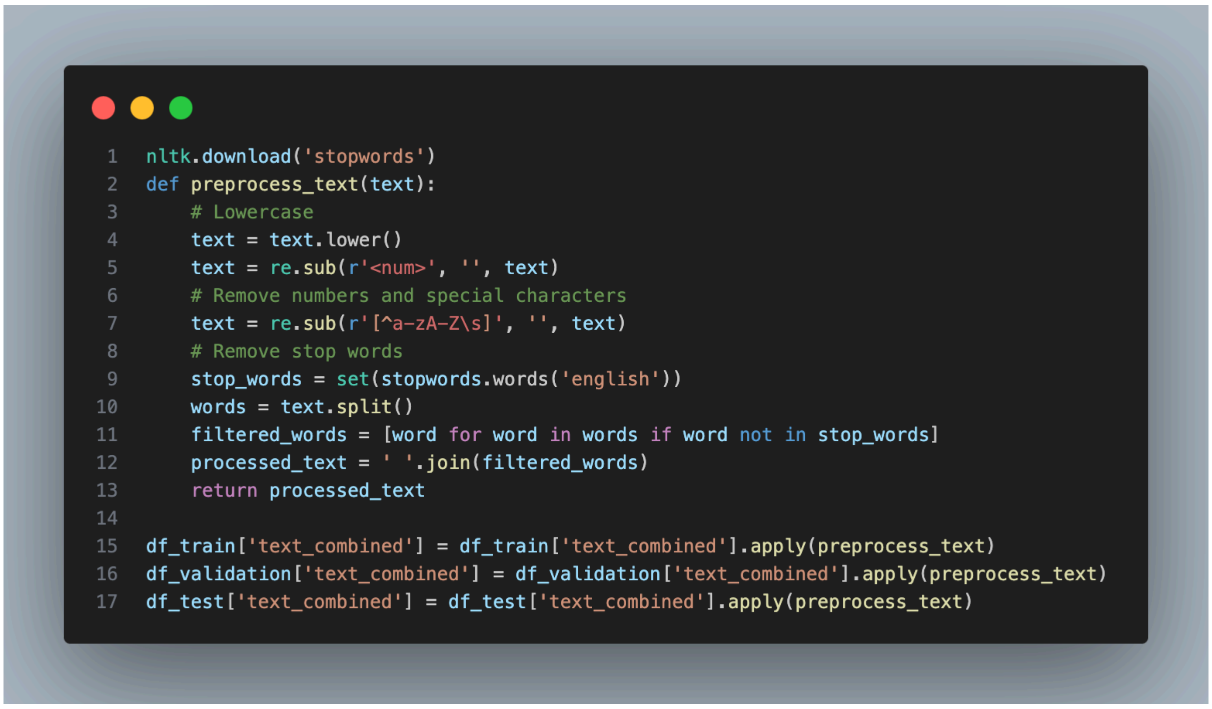


```python
df_train['text_combined'] = df_train['text'].apply(lambda paragraphs: ' '.join(paragraphs))
df_validation['text_combined'] = df_validation['text'].apply(lambda paragraphs: ' '.join(paragraphs))
df_test['text_combined'] = df_test['text'].apply(lambda paragraphs: ' '.join(paragraphs))
```

2. **Performed text preprocessing**

   2.1 Lowercasing: Convert all text to lowercase for uniformity going forward.
   2.2 Removing Numbers: Eliminate numbers and dates to focus on text.
   2.3 Removing Non-English Words: Utilize NLTK's stopwords to filter out non-English words and special characters.

2.4 Stopword Removal: Eliminate stopwords from the text and keep only meaningful content.

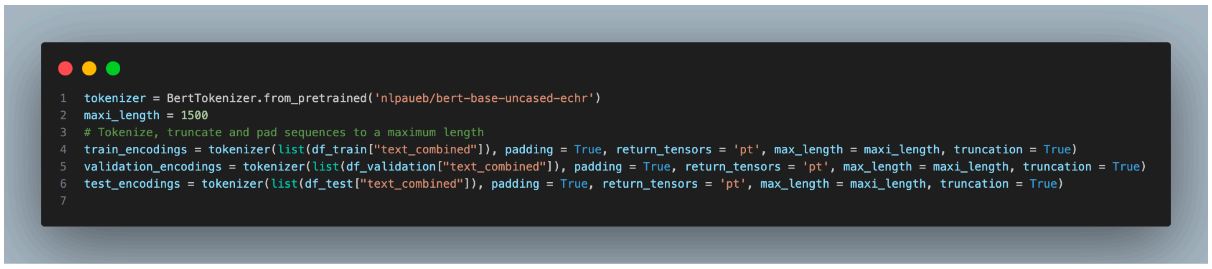


```python
nltk.download('stopwords')
def preprocess_text(text):
    # Lowercase
    text = text.lower()
    text = re.sub(r'<num>', '', text)
    # Remove numbers and special characters
    text = re.sub(r'[^a-zA-Z\s]', '', text)
    # Remove stop words
    stop_words = set(stopwords.words('english'))
    words = text.split()
    filtered_words = [word for word in words if word not in stop_words]
    processed_text = ' '.join(filtered_words)
    return processed_text

df_train['text_combined'] = df_train['text_combined'].apply(preprocess_text)
df_validation['text_combined'] = df_validation['text_combined'].apply(preprocess_text)
df_test['text_combined'] = df_test['text_combined'].apply(preprocess_text)
```

3. Tokenization and Dataset Creation

3.1 Tokenization: Utilized the BERT tokenizer from the transformers library to convert text into tokens.

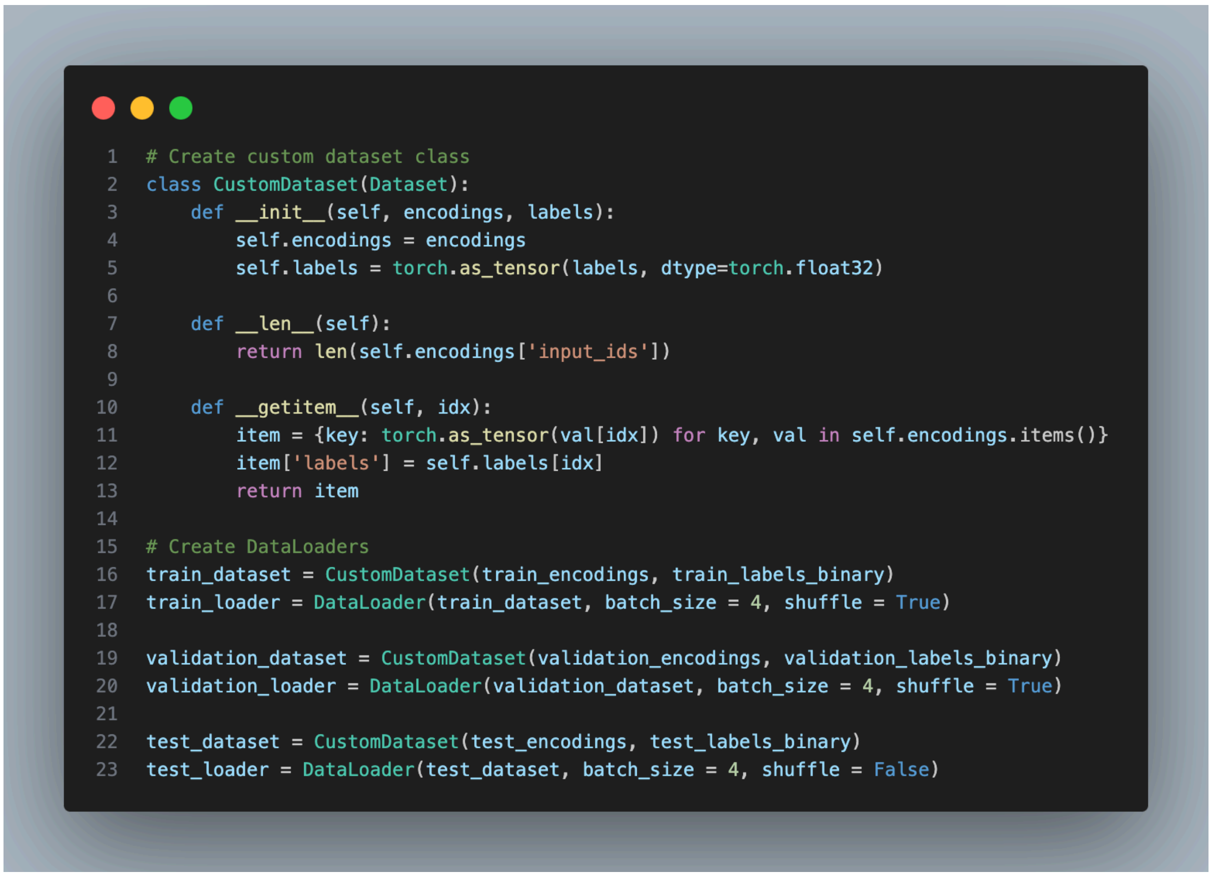
3.2 Padding and Truncation: Apply padding and truncation to make sure that all input sequences have the same length, this fixed length is defined as 1500.



```python
tokenizer = BertTokenizer.from_pretrained('nlpaueb/bert-base-uncased-echr')
maxi_length = 1500
# Tokenize, truncate and pad sequences to a maximum length
train_encodings = tokenizer(list(df_train["text_combined"]), padding = True, return_tensors = 'pt', max_length = maxi_length, truncation = True)
validation_encodings = tokenizer(list(df_validation["text_combined"]), padding = True, return_tensors = 'pt', max_length = maxi_length, truncation = True)
test_encodings = tokenizer(list(df_test["text_combined"]), padding = True, return_tensors = 'pt', max_length = maxi_length, truncation = True)

```

4. Custom datasets and dataloaders

Created custom datasets and data loaders for training, validation, and testing to efficiently load batches of data during training and evaluation.



```python
# Create custom dataset class
class CustomDataset(Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = torch.as_tensor(labels, dtype=torch.float32)

    def __len__(self):
        return len(self.encodings['input_ids'])

    def __getitem__(self, idx):
        item = {key: torch.as_tensor(val[idx]) for key, val in self.encodings.items()}
        item['labels'] = self.labels[idx]
        return item

# Create DataLoaders
train_dataset = CustomDataset(train_encodings, train_labels_binary)
train_loader = DataLoader(train_dataset, batch_size = 4, shuffle = True)

validation_dataset = CustomDataset(validation_encodings, validation_labels_binary)
validation_loader = DataLoader(validation_dataset, batch_size = 4, shuffle = True)

test_dataset = CustomDataset(test_encodings, test_labels_binary)
test_loader = DataLoader(test_dataset, batch_size = 4, shuffle = False)
```

5. Model Initialization

5.1 Architecture

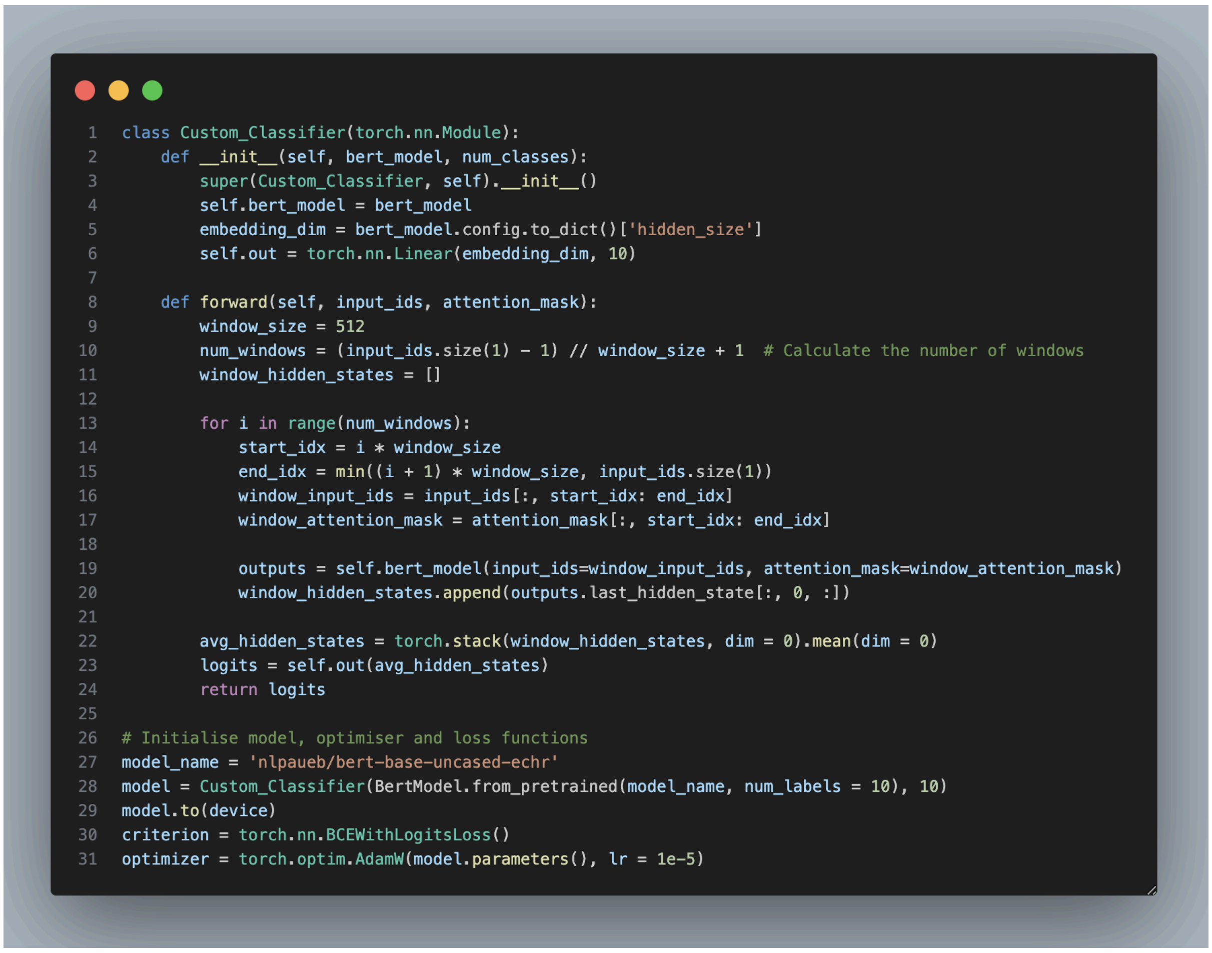
We use pre-trained BERT on legal cases, but since we want to use BERT for classification, we add our own custom linear layer at the end which gives us a result of length equal to the number of classes.

Since the sequences are much larger than 512 which BERT allows to give as an input, we run a 512 word sized window on the input sequence and get output for each 512 words. Then we take the output of all these windows and take their average.

Then the average is passed on to the classifier layer which gives us logits for the input.

5.1 Initialization:

Initialized the model, optimizer, and loss function. And since we saw some classes occurring more than others, we gave some extra weightage to the other classes.



```python
class Custom_Classifier(torch.nn.Module):
    def __init__(self, bert_model, num_classes):
        super(Custom_Classifier, self).__init__()
        self.bert_model = bert_model
        embedding_dim = bert_model.config.to_dict()['hidden_size']
        self.out = torch.nn.Linear(embedding_dim, 10)

    def forward(self, input_ids, attention_mask):
        window_size = 512
        num_windows = (input_ids.size(1) - 1) // window_size + 1  # Calculate the number of windows
        window_hidden_states = []

        for i in range(num_windows):
            start_idx = i * window_size
            end_idx = min((i + 1) * window_size, input_ids.size(1))
            window_input_ids = input_ids[:, start_idx: end_idx]
            window_attention_mask = attention_mask[:, start_idx: end_idx]

            outputs = self.bert_model(input_ids=window_input_ids, attention_mask=window_attention_mask)
            window_hidden_states.append(outputs.last_hidden_state[:, 0, :])

        avg_hidden_states = torch.stack(window_hidden_states, dim = 0).mean(dim = 0)
        logits = self.out(avg_hidden_states)
        return logits

# Initialise model, optimiser and loss functions
model_name = 'nlpaueb/bert-base-uncased-echr'
model = Custom_Classifier(BertModel.from_pretrained(model_name, num_labels = 10), 10)
model.to(device)
criterion = torch.nn.BCEWithLogitsLoss()
optimizer = torch.optim.AdamW(model.parameters(), lr = 1e-5)
```

6. Training

Trained the model over multiple epochs, monitoring training loss, accuracy, and making use of class weights for imbalanced classes.
Validated the model performance on a separate validation set to assess generalization.

7. Testing and Evaluation

Tested the trained model on the test set and evaluated performance using metrics such as accuracy, precision, recall, and F1 score. Additionally, eported classification results.

8. Inference

Implemented an inference function to predict labels for new legal texts.

Challenges Faced, Choices and Solutions

Class Imbalance

Observation:
During the data exploratory part of the process, we observed that in the data some classes occurred a lot more than others. For example class 3 occurred much more than any other.

Decision/ Solution:
To prevent the model to just focus more on that label and just give the most common label as answer, in the loss function, we added class weights based on the number of occurrences of classes. This would prevent the model from learning to just push out the most common class as an answer every time. Although the weighted classes work in theory for our particular case, hyperparameters and dataset, it didn't lead to any substantial result improvement.

Text Lengths

Observation:
Document text was huge. Documents ranged from a few hundreds to 5000+. This would cause problems with our pretrained model, BERT, since BERT only allows input sequences of max length 512 words. Average length of the text was around 1000 - 1500 after removal of stop words.

Decision/ Solution:
We had few choices, to use a model that can handle these long sequences such as BigBird or Longformer, but using these models crashed runtime on colab several times suggesting their need for higher computing power.

Then I just use BERT and let it truncate sequences at 512 words. It did give good results, but according to some research, it was shown that just truncating the last part of the sequence may cause loss of a lot of information and context, hence I kept searching for a better solution.

Then I decided to go with a windowed approach, in which I give input to BERT in 512 words and then move the window to next 512 words and do the same. All the outputs obtained from each window are averaged to get a single output. This output is passed through the classifier layer to get logits. This approach gave the best result. But I had to make some sacrifices due to time and computation limitations and work with a max length of 1500. This length covered around 75% of the sequences, which was good enough for our task.

Some other decisions

Loss function

I chose BCEWithLogits loss function since our task involves predicting multiple labels for each document (ie. multi-label classification). It independently computes the binary cross-entropy loss for each label, allowing the model to handle multiple labels for a single instance. Additionally, BCEWithLogits loss function supports the use of class weights, which can be used when there is class imbalance

Optimizer

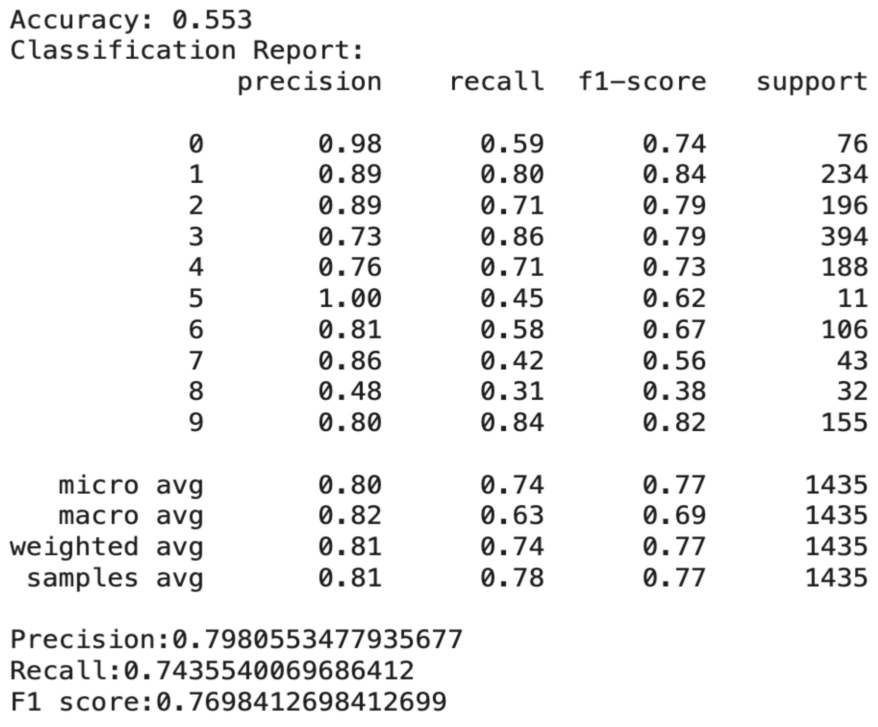
I choose AdamW because it combines the benefits of momentum like SGD with momentum and adaptive learning rate. This combination enables faster convergence along relevant dimensions and slower updates along noisy dimensions.

Findings and Results

The model achieved satisfactory performance on the validation and test sets, giving nearly similar performance as given in benchmarks on huggingface demonstrating its ability to classify legal texts into the relevant violated articles of the ECHR.
The preprocessing steps effectively handled noise in the legal texts, and the window-based approach in the model addressed challenges related to long text lengths.

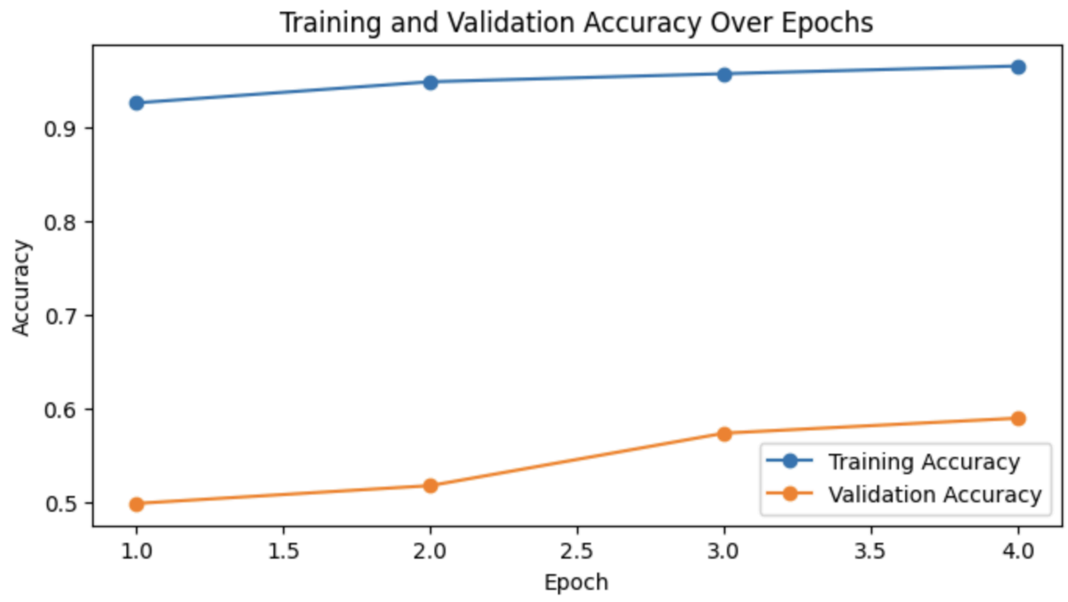
Compare Pretrained V Finetuned BERT

In fine tuned, we finetune both the newly added classifier layer and the BERT weights as well. Here are some results -
- Classification report

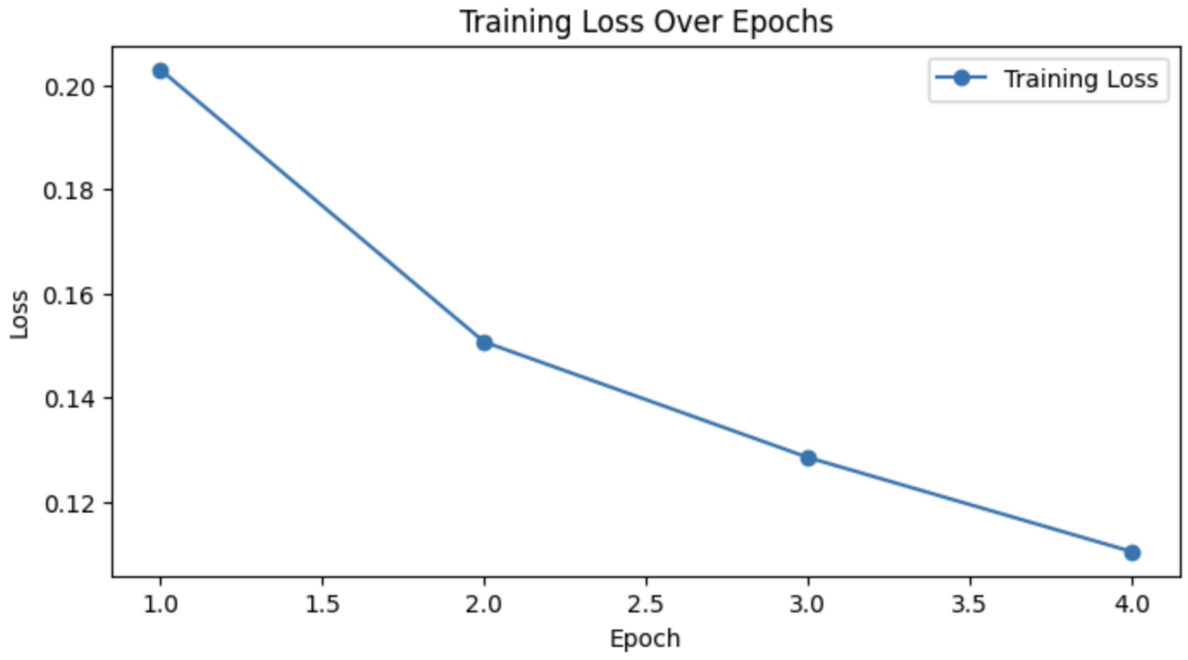


```
Accuracy: 0.553
Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.59      0.74        76
           1       0.89      0.80      0.84       234
           2       0.89      0.71      0.79       196
           3       0.73      0.86      0.79       394
           4       0.76      0.71      0.73       188
           5       1.00      0.45      0.62        11
           6       0.81      0.58      0.67       106
           7       0.86      0.42      0.56        43
           8       0.48      0.31      0.38        32
           9       0.80      0.84      0.82       155

   micro avg       0.80      0.74      0.77      1435
   macro avg       0.82      0.63      0.69      1435
weighted avg       0.81      0.74      0.77      1435
 samples avg       0.81      0.78      0.77      1435

Precision:0.7980553477935677
Recall:0.7435540069686412
F1 score:0.7698412698412699
```

- Fine-tuned accuracies

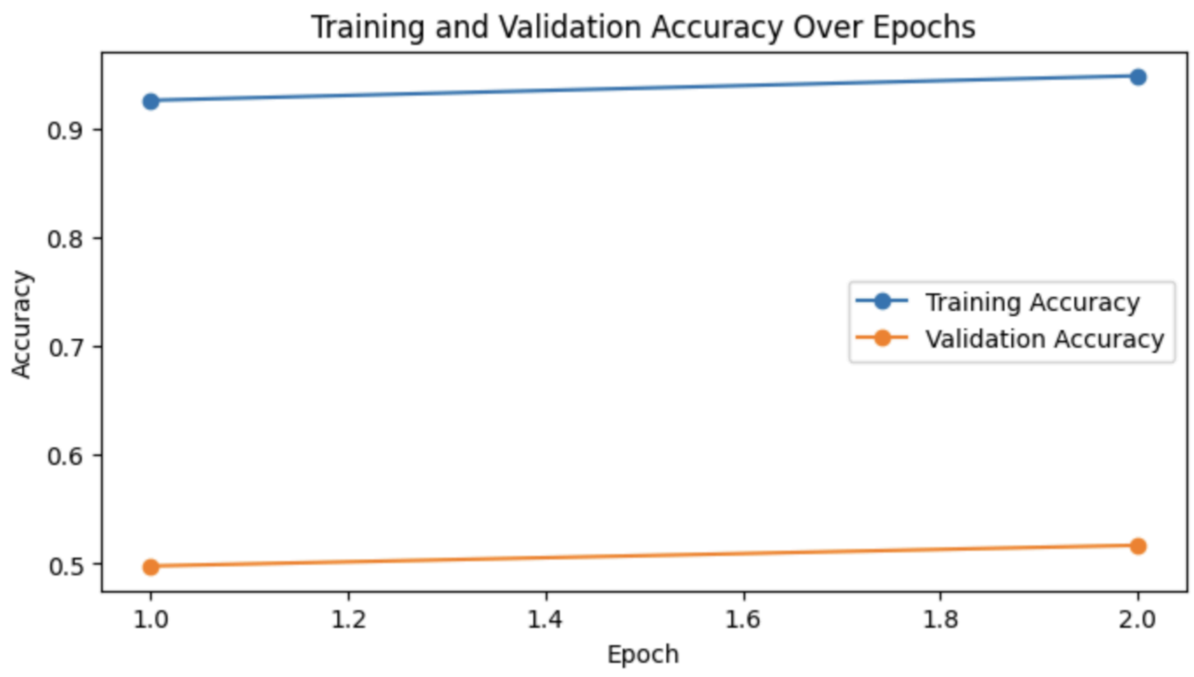


- Fine-tuned training loss per epoch

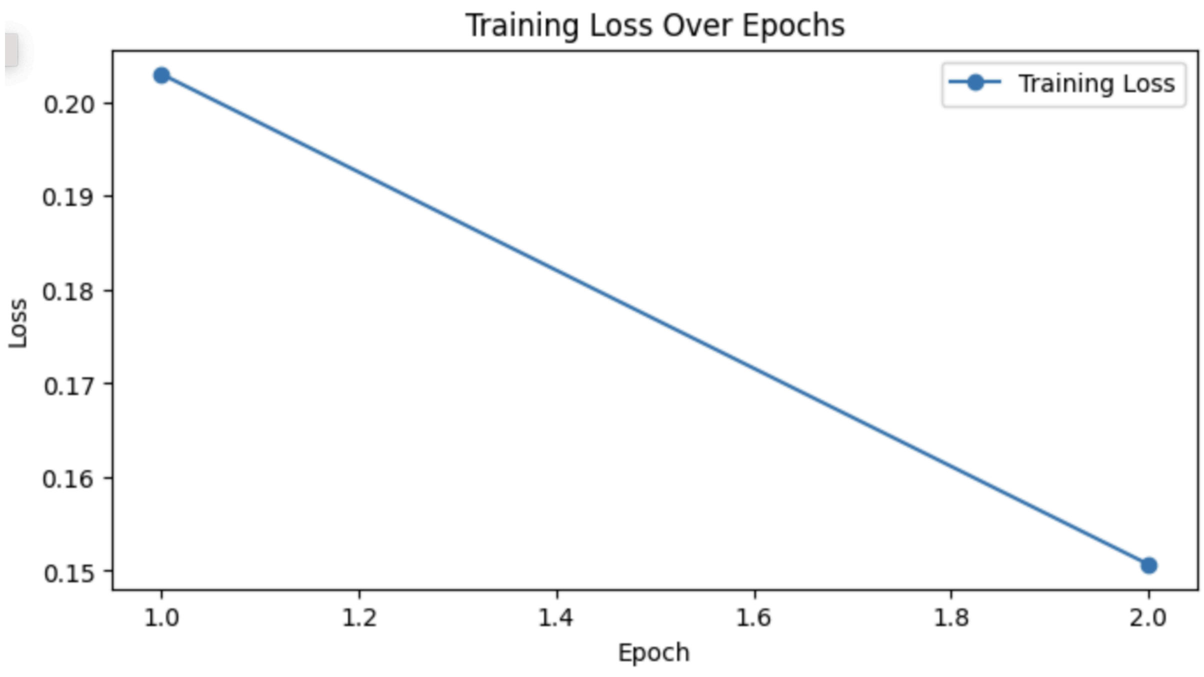


In pretrained BERT, we just a classifier layer and trained that layer only while keeping BERT layers frozen. Here are some results -

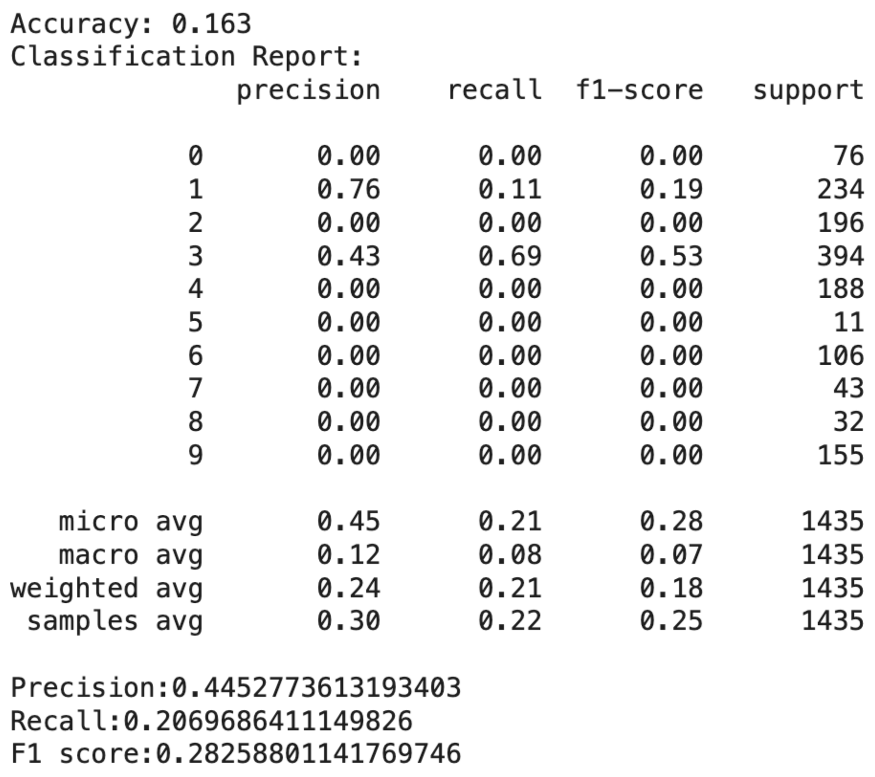
- Pretrained Accuracies



- Pretrained Training loss per epoch

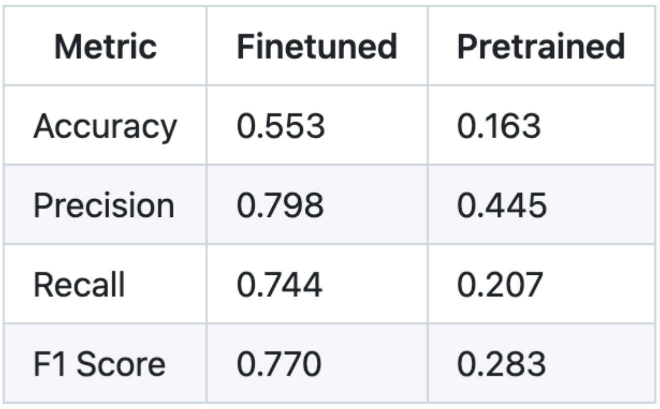


- Classification report



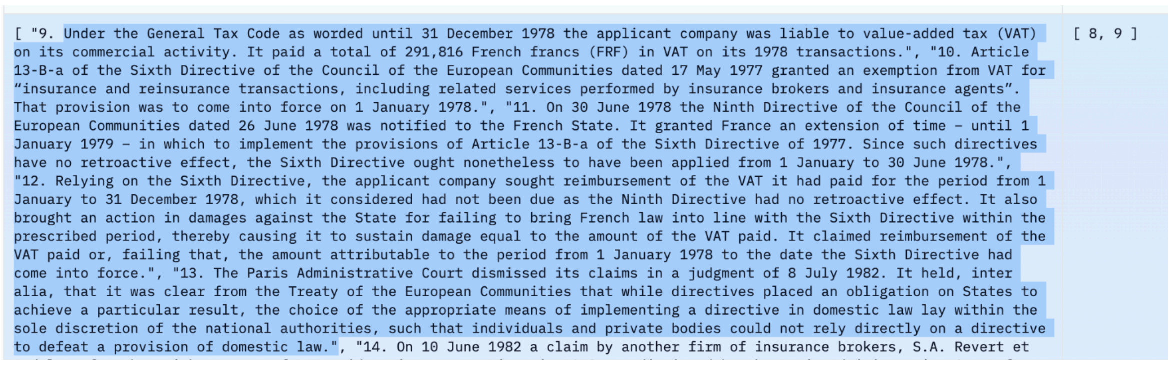
```
Accuracy: 0.163
Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        76
           1       0.76      0.11      0.19       234
           2       0.00      0.00      0.00       196
           3       0.43      0.69      0.53       394
           4       0.00      0.00      0.00       188
           5       0.00      0.00      0.00        11
           6       0.00      0.00      0.00       106
           7       0.00      0.00      0.00        43
           8       0.00      0.00      0.00        32
           9       0.00      0.00      0.00       155

   micro avg       0.45      0.21      0.28      1435
   macro avg       0.12      0.08      0.07      1435
weighted avg       0.24      0.21      0.18      1435
 samples avg       0.30      0.22      0.25      1435

Precision:0.4452773613193403
Recall:0.2069686411149826
F1 score:0.28258801141769746
```

Comparison



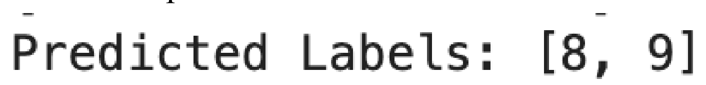
Metric	Finetuned	Pretrained
Accuracy	0.553	0.163
Precision	0.798	0.445
Recall	0.744	0.207
F1 Score	0.770	0.283

Sample inference



```
[ "9. Under the General Tax Code as worded until 31 December 1978 the applicant company was liable to value-added tax (VAT)     [ 8, 9 ]
on its commercial activity. It paid a total of 291,816 French francs (FRF) in VAT on its 1978 transactions.", "10. Article
13-B-a of the Sixth Directive of the Council of the European Communities dated 17 May 1977 granted an exemption from VAT for
“insurance and reinsurance transactions, including related services performed by insurance brokers and insurance agents”.
That provision was to come into force on 1 January 1978.", "11. On 30 June 1978 the Ninth Directive of the Council of the
European Communities dated 26 June 1978 was notified to the French State. It granted France an extension of time – until 1
January 1979 – in which to implement the provisions of Article 13-B-a of the Sixth Directive of 1977. Since such directives
have no retroactive effect, the Sixth Directive ought nonetheless to have been applied from 1 January to 30 June 1978.",
"12. Relying on the Sixth Directive, the applicant company sought reimbursement of the VAT it had paid for the period from 1
January to 31 December 1978, which it considered had not been due as the Ninth Directive had no retroactive effect. It also
brought an action in damages against the State for failing to bring French law into line with the Sixth Directive within the
prescribed period, thereby causing it to sustain damage equal to the amount of the VAT paid. It claimed reimbursement of the
VAT paid or, failing that, the amount attributable to the period from 1 January 1978 to the date the Sixth Directive had
come into force.", "13. The Paris Administrative Court dismissed its claims in a judgment of 8 July 1982. It held, inter
alia, that it was clear from the Treaty of the European Communities that while directives placed an obligation on States to
achieve a particular result, the choice of the appropriate means of implementing a directive in domestic law lay within the
sole discretion of the national authorities, such that individuals and private bodies could not rely directly on a directive
to defeat a provision of domestic law.", "14. On 10 June 1982 a claim by another firm of insurance brokers, S.A. Revert et
```

Inference Output



```
Predicted Labels: [8, 9]
```

Future scope/ Improvements

Since we ran our model within a computation and time limited environment, we were restricted to only a few experimentations. With a more powerful machine, we can try out different learning rates, higher epochs, bigger batch sizes etc.

Conclusion

This documentation outlines the comprehensive steps taken to develop a legal text multilabel classification model. The implemented approaches, preprocessing steps, and model architecture aim to address specific challenges associated with legal texts. The achieved results provide insights into the model's effectiveness in predicting violations of articles in the ECHR. The documentation serves as a guide for understanding the code, algorithms, and reasoning behind the decisions made during the development process.