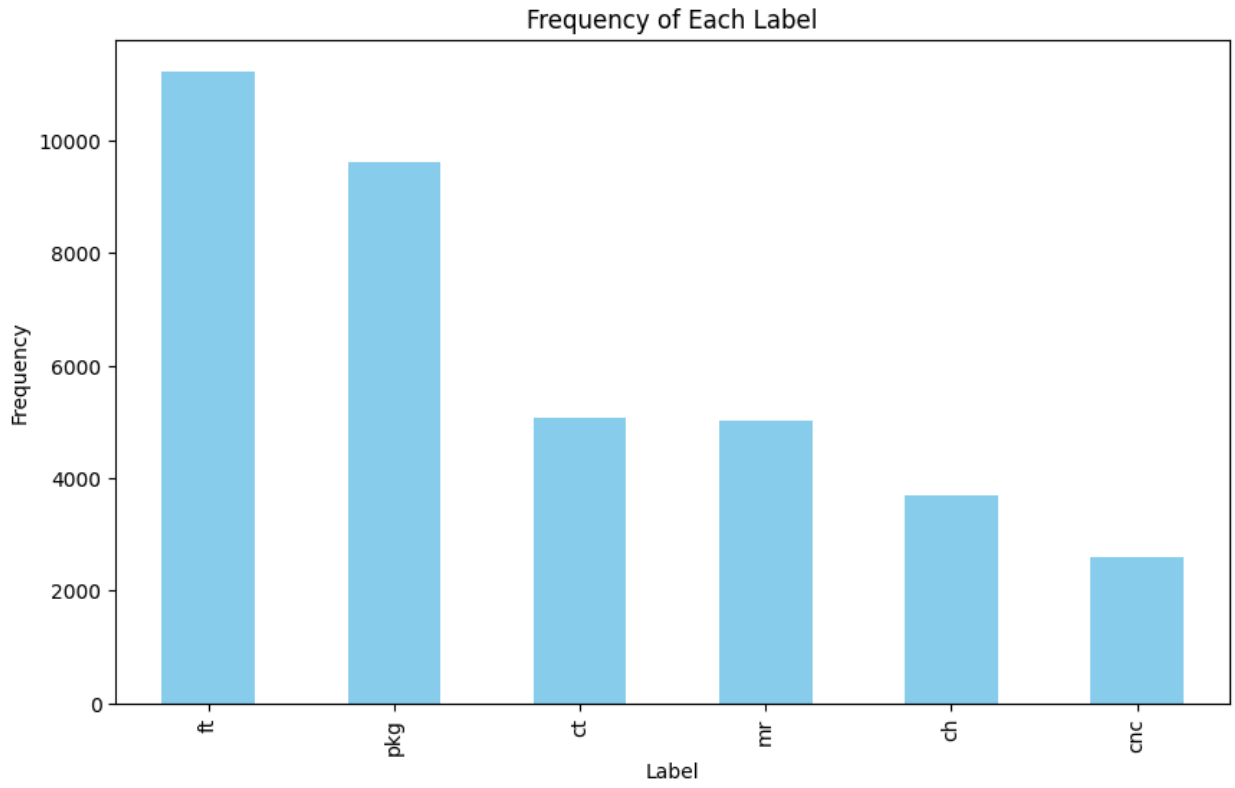
Task:

Develop and deploy a machine learning model to classify German text phrases into predefined categories

Overview of data:

The data has 37295 lines with 2 column, a text column and a label. There are 6 labels in total. Following is the class distribution in data:



There is not much class imbalance since three is a linear increase in class frequency. The absence of a significant difference in frequency between adjacent classes can contribute to a reduction in class imbalance.

Preprocessing (preprocessing.ipynb):

Fields that had missing label or test were removed. Fields with special characters such as emails were kept since they could provide meaningful information.

The data is cleaned and put in a csv file named “cleaned_data” as performed in the “preprocessing.py” file.

A word cloud is generated to gain insights to the data and see the most frequent words in the phrase.

There are chinese words in the data. Even thorough they are very small in number compared to the german phrases but since the task is to classify german phrases i chose to remove these fields. The words could not be detected by the langid and langdetect library, the words came to my observation upon going through the data manually.

36595 rows remain after removing empty fields and chinese words.

Model training:

A pretrained language model named “DistilBERT” was finetuned through the training data and then saved to be later tested on testing data.

DistilBERT was used for a couple of reasons:
It's a transformer-based model designed for NLP tasks and performs well in understanding and classifying text data. Additionally, it works for the German language, which aligns with our task requirements. The model can capture semantic relationships and contextual information from text, making it suitable for tasks like classifying texts based on emotion. However, since our data doesn't indicate the criteria for phrase classification, the need for such capabilities may vary. DistilBERT benefits from transfer learning, enabling me to fine-tune the pre-trained model on the provided dataset. Unlike traditional models, DistilBERT eliminates the need for extensive feature engineering as it automatically learns meaningful representations

While other state-of-the-art algorithms such as Naive Bayes, and SVM are viable options, each has its own set of limitations. Naive Bayes assumes feature independence, which may not hold

true for text classification tasks where contextual relationships matter. Support Vector Machines (SVM) are effective but might not capture complex linguistic nuances. Bag of Words (BoW) relies on fixed features based on word occurrences, potentially limiting its performance in tasks requiring a nuanced understanding of language. In this context, employing the DistilBERT model appeared to be a more robust and promising approach for my text classification problem.

Here's how the model training works (BARTmodel.py):

- It loads a cleaned dataset from a CSV file
- Encodes labels using a LabelEncoder
- Splits the data into training (80%) and testing sets (20%).
- Loads a smaller pre-trained DistilBERT model and tokenizer for the German language.
- Tokenizes and encodes the training data with a specified sequence length using the DistilBERT tokenizer.
- Saves the training and testing data with encoded labels to CSV files.
- Defines the DistilBERT model for sequence classification with a specified number of output labels.
- Sets up data loaders for training using PyTorch's DataLoader.
- Trains the model using the AdamW optimizer with gradient accumulation over a defined number of epochs.
- Saves the trained DistilBERT model and the label encoder to files.

The code allows flexibility in adjusting parameters such as sequence length, batch size, learning rate, and epochs.

Testing of the model (testing.py):

Here's what the testing.py file performs:

- Reads the test data from a CSV file (test_data.csv).
- Loads the fine-tuned DistilBERT model and label encoder saved during training.
- Iteratively tokenizes and encodes batches of test text data using the DistilBERT tokenizer.
- Performs inference using the loaded model to obtain predictions for each batch.
- Decodes the model predictions and actual labels using the loaded label encoder.
- Appends the decoded predictions and actual labels to the test data.
- Saves the test data with predictions and actual labels to a new CSV file (predictions.csv).
- Calculates and print various classification metrics, including accuracy, precision, recall, and F1 score..

```
Accuracy: 0.8737
Precision: 0.8781
Recall: 0.8737
F1 Score: 0.8743
Classification Report:
              precision    recall  f1-score   support

          ch       0.91      0.83      0.87       689
         cnc       0.80      0.72      0.76       504
          ct       0.90      0.87      0.89       971
          ft       0.94      0.88      0.91      2238
          mr       0.75      0.89      0.81       934
         pkg       0.86      0.91      0.88      1923

    accuracy                           0.87      7259
   macro avg       0.86      0.85      0.85      7259
weighted avg       0.88      0.87      0.87      7259
```

Development (main.py):

Application Setup:

- The FastAPI API is created.
- CORS middleware is added to allow cross-origin requests, allowing the frontend to interact with the backend.

Model Loading:

- A fine-tuned DistilBERT model, tokenizer, and label encoder are loaded into the application.

Data Models:

- Two Pydantic models (Item and PredictionResult) are defined to handle input data and prediction results.

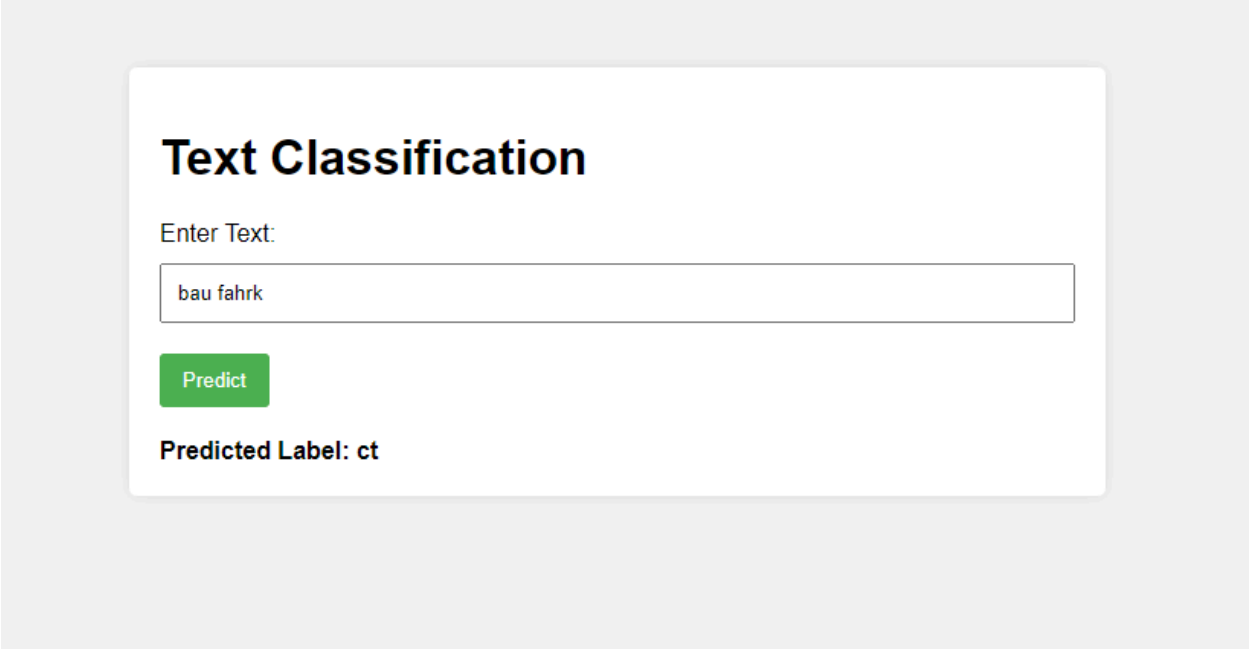
Root Endpoint:

- A root endpoint ("/") is set up to return a simple greeting message and instruct users to open the HTML code to start predictions.

Prediction Endpoint:

- A POST endpoint ("/predict") is implemented to receive text input, tokenize it using DistilBERT, and make predictions using a pre-trained model.
- The predicted label is decoded using a label encoder, and the result is returned in the response.

Additionally, there is a basic HTML, CSS, and JavaScript code for the frontend. This code receives the input of a string through a designated field, posts it to the API for prediction, and then displays the response.



Coverage Testing:

The provided tests aim to check the functionality and behavior of specific endpoints, handling of different inputs, and responses to ensure that the API behaves as expected. By executing these tests, we can determine the extent to which uur code is covered by the test cases.

Test Case 1 (test_read_root):
Checks if the root endpoint ("/") is accessible and returns a valid response.

Test Case 2 (test_invalid_endpoint):
Verifies the behavior when trying to access an invalid endpoint.

Test Case 3 (test_complete_request_cycle):
Tests the complete cycle of making a prediction request with a sample input ("bau fahrk").

Test Case 4 (test_large_input):
Assesses the server's handling of a large input string (10,000 characters).

Note:
The test case for null input is not explicitly performed, as the server-side check for empty input is implemented on the client side, preventing null input from reaching the server.

```
============================= test session starts ==============================
collecting ... collected 4 items

test_cases.py::test_read_root PASSED                                     [ 25%]
test_cases.py::test_invalid_endpoint PASSED                              [ 50%]
test_cases.py::test_complete_request_cycle PASSED                        [ 75%]
test_cases.py::test_large_input PASSED                                   [100%]

============================== 4 passed in 7.51s ===============================
```

Suggestions for future work and improvements if I had more time:

If I had more time, I'd explore using word embeddings, which are like super-smart word representations, to make the model even better at understanding the context of our text. Also, I'd compare our current model with some older models like decision trees or Support Vector Machines to see if there's a better fit for our problem. This would help me choose the right tool for the job and improve the text classification even more. I did obtain word embeddings and tried to use it with BERT model but it was computationally expensive for me and i couldn't find a way around to get it done in the given time.

Below is a snippet of the word embeddings i got:

	text	label	word_vectors	encoded_label
1	zulieferer für leben...	pkg	[0.01706123, 0.034633584, 0.104049675, 0.45882326, 0...	5
2	prawn craker	ft	[-0.25382102, 0.107287884, 0.4206393, -0.49894556, 0....	3
3	Silikon vakuumguss	ch	[-0.14916055, -0.4290634, -0.10622405, -0.08992568, 0....	0
4	erdbohrgeräte geb...	ct	[-0.54014397, 0.17828512, -0.28843042, 0.363127, 0.14...	2
5	pharmazeutische c...	ch	[0.7725972, 0.14039369, 0.5581987, -0.08985862, -0.25...	0
6	dynamische berech...	ct	[-0.7791578, -0.30780917, -0.2659576, -0.5631037, 0.17...	2
7	fruchtgummis nahr...	ft	[-0.15474533, -0.33027124, 0.028239017, 0.78858316, -...	3
8	kisten aus PUR-Kalt...	pkg	[-0.41981384, -0.122550935, -0.26658395, 0.2515771, 0...	5
9	Montagen von Alu...	ct	[-0.5087983, -0.0077935075, 0.0003078218, 0.7983208,...	2
10	bioaktive peptide	ch	[-0.37216717, -0.50164956, 0.24973151, 0.54452133, 0....	0
11	rako kisten 40x30...	pkg	[-0.6962658, -0.6359233, 0.29939133, 0.04211502, 0.42...	5
12	Parkbank fässer au...	pkg	[-0.41428667, 0.02696119, 0.037465952, 0.17521632, 0....	5
13	u-profil kunststoff	pkg	[-0.6968918, -0.16078167, -0.102691926, -0.23310177,...	5
14	herstellen papierta...	pkg	[-0.5092007, -0.40659863, 0.43535772, 0.04161539, 0.6...	5
15	folien beutel karton	pkg	[-0.7380489, -0.9186477, -0.0793767, 0.6428528, -0.138...	5
16	italienische weine...	ft	[-0.549814, -0.5781118, 0.18003201, 0.8175701, -0.104...	3
17	enzyme fleischwaren	ft	[-0.2569416, -0.28991273, -0.38639402, 0.52388304, -0....	3
18	Edelstahl schwingg...	mr	[-0.45647553, -0.027015833, -0.21721809, -0.01291308...	4