1. INTRODUCTION:

Welcome to the documentation for the German Phrases Classifier project. The objective of this project is to build an end-to-end machine-learning pipeline to classify German search queries into predefined classes. The entire process involves training a model on a provided dataset and deploying the final model using a REST API. The deployment process is dockerized for seamless deployment and scalability.

Problem Statement
The primary goal is to develop a robust and accurate classifier for German search queries. The provided dataset includes a CSV file with two columns: "text" and "label." The "text" column contains short German phrases representing search queries, and the "label" column contains their associated classes.

Key Tasks

- Train a machine learning model to classify German search queries accurately.
- Implement a REST API for model deployment.
- Dockerize the deployment process for easy scalability and reproducibility.
- Proper preprocessing steps will be taken to handle language-specific challenges such as tokenization and stemming.
- The model's performance will be evaluated based on standard classification metrics.
- The REST API will be designed to handle incoming search queries and return predicted classes.

Project Overview
The project involves multiple stages, including data understanding, preprocessing, model training, and deployment. Each stage will be discussed thoroughly to provide transparency into the decision-making process and methodology.

Let's dive into each section for a detailed exploration of the project.

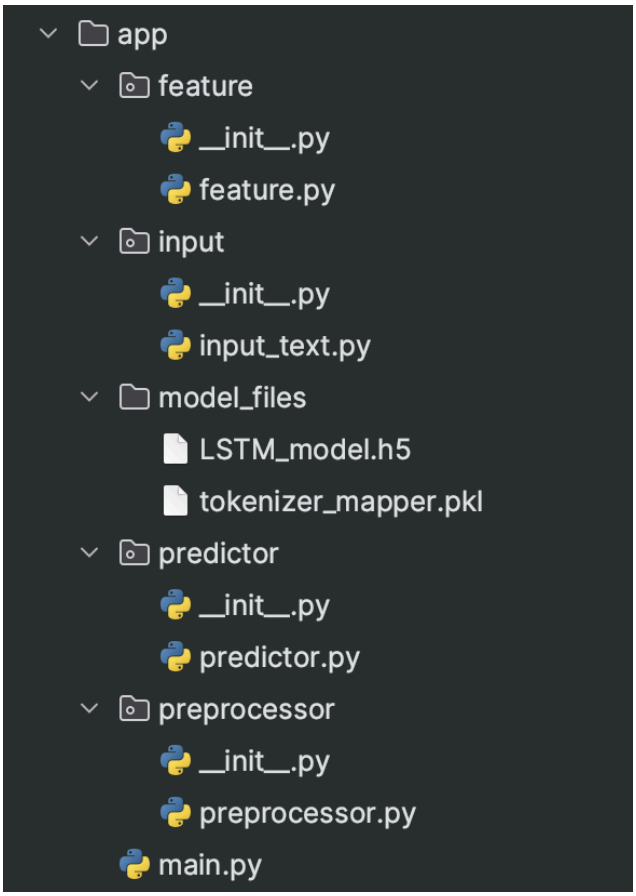
2. CODE STRUCTURE:

The code is organized into different modules, each responsible for specific functionalities. The project is structured to ensure modularity, clarity, and maintainability. Below is an overview of the code structure:

1. **.github/workflows/docker-image.yml:**

- This file is part of GitHub Actions and is responsible for automating the Docker image creation and deployment process.
- It defines the workflow for building and pushing the Docker image whenever changes are made to the repository.

2. **app Folder:**
   This folder contains all the application-related files, including the Main driver file, Predictor script, preprocessors, feature mappers, model files and the REST API implementation.
   Folder Structure:



3. **Dockerfile:**
   The Dockerfile is used to define the Docker image configuration. It specifies the base image, installs dependencies, and sets up the environment for running the application.

4. **Modeling.ipynb:**

This Jupyter notebook contains the code for data exploration, preprocessing, and model training. It serves as a comprehensive guide to understanding the modelling process.

5. **requirements.txt:**
   This file lists all the Python dependencies required for the project. It ensures consistency in the development and deployment environments.

This code structure provides a clear separation of concerns, making the project modular and maintainable. The Docker image automates deployment, ensuring a consistent environment for the application. The Jupyter Notebook serves as a detailed guide to the modelling process, and the requirements.txt file manages dependencies. The app folder contains various modules, each responsible for specific tasks within the application.

3. DATA UNDERSTANDING AND PREPROCESSING:

The data understanding and preprocessing steps aim to prepare the German search query dataset for effective model training. The preprocessing steps involve cleaning and transforming the text data to enhance the quality and relevance for subsequent modeling.

Data Understanding:

Modeling.ipynb:

1. Dataset Description:
   The dataset consists of two columns: "text" representing German search queries and "label" indicating their associated classes.
2. Class distribution:

Label	% contribution
ft	30.18%
pkg	25.18%
ct	13.60%
mr	13.48%
ch	9.91%

cnc	6.95%

Data Preprocessing:
The Preprocessor class is designed to handle the preprocessing tasks. The following steps are performed on the "text" column:

1. Remove Unwanted Text:
    - Removal of email addresses, HTML tags, URLs, and digits.
    - Cleaning the text by eliminating unwanted elements.
2. Remove Special Characters:
    - Define a regular expression pattern to retain only German letters (äöüßÄÖÜ), digits, and whitespace.
    - Replace non-matching characters with an empty string.
3. Preprocess German Text:
    - Convert the text to lowercase.
    - Tokenization of the German text.
    - Removal of stopwords (common words without significant meaning).
    - Stemming using the SnowballStemmer for linguistic normalization.
    - Filter out single letters if they are standalone tokens.
4. Remove Empty or Short Text:
    - Remove rows with empty text or text with a length less than or equal to 1.
    - Fill empty text with the label 'OTHERS' to handle any unseen text, since these data had prediction so using them for rare label handling.
5. Final Cleanup:
    - Remove leading and trailing whitespaces.
    - Reset the index to ensure consistency.

This comprehensive data understanding and preprocessing approach sets the foundation for building a robust and effective German search query classifier. The cleaned and transformed data will contribute to the model's ability to learn meaningful patterns and make accurate predictions.

4. MACHINE LEARNING MODEL:

1. **Part A:**
   In this section, we focus on the initial steps of preparing the text data for machine learning, including tokenization, creating word indices, and generating sequences.

Additionally, we create a pickle file to store the tokenizer for future use that can be mapped to the new inputs.

1. MAX_SEQUENCE_LENGTH: The decision to set the maximum sequence length is based on the analysis of the maximum number of words in each text. In this case, it is determined to be 75.
2. MAX_NB_WORDS: The maximum number of words to be used is chosen based on the most frequent word in the dataset. This decision helps balance computational efficiency with the inclusion of relevant vocabulary.
3. Tokenizer Initialization: The Tokenizer class from Keras is initialized with the specified parameters.
4. Fit Tokenizer: The tokenizer is fit on the text data to create a word index.
5. Word Index: The word index is obtained from the fitted tokenizer, representing a mapping of words to indices.
6. Save Tokenizer to Pickle File: The tokenizer is saved to a pickle file (tokenizer_mapper.pkl) for future use during model deployment or predictions.
7. Convert Text to Sequences: The text data is converted to sequences using the fitted tokenizer.
8. Pad Sequences: To ensure consistent sequence length, padding is applied to the sequences.

This completes the first part of preparing the text data for machine learning. The next steps involve the train-test split, model creation, and training

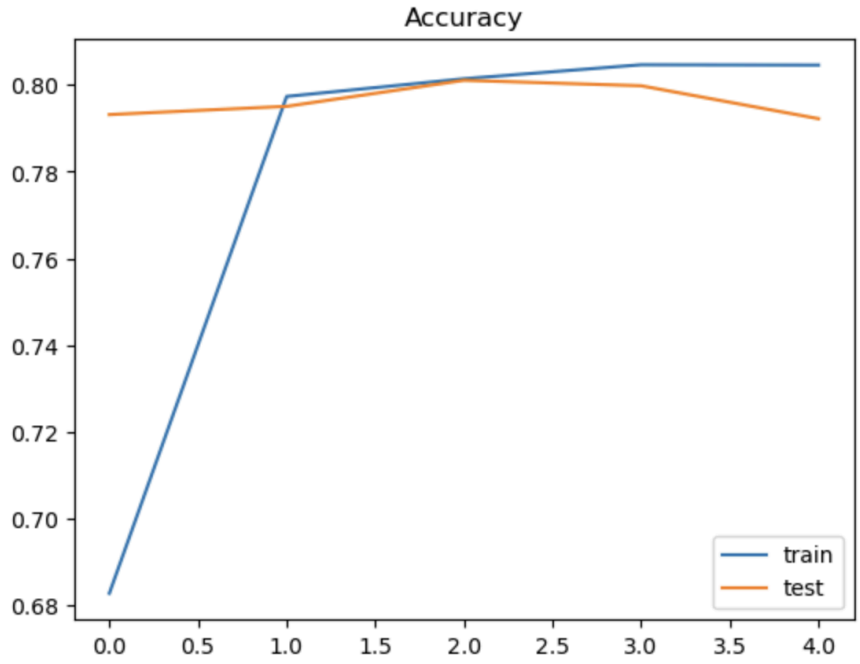
2. **Part B:**
   Now, we continue with the second part of the machine learning model, which involves preparing the output labels, performing train-test split, building the LSTM model, training the model, and evaluating its performance.

   1. Output Label: The labels are one-hot encoded using pd.get_dummies(data['label']) and stored in Y.
   2. Train-Test Split: The dataset is split into training and testing sets using train_test_split.
   3. LSTM Model: The LSTM model is defined using Keras Sequential API, consisting of an Embedding layer, SpatialDropout1D layer, LSTM layer, and a Dense output layer with softmax activation.
   4. Model Compilation: The model is compiled with categorical cross-entropy loss and Adam optimizer.

5. Model Training: The model is trained using the training data with **class weights for handling class imbalance.** The training is monitored for early stopping based on validation loss.

```
Epoch 1/5
445/445 [==============================] - 35s 77ms/step - loss: 0.9148 - accuracy: 0.6911 - val_loss: 0.4855 - val
_accuracy: 0.8134
Epoch 2/5
445/445 [==============================] - 35s 78ms/step - loss: 0.4870 - accuracy: 0.8149 - val_loss: 0.4803 - val
_accuracy: 0.8052
Epoch 3/5
445/445 [==============================] - 35s 79ms/step - loss: 0.4588 - accuracy: 0.8218 - val_loss: 0.4675 - val
_accuracy: 0.8065
Epoch 4/5
445/445 [==============================] - 35s 79ms/step - loss: 0.4464 - accuracy: 0.8235 - val_loss: 0.4727 - val
_accuracy: 0.8106
Epoch 5/5
445/445 [==============================] - 35s 79ms/step - loss: 0.4380 - accuracy: 0.8238 - val_loss: 0.4635 - val
_accuracy: 0.8061
```

6. Model Evaluation: The trained model is evaluated on the test set, and the loss and accuracy are displayed. Accuracy on test set is 80%
7. Save model to a File: The modelis saved to a file (model.save('LSTM_model.h5') for future use during model deployment or predictions.
8. Accuracy Graph:



This completes the second part of the machine learning model, including training and evaluation.

5. INFERENCE PIPELINE:

Execution Flow

1. Input:
   The main driver file receives a list of text inputs.

2. Preprocessing:
   The Preprocessor class is invoked to preprocess the raw text data, handling unwanted text, special characters, and other preprocessing steps.

3. Feature Mapping:
   The preprocessed data is passed to the FeatureMapper class, which utilizes the pre-trained tokenizer to convert text into numerical features.

4. Prediction:
   - The Predictor class loads the pre-trained LSTM model and tokenizer.
   - The preprocessed data is passed to the LSTM model for prediction.
   - The predicted probabilities are converted into class labels.
   - The results are returned as a DataFrame with the original text and predicted labels.

API Endpoints

1. Prediction Endpoint:
   URL: /predict
   Method: POST
   Purpose:
   - Accepts a list of text inputs.
   - Predicts the labels for each input using the pre-trained LSTM model.
   - Returns a JSON response with the original text and predicted labels.
2. Request/Response Formats:
   - Request Format: JSON payload containing a list of text inputs.
   - Response Format: JSON response containing a list of dictionaries with the original text and predicted labels.

6. HOW TO RUN AND TEST CASES:

How To Run:

Step 1: docker pull encore7/textclassifier
Step 2: git clone https://github.com/Encore7/TextClassifier.git
Step 3: docker run -p 8000:80 --volume <location on local system>\TextClassifier/app/model_files:/code/model_files encore7/textclassifier

Step 4: http://127.0.0.1:8000/docs

Test Cases:

Test Case 1: Predicting Labels for Sample Input Data

1. Input: Sample input data with various text entries.
2. Action: A POST request is made to the /predict endpoint with the sample input data.
3. Expected Results: The response status code should be 200 (OK). The predicted labels in the response should belong to the specified list of labels.
4. Assertion:Check that the response status code is 200. Verifing that the predicted labels match the expected list.

Test Case 2: Handling Empty Input Data

1. Input: Empty input data.
2. Action: A POST request is made to the /predict endpoint with empty input data.
3. Expected Results: The response status code should be 400 (Bad Request), indicating a client error due to empty input data.
4. Assertion: Check that the response status code for empty input data is 400.

7. FUTURE WORK:

- Hyperparameter Tuning: Conducting a more extensive search for optimal hyperparameters to enhance the model's performance. This includes adjusting LSTM layer parameters, embedding dimensions, and dropout rates.

- Model Architecture Exploration: Experiment with different neural network architectures, such as bidirectional LSTMs or attention mechanisms, to explore if alternative structures such as LLMs that can capture complex patterns better.

8. CONCLUSION:

Overview:
The text classification project aimed to develop a robust and scalable system for categorizing German text into predefined classes. Leveraging a Long Short-Term Memory (LSTM) neural network, the model demonstrates promising results in accurately predicting the labels of input text.

Achievements:

- Model Performance: The LSTM model exhibits competitive performance, achieving a commendable accuracy on both training and test datasets. The incorporation of class weights and careful preprocessing contributed to handling imbalanced class distributions.
- FastAPI Integration: The deployment of the model using FastAPI enables seamless interaction through a RESTful API. The integration provides a straightforward mechanism for users to obtain predictions for batches of input text.
- Inference Pipeline: The inference pipeline efficiently processes new input data, applies the pretrained model, and returns predictions with the corresponding labels. The incorporation of a feedback loop ensures continuous improvement based on user input.
- Dockerized Deployment: The entire project has been containerized using Docker, enhancing portability and simplifying deployment across different environments. Dockerization facilitates consistent and reproducible deployments, streamlining the integration of the system into various setups.