# Visable Challenge

Author: Wael Chabir

Date: 07/03/2024

Github link: <https://github.com/chabirOael/WaeCha024>

# Task 1:

Since the goal of this task is to build a German text classifier, I used a pre-trained model designed for German language: “dbmdz/bert-base-german-cased”.

## Dataset cleaning:

Before jumping in model implementation and training, a pre-processing step is crucial to clean the dataset provided. Below is a list of anomalies found in the dataset and has been cleaned:

* **Bank space** in ‘text’ column: blank spaces provide no information to the model, and also cannot be imputed. I dropped these rows.
* **Chinese texts**: The model is targeting German written texts. Finding Chinese text has no sense. I dropped these rows too.
* **Mail addresses**: mail addresses are not part of any type of products that the model need to classify. They must land there by error. To be dropped.

## Missing labels imputation:

Found some rows (100) having empty label in the dataset. The idea is either to drop these rows or impute them. After data analysis, I found out that all the rows having empty labels, have 'text' value containing the word 'drehteile'.

More than 80% of labeled rows where 'text' column containing the word 'drehteile' have the label 'mr'. I imputed missing labels with ‘text’ value of ‘mr’.

## Model training:

After setting up the model and its data input pipeline, I started the training.

The model scored great result after training with only 3 epochs: 92% accuracy on Test sub-dataset.

## Model serving:

The trained model is pushed to HuggingFace repository.

Following the requirements of this task to build a FastAPI app to serve predictions, I implemented the serving script and encapsulated it into a docker container for easy deployment/use. Once lunched, the container pulls the text classifier model from HuggingFace repo, put it into a pipeline for inference by request.

In the Task1 folder, I wrote a Readme file to guide you step-by-step in order to build, deploy and test the containerized application.

## Further improvements:

The current model reached great performance on Train and Test datasets. However, I believe there is always room for improvements. In order to achieve this goal, I think we can work on the following axels:

* Try other models (larger size)
* Fine-tune hyperparameters (learning rate…)
* Bring larger dataset for training

# Task 2:

For this task, I choose to target **website URLs.**

My solution for the AI system can be splitted into 3 main parts:

* Classic web crawler
* Real-time user pipeline
* Batch user pipeline

## Classic web crawler:

Given a list of websites to keep an eye on, the crawler works as a crone job (every n days) to extract products information from the desired websites and store them into a Structured DB (like PostgreSQL…). We may use Scrapy for web scrapping (open-source and powerful tool for dynamic websites).

We may be interested to have the following data structure (still we can update our structure depending on our needs):

* Brand
* URL
* Product name
* Description
* Photo link
* Update timestamp

## Real-time user pipeline:

In this scenario, the user types a prompt (example: “give me a list of pressure sensors that can handle temperature up to 200°C from Eddylab”. The system at this level encapsulate the user prompt into an envelope prompt for context. Using LLM pre-trained model, the app extracts the brand that the user is looking for. Make a SQL query to filter all products belonging to the desired brand.

Another call to the LLM model to sort products by relevance according to the user’s encapsulated prompt (check for context and similarity for each product’s name and description).

The top ***max\_products\_list*** *products* are then formatted as a JSON format and returned to the user.

**NOTE**: We may experience some latency (pseudo real-time). But I think the processing time is limited due to the fact that the ***Product Filter*** is looking to a subset of the ***Products registry* *DB*** (few hundreds).

## Batch user pipeline:

In this scenario, I choose to adapt micro-service architecture based on Brokers (JMS).

Given the user’s prompt, the ***Prompt Dispatcher*** splits the encapsulated prompt into sub-prompts, each sub-prompt targets a Brand/Link (we give room for multi-processing). **Prompt Dispatcher is the first microservice.** Its output (Brand/Link, sub-prompt) is injected into the JMS Broker.

The **second microservice** is **Products filter**. The first step is to filter Products by Brand/Link (got from the Prompt Dispatcher). Then, the sub-prompt is processed along with the filtered Products list. At this point, Products filter works exactly as described in the Real-time user pipeline above. The result is injected into the JMS broker too.

The **third microservice** is **Response collector**. Its role is simple: it waits that all the sub-requests are processed successfully, group and format them as JSON. The output is finally handed to the user.

## **Room for improvements:**