**Capstone Project Machine Learning Project Documentation**

**Project Title: AI-Powered Learning Recommender for Refugee Education**

**Team Members**

1. **Ekram Kedir (**[**ekedir37@gmail.com**](mailto:ekedir37@gmail.com)**)**
2. **Meron Birhanu (**[**meron9803@gmail.com**](mailto:meron9803@gmail.com)**)**

**Deployment**

1. **Overview**

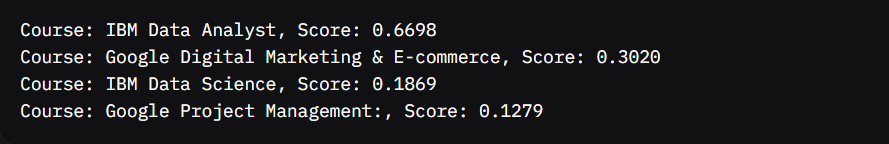
The deployment phase of this machine learning project focused on making the hybrid recommendation system, which combines collaborative filtering and content-based techniques, accessible for local testing and offline use. The system was designed to recommend Coursera courses to users based on their interaction data from the OULAD studentVle dataset, with filtering capabilities by difficulty levels such as Beginner, Intermediate, Advanced, and Mixed. Due to a tight schedule and our lack of prior experience with deployment, the system was not deployed to a production environment. Instead, we prioritized setting up an offline pipeline and a local Flask API for testing purposes. The process involved serializing the trained model, precomputing recommendations for all users, and creating a simple API to serve these recommendations locally. This approach allowed us to validate the system's functionality while working within our time constraints, ensuring the core recommendation engine was operational for local use.

**2. Model Serialization**

The recommendation system relies on a trained SVD (Singular Value Decomposition) model from the Surprise library, which was used for collaborative filtering. This model was serialized using the joblib library and saved as svd\_model.pkl in the C:/Users/Utente/Desktop/FTL/recommender\_system/models/ directory. Additionally, we precomputed recommendations for all users using the hybrid model (combining collaborative filtering and content-based scores) and saved them as recommendations.pkl in the same directory using joblib.dump. The joblib format was chosen because it efficiently handles Python objects, including large dictionaries and machine learning models, allowing for quick loading during offline processing. We ensured that the serialized files were compact, with svd\_model.pkl containing the trained model parameters and recommendations.pkl storing a dictionary mapping user IDs to their top recommended courses, including course titles and scores. This serialization approach minimized memory usage and ensured compatibility with our offline scripts.

**3. Model Serving**

The serialized model and precomputed recommendations are served in two ways. First, we developed an offline script, offline\_recommendations.py, which loads recommendations.pkl and filters recommendations based on user ID and difficulty level. For example, running this script for user 28400 produced outputs like:



It also supports filtering by difficulty, such as Beginner, showing only courses marked with Difficulty\_Beginner=True in the Coursera dataset. Second, we implemented a Flask API in app.py to serve predictions dynamically on a local machine. The API runs on http://localhost:5000/recommendations and was tested using a browser. The Flask server was set up to handle requests locally, allowing us to simulate how the system might work in a real-world scenario without actual deployment. Due to our inexperience and time limitations, we did not deploy this to a cloud platform or on-premises production server, keeping all testing local.

1. **API Integration**

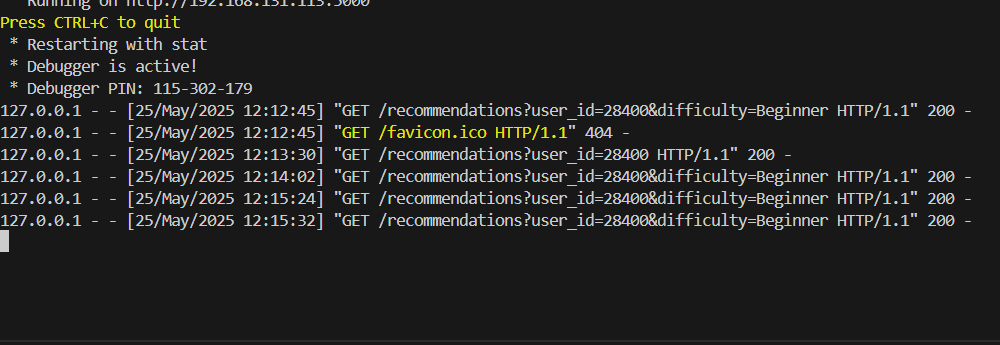
The Flask API provides a single endpoint, /recommendations, which accepts two query parameters: user\_id (required) and difficulty (optional). For example, a GET request to http://localhost:5000/recommendations?user\_id=28400&difficulty=Beginner returns a JSON response like:

{"recommendations": [{"course": "Google Cybersecurity", "score": 0.8192}, {"course": "IBM Data Analyst", "score": 0.6698}, ...]}

If no recommendations match the criteria (e.g., for "Intermediate" difficulty), it returns:

****

The API loads recommendations.pkl and the Coursera dataset (coursera\_courses\_processed.csv) to filter recommendations based on difficulty levels encoded as one-hot columns (e.g., Difficulty\_Beginner). We tested this locally using a browser, confirming successful responses with 200 OK status codes, as shown in the screenshot below. We did not integrate this API into a broader application or test it with external systems, limiting its use to local validation.



**Image Description**: The screenshot shows the Flask development server logs, indicating the API is running on http://127.0.0.1:5000. It logs multiple GET requests to /recommendations?user\_id=28400&difficulty=Beginner, each with a status code of 200, confirming successful responses. The server is in debug mode, running on a local machine, with timestamps and IP addresses (127.0.0.1) visible in the logs.

**5. Security Considerations**

Since the system was only tested locally and not deployed to a production environment, we did not implement robust security measures. The Flask API was run in debug mode on localhost, accessible only on the local machine, which inherently limited exposure to external threats. However, the server was configured to listen on 0.0.0.0, which could pose a risk if accidentally exposed to a network. Due to our lack of experience and the tight schedule, we did not implement authentication, authorization, or encryption (e.g., HTTPS). For a production deployment, we would need to add API key authentication, restrict access to specific IPs, and enable HTTPS to secure data in transit. These limitations were acceptable for our local testing purposes but highlight areas for improvement in future iterations.

**6. Monitoring and Logging**

Monitoring and logging were kept minimal due to time constraints. For the offline pipeline, we manually inspected the output of offline\_recommendations.py, checking recommendation scores and the accuracy of difficulty-based filtering. For the Flask API, we relied on the built-in logging of the Flask development server, which displayed request details in the terminal, such as the HTTP method (GET), endpoint (/recommendations), query parameters (user\_id=28400, difficulty=Beginner), and response status (200 OK). The screenshot of these logs confirms the API handled requests correctly. We did not implement automated monitoring or alerting mechanisms, such as tracking the number of requests, average response time, or recommendation quality metrics (e.g., precision or recall). In a production environment, we would use a logging library like logging in Python to save logs to a file and set up monitoring tools to track system performance and errors, but these were beyond the scope of our project given the time available.