Preparation

Pull the repository: https://github.com/i4talent/fastapi_harvest.git
Follow the instructions in README.md to run the API.

Use the FastAPI documentation when you are stuck: https://fastapi.tiangolo.com/

Or ask a question!

Part 0: ML as a system

1. Write down some ML objectives and translate them to business objectives

2. Create an MLOps architecture for our recommender system including code versioning, online/offline model training and model versioning, model registry, model serving and model monitoring for continuous learning.

Part 1: Model Serving

1. Add a path to *movie_data_router* with a string parameter *user* and a numeric parameter *n.* The path should return the first *n* movie recommendations for a given *user.*
   Hint: Use the predictor dependency.
   Hint: Check out retrain_models.py to find out how the model accepts input data.
   Hint: When passing a single user_id to the predictor, the model will return something like below:

   output = [
   [movie1, movie2, …]
   ]

   Bonus: Why does it return a list of lists?

2. Make sure the path added with assignment 2.1 only accepts user_ids that are known to the model.
   Hint: Check out https://fastapi.tiangolo.com/tutorial/path-params/#predefined-values. A UserEnum is already provided.

   Bonus: For an invalid user the API will return a 422 error. Add a custom description for this error to the SwaggerUI for this route.

3. Add a path to *movie_data_router* that accepts multiple user_ids and a numeric parameter *n.* It should return the first *n* movie recommendations for each user_id. Make the API return a list of Json objects:

   ```
   [
           {
                   "user_id": user1,
                   "movies": [recommendations]
           },
           {
                   "user_id": user2,
                   "movies": [recommendations]
           }
   ]
   ```

   Hint: The same predict method used in assignment 2.1 can do batch predictions using the same input format.
   Hint: For an arbitrary number of inputs it's easiest to use *Query* parameters. Check out:
   https://fastapi.tiangolo.com/tutorial/query-params-str-validations/#query-parameter-list-multiple-values
   Bonus: Add validation for the response, by adding a *response model* to the FastAPI decorator.

Part 2: Model and experiment tracking with MLFlow

1. Expand the function in *src/model_api/mlops/hyperparameter_tuning.py* to train a model and track the model's metric(s) and artifact(s) with MLFlow (Tracking API).
   Hint: You can use *src/model_api/retrain_models.py* as reference.

2. Expand this function to perform hyperparameter optimisation and register the metrics and artifacts with MLFlow.

3. Finish the *register_best_model* function in *src/model_api/mlops/model_serving.py*.

```
import mlflow
client = mlflow.tracking.MlflowClient()
runs = client.search_runs("my_experiment_id", "", order_by=["metrics.rmse DESC"],
max_results=1)
best_run = runs[0]
```

4. Finish up the *load_registered_retrieval_model* and *load_registered_predictor_model* functions in the same file. These are called on API startup using the *lifespan* function in *src/model_api/dependencies.py.* When they succeed, the API will use models trained using MLFlow. Otherwise it will default to the models we used earlier.

Part 3: ML monitoring and drift detection

1. Finish the function *monitor_performance* in *src/model_api/model_monitoring/ml_monitoring.py* to evaluate the model with different batches. You can use the function *get_model_evaluation* in *src/model_api/routers/model_monitoring_router.py* as reference. We will use *top_100_categorical_accuracy* as metric.

2. Finish the *check_for_retraining* function and get the monitoring experiment from assignment 3.1 to check whether the current performance is below a certain threshold.

3. In case the current performance is below the threshold, load the weights of the current model and retrain the model (stateful) on the new data. Hints: use the range of rating IDs from 0 to 90000.

4. Compare the new trained model and the current model on the same test set. If the new model is performing better, register the new model and transition it into the Production stage. You need to implement the function *register_model* in *src/model_api/mlops/model_serving.py* for this.
   Hint: compare the models on *top_100_categorical_accuracy*

Bonus: Docker and Gunicorn with Uvicorn workers

1. Create a dockerfile that runs our API using just uvicorn.

2. Change the dockerfile to run the API using gunicorn with uvicorn workers. Look at the documentation to figure out what options can be used with gunicorn.
   Bonus: Use a configuration file instead of commandline arguments to configure gunicorn.

3. Make sure your API write access logs and error logs to files in a *logs* directory.

4. Ensure the *logs* directory is accessible outside of the docker environment. You can use *volumes* for this.