Learning

In this milestone, we explored different approaches to recommender systems: **content-based**, **collaborative filtering**, and **hybrid**. While hybrid systems are generally strongest, we focused on a **content-based approach** as the first stage toward building a hybrid system.
A Content-based approach relies on movie attributes (e.g., genres, languages, release year) and user profiles (age, occupation, gender) to generate recommendations.

We experimented with three machine learning models to model user-movie watch times and inform recommendations:

Logistic Regression
Implemented in `src/train_model_logistic_regression.py`, this model treats the problem as a binary classification (watch time > 30 minutes). It encodes user/movie IDs as integer indices, splits data (80/20), and trains a scikit-learn LogisticRegression.

- Pros: Simple, fast, interpretable.
- Cons: Linear assumptions; only outputs binary predictions.

Multi-Layer Perceptron (MLP)
Implemented in `src/train_model_mlp.py`, the MLP uses TensorFlow/Keras embeddings (dimension 50 for users/movies), concatenated and fed through dense layers (128→64→1). Trained with Adam and MSE loss for 10 epochs, batch size 32.

- Pros: Learns latent factors and non-linear relationships.
- Cons: Higher computational cost, less interpretable, risk of overfitting.

XGBoost
Implemented in `src/train_model_xgb.py`, this is our primary model. We framed the task as regression, scaling watch times to [0,1] with MinMaxScaler. An XGBRegressor (100 trees, max depth=6, learning rate=0.1) was trained with early stopping.

- Pros: Strong performance on tabular data, captures feature interactions, interpretable via feature importance.
- Cons: Needs careful tuning; limited expressiveness for deep non-linear patterns.

All experiment are implemented in `src/experiments/`

We used XGBoost as the final model for its balance of performance, speed, and interpretability. Hyperparameter tuning with GridSearch was applied to optimize parameters. We trained on explicit ratings rather than watch time, since ratings provide a clearer and more reliable signal of user preference. The final training pipeline is implemented in `src/trainer.py`; feature extraction in `src/feature_builder.py`. Feature extraction rationale explained in `reports/wikis/feature_extraction_report.md`

Inference

The inference service produces ranked movie recommendations by scoring candidate (user, movie) pairs with a trained XGBoost model that predicts a predicted rating for each movie in the candidate pile based on the user information. The service is implemented as a Python component, which is `RecommenderEngine` that loads a trained model, generates candidate rows by joining one user to the movie catalog, runs the same feature pipeline used at training time, scores candidates with the model, and returns the top-K movie IDs sorted by predicted score.

How ranking is derived

- Candidate generation: create a table of candidate pairs by combining the target user with the entire movie catalog (`data/raw_data/movies.csv`). This produces one row per movie annotated with user fields.
- Feature construction: pass the candidate DataFrame through the same feature builder used at training time: `src/feature_builder.py::FeatureBuilder.build(df_override=...)`. That routine applies the feature transforms and returns a feature matrix compatible with the saved model.
- Scoring: call the trained model `predict(` on the transformed features (default model path: `src/models/xgb_recommender.joblib`). The output is a numeric score (the model predicts rating proxy).
- Ranking: sort candidates by predicted score in descending order and return the top-K movie IDs (the code currently returns a comma-separated list of `movie_id`).

Inputs/outputs

- Inputs: `user_id (int)`; the service pulls user metadata via `get_user_info` or accepts a programmatic user dict.
- Outputs: ranked list of movie IDs (top-K), returned as a Python list or a comma-separated string from the runnable example.
- Errors: missing model or missing movie CSV will raise an IO error; cold-start handled by default user values.
-

Code pointers

- Inference service: `src/inference.py` (class `RecommenderEngine`, `recommend` implementation)
- Feature transformations: `src/feature_builder.py` (method `FeatureBuilder.build`)
- Model training & save: `src/trainer.py` (training pipeline; output `src/models/xgb_recommender.joblib`)
- Inference notes: `src/reports/wikis/inference_readme.md`
- Docker for deployment: `docker/Dockerfile` and README docker run example

Team Process

Organization:
Our team organized itself by building upon the workflow proposed in Assignment 0. Originally, we assigned specific roles such as facilitator, devil's advocate, and recorder to different people. However, in practice, we didn't strictly stick to those roles because everyone was very active in participating and sharing ideas. Therefore, we have directly split the tasks based on each member's strengths in data engineering, model training, API development, deployment, and report writing. We believe this adjustment allowed for greater efficiency and flexibility.

Communication:
Originally, we decided that we were going to do weekly in-person meetings after the lecture and communicate via a Slack group chat. However, due to the fact that most of the group members use Discord, we have moved from Slack to a Discord Team Channel to host online meetings and send messages. Also, because some of the team members have varied schedules and aren't available after class, we have decided to use the tool when2meet to decide a time slot that is available for everyone. Overall, 4 meetings were held. The first two meetings were to brainstorm ideas and split the tasks, and the last two meetings were mainly for check-in on the progress and to raise questions. Throughout the work process, all team members were actively replying and helping each other in the group chat, communicating any surprises they encountered, which made the collaboration really smooth.

Work Division and Contributions
Tasks were divided both by each team member's skillset and by deliverable deadlines. Jessica led data extraction, implementing the final model architecture and HuggingFace deployment, while Eric trained the logistic regression model experiments and coordinated the final report. Samuel and Harry worked together on the MLP and XGBoost training experiments and Docker setup, with Harry also assisting in containerization. Marie focused on unit testing and model card documentation. Common tasks like deciding which features to use and the best way to encode them were decided within the technical meetings. Each member contributed to the final report by documenting progress in wikis as coding/testing was completed. Also, all the deadlines were met on time. For more detailed meeting notes on work division and contribution, please refer to the "Meeting Notes.md" file in the report folder.

Significant Commits
- Harry: docker/Dockerfile, train_model_mlp.py
- Sam: train_model_xgb.py, train_model_mlp.py, compare_complexity.py
- Jessica: download_data.py, upload_huggingface.py, inference.py, trainer.py
- Eric: Report_Milestone1.md, Meeting Notes.md, README.md, train_model_logistic_regression.py
- Marie: MODELCARD.md