DIT 637 Smart and Secure Systems
TT07A Experiencing MLOps
08/04/2024 Developed by Clark Ngo
08/04/2024 Reviewed by Sam Chung
School of Technology & Computing (STC) @ City University of Seattle (CityU)

References

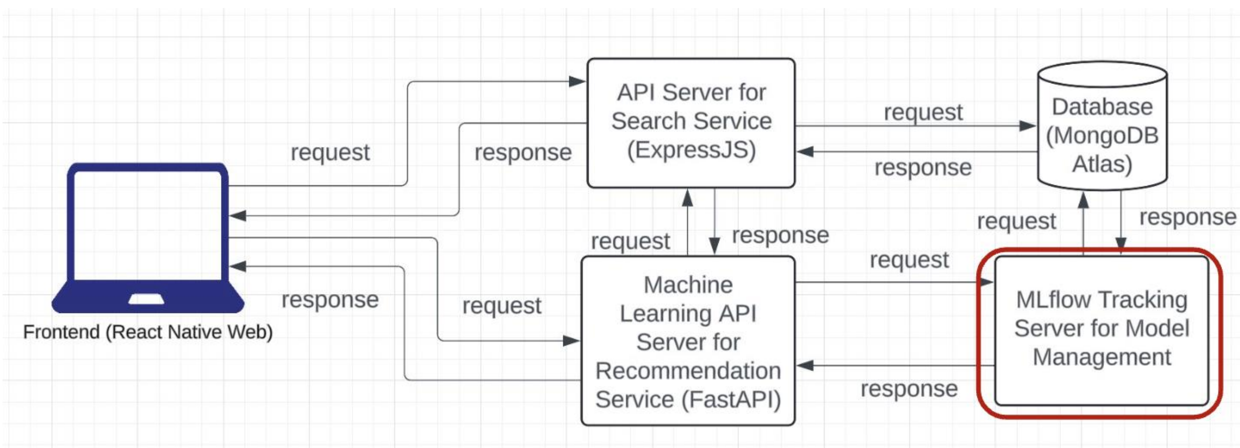
About MLOps

- Canuma, P. (2024, July 26). *Machine Learning Model Management: what it is, why you should care, and how to implement it*. neptune.ai. https://neptune.ai/blog/machine-learning-model-management

About MLfow

- *MLflow Tracking Quickstart — MLflow 2.15.1 documentation*. (n.d.). https://mlflow.org/docs/latest/getting-started/intro-quickstart/index.html

Key Concepts and Tools for Experiencing MLOps



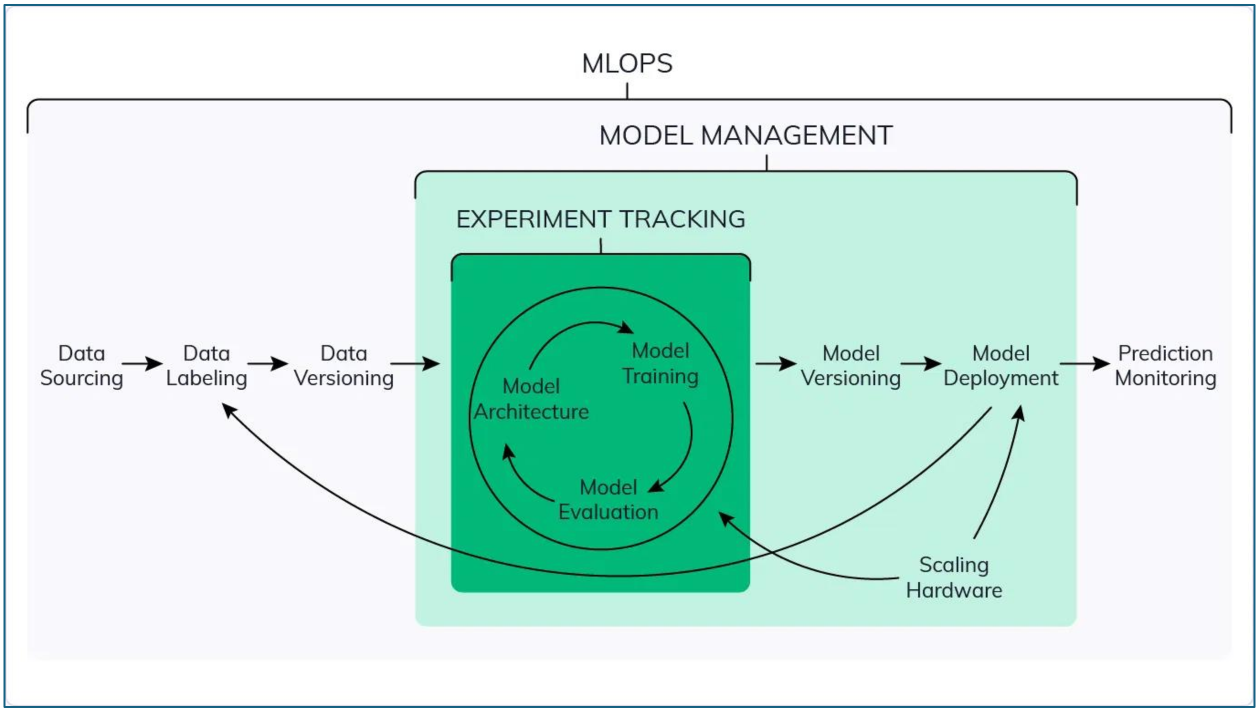


Image Source: *[Machine Learning Model Management]*

Who uses MLflow?

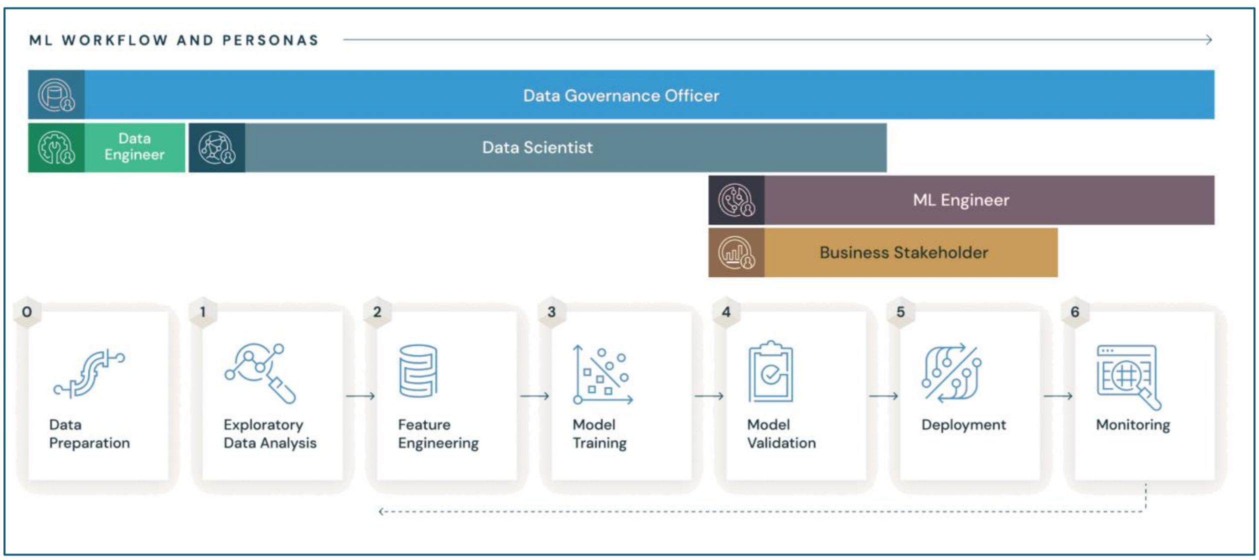


Image Source: *[MLflow Who Uses]*

What are MLOps, Model Management, and Experiment Tracking?

- **MLOps:** A practice that combines machine learning and operations to automate, manage, and scale ML models, ensuring smooth deployment and maintenance. Why MLflow?
    - **Streamlined Machine Learning Lifecycle:** MLflow simplifies the process of managing the machine learning lifecycle, including experimentation, reproducibility, and deployment.
    - **Flexibility and Scalability:** It supports various ML libraries and languages, and scales from local experiments to large-scale production environments.
    - **Comprehensive Tracking:** Provides detailed tracking of experiments, models, and parameters, making it easier to monitor and compare different runs.
- **Model Management:** The process of organizing, storing, and versioning ML models, ensuring they are easily accessible, reproducible, and up-to-date for deployment.
- **Experiment Tracking:** The systematic recording of experiments, including configurations, code, and results, to compare, reproduce, and improve ML models effectively.
- **Model Deployment:** The process of making ML models available for use, ensuring they are accessible for real-time predictions and updates.

What is MLOps without using MLflow?

- **MLOps without MLflow:** Manually coordinate model deployment, updates, and scaling using scripts and tools, ensuring smooth operation.
- **Model Management without MLflow:** Use version control systems like Git or organize models in structured folders to track changes and access.
- **Experiment Tracking without MLflow:** Record experiments in spreadsheets or documents, noting configurations, code, and results for comparison and improvements.
- **Model Deployment without MLflow:** Deploy models using scripts, cloud services, or containerization tools like Docker, ensuring they are accessible for making recommendations.

How will MLOps, Model Management, Experiment Tracking, and Model Deployment support a movie recommendation system?

- **MLOps:** Ensures the movie recommender system runs smoothly, automates updates, and scales effectively for better user experience.
- **Model Management:** Keeps track of different recommendation models, making it easy to update and improve them.
- **Experiment Tracking:** Helps test and compare different recommendation algorithms **to find the best one for accurate movie suggestions**.
- **Model Deployment:** This process ensures that the best recommendation models are available and running so that users can get personalized movie suggestions in real-time.

What models are we using and why?

- **Linear Regression:** A method that finds a straight line to predict outcomes based on input features, like predicting house prices from size.
- **Random Forest:** An ensemble method that uses multiple decision trees to make more accurate and stable predictions, improving reliability over individual trees.
- **Gradient Boosting:** A technique that builds models sequentially, each correcting errors of the previous, to create a strong overall prediction model.
- **Neural Network:** A model inspired by the human brain, using interconnected layers of nodes (neurons) to learn and make complex predictions.

Model	Pros	Cons	Best For
Linear Regression	• Simple and interpretable. • Fast to train and predict.	• Assumes linear relationships. • Sensitive to outliers.	When the relationship between features and target is approximately linear.
Random Forest	• Handles non-linear relationships. • Robust to overfitting. • Can handle large datasets with high dimensionality.	• Can be slower to train on very large datasets. • Less interpretable than linear models.	When you need robust performance and can afford a longer training time.
Gradient Boosting	• Can model complex patterns. • Often achieves high predictive accuracy. • Handles non-linearity and interactions well.	• Can be prone to overfitting if not tuned properly. • Longer training time.	When you need high accuracy and can manage longer training times.
Neural Network	• Can model highly complex patterns. • Flexible with architecture (e.g., layers, units).	• Requires a lot of data and computational resources. • Harder to interpret. • Can overfit if not properly regularized.	When you have large datasets and need to capture complex relationships.

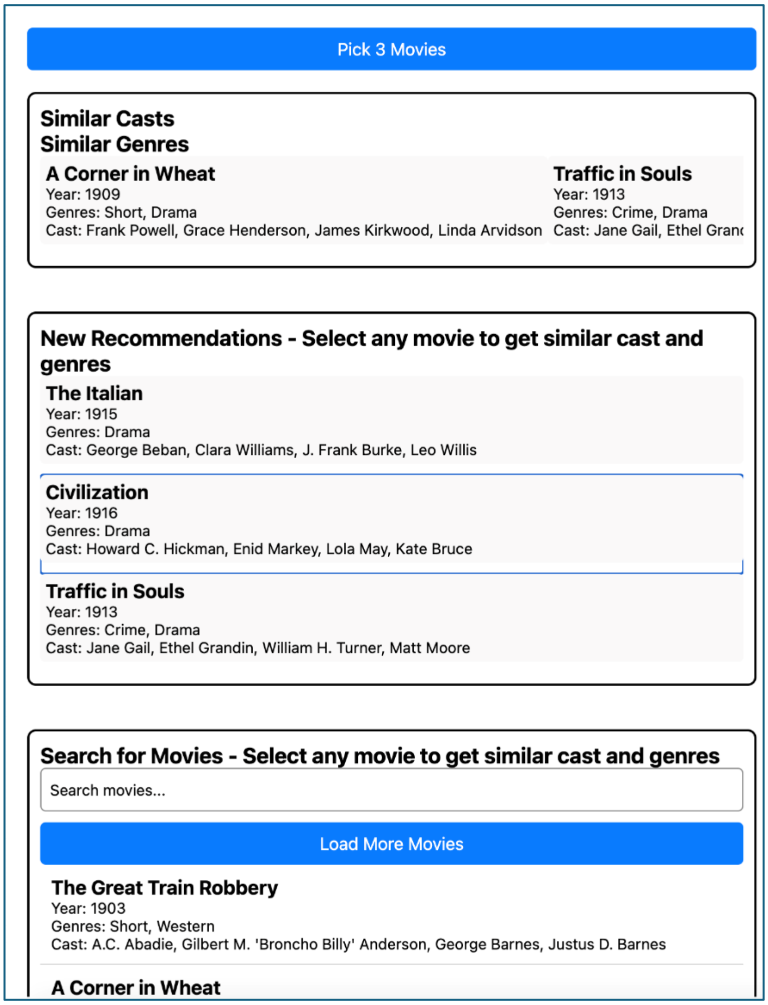
Uploading three image files to your GitHub Repository generated from GitHub Classroom

1. Take a screenshot of your 'MLFLOW TRACKING SERVER' as 'firstname_lastname_mlflow_tracking_server.png'.
2. The screenshot of your 'METRICS TAGS' as 'firstname_lastname_metrics_tags.png'.
3. The screenshot of your 'RECOMMEND V2' as 'firstname_lastname_recommend_v2.png'.

1) **Use Case**

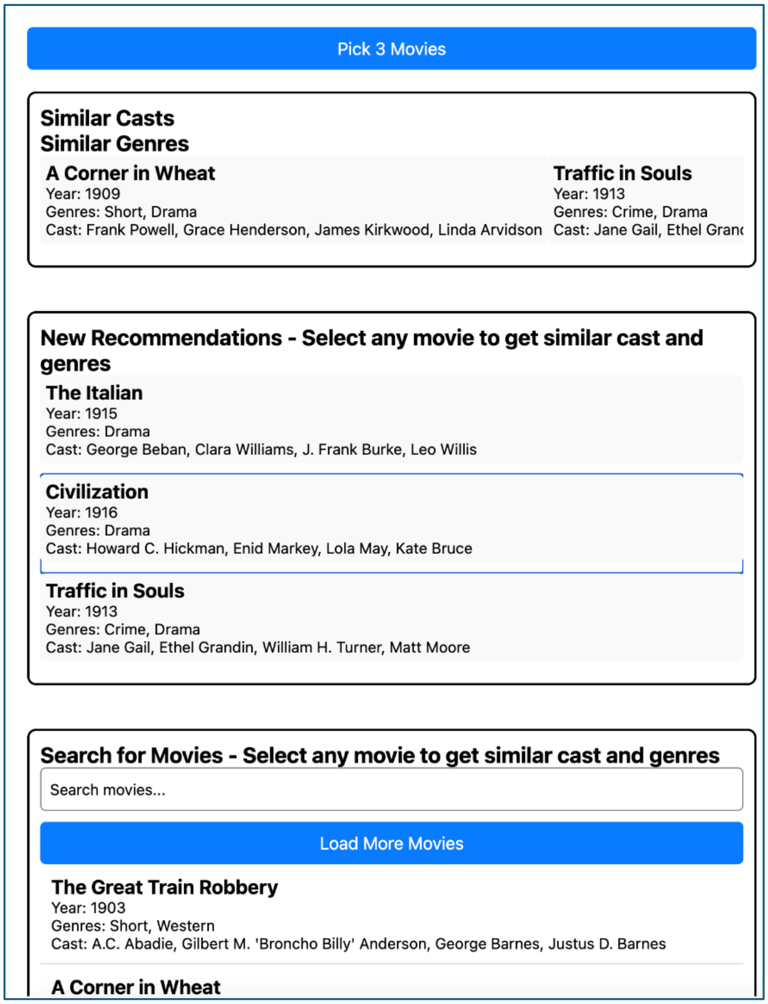
As a movie enthusiast using a mobile device, I want to search and browse a list of movies with details such as title, genre, and year so that I can easily find information about movies I am interested in while on the go.

As a movie enthusiast using a mobile device, I want to get movie recommendations from my selections.



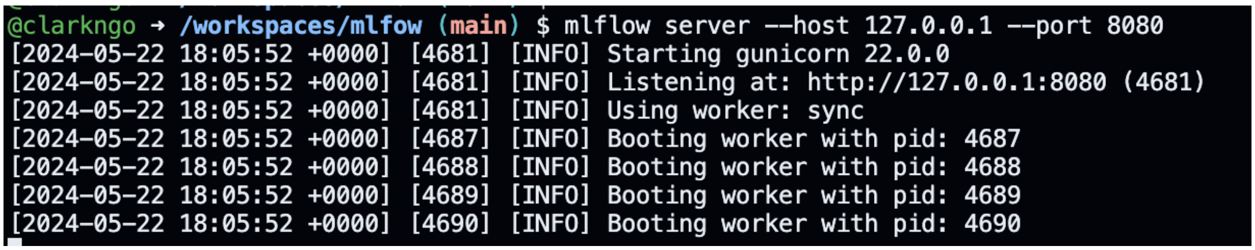
Features included:

- Movie List Display
- Search Functionality
- Initial Recommendations
- New Recommendations
- Similar Cast and Genres



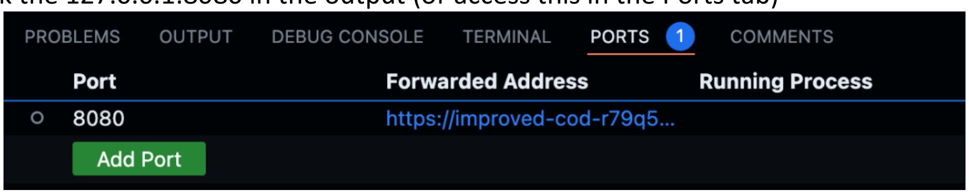
2) Setup

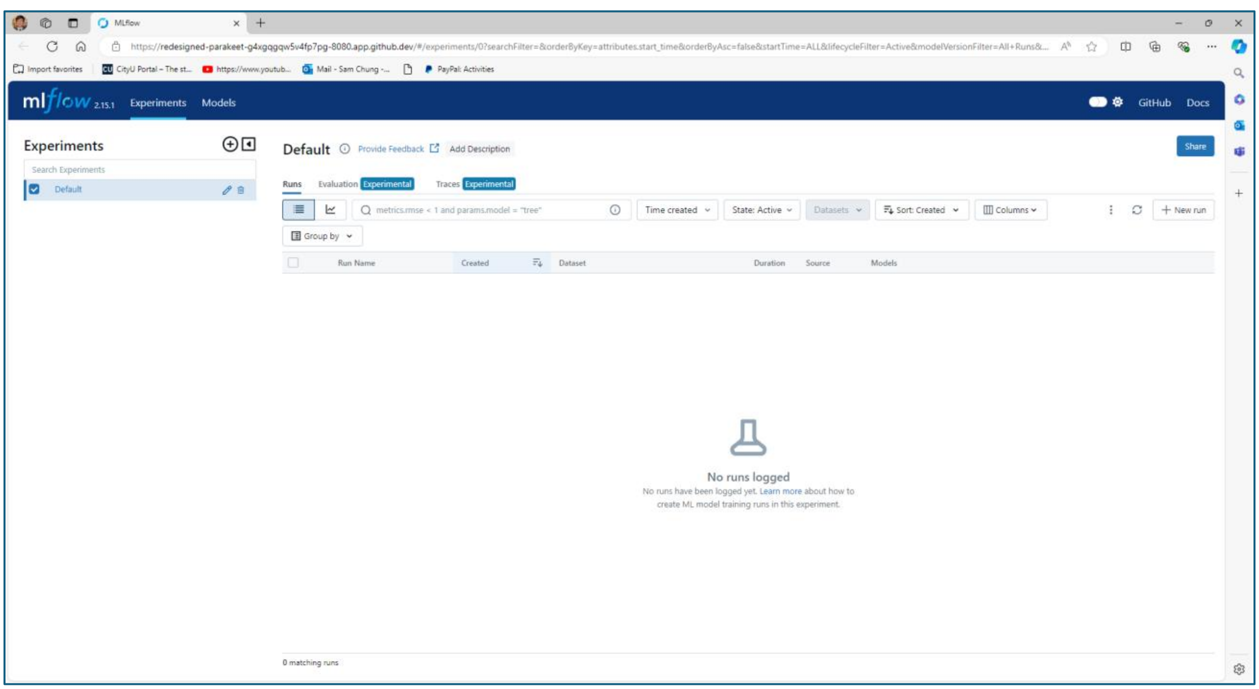
- Create/Open GitHub Codespaces.
- Open a terminal and type the following:
  - cd mlops_mlflow
  - pip install -r requirements.txt
  - mlflow server --host 127.0.0.1 --port 8080
- Make the Port Visibility `Public`.
- Test the Forwarded Address in the Browser



Access the MLflow Tracking Server

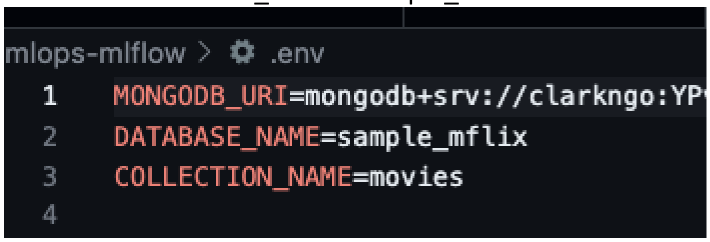
- Click the 127.0.0.1:8080 in the output (or access this in the Ports tab)





Train different model and prepare metadata for logging

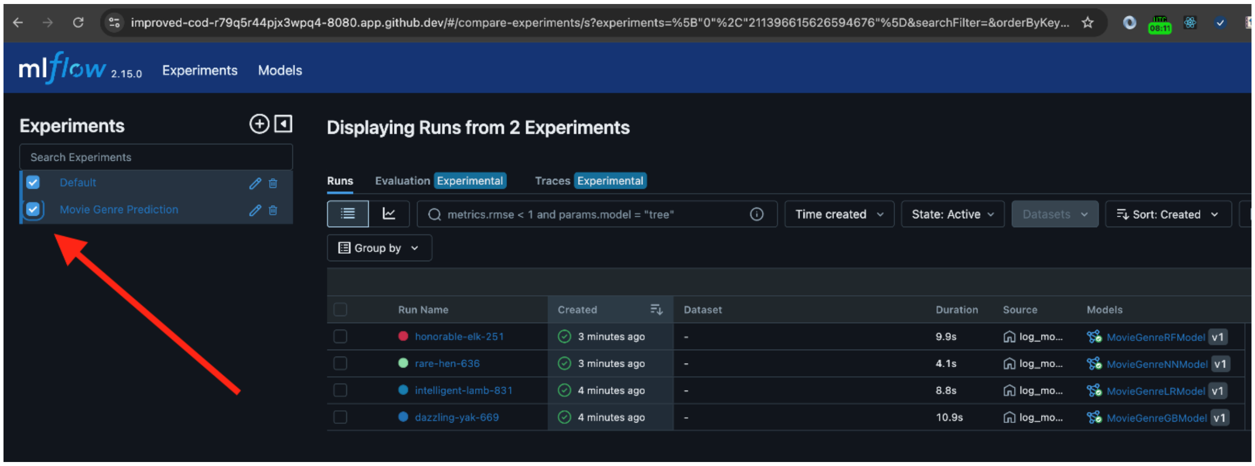
- Create a **.env** file inside mlops-mlflow folder (use example.env file as a reference)
  - Update the MONGODB_URI= connection string
  - Copy and paste both DATABASE_NAME=sample_mflix and COLLECTION_NAME=movies



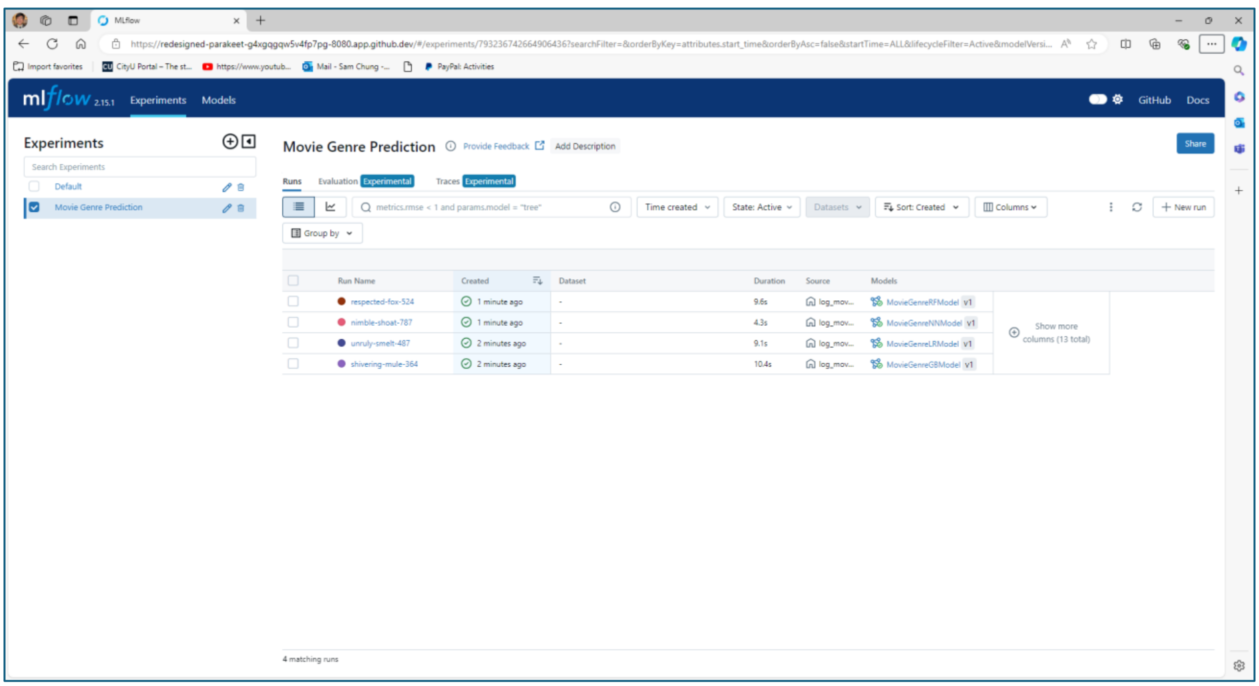
- Open a new terminal:
  - cd mlops-mlflow
  - python run_all_models.py
- New folders 'mlartifacts' and 'mlruns' and their respective files will be generated and will be used as data in the MLflow Tracking Server.
  - **ML Artifacts:** Data files, models, and code created during machine learning experiments, stored for reproducibility, tracking, and reuse in future projects.
  - **ML Runs:** Individual executions of machine learning experiments, including configurations, parameters, and results, tracked to compare and improve models.

View the Experiments in the MLflow Tracking Server

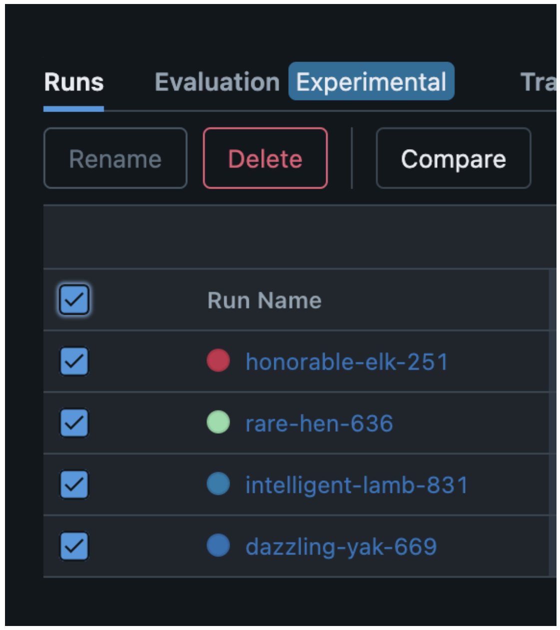
- Head back to the server by clicking the Forwarded Address in the Ports tab
- Tick the checkbox for Movie Genre Prediction to view all 4 models that were run



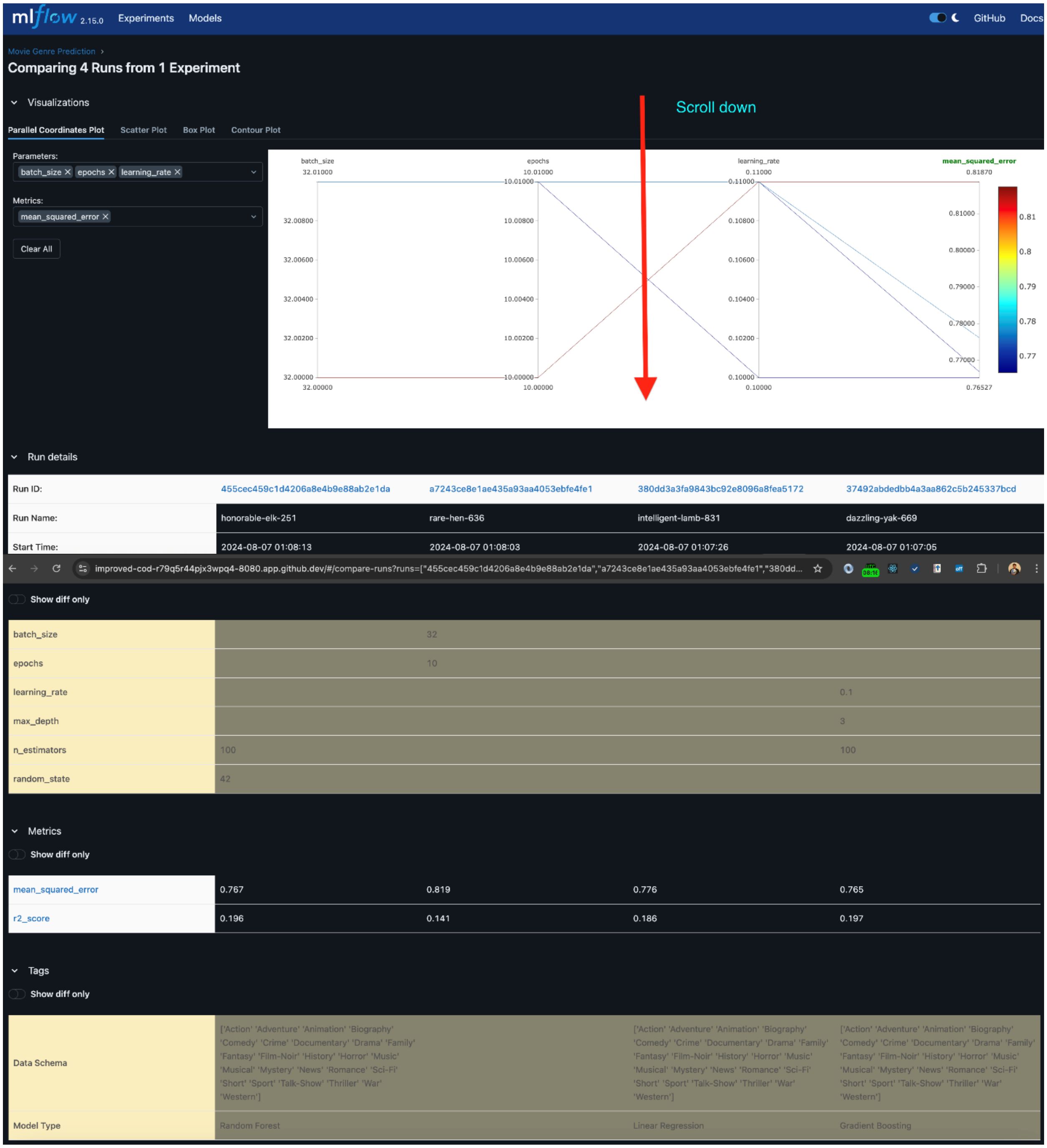
- Take a screenshot of your 'MLFLOW TRACKING SERVER' as 'firstname_lastname_mlflow_tracking_server.png'.
  For example,



- Select all checkboxes for all models and click Compare



- Scroll all the way down

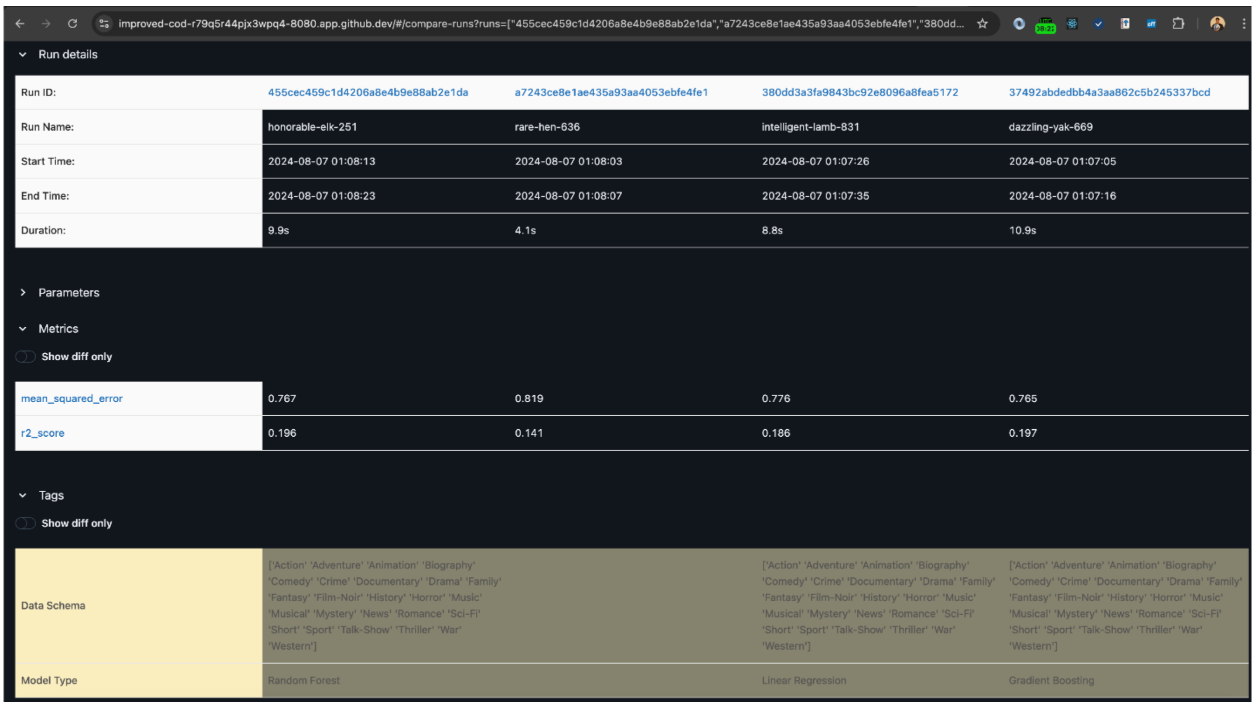


- Take a screenshot of your ‘METRICS TAGS’ as ‘firstname_lastname_metrics_tags.png’.

For the MFLIX dataset, if you aim for high accuracy and have the resources, **Gradient Boosting** or **Neural Networks** are strong contenders. For a more balanced approach with moderate complexity and interpretability, **Random Forest** is a solid choice. If the data is simpler or if you need quick results, you might explore **Linear Regression**, though it may not provide the best results for movie recommendations.

You can see four 'Run ID' with different model types:

- **Random Forest** provides a good balance between performance and complexity, making it a solid choice for movie recommendations with a variety of features.
- **Neural Networks** are powerful for sophisticated recommendation systems, particularly with large datasets and complex patterns, but they require significant computational power and are challenging to interpret.
- **Linear Regression** is likely not suitable for movie recommendations due to its inability to capture complex, non-linear relationships in user preferences and movie attributes.
- **Gradient Boosting** offers high accuracy and can handle complex interactions, making it ideal if you need the best predictive performance and have the computational resources for tuning.



improved-cod-r79q5r44pjx3wpq4-8080.app.github.dev/#/compare-runs?runs=["455cec459c1d4206a8e4b9e88ab2e1da","a7243ce8e1ae435a93aa4053ebfe4fe1","380dd...

Run details

Run ID:	455cec459c1d4206a8e4b9e88ab2e1da	a7243ce8e1ae435a93aa4053ebfe4fe1	380dd3a3fa9843bc92e8096a8fea5172	37492abdedbb4a3aa862c5b245337bcd
Run Name:	honorable-elk-251	rare-hen-636	intelligent-lamb-831	dazzling-yak-669
Start Time:	2024-08-07 01:08:13	2024-08-07 01:08:03	2024-08-07 01:07:26	2024-08-07 01:07:05
End Time:	2024-08-07 01:08:23	2024-08-07 01:08:07	2024-08-07 01:07:35	2024-08-07 01:07:16
Duration:	9.9s	4.1s	8.8s	10.9s

Parameters

Metrics

Show diff only

mean_squared_error	0.767	0.819	0.776	0.765
r2_score	0.196	0.141	0.186	0.197

Tags

Show diff only

Data Schema	['Action' 'Adventure' 'Animation' 'Biography' 'Comedy' 'Crime' 'Documentary' 'Drama' 'Family' 'Fantasy' 'Film-Noir' 'History' 'Horror' 'Music' 'Musical' 'Mystery' 'News' 'Romance' 'Sci-Fi' 'Short' 'Sport' 'Talk-Show' 'Thriller' 'War' 'Western']		['Action' 'Adventure' 'Animation' 'Biography' 'Comedy' 'Crime' 'Documentary' 'Drama' 'Family' 'Fantasy' 'Film-Noir' 'History' 'Horror' 'Music' 'Musical' 'Mystery' 'News' 'Romance' 'Sci-Fi' 'Short' 'Sport' 'Talk-Show' 'Thriller' 'War' 'Western']	['Action' 'Adventure' 'Animation' 'Biography' 'Comedy' 'Crime' 'Documentary' 'Drama' 'Family' 'Fantasy' 'Film-Noir' 'History' 'Horror' 'Music' 'Musical' 'Mystery' 'News' 'Romance' 'Sci-Fi' 'Short' 'Sport' 'Talk-Show' 'Thriller' 'War' 'Western']
Model Type	Random Forest		Linear Regression	Gradient Boosting

Based on the metrics provided—mean_squared_error (MSE) and r2_score—here's how to evaluate the performance of the models:

Metrics Summary:

- **Mean Squared Error (MSE)**:
  - Random Forest: 0.767
  - Neural Network: 0.819
  - Linear Regression: 0.776
  - Gradient Boosting: 0.765
- **R^2 Score**:
  - Random Forest: 0.196
  - Neural Network: 0.141
  - Linear Regression: 0.186
  - Gradient Boosting: 0.197

Metrics

Show diff only

mean_squared_error	0.767	0.779	0.776	0.765
r2_score	0.196	0.183	0.186	0.197

Evaluation:

1. **Mean Squared Error (MSE)**:
   - **Lower is Better**: MSE measures the average squared difference between the actual and predicted values. A lower MSE indicates better performance.
   - **Best Model**: **Gradient Boosting** has the lowest MSE (0.765), meaning it has the smallest average squared error among the models.
2. **R^2 Score**:
   - **Higher is Better**: The R^2 score measures the proportion of variance in the dependent variable that is predictable from the independent variables. A higher R^2 score indicates better performance.
   - **Best Model**: **Gradient Boosting** has the highest R^2 score (0.197), meaning it explains more of the variance in the target variable compared to the other models.

Conclusion:

Gradient Boosting is the best model based on these metrics. It has:

- The lowest Mean Squared Error (MSE), indicating the smallest average prediction error.
- The highest R^2 Score, indicating that it explains the most variance in the target variable.

Reasons for Gradient Boosting's Performance:

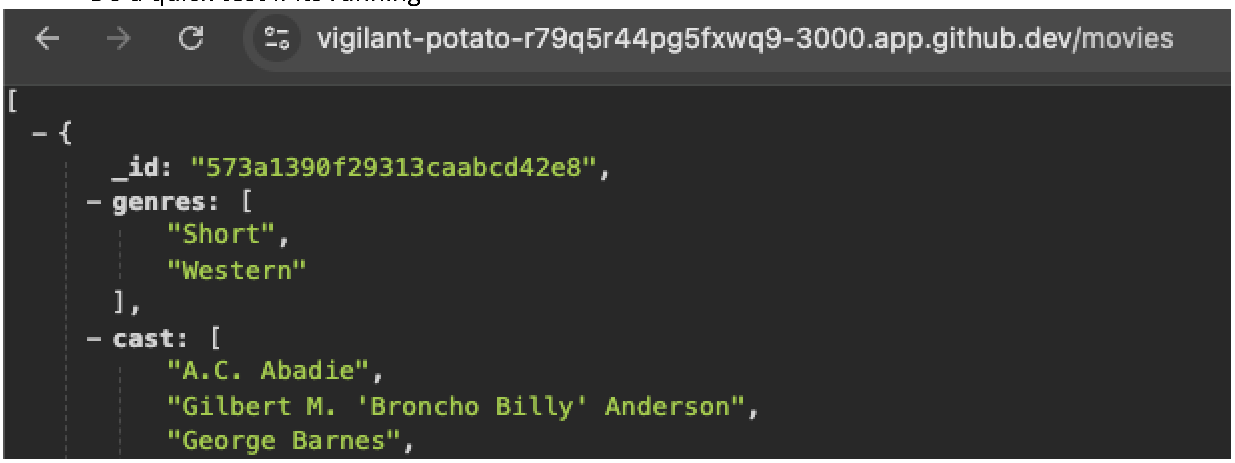
- **Boosting Techniques**: Gradient Boosting builds models sequentially, each correcting the errors of its predecessor, which can lead to improved performance over other methods.
- **Complexity Handling**: It can capture complex patterns in the data better than simpler models like Linear Regression.

Recommendation:

You should prefer the Gradient Boosting model for your use case based on the provided metrics. It offers the best balance between error reduction and variance explanation.

Running the Backend ExpressJS Application Server

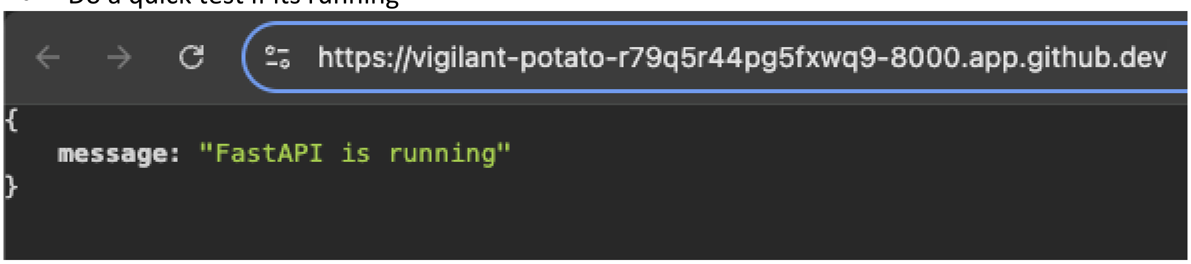
- Create a **.env** file inside 'backend-expressjs' folder (use example.env file as a reference)
  - Update the MONGODB_URI= connection string
- Open a new terminal:
  - cd backend-expressjs
  - npm install
  - npm run start
- Make the Port Public
- Copy the Forwarded Address (URL)
- Do a quick test if its running



Running the Machine Learning Model FastAPI Server

- Create a **.env** file inside 'modelserver-fastapi' folder (use example.env file as a reference)
  - Update the EXPRESSJS_BASE_URL= with ExpressJS Forwarded Address
  - Please be mindful to NOT include a "/" **at the end**

- Open a new terminal:
  - cd modelserver-fastapi
  - pip install –r requirements.txt
  - uvicorn main:app --reload
- The 'uvicorn main:app –reload' is used to start a FastAPI application with Uvicorn, an ASGI (Asynchronous Server Gateway Interface) server.
- Make the Port Public
- Copy the Forwarded Address (URL)
- Do a quick test if its running



Running the Frontend React Native Mobile/Web

- Create a **.env** file inside 'frontend-reactnative' folder (use the 'example.env' file as a reference)
    - Update the API_URL_SEARCH= with ExpressJS Forwarded Address
    - Update the API_URL_RECOMMEND= with FastAPI Forwarded Address
    - Please be mindful to NOT include a "/" **at the end**

```
frontend-reactnative > .env
# ExpressJS Forwarded Address
API_URL_SEARCH=https://vigilant-potato-r79q5r44pg5fxwq9-3000.app.github.dev

# FastAPI Forwarded Address
API_URL_RECOMMEND=https://vigilant-potato-r79q5r44pg5fxwq9-8000.app.github.dev
```

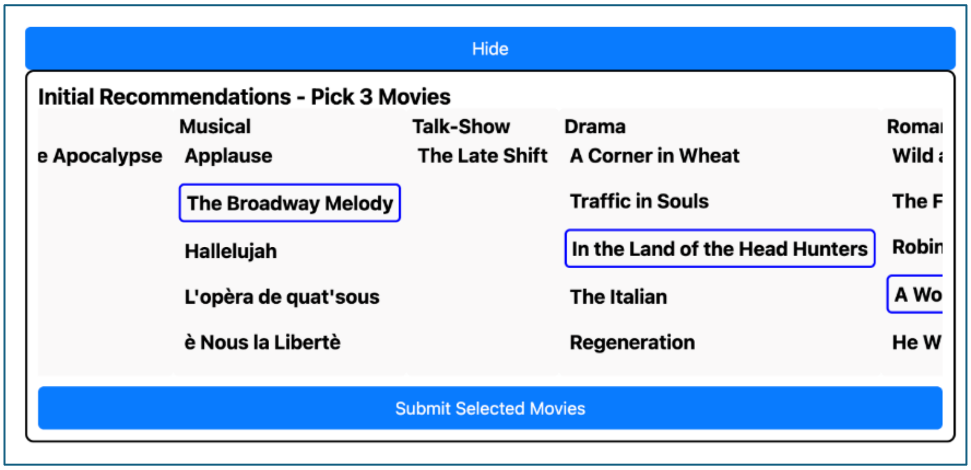
- Open a new terminal:
    - cd frontend-reactnative
    - npm install
    - npx expo login
    - npx expo start --web (or if mobile: npx expo start --tunnel)
- Open the Forwarded Address.
- Pick 3 Movies.
- Click Submit Selected Movies.

- Click Pick 3 Movies again.
- Take a screenshot of your ‘RECOMMEND V2’ as ‘firstname_lastname_recommend_v2.png’.

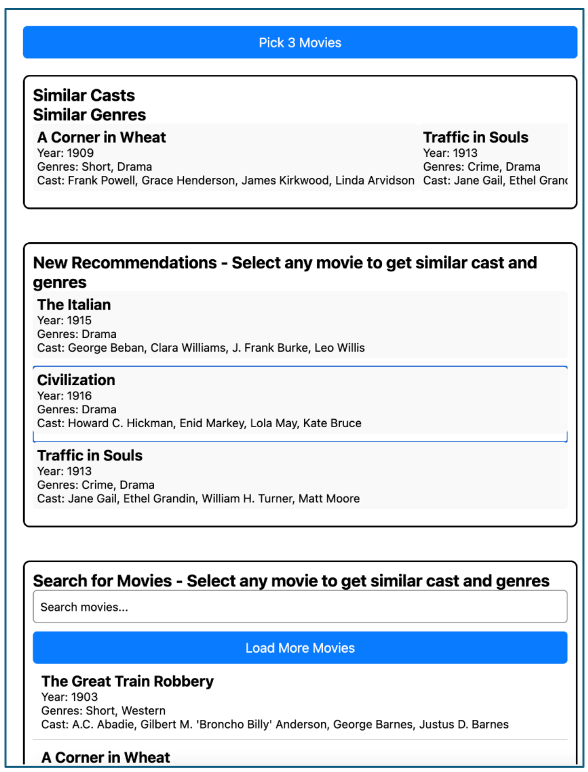
As you may have noticed, the interface is the same. We have improved the New Recommendations by building, deploying, selecting, and loading our own model instead of directly using a model in a library. Note that there are more powerful models out there to try out. Our existing models can also be improved further with training, fine-tuning, validation, etc.

3) Screenshot Summary

Initial Recommendations: Pick 3 Movies and then submit from randomly generated movies fetched from the database.



New Recommendation: List generated from Machine Learning API
Similar Casts and Genres: Select any movie from New Recommendations or Movie List



4) Code Breakdown – MLflow with Python Scripts

This code trains a machine learning model to predict IMDb ratings based on movie genres, using data stored in a MongoDB database. The entire process includes data fetching, preprocessing, model training, evaluation, and logging with MLflow.

ML Pipeline for Gradient Boosting Model – Python Script

- log_movie_model_br.py (very similar with log_movie_model_rf.py, log_movie_model_nn.py, and log_movie_model_lr.py)

Libraries and Environment Setup

Import Libraries

```
from pymongo import MongoClient
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.ensemble import GradientBoostingRegressor
import mlflow
from mlflow.models import infer_signature
from dotenv import load_dotenv
import os
import numpy as np
```

Load Environment Variables

```
load_dotenv()
MONGODB_URI = os.getenv("MONGODB_URI")
DATABASE_NAME = os.getenv("DATABASE_NAME")
COLLECTION_NAME = os.getenv("COLLECTION_NAME")
```

Data Processing and Preprocessing

Connect to MongoDB and Fetch Data

```
client = MongoClient(MONGODB_URI)
db = client[DATABASE_NAME]
collection = db[COLLECTION_NAME]
data = list(collection.find({}, {'_id': 0, 'genres': 1, 'imdb.rating': 1}))
df = pd.DataFrame(data)
```

Process Genre

```
def process_genres(value):
    if isinstance(value, dict):
        return value.get('genres', [])
    elif isinstance(value, str):
        return value.split(',')
    elif isinstance(value, list):
        return value
    else:
        return []

df['genres'] = df['genres'].apply(process_genres)
df = df[df['genres'].apply(lambda x: len(x) > 0)]
```

One-Hot Encode Genres

```
mlb = MultiLabelBinarizer()
X = mlb.fit_transform(df['genres'])
```

Extract IMDB Ratings

```
def extract_imdb_rating(imdb_info):
    if isinstance(imdb_info, dict):
        rating = imdb_info.get('rating', None)
        if isinstance(rating, str):
            try:
                return float(rating)
            except ValueError:
                return np.nan
        elif isinstance(rating, (float, int)):
            return float(rating)
    return np.nan

df['imdb_rating'] = df['imdb'].apply(extract_imdb_rating)
df['imdb_rating'] = df['imdb_rating'].fillna(df['imdb_rating'].mean())
y = df['imdb_rating']
```

Ensure Consistent Length of Features and Labels

```
if len(X) != len(y):
    raise ValueError("Length of features and labels do not match.")
```

Model Training and Evaluation

Split Data into Training and Testing Sets

```
# Model Training
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Define and Train the Gradient Boosting Model

```
# Define and train the Gradient Boosting model
gbr = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, max_depth=3, random_state=42)
gbr.fit(X_train, y_train)
```

Make Predictions and Evaluate

```
y_pred = gbr.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

Logging with MLflow

Set MLflow Tracking URI and Experiment

```
mlflow.set_tracking_uri("http://127.0.0.1:8080")
mlflow.set_experiment("Movie Genre Prediction")
```

Log Parameters, Metrics, and Model

```
with mlflow.start_run():
    mlflow.log_params({
        "n_estimators": 100,
        "learning_rate": 0.1,
        "max_depth": 3
    })
    mlflow.log_metrics({
        "mean_squared_error": mse,
        "r2_score": r2
    })
    signature = infer_signature(X_train, gbr.predict(X_train))
    model_info = mlflow.sklearn.log_model(
        sk_model=gbr,
        artifact_path="movie_genre_model",
        signature=signature,
        input_example=X_train,
        registered_model_name="MovieGenreGBModel"
    )
    mlflow.set_tag("Data Schema", mlb.classes_)
    mlflow.set_tag("Model Type", "Gradient Boosting")
```

Loading and Predicting with the Model

Load the Model and Make Predictions

```
loaded_model = mlflow.pyfunc.load_model(model_info.model_uri)
predictions = loaded_model.predict(X_test)
result = pd.DataFrame(predictions, columns=["predicted_rating"])
print(result.head())
```

Cleanup

Close MongoDB connection

```
client.close()
```

Run All Models - Python Script

This script runs all models: linear regression, gradient boosting, neural network, and random forest.

Import Libraries

- subprocess: Used to execute external scripts or commands.
- os: Used for interacting with the operating system, such as changing the current working directory.

```
import subprocess
import os
```

Define List of Scripts

- A list of script filenames that you want to run.

```
scripts = [
    'log_movie_model_gb.py',
    'log_movie_model_lr.py',
    'log_movie_model_nn.py',
    'log_movie_model_rf.py'
]
```

Set Path for Scripts

- The directory where the scripts are located. You should update this path to where your scripts are stored.

```
script_path = '/workspaces/DIT637-TT07/mlops_mlflow'
```

Change Working Directory

- Changes the current working directory to script_path, so that the scripts can be found and executed.

```
os.chdir(script_path)
```

Define Function to Run Scripts

- run_script(script_name): A function to run a given script using subprocess.run(), which captures the output and errors, printing them for review.

```
# Function to run a script
def run_script(script_name):
    try:
        print(f"Running {script_name}...")
        result = subprocess.run(['python', script_name], capture_output=True, text=True)
        print(result.stdout)
        if result.stderr:
            print(f"Errors encountered in {script_name}:\n{result.stderr}")
    except Exception as e:
        print(f"Failed to run {script_name} due to: {e}")
```

Run Each Script

- Iterates over each script in the scripts list and calls run_script() to execute it.

```
for script in scripts:
    run_script(script)
```

5) **Code Breakdown – Model Server with FastAPI**

The codebase sets up a FastAPI application that provides movie recommendations based on genre similarity. It uses MLflow to load a machine learning model for predicting movie ratings and utilizes various Python libraries for data processing and API interaction.

Import

- FastAPI, HTTPException: For building the API and handling HTTP exceptions.
- pydantic.BaseModel: For defining data models used in request validation.
- requests: For making HTTP requests to external services.
- typing.List: For type hinting lists in function signatures.
- os, dotenv: For loading environment variables from a .env file.
- fastapi.middleware.cors.CORSMiddleware: For handling Cross-Origin Resource Sharing (CORS) settings.
- numpy, scipy.spatial.distance.cosine: For numerical operations and computing cosine similarity.
- sklearn.feature_extraction.text.CountVectorizer: Not used in the provided code but imported for potential future use in text processing.
- mlflow.pyfunc: For loading and interacting with the machine learning model.
- logging, math: For logging and mathematical operations.

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import requests
from typing import List
import os
from dotenv import load_dotenv
from fastapi.middleware.cors import CORSMiddleware
import numpy as np
from scipy.spatial.distance import cosine
from sklearn.feature_extraction.text import CountVectorizer
import mlflow.pyfunc
import logging
import math
```

Environment Setup

- **load_dotenv()**: Loads environment variables from a .env file.
- **logging.basicConfig(level=logging.INFO)**: Configures the logging level to INFO.

```
# Load environment variables from .env file
load_dotenv()

# Configure logging
logging.basicConfig(level=logging.INFO)
```

FastAPI Setup

- **app = FastAPI()**: Creates a FastAPI application instance.
- **app.add_middleware()**: Adds CORS middleware to the application to allow cross-origin requests from any origin.

```
# CORS Middleware setup
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Models

- **MovieIdsRequest**: Defines the request model for the /recommend endpoint, expecting a list of movie IDs.
- **SimilarRequest**: Defines the request model for the /similar endpoint

```
# Define request models
class MovieIdsRequest(BaseModel):
    movieIds: List[str]

class SimilarRequest(BaseModel):
    genres: List[str]
    cast: List[str]
    title: str
```

MLflow Model Setup

- **MLFLOW_TRACKING_URI**: URI to connect to the MLflow tracking server.
- **mlflow.set_tracking_uri()**: Sets the tracking URI for MLflow.
- **model_uri**: Constructs the URI for the specific MLflow model version.
- **model = mlflow.pyfunc.load_model(model_uri=model_uri)**: Loads the machine learning model from MLflow.

```python
# Set MLflow tracking URI
MLFLOW_TRACKING_URI = "http://127.0.0.1:8080"
mlflow.set_tracking_uri(MLFLOW_TRACKING_URI)

# Define model details
MODEL_NAME = "MovieGenreGBModel"
MODEL_VERSION = 1

# Load model from MLflow
model_uri = f"models:/{MODEL_NAME}/{MODEL_VERSION}"
model = mlflow.pyfunc.load_model(model_uri=model_uri)
```

Utility Functions

- **genre_names_to_vector(genres, num_genres=25)**: Converts a list of genre names into a binary vector representing the presence of each genre.

```python
# Functions for processing and transforming data
def genre_names_to_vector(genres, num_genres=25):
    genre_vector = [0] * num_genres
    genre_indices = {
        'Action': 0, 'Adventure': 1, 'Animation': 2, 'Biography': 3,
        'Comedy': 4, 'Crime': 5, 'Documentary': 6, 'Drama': 7, 'Family': 8,
        'Fantasy': 9, 'FilmNoir': 10, 'History': 11, 'Horror': 12, 'Music': 13,
        'Musical': 14, 'Mystery': 15, 'Romance': 16, 'SciFi': 17, 'Short': 18,
        'Sport': 19, 'Thriller': 20, 'War': 21, 'Western': 22, 'Other': 23, 'Unknown': 24
    }
    for genre in genres:
        if genre in genre_indices:
            genre_vector[genre_indices[genre]] = 1
    return genre_vector
```

- **process_movies(movies_data)**: Converts movie genre data into a vector format suitable for comparison.

```
def process_movies(movies_data):
    genre_names = [
        'Action', 'Adventure', 'Animation', 'Biography', 'Comedy', 'Crime',
        'Documentary', 'Drama', 'Family', 'Fantasy', 'FilmNoir', 'History',
        'Horror', 'Music', 'Musical', 'Mystery', 'Romance', 'SciFi', 'Short',
        'Sport', 'Thriller', 'War', 'Western', 'Other', 'Unknown'
    ]

    processed_data = []
    genre_vector = {genre: 0 for genre in genre_names}
    for movie in movies_data:
        for genre in movie['genres']:
            if genre in genre_vector:
                genre_vector[genre] = 1

    return genre_vector
```

- **fetch_movie_data(movie_ids=None, genres=None, cast=None)**: Fetches movie data from an external service based on provided parameters. Handles both fetching specific movies by IDs and fetching similar movies based on genres and cast.

```
# Unified function to fetch movie details or similar movies
def fetch_movie_data(movie_ids: List[str] = None, genres: List[str] = None, cast: List[str] = None):
    try:
        expressjs_base_url = os.getenv("EXPRESSJS_BASE_URL")
        if not expressjs_base_url:
            raise ValueError("Express.js base URL is not set in environment variables")

        if movie_ids:
            url = f"{expressjs_base_url}/movies"
            payload = {"movie_ids": movie_ids}
        elif genres and cast:
            url = f"{expressjs_base_url}/similar"
            payload = {"genres": genres, "cast": cast}
        else:
            raise ValueError("Insufficient parameters provided for request")

        response = requests.post(url, json=payload)
        response.raise_for_status()
        return response.json()
    except requests.RequestException as e:
        print(f"Error fetching data from Express.js: {e}")
        return None
    except ValueError as e:
        print(e)
        return None
```

- **fetch_movies()**: Fetches a list of all movies from the external service.

```python
def fetch_movies():
    expressjs_base_url = os.getenv("EXPRESSJS_BASE_URL")
    url = f"{expressjs_base_url}/movies"
    response = requests.get(url)
    response.raise_for_status()
    return response.json()
```

Recommendation Logic

- **recommend_based_on_genres(selected_movies)**:
    - Converts selected movies' genres into vectors.
    - Fetches all movies from the external service.
    - Calculates similarity scores between the selected movies and all other movies.
    - Uses the loaded ML model to predict ratings for each movie.
    - Computes a combined score based on genre similarity and predicted rating.
    - Sorts and returns the top recommended movies based on the combined score.

```python
def recommend_based_on_genres(selected_movies):
    logging.info(f"Selected Movies Combined Genres: {selected_movies}")

    selected_genre_vectors_np = np.array(list(selected_movies.values()), dtype=np.int64).reshape(1, -1)
    movies = fetch_movies()

    similar_movies = []

    for movie in movies:
        movie_genres = genre_names_to_vector(movie.get("genres", []))
        movie_genres_np = np.array(movie_genres, dtype=np.int64).reshape(1, -1)

        if np.any(movie_genres_np) and selected_genre_vectors_np.size > 0:
            predicted_rating = model.predict(movie_genres_np)[0]
            genre_similarity_scores = []

            for selected_genre_vector in selected_genre_vectors_np:
                if np.any(selected_genre_vector):
                    score = 1 - cosine(movie_genres_np.flatten(), selected_genre_vector.flatten())
                    genre_similarity_scores.append(score)

            genre_similarity = max(genre_similarity_scores) if genre_similarity_scores else 0.0
            genre_similarity = 0.0 if math.isnan(genre_similarity) else genre_similarity

            combined_score = predicted_rating * genre_similarity

            similar_movies.append(
                {"title": movie.get("title", "Unknown"),
                 "genres": movie.get("genres", []),
                 "cast": movie.get("cast", []),
                 "predicted_rating": float(predicted_rating),
                 "genre_similarity": float(genre_similarity),
                 "combined_score": float(combined_score)}
            )
        else:
            logging.info(f"Skipping movie with invalid genres: {movie.get('title', 'Unknown')}")

    similar_movies.sort(key=lambda x: x["combined_score"], reverse=True)
    return similar_movies
```

- **transform_recommendations(recommendations)**:
    - Transforms the raw recommendation data into a format suitable for the response.
    - Maps fields from the movie data to a standardized response format.

```
def transform_recommendations(recommendations):
    transformed_recommendations = []

    for movie in recommendations:
        transformed_movie = {
            "cast": movie.get('cast', []),
            "genres": movie.get('genres', []),
            "imdb": {"rating": movie.get('predicted_rating', 0)},
            "title": movie.get('title', ""),
            "_id": movie.get('id', "")
        }
        transformed_recommendations.append(transformed_movie)

    return {"recommendations": transformed_recommendations}
```

API Endpoints

- **@app.post("/recommend")**:
    - **Purpose**: Provides movie recommendations based on a list of movie IDs.
    - **Functionality**:
        - Receives a list of movie IDs.
        - Fetches movie data using these IDs.
        - Processes the fetched movie data to get genre vectors.
        - Generates recommendations based on genre similarity and predicted ratings.
        - Returns the top 5 recommended movies in a transformed format.

```
# Define API endpoints
@app.post("/recommend")
async def recommend(request: MovieIdsRequest):
    movie_ids = request.movieIds
    selected_movies = fetch_movie_data(movie_ids=movie_ids)

    if not selected_movies:
        raise HTTPException(status_code=500, detail="Error fetching movie details")

    processed_movies = process_movies(selected_movies)
    recommended_movies = recommend_based_on_genres(processed_movies)
    top_5_movies = recommended_movies[:5]

    logging.info(f"Movie Recommendations: {top_5_movies}")
    response = transform_recommendations(top_5_movies)
    return response
```

- **@app.post('/similar')**:
    - **Purpose**: Fetches similar movies based on provided genres and cast.
    - **Functionality**:
        - Receives genres and cast information.
        - Fetches similar movies from the external service.
        - Filters out the current movie title from the recommendations.
        - Returns a dictionary with filtered similar movies categorized by actors and genres.

```
@app.post('/similar')
async def similar(request: SimilarRequest):
    similars = fetch_movie_data(genres=request.genres, cast=request.cast)
    if not similars:
        raise HTTPException(status_code=500, detail="Error fetching similar movies")

    filtered_recommendations = {
        "actors": [movie for movie in similars.get('actors', []) if movie['title'] != request.title],
        "genres": [movie for movie in similars.get('genres', []) if movie['title'] != request.title]
    }

    return filtered_recommendations
```

- **@app.get('/')**:
    - **Purpose**: Provides a health check endpoint to confirm that the FastAPI application is running.
    - **Functionality**: Returns a simple message indicating that the FastAPI application is operational.

```
@app.get('/')
def read_root():
    return {"message": "FastAPI is running"}
```

Application Execution

- **if name == "main":**:
    - Runs the FastAPI application using uvicorn when the script is executed directly.

```
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

6) Pushing your work to GitHub

1. Go to Source Control on your GitHub codespaces and observe the pending changes.

2. Type the Message for your changes in the Message box on the top. For example,” **Submission for TT07 – Your Name**”

3. Click on the dropdown beside the commit button and select **Commit & Push** to update the changes to your repository main branch.

4. Select **Yes** when prompted.