PROJECT OVERVIEW

Project Title

Movie Recommendation System

Objective

In this assignment, students will build a machine learning system focusing on integrating pre-trained models and an online dataset. The assignment emphasizes understanding the key terminology (data, model, component, system, pipeline, feedback) and applying systems thinking by considering interactions with users and the environment. Students will not create machine learning models from scratch but will work with a pre-trained model and dataset from an online source (e.g., Kaggle, HuggingFace).

The system should:

1. Download and preprocess data from an online source (e.g., Kaggle, HuggingFace).
2. Use a pre-trained machine learning model for inference.
3. Implement feedback loops to simulate user interactions and adjust the system's predictions.
4. Organize the system into logical components (data ingestion, preprocessing, model integration, feedback).
5. Present the final system as a working Python program.

GOALS and HOW WE COVERED

System and Feature Goals

1. Create a machine learning pipeline that seamlessly connects data ingestion, preprocessing, model inference, feedback loops and re-training. -> System can perform every step that mentioned in our code. Only hypothetical steps left as hypothetical. Will be explained deeply.
2. Ensure all components operate in a logical sequence, enabling an end-to-end flow from data download to model prediction and feedback. -> Components are connected to each other and works properly. All components will be explained.

3. Simulate user interactions that allow the system to adjust based on feedback, reflecting realistic user interaction scenarios. -> UX will get needed data like age, gender, interest of genres (in the exact time) to give movie recommendation to user, also UX will receive the feedback of the user.
4. Integrate a pre-trained model effectively within the system to perform prediction. -> Integrated pre-trained model which is GradientBoosting in 3rd section of code block. After the implementation, we became able to perform prediction at the 4th section. Will be explained deeply.
5. Clearly separate the system into distinct modules: data ingestion, preprocessing, model inference, pipeline and feedback. -> System has clear boundaries of modules. There are 5 different modules in our code. Modules might be called in different modules but they are clearly separate from each other.
6. Design a feedback loop that adjusts system predictions based on simulated user responses or corrections. -> Feedback of the user will collect. If there is a needing system will re-train and improve itself.

User Goals

1. Users should get to receive movie recommendations based on their interests. -> We will get data for prediction. Tried to train a model which can perform well.
2. The system's interface should not be complicated for users. -> We didn't design an UI but if there would be an UI, it would be user friendly, easy to understand, well colored. UI would take necessary datas like, age, gender, current interested movie topics etc.
3. The output provided to users should be in a clear and visually appealing format. -> If there was an UI, it would show the name of the recommended movie, maybe a little cover of movie and direct link to watch it.
4. Users should be able to rate the movies they've watched, and recommendations should be updated accordingly. -> If there was an UI, users could vote about the movie, with that way we could improve our training set and could update the pipeline.

Model Goals

1. The model flow should work seamlessly. -> Code runs without any issue.
2. The model should respond quickly. -> Model runtime is really fast.
3. The model's accuracy, precision, and recall values should be high, providing users with accurate movie recommendations. -> Tried to find the best model. Will be explained deeply.
4. The model should be able to successfully retrain and improve itself. -> Code has the background to be retrained in real-life scenario.
5. Integrate a pre-trained model effectively within the system to perform prediction. -> Integrated pre-trained model which is GradientBoosting in 3rd section of code block. After the implementation, we became able to perform prediction at the 4th section. Will be explained deeply.
6. Provide a feature to download and preprocess data from online sources automatically. -> System can download data from online sources with the 1st section of the code. Will be explained deeply.

CHOICE of DATASET and MODEL

There were 3 different datasets which are Movielens 100K, Movielens 2M and IMDB.

We initially considered using the IMDb dataset, as it includes features like title, release date, genres, ratings, actors, directors, and plot summaries. However, we decided not to proceed with it due to a limited number of distinctive columns suitable for model differentiation and the presence of noise within the dataset.

For MovieLens datasets, both the 100K and 2M versions include similar columns: age, user_id, title, genres, timestamp, etc. The 2M dataset contains nearly 6,000 unique films, whereas the 100K dataset has around 1,600 unique films. The 2M dataset's large label variety and limited number of distinctive columns led to reduced model performance and significantly extended training time. Hence, we opted to proceed with the 100K dataset.

MovieLens dataset includes those columns: 'user_id', 'item_id', 'rating', 'timestamp', 'title', 'release_date', 'video_release_date', 'IMDb_URL', 'Action', 'Adventure', 'Animation', 'Children', 'Comedy', 'Crime', 'Documentary', 'Drama', 'Fantasy', 'Film-Noir', 'Horror', 'Musical', 'Mystery', 'Romance', 'Sci-Fi', 'Thriller', 'War', 'Western', 'age', 'gender', 'occupation', 'zip_code'.

However, the 100K dataset also has relatively few distinctive columns, so we decided to train our model on a restricted subset of 101 films for enhanced interpretability and performance.

For this project, we considered using Gradient Boosting, Random Forest, and Support Vector Classifier (SVC) because each has specific strengths that fit well with our dataset.

1. Gradient Boosting:
    - Why it's useful: Gradient Boosting builds a model by adding several simple models (called weak learners) that gradually improve by correcting the errors of previous ones. This helps it handle complex data with lots of features and capture detailed patterns.
    - Drawbacks: It can be slow to train, especially with large datasets, and it needs careful tuning of parameters.
    - Best for: When we want a strong model that can handle complex classification boundaries and capture subtle patterns in the data.
2. Random Forest:
    - Why it's useful: Random Forest builds multiple decision trees on different parts of the data and combines their predictions, which makes it very stable and resistant to overfitting (fitting too closely to training data). It's also great for figuring out which features are most important.
    - Drawbacks: It can be slower on large datasets, and it needs more memory when we use a lot of trees.
    - Best for: When we need a model that generalizes well and can help us see which features matter most for our predictions.

3. Support Vector Classifier (SVC):

    - Why it’s useful: SVC tries to find the best line (or hyperplane) that separates different classes. It’s very effective for datasets that have clear separations between classes and works especially well on smaller datasets.
    - Drawbacks: It can be slower on larger datasets and may struggle with data that has many features. It’s also sensitive to parameter choices.
    - Best for: When we have a clear boundary between classes or when the dataset is small to medium-sized.

Why We Chose Gradient Boosting:

While Random Forest and SVC could have been helpful (Random Forest for feature importance and SVC for clear class separation), we chose Gradient Boosting because it performed a bit better on our dataset. It gave us the highest accuracy for predicting the `title` column, which was our main goal.

DATA PREPROCESSING

To make the data suitable for modeling, we developed a clean_data method that performs a sequence of preprocessing steps:

- Column Selection:
    - We defined a set of relevant columns, such as title, age, gender, and various genre columns (like 'Action', 'Adventure', 'Drama', etc.), which are essential for defining user preferences and demographic features. We filtered the dataset to keep only these columns, excluding any non-essential information to streamline the model's input. (There were 31 total columns before. As we train the model we found p value of the most columns are higher than expected or columns are nominal. That lead us to drop unnecessary columns)
- Handling Missing Values:
    - We used the dropna() function to remove rows with any missing values. This ensures that our dataset is free from nulls, which could otherwise disrupt the model training process.

- Encoding Gender:
    - To convert the gender column into a numeric format, we mapped values in the gender column such that 'F' is mapped to 0 and 'M' is mapped to 1. This transformation is essential because most machine learning models require numeric inputs, and this simple encoding allows the model to distinguish between male and female users effectively.
- Encoding Titles:
    - We applied a LabelEncoder to the title column to convert each unique movie title into a numeric ID. This encoding helps the model treat each title as a unique identifier without assigning any implicit order. After encoding, we filtered the dataset to retain only titles with IDs between 0 and 100. This selection limits the dataset to the top 101 movies, potentially to focus the model on a smaller, more manageable subset of popular or relevant titles, thus reducing computational load and training time.

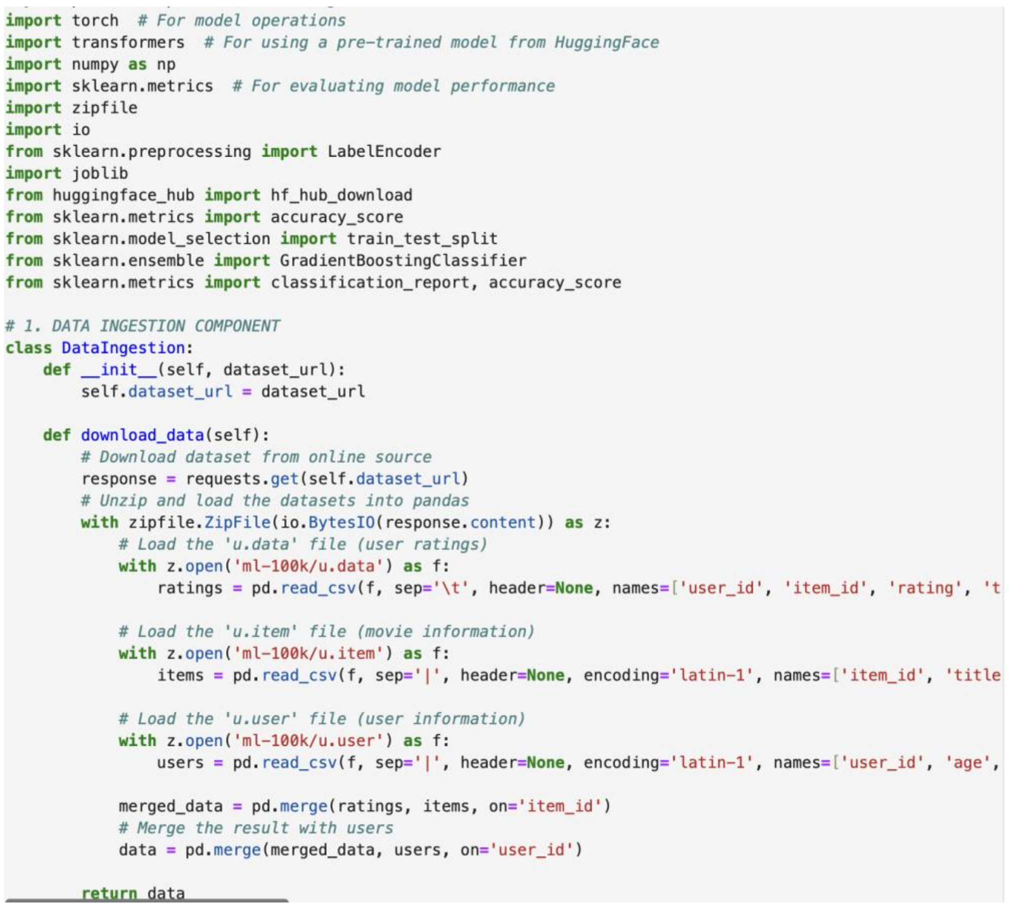
PIPELINE

Our pipeline consists of six stages, each serving a specific role in the recommendation workflow:

1. Data Ingestion: In the first stage, we download the raw dataset. To handle this, we set up a class that fetches the MovieLens100k data, which comprises three separate zip files. Using unique `item_id` and `user_id` columns, we can merge these files seamlessly. We initialize the class with the dataset URL and use the `download_data` function to request, download, and merge the zip files, returning the raw dataset.
2. Data Preprocessing: In the second stage, we clean and prepare the data for modeling. This involves filtering relevant columns, handling missing values, and encoding categorical data. The processed dataset is then structured for input into later stages.
3. Model: This stage is responsible for generating a user profile by analyzing user preferences and demographic data, creating a foundation for personalized recommendations.

4. Recommendation: In the fourth stage, we created a class that combines the first three stages to complete the entire pipeline, from data ingestion to prediction. This class interacts with the preceding steps, allowing for seamless data downloading, preprocessing, and profile-based predictions.
5. Evaluation and Retraining (if needed): In this stage, we compare the actual values with predictions to assess model performance. If the results are inadequate, in a real-world scenario, we would typically return to either the Model or Recommendation stages for further training or tuning. However, for this assignment, we simply display the results on the screen with print.
6. Main Execution: Finally, we bring everything together in the sixth stage, where the main code calls all the classes sequentially to execute the pipeline end-to-end.

This structured pipeline enables effective data handling, model training, evaluation, and retraining, simulating a real-world recommendation system.

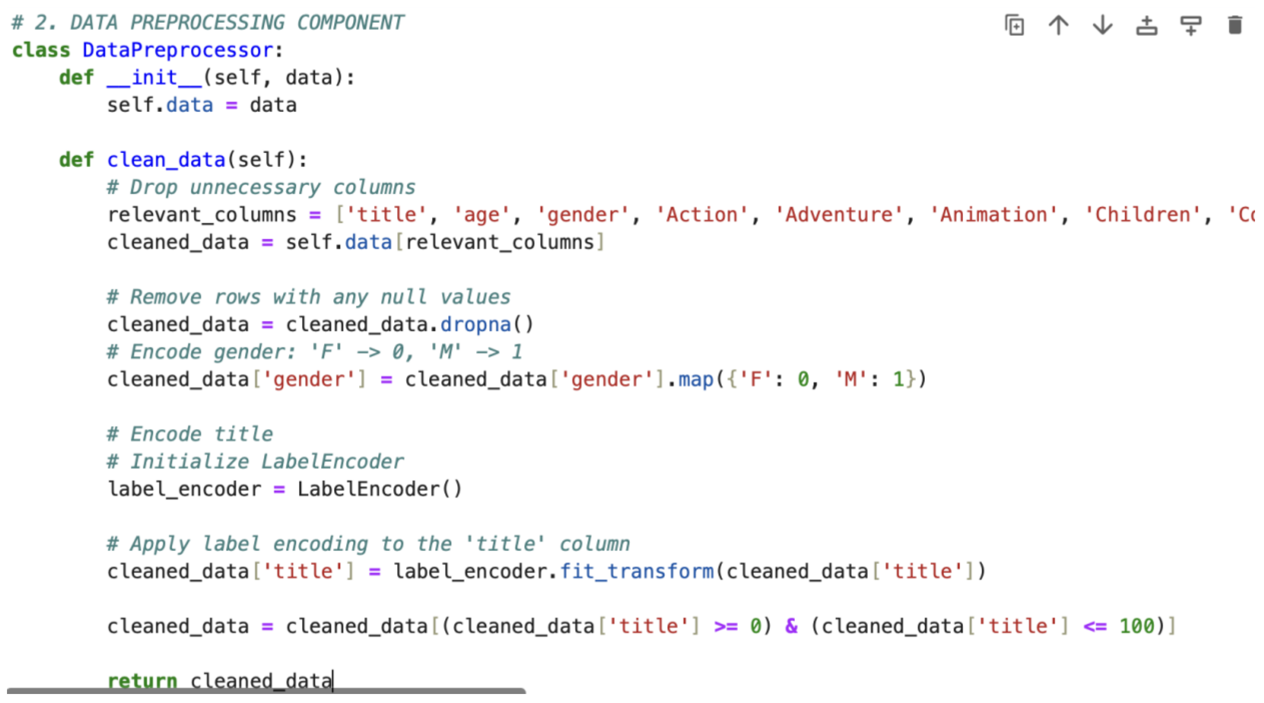


```python
import torch  # For model operations
import transformers  # For using a pre-trained model from HuggingFace
import numpy as np
import sklearn.metrics  # For evaluating model performance
import zipfile
import io
from sklearn.preprocessing import LabelEncoder
import joblib
from huggingface_hub import hf_hub_download
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import classification_report, accuracy_score

# 1. DATA INGESTION COMPONENT
class DataIngestion:
    def __init__(self, dataset_url):
        self.dataset_url = dataset_url

    def download_data(self):
        # Download dataset from online source
        response = requests.get(self.dataset_url)
        # Unzip and load the datasets into pandas
        with zipfile.ZipFile(io.BytesIO(response.content)) as z:
            # Load the 'u.data' file (user ratings)
            with z.open('ml-100k/u.data') as f:
                ratings = pd.read_csv(f, sep='\t', header=None, names=['user_id', 'item_id', 'rating', 't

            # Load the 'u.item' file (movie information)
            with z.open('ml-100k/u.item') as f:
                items = pd.read_csv(f, sep='|', header=None, encoding='latin-1', names=['item_id', 'title

            # Load the 'u.user' file (user information)
            with z.open('ml-100k/u.user') as f:
                users = pd.read_csv(f, sep='|', header=None, encoding='latin-1', names=['user_id', 'age',

            merged_data = pd.merge(ratings, items, on='item_id')
            # Merge the result with users
            data = pd.merge(merged_data, users, on='user_id')

        return data
```

We started by importing the necessary libraries. In the first part of the code, we created a class structure for downloading the data. The selected MovieLens100k dataset consists of three separate zip files, which can be merged using unique `item_id` and `user_id` columns. In the `__init__` function, we set the URL of the dataset. In the `download_data` function, we first made a request to the URL and then downloaded and merged the three required zip files. This function returns the raw, ready-to-use dataset.

IMPORTANT:

In real world scenario, DataIngestion Class takes data from platform. We will ask couple questions to user like age, gender, which genres are she/he interested in now. These will be our data for prediction. We will firstly fill title column with null in the website cause we only need it for re-training. In the end we will also track what she/he watched after those inputs. We will update title and combine inputs and title into a dataset. So basically we assume movielens data is an example of what we receive from user and which movie did she/he choose. Re-training will work as batch not real time component because we need information of the watched movie for feedback.



```python
# 2. DATA PREPROCESSING COMPONENT
class DataPreprocessor:
    def __init__(self, data):
        self.data = data

    def clean_data(self):
        # Drop unnecessary columns
        relevant_columns = ['title', 'age', 'gender', 'Action', 'Adventure', 'Animation', 'Children', 'C
        cleaned_data = self.data[relevant_columns]

        # Remove rows with any null values
        cleaned_data = cleaned_data.dropna()
        # Encode gender: 'F' -> 0, 'M' -> 1
        cleaned_data['gender'] = cleaned_data['gender'].map({'F': 0, 'M': 1})

        # Encode title
        # Initialize LabelEncoder
        label_encoder = LabelEncoder()

        # Apply label encoding to the 'title' column
        cleaned_data['title'] = label_encoder.fit_transform(cleaned_data['title'])

        cleaned_data = cleaned_data[(cleaned_data['title'] >= 0) & (cleaned_data['title'] <= 100)]

        return cleaned_data
```

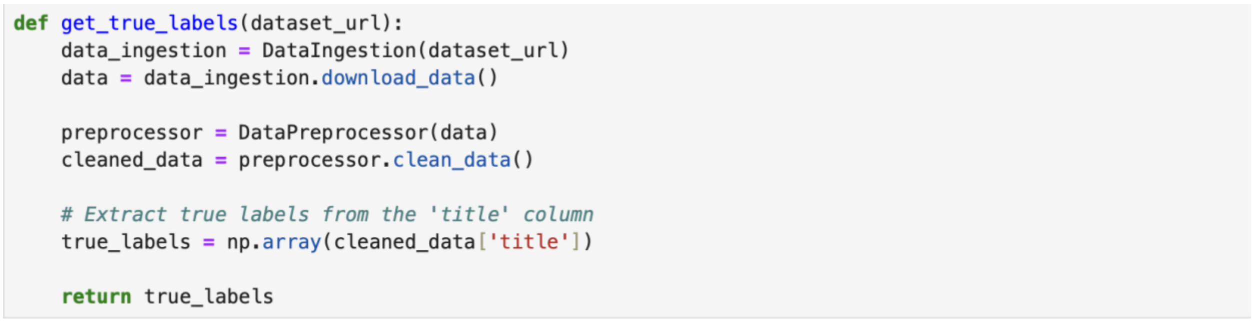
In this class, we use the `__init__` function to implement the raw data. With the `clean_data` function, we first keep only the relevant columns by excluding irrelevant ones, and then we get rid of null values. We encode the `gender` and `title` columns to numeric values. To improve model performance, we reduce the 1,664 unique movies down to 101, as otherwise, the model performs with very low accuracy. In real world scenario we will only take relevant columns from user but as we assume movielens data as input, we needed to drop irrelevant columns.

```python
# 3. MODEL COMPONENT
class PretrainedModel:
    def __init__(self, model_name):
        with open(model_name, 'rb') as f:
            self.model = joblib.load(f)

    def predict(self, inputs):
        inputs = inputs.drop(columns=['title'])
        return self.model.predict(inputs)
```

In this class, we write the code to call and implement the pretrained model that we previously trained and uploaded to Hugging Face using the `__init__` function. The `predict` function contains the necessary code to generate outputs from the predictions based on the inputs. We drop title column cause in the prediction we don't need it. Also in real world scenario it will be null in this moment.

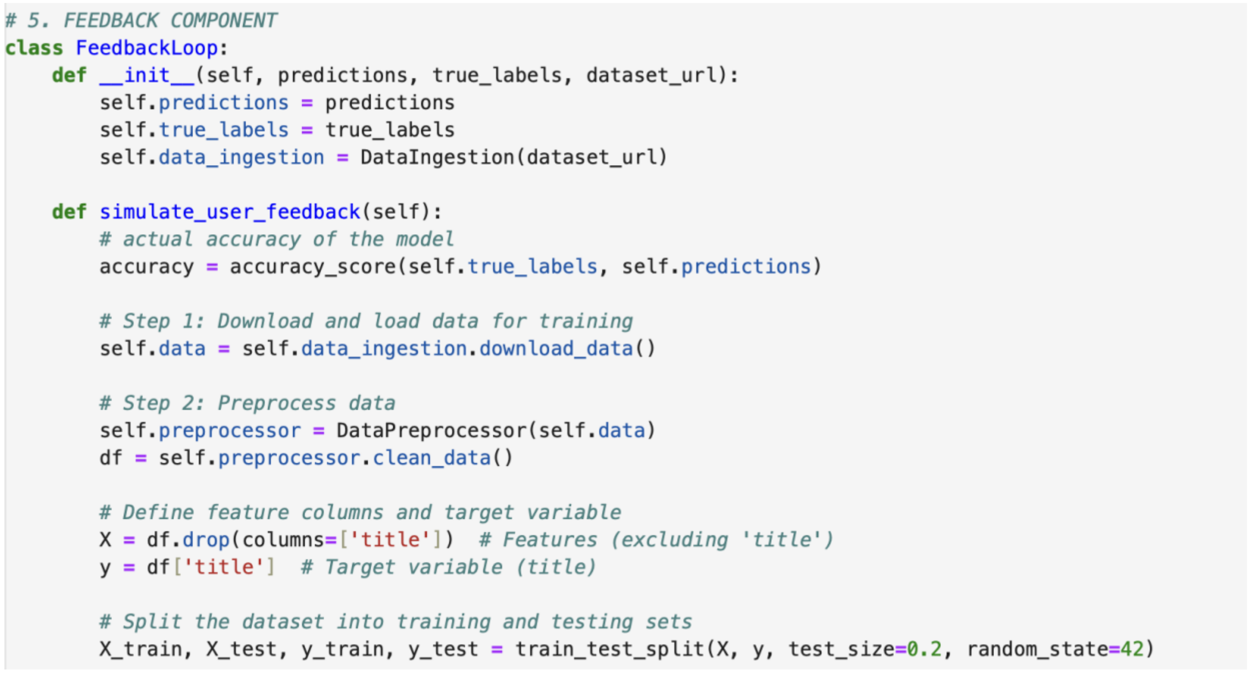
```python
# 4. SYSTEM COMPONENT - PIPELINE
class MLPipeline:
    def __init__(self, dataset_url, model_name):
        self.data_ingestion = DataIngestion(dataset_url)
        self.preprocessor = None
        self.model = PretrainedModel(model_name)
        self.data = None

    def build_pipeline(self):
        # Step 1: Download and load data
        self.data = self.data_ingestion.download_data()

        # Step 2: Preprocess data
        self.preprocessor = DataPreprocessor(self.data)
        cleaned_data = self.preprocessor.clean_data()

        # Step 3: Pass data through the model for prediction
        predictions = self.model.predict(cleaned_data)
        return predictions
```

This class is where we create our pipeline. The `__init__` function takes the dataset URL and model name and interacts with the 1st and 3rd classes, which are the data ingestion and pretrained model classes. The `build_pipeline` function first downloads the data by using a function from the 1st class, then interacts with the 2nd class to send the downloaded raw data for preprocessing. Finally, it uses the `predict` function of the 3rd class to generate prediction outputs from the preprocessed data obtained from the 2nd class.

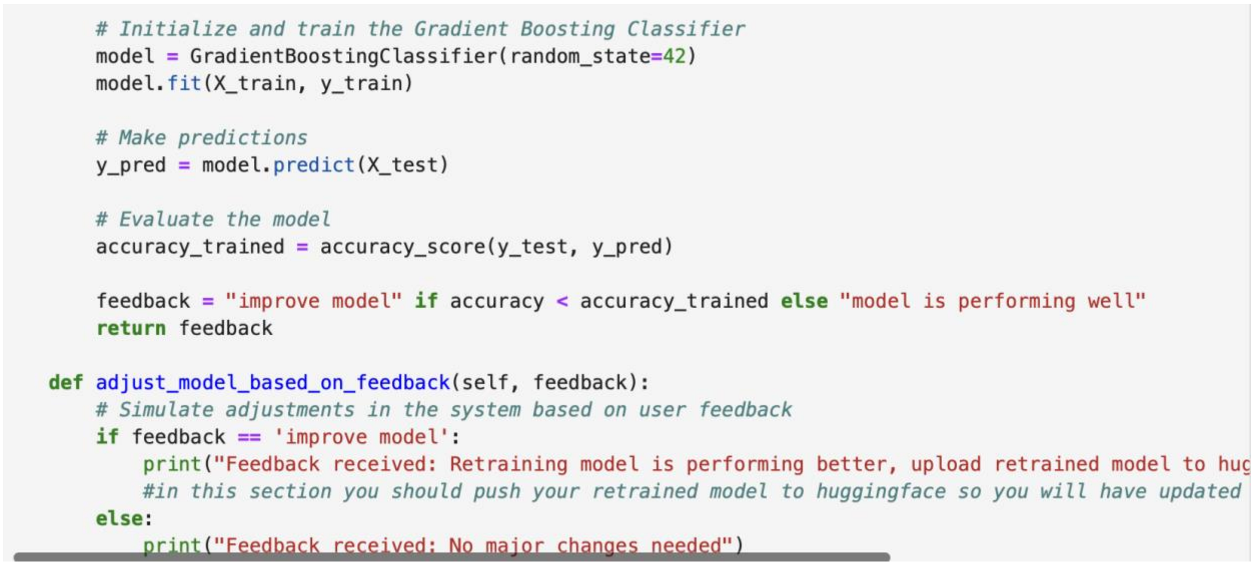


```python
def get_true_labels(dataset_url):
    data_ingestion = DataIngestion(dataset_url)
    data = data_ingestion.download_data()

    preprocessor = DataPreprocessor(data)
    cleaned_data = preprocessor.clean_data()

    # Extract true labels from the 'title' column
    true_labels = np.array(cleaned_data['title'])

    return true_labels
```

After the prediction user will watch a movie. We will update the title column with the name of movie in the exact dataset_url. After the update with get_true_labels function we will be able to get true result of what movie watched.

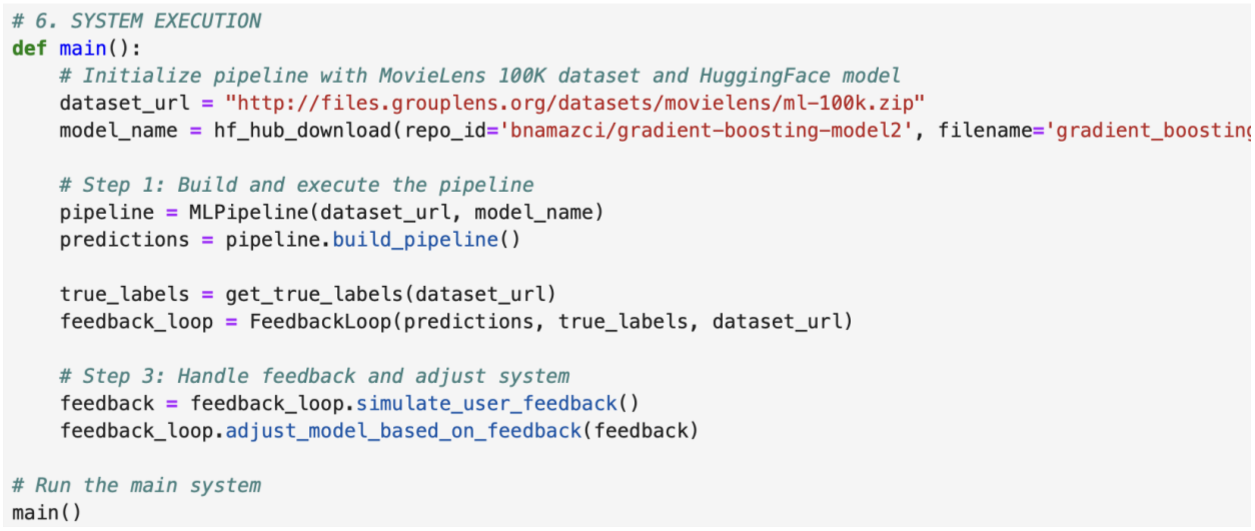


```python
# 5. FEEDBACK COMPONENT
class FeedbackLoop:
    def __init__(self, predictions, true_labels, dataset_url):
        self.predictions = predictions
        self.true_labels = true_labels
        self.data_ingestion = DataIngestion(dataset_url)

    def simulate_user_feedback(self):
        # actual accuracy of the model
        accuracy = accuracy_score(self.true_labels, self.predictions)

        # Step 1: Download and load data for training
        self.data = self.data_ingestion.download_data()

        # Step 2: Preprocess data
        self.preprocessor = DataPreprocessor(self.data)
        df = self.preprocessor.clean_data()

        # Define feature columns and target variable
        X = df.drop(columns=['title'])  # Features (excluding 'title')
        y = df['title']  # Target variable (title)

        # Split the dataset into training and testing sets
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```



```python
        # Initialize and train the Gradient Boosting Classifier
        model = GradientBoostingClassifier(random_state=42)
        model.fit(X_train, y_train)

        # Make predictions
        y_pred = model.predict(X_test)

        # Evaluate the model
        accuracy_trained = accuracy_score(y_test, y_pred)

        feedback = "improve model" if accuracy < accuracy_trained else "model is performing well"
        return feedback

    def adjust_model_based_on_feedback(self, feedback):
        # Simulate adjustments in the system based on user feedback
        if feedback == 'improve model':
            print("Feedback received: Retraining model is performing better, upload retrained model to hug
            #in this section you should push your retrained model to huggingface so you will have updated
        else:
            print("Feedback received: No major changes needed")
```

In this class, we set the true values, predictions, and the dataset URL using the `__init__` function. First, we calculate the accuracy score of the model. Then, we retrain our model using a dataset created by combining the data we receive from the user with the movies they watched. This way, we obtain the accuracy scores of both the retrained model and the original model. By comparing the accuracy scores, we get feedback. If the feedback indicates that the retrained model performs better, it updates the model on Hugging Face, but we haven't coded this part yet.



```python
# 6. SYSTEM EXECUTION
def main():
    # Initialize pipeline with MovieLens 100K dataset and HuggingFace model
    dataset_url = "http://files.grouplens.org/datasets/movielens/ml-100k.zip"
    model_name = hf_hub_download(repo_id='bnamazci/gradient-boosting-model2', filename='gradient_boosting

    # Step 1: Build and execute the pipeline
    pipeline = MLPipeline(dataset_url, model_name)
    predictions = pipeline.build_pipeline()

    true_labels = get_true_labels(dataset_url)
    feedback_loop = FeedbackLoop(predictions, true_labels, dataset_url)

    # Step 3: Handle feedback and adjust system
    feedback = feedback_loop.simulate_user_feedback()
    feedback_loop.adjust_model_based_on_feedback(feedback)

# Run the main system
main()
```

In the main function, we specify the data URL and model name and then call all the classes in chronological order. Afterward, we activated the pipeline and provided a movie recommendation to the user. Once the user selects and watches a movie, the *title* section of the dataset on the URL needs to be updated—a task that should be handled by the site from which we retrieve the data. After the dataset is updated, we obtain the true labels via a function and activate classes in the feedback system that compare our model with the retrained model.

UX and SYSTEM LOOP

We haven't made any developments on the UX side, but we should have a UX that initially takes inputs for the user's age, gender, and current genre of interest. The *title* field should be entered as null in the system. Then, our system will activate the 1st and 2nd classes to predict the data and, through the UX, recommend a movie to the user. This involves the 3rd and 4th classes functioning. After the user watches the movie, the title of the watched movie will be updated on the website, and our function along with the 5th class, which is the feedback system, will be triggered. If the system needs updating, the model will be updated, thus creating a loop system where the 3rd and 4th classes are refreshed.