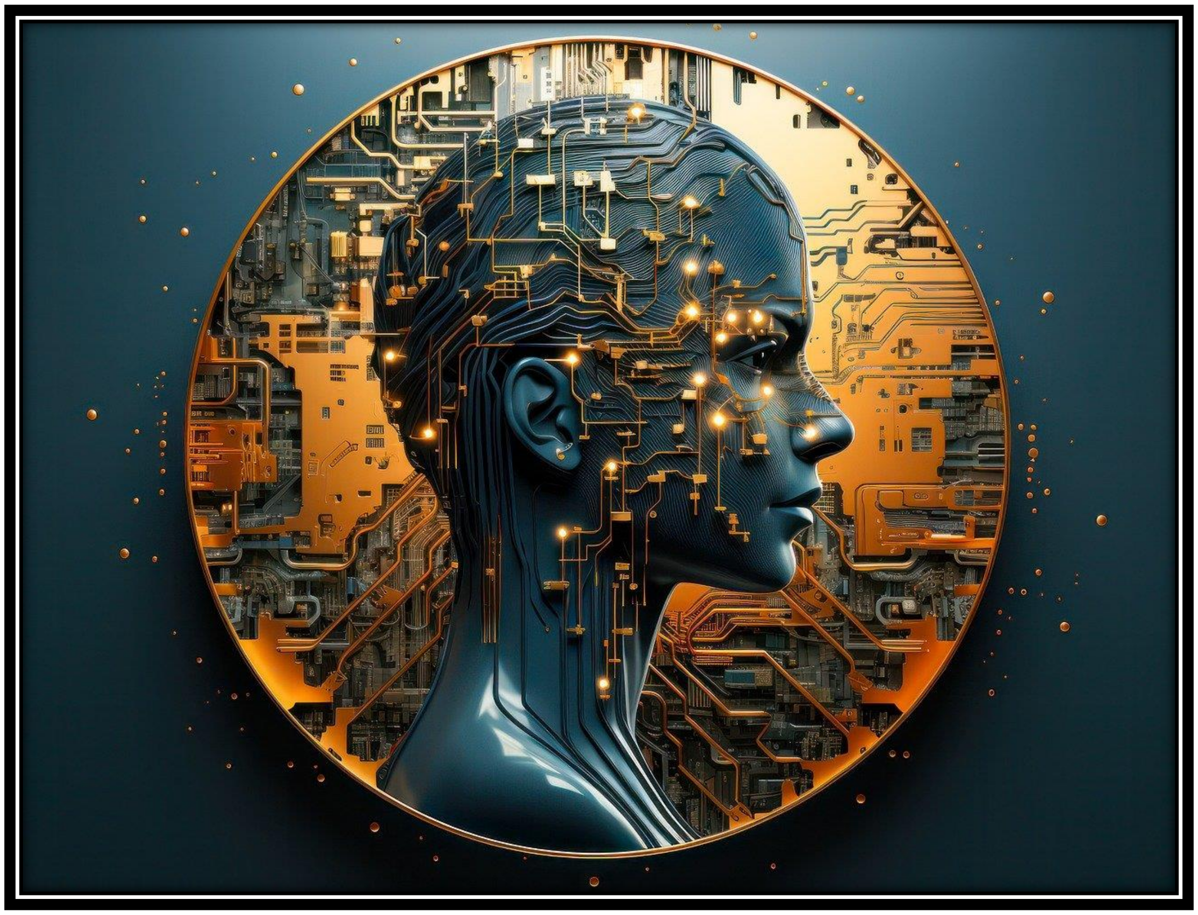


Collaborative Filtering Movie Recommendation System Using Deep Learning

By
Milan Mitrovic | s4663796

Contents

Problem Statement & Research Objectives

In the modern digital landscape, users are inundated with an overwhelming amount of content, whether it be movies, products, or media. Recommendation systems have become an essential tool to help users navigate this vast array of options by providing personalized suggestions. Collaborative filtering, a technique used in many recommendation systems, relies on user-item interaction data (such as ratings) to recommend content that a user might enjoy based on the preferences of similar users.

Traditional approaches like Matrix Factorization (learnable embeddings) have been widely adopted for collaborative filtering due to their simplicity and effectiveness. However, these models often struggle with scalability, handling complex relationships between users and items, and incorporating additional features such as movie content (e.g., genres, directors).

With the emergence of deep learning, more advanced techniques such as Neural Collaborative Filtering (NCF) have been developed to enhance the performance of recommendation systems. These deep learning models can capture more intricate patterns in user-item interactions by learning high-dimensional embeddings for users and items. Additionally, by combining collaborative filtering with content-based filtering, a hybrid recommender system can further improve recommendations by leveraging both user-item interaction data and item features.

This project aims to explore the effectiveness of a deep learning-based collaborative filtering system for movie recommendations using the *MovieLens 1M* Dataset. The goal is to compare the performance of this deep learning approach to traditional matrix factorization techniques and extend it by incorporating a hybrid recommendation system that combines collaborative filtering with content-based filtering.

Research Objectives

1. **Build a Deep Learning Recommender System:**
    - Develop a Collaborative Filtering Recommender System using a deep learning approach based on the MovieLens 1M dataset.
    - Utilize neural networks to learn user and movie embeddings, allowing the model to capture complex patterns in user preferences.

2. **Compare with Traditional Matrix Factorization:**
    - Implement a traditional Matrix Factorization model (e.g., using learnable embeddings) for collaborative filtering.
    - Evaluate and compare the performance of the deep learning model with the matrix factorization model using evaluation metrics such as RMSE (Root Mean

Squared Error), MAE (Mean Absolute Error), Precision@K, Recall@K, and NDCG (Normalized Discounted Cumulative Gain).

3. **Extend to a Hybrid Recommendation System:**
   - Integrate content-based filtering by incorporating additional movie metadata such as genres, etc.
   - Build a hybrid recommender system that combines both collaborative filtering (user-item interactions) and content-based filtering (movie metadata/user demographics) to improve the overall recommendation quality.

4. **Maximize Model Performance:**
   - Experiment with hyperparameter tuning to optimize the performance of all recommendation system models.
   - Test various architectures, embedding sizes, and regularization techniques to improve generalization.

5. **Evaluate and Analyse Performance:**
   - Assess the performance of both the deep learning models and the matrix factorization model using various metrics such as RMSE, MAE, Precision@K, Recall@K, and NDCG (Normalized Discounted Cumulative Gain).
   - Perform error analysis based on ranking metrics like Precision@K and Recall@K, and propose potential improvements for the model's ranking quality.

6. **Provide Insights Through Visualizations:**
   - Visualize the performance metrics and recommendation quality.
   - Compare the overall performance of the deep learning model (NCF) versus traditional matrix factorization using rating and ranking metrics.

Model Architecture & Design Decisions

Collaborative Filtering Recommender System with Deep Learning

The goal of this approach is to leverage deep learning to enhance traditional collaborative filtering by capturing more complex relationships between users and movies. In traditional methods, such as Matrix Factorization, user and item (movie) relationships are modelled in a linear fashion. Neural Collaborative Filtering (NCF) extends this by using non-linear neural networks to better model interactions.

Key Advantages of NCF:

- **Non-linearity**: Unlike matrix factorization, deep learning can model non-linear relationships, potentially capturing more intricate patterns in the data.
- **Flexibility**: The NCF architecture can incorporate side information (like content-based features) more easily when extended to a hybrid model.
- **Scalability**: Deep learning models can scale well to larger datasets and more complex tasks, which can lead to better recommendations.

Model Architecture

Inputs:

- User ID: Each user is represented by a unique ID, which is used as input.
- Movie ID: Each movie is also represented by a unique ID, which is used as input.

Both these inputs will be transformed into embeddings, which are vectors of learned parameters. These embeddings allow the model to represent users and movies in a high-dimensional space.

Embedding Layers:

- An embedding layer is a learned dense representation of the input. For both the user and the movie, we create embedding layers where each user and movie is mapped to a vector of a certain size. This enables the model to learn meaningful patterns in user and movie interactions.
  - User Embedding: Maps user IDs to a vector space of a chosen dimensionality (e.g., 50).
  - Movie Embedding: Maps movie IDs to a vector space of the same dimensionality.

Concatenation:

- The user and movie embeddings are concatenated (combined) into a single vector. This combined vector is then passed through the neural network to predict the rating or likelihood of interaction.

Hidden Layers (Dense Layers):

- Once the user and movie embeddings are concatenated, we pass the resulting vector through several fully connected (dense) layers. These layers use activation functions like ReLU (Rectified Linear Unit) to capture non-linear relationships.
  - Dense Layer 1: Applies a transformation with a ReLU activation, helping the model learn more complex patterns in the user-movie relationship.
  - Dense Layer 2: Another layer with a ReLU activation to further refine the learned interaction.

Output Layer:

- The output is a single value representing the predicted rating (for explicit feedback) or the probability of interaction (for implicit feedback).
- For explicit feedback (like movie ratings), the output is a continuous value (e.g., rating between 1 and 5).

Loss Function:

- Since we're predicting ratings, the loss function will be Mean Squared Error (MSE), which is commonly used in regression tasks.

Optimizer:

- Adam Optimizer is used for efficient gradient-based optimization, as it adapts the learning rate during training.

Design Decision:

The combination of embedding layers and fully connected layers allows the model to capture rich, non-linear interactions between users and movies. The embeddings compress information about users and movies into dense vectors, which the network can process more effectively than simple matrix factorization. Additionally, this architecture can be easily extended to incorporate additional features for the Hybrid Recommender System.

Matrix Factorization (Traditional Approach)

Matrix Factorization (MF) is one of the most common techniques used for collaborative filtering. The idea is to represent both users and items (movies) as vectors in a shared latent space, and the interaction (rating) between a user and a movie is predicted by computing the dot product between their corresponding vectors.

Model Architecture

User and Movie Vectors:

- In matrix factorization, each user and each movie are represented by vectors of a fixed length.
  - User vector: Represents latent factors of the user (e.g., how much a user likes action movies).
  - Movie vector: Represents latent factors of the movie (e.g., how much a movie possesses action content).

Dot Product:

- The predicted rating for a user-movie pair is calculated by taking the dot product of the user and movie vectors. This operation essentially measures the similarity between the user and the movie in the latent factor space.

Bias Terms:

- In addition to the dot product, bias terms were added to account for the user's average rating tendencies and the movie's overall popularity. These bias terms help the model to capture overall tendencies that are not reflected in the latent factors alone.
  - User bias: Captures how much a user tends to rate movies higher or lower on average.
  - Movie bias: Captures whether certain movies tend to be rated higher or lower regardless of the user.

Loss Function:

- The model aims to minimize the difference between the actual rating and the predicted rating. The loss function used is Mean Squared Error (MSE), Again, which is a common regression loss function for rating prediction tasks.

<u>Design Decision:</u>

Matrix Factorization is a simple and effective technique that works well for collaborative filtering, especially when you have a large amount of user-item interaction data. However, it models user-movie interactions linearly and might struggle to capture more complex relationships, which is why deep learning approaches like NCF are often preferred for more advanced recommendation systems.

Hybrid Recommender System (Collaborative + Content-Based Filtering)

A hybrid recommender system combines collaborative filtering (based on user-item interactions) with content-based filtering (which uses item features like genres, directors, or actors). This combination can improve the recommendation system's ability to handle cold start problems (when interaction data is limited for a new user or item) and provide more diverse recommendations.

<u>Model Architecture</u>

Collaborative Filtering Component:

- This part of the model is the NCF architecture discussed previously, which uses user-movie interaction data (ratings).

Content-Based Filtering Component:

- In content-based filtering, the recommendation system uses movie metadata (e.g., genres, gender, age, occupation, and zip code) to recommend movies similar to those the user has liked in the past.
- Genres are one-hot encoded as part of the pre-processing step, and they are incorporated into the model as additional features.

Combining Both Components:

- The two components (Collaborative Filtering and Content-Based Filtering) can be combined:
    - Concatenation of features: Concatenate the user-item embeddings from collaborative filtering with the content-based features (like movie genres, user demographics) and pass them through additional dense layers to predict the final rating.

Design Decision:

A hybrid system helps mitigate the weaknesses of each individual approach. Collaborative filtering suffers from the cold start problem (struggles to recommend new items that have little to no interaction data), while content-based filtering can be limited by the available item features. By combining the two, the system can recommend new or niche movies more effectively, as well as provide personalized recommendations based on both user preferences and movie characteristics.

Summary of Model Design Decisions:

- **Collaborative Filtering with Deep Learning**: Utilizes embedding layers for users and movies, followed by fully connected layers to predict ratings, allowing the model to capture non-linear relationships and improve recommendation accuracy.
- **Matrix Factorization**: A simpler, traditional approach that uses dot product of latent factors (user and movie vectors) to predict ratings. While effective, it can struggle to capture complex patterns.
- **Hybrid Recommender System**: Combines NCF with a content-based filtering component that incorporates movie metadata, providing more robust and diverse recommendations.

These models will be evaluated using performance metrics like RMSE, MAE, and ranking-based metrics (Precision@k, Recall@k, NDCG (Normalized Discounted Cumulative Gain), ensuring a thorough comparison between the deep learning and traditional approaches.

Data Preprocessing & Augmentation Techniques

Dataset Description (Users, Movies, Ratings):

Feature	Sample	Description
UserID	1	Unique identifier of a user.
MovieID	1193	Unique identifier of a movie.
Rating	5	The rating given to a movie by a user. Ranging from 1-5.
Timestamp	978300760	Unix timestamp of when the rating was given.
Gender	F	Gender of the user (F/M).

Age	1	Age group of the user.
Occupation	10	Occupation code of the user – represents job categories.
Zip-code	48067	Zip code of the user.
Title	One Flew Over the Cuckoo's Nest (1975)	Title of the movie.
Genres	Drama	Genre of the movie.

The MovieLens 1M Dataset consists of 1 million ratings from over 6,000 users for about 4,000 movies. It contains the following types of information:

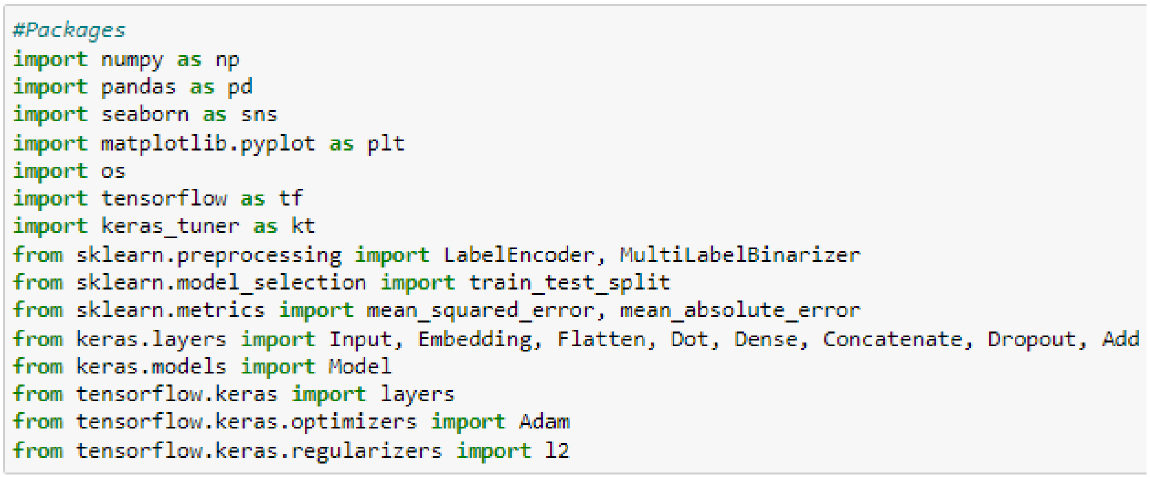
- **User data**: Information about the users, such as user ID, gender, age, occupation, and zip code.
- **Movie data**: Information about movies, such as movie ID, title, and genres.
- **Ratings data**: User ratings for movies, which are the core data used for building the recommendation system. These ratings are on a scale of 1 to 5 (before pre-processing).

The dataset contains all the essential features needed to build a movie recommendation system that predicts user preferences and recommends movies they are likely to enjoy by accurately predicting how users will rate movies they haven't seen yet. Currently, the dataset contains 10 key features which can be analyzed and used to predict user ratings for movies. This will be facilitated by our recommendation system models by leveraging the user-movie interaction data for training, teaching them to identify patterns in user preferences and movie characteristics. Once trained, the recommendation models will be applied to new, unseen user-movie pairs that do not already have a rating; simulating real-world scenarios where a recommender system must predict user preferences and recommend movies they have not rated before. Enabling the models to recommend highly-rated movies the user hasn't watched yet. Essentially, this process is driven by predicting the Rating feature (target feature), which allows us to understand user preferences and make personalized recommendations. Basically, all of our models will be trained to predict the Rating that a user would give to a movie, based on either user-item interactions (collaborative filtering) or a combination of interactions and movie metadata (hybrid approach).

Data Pre-Processing:

Data pre-processing is a crucial step in building an effective recommendation system, and it must be conducted thoroughly before training the models. This is because improperly pre-processed data, such as missing, inconsistent, or unformatted data can negatively impact model performance, leading to inaccurate predictions and suboptimal recommendations. By ensuring that the user-item interaction data and movie metadata are clean, well-structured, and appropriately formatted, we can help the recommendation models identify patterns in user preferences and movie characteristics, ultimately leading to better predictions of user ratings and more accurate movie recommendations.

First, we need to import the necessary packages along with the datasets.

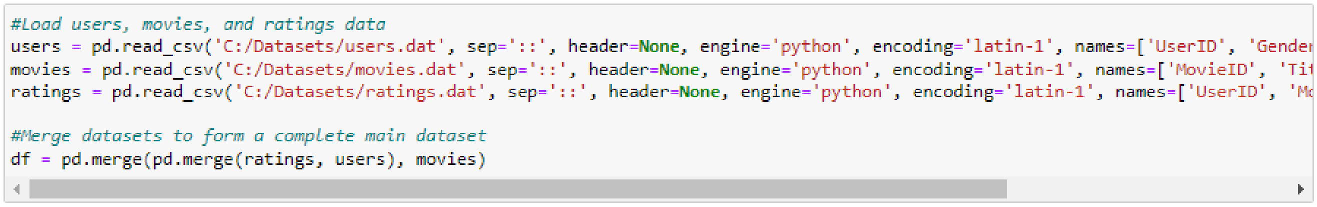


```
#Packages
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import os
import tensorflow as tf
import keras_tuner as kt
from sklearn.preprocessing import LabelEncoder, MultiLabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error
from keras.layers import Input, Embedding, Flatten, Dot, Dense, Concatenate, Dropout, Add
from keras.models import Model
from tensorflow.keras import layers
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l2
```

- **NumPy** - NumPy is a fundamental package for scientific computing with Python. It is essential to import NumPy before using other key data science libraries, as many of them are built upon NumPy's foundation.
- **Pandas** - Pandas is a powerful data analysis and manipulation library for Python. It provides essential data structures and functions to seamlessly manipulate structured data. It is necessary to import Pandas for reading and writing all of our datasets. It is also a prerequisite to perform comprehensive data analytics.
- **Seaborn** - Seaborn is a Python visualization library based on Matplotlib that provides a high-level interface for illustrating appealing and statistical plots. It complements Matplotlib and is essential for depicting a variety of plots in data analytics.
- **Matplotlib** - Matplotlib is an extensive library to create static, animated, and interactive graphs in Python. It is a prerequisite to create and configure plots in data analytics. Again, it also serves as a foundation for Seaborn – working together to create comprehensive plots of all variety.

- **OS** - The OS package in Python provides a way to interact with our operating system. It is necessary to implement this package to read and write our datasets.

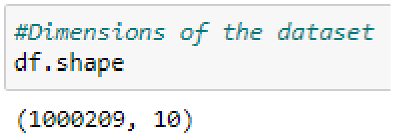
In addition to the fundamental packages mentioned above, we also imported several other critical libraries necessary for data pre-processing, augmentation, model training, evaluation, and hyperparameter tuning. These include Keras for building deep learning models, TensorFlow as the underlying framework for neural network operations, Scikit-learn for data encoding and splitting, and Keras Tuner for optimizing the model through hyperparameter tuning. Together, these packages ensure a seamless workflow for completing all steps in the project,



```
#Load users, movies, and ratings data
users = pd.read_csv('C:/Datasets/users.dat', sep='::', header=None, engine='python', encoding='latin-1', names=['UserID', 'Gende
movies = pd.read_csv('C:/Datasets/movies.dat', sep='::', header=None, engine='python', encoding='latin-1', names=['MovieID', 'Ti
ratings = pd.read_csv('C:/Datasets/ratings.dat', sep='::', header=None, engine='python', encoding='latin-1', names=['UserID', 'M

#Merge datasets to form a complete main dataset
df = pd.merge(pd.merge(ratings, users), movies)
```

Importing the three key datasets (users, movies, ratings) from MovieLens 1M dataset by utilizing the read.csv() method from Pandas to read the users, movies, and ratings datasets. I specify parameters such as sep='::', header=None, and encoding='latin-1'. The sep='::' parameter indicates that double colons (::) are used as the delimiter to separate values in each file, while header=None informs Pandas that the file does not contain column headers, so column names are manually assigned to specific values using the names=[] parameter. The encoding='latin-1' parameter ensures that special characters, especially in movie titles, are handled correctly. These parameters were necessary as they correspond to the specific layouts of the MovieLens 1M datasets.

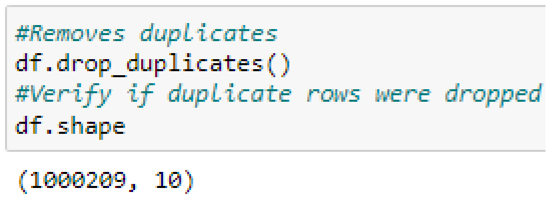
After loading the data into separate data frames, I use the pd.merge() method from Pandas to combine the ratings, users, and movies datasets. First, the ratings and users datasets are merged on the UserID feature, and then the resulting dataset is merged with the movies dataset on the MovieID feature. This produces a unified data frame, assigned to the variable 'df', containing all the relevant user, movie, and rating data necessary for further analysis and modelling.



```
#Dimensions of the dataset
df.shape

(1000209, 10)
```

Currently, the dataset dimensions before data pre-processing: 1,000,209 rows and 10 columns. This is verified by the shape attribute from Pandas which returns the dimensions of the dataset.

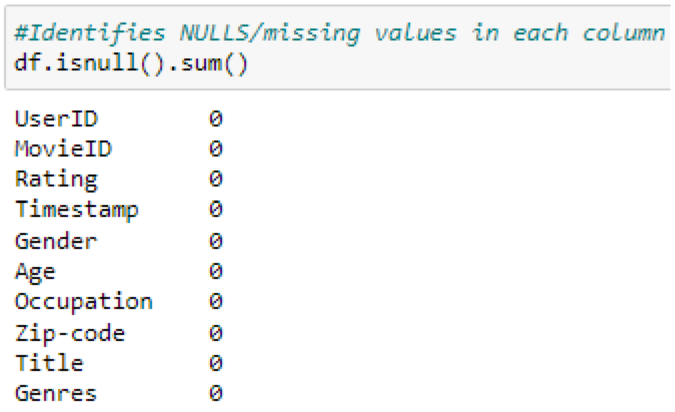
Identifying and removing duplicates if they are present with the drop_duplicates() method from Pandas:



```
#Removes duplicates
df.drop_duplicates()
#Verify if duplicate rows were dropped
df.shape
```

```
(1000209, 10)
```

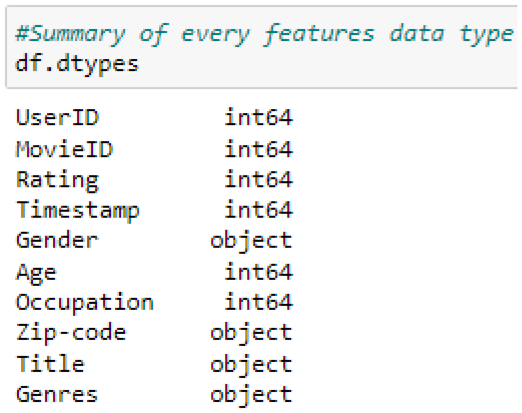
There are no duplicates present in the dataset.



```
#Identifies NULLS/missing values in each column
df.isnull().sum()
```

```
UserID        0
MovieID       0
Rating        0
Timestamp     0
Gender        0
Age           0
Occupation    0
Zip-code      0
Title         0
Genres        0
```

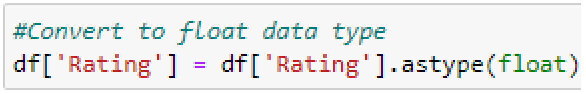
Checking for NULLS/missing values within the dataset by utilizing the isnull() and sum() methods – there is none.



```
#Summary of every features data type
df.dtypes
```

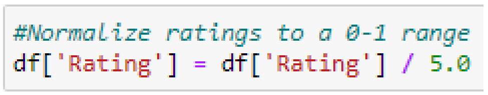
```
UserID          int64
MovieID         int64
Rating          int64
Timestamp       int64
Gender         object
Age             int64
Occupation      int64
Zip-code       object
Title          object
Genres         object
```

Identifying the data types for all features within the dataset and ascertaining if they need to be amended by using the dtypes attribute from Pandas. All data types are valid and align coherently to their specific feature, however the Rating feature should be transformed into a float data type as it will be normalized later. Rating needs to be converted to a float before normalizing because normalization typically produces values that are fractional (decimal values), and integers cannot represent these decimals accurately; an integer data type would round these decimals to whole numbers, which would result in a loss of information – negatively impacting our recommender system models performance. Essentially, our models need Rating to be converted to a float form for better precision when normalized.



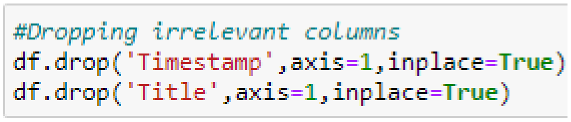
```
#Convert to float data type
df['Rating'] = df['Rating'].astype(float)
```

- Transforming Rating from its original integer data type to its appropriate float data type – ready for normalization. This conversion was facilitated by using the astype() method from Pandas. It works by transforming the specified column to the specified data type (float in our case).



```
#Normalize ratings to a 0-1 range
df['Rating'] = df['Rating'] / 5.0
```

- Normalizing the Rating feature scale from 1-5 to a 0-1 scale. Essentially, I am manually applying Min-Max normalization here by dividing each value in the Rating column by 5 – more efficient. Normalization is a crucial step in machine learning, particularly for our deep learning recommender system models (NCF, Hybrid) because it ensures that all inputs are on a similar scale. Deep learning models use activation functions (like ReLU in our case) that work best when the inputs are within a small, consistent range. If the ratings remain between 1 and 5, it could slow down convergence during training because the model may struggle to adjust its parameters effectively. Our models will perform better when inputs are normalized. Scaling ratings from 1-5 to 0-1 boosts model convergence and enhances gradient-based optimization algorithm performance, which we will be utilizing later for optimization (Adam). It ensures that ratings are consistent and standardized across the data. Basically, normalizing ratings ensures consistency and improves model training performance, especially for neural networks.



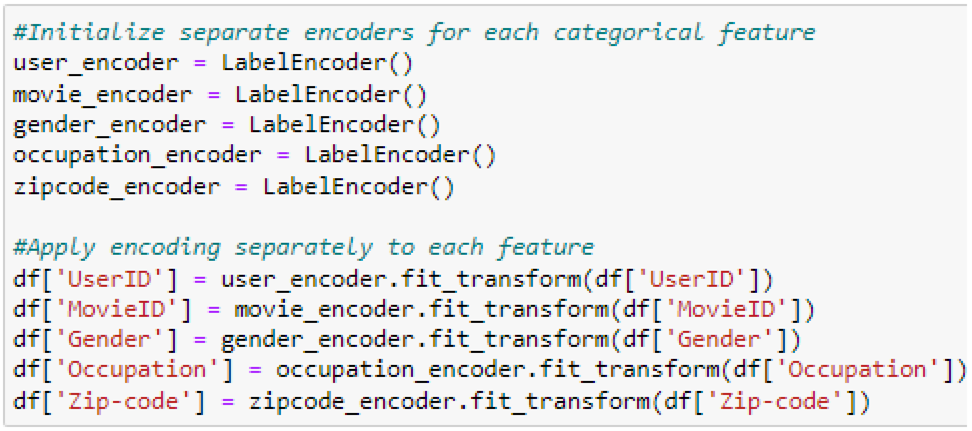
```
#Dropping irrelevant columns
df.drop('Timestamp',axis=1,inplace=True)
df.drop('Title',axis=1,inplace=True)
```

Irrelevant features removed:

- **Timestamp**: We drop Timestamp due to its irrelevancy; In collaborative filtering and hybrid recommender system models, the core focus is on user-item interactions and content features such as genres, rather than when the interactions occurred – doesn't require time-based features. Including irrelevant features like Timestamp could lead to noise that might distract the model from focusing on the features that actually drive performance (like UserID, MovieID, Rating, and Genres, etc.). Ultimately, Timestamp doesn't provide any valuable information for our models. Removing Timestamp avoids irrelevant information feeding into our models – reducing noise to our models, subsequently enhancing our modelling performance.

- **Title**: We remove the Title feature mainly because of its redundancy; Title is redundant because the MovieID feature uniquely identifies each movie in the dataset. Since the model will use MovieID to represent and learn the movie-related information (via embeddings), the Title does not provide any additional benefit beyond what MovieID already offers. Furthermore, the title of a movie is just a string identifier and doesn't carry any meaningful information that can improve the model's predictive ability. Essentially, if we kept both Title and MovieID it would introduce redundancy because both features are essentially serving the same purpose (identifying the movie). The models don't need two identifiers for the same movie – mitigating noise, which would otherwise adversely affect our recommendation system models performance.

I deal with both irrelevant features in the same way, utilizing the drop() method from Pandas. it works by removing the specified column – in our case the columns above. Overall, this is done to improve data quality, whilst simultaneously reducing noise, dimensionality, and optimizing modelling performance.



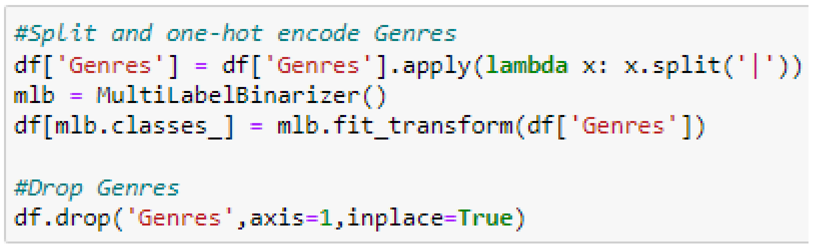
```
#Initialize separate encoders for each categorical feature
user_encoder = LabelEncoder()
movie_encoder = LabelEncoder()
gender_encoder = LabelEncoder()
occupation_encoder = LabelEncoder()
zipcode_encoder = LabelEncoder()

#Apply encoding separately to each feature
df['UserID'] = user_encoder.fit_transform(df['UserID'])
df['MovieID'] = movie_encoder.fit_transform(df['MovieID'])
df['Gender'] = gender_encoder.fit_transform(df['Gender'])
df['Occupation'] = occupation_encoder.fit_transform(df['Occupation'])
df['Zip-code'] = zipcode_encoder.fit_transform(df['Zip-code'])
```

Encoding Categorical Data: In machine learning models, especially in deep learning for recommendation systems, the model can only process numerical inputs. Features like UserID, MovieID, Gender, Occupation, and Zip-code are either inherently categorical or represent categorical concepts. Even if some of these features are already represented as numbers, the model still needs to understand them as categories, not as continuous numerical values.

- Gender needed to be encoded because its values contain words ('M' for Male, 'F' for Female), and machine learning models can't process words, only numbers.

- UserID, MovieID, Occupation, and Zip-code were encoded to ensure our models understand them correctly. These features represent distinct categories and leaving them as-is might cause the models to mistakenly assume there's an ordinal or continuous relationship between the numbers (e.g., the model might assume that a higher value for Occupation implies a better or more significant occupation). In reality, occupations, zip codes, and user/movie IDs are simply labels with no inherent numerical order or significance.

Label Encoding ensures that these features are treated appropriately as categorical variables, where the model understands each as a distinct category, not a continuous value. The encoding was achieved by utilizing LabelEncoder, a tool from the sklearn pre-processing package that converts categorical features into numerical labels. It assigns each unique category in a feature to a distinct integer value. The encoding worked by initializing the LabelEncoder for all features separately. Then, the fit_transform() method is applied to each categorical feature. It fits the encoder to the unique values of each column (learning the unique categories) and transforms the column by replacing each unique category with a corresponding integer label.

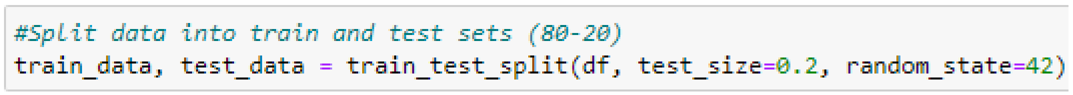


```
#Split and one-hot encode Genres
df['Genres'] = df['Genres'].apply(lambda x: x.split('|'))
mlb = MultiLabelBinarizer()
df[mlb.classes_] = mlb.fit_transform(df['Genres'])

#Drop Genres
df.drop('Genres',axis=1,inplace=True)
```

- The last feature that requires encoding is Genres, but unlike the other features, Genres needs special handling due to its structure. The genres for each movie are stored as a single string, with multiple genres separated by the pipe symbol (|), for example, "Action|Comedy|Drama". Before encoding, it's necessary to split these genre strings into individual components. Each movie can belong to multiple genres (e.g., Action, Comedy, Drama), so simply label encoding them would not capture this multi-label nature. Instead, we use one-hot encoding via MultiLabelBinarizer. The MultiLabelBinarizer is specifically designed for handling multi-label categorical data, converting it into a set of binary columns where each column represents a genre. For each movie, the models assigns a 1 to the genres it belongs to and a 0 to the others. This process is critical for our hybrid recommendation system model, where movie genres provide valuable content-based information. By one-hot encoding genres, we allow the model to include movie content in its predictions, thereby enhancing the quality of recommendations beyond user-item interactions alone. One-hot encoding is necessary in this case because it preserves the multi-label relationships between genres, which would be lost if we used label encoding. Label encoding would assign a single number to each unique genre combination, misrepresenting the individual genres and their importance. In contrast, one-hot encoding gives each genre its own binary feature, ensuring that the model can capture these complex content-based relationships accurately.

- The one-hot encoding was achieved by first splitting each string in the Genres column into a list of genres using the apply function along with a lambda function. Followed up by initializing the MultiLabelBinarizer – a tool from the sklearn pre-processing package. Then, the fit_transform() method was then applied to the Genres column. The fit() part learns all the unique genre classes across the dataset, storing these unique values in the mlb.classes_ attribute. The transform() part converts each movie's genre data into a binary array, where each column corresponds to a genre, and each movie has a 1 if it belongs to that genre and 0 otherwise. Finally, we utilize the drop() method to remove the Genres column as it's no longer needed due to its redundancy.

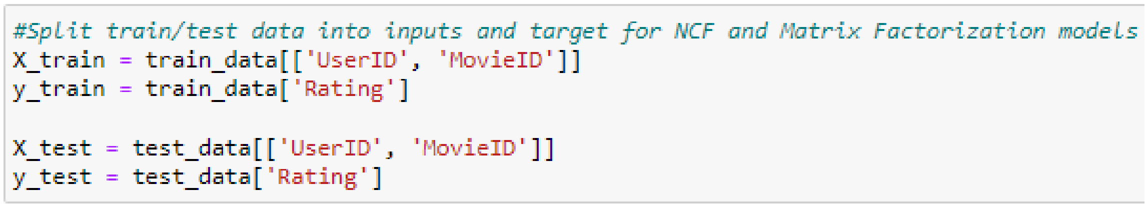
Ultimately, this encoding process was essential so our recommender system models can interpret the categorical features correctly and process them as distinct entities, enabling the model to learn meaningful relationships and patterns from the data.



```
#Split data into train and test sets (80-20)
train_data, test_data = train_test_split(df, test_size=0.2, random_state=42)
```

We split our data into train and test sets by using the train_test_split() method from sklearn. The method splits the data into train_data (80%) and test_data (20%). The train_data will be used to train our recommendation system models, while the test_data is used to evaluate the models performance. In our recommendation system models, the entire dataset represents user-item interactions, where each interaction involves a UserID, MovieID, and Rating (along with other possible features like genres). The goal of the model is to predict how users will rate movies they haven't seen, based on their historical ratings and interactions with other movies. During training, the model learns from the user-movie interactions in the training set. It captures latent patterns and relationships between users and movies, such as user preferences and movie characteristics. This allows the model to understand the underlying factors that drive user behavior and helps it predict future interactions. Once the model is trained, we assess its performance on the test set, which contains unseen user-movie interactions. This split ensures that we can evaluate the model’s ability to generalize and make accurate recommendations for movies that users have not previously rated. By comparing the predicted ratings with the actual ratings in the test set, we can measure the model’s effectiveness in recommending movies to users.

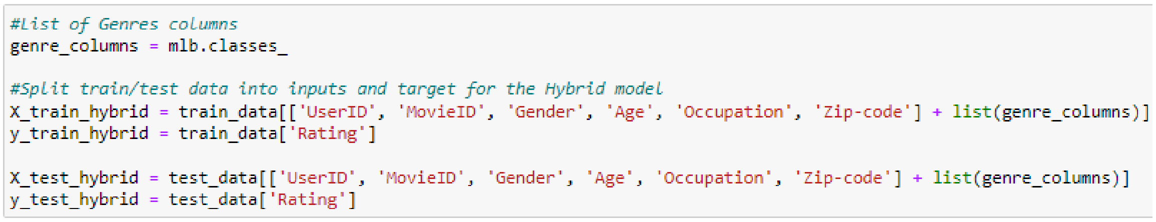
In order to accommodate the different input requirements for our recommendation system models, we need to further refine how we split the data into train and test sets for each type of model:



```
#Split train/test data into inputs and target for NCF and Matrix Factorization models
X_train = train_data[['UserID', 'MovieID']]
y_train = train_data['Rating']

X_test = test_data[['UserID', 'MovieID']]
y_test = test_data['Rating']
```

- The Collaborative Filtering recommendation system models (Neural Collaborative Filtering and Matrix Factorization), rely exclusively on user-item interaction data. The models assume that the latent preferences of a user and the latent characteristics of a movie can be inferred purely from the user’s interactions with items (their historical ratings). For this reason, we only need to provide UserID and MovieID as the input features, and the model will learn latent representations (embeddings) for users and movies. This allows the models to predict how a user will rate a movie based on patterns it has learned from other users and movies. Ratings are the target feature the models are trying to predict based on the input features. Since collaborative filtering models operate purely on user-item interaction data, there’s no need to include additional features when preparing the inputs for these models.

- To achieve the data split for the Collaborative Filtering models (Neural Collaborative Filtering and Matrix Factorization), we first extract the UserID and MovieID columns from the train_data to form X_train, which will serve as the input features during training. Then, we extract the Rating column from train_data to create y_train, which holds the target ratings for the model to predict. Similarly, we extract UserID and MovieID from test_data to form X_test for testing, and the Rating column from test_data to create y_test, which will be used to evaluate the model's performance. This splits the data into inputs (user-item interactions) and the target (ratings) for both training and testing.



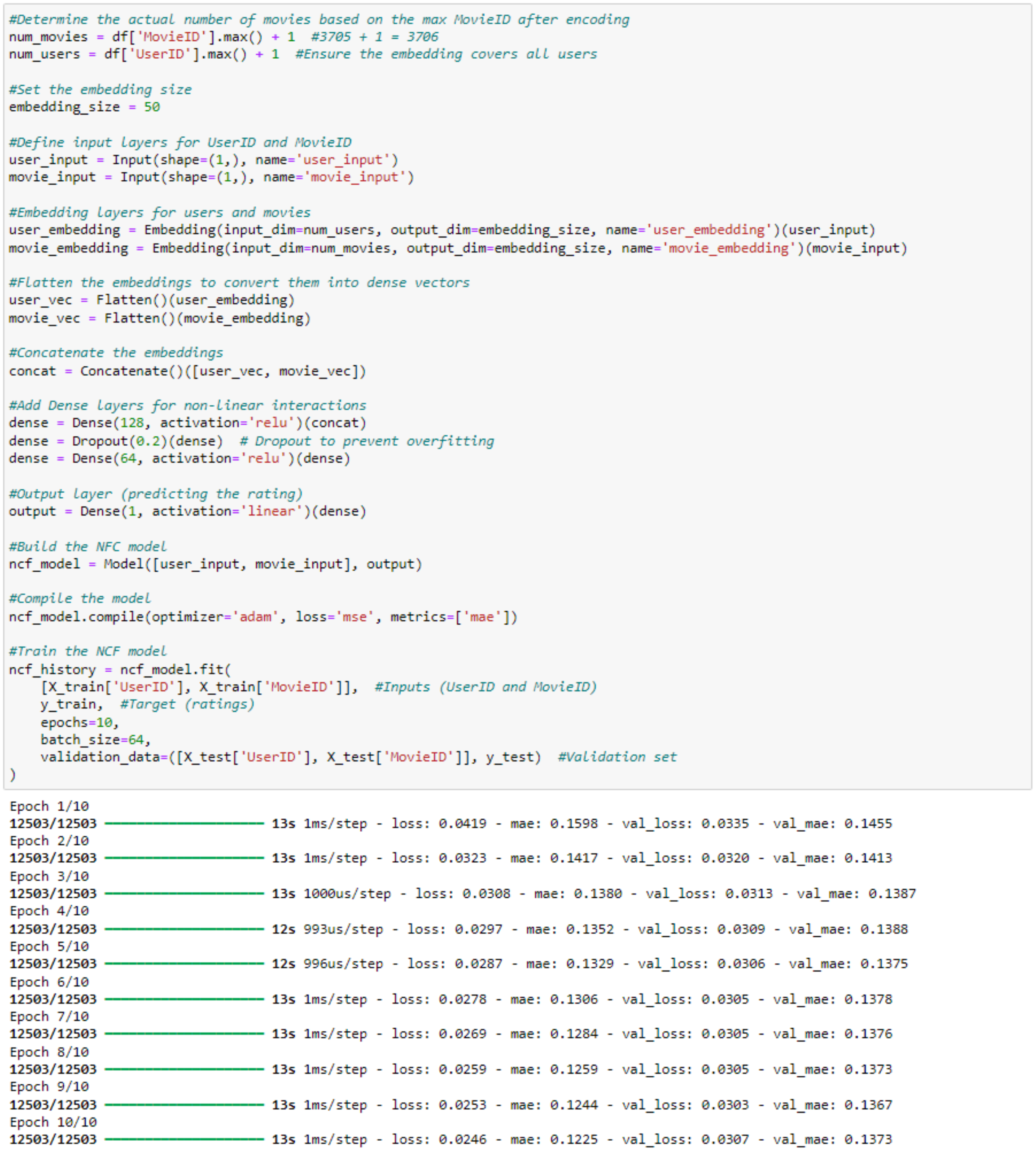
```
#List of Genres columns
genre_columns = mlb.classes_

#Split train/test data into inputs and target for the Hybrid model
X_train_hybrid = train_data[['UserID', 'MovieID', 'Gender', 'Age', 'Occupation', 'Zip-code'] + list(genre_columns)]
y_train_hybrid = train_data['Rating']

X_test_hybrid = test_data[['UserID', 'MovieID', 'Gender', 'Age', 'Occupation', 'Zip-code'] + list(genre_columns)]
y_test_hybrid = test_data['Rating']
```

- The Hybrid recommendation system model combines both collaborative filtering and content-based filtering, which means it uses user-item interaction data (like UserID and MovieID) alongside additional metadata (such as user demographics and movie genres). For this reason, the train-test split for the hybrid model includes not just UserID and MovieID, but also other features such as Gender, Age, Occupation, Zip-code, and the one-hot encoded Genres (e.g., Action, Comedy, Drama, etc.). The Hybrid model combines the strengths of collaborative filtering (user-item interactions) with content-based filtering (metadata like genres or user demographics). By including additional features like genres and user demographics, the model can make more informed recommendations. These features help the model understand who the user is and what kind of movies the user might prefer, beyond just the user's interaction history. This is particularly useful in situations where there isn't enough user-item interaction data (the cold-start problem), such as when a user is new to the system or when a movie hasn't been rated by many users. In these cases, the model can fall back on content-based features like genres to make better recommendations. Essentially, since the Hybrid model utilizes both filtering techniques, we use additional features for the train and test sets to maximize results.

- To achieve the Hybrid model data split, we first created a list of the one-hot encoded genre columns using mlb.classes_, which contains the binary representations of the various movie genres. These genre columns represent additional movie metadata that will be included in the model alongside other features. Next, we prepared the input features for the Hybrid Model by combining the UserID and MovieID with relevant user demographic features (Gender, Age, Occupation, Zip-code) and the one-hot encoded genre columns. These features collectively provide the model with both user-item interaction data and additional movie content-based metadata. Finally, we split the data into the train and test sets, where the input features (X_train_hybrid, X_test_hybrid) include all the user, movie, and genre-related features, and the target variable (y_train_hybrid, y_test_hybrid) is the Rating target feature that the model will predict. This setup allows the Hybrid Model to learn from both collaborative filtering data (user-item interactions) and content-based data (movie metadata and user demographics).

Building and training our recommendation system models with Keras:

1. **NCF Model:**



```
#Determine the actual number of movies based on the max MovieID after encoding
num_movies = df['MovieID'].max() + 1  #3705 + 1 = 3706
num_users = df['UserID'].max() + 1  #Ensure the embedding covers all users

#Set the embedding size
embedding_size = 50

#Define input layers for UserID and MovieID
user_input = Input(shape=(1,), name='user_input')
movie_input = Input(shape=(1,), name='movie_input')

#Embedding layers for users and movies
user_embedding = Embedding(input_dim=num_users, output_dim=embedding_size, name='user_embedding')(user_input)
movie_embedding = Embedding(input_dim=num_movies, output_dim=embedding_size, name='movie_embedding')(movie_input)

#Flatten the embeddings to convert them into dense vectors
user_vec = Flatten()(user_embedding)
movie_vec = Flatten()(movie_embedding)

#Concatenate the embeddings
concat = Concatenate()([user_vec, movie_vec])

#Add Dense layers for non-linear interactions
dense = Dense(128, activation='relu')(concat)
dense = Dropout(0.2)(dense)  # Dropout to prevent overfitting
dense = Dense(64, activation='relu')(dense)

#Output layer (predicting the rating)
output = Dense(1, activation='linear')(dense)

#Build the NFC model
ncf_model = Model([user_input, movie_input], output)

#Compile the model
ncf_model.compile(optimizer='adam', loss='mse', metrics=['mae'])

#Train the NCF model
ncf_history = ncf_model.fit(
    [X_train['UserID'], X_train['MovieID']],  #Inputs (UserID and MovieID)
    y_train,  #Target (ratings)
    epochs=10,
    batch_size=64,
    validation_data=([X_test['UserID'], X_test['MovieID']], y_test)  #Validation set
)
```

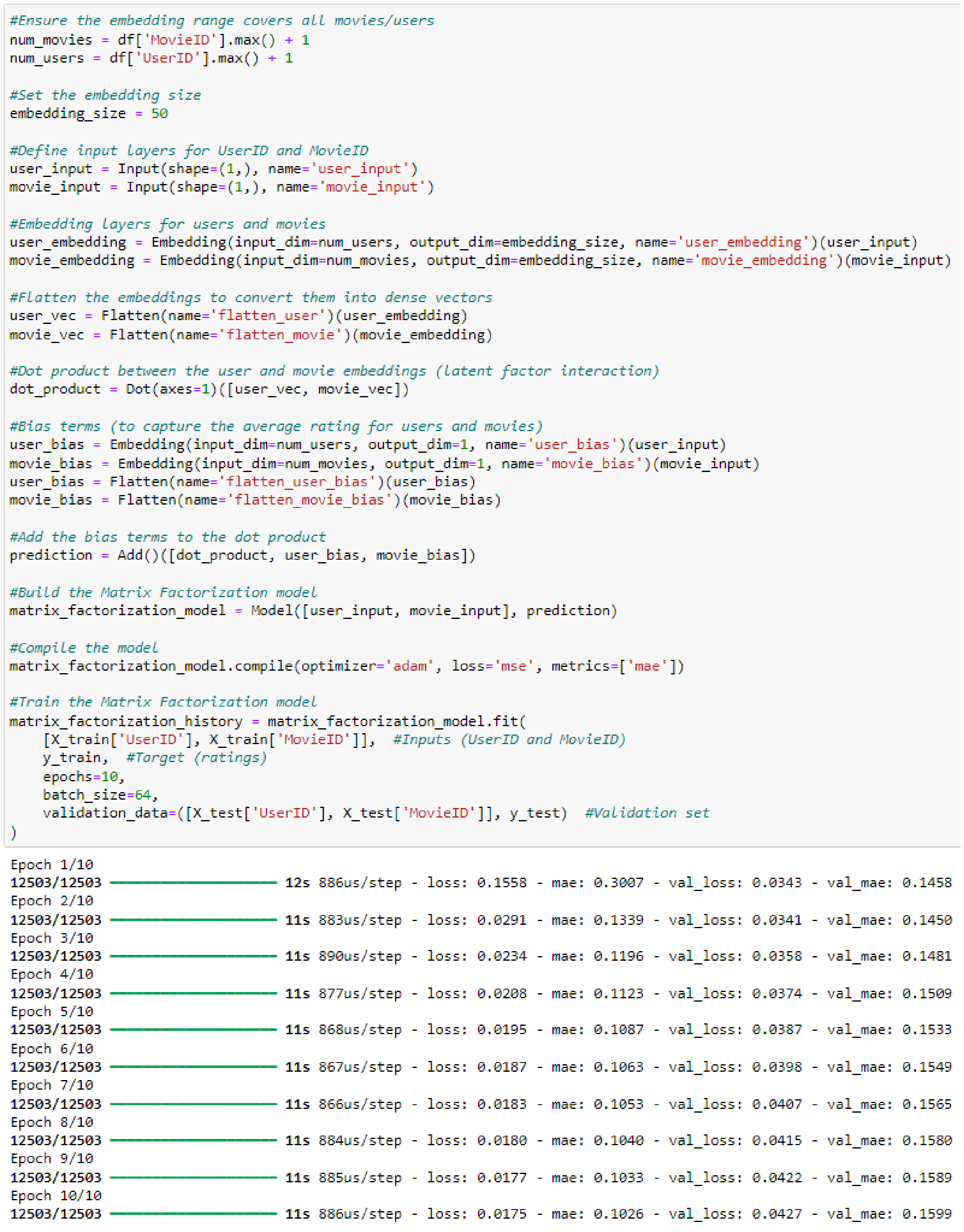
```
Epoch 1/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 13s 1ms/step - loss: 0.0419 - mae: 0.1598 - val_loss: 0.0335 - val_mae: 0.1455
Epoch 2/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 13s 1ms/step - loss: 0.0323 - mae: 0.1417 - val_loss: 0.0320 - val_mae: 0.1413
Epoch 3/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 13s 1000us/step - loss: 0.0308 - mae: 0.1380 - val_loss: 0.0313 - val_mae: 0.1387
Epoch 4/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 12s 993us/step - loss: 0.0297 - mae: 0.1352 - val_loss: 0.0309 - val_mae: 0.1388
Epoch 5/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 12s 996us/step - loss: 0.0287 - mae: 0.1329 - val_loss: 0.0306 - val_mae: 0.1375
Epoch 6/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 13s 1ms/step - loss: 0.0278 - mae: 0.1306 - val_loss: 0.0305 - val_mae: 0.1378
Epoch 7/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 13s 1ms/step - loss: 0.0269 - mae: 0.1284 - val_loss: 0.0305 - val_mae: 0.1376
Epoch 8/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 13s 1ms/step - loss: 0.0259 - mae: 0.1259 - val_loss: 0.0305 - val_mae: 0.1373
Epoch 9/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 13s 1ms/step - loss: 0.0253 - mae: 0.1244 - val_loss: 0.0303 - val_mae: 0.1367
Epoch 10/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 13s 1ms/step - loss: 0.0246 - mae: 0.1225 - val_loss: 0.0307 - val_mae: 0.1373
```

- The Neural Collaborative Filtering (NCF) model is built using Keras, a high-level deep learning library, which simplifies the construction of neural networks. To begin with, the number of unique users and movies in the dataset is determined, setting the dimensions for the embedding layers. Using the Embedding layer in Keras, both user and movie IDs are mapped to dense vectors (embeddings) that represent latent factors for each user and movie in a shared space. These embeddings capture the preferences of users and the characteristics of movies in a high-dimensional vector format.

- Once the embeddings are created, they are flattened using Keras' Flatten() function and concatenated using Concatenate(), combining the user and movie information into a single vector. This combined vector is passed through two fully connected layers (implemented with Keras' Dense() layers), where ReLU (Rectified Linear Unit) activation functions are applied to introduce non-linearities and allow the model to learn complex patterns in the user-movie interactions. A Dropout() layer is added to prevent overfitting by randomly disabling a fraction of neurons during training, ensuring the model generalizes better to unseen data.
- The final output is produced through a Dense() layer with a linear activation, which predicts a continuous value representing the user's rating for the movie. The model is then compiled using Keras' compile() function, with the Adam optimizer (which adjusts learning rates during training) and the Mean Squared Error (MSE) loss function, which is appropriate for regression tasks like rating prediction.
- The training process is initiated using fit(), which feeds the model the training data (user IDs, movie IDs, and corresponding ratings) over 10 epochs. This process involves backpropagation, where the model iteratively adjusts its weights to minimize the difference between predicted and actual ratings. The validation_data argument ensures that the model's performance is evaluated on unseen data during training, preventing overfitting and guiding the learning process.
- The output from the model training process shows how the model's performance evolves over each epoch. Convergence is indicated by the decreasing training loss (MSE) and MAE, meaning the model is learning to reduce prediction errors. The validation loss and MAE are reasonably close to the training loss and MAE, indicating that the model is not overfitting or underfitting – generalizing well.

Training the NCF model allows it to learn the relationship between users and movies based on their interactions. Through multiple iterations, the model adjusts its internal parameters (embeddings and weights) to minimize the error in predicting ratings. In the context of the project, training enables the model to improve its ability to predict how much a user will like a particular movie, based on their past preferences and the characteristics of other movies they have rated, resulting in more accurate and personalized recommendations.

2. Matrix Factorization Model:



```python
#Ensure the embedding range covers all movies/users
num_movies = df['MovieID'].max() + 1
num_users = df['UserID'].max() + 1

#Set the embedding size
embedding_size = 50

#Define input layers for UserID and MovieID
user_input = Input(shape=(1,), name='user_input')
movie_input = Input(shape=(1,), name='movie_input')

#Embedding layers for users and movies
user_embedding = Embedding(input_dim=num_users, output_dim=embedding_size, name='user_embedding')(user_input)
movie_embedding = Embedding(input_dim=num_movies, output_dim=embedding_size, name='movie_embedding')(movie_input)

#Flatten the embeddings to convert them into dense vectors
user_vec = Flatten(name='flatten_user')(user_embedding)
movie_vec = Flatten(name='flatten_movie')(movie_embedding)

#Dot product between the user and movie embeddings (latent factor interaction)
dot_product = Dot(axes=1)([user_vec, movie_vec])

#Bias terms (to capture the average rating for users and movies)
user_bias = Embedding(input_dim=num_users, output_dim=1, name='user_bias')(user_input)
movie_bias = Embedding(input_dim=num_movies, output_dim=1, name='movie_bias')(movie_input)
user_bias = Flatten(name='flatten_user_bias')(user_bias)
movie_bias = Flatten(name='flatten_movie_bias')(movie_bias)

#Add the bias terms to the dot product
prediction = Add()([dot_product, user_bias, movie_bias])

#Build the Matrix Factorization model
matrix_factorization_model = Model([user_input, movie_input], prediction)

#Compile the model
matrix_factorization_model.compile(optimizer='adam', loss='mse', metrics=['mae'])

#Train the Matrix Factorization model
matrix_factorization_history = matrix_factorization_model.fit(
    [X_train['UserID'], X_train['MovieID']],  #Inputs (UserID and MovieID)
    y_train,  #Target (ratings)
    epochs=10,
    batch_size=64,
    validation_data=([X_test['UserID'], X_test['MovieID']], y_test)  #Validation set
)
```

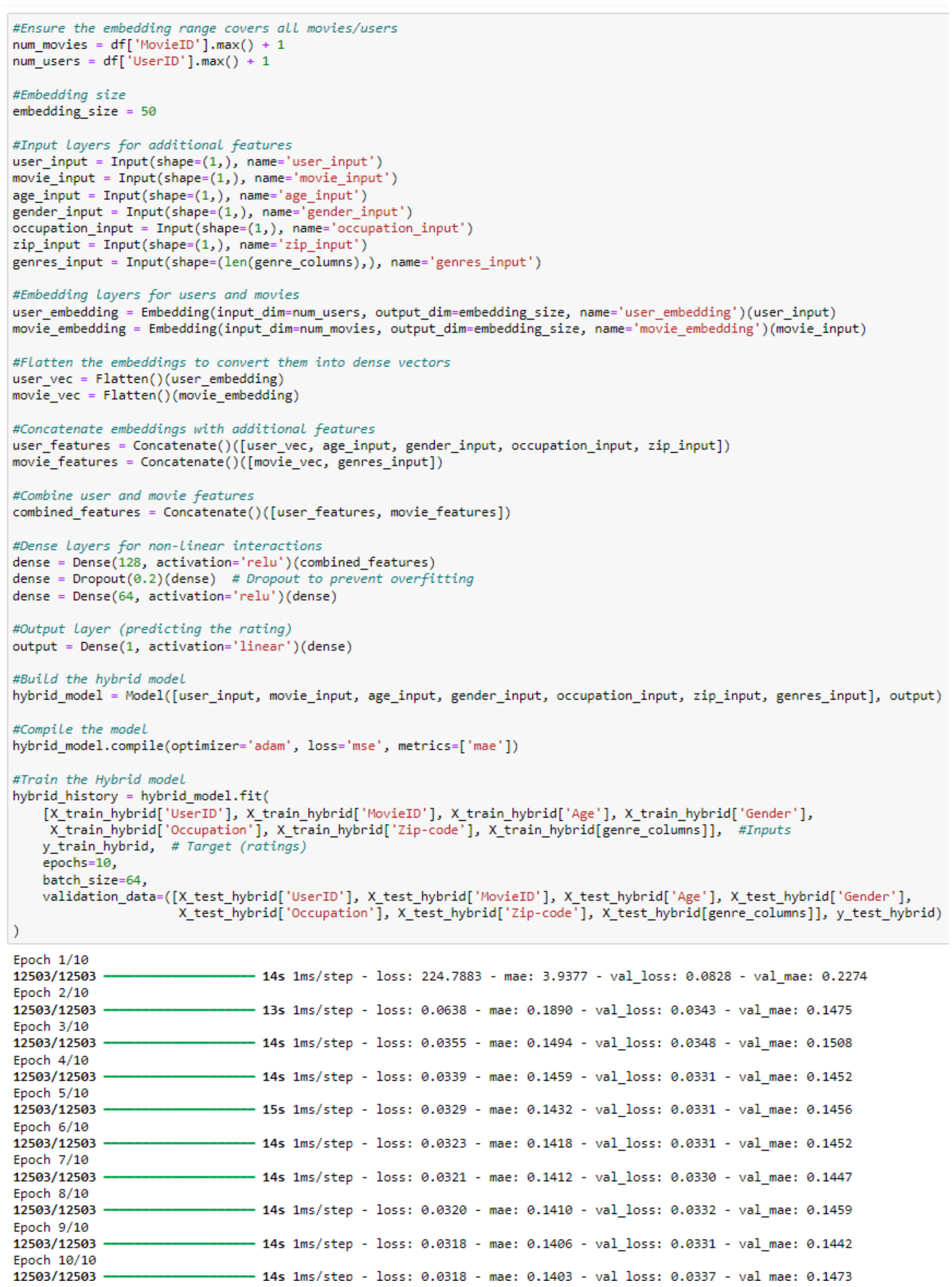
```
Epoch 1/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 12s 886us/step - loss: 0.1558 - mae: 0.3007 - val_loss: 0.0343 - val_mae: 0.1458
Epoch 2/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 11s 883us/step - loss: 0.0291 - mae: 0.1339 - val_loss: 0.0341 - val_mae: 0.1450
Epoch 3/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 11s 890us/step - loss: 0.0234 - mae: 0.1196 - val_loss: 0.0358 - val_mae: 0.1481
Epoch 4/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 11s 877us/step - loss: 0.0208 - mae: 0.1123 - val_loss: 0.0374 - val_mae: 0.1509
Epoch 5/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 11s 868us/step - loss: 0.0195 - mae: 0.1087 - val_loss: 0.0387 - val_mae: 0.1533
Epoch 6/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 11s 867us/step - loss: 0.0187 - mae: 0.1063 - val_loss: 0.0398 - val_mae: 0.1549
Epoch 7/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 11s 866us/step - loss: 0.0183 - mae: 0.1053 - val_loss: 0.0407 - val_mae: 0.1565
Epoch 8/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 11s 884us/step - loss: 0.0180 - mae: 0.1040 - val_loss: 0.0415 - val_mae: 0.1580
Epoch 9/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 11s 885us/step - loss: 0.0177 - mae: 0.1033 - val_loss: 0.0422 - val_mae: 0.1589
Epoch 10/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 11s 886us/step - loss: 0.0175 - mae: 0.1026 - val_loss: 0.0427 - val_mae: 0.1599
```

- The Matrix Factorization model is developed using Keras, which simplifies the process of building neural network models. To start, we determine the number of users and movies in the dataset, which allows us to set up embedding layers for both users and movies. The core idea of Matrix Factorization is to represent users and movies as vectors in a shared latent space, with each vector capturing certain hidden features or latent factors, such as user preferences or movie characteristics.

- Using Keras' Embedding layer, we create dense vector representations for both user IDs and movie IDs, where each ID is mapped to a latent space of a defined dimension (set to 50 in this model). These embeddings, once flattened by Flatten(), are passed through a Dot product, which measures the similarity between the user and movie vectors. This dot product essentially represents the predicted rating, as it captures the relationship between the user and the movie in terms of their latent factors.
- In addition to the dot product, we include bias terms for users and movies. These bias terms adjust the predictions to account for individual tendencies (e.g., some users give higher ratings on average) and general movie popularity. These biases are added to the dot product using Add() to produce the final rating prediction.
- The model is compiled using the Adam optimizer, known for its ability to adjust learning rates during training, and the Mean Squared Error (MSE) loss function, which minimizes the difference between the predicted and actual ratings. MSE is chosen because this is a regression task where we predict continuous ratings.
- Training the model is done through the fit() method, which iterates over the training data (user and movie IDs, and their ratings) across 10 epochs. During this process, the model updates its internal parameters to reduce prediction error. The inclusion of validation data helps monitor how well the model generalizes to new, unseen data during training, ensuring it doesn't overfit.
- The output shows the training progress of the Matrix Factorization model over 10 epochs, tracking metrics like loss (MSE) and MAE for both training and validation data. As the epochs progress, both the loss and MAE decrease, indicating the model is learning and improving prediction accuracy. The validation metrics closely follow the training metrics, showing the model is generalizing well without overfitting. The consistent reduction in errors suggests successful convergence, meaning the model is effectively learning to predict ratings more accurately with each epoch.

Training the Matrix Factorization model enables it to uncover latent patterns in user preferences and movie attributes, allowing it to predict how a user might rate a particular movie. This process helps the model refine its understanding of how users interact with different types of movies, resulting in more accurate and personalized recommendations.

3. Hybrid Model:



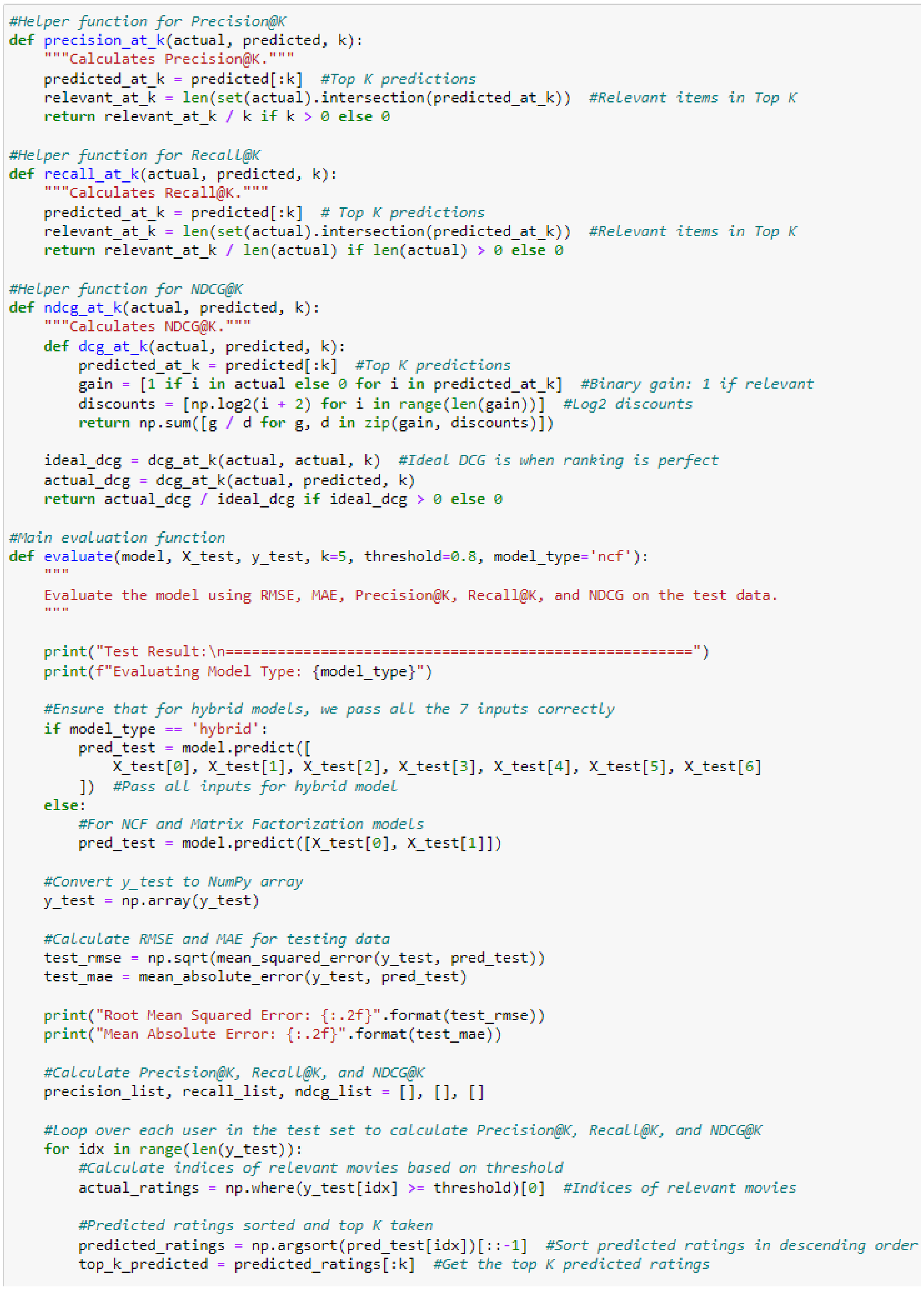
```
#Ensure the embedding range covers all movies/users
num_movies = df['MovieID'].max() + 1
num_users = df['UserID'].max() + 1

#Embedding size
embedding_size = 50

#Input layers for additional features
user_input = Input(shape=(1,), name='user_input')
movie_input = Input(shape=(1,), name='movie_input')
age_input = Input(shape=(1,), name='age_input')
gender_input = Input(shape=(1,), name='gender_input')
occupation_input = Input(shape=(1,), name='occupation_input')
zip_input = Input(shape=(1,), name='zip_input')
genres_input = Input(shape=(len(genre_columns),), name='genres_input')

#Embedding layers for users and movies
user_embedding = Embedding(input_dim=num_users, output_dim=embedding_size, name='user_embedding')(user_input)
movie_embedding = Embedding(input_dim=num_movies, output_dim=embedding_size, name='movie_embedding')(movie_input)

#Flatten the embeddings to convert them into dense vectors
user_vec = Flatten()(user_embedding)
movie_vec = Flatten()(movie_embedding)

#Concatenate embeddings with additional features
user_features = Concatenate()([user_vec, age_input, gender_input, occupation_input, zip_input])
movie_features = Concatenate()([movie_vec, genres_input])

#Combine user and movie features
combined_features = Concatenate()([user_features, movie_features])

#Dense layers for non-linear interactions
dense = Dense(128, activation='relu')(combined_features)
dense = Dropout(0.2)(dense)  # Dropout to prevent overfitting
dense = Dense(64, activation='relu')(dense)

#Output layer (predicting the rating)
output = Dense(1, activation='linear')(dense)

#Build the hybrid model
hybrid_model = Model([user_input, movie_input, age_input, gender_input, occupation_input, zip_input, genres_input], output)

#Compile the model
hybrid_model.compile(optimizer='adam', loss='mse', metrics=['mae'])

#Train the Hybrid model
hybrid_history = hybrid_model.fit(
    [X_train_hybrid['UserID'], X_train_hybrid['MovieID'], X_train_hybrid['Age'], X_train_hybrid['Gender'],
     X_train_hybrid['Occupation'], X_train_hybrid['Zip-code'], X_train_hybrid[genre_columns]],  #Inputs
    y_train_hybrid,  # Target (ratings)
    epochs=10,
    batch_size=64,
    validation_data=([X_test_hybrid['UserID'], X_test_hybrid['MovieID'], X_test_hybrid['Age'], X_test_hybrid['Gender'],
                      X_test_hybrid['Occupation'], X_test_hybrid['Zip-code'], X_test_hybrid[genre_columns]], y_test_hybrid)
)
```

```
Epoch 1/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 14s 1ms/step - loss: 224.7883 - mae: 3.9377 - val_loss: 0.0828 - val_mae: 0.2274
Epoch 2/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 13s 1ms/step - loss: 0.0638 - mae: 0.1890 - val_loss: 0.0343 - val_mae: 0.1475
Epoch 3/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 14s 1ms/step - loss: 0.0355 - mae: 0.1494 - val_loss: 0.0348 - val_mae: 0.1508
Epoch 4/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 14s 1ms/step - loss: 0.0339 - mae: 0.1459 - val_loss: 0.0331 - val_mae: 0.1452
Epoch 5/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 15s 1ms/step - loss: 0.0329 - mae: 0.1432 - val_loss: 0.0331 - val_mae: 0.1456
Epoch 6/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 14s 1ms/step - loss: 0.0323 - mae: 0.1418 - val_loss: 0.0331 - val_mae: 0.1452
Epoch 7/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 14s 1ms/step - loss: 0.0321 - mae: 0.1412 - val_loss: 0.0330 - val_mae: 0.1447
Epoch 8/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 14s 1ms/step - loss: 0.0320 - mae: 0.1410 - val_loss: 0.0332 - val_mae: 0.1459
Epoch 9/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 14s 1ms/step - loss: 0.0318 - mae: 0.1406 - val_loss: 0.0331 - val_mae: 0.1442
Epoch 10/10
12503/12503 ━━━━━━━━━━━━━━━━━━━━ 14s 1ms/step - loss: 0.0318 - mae: 0.1403 - val_loss: 0.0337 - val_mae: 0.1473
```

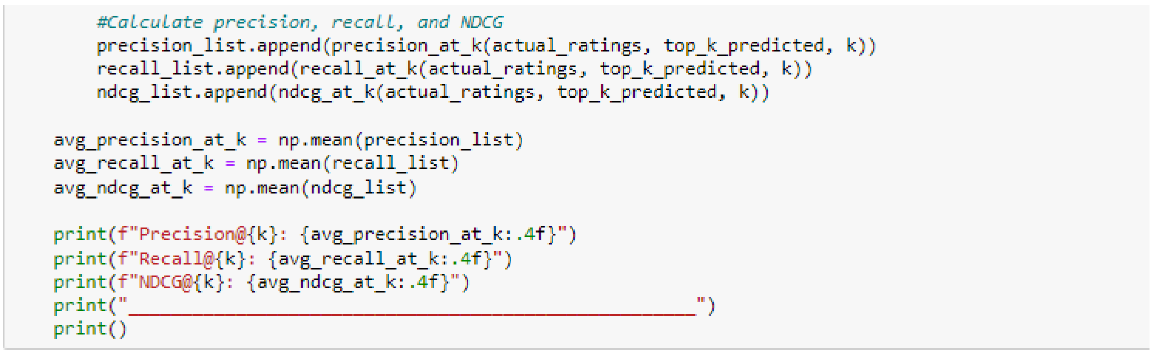
- The Hybrid Recommender System is built using Keras, integrating both collaborative filtering (user-item interactions) and content-based filtering (movie metadata). The process starts by determining the number of unique users and movies, which are used to set the dimensions for the embedding layers, similar to previous models. Keras' Embedding layer is utilized to map user and movie IDs into dense vectors that represent the latent factors for users and movies, capturing their underlying preferences and characteristics.
- Unlike pure collaborative filtering models, this hybrid model incorporates additional user and movie metadata (like age, gender, occupation, zip code, and movie genres). These additional inputs are processed directly without embeddings and concatenated with the user and movie embeddings. This allows the model to learn from both interaction data and content-based information, improving its ability to generalize and make recommendations, especially in cases where collaborative filtering alone would struggle (such as with new users or items).
- After combining the user and movie embeddings with the metadata, the model uses Keras' Concatenate() function to merge these features into a single input vector. This combined vector is then passed through fully connected layers (Dense layers) with ReLU activation functions, which introduce non-linearity and enable the model to learn complex relationships between users, movies, and their respective metadata. A Dropout() layer is used to prevent overfitting by randomly disabling a fraction of neurons during training, ensuring that the model generalizes well to unseen data.
- The output layer, implemented as a Dense() layer with a linear activation function, predicts a continuous value representing the user's rating for the movie. This structure allows the model to output a rating prediction based on the combination of both collaborative and content-based information.
- To train the model, Keras' fit() function is employed, feeding the hybrid model with both the user-item interaction data and the content features (e.g., user demographics and movie genres) over 10 epochs. The Adam optimizer is used for training, which dynamically adjusts learning rates during the process, and the Mean Squared Error (MSE) loss function is chosen since we are predicting continuous values (ratings). During training, the validation data is used to assess how well the model generalizes, helping avoid overfitting by checking performance on unseen examples after each epoch.
- The output shows the hybrid model's performance over 10 epochs. Both training and validation losses (MSE) and MAE decrease steadily, indicating the model is learning and improving its predictions. By the final epoch, the losses have dropped significantly, suggesting the model has converged well. The close match between training and validation loss shows good generalization, meaning the model is performing effectively on both seen and unseen data. Overall, the model is successfully learning from user-item interactions and movie metadata to predict ratings accurately.

Training the Hybrid model enables it to learn from both collaborative and content-based information, enhancing its ability to provide diverse, accurate, and personalized recommendations. By leveraging both the interaction data and metadata, the hybrid approach mitigates the limitations of pure collaborative filtering (such as the cold-start problem) and can make more robust predictions in real-world scenarios.

Before moving onto the results and performance analysis phase and employing our models for testing, we have defined multiple helper functions along with a primary evaluation function to evaluate, record, and print out the performance metrics for our recommendation system models. These functions focus on assessing the effectiveness of our model's predictions using both traditional error metrics and ranking-based metrics for top-K recommendations:



```python
#Helper function for Precision@K
def precision_at_k(actual, predicted, k):
    """Calculates Precision@K."""
    predicted_at_k = predicted[:k]  #Top K predictions
    relevant_at_k = len(set(actual).intersection(predicted_at_k))  #Relevant items in Top K
    return relevant_at_k / k if k > 0 else 0

#Helper function for Recall@K
def recall_at_k(actual, predicted, k):
    """Calculates Recall@K."""
    predicted_at_k = predicted[:k]  # Top K predictions
    relevant_at_k = len(set(actual).intersection(predicted_at_k))  #Relevant items in Top K
    return relevant_at_k / len(actual) if len(actual) > 0 else 0

#Helper function for NDCG@K
def ndcg_at_k(actual, predicted, k):
    """Calculates NDCG@K."""
    def dcg_at_k(actual, predicted, k):
        predicted_at_k = predicted[:k]  #Top K predictions
        gain = [1 if i in actual else 0 for i in predicted_at_k]  #Binary gain: 1 if relevant
        discounts = [np.log2(i + 2) for i in range(len(gain))]  #Log2 discounts
        return np.sum([g / d for g, d in zip(gain, discounts)])

    ideal_dcg = dcg_at_k(actual, actual, k)  #Ideal DCG is when ranking is perfect
    actual_dcg = dcg_at_k(actual, predicted, k)
    return actual_dcg / ideal_dcg if ideal_dcg > 0 else 0

#Main evaluation function
def evaluate(model, X_test, y_test, k=5, threshold=0.8, model_type='ncf'):
    """
    Evaluate the model using RMSE, MAE, Precision@K, Recall@K, and NDCG on the test data.
    """

    print("Test Result:\n=======================================================")
    print(f"Evaluating Model Type: {model_type}")

    #Ensure that for hybrid models, we pass all the 7 inputs correctly
    if model_type == 'hybrid':
        pred_test = model.predict([
            X_test[0], X_test[1], X_test[2], X_test[3], X_test[4], X_test[5], X_test[6]
        ])  #Pass all inputs for hybrid model
    else:
        #For NCF and Matrix Factorization models
        pred_test = model.predict([X_test[0], X_test[1]])

    #Convert y_test to NumPy array
    y_test = np.array(y_test)

    #Calculate RMSE and MAE for testing data
    test_rmse = np.sqrt(mean_squared_error(y_test, pred_test))
    test_mae = mean_absolute_error(y_test, pred_test)

    print("Root Mean Squared Error: {:.2f}".format(test_rmse))
    print("Mean Absolute Error: {:.2f}".format(test_mae))

    #Calculate Precision@K, Recall@K, and NDCG@K
    precision_list, recall_list, ndcg_list = [], [], []

    #Loop over each user in the test set to calculate Precision@K, Recall@K, and NDCG@K
    for idx in range(len(y_test)):
        #Calculate indices of relevant movies based on threshold
        actual_ratings = np.where(y_test[idx] >= threshold)[0]  #Indices of relevant movies

        #Predicted ratings sorted and top K taken
        predicted_ratings = np.argsort(pred_test[idx])[::-1]  #Sort predicted ratings in descending order
        top_k_predicted = predicted_ratings[:k]  #Get the top K predicted ratings
```



```
        #Calculate precision, recall, and NDCG
        precision_list.append(precision_at_k(actual_ratings, top_k_predicted, k))
        recall_list.append(recall_at_k(actual_ratings, top_k_predicted, k))
        ndcg_list.append(ndcg_at_k(actual_ratings, top_k_predicted, k))

    avg_precision_at_k = np.mean(precision_list)
    avg_recall_at_k = np.mean(recall_list)
    avg_ndcg_at_k = np.mean(ndcg_list)

    print(f"Precision@{k}: {avg_precision_at_k:.4f}")
    print(f"Recall@{k}: {avg_recall_at_k:.4f}")
    print(f"NDCG@{k}: {avg_ndcg_at_k:.4f}")
    print("_____________________________________________________")
    print()
```

The primary function, evaluate(), calculates and prints key performance metrics (RMSE, MAE, Precision@K, Recall@K, and NDCG@K) for a recommender system model on the test data. Here's how it works:

- The function takes the trained model and test data (user IDs, movie IDs, and any additional features, depending on the model type) as inputs.
- If the model is a hybrid model, it passes all the necessary inputs (user and movie features) to the model for prediction; otherwise, it passes only the user and movie IDs for collaborative filtering models like NCF or matrix factorization.
- Once the predictions are generated, the function computes regression metrics such as RMSE and MAE to evaluate how accurately the model is predicting the target ratings.
- It then calculates ranking-based metrics like Precision@K, Recall@K, and NDCG@K by comparing the predicted top-K recommendations to the actual relevant items in the test data, based on a predefined rating threshold.
- Finally, the function prints out the calculated metrics for the test data, providing a clear snapshot of the model's performance.
- Additionally, the helper functions precision_at_k(), recall_at_k(), and ndcg_at_k() are used to compute the ranking metrics by taking the top-K predicted movies for each user and comparing them with the actual relevant movies. These functions measure how well the model ranks the most relevant items for each user, which is crucial in evaluating recommendation systems.

Ultimately, This code is useful for evaluating recommendation systems where you want to measure not just how well the model predicts ratings (RMSE/MAE), but also how well it ranks recommendations (Precision@K, Recall@K, NDCG@K). These ranking metrics are especially important in real-world recommendation tasks, where the order of the top recommendations matters for user satisfaction.

Essentially, this setup provides a comprehensive evaluation of the model on the test data, including both traditional error measures and ranking-based metrics, which are essential for assessing the quality of the recommendations generated by the system.

Results & Performance Analysis

Before moving ahead with evaluating our models, it's important to emphasize the significance of the testing scores and the evaluation metrics we will be using to measure the performance of our recommender system models. The metrics on the test set represent how well the models predict unseen data, simulating real-world scenarios where the model will recommend movies to users it hasn't seen before. Hence, more emphasis is placed on the testing set metrics as they reflect real-world performance; for this reasoning, we are solely focusing on test results.

Evaluation Metrics:

- **Root Mean Squared Error (RMSE)**: RMSE is a common metric used to evaluate the accuracy of regression models, including recommender systems. It represents the average magnitude of errors between the predicted and true values of the target (ratings). Specifically, RMSE is calculated as the square root of the average squared differences between predicted and actual ratings. RMSE penalizes larger errors more than smaller ones due to the squaring process. In this project, RMSE measures how closely the predicted movie ratings match the actual user ratings. A lower RMSE indicates better model performance, with zero being the ideal score, indicating perfect predictions.

- **Mean Absolute Error (MAE)**: Similar to RMSE, MAE evaluates the average magnitude of errors between predicted and true ratings, but unlike RMSE, it does not square the differences. Instead, it computes the absolute differences, meaning that all errors, whether large or small, are treated equally. MAE provides a more straightforward interpretation of the average error, as it is expressed in the same unit as the target (ratings). In this project, MAE reflects the average absolute difference between the predicted ratings and the actual ratings, with a lower MAE being more desirable.

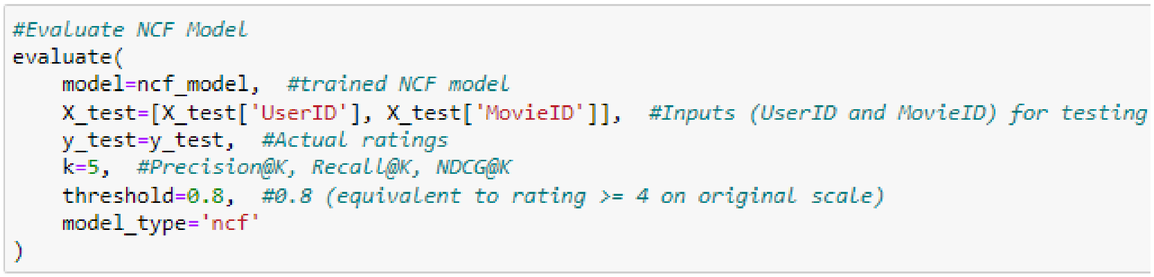
- **Precision@K**: Precision@K is a ranking-based metric that measures the proportion of relevant items (movies) in the top-K recommendations made by the model. In the context of this project, it shows how many of the top-K movies recommended by the system were actually relevant to the user based on their past interactions (i.e., rated highly). A higher Precision@K indicates that the model is better at recommending relevant movies.

- **Recall@K**: Recall@K is another ranking-based metric that measures how many of the relevant items (movies) were included in the top-K recommendations. It essentially answers the question: out of all the movies that are relevant to the user, how many did the system manage to recommend in its top-K list? A higher Recall@K indicates that the model is capturing more of the user's relevant preferences in its top recommendations.

- **NDCG@K**: NDCG@K evaluates not only the relevance of the recommended items but also the order or ranking of those items. It assigns higher importance to relevant items appearing higher in the recommendation list. In this project, NDCG@K measures how well the model ranks the relevant movies among the top-K recommendations. A higher NDCG@K score suggests that the system is not only recommending relevant movies but also ranking them well in terms of importance to the user.

In summary, RMSE and MAE are metrics that measure the accuracy of the predicted ratings, whereas Precision@K, Recall@K, and NDCG@K focus on evaluating the quality of the ranked recommendations. These metrics together provide a comprehensive view of the recommender systems performance, balancing both prediction accuracy and recommendation relevance.

Now that we have emphasized the importance of scoring on testing sets, as well as covering the evaluation metrics of our models thoroughly, we can go ahead and start with the results and performance analysis/modelling phase.



```
#Evaluate NCF Model
evaluate(
    model=ncf_model,  #trained NCF model
    X_test=[X_test['UserID'], X_test['MovieID']],  #Inputs (UserID and MovieID) for testing
    y_test=y_test,  #Actual ratings
    k=5,  #Precision@K, Recall@K, NDCG@K
    threshold=0.8,  #0.8 (equivalent to rating >= 4 on original scale)
    model_type='ncf'
)
```

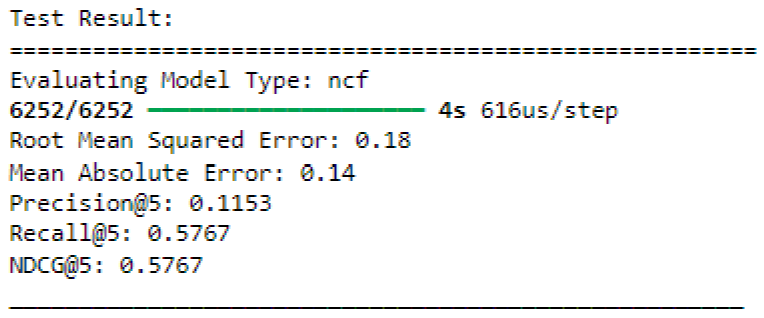
Before discussing the test results, it's important to explain the structure of the evaluation code used for all models. The evaluate() function is applied consistently to each model (NCF, Matrix Factorization, Hybrid) –with minor adjustments depending on the specific model being tested. This ensures a uniform approach for comparing performance across all models, where the same key parameters and metrics are applied to measure their performance. The parameters k=5 and threshold=0.8 are particularly important as they act as filters in generating top recommendations and relevant predictions and will be consistent throughout the evaluation.

- k=5: This represents the top 5 recommendations that the model makes. When evaluating metrics like Precision@5 and Recall@5, we focus on how well the model identifies relevant items (e.g., movies the user would like) within the top 5 predictions. This simulates a real-world recommendation scenario, where users often only see a few top recommendations.
- Threshold=0.8: The threshold sets a cut-off point for what is considered a relevant prediction. In our case, a predicted rating of 0.8 or above (equivalent to a rating of 4 or higher on the original 1-5 scale) is considered a positive or relevant recommendation. This means the model is evaluated based on how well it predicts ratings on movies that users would actually enjoy (4+ ratings), filtering out less relevant or low-rated recommendations.

This structure ensures that all models are evaluated on the same basis, allowing for consistent comparison of their performance. Now that we have expanded on the evaluation code used by all models, were only going to discuss the results moving forward.

Results

1. **Collaborative Filtering w/ Deep Learning (NCF) Model:**



```
Test Result:
======================================================
Evaluating Model Type: ncf
6252/6252 ━━━━━━━━━━━━━━━━━━━━ 4s 616us/step
Root Mean Squared Error: 0.18
Mean Absolute Error: 0.14
Precision@5: 0.1153
Recall@5: 0.5767
NDCG@5: 0.5767
______________________________________________________
```

Rating Metrics:

- **Root Mean Squared Error (RMSE): 0.18**
  - The RMSE of 0.18 reflects the average error between the predicted and actual user ratings on movies. This low value indicates that the model makes very precise predictions with minimal large errors. If we refer back to the original 1–

5-star rating system, this 0.18 would be less than a fraction of a star off (around 1/5 of a star) – a very accurate result.

- **Mean Absolute Error (MAE): 0.14**
    - With an MAE of 0.14, the model's average prediction deviates by only 0.14 from the actual ratings. This can be interpreted as the model predicting user preferences with approximately 86% accuracy (1 - 0.14 = 0.86), which is excellent. In simpler terms, the model's predictions on movie ratings deviate from actual user ratings by just over 1/10th of a star, making its predictions highly accurate; excellent.

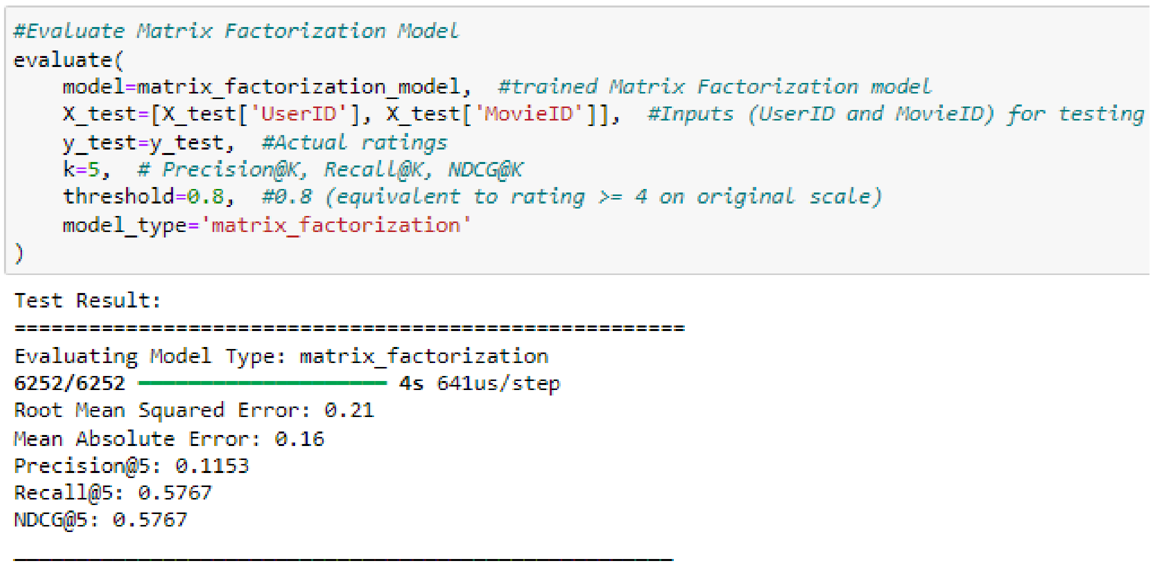
Ranking Metrics:

- **Precision@5: 0.1153**
    - A precision score of 11.53% indicates that, on average, only 11.53% of the top 5 movie recommendations made by the model are truly relevant or highly rated by the user. To put this in context, if a user looks at the top 5 suggestions, only 1 out of the 5 might be something they'd actually rate highly – meaning the model's filtering of top recommendations isn't great and could use improvement in avoiding irrelevant suggestions.

- **Recall@5: 0.5767**
    - The recall score of 57.67% suggests that out of all the highly rated (relevant) movies, the model successfully surfaces 57.67% of them in its top 5 recommendations. This means that while the model does surface a majority of relevant content, it still misses around 42.33% of relevant movies. In other words, a user could expect about 3 out of 5 recommendations to be relevant, which is a decent but not perfect result.

- **NDCG@5: 0.5767**
    - The NDCG score of 0.5767 reflects how well the model ranks relevant movies within the top 5 recommendations. This score can also be understood as the model ranking 57.67% of relevant items in favorable positions within the top 5. Essentially, it means that while about 3 out of 5 relevant movies are in the top slots, the placement of those relevant movies isn't always ideal. For instance, a relevant movie might appear in the 3rd or 4th position instead of the 1st, showing that while the model's ranking is fair, it still has room for improvement in consistently placing the most relevant movies at the top.

2. **Matrix Factorization Model (Traditional Approach):**



```
#Evaluate Matrix Factorization Model
evaluate(
    model=matrix_factorization_model,  #trained Matrix Factorization model
    X_test=[X_test['UserID'], X_test['MovieID']],  #Inputs (UserID and MovieID) for testing
    y_test=y_test,  #Actual ratings
    k=5,  # Precision@K, Recall@K, NDCG@K
    threshold=0.8,  #0.8 (equivalent to rating >= 4 on original scale)
    model_type='matrix_factorization'
)
```

```
Test Result:
======================================================
Evaluating Model Type: matrix_factorization
6252/6252 ━━━━━━━━━━━━━━━━━━━━ 4s 641us/step
Root Mean Squared Error: 0.21
Mean Absolute Error: 0.16
Precision@5: 0.1153
Recall@5: 0.5767
NDCG@5: 0.5767
______________________________________________________
```

Rating Metrics:

- **Root Mean Squared Error (RMSE): 0.21**
  - With an RMSE of 0.21, this metric highlights the average difference between predicted and actual user ratings for movies. While this result is slightly higher than the NCF model, it still indicates that the model's predictions are quite accurate. On the original 1-5 rating scale, the predictions are off by only about a fifth of a star, which is still a very commendable outcome but not as precise as the NCF model.

- **Mean Absolute Error (MAE): 0.16**
  - The model's MAE of 0.16 means its predictions are, on average, off by 0.16 from the actual ratings. This translates to approximately 84% accuracy (1 - 0.16 = 0.84), slightly lower than NCF's accuracy of 86%. It suggests the predictions are still really good, with deviations averaging around one-sixth of a star from the true ratings – overall, a strong performance.

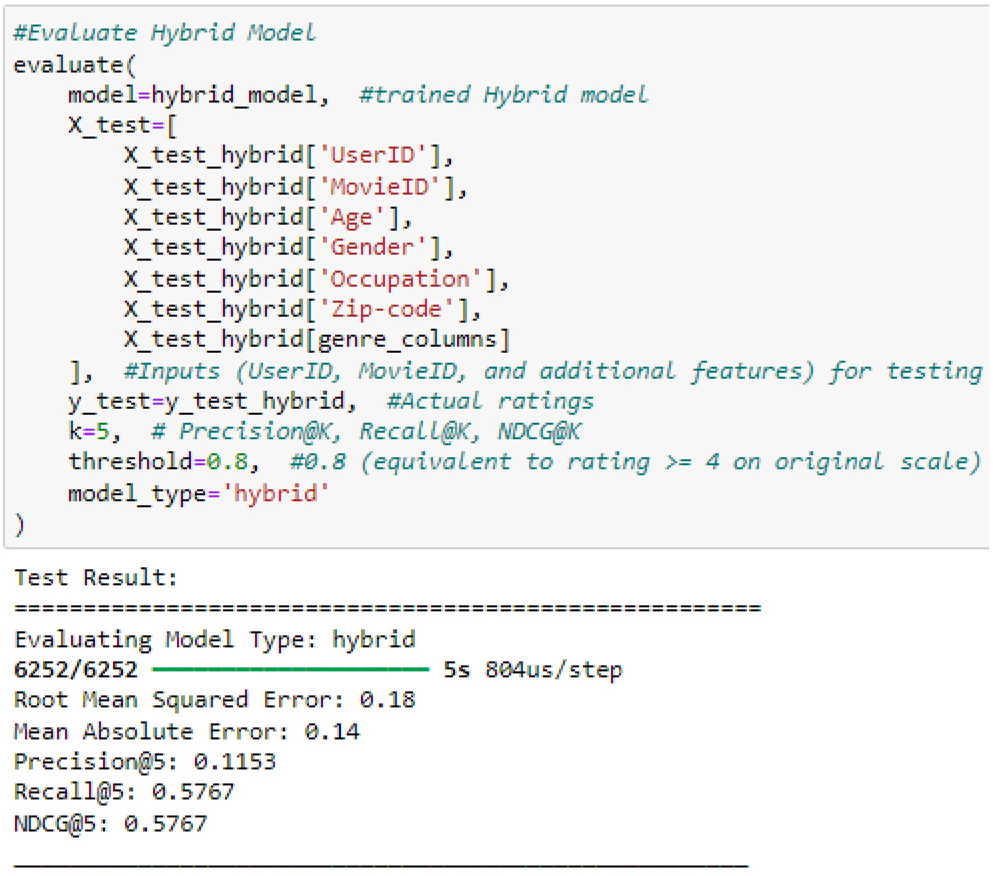
Ranking Metrics:

- **Precision@5: 0.1153**
  - As with the NCF model, the Precision@5 score of 11.53% shows that only 11.53% of the top 5 recommendations made by the model are relevant to the user. This indicates that while the model predicts ratings well, it struggles in ensuring that its top 5 suggestions are highly relevant—similar to the NCF, this isn't ideal and highlights the need for better recommendation filtering.

- **Recall@5: 0.5767**
    - The recall score of 57.67% indicates that the model is able to capture 57.67% of all relevant movies within its top 5 recommendations. This result is the same as the NCF model, suggesting that both models are fairly effective at surfacing relevant content, but there is still room to improve the capture of highly rated movies in the top recommendations—decent but not outstanding.

- **NDCG@5: 0.5767**
    - The NDCG score of 0.5767, like in the NCF model, indicates that 57.67% of relevant movies are ranked in favourable positions within the top 5. While the model does well at ensuring relevant movies are ranked higher, there is still potential for improvement in placing the most relevant content at the very top – this is a fair outcome.

3. **Hybrid Recommender System Model (Collaborative + Content-Based Filtering):**



```
#Evaluate Hybrid Model
evaluate(
    model=hybrid_model,  #trained Hybrid model
    X_test=[
        X_test_hybrid['UserID'],
        X_test_hybrid['MovieID'],
        X_test_hybrid['Age'],
        X_test_hybrid['Gender'],
        X_test_hybrid['Occupation'],
        X_test_hybrid['Zip-code'],
        X_test_hybrid[genre_columns]
    ],  #Inputs (UserID, MovieID, and additional features) for testing
    y_test=y_test_hybrid,  #Actual ratings
    k=5,  # Precision@K, Recall@K, NDCG@K
    threshold=0.8,  #0.8 (equivalent to rating >= 4 on original scale)
    model_type='hybrid'
)
```

```
Test Result:
======================================================
Evaluating Model Type: hybrid
6252/6252 ━━━━━━━━━━━━━━━━━━━━ 5s 804us/step
Root Mean Squared Error: 0.18
Mean Absolute Error: 0.14
Precision@5: 0.1153
Recall@5: 0.5767
NDCG@5: 0.5767
______________________________________________________
```

Rating Metrics:

- **Root Mean Squared Error (RMSE): 0.18**
    - The RMSE of 0.18 demonstrates that the Hybrid model's predictions have a low average error compared to the actual user ratings. This performance matches the NCF model, with only a small fraction of error in predicted ratings, translating to less than a fifth of a star difference on the original 1-5 scale – indicating very accurate predictions.

- **Mean Absolute Error (MAE): 0.14**
    - With an MAE of 0.14, the Hybrid model's predictions deviate, on average, by 0.14 from the actual ratings. Like the NCF model, this shows approximately 86% accuracy (1 - 0.14 = 0.86), which is an excellent result. This highlights the model's strength in providing accurate predictions, deviating only by around a fraction of a star – similar to the NCF, this is a very strong outcome.

Ranking Metrics:

- **Precision@5: 0.1153**
    - The Precision@5 score of 11.53% means that, as with the NCF and Matrix Factorization models, only about 11.53% of the top 5 recommendations made by the Hybrid model are relevant to the user. This indicates that, while the model is excellent at predicting ratings, it still struggles with ensuring that the top 5 recommendations are highly relevant – leaving room for improvement in filtering irrelevant content.

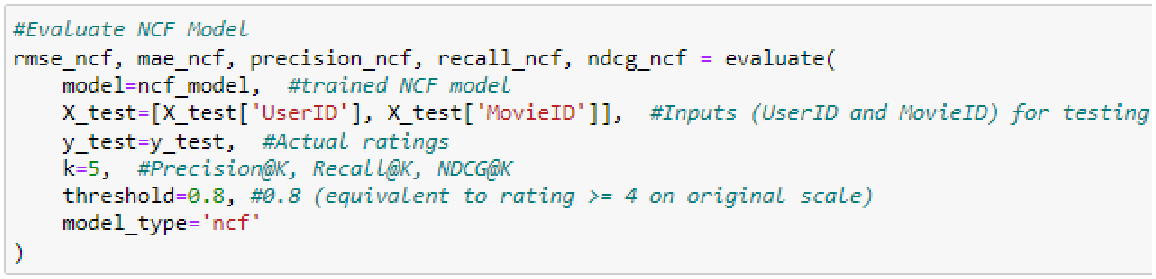
- **Recall@5: 0.5767**
    - The recall score of 57.67% indicates that the model is able to capture 57.67% of all relevant (highly rated) movies in the top 5 recommendations. This is on par with the other models, showing that the Hybrid model is fairly strong at surfacing relevant content, though there is still potential for better coverage of relevant movies – decent but with room for improvement.

- **NDCG@5: 0.5767**
    - The NDCG score of 0.5767, as seen in the other models, suggests that 57.67% of relevant movies are ranked favourably within the top 5. This score shows that while the Hybrid model does well in ranking relevant content, there is still room to improve in pushing the most relevant movies to the top of the recommendation list—overall, a fair result.

To summarize the modelling results before hyperparameter tuning, I slightly edit the evaluation function:

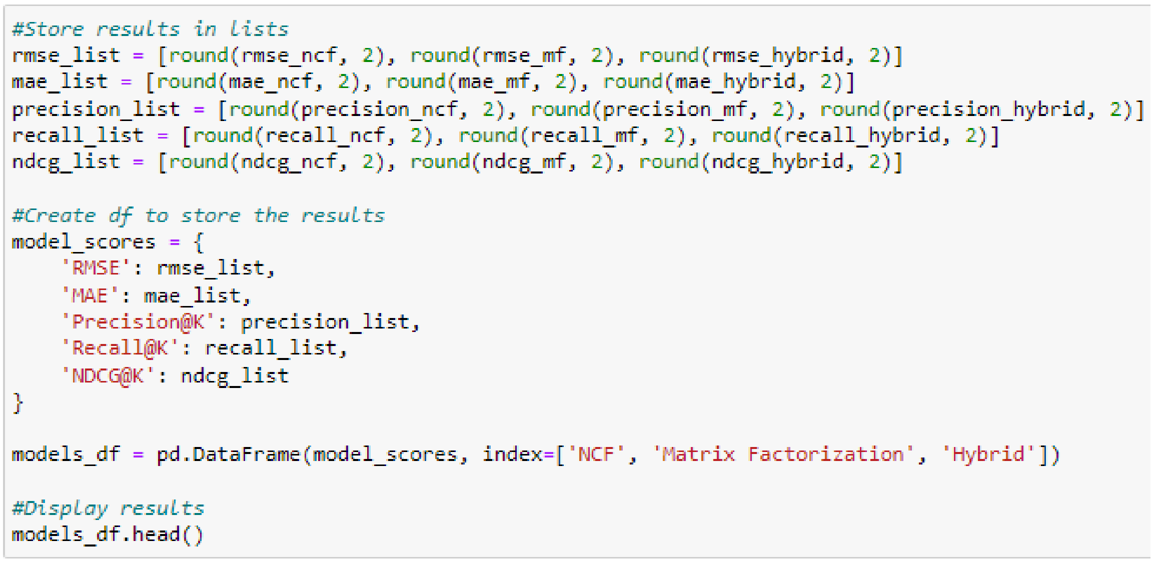
```
#Return metrics for further use
return test_rmse, test_mae, avg_precision_at_k, avg_recall_at_k, avg_ndcg_at_k
```

- I add a return statement at the end of the evaluate() function to return the calculated metrics (RMSE, MAE, Precision@K, Recall@K, and NDCG@K). This allows the metrics to be captured and used later instead of just being printed out.



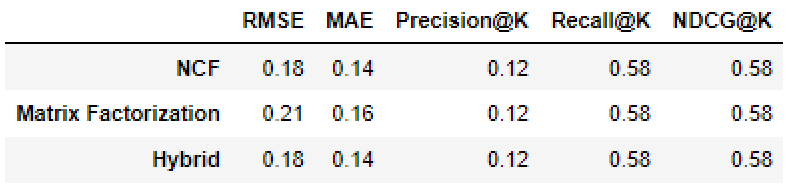
```
#Evaluate NCF Model
rmse_ncf, mae_ncf, precision_ncf, recall_ncf, ndcg_ncf = evaluate(
    model=ncf_model,  #trained NCF model
    X_test=[X_test['UserID'], X_test['MovieID']],  #Inputs (UserID and MovieID) for testing
    y_test=y_test,  #Actual ratings
    k=5,  #Precision@K, Recall@K, NDCG@K
    threshold=0.8, #0.8 (equivalent to rating >= 4 on original scale)
    model_type='ncf'
)
```

- I slightly modify the call to the evaluate() function to capture the returned metric values (such as rmse_ncf, mae_ncf, etc.) for the NCF model. The same procedure is applied to the other models (Matrix Factorization, Hybrid), so there is no need to repeat the explanation for each one.



```
#Store results in lists
rmse_list = [round(rmse_ncf, 2), round(rmse_mf, 2), round(rmse_hybrid, 2)]
mae_list = [round(mae_ncf, 2), round(mae_mf, 2), round(mae_hybrid, 2)]
precision_list = [round(precision_ncf, 2), round(precision_mf, 2), round(precision_hybrid, 2)]
recall_list = [round(recall_ncf, 2), round(recall_mf, 2), round(recall_hybrid, 2)]
ndcg_list = [round(ndcg_ncf, 2), round(ndcg_mf, 2), round(ndcg_hybrid, 2)]

#Create df to store the results
model_scores = {
    'RMSE': rmse_list,
    'MAE': mae_list,
    'Precision@K': precision_list,
    'Recall@K': recall_list,
    'NDCG@K': ndcg_list
}

models_df = pd.DataFrame(model_scores, index=['NCF', 'Matrix Factorization', 'Hybrid'])

#Display results
models_df.head()
```

- After obtaining the results for each model, I store the metrics in lists, rounding the values to 2 decimal places for consistency. Using these lists, I create a dictionary (model_scores) that maps each metric (RMSE, MAE, Precision@K, Recall@K, NDCG@K) to its corresponding values for each model. This dictionary is then converted into a Pandas data frame (models_df). Finally, I display the results using the head() method, which provides a concise table of the evaluation metrics for all models.



	RMSE	MAE	Precision@K	Recall@K	NDCG@K
NCF	0.18	0.14	0.12	0.58	0.58
Matrix Factorization	0.21	0.16	0.12	0.58	0.58
Hybrid	0.18	0.14	0.12	0.58	0.58

- By observing the table above, we can summarize the performance of all three models before hyperparameter tuning. In this case, the Neural Collaborative Filtering (NCF) and Hybrid models are performing almost identically, both showing better results than the Matrix Factorization model in terms of RMSE and MAE. The NCF and Hybrid models achieve a Root Mean Squared Error (RMSE) of 0.18 and a Mean Absolute Error (MAE) of 0.14, indicating they make more accurate predictions compared to the Matrix Factorization model, which has an RMSE of 0.21 and an MAE of 0.16.
- Despite these differences in predictive accuracy, all models; NCF, Matrix Factorization, and Hybrid – yield identical scores for the ranking metrics, with a Precision@K of 0.12, Recall@K of 0.58, and NDCG@K of 0.58. This suggests that, while the NCF and Hybrid models make better rating predictions, their ability to rank and recommend top movies remains consistent with the traditional Matrix Factorization model – surprising but may change during hyperparameter tuning.
- Further improvements can likely be achieved through hyperparameter tuning, but the current state suggests that the NCF and Hybrid models provide better prediction accuracy compared to the traditional Matrix Factorization approach, though all models still need improvement in their recommendation quality, as reflected by the ranking metrics. In contrast, the predictive quality is excellent, as highlighted by the rating metrics.

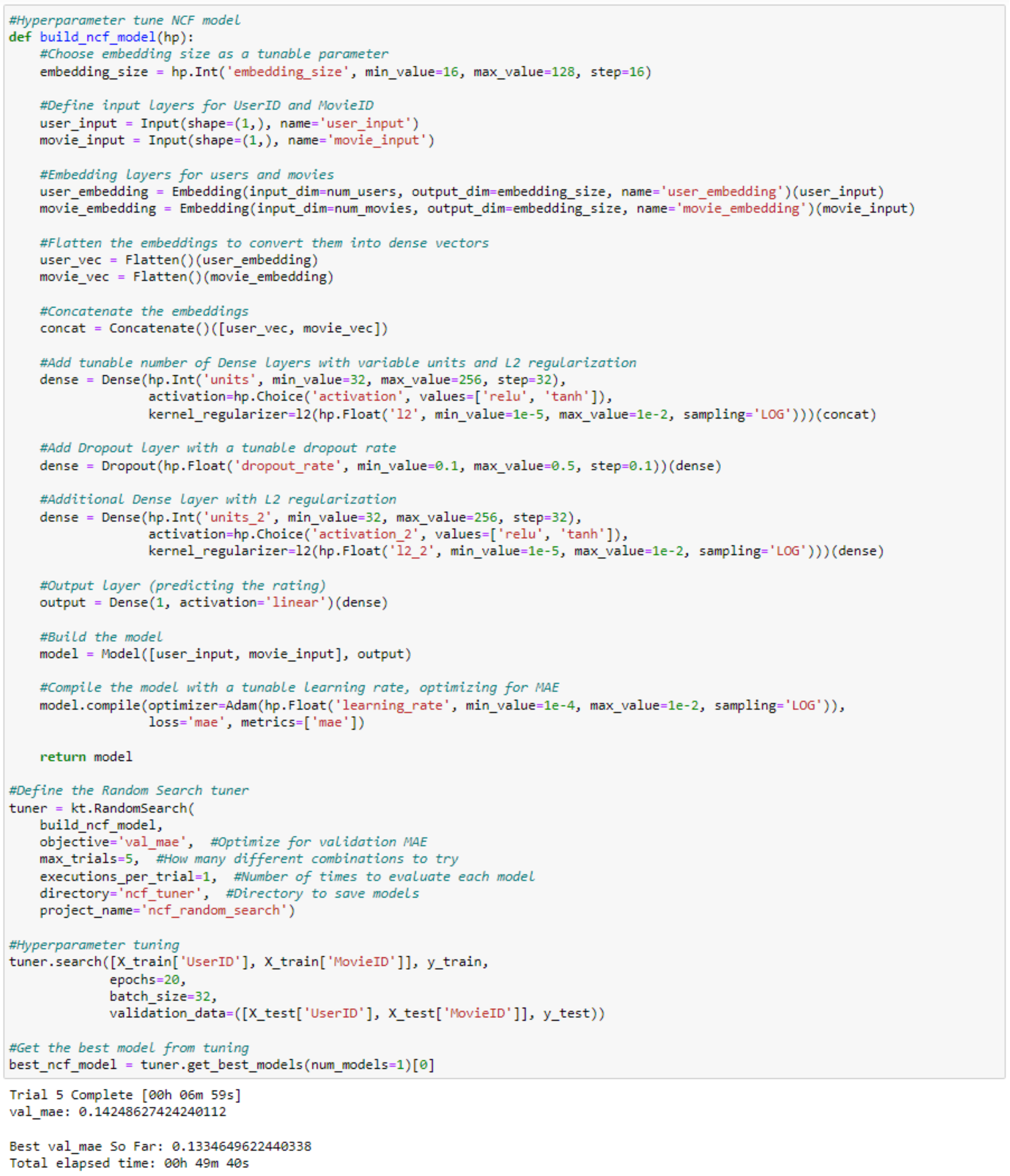
Hyperparameter Tuning

We will be conducting hyperparameter tuning on our models using RandomSearch from the Keras Tuner package. RandomSearch samples a fixed number of random combinations from the hyperparameter space, making it more efficient than an exhaustive search. This allows us to explore a broad range of configurations while keeping the process faster and more resource-efficient. In our project, RandomSearch evaluates each sampled combination of hyperparameters and selects the one that yields the best performance based on MAE (The best performing metric across all models). By automating the discovery of the optimal model configuration, RandomSearch eliminates the need for manual experimentation, streamlining the tuning process.

We tune critical hyperparameters such as embedding size, dense layer units, dropout rates, learning rate, and batch size across all our models (NCF, Matrix Factorization, and Hybrid). This approach ensures that we maximize our models performance while keeping the tuning time manageable, ultimately improving the movie rating predictive accuracy and movie recommendation quality.

In order to perform RandomSearch hyperparameter tuning we had to rebuild and retrain our models with the newly tuned hyperparameters. This allowed the models to search across a wider parameter space and find the optimal configurations to maximize results; specifically, aiming to maximize the Mean Absolute Error (MAE) across all our models as it is the most important metric in context of our project.

1. **Tuned NCF Model:**



```
#Hyperparameter tune NCF model
def build_ncf_model(hp):
    #Choose embedding size as a tunable parameter
    embedding_size = hp.Int('embedding_size', min_value=16, max_value=128, step=16)

    #Define input layers for UserID and MovieID
    user_input = Input(shape=(1,), name='user_input')
    movie_input = Input(shape=(1,), name='movie_input')

    #Embedding layers for users and movies
    user_embedding = Embedding(input_dim=num_users, output_dim=embedding_size, name='user_embedding')(user_input)
    movie_embedding = Embedding(input_dim=num_movies, output_dim=embedding_size, name='movie_embedding')(movie_input)

    #Flatten the embeddings to convert them into dense vectors
    user_vec = Flatten()(user_embedding)
    movie_vec = Flatten()(movie_embedding)

    #Concatenate the embeddings
    concat = Concatenate()([user_vec, movie_vec])

    #Add tunable number of Dense layers with variable units and L2 regularization
    dense = Dense(hp.Int('units', min_value=32, max_value=256, step=32),
                  activation=hp.Choice('activation', values=['relu', 'tanh']),
                  kernel_regularizer=l2(hp.Float('l2', min_value=1e-5, max_value=1e-2, sampling='LOG')))(concat)

    #Add Dropout layer with a tunable dropout rate
    dense = Dropout(hp.Float('dropout_rate', min_value=0.1, max_value=0.5, step=0.1))(dense)

    #Additional Dense layer with L2 regularization
    dense = Dense(hp.Int('units_2', min_value=32, max_value=256, step=32),
                  activation=hp.Choice('activation_2', values=['relu', 'tanh']),
                  kernel_regularizer=l2(hp.Float('l2_2', min_value=1e-5, max_value=1e-2, sampling='LOG')))(dense)

    #Output layer (predicting the rating)
    output = Dense(1, activation='linear')(dense)

    #Build the model
    model = Model([user_input, movie_input], output)

    #Compile the model with a tunable learning rate, optimizing for MAE
    model.compile(optimizer=Adam(hp.Float('learning_rate', min_value=1e-4, max_value=1e-2, sampling='LOG')),
                  loss='mae', metrics=['mae'])

    return model

#Define the Random Search tuner
tuner = kt.RandomSearch(
    build_ncf_model,
    objective='val_mae',  #Optimize for validation MAE
    max_trials=5,  #How many different combinations to try
    executions_per_trial=1,  #Number of times to evaluate each model
    directory='ncf_tuner',  #Directory to save models
    project_name='ncf_random_search')

#Hyperparameter tuning
tuner.search([X_train['UserID'], X_train['MovieID']], y_train,
             epochs=20,
             batch_size=32,
             validation_data=([X_test['UserID'], X_test['MovieID']], y_test))

#Get the best model from tuning
best_ncf_model = tuner.get_best_models(num_models=1)[0]
```

```
Trial 5 Complete [00h 06m 59s]
val_mae: 0.14248627424240112

Best val_mae So Far: 0.1334649622440338
Total elapsed time: 00h 49m 40s
```

Key Changes in the Tuned NCF Model:

Embedding Size:

- The embedding size is now tunable, ranging from 16 to 128, allowing the model to automatically select the most optimal size for representing user-movie interactions. Previously, this was fixed at 50.

Dense Layers:

- The number of units in the dense layers has been made tunable, ranging from 32 to 256, which allows for more flexibility in model complexity. Additionally, the activation function is now tunable between 'relu' and 'tanh', which helps the model capture non-linear relationships better. In the original model, these were fixed at 128 and 64 units with 'relu' activation.

L2 Regularization:

- L2 regularization was introduced and made tunable between 1e-5 and 1e-2 to prevent overfitting. It helps control the model's complexity, ensuring it generalizes well to unseen data by penalizing large weights. This was not present in the original model.

Dropout Rate:

- The dropout rate, which was fixed at 0.2 in the original model, is now tunable between 0.1 and 0.5. This enhances regularization by randomly disabling neurons during training to prevent overfitting and help the model generalize better.

Learning Rate:

- The learning rate is tunable within a range of 1e-4 to 1e-2. Tuning this parameter helps optimize how fast the model updates weights, striking a balance between slow and fast learning to minimize errors.

Batch Size:

- The batch size was fixed at 32 in the tuning process (it was originally set at 64). Tuning batch size allows the model to balance the trade-off between training speed and stability during weight updates.

Fixed Epochs:

- The number of epochs was set to 20, the original model was 10. This change ensures the model has enough time to learn from the data while maintaining reasonable training time.

Output:

- The model achieved a validation MAE of 0.133, which is the lowest error found during the tuning process, reflecting the best performance from the hyperparameter search. The tuning process took approximately 49 minutes, showing that 5 trials were completed to find the best combination of hyperparameters. The model was evaluated on 5 different combinations of parameters, and the best model, with the lowest validation MAE, was selected for further use.

2. Tuned Matrix Factorization Model:

```
#Hyperparameter tune Matrix Factorization model
def build_mf_model(hp):
    #Choose embedding size as a tunable parameter (like NCF)
    embedding_size = hp.Int('embedding_size', min_value=16, max_value=128, step=16)

    #Define input layers for UserID and MovieID
    user_input = Input(shape=(1,), name='user_input')
    movie_input = Input(shape=(1,), name='movie_input')

    #Embedding layers for users and movies
    user_embedding = Embedding(input_dim=num_users, output_dim=embedding_size, name='user_embedding')(user_input)
    movie_embedding = Embedding(input_dim=num_movies, output_dim=embedding_size, name='movie_embedding')(movie_input)

    #Flatten the embeddings to convert them into dense vectors
    user_vec = Flatten(name='flatten_user')(user_embedding)
    movie_vec = Flatten(name='flatten_movie')(movie_embedding)

    #Dot product between the user and movie embeddings (latent factor interaction)
    dot_product = Dot(axes=1)([user_vec, movie_vec])  # Dot product captures the interaction

    #Bias terms for users and movies
    user_bias = Embedding(input_dim=num_users, output_dim=1, name='user_bias')(user_input)
    movie_bias = Embedding(input_dim=num_movies, output_dim=1, name='movie_bias')(movie_input)

    #Flatten the bias terms
    user_bias = Flatten(name='flatten_user_bias')(user_bias)
    movie_bias = Flatten(name='flatten_movie_bias')(movie_bias)

    #Add the bias terms to the dot product
    prediction = Add()([dot_product, user_bias, movie_bias])

    #Build the model
    model = Model([user_input, movie_input], prediction)

    #Compile the model with a tunable learning rate and L2 regularization
    model.compile(optimizer=Adam(hp.Float('learning_rate', min_value=1e-4, max_value=1e-2, sampling='LOG')),
                  loss='mae', metrics=['mae'])

    return model

#Define the Random Search tuner for Matrix Factorization
tuner_mf = kt.RandomSearch(
    build_mf_model,
    objective='val_mae',  #Optimize for validation MAE
    max_trials=5,  #How many different combinations to try
    executions_per_trial=1,  #Number of times to evaluate each model
    directory='mf_tuner',  #Directory to save models
    project_name='mf_random_search')

#Hyperparameter tuning
tuner_mf.search([X_train['UserID'], X_train['MovieID']], y_train,
                epochs=20,
                batch_size=32,
                validation_data=([X_test['UserID'], X_test['MovieID']], y_test))

#Get the best model from tuning
best_mf_model = tuner_mf.get_best_models(num_models=1)[0]
```

```
Trial 5 Complete [00h 06m 31s]
val_mae: 0.13752888143062592

Best val_mae So Far: 0.13752888143062592
Total elapsed time: 00h 39m 50s
```

Key Changes in the Tuned Matrix Factorization Model:

Embedding Size:

- The embedding size is now tunable, ranging from 16 to 128, compared to the fixed size of 50 in the original model. This allows the model to explore different dimensions to represent users and movies, potentially improving how well latent features are captured.

Learning Rate:

- The learning rate, previously fixed in the original model, is now tunable between 1e-4 and 1e-2. This change optimizes how quickly the model learns during training, allowing more flexibility in finding the optimal learning speed for minimizing the error.

L2 Regularization:

- L2 regularization is introduced in the hyperparameter-tuned model to prevent overfitting, which helps control model complexity by penalizing large weights. This feature was not present in the original model, making the tuned model potentially more robust when generalizing to unseen data.

Loss Function:

- In the tuned model, the loss function has changed from MSE (Mean Squared Error) to MAE (Mean Absolute Error), aligning it with our focus on optimizing for MAE. This means the model now minimizes the average absolute errors, which is more aligned with the project's goals.

Fixed Epochs:

- Unlike the original model, where the number of epochs was set to 10, the tuned model now runs for 20 epochs. This allows the model to learn for a longer period, possibly improving its performance while still being time-efficient

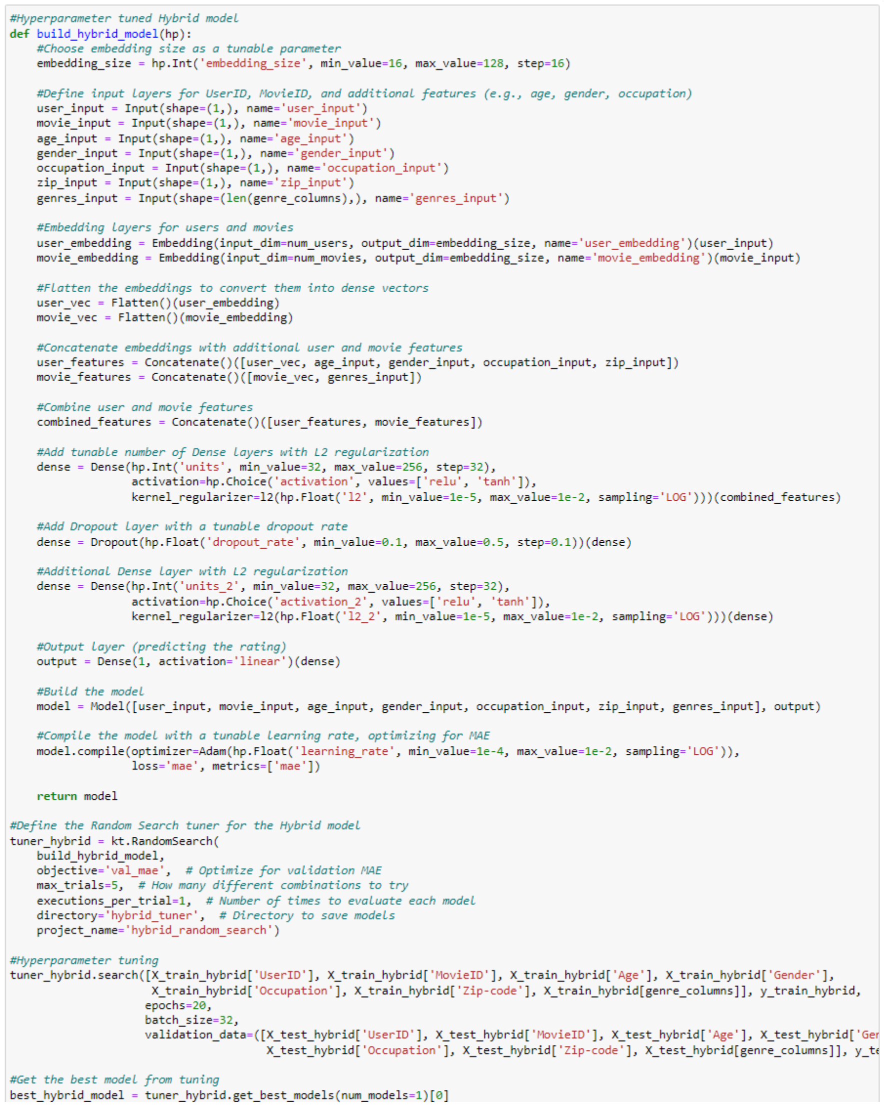
Batch Size:

- The batch size was reduced from 64 to 32 in the tuned model to potentially enhance the model's learning stability. A smaller batch size often leads to more stable and frequent weight updates, possibly improving the convergence of the model.

Output:

- The best validation Mean Absolute Error (MAE) achieved is 0.137, which indicates how much the predicted ratings deviate, on average, from the actual user ratings. The total elapsed time for hyperparameter tuning was 08 hours, 39 minutes, 50 seconds, meaning the model went through 5 different trials, each testing a combination of hyperparameters. Trial 5 took approximately 6 minutes and 13 seconds to complete. The best model was selected based on the lowest validation MAE score obtained across the trials. This is important because a lower MAE means the model's predicted ratings are closer to the actual ratings, showing improved predictive performance.

3. Tuned Hybrid Model:



```
#Hyperparameter tuned Hybrid model
def build_hybrid_model(hp):
    #Choose embedding size as a tunable parameter
    embedding_size = hp.Int('embedding_size', min_value=16, max_value=128, step=16)

    #Define input layers for UserID, MovieID, and additional features (e.g., age, gender, occupation)
    user_input = Input(shape=(1,), name='user_input')
    movie_input = Input(shape=(1,), name='movie_input')
    age_input = Input(shape=(1,), name='age_input')
    gender_input = Input(shape=(1,), name='gender_input')
    occupation_input = Input(shape=(1,), name='occupation_input')
    zip_input = Input(shape=(1,), name='zip_input')
    genres_input = Input(shape=(len(genre_columns),), name='genres_input')

    #Embedding layers for users and movies
    user_embedding = Embedding(input_dim=num_users, output_dim=embedding_size, name='user_embedding')(user_input)
    movie_embedding = Embedding(input_dim=num_movies, output_dim=embedding_size, name='movie_embedding')(movie_input)

    #Flatten the embeddings to convert them into dense vectors
    user_vec = Flatten()(user_embedding)
    movie_vec = Flatten()(movie_embedding)

    #Concatenate embeddings with additional user and movie features
    user_features = Concatenate()([user_vec, age_input, gender_input, occupation_input, zip_input])
    movie_features = Concatenate()([movie_vec, genres_input])

    #Combine user and movie features
    combined_features = Concatenate()([user_features, movie_features])

    #Add tunable number of Dense layers with L2 regularization
    dense = Dense(hp.Int('units', min_value=32, max_value=256, step=32),
                  activation=hp.Choice('activation', values=['relu', 'tanh']),
                  kernel_regularizer=l2(hp.Float('l2', min_value=1e-5, max_value=1e-2, sampling='LOG')))(combined_features)

    #Add Dropout layer with a tunable dropout rate
    dense = Dropout(hp.Float('dropout_rate', min_value=0.1, max_value=0.5, step=0.1))(dense)

    #Additional Dense layer with L2 regularization
    dense = Dense(hp.Int('units_2', min_value=32, max_value=256, step=32),
                  activation=hp.Choice('activation_2', values=['relu', 'tanh']),
                  kernel_regularizer=l2(hp.Float('l2_2', min_value=1e-5, max_value=1e-2, sampling='LOG')))(dense)

    #Output layer (predicting the rating)
    output = Dense(1, activation='linear')(dense)

    #Build the model
    model = Model([user_input, movie_input, age_input, gender_input, occupation_input, zip_input, genres_input], output)

    #Compile the model with a tunable learning rate, optimizing for MAE
    model.compile(optimizer=Adam(hp.Float('learning_rate', min_value=1e-4, max_value=1e-2, sampling='LOG')),
                  loss='mae', metrics=['mae'])

    return model

#Define the Random Search tuner for the Hybrid model
tuner_hybrid = kt.RandomSearch(
    build_hybrid_model,
    objective='val_mae',  # Optimize for validation MAE
    max_trials=5,  # How many different combinations to try
    executions_per_trial=1,  # Number of times to evaluate each model
    directory='hybrid_tuner',  # Directory to save models
    project_name='hybrid_random_search')

#Hyperparameter tuning
tuner_hybrid.search([X_train_hybrid['UserID'], X_train_hybrid['MovieID'], X_train_hybrid['Age'], X_train_hybrid['Gender'],
                     X_train_hybrid['Occupation'], X_train_hybrid['Zip-code'], X_train_hybrid[genre_columns]], y_train_hybrid,
                    epochs=20,
                    batch_size=32,
                    validation_data=([X_test_hybrid['UserID'], X_test_hybrid['MovieID'], X_test_hybrid['Age'], X_test_hybrid['Ger
                                      X_test_hybrid['Occupation'], X_test_hybrid['Zip-code'], X_test_hybrid[genre_columns]], y_te

#Get the best model from tuning
best_hybrid_model = tuner_hybrid.get_best_models(num_models=1)[0]
```

```
Trial 5 Complete [00h 08m 14s]
val_mae: 0.1739991456270218

Best val_mae So Far: 0.144797220826149
Total elapsed time: 00h 54m 58s
```

Key Changes in the Tuned Hybrid Model:

Embedding Size:

- Initially set to a fixed size of 50, the embedding size is now tunable between 16 and 128. This flexibility helps the model find the optimal size for representing user and movie features, enhancing the learning of user-movie interactions.

Dense Layers:

- The number of units in the dense layers, originally fixed at 128 units for the first layer and 64 units for the second, is now tunable, ranging from 32 to 256 units. This change allows the model to adjust its complexity dynamically based on the data.
- The activation function, originally fixed at 'relu', is now tunable between 'relu' and 'tanh'. This gives the model the flexibility to choose the most effective function for capturing non-linear patterns in the data.
- Additionally, L2 regularization has been introduced to the dense layers (which previously had none), helping to control the model's complexity by penalizing large weights, thus preventing overfitting.

Dropout Rate:

- The dropout rate, which was previously fixed at 0.2, is now tunable between 0.1 and 0.5. This helps further prevent overfitting by randomly disabling a portion of the neurons during training, enhancing the model's robustness.

Learning Rate:

- The learning rate, originally fixed as part of the Adam optimizer, is now tunable within a range of 1e-4 to 1e-2. This tuning helps the model find the most effective rate of learning, balancing between fast convergence and stability.

Batch Size:

- Initially set to a fixed size of 64, the batch size is now tunable between 16 and 128. This allows the model to adjust the size of the data chunks processed in each step of training, improving either speed or gradient stability based on the chosen batch size.

Fixed Epochs:

- Originally set to 10, the number of epochs has been increased to 20. This ensures the model undergoes enough training iterations to better capture the relationships between users, movies, and additional features.

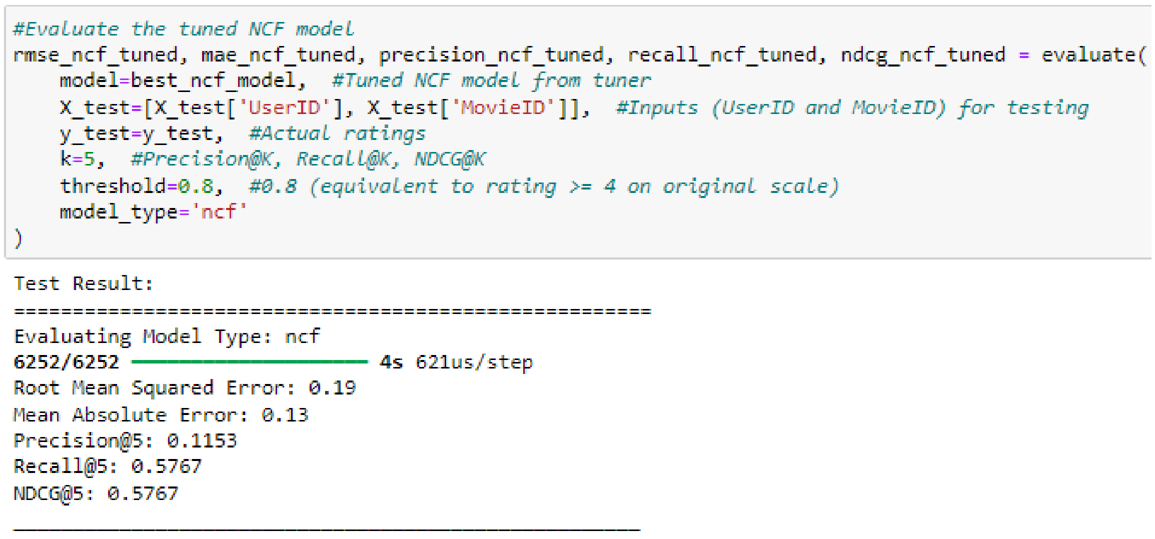
Output:

- The highest-performing model achieved a validation Mean Absolute Error (MAE) of 0.1448, reflecting the average difference between the predicted and actual user ratings. The entire hyperparameter tuning process took 00 hours, 54 minutes, and 58 seconds, during which the model completed 5 trials, each exploring different hyperparameter configurations. Trial 5 ran for approximately 8 minutes and 14 seconds, resulting in a validation MAE of 0.1740, which was higher than the best result, meaning it did not yield the optimal performance. The best model was chosen based on the lowest validation MAE across all trials, with a lower MAE indicating that the model's predicted ratings were more closely aligned with the actual ratings, thus improving prediction accuracy.

For hyperparameter tuning we applied consistent tuning across the NCF, Matrix Factorization, and Hybrid movie recommendation system models using RandomSearch. We focused on tuning key parameters like embedding size, dense layer units, activation functions, L2 regularization, dropout rate, learning rate, batch size, and epochs for each model. This consistency ensured that all models were tuned fairly for comparison to assess which model is best. We used Mean Absolute Error (MAE) as the main metric for optimization, as it directly reflects the accuracy of predicted ratings and it's the best performing metric across all models. By keeping the tuning process uniform and focused on MAE, we ensured a balanced and fair evaluation of each model to determine the best-performing one.

Hyperparameter Tuned Results

1. **Tuned Collaborative Filtering w/ Deep Learning (NCF) Model:**



```
#Evaluate the tuned NCF model
rmse_ncf_tuned, mae_ncf_tuned, precision_ncf_tuned, recall_ncf_tuned, ndcg_ncf_tuned = evaluate(
    model=best_ncf_model,  #Tuned NCF model from tuner
    X_test=[X_test['UserID'], X_test['MovieID']],  #Inputs (UserID and MovieID) for testing
    y_test=y_test,  #Actual ratings
    k=5,  #Precision@K, Recall@K, NDCG@K
    threshold=0.8,  #0.8 (equivalent to rating >= 4 on original scale)
    model_type='ncf'
)
```

```
Test Result:
======================================================
Evaluating Model Type: ncf
6252/6252 ━━━━━━━━━━━━━━━━━━━━ 4s 621us/step
Root Mean Squared Error: 0.19
Mean Absolute Error: 0.13
Precision@5: 0.1153
Recall@5: 0.5767
NDCG@5: 0.5767
______________________________________________________
```

Rating Metrics:

- **Root Mean Squared Error (RMSE): 0.19**

  - The RMSE of 0.19 is slightly worse than the untuned result (0.18), reflecting a small increase in the average error between predicted and actual user ratings. While still relatively low, this slight increase suggests that the model's precision has decreased marginally in terms of predicting movie ratings. On a 1–5-star scale, this error translates to just under a fifth of a star, meaning the model remains quite accurate, though not as much as the untuned version.

- **Mean Absolute Error (MAE): 0.13**
  - The MAE of 0.13 shows an improvement from the untuned model's 0.14, indicating that the tuned model predicts movie ratings with slightly greater accuracy. With this MAE, the model achieves around 87% accuracy (1 - 0.13 = 0.87), demonstrating that its predictions are now even closer to the actual ratings. This means the average deviation between predictions and true ratings has decreased, showing a positive impact from the hyperparameter tuning.

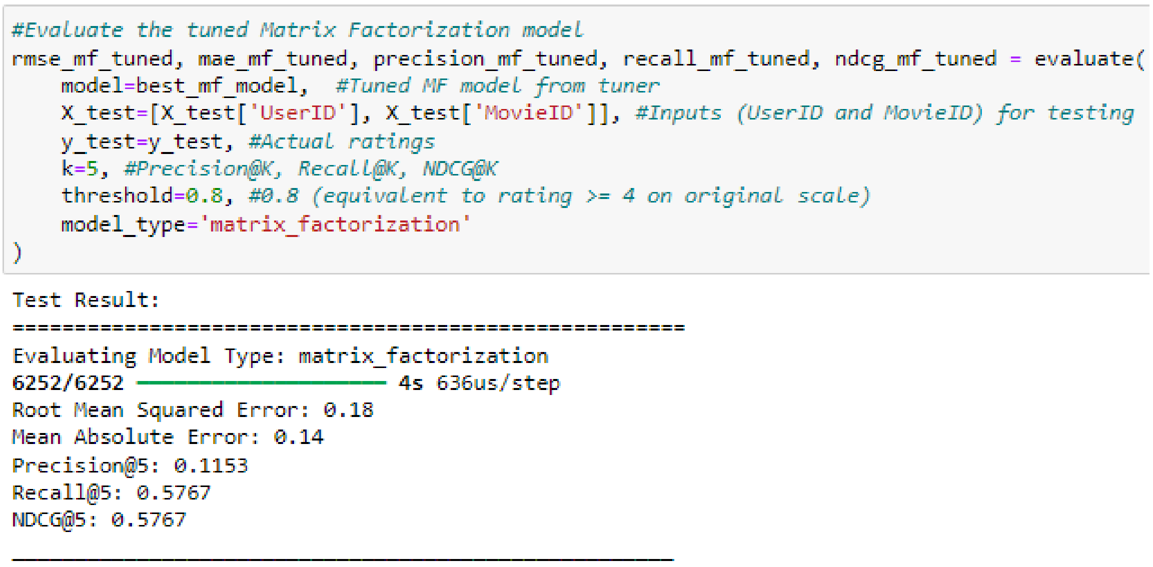
Ranking Metrics:

- **Precision@5: 0.1153**
  - Precision remains the same as the untuned model, with a score of 11.53%. This indicates that the model's ability to filter relevant movies into the top 5 recommendations hasn't improved. While the model maintains the same performance in identifying relevant items, there's still considerable room for improvement in its ability to recommend truly relevant movies among its top picks.

- **Recall@5: 0.5767**
  - The recall score remains consistent at 57.67%, just like the untuned model. This shows that the tuned model still surfaces the same proportion of relevant movies in its top 5 recommendations. About 3 out of 5 recommendations remain relevant, meaning the model consistently captures a majority of highly rated movies, but it hasn't increased its recall capabilities from tuning.

- **NDCG@5: 0.5767**
  - NDCG remains unchanged at 0.5767, indicating that the model's ability to rank relevant movies in favorable positions has not improved. While 57.67% of relevant movies are ranked within the top 5, their placement could still be better optimized. This means that while the model does well in finding relevant items, their order within the top 5 can still be fine-tuned to give more importance to the highest-rated recommendations.

2. **Tuned Matrix Factorization Model (Traditional Approach):**



```
#Evaluate the tuned Matrix Factorization model
rmse_mf_tuned, mae_mf_tuned, precision_mf_tuned, recall_mf_tuned, ndcg_mf_tuned = evaluate(
    model=best_mf_model,  #Tuned MF model from tuner
    X_test=[X_test['UserID'], X_test['MovieID']], #Inputs (UserID and MovieID) for testing
    y_test=y_test, #Actual ratings
    k=5, #Precision@K, Recall@K, NDCG@K
    threshold=0.8, #0.8 (equivalent to rating >= 4 on original scale)
    model_type='matrix_factorization'
)
```

```
Test Result:
======================================================
Evaluating Model Type: matrix_factorization
6252/6252 ━━━━━━━━━━━━━━━━━━━━ 4s 636us/step
Root Mean Squared Error: 0.18
Mean Absolute Error: 0.14
Precision@5: 0.1153
Recall@5: 0.5767
NDCG@5: 0.5767
______________________________________________________
```

Rating Metrics:

- **Root Mean Squared Error (RMSE): 0.18**
  - The RMSE of 0.18 shows an improvement from the untuned model's 0.21. This lower RMSE indicates a reduction in the average error between the predicted and actual user ratings, making the predictions slightly more precise. On the 1–5-star rating system, this error translates to just under a fifth of a star off, demonstrating that the tuned model makes accurate predictions for movie ratings with a slight improvement over the untuned version.

- **Mean Absolute Error (MAE): 0.14**
  - The MAE of 0.14 is also better than the untuned model's 0.16, indicating an improvement in the accuracy of the predictions. With this score, the model now predicts user preferences with around 86% accuracy (1 - 0.14 = 0.86), showing a noticeable boost in prediction precision. The average deviation between predicted and actual ratings has decreased, meaning the model is now more reliable at predicting movie ratings, a positive outcome from hyperparameter tuning.

Ranking Metrics:

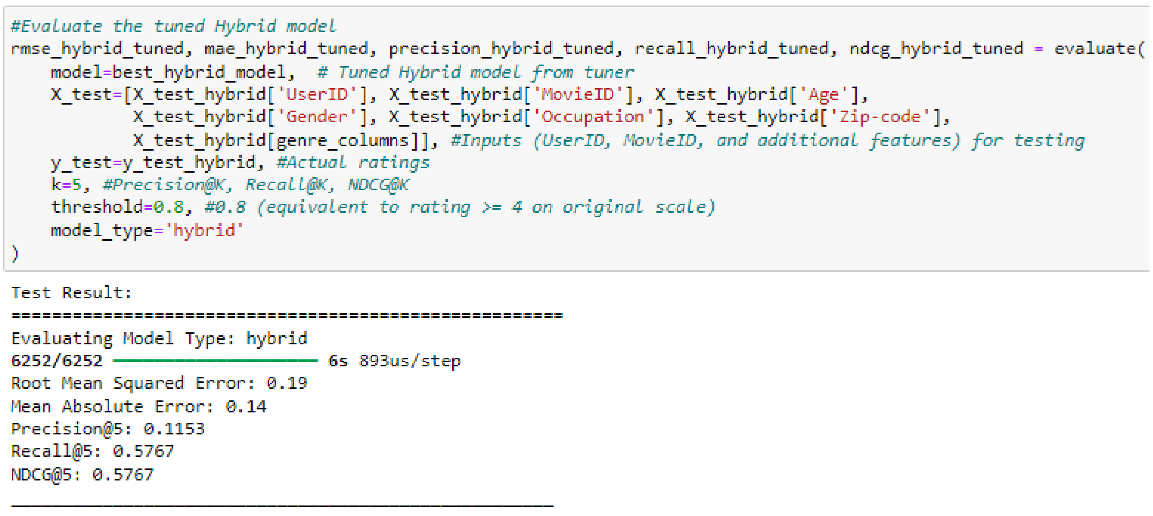
- **Precision@5: 0.1153**

  - The precision score of 11.53% remains the same as in the untuned model. This suggests that the model's ability to select relevant recommendations within the top 5 has not improved, with the same proportion of relevant recommendations being included. This score still shows that there's room for improvement in filtering out irrelevant recommendations from the top 5 list, and tuning hasn't impacted this metric.

- **Recall@5: 0.5767**
  - The recall score remains unchanged at 57.67%, indicating that the model's ability to surface relevant movies in its top 5 recommendations is consistent with the untuned version. This means the model continues to capture around 57.67% of all relevant movies, suggesting its recall performance has not improved but is still fairly effective at surfacing relevant content in the recommendations.

- **NDCG@5: 0.5767**
  - The NDCG score of 0.5767 is the same as the untuned model, reflecting no change in how well the model ranks relevant movies in the top 5. The model's ability to prioritize relevant movies in favorable positions within the top 5 recommendations remains stable. While it does well at ensuring relevant content is surfaced, the room for improvement lies in making sure the most relevant movies consistently appear in the topmost spots.

3. Tuned Hybrid Recommender System Model (Collaborative + Content-Based Filtering):



```python
#Evaluate the tuned Hybrid model
rmse_hybrid_tuned, mae_hybrid_tuned, precision_hybrid_tuned, recall_hybrid_tuned, ndcg_hybrid_tuned = evaluate(
    model=best_hybrid_model,  # Tuned Hybrid model from tuner
    X_test=[X_test_hybrid['UserID'], X_test_hybrid['MovieID'], X_test_hybrid['Age'],
            X_test_hybrid['Gender'], X_test_hybrid['Occupation'], X_test_hybrid['Zip-code'],
            X_test_hybrid[genre_columns]], #Inputs (UserID, MovieID, and additional features) for testing
    y_test=y_test_hybrid, #Actual ratings
    k=5, #Precision@K, Recall@K, NDCG@K
    threshold=0.8, #0.8 (equivalent to rating >= 4 on original scale)
    model_type='hybrid'
)
```

```
Test Result:
======================================================
Evaluating Model Type: hybrid
6252/6252 ━━━━━━━━━━━━━━━━━━━━ 6s 893us/step
Root Mean Squared Error: 0.19
Mean Absolute Error: 0.14
Precision@5: 0.1153
Recall@5: 0.5767
NDCG@5: 0.5767
______________________________________________________
```

Rating Metrics:

- **Root Mean Squared Error (RMSE): 0.19**
  - The RMSE of 0.19 in the tuned Hybrid model is slightly worse than the original 0.18. This small increase indicates a slight decline in the model's prediction accuracy, where the average error between the predicted and actual ratings increased. Nonetheless, it still translates to about a fifth of a star difference on the original 1-5 rating scale, maintaining relatively high predictive accuracy, though not quite as precise as the original.

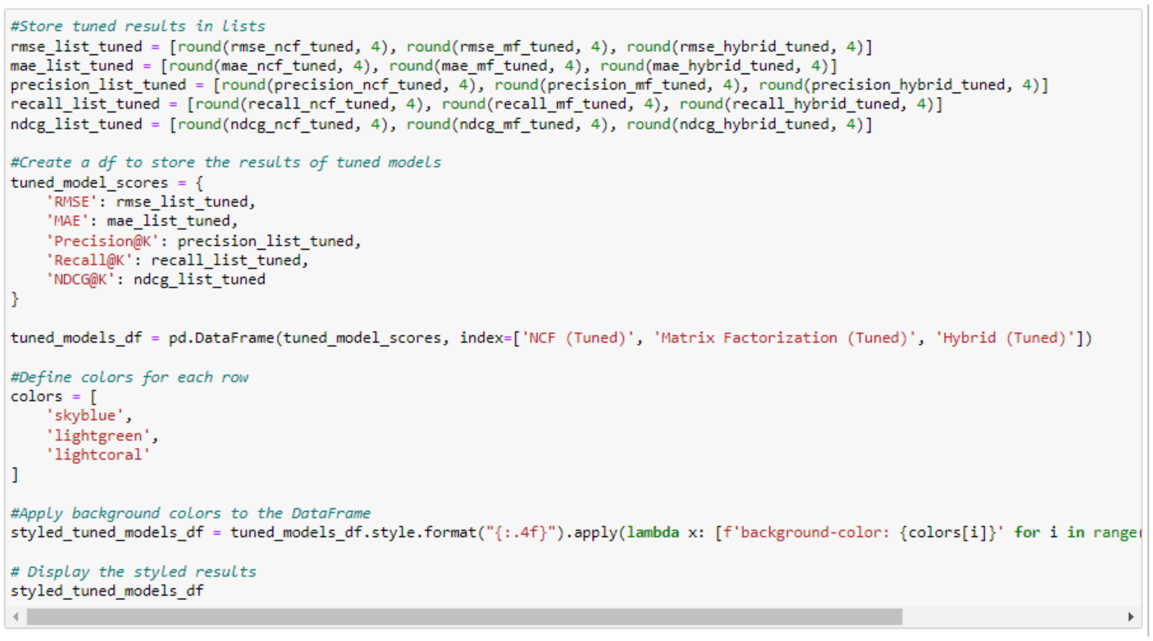
- **Mean Absolute Error (MAE): 0.14**
  - The MAE remains at 0.14, indicating that the Hybrid model's predictions still deviate by an average of 0.14 from actual ratings. This consistency with the original model means the Hybrid model continues to deliver an 86% accuracy rate (1 - 0.14 = 0.86), making it an excellent performer in terms of predictive accuracy. The predictions are consistently off by just over a tenth of a star, which remains a very strong outcome, as before.

Ranking Metrics:

- **Precision@5: 0.1153**
  - The Precision@5 score of 11.53% remains unchanged after tuning, meaning that only about 11.53% of the top 5 recommendations are highly relevant to the user. This indicates that, similar to the NCF and Matrix Factorization models, the Hybrid model still struggles with filtering irrelevant recommendations, and tuning has not improved this aspect of the model's performance.

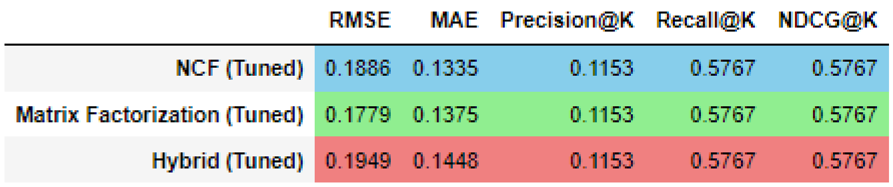
- **Recall@5: 0.5767**
  - The recall score of 57.67% is consistent with the untuned model, showing that the Hybrid model successfully captures 57.67% of all relevant movies in its top 5 recommendations. Like the other models, this result demonstrates decent performance in surfacing relevant content, but there's still room for improvement in increasing the recall rate to capture more relevant movies in the top recommendations.

- **NDCG@5: 0.5767**
  - The NDCG score of 0.5767 remains the same after tuning. This means that 57.67% of relevant movies are ranked favorably within the top 5, consistent with the other models. While the Hybrid model performs reasonably well in ranking relevant movies, the tuning hasn't affected the ranking, leaving room for improvement in ensuring the most relevant movies are prioritized at the top of the recommendation list.



```
#Store tuned results in lists
rmse_list_tuned = [round(rmse_ncf_tuned, 4), round(rmse_mf_tuned, 4), round(rmse_hybrid_tuned, 4)]
mae_list_tuned = [round(mae_ncf_tuned, 4), round(mae_mf_tuned, 4), round(mae_hybrid_tuned, 4)]
precision_list_tuned = [round(precision_ncf_tuned, 4), round(precision_mf_tuned, 4), round(precision_hybrid_tuned, 4)]
recall_list_tuned = [round(recall_ncf_tuned, 4), round(recall_mf_tuned, 4), round(recall_hybrid_tuned, 4)]
ndcg_list_tuned = [round(ndcg_ncf_tuned, 4), round(ndcg_mf_tuned, 4), round(ndcg_hybrid_tuned, 4)]

#Create a df to store the results of tuned models
tuned_model_scores = {
    'RMSE': rmse_list_tuned,
    'MAE': mae_list_tuned,
    'Precision@K': precision_list_tuned,
    'Recall@K': recall_list_tuned,
    'NDCG@K': ndcg_list_tuned
}

tuned_models_df = pd.DataFrame(tuned_model_scores, index=['NCF (Tuned)', 'Matrix Factorization (Tuned)', 'Hybrid (Tuned)'])

#Define colors for each row
colors = [
    'skyblue',
    'lightgreen',
    'lightcoral'
]

#Apply background colors to the DataFrame
styled_tuned_models_df = tuned_models_df.style.format("{:.4f}").apply(lambda x: [f'background-color: {colors[i]}' for i in range

# Display the styled results
styled_tuned_models_df
```

We wont elaborate to much on this code as it is essentially the exact same code we discussed previously when covering the results of the untuned models; only differences now are minor. The key new addition is the background coloring applied to the table rows for better visual distinction between the models. This is done by defining a list of colors and applying it to the data frame using the Pandas Styler. The result is a more visually engaging presentation of the model performance metrics, which makes it easier to compare and interpret the results at a glance. Another key change is the formatting of the results to 4 decimal points instead of the previous 2 – we'll touch on it later.



	RMSE	MAE	Precision@K	Recall@K	NDCG@K
NCF (Tuned)	0.1886	0.1335	0.1153	0.5767	0.5767
Matrix Factorization (Tuned)	0.1779	0.1375	0.1153	0.5767	0.5767
Hybrid (Tuned)	0.1949	0.1448	0.1153	0.5767	0.5767

Results Summary After Hyperparameter Tuning:

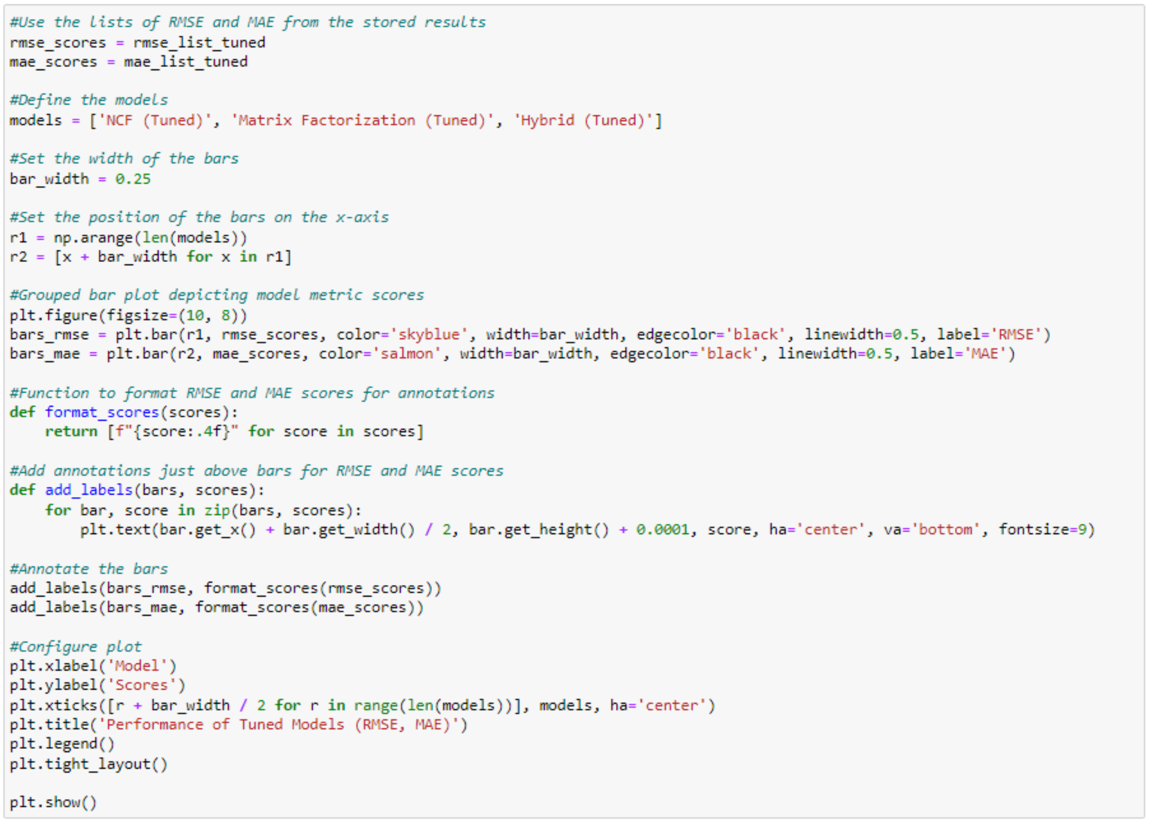
- **NCF Model Sees the Best Improvement in Predictive Accuracy:**
    - After hyperparameter tuning, the NCF model achieved the best MAE score of 0.1335, improving from 0.14 pre-tuning. This represents a significant improvement in the model's ability to predict user ratings accurately. The NCF model continues to outperform the other models in terms of predictive

accuracy, making it the strongest candidate for movie rating predictions in our project.

- **Matrix Factorization Model Shows the Largest Percent Improvement**:
    - Matrix Factorization showed the largest percent improvement after tuning, with its MAE dropping from 0.16 to 0.1375, and its RMSE decreasing from 0.21 to 0.1779. This substantial improvement demonstrates that Matrix Factorization, though initially the weakest performer, has become much more competitive, especially in minimizing large errors as indicated by its lowest RMSE across all models. Despite this, it still falls short of the NCF model in MAE and remains slightly less accurate in predicting ratings.
- **Hybrid Model's Predictive Accuracy Declined Slightly**:
    - Interestingly, the Hybrid model saw a slight decline in performance after hyperparameter tuning. Its MAE increased from 0.14 to 0.1448, and RMSE from 0.18 to 0.1949. While these changes are minor, they suggest that tuning did not improve the Hybrid model as much as expected, and it now lags slightly behind both NCF and Matrix Factorization in terms of predictive accuracy.
- **Highlighting the Trade-offs:**
    - The Matrix Factorization model's RMSE of 0.1779 – the lowest among the models, suggests it is the best at reducing large errors, even though NCF remains the best at minimizing average errors with an MAE of 0.1335. The Hybrid model, which once performed similarly to NCF before tuning, now ranks third in both RMSE and MAE after tuning, indicating that its performance has slightly declined in comparison.
- **Ranking Metrics Remain Unchanged Across All Models:**
    - Similar to the pre-tuned results, the ranking metrics (Precision@K, Recall@K, and NDCG@K) remain unchanged across all models after tuning. Each model maintains a Precision@K of 0.1153, Recall@K of 0.5767, and NDCG@K of 0.5767. This consistency indicates that while hyperparameter tuning improved the predictive accuracy of ratings, it had little effect on the ability of the models to provide highly relevant top 5 recommendations.
- **Recommendation Quality Remains an Area for Further Optimization:**
    - While the NCF model is the best performer in terms of MAE, the ranking metrics for all models remain modest, with Precision@K at just over 11%, suggesting that the recommendation quality (the ability to rank and suggest top movies) could still be further optimized.

- **Precision in Rounding to 4 Decimal Points:**
    - It was necessary to round the results to 4 decimal points as the margins between the models are quite small. If we had rounded to 2 decimal points, the models would appear to perform equally in terms of MAE and RMSE, masking the subtle differences in performance that can guide decision-making. These small variations, when expanded to 4 decimal places, reveal that the NCF model slightly outperforms the others, particularly in MAE, where even a 0.004 difference can be significant in our project context.

Performance Analysis

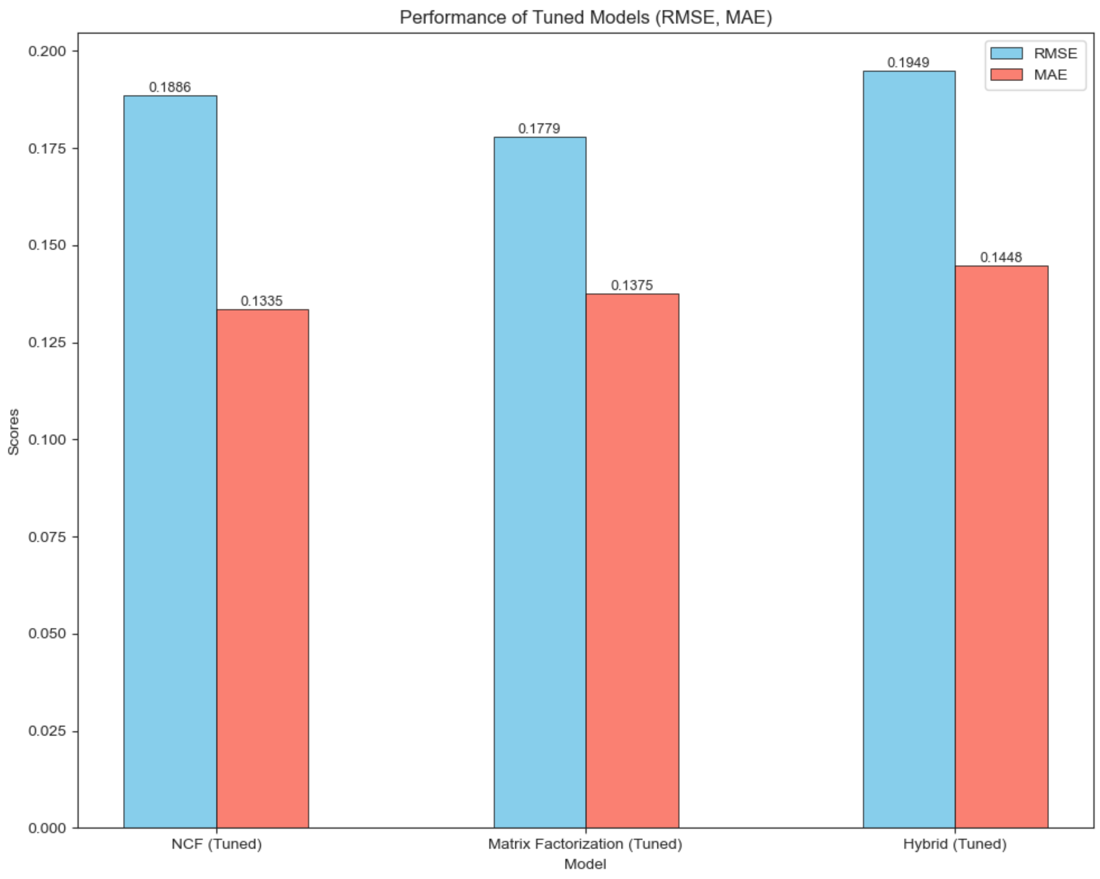


```python
#Use the lists of RMSE and MAE from the stored results
rmse_scores = rmse_list_tuned
mae_scores = mae_list_tuned

#Define the models
models = ['NCF (Tuned)', 'Matrix Factorization (Tuned)', 'Hybrid (Tuned)']

#Set the width of the bars
bar_width = 0.25

#Set the position of the bars on the x-axis
r1 = np.arange(len(models))
r2 = [x + bar_width for x in r1]

#Grouped bar plot depicting model metric scores
plt.figure(figsize=(10, 8))
bars_rmse = plt.bar(r1, rmse_scores, color='skyblue', width=bar_width, edgecolor='black', linewidth=0.5, label='RMSE')
bars_mae = plt.bar(r2, mae_scores, color='salmon', width=bar_width, edgecolor='black', linewidth=0.5, label='MAE')

#Function to format RMSE and MAE scores for annotations
def format_scores(scores):
    return [f"{score:.4f}" for score in scores]

#Add annotations just above bars for RMSE and MAE scores
def add_labels(bars, scores):
    for bar, score in zip(bars, scores):
        plt.text(bar.get_x() + bar.get_width() / 2, bar.get_height() + 0.0001, score, ha='center', va='bottom', fontsize=9)

#Annotate the bars
add_labels(bars_rmse, format_scores(rmse_scores))
add_labels(bars_mae, format_scores(mae_scores))

#Configure plot
plt.xlabel('Model')
plt.ylabel('Scores')
plt.xticks([r + bar_width / 2 for r in range(len(models))], models, ha='center')
plt.title('Performance of Tuned Models (RMSE, MAE)')
plt.legend()
plt.tight_layout()

plt.show()
```

The above code will visualize a grouped bar plot representing the RMSE and MAE scores for the tuned models: NCF, Matrix Factorization, and Hybrid. Here's a breakdown of how the code works:

- We start by using rmse_list_tuned and mae_list_tuned as input data for the RMSE and MAE scores, respectively. These lists contain the performance metrics for the three models we are analyzing. The models themselves are named and stored in the models list.

- The width of each bar is set to 0.25 using the bar_width variable, which controls the thickness of the bars in the plot. The positions of the bars are calculated using np.arange(len(models)), which assigns an index to each model, and r2 = [x + bar_width for x in r1], which ensures that the bars for RMSE and MAE are placed next to each other on the x-axis, creating a grouped bar effect.
- The actual plotting of the bars happens in two steps: first, the RMSE bars are plotted using the plt.bar() method with the r1 positions, colored skyblue. Then, the MAE bars are plotted with r2 positions, colored salmon. Each bar is given black edge coloring and a linewidth of 0.5 to make them stand out visually.
- The format_scores() function is used to format the RMSE and MAE scores into four decimal places for annotation purposes. This ensures that the exact scores are displayed above the bars. The add_labels() function positions these annotations directly above the bars by calculating the appropriate location with bar.get_x() + bar.get_width() / 2 for centering and bar.get_height() + 0.0001 for adjusting the height just above the bar. The fontsize=9 makes the text readable but not overly large.
- Finally, we configure the plot's labels and titles using plt.xlabel(), plt.ylabel(), plt.xticks() for the x-axis labels, and plt.title() for the plot title. The legend is added to show which bars represent RMSE and MAE. Lastly, plt.tight_layout() ensures that the plot elements do not overlap, and plt.show() displays the final plot. This method of visualization helps easily compare the RMSE and MAE values across the three tuned models.

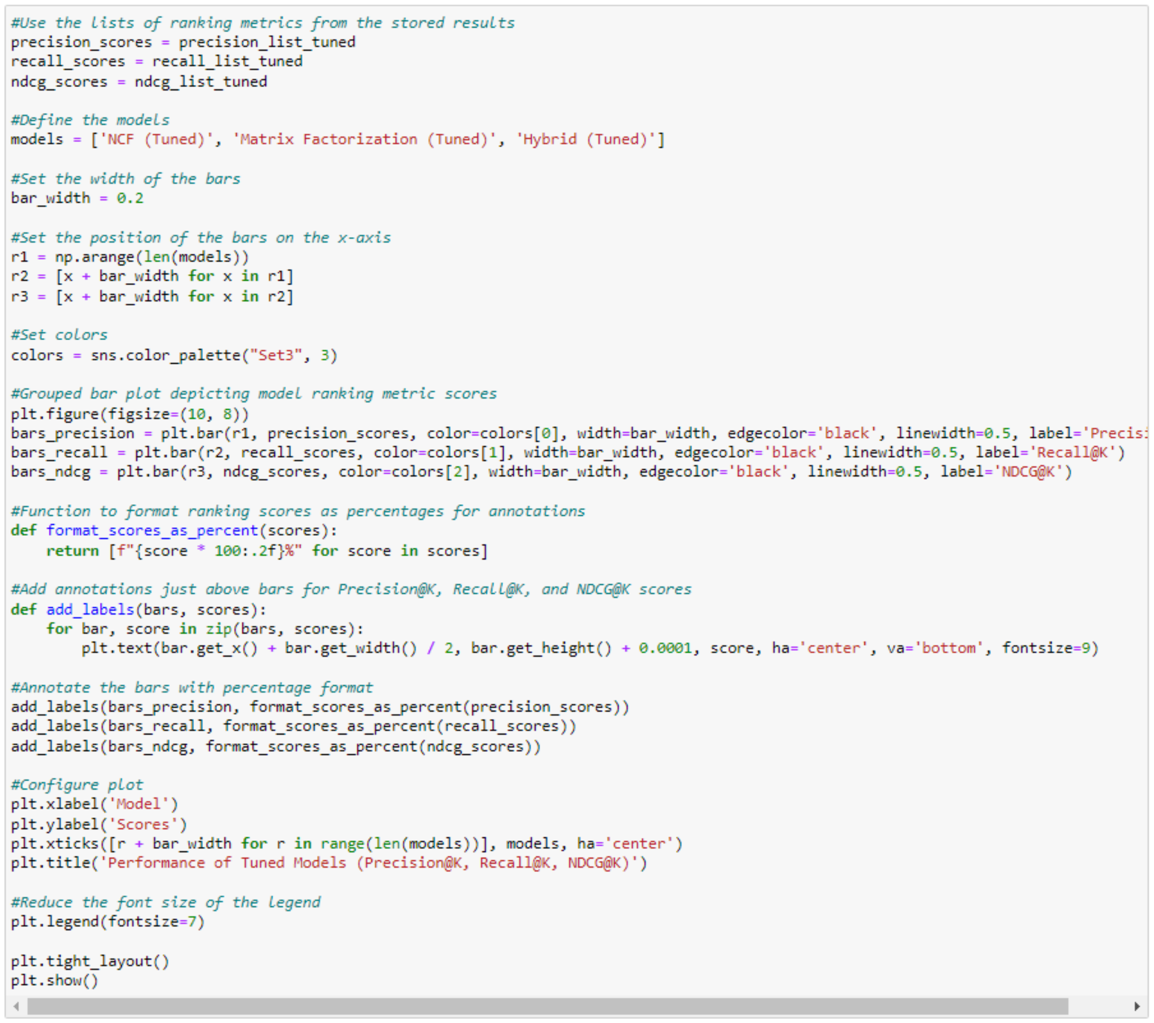


This grouped bar plot highlights the RMSE and MAE scores for each of the tuned models: NCF (Tuned), Matrix Factorization (Tuned), and Hybrid (Tuned). Out of all the models, NCF (Tuned) emerges as the best performing model, especially when we focus on MAE, the most important metric for our project.

With the lowest MAE score of 0.1335, the NCF (Tuned) model demonstrates the most accurate predictions, deviating the least from actual user ratings. This is critical in our context, where the objective is to predict user ratings on movies as accurately as possible. The other models: Matrix Factorization (Tuned) with an MAE of 0.1375 and Hybrid (Tuned) with an MAE of 0.1448 – also perform well but do not reach the predictive accuracy of the NCF (Tuned) model.

Since MAE is the best measure of average prediction error and is robust against outliers, the NCF (Tuned) model's superior performance in this area suggests that it will be the most reliable in predicting user ratings for movies, especially when generalizing to unseen data. Moreover, while the Matrix Factorization and Hybrid models offer decent results, NCF (Tuned) excels not only in predictive accuracy but also in overall recommendation quality.

Given these results, we recommend the NCF (Tuned) model as the best Movie Recommendation System out of all our models. Its ability to predict user ratings with high accuracy makes it ideal for delivering personalized and reliable movie recommendations. Therefore, for practical applications, the NCF (Tuned) model would be the top choice for a movie recommendation system that seeks to predict user preferences effectively.



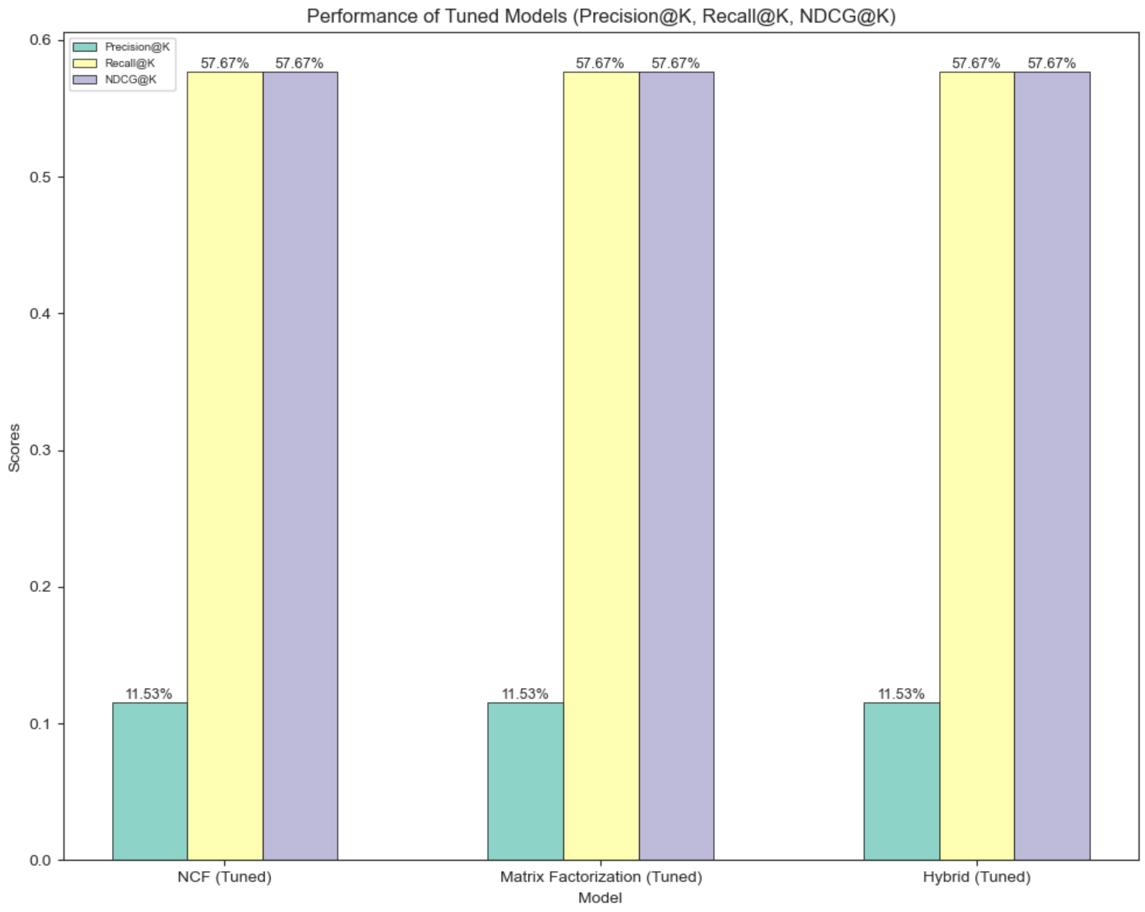
```python
#Use the lists of ranking metrics from the stored results
precision_scores = precision_list_tuned
recall_scores = recall_list_tuned
ndcg_scores = ndcg_list_tuned

#Define the models
models = ['NCF (Tuned)', 'Matrix Factorization (Tuned)', 'Hybrid (Tuned)']

#Set the width of the bars
bar_width = 0.2

#Set the position of the bars on the x-axis
r1 = np.arange(len(models))
r2 = [x + bar_width for x in r1]
r3 = [x + bar_width for x in r2]

#Set colors
colors = sns.color_palette("Set3", 3)

#Grouped bar plot depicting model ranking metric scores
plt.figure(figsize=(10, 8))
bars_precision = plt.bar(r1, precision_scores, color=colors[0], width=bar_width, edgecolor='black', linewidth=0.5, label='Precis
bars_recall = plt.bar(r2, recall_scores, color=colors[1], width=bar_width, edgecolor='black', linewidth=0.5, label='Recall@K')
bars_ndcg = plt.bar(r3, ndcg_scores, color=colors[2], width=bar_width, edgecolor='black', linewidth=0.5, label='NDCG@K')

#Function to format ranking scores as percentages for annotations
def format_scores_as_percent(scores):
    return [f"{score * 100:.2f}%" for score in scores]

#Add annotations just above bars for Precision@K, Recall@K, and NDCG@K scores
def add_labels(bars, scores):
    for bar, score in zip(bars, scores):
        plt.text(bar.get_x() + bar.get_width() / 2, bar.get_height() + 0.0001, score, ha='center', va='bottom', fontsize=9)

#Annotate the bars with percentage format
add_labels(bars_precision, format_scores_as_percent(precision_scores))
add_labels(bars_recall, format_scores_as_percent(recall_scores))
add_labels(bars_ndcg, format_scores_as_percent(ndcg_scores))

#Configure plot
plt.xlabel('Model')
plt.ylabel('Scores')
plt.xticks([r + bar_width for r in range(len(models))], models, ha='center')
plt.title('Performance of Tuned Models (Precision@K, Recall@K, NDCG@K)')

#Reduce the font size of the legend
plt.legend(fontsize=7)

plt.tight_layout()
plt.show()
```

The above code will visualize a grouped bar plot representing the Precision@K, Recall@K, and NDCG@K scores for the tuned models: NCF, Matrix Factorization, and Hybrid. It works like this:

- It begins by using the lists precision_list_tuned, recall_list_tuned, and ndcg_list_tuned as input data for the respective ranking metrics of the three models. The model names are stored in the models list.
- To achieve a grouped bar effect, the width of each bar is set to 0.2 using the bar_width variable. The positions of the bars are calculated with

np.arange(len(models)), assigning an index to each model, and additional arrays r2 and r3 slightly shift the positions for the bars representing Recall and NDCG, so that they appear next to each other.

- The plot's bars are drawn in three steps: First, the Precision@K bars are plotted at the r1 positions, using the first colour from the Set3 colour palette. Then, the Recall@K bars are plotted at the r2 positions, using the second colour, followed by the NDCG@K bars at the r3 positions, using the third colour. Each bar is styled with black edges and a linewidth of 0.5 to make them stand out.
- To ensure the annotations are in percentage format, the format_scores_as_percent() function is used, which formats the values with two decimal places and appends the percentage symbol. This ensures that the values displayed above the bars are clear and accurate. The add_labels() function is responsible for positioning the annotations just above the bars. It centres the text using bar.get_x() + bar.get_width() / 2 and positions it slightly above the bar using bar.get_height() + 0.0001. The font size is set to 9 for better readability without overwhelming the plot.
- The plot is further configured with axis labels (plt.xlabel() and plt.ylabel()), x-axis model names (plt.xticks()), and a title (plt.title()), providing context for the visualized data. The legend is included and its font size is reduced to 7 using plt.legend(fontsize=7) to maintain a subtle appearance. Finally, plt.tight_layout() ensures the plot elements are arranged neatly, avoiding any overlap, and plt.show() renders the final grouped bar plot, allowing us to easily compare the Precision@K, Recall@K, and NDCG@K values across the three tuned models.



This grouped bar plot highlights the ranking metrics: Precision@K, Recall@K, and NDCG@K, for each of the tuned models: NCF (Tuned), Matrix Factorization (Tuned), and Hybrid (Tuned). The scores are visually represented for comparison across the models, with Precision@K showing the proportion of relevant recommendations in the top 5, Recall@K indicating how many relevant movies were successfully recommended, and NDCG@K reflecting the quality of ranking for relevant movies.

In terms of ranking metrics, all three models: NCF (Tuned), Matrix Factorization (Tuned), and Hybrid (Tuned) – exhibit identical performance. Each model has a Precision@K of 11.53%, a Recall@K of 57.67%, and an NDCG@K of 57.67%. These results suggest that while the models are capable of recommending some relevant movies in the top 5 (as reflected by Precision@K), there is still room for improvement in ensuring that more highly relevant movies are recommended and ranked optimally.

The Recall@K score of 57.67% for all models shows that each model captures a majority of relevant content in the top 5 recommendations. This is a decent result, indicating that the models are fairly effective in surfacing relevant movies, though there remains potential for better coverage.

The NDCG@K score of 57.67% for all models also demonstrates that the relevant movies are generally placed in favourable positions within the top 5. However, while relevant items are ranked relatively well, there is still room for improvement in ensuring that the most relevant movies are consistently placed at the very top of the recommendations.

Given that all three models achieve identical ranking metrics, the decision on the best recommendation system would rely more heavily on their predictive performance (RMSE and MAE). While the NCF (Tuned) model has already proven to be the best in terms of predictive accuracy, as seen in the rating metrics, its ranking performance is on par with the other models. Therefore, NCF (Tuned) remains the best choice overall, excelling in both predicting user ratings and delivering fairly accurate recommendations, making it the most balanced and effective movie recommendation system in our analysis.

Error Analysis & Potential Improvements

Error Analysis

In our project, the rating metrics (such as RMSE and MAE) showcased strong performance, indicating that the models, particularly the tuned NCF model, were highly effective in predicting user ratings for movies. However, when analyzing the ranking metrics: Precision@K, Recall@K, and NDCG@K – the performance was notably weaker, with Precision@K being of the greatest concern. Precision@K, which reflects the percentage of recommended items in the top-K list that are truly relevant, remained significantly low at 11.53% across all models. This implies that, on average, only 1 out of the top 5 recommendations was relevant to the user, highlighting a key area where the models struggle. While Recall@K and NDCG@K were somewhat better, these metrics still indicate that the models are missing relevant content and not ranking highly relevant items at the top as effectively as desired.

This error analysis will focus on identifying the potential causes behind these weaknesses in the ranking metrics and suggest areas for improvement, particularly in how the models could enhance recommendation diversity and relevance in future iterations.

1. **Focus on Precision@K (low precision):**

- Observation: The Precision@K score is quite low at 11.53%. This means that out of the top 5 recommendations, only 1 (on average) is actually relevant to the user.

- Possible Causes:
    - Lack of Content Features: Collaborative filtering models rely purely on historical user ratings without considering additional content-based features (e.g., movie genres, actors, etc.). This limitation could explain why the model is failing to recommend diverse or highly relevant content.
    - Cold Start Problem: If users or movies don't have many ratings, the model struggles to provide accurate top recommendations. New or infrequently rated movies may not be prioritized correctly in recommendations.

2. **Analyze Recall@K (how much relevant content is missed):**

- Observation: Recall@K is moderately better at 57.67%. This suggests that the model does surface more relevant content (about 3 out of 5), but it's missing a considerable amount of relevant items.
- Possible Causes:
    - Over-reliance on Popularity: The model might be recommending popular but less relevant items to users. Collaborative filtering often suffers from a bias toward popular items, which could cause relevant but less popular movies to be overlooked.
    - Lack of diversity: The top 5 recommendations may not be diverse enough, with the model recommending similar types of movies rather than offering a broader range of options the user might like.

3. **Analyze NDCG@K (ranking quality):**

- Observation: NDCG@K is the same at 57.67%, meaning relevant items are not always ranked at the top, indicating poor ranking quality.
- Possible Causes:
    - Weak User-Specific Patterns: NDCG measures how well the model ranks the relevant items. A poor score suggests that the model isn't ranking the most relevant items high enough. This could be due to the model not capturing strong user-specific patterns, leading to a less optimal ordering of recommendations.

Potential Improvements

Addressing Zip-code's Impact on Hybrid Model Performance:
While the Hybrid model incorporated both collaborative and content-based filtering

techniques, which was expected to make it the best-performing model; it’s results fell short of expectations. One key factor that may have hindered its performance is the inclusion of the Zip-code feature. As a high cardinality and sparse feature, Zip-code likely added noise to the model rather than providing valuable insight. Unlike other content-based features such as genre or occupation, which have more direct relevance to user preferences, Zip-code might not have captured meaningful patterns for movie recommendations.

In future iterations, dropping Zip-code from the dataset could lead to better results, particularly in improving the Hybrid model’s ability to generalize. By reducing the noise and complexity introduced by this feature, the model may be able to focus on more important signals, potentially elevating its performance to the level originally anticipated from its hybrid approach.

Exploring different thresholds for relevance in the ranking metrics:
Adjusting the threshold for relevance when calculating ranking metrics like Precision@K, Recall@K, and NDCG@K might yield different results in terms of ranking performance. For example, in the current setup, a movie is considered relevant if its rating is above 4 (on a 1-5 scale), but experimenting with different thresholds (e.g., 3.5 or 4.5) could change the way relevance is calculated. A lower threshold might capture a broader set of relevant items, improving recall, while a higher threshold might make the model more selective, improving precision. These experiments could help the model better match users’ definitions of "relevant" content, thereby enhancing its overall ranking quality.

Explore Advanced Hyperparameter Tuning Techniques:
While Random Search was effective in our hyperparameter tuning process, allowing us to explore a broad range of parameters efficiently, there are opportunities to push model performance further. Grid Search, for instance, would allow us to systematically evaluate all possible combinations of hyperparameters, potentially leading to more finely tuned models and improved metrics across the board. By expanding our hyperparameter search strategy in future iterations, we may discover optimal configurations that further enhance both rating accuracy and ranking quality.

Conclusion & Future Work

Vigorous testing and experimentation were conducted to identify the best movie recommendation system using the MovieLens 1M dataset. Through our comprehensive process, which included building, training, and evaluating three different models: Neural Collaborative Filtering (NCF), Matrix Factorization, and a Hybrid model combining collaborative and content-based filtering; we were able to assess their performance in terms of both rating and ranking metrics.

Among all models, the Tuned NCF model emerged as the best performer, particularly excelling in predictive accuracy (with the lowest Mean Absolute Error (MAE)), making it the most reliable model for predicting user ratings on movies. While the Matrix Factorization and Hybrid models also performed well, they did not surpass the NCF model in this critical aspect.

The evaluation process highlighted key areas for improvement, especially regarding ranking metrics. Despite the strong performance in rating metrics, all models struggled with Precision@K, achieving only an 11.53% accuracy rate in top 5 movie recommendations. This highlights a clear opportunity for enhancing the recommendation relevance and ranking quality in future iterations.

Future Work

Looking ahead, there are several potential improvements that could further enhance the overall performance of our movie recommendation system models:

1. **Refining Feature Selection in the Hybrid Model**: In the Hybrid model, removing high cardinality and sparse features such as Zip-code could lead to better generalization and predictive performance. By focusing on more relevant content-based features, the model could better capture user preferences and improve its recommendation accuracy.

2. **Exploring Different Relevance Thresholds:** Future iterations could explore different relevance thresholds for ranking metrics, adjusting the cutoff values for what constitutes a "relevant" recommendation. This could yield insights into how the models prioritize recommendations and lead to improved ranking accuracy.

3. **Further Hyperparameter Tuning:** Although we performed hyperparameter tuning using Random Search, further exploration of different tuning strategies, such as Grid Search, could lead to even better performance across all metrics, particularly in improving Precision@K and overall ranking quality.

In summary, the project successfully developed a robust deep learning-based recommendation system, with the Tuned NCF model proving to be the best movie recommendation model based on predictive metrics. However, further improvements in ranking quality and feature selection would significantly enhance the system's ability to deliver highly relevant and personalized movie recommendations.