NAGALAND UNIVERSITY

(A Central University Established by an Act of Parliament 1989)

SCHOOL OF ENGINEERING AND TECHNOLOGY

PROJECT REPORT ON

MOVIE RECOMMENDATION SYSTEM

(Based on collaborative filtering)

“MOVIE RECOMMMENDATION SYSTEM”

(based on collaborative filtering)

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE OF

BACHELOR OF TECHNOLOGY
IN
INFORMATION TECHNOLOGY

IN THE SCHOOL OF ENGINEERING & TECHNOLOGY

NAGALAND UNIVERSITY

By

JOONMONI KAKOTY & BIBEK DEBNATH

Roll no- 2005159 Roll no -2105188

Under the supervision of

Mr. Teisovi Angami (Project Guide)

Assistant Professor

Dept. of Information Technology SET, Nagaland University

Department of Information Technology
School of Engineering & Technology
NAGALAND UNIVERSITY, Dimapur, Nagaland, India

June 2024

SCHOOL OF ENGINEERING AND TECHNOLOGY

NAGALAND UNIVERSITY

(A central university established by an act of parliament 1989)

D.C COURT JUNCTION DIMAPUR NAGALAND-797112

NAGALAND UNIVERSITY

(A Central University Established by an Act of Parliament 1989)

SCHOOL OF ENGINEERING AND TECHNOLOGY

D.C COURT JUNCTION DIMAPUR NAGALAND-797112

BONAFIDE CERTIFICATE

This is to certify that this project report entitled "MOVIE RECOMMENDATION SYSTEM (Based on Collaborative Filter)" is a bonafide record of work done by JOONMONI KAKOTY (ROLL NO. 2005159), BIBEK DEBNATH (ROLL NO. 2105188), for the partial fulfilment of the requirements for the Bachelor Degree in Department of Information & Technology, Nagaland University, and has been carried out under my supervision and guidance.

Prof. Sujata Dash
Head of the department
Department of IT
Date:
Place: Dimapur, Nagaland

Mr. Teisovi Angami
Project Guide & Assistant Professor
Department of IT
Date:
Place: Dimapur, Nagaland

Examiner:

ACKNOWLEDGEMENT

At the very outset we take this opportunity to convey our heartfelt gratitude to those people whose co-operation, suggestions and support helped us to accomplish the project successfully.

An in-depth study of the project topic chosen requires continued interest and persistent guidance from the project guide, **Mr. Teisovi Angami**, Assistant Professor, Department of Information Technology, SET, Nagaland University. He has always been associated with us and given useful suggestions. We wish to express our deep gratitude and indebtedness to him. We respectfully recollect his constant encouragement, kind attention and keen interest throughout the course of our work. We are highly indebted to him for the way he modeled and structured our work with his valuable tips and suggestions that he accorded to us in every respect of our work.

We express our sincere thanks to all the faculty members of the Department of Information Technology, for providing the encouragement and environment for the success of our project.

Our heartfelt thanks are also due to our friends for their encouragement, co-operation and suggestions without which this project work could not have been completed.

In the end, we would be failing in our duties if we do not express our heartfelt gratitude to our family whose constant inspiration and patience have helped us to complete this work. And last but not the least we would like to thank God for all he has given us till today.

Name of Students: Signature

JOONMONI KAKOTY

BIBEK DEBNATH

DECLARATION

We hereby declare that the project work entitled “MOVIE RECOMMENDATION SYSTEM (Based on collaborative filter)” submitted to the Department of Information Technology, School of Engineering and Technology, Nagaland University, Dimapur- 797112 Nagaland is a record of an original work done by us under the guidance of **Mr. Teisovi Angami** Assistant Professor Department of Information Technology, SET-NU and this project is submitted as a partial fulfillment required for the award of degree of Bachelor of Technology in Information Technology.

We further declare that the result embodied in this project have not been submitted to any other University or Institute for the award of any degree or diploma. Due acknowledgement have been extended to the sources for any information relevant to the project work.

Date:
Place: School of Engineering and Technology, Dimapur, Nagaland.

Project Members	Signature
JOONMONI KAKOTY (2005159)	
BIBEK DEBNATH (2105188)	

CONTENTS

ABSTRACT

In the era of digital entertainment, movie recommendation systems have become an essential tool for enhancing user experience by suggesting films tailored to individual preferences. This project focuses on developing a movie recommendation system using collaborative filtering, a popular method that leverages user interaction data to provide personalized recommendations. The system is designed to predict a user's potential interests by analyzing patterns and relationships between users and movies.

Collaborative filtering techniques can be broadly categorized into two types: user-based and item-based filtering. User-based collaborative filtering recommends movies to a user by finding similar users who have shown comparable tastes, while item-based collaborative filtering suggests movies based on the similarity between items liked by the user. This project implements both approaches to evaluate their effectiveness and performance.

The dataset utilized for this project includes user ratings for a diverse set of movies, which is split into training and testing sets to assess the accuracy of the recommendations. Key metrics such as Mean Squared Error (MSE) and Mean Absolute Error (MAE) are employed to measure the performance of the algorithms. Additionally, techniques such as Singular Value Decomposition (SVD) and k-Nearest Neighbors (k-NN) are explored to enhance the recommendation accuracy.

The results demonstrate the ability of collaborative filtering to provide high-quality recommendations, with item-based filtering showing slightly better performance in certain contexts.

Overall, this project underscores the potential of collaborative filtering as a robust foundation for movie recommendation systems, highlighting its capacity to adapt to individual user preferences and enhance the overall viewing experience.

Keywords: Recommendation System, Collaborative Filter, Feature Engineering, Sparse Matrix, Surprise Library Algorithms, Dataset.

List of Figures

Chapter 1

Introduction

1.1 About the project

A recommendation system or recommendation engine is a model used for information filtering where it tries to predict the preferences of a user and provide suggests based on these preferences. These systems have become increasingly popular nowadays and are widely used today in areas such as movies, music, books, videos, clothing, restaurants, food, places and other utilities. These systems collect information about a user's preferences and behaviour, and then use this information to improve their suggestions in the future.

Movies are a part and parcel of life. There are different types of movies like some for entertainment, some for educational purposes, some are animated movies for children, and some are horror movies or action films. Movies can be easily differentiated through their genres like comedy, thriller, animation, action etc. Other way to distinguish among movies can be either by releasing year, language, director etc. Watching movies online, there are a number of movies to search in our most liked movies. Movie Recommendation Systems helps us to search our preferred movies among all of these different types of movies and hence reduce the trouble of spending a lot of time searching our favourable movies. So, it requires that the movie recommendation system should be very reliable and should provide us with the recommendation of movies which are exactly same or most matched with our preferences.

A large number of companies are making use of recommendation systems to increase user interaction and enrich a user's shopping experience. Recommendation systems have several benefits, the most important being customer satisfaction and revenue. Movie Recommendation system is very powerful and important system. But, due to the problems associated with pure collaborative approach, movie recommendation systems also suffer with poor recommendation quality and scalability issues.

We can segregate the recommender structures in two general orders:
Given below are the two types of Movie Recommendation Systems----

1 Content-based sifting method

2 Collaborative sifting method

1.2 Proposed system

Collaborative filtering is a method used in recommendation systems to make automatic predictions about a user's interests by collecting preferences or taste information from many users. The assumption behind collaborative filtering is that if users agreed in the past, they will agree in the future.

There are two main types of collaborative filtering:

1. **User-based Collaborative Filtering**:
   - This method finds users who are similar to the active user (the user for whom the recommendation is being made) and uses their preferences to recommend items. For example, if two users have rated many items similarly, the items that one user likes may be recommended to the other.
2. **Item-based Collaborative Filtering**:
   - This method looks at the similarity between items rather than users. It recommends items that are similar to those that the user has liked or interacted with in the past. For instance, if a user likes a particular movie, the system will recommend other movies that are similar based on the preferences of other users.

3. Example:

	Movie A	Movie B	Movie C	Movie D
User 1	5	3	4	0
User 2	4	0	5	2
User 3	0	2	3	5
User 4	2	5	0	3

Fig 1.1:user-item matrix

Let's say we have the following user-item interaction matrix, where each cell represents the rating a user gave to a movie.

In user-based collaborative filtering, to recommend a movie to User 1, the system might look at the preferences of Users 2, 3, and 4 and find those most similar to User 1. If User 2 is found to be the most similar, the system would recommend movies that User 2 liked but User 1 hasn't watched yet.

In item-based collaborative filtering, if we want to recommend a movie similar to Movie A, the system would find movies that have similar ratings across users. If Movie C is found to be the most similar, it would be recommended to users who liked Movie A.

Advantages

- Can provide personalized recommendations without needing much domain knowledge.
- Works well in environments where user behavior is critical, such as online shopping or streaming services

Challenges

- **Cold Start Problem**: New users or items with little interaction data make it difficult to generate recommendations.
- **Scalability**: As the number of users and items grows, the computation of similarities and recommendations can become intensive.
- **Sparsity**: In many real-world datasets, most users interact with a small fraction of items, leading to a sparse interaction matrix, which can affect the quality of recommendations.

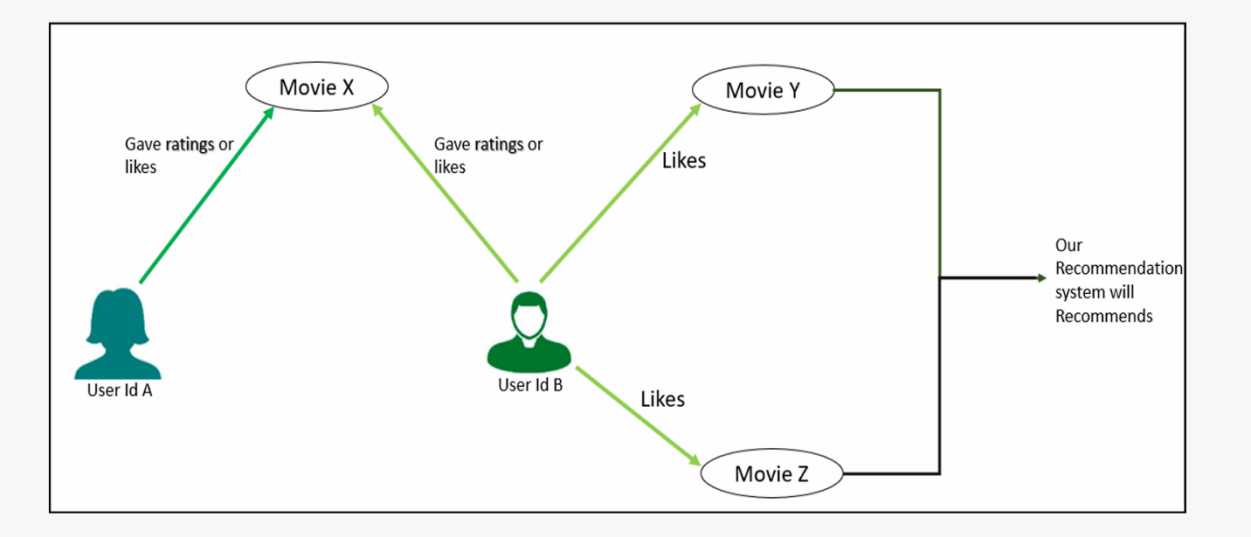


Fig 1.2: Structure of collaborative recommendation system

1.3 System Architecture

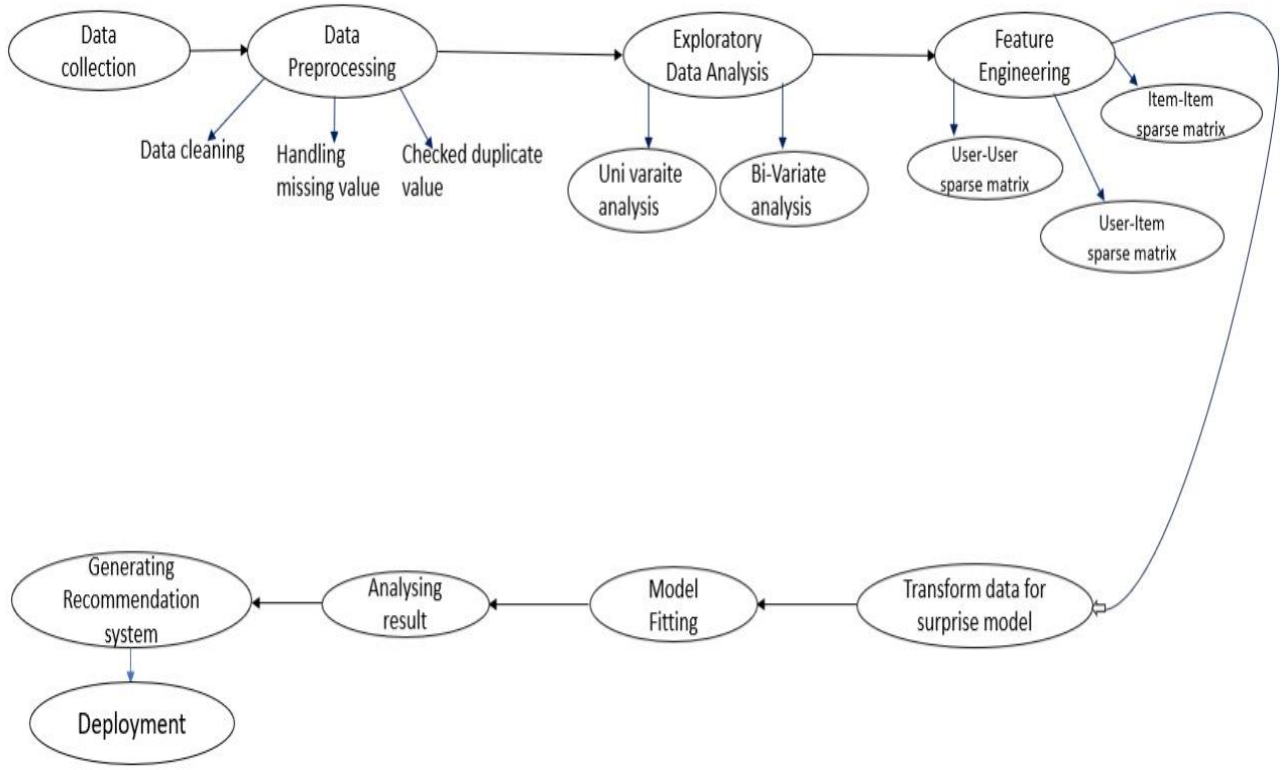


Fig 1.3: Architecture of the System

1.4 Objectives of the Study

The main Objective of this project is to find a small group of the best items and make recommendations from this smaller group and compare these recommendations with those which are made using the entire dataset, using collaborative filtering.

Major Objective:

1. To create a Collaborative Filtering based Movie Recommendation System.
2. To Minimize the difference between predicted and actual rating (RMSE and MAPE).

Chapter 2

Literature Review

Paper Name: *Machine Learning Model for Movie Recommendation System[1]*
Author: *M. Chenna Keshava*
Year of Published*: April 2020*

Observation: *The paper focuses on enhancing the CineMatch algorithm, which is designed for movie recommendation systems. The primary aim of recommendation systems, as highlighted, is to suggest relevant movies to users based on historical data and user behaviors. Specifically, the paper proposes improving the CineMatch algorithm by integrating advanced collaborative filtering techniques, aiming to enhance its recommendation accuracy by 10%.*

Paper Name: *Movie Recommendation Systems Based on Collaborative Filtering[2]*
Author: *Muhammed Sütcü*
Year of Published: *December 30, 2021*

Observation: *In the study "Movie Recommendation Systems Based on Collaborative Filtering: A Case Study on Netflix," the researchers explored the efficiency of various Collaborative Filtering methods for movie recommendations. They used the Netflix Prize dataset to conduct their analysis and focused on comparing four well-known Collaborative Filtering methods: Singular Value Decomposition (SVD), Singular Value Decomposition++ (SVD++), K-Nearest Neighbour (KNN), and Co-Clustering.*
The primary goal was to determine which method provided the most accurate predictions of user preferences. The accuracy of each method was measured using the Root Mean Square Error (RMSE), a standard metric for evaluating the performance of recommendation algorithms.

Paper Name: *Movie Recommender System using Content-Based and Collaborative Filtering[3]*

Author: *Mihir Bhoite, Prajwal Kanfade, Ashutosh Yadav, Madhuri Sahu, Achamma Thomas*

Year of Published*: 2023*

Observation*: CF recommends movies based on similar user preferences, but its accuracy varies. To enhance performance, the authors integrate advanced methods like the Surprise library and XGBoost regressor, boosting prediction accuracy beyond standard CF.*

CBF suggests movies similar to those previously liked by a user, aiming for more relevant recommendations. By merging CF and CBF, the system not only improves accuracy but also enhances user engagement and retention on platforms like Netflix.

Practically, they develop a user-friendly interface tailored for Netflix and other streaming platforms, ensuring their hybrid CF-CBF model seamlessly integrates into existing systems and enhances overall user satisfaction.

Paper Name: *Movie Recommender System Using Matrix Factorization[4]*

Author: *Roland Fiagbe*

Year of published: *2023*

Observation: *The paper "Movie Recommender System Using Matrix Factorization" explores enhancing movie recommendations through collaborative filtering, specifically using matrix factorization.*

In recommendation systems, navigating vast movie libraries is challenging. Collaborative filtering, unlike content-based methods, suggests movies based on shared user interests rather than individual preferences.

Matrix factorization simplifies complex user-movie interactions into latent factors, predicting how users might rate unseen movies accurately. Validation with the MovieLens dataset confirms the model's reliability, accurately predicting and recommending movies that align with user preferences.

This study highlights matrix factorization's role in refining movie recommendations, boosting user satisfaction by offering personalized suggestions on streaming platforms.

Chapter 3

Methodology

3.1 System Requirements:

Anaconda is a free and open-source distribution of the Python programming languages for scientific computing (data science, machine learning applications, large-scale data processing, predictive analytics, etc.), that aims to simplify package management system and deployment. Package versions are managed by the package management system conda. The anaconda distribution includes data-science packages suitable for Windows, Linux and MacOS.3.

3.2 Hardware Requirements:

A PC with Windows/Linux OS
Processor with 1.7-2.4gHz speed
Minimum of 8gb RAM
2gb Graphic card

3.3 Software Requirements:

Operating system: Windows 10
Tool: Anaconda Navigator – 64bit
Scripting Tool: Jupyter Notebook

3.4 Overview of the Platform:

Python

Python's simplicity, versatility, large standard library, and strong community support make it a powerful and widely adopted programming language across various domains and industries.

What can Python do?

Python can be used on a server to create web applications.
Python can be used alongside software to create workflows.
Python can connect to database systems. It can also read and modify files.
Python can be used to handle big data and perform complex mathematics.

3.5 Algorithms and Methods

3.5.1 Python libraries:

For the computation and analysis we need certain python libraries which are used to perform analytics. Packages such as Numpy, pandas, Matplotlib, Scikit-learn, Flask framework, Streamlit,etc are needed.

NumPy: Pandas is a Python library used for data manipulation and analysis. It offers easy-to-use data structures and functions for working with structured data like tables and time series. With Pandas, you can load, clean, transform, and analyze datasets quickly and efficiently.

Pandas: Pandas is a powerful data manipulation and analysis library for Python. We used it to read and process CSV files containing student data efficiently. Using Pandas, we could clean, transform, and prepare the data for subsequent analysis and visualization.

Matplotlib: Matplotlib is a widely-used data visualization library in Python. We employed Matplotlib to create graphs that visually represent students' performance over time. These visualizations were crucial for understanding trends and communicating our findings clearly.

Scikit-learn: Scikit-learn, often called sklearn, is a Python library for machine learning. It offers tools and algorithms to build and evaluate models. Sklearn is user-friendly, efficient, and covers tasks like classification, regression, clustering, and more.

Flask: Flask is a lightweight web framework for Python, used to build web applications and APIs. It's known for its simplicity and flexibility, making it ideal for small to medium-sized projects and rapid prototyping. With Flask, developers can quickly create web applications for tasks like building websites, handling HTTP requests, and serving data through APIs.

Streamlit: Streamlit is a popular open-source Python library used for building interactive web applications quickly and easily. It's particularly well-suited for data scientists and machine learning engineers who want to create web interfaces for their projects without dealing with the complexities of web development.

3.5.2 Data Collection:

We've obtained the dataset from GroupLens, available at the following link: https://grouplens.org/datasets/movielens/20m/.
This dataset, labeled ml-20m, documents user interactions with movies, including 5-star ratings and free-text tagging activity. It encompasses an extensive collection of 20,000,263 ratings and 465,564 tag applications across 27,278 movies. These interactions were contributed by 138,493 users over a period spanning from January 9, 1995, to March 31, 2015. It's worth noting that this dataset was compiled on October 17, 2016.
we can explore various avenues for analysis and derive valuable insights into user preferences, movie trends, and recommendation system performance.

3.5.3 Data Preprocessing:

Data preprocessing is a crucial step in data analysis and machine learning workflows. It involves cleaning and transforming raw data into a format that is suitable for analysis or modeling. Here's an overview of common data preprocessing techniques:

Data Handling: This involves loading the data into a suitable data structure, such as a pandas DataFrame in Python. It ensures that the data is organized and accessible for further processing.

Handling Missing Values: Missing values are quite common in real-world datasets and can adversely affect the analysis or modeling process. Data preprocessing techniques for handling missing values include:

Checking Duplicate Values: Duplicate values can skew analysis results or model performance.

3.5.4 EDA Analysis:

Exploratory Data Analysis (EDA) is a crucial step in understanding the underlying patterns, relationships, and insights within a dataset. It involves visually and statistically exploring the data to summarize its main characteristics, often using techniques such as summary statistics, data visualization, and hypothesis testing.
we utilize two primary types of EDA:

Univariate Analysis: Univariate analysis focuses on analyzing a single variable at a time. In the context of a movie recommendation system, univariate analysis might involve examining individual attributes of movies or user ratings without considering their relationships with other variables.

Bivariate Analysis: Bivariate analysis focuses on analyzing the relationship between two variables simultaneously. In the context of a movie recommendation system, bivariate analysis might involve exploring relationships between pairs of variables, such as movie genres and user ratings.

3.5.5 Feature Engineering:

Feature engineering in collaborative recommendation systems means creating and modifying features to make recommendation algorithms work better. In collaborative filtering, we mainly use data about how users interact with items, such as ratings or other feedback. Here's how we can use feature engineering in our system:

A **sparse matrix** is a data structure that represents a matrix in which most of the elements are zero. In contrast, a dense matrix stores every element, regardless of whether it is zero or non-zero. Sparse matrices are used in feature engineering.

Why Use Sparse Matrices?

Efficient Storage: Sparse matrices save space because they don't store all the zeros. This means they use less memory.
Faster Calculations: When doing math with sparse matrices, you can skip over the zeros. This makes calculations faster.
Handling Large Data: Sparse matrices are great for working with large datasets. They let you store and process big data without using too much memory.

What is Sparsity matrix?

A **sparsity matrix** is not a distinct type of matrix but rather a way to describe a matrix in terms of its sparsity characteristics. More commonly, the term used is **sparse matrix**. However, if we consider a sparsity matrix as an abstract concept, it would be a representation or measure indicating which elements in a matrix are zero and which are non-zero.
Here, three types of Sparse Matrix ,User-User Sparse Matrix, Item-Item Sparse Matrix,

and User-Item Sparse Matrix are utilized in feature engineering part:

User-User Sparse Matrix: A User-User Sparse Matrix represents the similarity between users based on their interactions with movies. Each row corresponds to a user, and each column represents another user. The values in the matrix indicate the degree of similarity between pairs of users.

Item-Item Sparse Matrix: An Item-Item Sparse Matrix represents the similarity between movies based on user interactions. Each row corresponds to a movie, and each column represents another movie. The values in the matrix indicate the degree of similarity between pairs of movies.

User-Item Sparse Matrix: A User-Item Sparse Matrix represents user interactions with movies. Each row corresponds to a user, and each column represents a movie. The values in the matrix indicate user ratings, views, or other forms of interaction with each movie.

3.5.6 Feature Extraction

Feature extraction is a subset of feature engineering. Feature extraction is the process of automatically identifying and extracting useful information or features from raw data. In the context of machine learning and data analysis, it typically refers to transforming raw input data into a format that is suitable for use in a model. Feature extraction is crucial for tasks such as classification, clustering, and recommendation systems.

In the context of a movie recommendation system, feature extraction involves converting movie-related data, such as movie titles, genres, ratings, and user preferences, into a format that can be used to build a recommendation model.

3.6 System Model

The system model for our collaborative recommendation system integrates various machine learning techniques to generate accurate and personalized recommendations.

3.6.1 XGBoost Regression

XGBoost (Extreme Gradient Boosting) is a powerful machine learning algorithm based on gradient boosting. It is known for its efficiency, flexibility, and high performance. In our recommendation system, XGBoost regression is used to predict user ratings for movies based on various features extracted from the data.
Collaborative filtering is a technique commonly used in recommendation systems, where the preferences of a user are inferred from preferences of similar users. In this example, let's assume we have a dataset of user ratings for various movies. Each row in the dataset represents a rating given by a user to a movie.
Here's a simplified example of the collaborative filtering formula for predicting the rating ru,i of user u for item i:

$$r_{u,i} = \frac{\sum_{v \in N(u)} \text{sim}(u,v) \times r_{v,i}}{\sum_{v \in N(u)} \text{sim}(u,v)}$$

Where:
N(u) is the neighbourhood of similar users to user u.
sim(u,v) is the similarity between users u and v.
ru,i is the rating of user v for item i.

For a given user and a movie they haven't rated, we predict the rating that the user would give to that movie. This prediction is based on the ratings of the selected neighborhood.

3.6.2 Surprise library

The Surprise library is a Python scikit specifically designed for building and evaluating recommendation systems. It provides various algorithms and tools for collaborative filtering, including matrix factorization-based algorithms, neighbourhood-based methods, and other techniques commonly used in recommendation systems.
In the Surprise library, there are several key models or algorithms commonly used for building recommendation systems. Some of these algorithms:

a) BaselineOnly: BaselineOnly is a simple yet effective model used in collaborative filtering recommendation systems. It works by estimating baseline ratings for users and items and then combining these baseline estimates with deviations from the baseline to predict ratings.
Here's how BaselineOnly works in collaborative recommendation systems:
Prediction Using BaselineOnly:
The prediction formula is:

The prediction formula is:

$$\hat{r}_{ui} = b_{\text{global}} + b_u + b_i$$

So, using the BaselineOnly approach, we predict that Alice would rate Movie2 as 4.17. This prediction combines the global average rating with Alice's tendency to rate slightly higher than average and the tendency of Movie2 to receive higher-than-average ratings.

Global Average Rating (global): Represents the overall tendency in the dataset.
User Bias (bu): Adjusts for the user's personal rating behavior.
Item Bias (bi): Adjusts for the item's general rating pattern.

b) KNN Baseline : KNNBaseline is a variation of the k-nearest neighbors (KNN) algorithm used in collaborative filtering recommendation systems. It combines the collaborative filtering approach of KNN with baseline predictions for users and items. For each user or item, KNBaseline identifies the k nearest neighbors based on similarity scores. The number k is a hyperparameter that determines the size of the neighbourhood.
Prediction:
To predict the rating for a user-item pair, KNNBaseline takes a weighted average of the ratings given by the k nearest neighbors.

The predicted rating $\hat{r}_{ui}$ for user u and item i is calculated as:

$$\hat{r}_{ui} = \frac{\sum_{v \in N(u,i)} \text{sim}(u,v) \times (r_{vi} - b_{vi})}{\sum_{v \in N(u,i)} \text{sim}(u,v)} + b_{ui}$$

where:

- $N(u,i)$ is the neighborhood of users who have rated item i.
- $\text{sim}(u,v)$ is the similarity between users u and v.
- r_{vi} is the rating of user v for item i.
- b_{vi} is the baseline prediction for user v and item i.

Differences between BaselineOnly and KNNBaseline:

BaselineOnly: Does not use neighbor. It makes predictions based only on overall average ratings and specific biases for users and items . Does not calculate similarities.

KNNBaseline: Looks at similar users or items (neighbor) to make predictions. It uses their ratings, weighted by how similar they are. Requires finding how similar users or items are.

c) SlopeOne: Slope One is a simple and efficient algorithm used for collaborative filtering in recommendation systems. It's based on the idea that the difference in ratings between two items is consistent across different users. This consistency allows us to make predictions about how a user might rate an item they haven't yet rated.

How Does Slope One Work?

Calculate Average Differences:
For each pair of items, calculate the average difference in ratings given by users who have rated both items. This average difference helps in understanding how one item is generally rated compared to another.

Make Predictions:
To predict a user's rating for an item they haven't rated, use their ratings for similar items and adjust based on the average differences calculated in the previous step.

To predict the rating $\hat{r}_{u,j}$ that user u would give to item j, based on the user's ratings for other items, we use the average differences. If the user has rated item i, the prediction is adjusted by the average difference between item i and item j:

$$\hat{r}_{u,j} = \frac{\sum_{i \in I_u} (r_{u,i} + \text{Diff}_{ij}) \times |U_{ij}|}{\sum_{i \in I_u} |U_{ij}|}$$

where:

- I_u is the set of items rated by user u.
- Diff_{ij} is the average difference between item i and item j.
- $|U_{ij}|$ is the number of users who have rated both items i and j.

d) SVD: Singular Value Decomposition (SVD) is a mathematical technique used in many applications, including collaborative filtering for recommendation systems. It is a type of matrix factorization that breaks down a large matrix into three simpler matrices, revealing underlying patterns in the data.

Make prediction:

- Reconstruct the approximate ratings matrix $\hat{R}$ using the reduced matrices:

$$\hat{R} = U_k \Sigma_k V_k^T$$

- The predicted rating for user u and item i can be found in $\hat{R}$.

Formula for Prediction:

To predict the rating $\hat{r}_{ui}$ for user u and item i:

$$\hat{r}_{ui} = \mathbf{u}_u^T \Sigma_k \mathbf{v}_i$$

where:

- $\mathbf{u}_u$ is the u-th row of U_k (user latent factors).
- $\mathbf{v}_i$ is the i-th column of V_k (item latent factors).
- Σ_k contains the top k singular values.

SVD is a matrix factorization method that decomposes a user-item ratings matrix into simpler matrices to uncover latent factors, enabling accurate predictions in collaborative filtering. It simplifies complex data and helps make personalized recommendations by understanding patterns in user behavior and item characteristics.

e) SVDPP: SVD++ (SVD Plus Plus) is an extension of the Singular Value Decomposition (SVD) algorithm, specifically designed to enhance collaborative filtering recommendation systems. It incorporates additional information about users' implicit feedback, such as how many items a user has rated, to improve recommendation accuracy.

$$\hat{r}_{ui} = \mu + b_u + b_i + \mathbf{q}_i^T \left(\mathbf{p}_u + \frac{1}{\sqrt{|N(u)|}} \sum_{j \in N(u)} \mathbf{y}_j \right)$$

Where:

- $\hat{r}_{ui}$ is the predicted rating for user u and item i.
- μ is the global average rating.
- b_u is the user bias term.
- b_i is the item bias term.
- $\mathbf{p}_u$ represents the latent factors for user u.
- $\mathbf{q}_i$ represents the latent factors for item i.
- $\mathbf{y}_j$ represents the implicit feedback factors for item j.
- $N(u)$ represents the set of items rated by user u.

Differences between **SVD** and **SVD++** :

SVD:

Basic matrix factorization.

No consideration of implicit feedback.
No user bias terms.
Predictions solely based on latent factors.
Limited handling of sparse data.

SVD++:

Extends SVD with implicit feedback.
Incorporates implicit feedback to improve accuracy.
Includes user bias terms for rating tendencies.

f) Evaluation Metrics:

Evaluation metrics are quantitative measures used to assess the performance, effectiveness, and quality of a system, model, or process. They are essential in various fields, including machine learning, data science, and information retrieval. Here are some key evaluation metrics commonly used across different domains:

Accuracy: Evaluate how accurately the system predicts user preferences and ratings for movies based on collaborative filtering algorithms.

RMSE (Root Mean Squared Error):

RMSE measures the average of the squared differences between predicted values and actual values, taking the square root of the result to obtain the scale of the original data.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}}$$

Where:
n is the total number of observations.
yi is the actual value of the target variable for observation
Ŷi is the predicted value of the target variable for observation

MAPE (Mean Absolute Percentage Error):

MAPE measures the average absolute percentage difference between predicted values and actual values, expressed as a percentage.

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100$$

Where:
n is the total number of observations.
yi is the actual value of the target variable for observation.
Ŷi is the predicted value of the target variable for observation.

Chapter 4

Implementation

1.Loading the data:

```
# Loading the dataset

movie_ratings = pd.read_csv("ratings.csv")
movies = pd.read_csv("movies.csv")
```

```
movies.head()
```

	movieId	title	genres
0	1	Toy Story	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji	Adventure|Children|Fantasy
2	3	Grumpier Old Men	Comedy|Romance
3	4	Waiting to Exhale	Comedy|Drama|Romance
4	5	Father of the Bride Part II	Comedy

```
Number of rows: 9742
Number of columns: 3
```

```
movie_ratings.head()
```

	userId	movieId	rating	timestamp
0	1	1	4.0	964982703
1	1	3	4.0	964981247
2	1	6	4.0	964982224
3	1	47	5.0	964983815
4	1	50	5.0	964982931

```
Number of rows: 100836
Number of columns: 4
```

2. Merging the movies and ratings data files:

```
# Merging the movies and ratings data files
movie_ratings = movie_ratings.merge(movies, how="left", on="movieId")
```

```
# Checking the features and no. of records in the merged dataset

print("The number of records are : ", movie_ratings.shape[0])
print("The number of features are : ", movie_ratings.shape[1])
print("The list of features is : ", movie_ratings.columns)
movie_ratings.head()
```

```
The number of records are :  100836
The number of features are :  6
The list of features is :  Index(['userId', 'movieId', 'rating', 'date', 'title', 'genres'], dtype='object')
```

	userId	movieId	rating	date	title	genres
0	429	507	5.0	1996-03-29	Aladdin	Adventure|Animation|Children|Comedy|Musical
1	429	204	4.0	1996-03-29	Forget Paris	Comedy|Romance
2	429	203	4.0	1996-03-29	French Kiss	Action|Comedy|Romance
3	429	201	2.0	1996-03-29	Exit to Eden	Comedy
4	429	194	3.0	1996-03-29	Drop Zone	Action|Thriller

3. Data Cleaning:

Checking Duplicate values and the Datatypes---

Checking null values-

```
# Checking for duplicates

print("No. of duplicates records in the dataset : ", movie_ratings.columns.duplicated().sum())
```

```
No. of duplicates records in the dataset :  0
```

```
# Checking the datatypes

movie_ratings.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100836 entries, 0 to 100835
Data columns (total 6 columns):
 #   Column   Non-Null Count   Dtype
---  ------   --------------   -----
 0   userId   100836 non-null  int64
 1   movieId  100836 non-null  int64
 2   rating   100836 non-null  float64
 3   date     100836 non-null  object
 4   title    100836 non-null  object
 5   genres   100836 non-null  object
dtypes: float64(1), int64(2), object(3)
memory usage: 4.6+ MB
```

```
# Checking the number of missing values in data

movie_ratings.isna().sum()
```

```
userId     0
movieId    0
rating     0
date       0
title      0
genres     0
dtype: int64
```

4. EDA (Exploratory Data Analysis):

- Univariate Analysis—

Here, checking each and every features from the dataset individually.

a) First we are checking no of unique “userId” and finding the top 5 “userId” in our dataset.

```
# Checking the feature "userID"

total_users = len(np.unique(movie_ratings["userId"]))
print("The count of unique userID in the dataset is : ", total_users)
print("The top 5 userID in the dataset are : \n", movie_ratings["userId"].value_counts()[:5])
```

```
The count of unique userID in the dataset is :  610
The top 5 userID in the dataset are :
 414    2698
599    2478
474    2108
448    1864
274    1346
Name: userId, dtype: int64
```

b) Similarly, Checking no of total unique “MovieId” and top 5 “Movieid” in the dataset

```
# Checking the feature "movieID"

total_movies = len(np.unique(movie_ratings["movieId"]))
print("The count of unique movieID in the dataset is : ", total_movies)
print("The top 5 movieID in the dataset are : \n", movie_ratings["movieId"].value_counts()[:5])
```

```
The count of unique movieID in the dataset is :  9724
The top 5 movieID in the dataset are :
 315     329
278     317
258     307
511     279
1940    278
Name: movieId, dtype: int64
```

c) Checking the distribution of the features “Rating” and “Genres” in our dataset.

```
# Checking the feature "rating"

import matplotlib.pyplot as plt
import seaborn as sns

sns.set(style="darkgrid")
fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)

sns.countplot(x="rating", data=movie_ratings, ax=axes)
axes.set_yticklabels([changingLabels(num) for num in axes.get_yticks()])
for p in axes.patches:
    axes.annotate('{}'.format(p.get_height()), (p.get_x()+0.2, p.get_height()+100))

plt.tick_params(labelsize=15)
plt.title("Distribution of Ratings in the dataset", fontsize=20)
plt.xlabel("Ratings", fontsize=10)
plt.ylabel("Counts(in Millions)", fontsize=10)
plt.show()
```

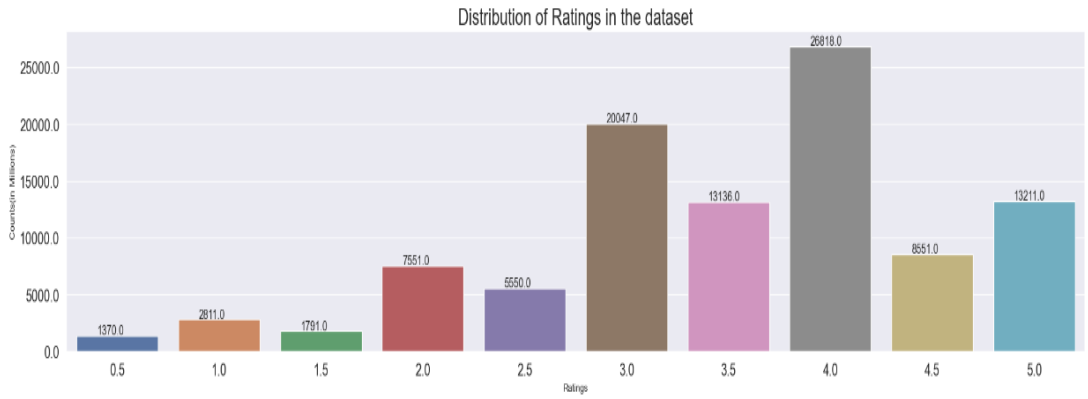


Fig:4.1

Observations—

1. From the above diagram we can see that 4.0 ratings are the highest no of Ratings present in our dataset.
2. Also, we can see that there are few no of movies present in the dataset who got ratings below 2.0.

And the for the Feature “genre”—

```
# Visualizing the feature "Genres"
genres_df = pd.DataFrame(list(unique_genres.items()))
genres_df.columns = ["Genre", "Count"]

sns.set(style="darkgrid")
fig, axes = plt.subplots(1, 1, figsize=(25, 8), sharey=True)

sns.barplot(y="Count", x="Genre", data=genres_df, ax=axes)
axes.set_yticklabels([changingLabels(num) for num in axes.get_yticks()])
for p in axes.patches:
    axes.annotate('{}'.format(int(p.get_height())), (p.get_x(), p.get_height()+100))

plt.tick_params(labelsize = 15)
plt.title("Distribution of Genres in the dataset", fontsize = 20)
plt.xlabel("Genres", fontsize = 15)
plt.xticks(rotation=60, fontsize=10)
plt.yticks(fontsize=10)
plt.ylabel("Counts (in Millions)", fontsize = 15)
plt.show()
```

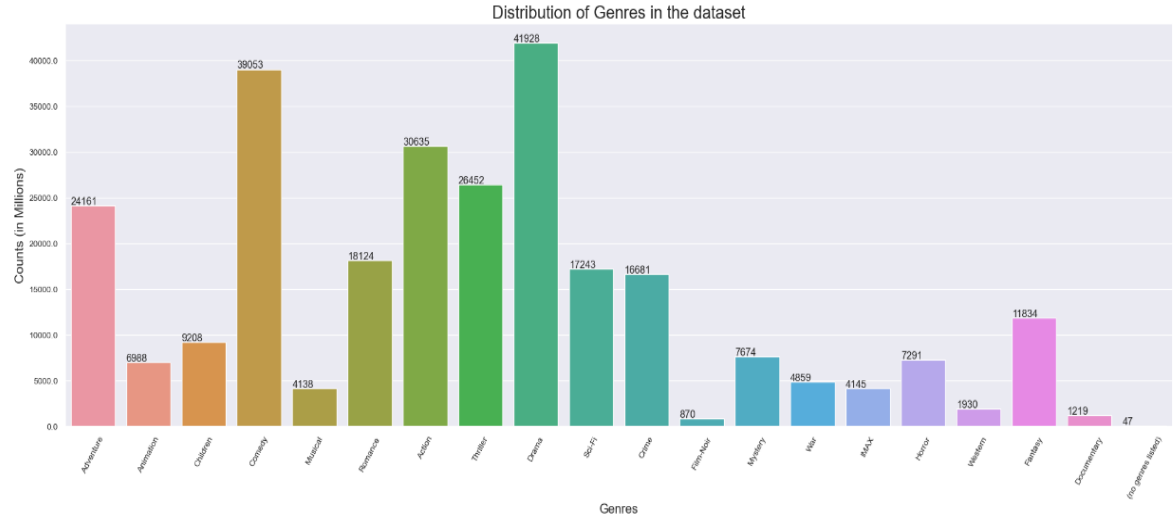


Fig:4.2

Observation---

From the above diagram we can see that the highest no of Genres present in our dataset are- Drama, Comedy, Action, Adventure & Thriller.

d) Checking “Date” & “Title” Feature.

Date-

```
# Checking the feature "date"

print("The count of unique date in the dataset is : ", movie_ratings["date"].nunique())
print("The first rating was given on : ", movie_ratings["date"].min())
print("The latest rating was given on : ", movie_ratings["date"].max())
print("The top 5 date in the dataset are : \n", movie_ratings["date"].value_counts()[:5])
```

```
The count of unique date in the dataset is :  4110
The first rating was given on :  1996-03-29
The latest rating was given on :  2018-09-24
The top 5 date in the dataset are :
 2017-06-26    1014
2007-05-14     878
2017-05-03     866
2000-08-08     709
2015-06-28     606
Name: date, dtype: int64
```

Title-

```
# Checking the feature "title"

movie_list = movie_ratings["title"].unique()
print("The count of unique title in the dataset is : ", movie_ratings["title"].nunique())
print("The top 5 title in the dataset are : \n", movie_ratings["title"].value_counts()[:5])
```

```
The count of unique title in the dataset is :  9719
The top 5 title in the dataset are :
 Forrest Gump (1994)                329
Shawshank Redemption, The (1994)    317
Pulp Fiction (1994)                 307
Silence of the Lambs, The (1991)    279
Matrix, The (1999)                  278
Name: title, dtype: int64
```

Now, before performing Bi-variate Analysis we have split our dataset into train and test dataset with the ratio of 80% for the train data and 20% for the test data and loaded both the datasets as Train_Data and Test_Data.

```
# Creating the train test set
if not os.path.isfile("TrainData.pkl"):
    print("Creating Train Data and saving it..")
    movie_ratings.iloc[:int(movie_ratings.shape[0] * 0.80)].to_pickle("TrainData.pkl")
    Train_Data = pd.read_pickle("TrainData.pkl")
    Train_Data.reset_index(drop=True, inplace=True)
else:
    print("Loading Train Data..")
    Train_Data = pd.read_pickle("TrainData.pkl")
    Train_Data.reset_index(drop=True, inplace=True)

if not os.path.isfile("TestData.pkl"):
    print("Creating Test Data and saving it..")
    movie_ratings.iloc[int(movie_ratings.shape[0] * 0.80):].to_pickle("TestData.pkl")
    Test_Data = pd.read_pickle("TestData.pkl")
    Test_Data.reset_index(drop=True, inplace=True)
else:
    print("Loading Test Data..")
    Test_Data = pd.read_pickle("TestData.pkl")
    Test_Data.reset_index(drop=True, inplace=True)
```

```
Loading Train Data..
Loading Test Data..
```

Now our train data is----

```
Train_Data.head()
```

	userId	movieId	rating	date	title	genres
0	429	507	5.0	1996-03-29	Aladdin	Adventure|Animation|Children|Comedy|Musical
1	429	204	4.0	1996-03-29	Forget Paris	Comedy|Romance
2	429	203	4.0	1996-03-29	French Kiss	Action|Comedy|Romance
3	429	201	2.0	1996-03-29	Exit to Eden	Comedy
4	429	194	3.0	1996-03-29	Drop Zone	Action|Thriller

And the test data ---

```
Test_Data.head()
```

	userId	movieId	rating	date	title	genres
0	495	8948	4.5	2016-03-22	Steve Jobs	Drama
1	495	9025	5.0	2016-03-22	Straight Outta Compton	Drama
2	495	8453	5.0	2016-03-22	Birdman: Or (The Unexpected Virtue of Ignorance)	Comedy|Drama
3	495	8312	4.5	2016-03-22	American Hustle	Crime|Drama
4	495	8427	3.5	2016-03-22	Godzilla	Action|Adventure|Sci-Fi|IMAX

- Bi-variate Analysis—

Analysing the number of ratings with date

In Bi-variate Analysis, we are analysing the distribution of the feature “Rating” with Date.

a) We are extracting the day of week from the “Date” feature when rating was provided and converting the numbers into string values.

```
# Extracting the day of week from the date when rating was provided

Train_Data["date"] = pd.to_datetime(Train_Data["date"], errors='coerce')
Train_Data["DayOfWeek"] = Train_Data["date"].dt.strftime('%A')
Train_Data["Weekday"] = Train_Data["date"].apply(lambda x : 1 if x.dayofweek > 5 else 0)
```

```
# Converting the numbers into string value

def ChangingLabelsInK(number):
    return str(int(number/10**3)) + "K"
```

b) Visualizing the count of total ratings made per month.

```
# Visualizing the count of total ratings made per month
sns.set(style="darkgrid")
fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)
axes = Train_Data.resample("m", on = "date")["rating"].count().plot()

axes.set_yticklabels([ChangingLabelsInK(num) for num in axes.get_yticks()])
axes.set_title("Count of Total Ratings per Month", fontsize = 20)
axes.set_xlabel("Date", fontsize = 15)
axes.set_ylabel("Number of Ratings (in Millions)", fontsize = 15)
plt.tick_params(labelsize = 15)
plt.show()
```

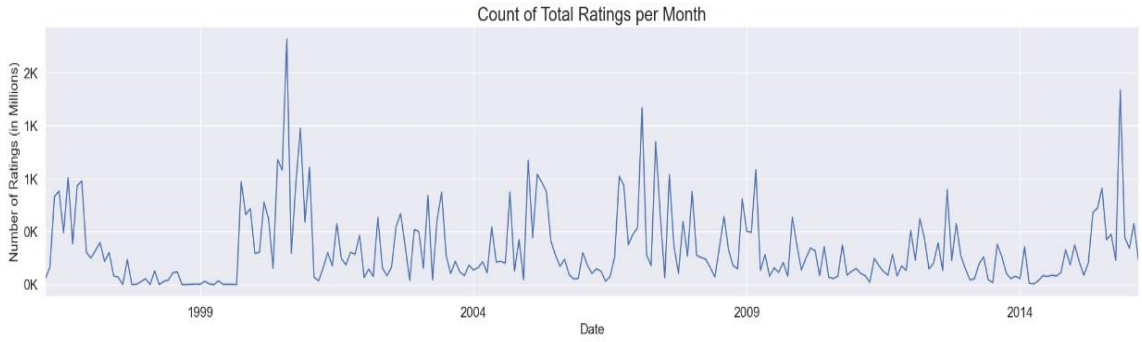


Fig:4.3

Observation--- The no. of ratings per month was very high in few of the months between 1999 to 2000.

c) Visualization of the count of ratings by weekday.

```
# Visualizing the count of ratings by weekday

sns.set(style="darkgrid")
fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)

sns.barplot(x="Weekday", y="rating" , data=Train_Data.groupby(by=["Weekday"], as_index=False)["rating"].count(), ax=axes)
for p in axes.patches:
    axes.annotate('{}'.format(int(p.get_height())), (p.get_x(), p.get_height()+100))

axes.set_yticklabels([changingLabels(num) for num in axes.get_yticks()])
plt.tick_params(labelsize = 15)
plt.title("Distribution of number of ratings by Weekday", fontsize = 20)
plt.xlabel("Weekday", fontsize = 15)
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)
plt.ylabel("Counts (in Millions)", fontsize = 15)
plt.show()
```

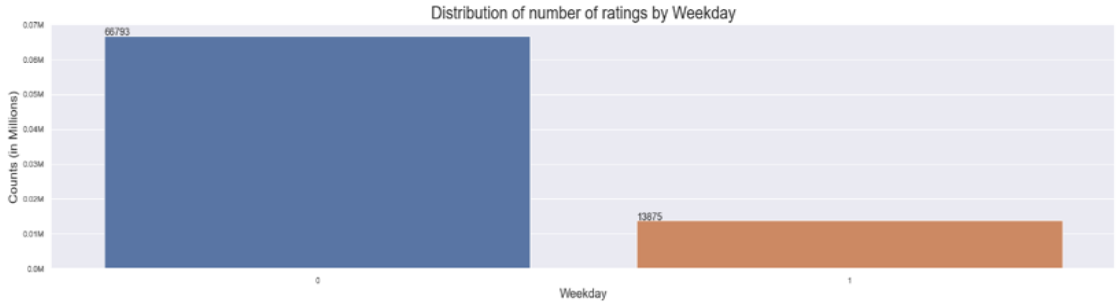


Fig:4.4

d) Visualizing the count of ratings by individual days of the week.

```
# Visualizing the count of ratings by individual days of the week

sns.set(style="darkgrid")
fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)

sns.barplot(x="DayOfWeek", y="rating" , data=Train_Data.groupby(by=["DayOfWeek"], as_index=False)["rating"].count(), ax=axes)
for p in axes.patches:
    axes.annotate('{}'.format(int(p.get_height())), (p.get_x(), p.get_height()+100))

axes.set_yticklabels([changingLabels(num) for num in axes.get_yticks()])
plt.tick_params(labelsize = 15)
plt.title("Distribution of number of ratings by individual days", fontsize = 20)
plt.xlabel("Days", fontsize = 15)
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)
plt.ylabel("Counts (in Millions)", fontsize = 15)
plt.show()
```

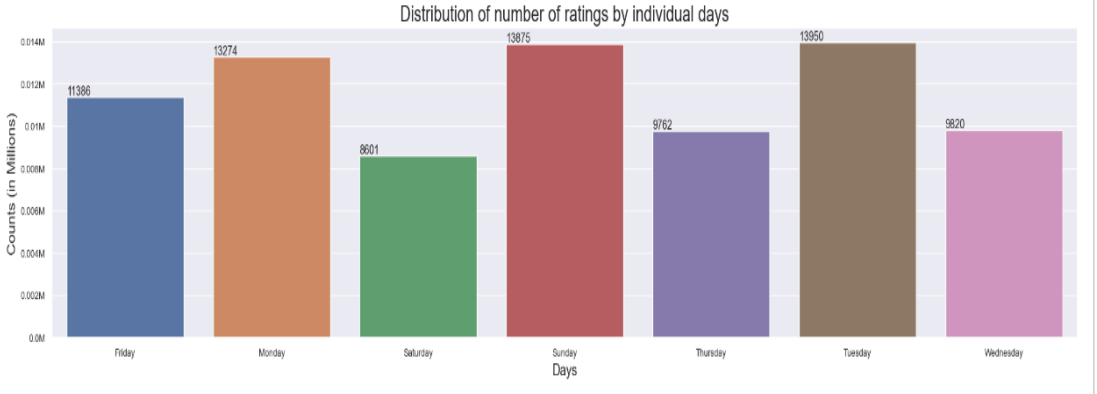


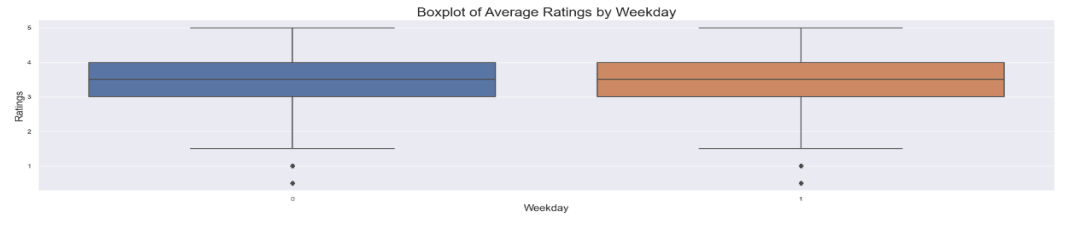
Fig:4.5

Observation--- In this dataset highest no of ratings are received on Sunday, Monday & Tuesday of the week.

Analysing the average ratings by date

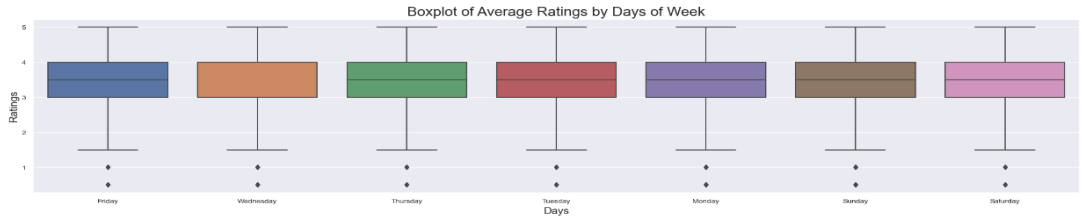
a) Visualizing the average ratings by weekday.

```
# Visualizing the average ratings by weekday

sns.set(style="darkgrid")
fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)

sns.boxplot(x="Weekday", y="rating" , data=Train_Data, ax=axes)

plt.tick_params(labelsize = 15)
plt.title("Boxplot of Average Ratings by Weekday", fontsize = 20)
plt.xlabel("Weekday", fontsize = 15)
plt.xticks(fontsize=10)
plt.ylabel("Ratings", fontsize = 15)
plt.yticks(fontsize=10)
plt.show()
```



b) Visualizing the average ratings by individual Days of the Week.

```
# Visualizing the average ratings by individual Days of the Week

sns.set(style="darkgrid")
fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)

sns.boxplot(x="DayOfWeek", y="rating", data=Train_Data, ax=axes)

plt.tick_params(labelsize = 15)
plt.title("Boxplot of Average Ratings by Days of Week", fontsize = 20)
plt.xlabel("Days", fontsize = 15)
plt.xticks(fontsize=10)
plt.ylabel("Ratings", fontsize = 15)
plt.yticks(fontsize=10)
plt.show()
```



Observation---

1) The average ratings given by the user does not seem to differ by weekday and weekends and also by individual days, they seem to be similar too.

Analysing the Ratings given by Users

a) Here we are calculating the total number of ratings given by individual users and also calculating no of ratings given by top 15 users.

```
# Calculating the number of ratings given by individual users

no_of_rated_movies_per_user = Train_Data.groupby(by=["userId"],as_index=False)["rating"].count().sort_values(by="rating", ascendi
no_of_rated_movies_per_user.reset_index(drop=True, inplace=True)
```

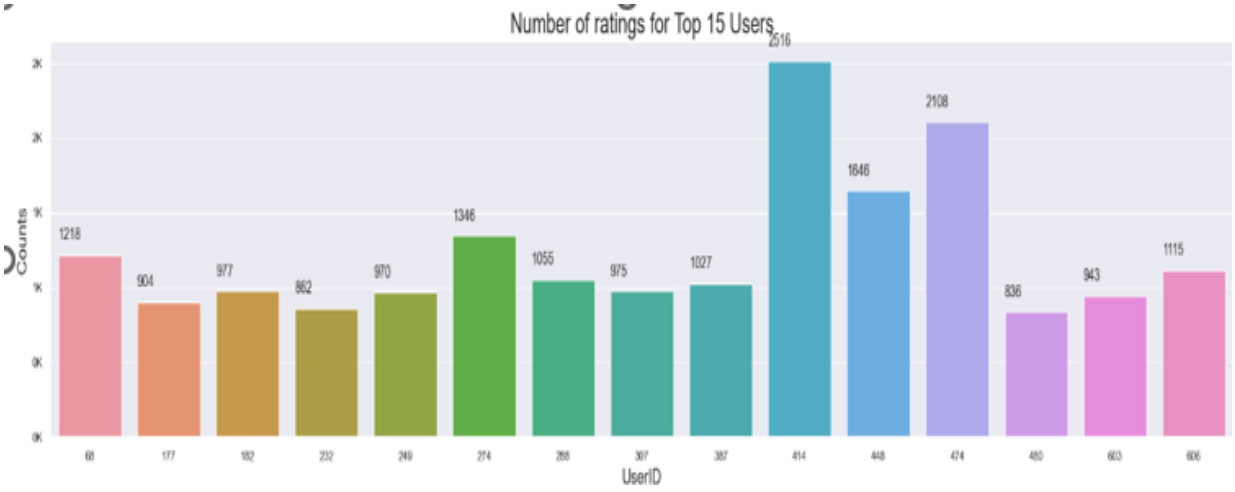


Fig:4.6

Observation--- Here among top 15 users, user Ids (414, 474, 445) gave highest ratings more than 1k.

b) Calculating average ratings given by individual users and also average ratings given by top 15 users.

```
# Calculating average ratings given by individual users
avg_ratings_per_user = Train_Data.groupby(by = ["userId"], as_index=False)["rating"].mean()
avg_ratings_per_user = avg_ratings_per_user.reset_index(drop=True)
avg_ratings_per_user = avg_ratings_per_user.merge(no_of_rated_movies_per_user[["userId", "rating"]], how="left", on="userId")
avg_ratings_per_user.rename(columns={"rating_x":"avg_rating", "rating_y": "num_of_rating"}, inplace=True)
avg_ratings_per_user = avg_ratings_per_user.sort_values("num_of_rating", ascending=False)
```

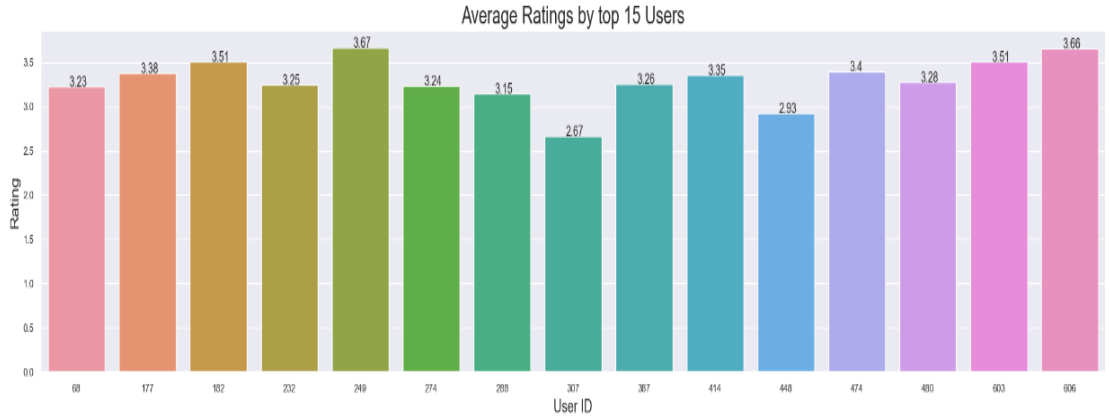


Fig:4.7

Observation---Here we can see that 249, 606, 603 & 182 users gave highest ratings compared to others.

Analysing the Ratings given to Movies

a) Here we are calculating no of ratings received for movies.

```
# Calculating count of ratings received for movies

no_of_ratings_per_movie = Train_Data.groupby(by = ["movieId", "title"],as_index=False)["rating"].count().sort_values(by=["rating"
no_of_ratings_per_movie = no_of_ratings_per_movie.reset_index(drop=True)
```

b) Visualizing the number of ratings for the movies.

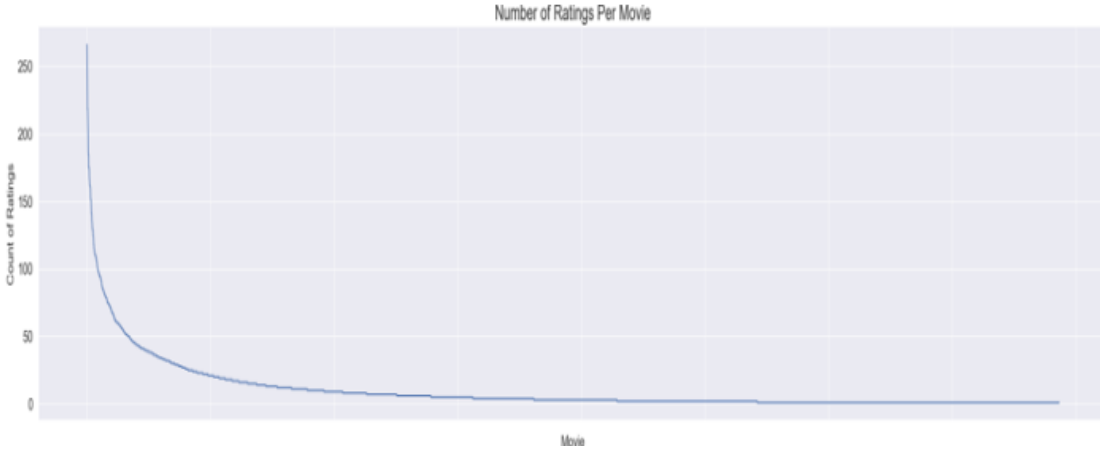


Fig:4.8

Observation---

1) It is quite clear that there are some movies which are very popular and were rated by many users as compared to other movies which has caused the plot to be skewed.

c) Visualizing top 15 movies heavily rated movies.

```
# Visualizing top 15 movies heavily rated movies.

sns.set(style="darkgrid")
fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)

sns.barplot(x="title", y="rating", data=no_of_ratings_per_movie[:15], ax=axes)
for p in axes.patches:
    axes.annotate('{}'.format(int(p.get_height())), (p.get_x(), p.get_height()+100))

axes.set_yticklabels([ChangingLabelsInK(num) for num in axes.get_yticks()])
plt.tick_params(labelsize = 15)
plt.title("Number of ratings for Top 15 Movies", fontsize = 20)
plt.xlabel("Movie", fontsize = 15)
plt.xticks(rotation=70, fontsize=10)
plt.ylabel("Counts", fontsize = 15)
plt.yticks(fontsize=10)
plt.show()
```

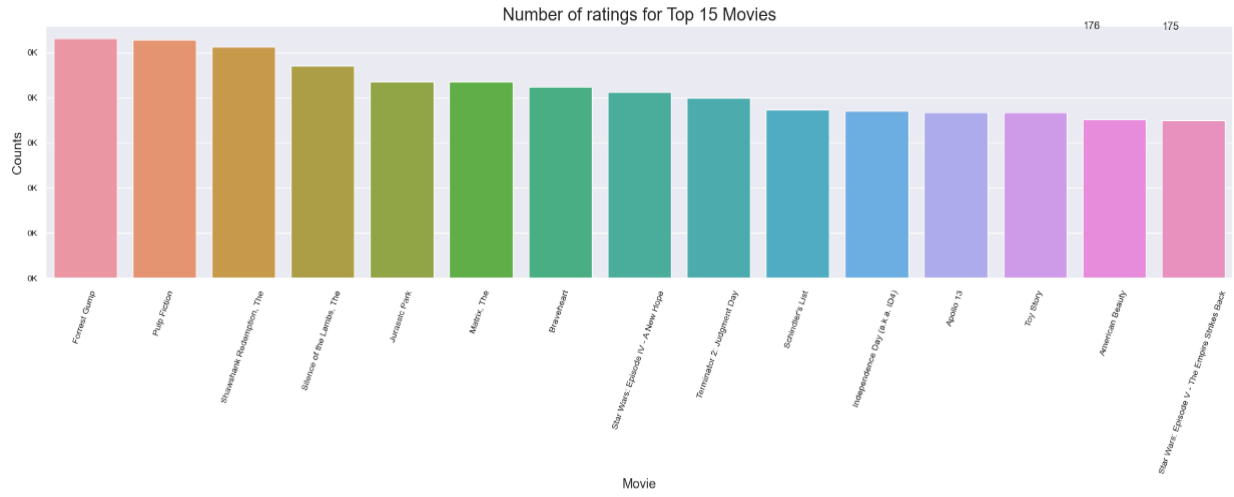


Fig:4.9

Observation---

Here we can see few movies got highest ratings among top 15 and those movies are “Forrest Gump” , ”Pulp Fiction” ,”The Shawshank Redemption .

d) Now Calculating average ratings for all the movies and Visualizing the average ratings for top 15 movies.

```
# Calculating average ratings for movies

avg_ratings_per_movie = Train_Data.groupby(by = ["movieId", "title"], as_index=False)["rating"].mean()
avg_ratings_per_movie = avg_ratings_per_movie.reset_index(drop=True)
avg_ratings_per_movie = avg_ratings_per_movie.merge(no_of_ratings_per_movie[["movieId", "rating"]], how="left", on="movieId")
avg_ratings_per_movie.rename(columns={"rating_x":"avg_rating", "rating_y": "num_of_rating"}, inplace=True)
avg_ratings_per_movie = avg_ratings_per_movie.sort_values("num_of_rating", ascending=False)
```

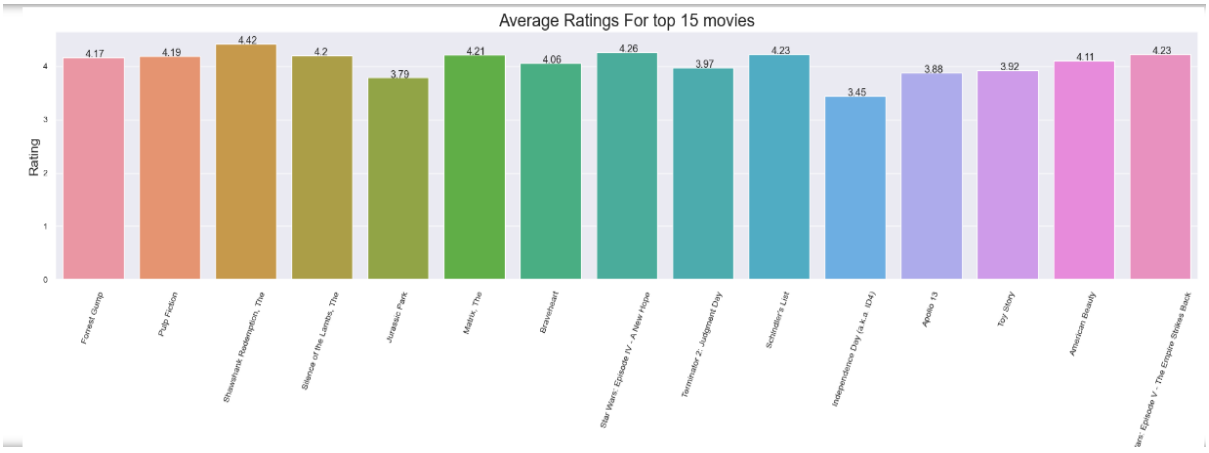


Fig:4.10

Observation---Here we can see that all 15 movies are getting above 3.0star ratings.

5. Feature Engineering:

Under Feature Engineering part we are creating 3 types of sparse matrix—

- User-Item Sparse Matrix
- Item-Item Sparse Matrix
- User-User Sparse Matrix

User-Item Sparse Matrix:

a) Take the Train_Data which already split before and create the User-Item Sparse Matrix for Train data.

```
from datetime import datetime
import os
from scipy import sparse

# Define startTime
startTime = datetime.now()

# Round the ratings to the nearest integer
Train_Data["rating"] = Train_Data["rating"].round().astype(int)

if os.path.isfile("TrainUISparseData.npz"):
    print("Sparse Data is already present on your disk. Loading Sparse Matrix...")
    TrainUISparseData = sparse.load_npz("TrainUISparseData.npz")
    print("Shape of Train Sparse Matrix: ", TrainUISparseData.shape)
else:
    print("Creating sparse data...")
    TrainUISparseData = sparse.csr_matrix((Train_Data.rating, (Train_Data.userId, Train_Data.movieId)))
    print("Creation done. Shape of sparse matrix: ", TrainUISparseData.shape)
    print("Saving it onto disk for further usage.")
    sparse.save_npz("TrainUISparseData.npz", TrainUISparseData)
    print("Done")

print("Time taken:", datetime.now() - startTime)
```

```
Sparse Data is already present on your disk. Loading Sparse Matrix...
Shape of Train Sparse Matrix:  (610, 9246)
Time taken: 0:00:00.017165
```

```
rows,cols = TrainUISparseData.shape
presentElements = TrainUISparseData.count_nonzero()

print("Sparsity Of Train matrix : {}% ".format((1-(presentElements/(rows*cols)))*100))
```

```
Sparsity Of Train matrix : 98.58669943227555%
```

Here the sparsity of Train matrix we got: 98.586%.

b) Take the Test_Data and create the User-Item sparse Matrix for Test Data.

```
# Creating/loading user-movie sparse matrix for test data

startTime = datetime.now()

print("Creating USER_ITEM sparse matrix for test Data..")

if os.path.isfile("TestUISparseData.npz"):
    print("Sparse Data is already present in your disk, no need to create further. Loading Sparse Matrix")
    TestUISparseData = sparse.load_npz("TestUISparseData.npz")
    print("Shape of Test Sparse Matrix : ", str(TestUISparseData.shape))
else:
    print("We are creating sparse data..")
    # Assuming Test_Data is defined
    TestUISparseData = sparse.csr_matrix((Test_Data.rating, (Test_Data.userId, Test_Data.movieId)))
    print("Creation done. Shape of sparse matrix : ", str(TestUISparseData.shape))
    print("Saving it into disk for further usage.")
    sparse.save_npz("TestUISparseData.npz", TestUISparseData)
    print("Done\n")

print("Time Taken : ", datetime.now() - startTime)
```

```
Creating USER_ITEM sparse matrix for test Data..
Sparse Data is already present in your disk, no need to create further. Loading Sparse Matrix
Shape of Test Sparse Matrix :  (611, 9743)
Time Taken :  0:00:00
```

```
rows,cols = TestUISparseData.shape
presentElements = TestUISparseData.count_nonzero()

print("Sparsity Of Test matrix : {}% ".format((1-(presentElements/(rows*cols)))*100))
```

```
Sparsity Of Test matrix : 99.66121129727952%
```

Here the sparsity of Test matrix we got: 99.66%

.

c) After that we hv created a function to calculate the Average rating for users or movies from User-Item sparse matrix.

```
# Function to Calculate Average rating for users or movies from User-movie sparse matrix

def getAverageRatings(sparseMatrix, if_user):

    #axis = 1 means rows and axis = 0 means columns
    ax = 1 if if_user else 0

    sumOfRatings = sparseMatrix.sum(axis = ax).A1
    noOfRatings = (sparseMatrix!=0).sum(axis = ax).A1
    rows, cols = sparseMatrix.shape
    averageRatings = {i: sumOfRatings[i]/noOfRatings[i] for i in range(rows if if_user else cols) if noOfRatings[i]!=0}

    return averageRatings
```

After calculating Average rating, find the total no of Users present & not present in the train data

```
AvgRatingUser = getAverageRatings(TrainUISparseData, True)
AvgRatingMovie = getAverageRatings(TrainUISparseData, False)
train_users = len(AvgRatingUser)
uncommonUsers = total_users - train_users

print("Total no. of Users : ", total_users)
print("No. of Users in Train data : ", train_users)
print("No. of Users not present in Train data : {}({}%)".format(uncommonUsers, np.round((uncommonUsers/total_users)*100), 2))
```

```
Total no. of Users :  610
No. of Users in Train data :  522
No. of Users not present in Train data : 88(14.0%)
```

Also, total no of Movies present & not present in the train data

```
train_movies = len(AvgRatingMovie)
uncommonMovies = total_movies - train_movies

print("Total no. of Movies : ", total_movies)
print("No. of Movies in Train data : ", train_movies)
print("No. of Movies not present in Train data = {}({}%)".format(uncommonMovies, np.round((uncommonMovies/total_movies)*100), 2))
```

```
Total no. of Movies :  9724
No. of Movies in Train data :  7815
No. of Movies not present in Train data = 1909(20.0%)
```

Item-Item Sparse Matrix:

a) Creating Item-Item Sparse matrix for the Train_Data

```
# Computing item-item  similarity matrix for the train data
start = datetime.now()

print("Movie-Movie Similarity file does not exist in your disk. Creating Movie-Movie Similarity Matrix...")
if not os.path.isfile("m_m_similarity.npz"):
    m_m_similarity = cosine_similarity(TrainUISparseData.T, dense_output=False)
    print("Dimension of Matrix : ", m_m_similarity.shape)
    print("Storing the Movie Similarity matrix on disk for further usage")
    sparse.save_npz("m_m_similarity.npz", m_m_similarity)
else:
    print("File exists in the disk. Loading the file...")
    m_m_similarity = sparse.load_npz("m_m_similarity.npz")
    print("Dimension of Matrix : ", m_m_similarity.shape)

print("The time taken to compute movie-movie similarity matrix is : ", datetime.now() - start)
```

```
Movie-Movie Similarity file does not exist in your disk. Creating Movie-Movie Similarity Matrix...
File exists in the disk. Loading the file...
Dimension of Matrix :  (9246, 9246)
The time taken to compute movie-movie similarity matrix is :  0:00:00.751897
```

b) Creating a function to take Movie Name and generate the top matched name and generate its N similar movies based on Item-Item or Movie-Movie Similarity.

```
def GetSimilarMoviesUsingMovieMovieSimilarity(movie_name, num_of_similar_movies):
    matches = process.extract(movie_name, movie_list_in_training["title"], scorer=fuzz.partial_ratio)
    if len(matches) == 0:
        return "No Match Found"
    movie_id = movie_list_in_training.iloc[matches[0][2]]["movieId"]
    similar_movie_id_list = np.argsort(-m_m_similarity[movie_id].toarray().ravel())[0:num_of_similar_movies+1]
    sm_df = movie_list_in_training[movie_list_in_training["movieId"].isin(similar_movie_id_list)]
    sm_df["order"] = sm_df.apply(lambda x: list(similar_movie_id_list).index(x["movieId"]), axis=1)

    return sm_df.sort_values("order")
```

```
# Picking random movie and checking it's top 10 most similar movies

GetSimilarMoviesUsingMovieMovieSimilarity("Get Shorty (1995)", 10)
```

	movieId	title	genres	order
84	21	Get Shorty	Comedy|Crime|Thriller	0
166	399	Fugitive, The	Thriller	1
202	384	Dave	Comedy|Romance	2
256	316	Four Weddings and a Funeral	Comedy|Romance	3
261	473	Sleepless in Seattle	Comedy|Drama|Romance	4
37	338	True Lies	Action|Adventure|Comedy|Romance|Thriller	5
251	413	In the Line of Fire	Action|Thriller	6
128	258	Pulp Fiction	Comedy|Crime|Drama|Thriller	7
57	261	Quiz Show	Drama	8
24	510	Batman	Action|Crime|Thriller	9
125	202	Ed Wood	Comedy|Drama	10

Here,we take the movie “Get Shorty(1995)” and searched top 10 movies similar to this movie.

User-User Sparse Matrix:

a) Checking the maximum userId in the dataset , which is 609.

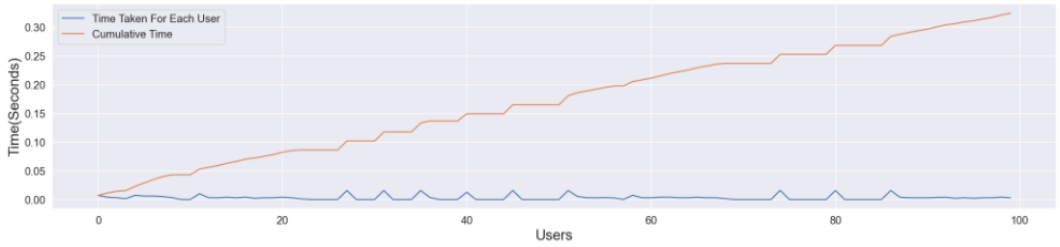
```
# checking maximum uder id

row_index, col_index = TrainUISparseData.nonzero()
unique_user_id = np.unique(row_index)

print("Max User id is :", np.max(unique_user_id))
```

```
Max User id is : 609
```

b) Creating sparse matrix for user-user similarity.

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime

def getUser_UserSimilarity(sparseMatrix, top=100):
    startTimestamp20 = datetime.now()

    row_index, col_index = sparseMatrix.nonzero()
    rows = np.unique(row_index)
    similarMatrix = np.zeros((sparseMatrix.shape[0], top))
    timeTaken = []
    howManyDone = 0
    for row in rows[:top]:
        howManyDone += 1
        startTimestamp = datetime.now().timestamp()
        sim = cosine_similarity(sparseMatrix.getrow(row), sparseMatrix).ravel()
        top_similar_indices = sim.argsort()[-top:]
        top_similar = sim[top_similar_indices]
        similarMatrix[row, :len(top_similar)] = top_similar
        timeforOne = datetime.now().timestamp() - startTimestamp
        timeTaken.append(timeforOne)
        if howManyDone % 20 == 0:
            print("Time elapsed for {} users = {}sec".format(howManyDone, (datetime.now() - startTimestamp20)))
    print("Average Time taken to compute similarity matrix for 1 user = " + str(sum(timeTaken) / len(timeTaken)) + "seconds")

    sns.set(style="darkgrid")
    fig = plt.figure(figsize=(25, 5))
    plt.plot(timeTaken, label='Time Taken For Each User')
    plt.plot(np.cumsum(timeTaken), label='Cumulative Time')
    plt.legend(loc='upper left', fontsize=15)
    plt.xlabel('Users', fontsize=20)
    plt.ylabel('Time(Seconds)', fontsize=20)
    plt.tick_params(labelsize=15)
    plt.show()

    return similarMatrix

# Assuming TrainUISparseData is defined elsewhere
simMatrix = getUser_UserSimilarity(TrainUISparseData, 100)
```

```
Time elapsed for 20 users = 0:00:00.080143sec
Time elapsed for 40 users = 0:00:00.138594sec
Time elapsed for 60 users = 0:00:00.210068sec
Time elapsed for 80 users = 0:00:00.254526sec
Time elapsed for 100 users = 0:00:00.326099sec
Average Time taken to compute similarity matrix for 1 user = 0.0032410883903503 42seconds
```



c) Calculating user-user similarity only for particular users in our sparse matrix and return user_ids.

```
# Calculating user-user similarity only for particular users in our sparse matrix and return user_ids

def Calculate_User_User_Similarity(sparseMatrix, user_id, num_of_similar_users=10):

    if user_id in unique_user_id:
        # Calculating the cosine similarity for user_id with all the "userId"
        sim = cosine_similarity(sparseMatrix.getrow(user_id), sparseMatrix).ravel()
        # Sorting the indexs(user_id) based on the similarity score for all the user ids
        top_similar_user_ids = sim.argsort()[::-1]
        # Sorted the similarity values
        top_similarity_values = sim[top_similar_user_ids]

    return top_similar_user_ids[1: num_of_similar_users+1]
```

d) Example, suppose we are getting top 5 users similar to userId: 1

```
# Getting top 5 users similar to userId: 1

similar_users_1 = Calculate_User_User_Similarity(TrainUISparseData, 1, 5)
similar_users_1
```

```
array([266, 313, 368,  57, 469], dtype=int64)
```

6. Feature Extraction:

a) Creating sample of Sparse Matrix.

```
#creating sample sparse matrix
import numpy as np
import scipy.sparse as sparse

def get_sample_sparse_matrix(sparseMatrix, n_users, n_movies, matrix_name):
    """
    Function to get a sample sparse matrix
    :param sparseMatrix: Original sparse matrix
    :param n_users: Number of users for sampling
    :param n_movies: Number of movies for sampling
    :param matrix_name: Name of the sample matrix file
    :return: Sampled sparse matrix
    """
    users, movies, ratings = sparse.find(sparseMatrix)
    uniq_users = np.unique(users)
    uniq_movies = np.unique(movies)

    if n_users > len(uniq_users) or n_movies > len(uniq_movies):
        replace_users = n_users > len(uniq_users)
        replace_movies = n_movies > len(uniq_movies)
        print("Sample size larger than population. Setting replace to", replace_users, "for users and", replace_movies, "for movi
    else:
        replace_users = False
        replace_movies = False

    userS = np.random.choice(uniq_users, n_users, replace=replace_users)
    movieS = np.random.choice(uniq_movies, n_movies, replace=replace_movies)
    mask = np.logical_and(np.isin(users, userS), np.isin(movies, movieS))
    sample_sparse_matrix = sparse.csr_matrix((ratings[mask], (users[mask], movies[mask])),
                                             shape=(max(userS)+1, max(movieS)+1))
    sparse.save_npz(matrix_name, sample_sparse_matrix)
    return sample_sparse_matrix
```

b) Creating the sample sparse matrix for train data set.

```
# creating the sample sparse matrix for train data set
if not os.path.isfile("TrainUISparseData_Sample.npz"):
    print("Sample sparse matrix is not present in the disk. We are creating it...")
    train_sample_sparse = get_sample_sparse_matrix(TrainUISparseData, 5000, 1000, "TrainUISparseData_Sample.npz")
else:
    print("File is already present in the disk. Loading the file...")
    train_sample_sparse = sparse.load_npz("TrainUISparseData_Sample.npz")
    print("Shape of Train Sample Sparse Matrix =", train_sample_sparse.shape)
```

```
File is already present in the disk. Loading the file...
Shape of Train Sample Sparse Matrix = (610, 9169)
```

c) Creating the sample sparse matrix for test data set.

```
# Creating Sample Sparse Matrix for Test Data

if not os.path.isfile(file_path + "/TestUISparseData_Sample.npz"):
    print("Sample sparse matrix is not present in the disk. We are creating it...")
    test_sample_sparse = get_sample_sparse_matrix(TestUISparseData, 2000, 200, "TestUISparseData_Sample.npz")
else:
    print("File is already present in the disk. Loading the file...")
    test_sample_sparse = sparse.load_npz(file_path + "/TestUISparseData_Sample.npz")
    print("Shape of Test Sample Sparse Matrix = " + str(test_sample_sparse.shape))
```

```
Sample sparse matrix is not present in the disk. We are creating it...
Sample size larger than population. Setting replace to True for users and False for movies
```

And checking the shape of Train Sparse Matrix and test Sparse Matrix:

```
# Checking the shape of Training and test data

print("Shape of Train Sparse Matrix : ", train_sample_sparse.shape)
print("Shape of Test Sparse Matrix : ", test_sample_sparse.shape)
```

```
Shape of Train Sparse Matrix :  (610, 9169)
Shape of Test Sparse Matrix :  (611, 9719)
```

d) Calculating GlobalAvgRating, GlobalAvgMovies, GlobalAvgUsers.

```
# Calculating GlobalAverageRating, globalAvgMovies, globalAvgUsers

globalAvgRating = np.round((train_sample_sparse.sum()/train_sample_sparse.count_nonzero()), 2)
globalAvgMovies = getAverageRatings(train_sample_sparse, False)
globalAvgUsers = getAverageRatings(train_sample_sparse, True)
print("Global average of all movies ratings in Train Set is : ", globalAvgRating)
print("No. of ratings in the train matrix is : ", train_sample_sparse.count_nonzero())
```

```
Global average of all movies ratings in Train Set is :  3.54
No. of ratings in the train matrix is :  9631
```

e) Creating a function to Extract Features and create Row using the sparse matrix.

```
def CreateFeaturesForTrainData(SampledSparseData, TrainSampledSparseData):

    startTime = datetime.now()

    # Extracting userId list, movieId list and Ratings
    sample_users, sample_movies, sample_ratings = sparse.find(SampledSparseData)

    print("No. of rows in the returned dataset : ", len(sample_ratings))

    count = 0
    data = []

    for user, movie, rating in zip(sample_users, sample_movies, sample_ratings):

        row = list()
```

f) Appending features "user Id" average, "movie Id" average & global average rating and also appending globalAvgUsers , globalAvgRating , GlobalAvgMmovie.

```
#--------Appending "user Id" average, "movie Id" average & global average rating-----------#
        row.append(user)
        row.append(movie)
        row.append(globalAvgRating)

#-------Appending globalAvgUsers , globalAvgRating , GlobalAvgMmovie-----------#
        try:
            row.append(globalAvgUsers[user])
        except (KeyError):
            global_average_rating = globalAvgRating
            row.append(global_average_rating)
        except:
            raise
        try:
            row.append(globalAvgMovies[movie])
        except (KeyError):
            global_average_rating = globalAvgRating
            row.append(global_average_rating)
        except:
            raise
```

g) Finding the Ratings given to "movie" by top 5 similar users with "user" , also Ratings given by "user" to top 5 similar movies with "movie" and finally appending rating of "user" & "movie".

```
#----Ratings given to "movie" by top 5 similar users with "user"----#
        try:
            similar_users = cosine_similarity(TrainSampledSparseData[user], TrainSampledSparseData).ravel()
            similar_users_indices = np.argsort(-similar_users)[1:]
            similar_users_ratings = TrainSampledSparseData[similar_users_indices, movie].toarray().ravel()
            top_similar_user_ratings = list(similar_users_ratings[similar_users_ratings != 0][:5])
            top_similar_user_ratings.extend([globalAvgMovies[movie]]*(5-len(top_similar_user_ratings)))
            #above line means that if top 5 ratings are not available then rest of the ratings will be filled by "movie" average
            #rating. Let say only 3 out of 5 ratings are available then rest 2 will be "movie" average rating.
            row.extend(top_similar_user_ratings)
        #########Cold Start Problem, for a new user or a new movie#########
        except (IndexError, KeyError):
            global_average_rating = [globalAvgRating]*5
            row.extend(global_average_rating)
        except:
            raise
```

```
#----Ratings given by "user" to top 5 similar movies with "movie"----#
        try:
            similar_movies = cosine_similarity(TrainSampledSparseData[:,movie].T, TrainSampledSparseData.T).ravel()
            similar_movies_indices = np.argsort(-similar_movies)[1:]
            similar_movies_ratings = TrainSampledSparseData[user, similar_movies_indices].toarray().ravel()
            top_similar_movie_ratings = list(similar_movies_ratings[similar_movies_ratings != 0][:5])
            top_similar_movie_ratings.extend([globalAvgUsers[user]]*(5-len(top_similar_movie_ratings)))
            #above line means that if top 5 ratings are not available then rest of the ratings will be filled by "user" average
            #rating. Let say only 3 out of 5 ratings are available then rest 2 will be "user" average rating.
            row.extend(top_similar_movie_ratings)
        ########Cold Start Problem, for a new user or a new movie#########
        except (IndexError, KeyError):
            global_average_rating = [globalAvgRating] * 5
            row.extend(global_average_rating)
        except:
            raise
```

```
#-----Appending rating of "user""movie"----------#
        row.append(rating)

        count += 1

        data.append(row)

        if count % 5000 == 0:
            print("Done for {}. Time elapsed: {}".format(count, (datetime.now() - startTime)))

    print("Total Time for {} rows = {}".format(len(data), (datetime.now() - startTime)))
    print("Completed..")
    return data
```

h) Using sampled train data and sampled test data creating Features for each row and saving it into the list.

```
# Using sampled train data, creating Features for each row and saving it into the list

data_rows = CreateFeaturesForTrainData(train_sample_sparse, train_sample_sparse)
```

```
No. of rows in the returned dataset :  9631
Done for 5000. Time elapsed: 0:00:14.628825
Total Time for 9631 rows = 0:00:28.449476
Completed..
```

```
# Using sampled test data, creating Features for each row and saving it into the list

test_data_rows = CreateFeaturesForTrainData(test_sample_sparse, train_sample_sparse)
```

```
No. of rows in the returned dataset :  678
Total Time for 678 rows = 0:00:01.939177
Completed..
```

i) Creating the pandas dataframe from the data rows extracted from the sparse matrix for train and test set and save the trainset as train_regression_data and the testset as test_regression_data.

```
# Creating the pandas dataframe from the data rows extracted from the sparse matrix for train and test set

names = ["User_ID", "Movie_ID", "Global_Average", "User_Average", "Movie_Average", "SUR1", "SUR2", "SUR3",
         "SUR4", "SUR5", "SMR1", "SMR2", "SMR3", "SMR4", "SMR5", "Rating"]
train_regression_data = pd.DataFrame(data_rows, columns=names)
test_regression_data = pd.DataFrame(test_data_rows, columns=names)
```

```
train_regression_data.to_csv("Training_Data_For_Regression.csv")
test_regression_data.to_csv("Testing_Data_For_Regression.csv")
```

Here train_regression_data and test_regression_data consists of total 16 columns:

train_regression_data:

```
# Checking the shape and first few records for train data

print("The shape of the dataframe is : ", train_regression_data.shape)
print("Number of missing Values : ", train_regression_data.isnull().sum().sum())
train_regression_data.head()
```

```
The shape of the dataframe is :  (9631, 16)
Number of missing Values :  0
```

	User_ID	Movie_ID	Global_Average	User_Average	Movie_Average	SUR1	SUR2	SUR3	SUR4	SUR5	SMR1	SMR2	SMR3	SMR4	SMR5	Rating
0	6	2	3.54	3.394737	3.404255	4.0	3.0	3.0	4.0	5.0	5.0	5.0	5.0	4.0	3.0	4
1	8	2	3.54	3.714286	3.404255	3.0	3.0	4.0	3.0	4.0	5.0	2.0	3.0	4.0	4.0	4
2	18	2	3.54	3.826087	3.404255	4.0	2.0	4.0	2.0	4.0	3.0	4.0	4.0	2.0	4.0	3
3	19	2	3.54	2.488636	3.404255	2.0	3.0	2.0	4.0	4.0	3.0	2.0	3.0	2.0	2.0	3
4	20	2	3.54	3.666667	3.404255	4.0	3.0	2.0	4.0	3.0	5.0	4.0	4.0	5.0	5.0	3

test_regression_data:

```
# Checking the shape and first few records for test data

print("The shape of the dataframe is : ", test_regression_data.shape)
print("Number of missing Values : ", test_regression_data.isnull().sum().sum())
test_regression_data.head()
```

```
The shape of the dataframe is :  (678, 16)
Number of missing Values :  0
```

	User_ID	Movie_ID	Global_Average	User_Average	Movie_Average	SUR1	SUR2	SUR3	SUR4	SUR5	SMR1	SMR2	SMR3	SMR4	SMR5	Rating
0	68	18	3.54	3.301887	3.54	3.54	3.54	3.54	3.54	3.54	4.00	3.00	4.00	4.00	3.00	2.0
1	380	18	3.54	3.540000	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	4.0
2	599	18	3.54	3.540000	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.0
3	599	66	3.54	3.540000	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.0
4	380	206	3.54	3.540000	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.0

7. Preparing data for the Surprise library Models:

a) Taking **train_regression_data** with only three columns **'User_ID', 'Movie_ID', 'Rating'** out of 16 columns and using Surprise library Data Structures to store train data as **"trainset" and** Creating tuple for test data and save as **"testset".**

```
train_regression_data[['User_ID', 'Movie_ID', 'Rating']].head(5)
```

	User_ID	Movie_ID	Rating
0	6	2	4
1	8	2	4
2	18	2	3
3	19	2	3
4	20	2	3

```
# Using Surprise library Data Structures to store train data

reader = Reader(rating_scale=(1, 5))
data = Dataset.load_from_df(train_regression_data[["User_ID", "Movie_ID", "Rating"]], reader)
trainset = data.build_full_trainset()
```

```
# Creating tuple for test set

testset = list(zip(test_regression_data["User_ID"].values, test_regression_data["Movie_ID"].values,
                   test_regression_data["Rating"].values))
```

b) Utilities to save the modelling results

```
# Utilities to save the modelling results
error_cols = ["Model", "Train RMSE", "Train MAPE", "Test RMSE", "Test MAPE"]
error_table = pd.DataFrame(columns=["Model", "RMSE_train", "MAPE_train", "RMSE_test", "MAPE_test"])
#error_table = pd.DataFrame(columns = error_cols)
model_train_evaluation = dict()
model_test_evaluation = dict()
```

c) Function to calculate **RMSE** and **MAPE** values and also creating a function to save modelling results in a table **"Error Table".**

```
# Function to calulate RMSE and MAPE values

def error_metrics(y_true, y_pred):
    rmse = np.sqrt(mean_squared_error(y_true, y_pred))
    mape = np.mean(abs((y_true - y_pred)/y_true))*100
    return rmse, mape
```

```
# Function to save modelling results in a table
def make_table(model_name, rmse_train, mape_train, rmse_test, mape_test):
    global error_table
    # Append data to error_table
    error_table = error_table._append(pd.DataFrame([[model_name, rmse_train,mape_train, rmse_test,mape_test]], columns=["Model",
                                                    "RMSE_train", "MAPE_train", "RMSE_test", "MAPE_test"]), ignore_index=True)
```

d) Function to plot feature importance for a model prediction.

```
# Function to plot feature importance for a model prediction

def plot_importance(model, clf):

    sns.set(style="darkgrid")
    fig = plt.figure(figsize = (25, 5))
    ax = fig.add_axes([0, 0, 1, 1])

    model.plot_importance(clf, ax = ax, height = 0.3)
    plt.xlabel("F Score", fontsize = 20)
    plt.ylabel("Features", fontsize = 20)
    plt.title("Feature Importance", fontsize = 20)
    plt.tick_params(labelsize = 15)
    plt.show()
```

e) Function to get predicted ratings and actual ratings. In surprise prediction of every data point is returned as dictionary and here in this dictionary, **"r_ui"** is a key for actual rating and **"est"** is a key for predicted rating.

```
def get_ratings(predictions):
    actual = np.array([pred.r_ui for pred in predictions])
    predicted = np.array([pred.est for pred in predictions])
    return actual, predicted

def get_error(predictions):
    actual, predicted = get_ratings(predictions)
    rmse = np.sqrt(mean_squared_error(actual, predicted))
    mape = np.mean(abs((actual - predicted)/actual))*100
    return rmse, mape
```

f) Running Surprise model and Evaluating model's performances on Train Data as well as on Test Data.

```
def run_surprise(algo, trainset, testset, model_name):

    startTime = datetime.now()

    train = dict()
    test = dict()

    algo.fit(trainset)
#--Evaluating model's performances on Train Data--#
    print("-"*50)
    print("TRAIN DATA")
    train_pred = algo.test(trainset.build_testset())
    train_actual, train_predicted = get_ratings(train_pred)
    train_rmse, train_mape = get_error(train_pred)
    print("RMSE = {}".format(train_rmse))
    print("MAPE = {}".format(train_mape))
    train = {"RMSE":train_rmse,"MAPE": train_mape,"Prediction":train_predicted}
```

```
#--Evaluating model's performances on Test Data--#
    print("-"*50)
    print("TEST DATA")
    test_pred = algo.test(testset)
    test_actual, test_predicted = get_ratings(test_pred)
    test_rmse, test_mape = get_error(test_pred)
    print("RMSE = {}".format(test_rmse))
    print("MAPE = {}".format(test_mape))
    test = {"RMSE": test_rmse, "MAPE": test_mape, "Prediction": test_predicted}
    print("-"*50)
    print("Time Taken = "+str(datetime.now() - startTime))
    make_table(model_name, train_rmse, train_mape, test_rmse, test_mape)
    return train, test
```

Train/test Splitting:

Taking **“train_regression_data”** and dropping 3 columns **'User_ID', 'Movie_ID', 'Rating'** which we used in data preparing for Surprise Library algorithms. Now split the **“train_regression_data”** with rest of the 13 features or columns into **x_train, x_test, y_train, y_test.**

```
# Creating the train-test X and y variables for the ML algos

x_train = train_regression_data.drop(["User_ID", "Movie_ID", "Rating"], axis = 1)
x_test = test_regression_data.drop(["User_ID", "Movie_ID", "Rating"], axis = 1)
y_train = train_regression_data["Rating"]
y_test = test_regression_data["Rating"]
```

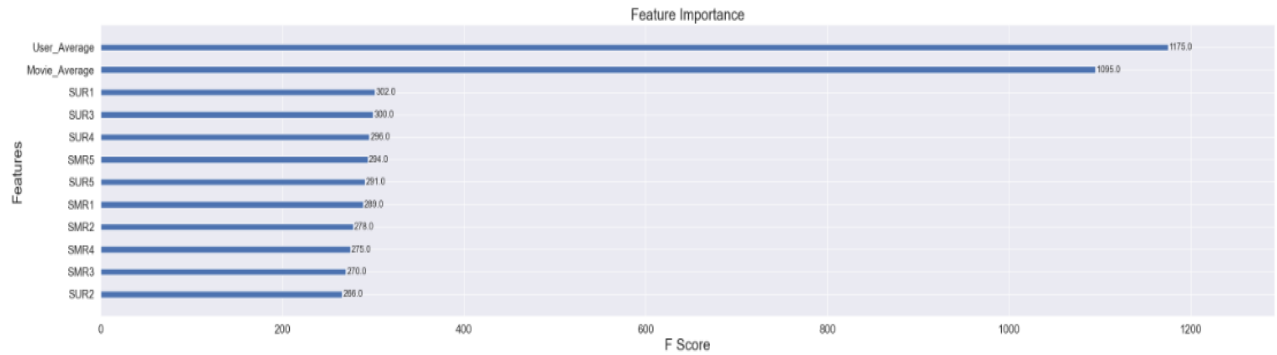
8. Model Fitting:

XGBoost Regression:

a) Training the XGboost Regression Model on with the 13 features.

```
train_result, test_result = train_test_xgboost(x_train, x_test, y_train, y_test, "XGBoost_13")

model_train_evaluation["XGBoost_13"] = train_result
model_test_evaluation["XGBoost_13"] = test_result
```

```
-------------------------------------------------
TRAIN DATA
RMSE : 0.5022152039678237
MAPE : 12.95033060455976
-------------------------------------------------
TEST DATA
RMSE : 1.254294089019945
MAPE : 51.87903766915186
-------------------------------------------------
Time Taken :  0:00:00.188362
```



Observations:

1. "User_Average" by far seems to be the most important feature for rating prediction.
2. "Movie_Average" is the second most important feature to predict the ratings.
3. The top 5 Similar User ratings and top 5 Similar Movie Ratings doesn't seem to be the effective features.

BaselineOnly:

a) Applying BaselineOnly from the surprise library to predict the ratings.

```
# Applying BaselineOnly from the surprise library to predict the ratings

bsl_options = {"method":"sgd", "learning_rate":0.01, "n_epochs":25}

algo = BaselineOnly(bsl_options=bsl_options)

train_result, test_result = run_surprise(algo, trainset, testset, "BaselineOnly")

model_train_evaluation["BaselineOnly"] = train_result
model_test_evaluation["BaselineOnly"] = test_result
```

```
Estimating biases using sgd...
-------------------------------------------------
TRAIN DATA
RMSE = 0.7746663613975766
MAPE = 22.90206660554954
-------------------------------------------------
TEST DATA
RMSE = 1.0640940272006976
MAPE = 41.91607525843126
-------------------------------------------------
Time Taken = 0:00:00.090753
```

b) Adding predicted ratings from Surprise BaselineOnly model to our **"train_regression_data"** and **"test_regression_data"** dataframe and fitting XGBoost with new BaselineOnly features.

"train_regression_data":

```
train_regression_data.head()
```

User_ID	Movie_ID	Global_Average	User_Average	Movie_Average	SUR1	SUR2	SUR3	SUR4	SUR5	SMR1	SMR2	SMR3	SMR4	SMR5	Rating	BaselineOnly
6	2	3.54	3.394737	3.404255	4.0	3.0	3.0	4.0	5.0	5.0	5.0	5.0	4.0	3.0	4	3.462976
8	2	3.54	3.714286	3.404255	3.0	3.0	4.0	3.0	4.0	5.0	2.0	3.0	4.0	4.0	4	3.150840

"test_regression_data":

```
test_regression_data.head()
```

User_ID	Movie_ID	Global_Average	User_Average	Movie_Average	SUR1	SUR2	SUR3	SUR4	SUR5	SMR1	SMR2	SMR3	SMR4	SMR5	Rating	BaselineOnly
68	18	3.54	3.301887	3.54	3.54	3.54	3.54	3.54	3.54	4.00	3.00	4.00	4.00	3.00	2.0	3.310096
380	18	3.54	3.540000	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	4.0	3.544388
599	18	3.54	3.540000	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.0	3.544388

RMSE & MAPE value after fitting XGBoost:

```
-------------------------------------------------
TRAIN DATA
RMSE : 0.4790442133142484
MAPE : 12.404605706846365
-------------------------------------------------
TEST DATA
RMSE : 1.1892565091695335
MAPE : 48.54017417227814
-------------------------------------------------
Time Taken :  0:00:00.173240
```

KNN-Baseline:

a) Finding the suitable parameter for Surprise KNN-Baseline with User-User Similarity.

```
# Finding the suitable parameter for Surprise KNN-Baseline with User-User Similarity

param_grid  = {'sim_options':{'name': ["pearson_baseline"], "user_based": [True],
                              "min_support": [2], "shrinkage": [60, 80, 80, 140]}, 'k': [5, 20, 40, 80]}
gs = GridSearchCV(KNNBaseline, param_grid, measures=['rmse', 'mae'], cv=3)
gs.fit(data)

# best RMSE score
print(gs.best_score['rmse'])
# combination of parameters that gave the best RMSE score
print(gs.best_params['rmse'])
```

Applying the KNN-Baseline with the searched parameters.

```
# Applying the KNN-Baseline with the searched parameters

sim_options = {'name':'pearson_baseline', 'user_based':True, 'min_support':2, 'shrinkage':gs.best_params['rmse']
               ['sim_options']['shrinkage']}

bsl_options = {'method': 'sgd'}

algo = KNNBaseline(k = gs.best_params['rmse']['k'], sim_options = sim_options, bsl_options=bsl_options)

train_result, test_result = run_surprise(algo, trainset, testset, "KNNBaseline_User")

model_train_evaluation["KNNBaseline_User"] = train_result
model_test_evaluation["KNNBaseline_User"] = test_result
```

```
Estimating biases using sgd...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
-------------------------------------------------
TRAIN DATA
RMSE = 0.19075888578975594
MAPE = 4.398733354382546
-------------------------------------------------
TEST DATA
RMSE = 1.0634244356668774
MAPE = 41.92563204317548
-------------------------------------------------
Time Taken = 0:00:00.658978
```

b) finding best parameters for Surprise KNN-Baseline with Item-Item Similarity.

```
# Similarly finding best parameters for Surprise KNN-Baseline with Item-Item Similarity

param_grid  = {'sim_options':{'name': ["pearson_baseline"], "user_based": [False],
                              "min_support": [2], "shrinkage": [60, 80, 80, 140]}, 'k': [5, 20, 40, 80]}

gs = GridSearchCV(KNNBaseline, param_grid, measures=['rmse', 'mae'], cv=3)

gs.fit(data)

# best RMSE score
print(gs.best_score['rmse'])

# combination of parameters that gave the best RMSE score
print(gs.best_params['rmse'])
```

Applying KNN-Baseline with best parameters searched.

```
# Applying KNN-Baseline with best parameters searched

sim_options = {'name':'pearson_baseline', 'user_based':False, 'min_support':2,
               'shrinkage':gs.best_params['rmse']['sim_options']['shrinkage']}

bsl_options = {'method': 'sgd'}

algo = KNNBaseline(k = gs.best_params['rmse']['k'], sim_options = sim_options, bsl_options=bsl_options)

train_result, test_result = run_surprise(algo, trainset, testset, "KNNBaseline_Item")

model_train_evaluation["KNNBaseline_Item"] = train_result
model_test_evaluation["KNNBaseline_Item"] = test_result
```

```
Estimating biases using sgd...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix.
-------------------------------------------------
TRAIN DATA
RMSE = 0.19685500976057593
MAPE = 4.723898587978162
-------------------------------------------------
TEST DATA
RMSE = 1.0632504881878315
MAPE = 41.89906276147251
-------------------------------------------------
Time Taken = 0:00:00.719869
```

c) Adding the KNNBaseline features to the **"train_regression_data"** and **"test_regression_data"** dataframe and fitting Xgboost with the KNN-Baseline newly added features.

"train_regression_data":

```
train_regression_data.head()
```

er_Average	Movie_Average	SUR1	SUR2	SUR3	SUR4	SUR5	SMR1	SMR2	SMR3	SMR4	SMR5	Rating	BaselineOnly	KNNBaseline_User	KNNBaseline_Item
3.394737	3.404255	4.0	3.0	3.0	4.0	5.0	5.0	5.0	5.0	4.0	3.0	4	3.462976	3.808953	4.031026
3.714286	3.404255	3.0	3.0	4.0	3.0	4.0	5.0	2.0	3.0	4.0	4.0	4	3.150840	3.821575	3.852293

"test_regression_data":

```
test_regression_data.head()
```

er_Average	Movie_Average	SUR1	SUR2	SUR3	SUR4	SUR5	SMR1	SMR2	SMR3	SMR4	SMR5	Rating	BaselineOnly	KNNBaseline_User	KNNBaseline_Item
3.301887	3.54	3.54	3.54	3.54	3.54	3.54	4.00	3.00	4.00	4.00	3.00	2.0	3.310096	3.296834	3.296834

RMSE & MAPE value after fitting XGBoost:

```
-------------------------------------------------
TRAIN DATA
RMSE : 0.46261696664297497
MAPE : 11.99007174978044
-------------------------------------------------
TEST DATA
RMSE : 1.0947622423861443
MAPE : 43.473622387489385
-------------------------------------------------
Time Taken :  0:00:00.205857
```

SlopeOne:

a) Appling the SlopeOne algorithm from the Surprise library.

```
# Appling the SlopeOne algorithm from the Surprise library

so = SlopeOne()

train_result, test_result = run_surprise(so, trainset, testset, "SlopeOne")

model_train_evaluation["SlopeOne"] = train_result
model_test_evaluation["SlopeOne"] = test_result
```

```
--------------------------------------------------
TRAIN DATA
RMSE = 0.5861019761008184
MAPE = 15.68407837857041
--------------------------------------------------
TEST DATA
RMSE = 1.0870784175804353
MAPE = 43.25843002990793
--------------------------------------------------
Time Taken = 0:00:00.442591
```

And Adding the SlopeOne predictions to the **"train_regression_data"** and **"test_regression_data"** dataframe.

```
# Adding the SlopOne predictions to the train and test datasets

train_regression_data["SlopeOne"] = model_train_evaluation["SlopeOne"]["Prediction"]
train_regression_data["SlopeOne"] = model_train_evaluation["SlopeOne"]["Prediction"]

test_regression_data["SlopeOne"] = model_test_evaluation["SlopeOne"]["Prediction"]
test_regression_data["SlopeOne"] = model_test_evaluation["SlopeOne"]["Prediction"]
```

SVD:

a) search best parameter for SVD.

```
# search best parameter for SVD
param_grid  = {'n_factors': [5,7,10,15,20,25,35,50,70,90]}

gs = GridSearchCV(SVD, param_grid, measures=['rmse', 'mae'], cv=3)
gs.fit(data)

# best RMSE score
print(gs.best_score['rmse'])

# combination of parameters that gave the best RMSE score
print(gs.best_params['rmse'])
```

```
0.871732086986234
{'n_factors': 7}
```

Applying SVD with best parameters.

```
# Applying SVD with best parameters

algo = SVD(n_factors = gs.best_params['rmse']['n_factors'], biased=True, verbose=True)

train_result, test_result = run_surprise(algo, trainset, testset, "SVD")

model_train_evaluation["SVD"] = train_result
model_test_evaluation["SVD"] = test_result
```

```
TRAIN DATA
RMSE = 0.7888023697619427
MAPE = 23.734543211794065
--------------------------------------------------
TEST DATA
RMSE = 1.0631739047499138
MAPE = 41.881903808854744
--------------------------------------------------
Time Taken = 0:00:00.089597
```

SVDPP:

a) search best parameter for SVDPP

```
# search best parameter for SVDPP


param_grid = {'n_factors': [10, 30, 50, 80, 100], 'lr_all': [0.002, 0.006, 0.018, 0.054, 0.10]}

gs = GridSearchCV(SVDpp, param_grid, measures=['rmse', 'mae'], cv=3)
gs.fit(data)

# best RMSE score
print(gs.best_score['rmse'])

# combination of parameters that gave the best RMSE score
print(gs.best_params['rmse'])
```

```
0.8732253924841299
{'n_factors': 10, 'lr_all': 0.006}
```

Applying SVDPP with best parameters.

```
algo = SVDpp(n_factors = gs.best_params['rmse']['n_factors'], lr_all = gs.best_params['rmse']["lr_all"], verbose=True)

train_result, test_result = run_surprise(algo, trainset, testset, "SVDpp")

model_train_evaluation["SVDpp"] = train_result
model_test_evaluation["SVDpp"] = test_result
```

```
-------------------------------------------------
TRAIN DATA
RMSE = 0.7475182208162305
MAPE = 22.24700473328278
-------------------------------------------------
TEST DATA
RMSE = 1.0628065878075632
MAPE = 41.88417820184062
-------------------------------------------------
Time Taken = 0:00:01.230600
```

Now, adding SVD and SVDPP with the **"train_regression_data"** and **"test_regression_data"** dataframe.

"train_regression_data":

```
train_regression_data.head()
```

R1	SUR2	SUR3	SUR4	SUR5	SMR1	SMR2	SMR3	SMR4	SMR5	Rating	BaselineOnly	KNNBaseline_User	KNNBaseline_Item	SlopeOne	SVD	SVDpp
4.0	3.0	3.0	4.0	5.0	5.0	5.0	5.0	4.0	3.0	4	3.462976	3.808953	4.031026	3.688111	3.443984	3.432187
3.0	3.0	4.0	3.0	4.0	5.0	2.0	3.0	4.0	4.0	4	3.150840	3.821575	3.852293	3.507498	3.070909	3.199458
4.0	2.0	4.0	2.0	4.0	3.0	4.0	4.0	2.0	4.0	3	3.329387	3.898359	4.034416	3.588252	3.360160	3.373855
2.0	3.0	2.0	4.0	4.0	3.0	2.0	3.0	2.0	2.0	3	4.054601	3.775771	3.784927	4.003358	3.928160	3.937541

"test_regression_data":

```
test_regression_data.head()
```

IR1	SUR2	SUR3	SUR4	SUR5	SMR1	SMR2	SMR3	SMR4	SMR5	Rating	BaselineOnly	KNNBaseline_User	KNNBaseline_Item	SlopeOne	SVD	SVDpp
3.54	3.54	3.54	3.54	3.54	4.00	3.00	4.00	4.00	3.00	2.0	3.310096	3.296834	3.296834	3.544388	3.297374	3.292981
3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	4.0	3.544388	3.544388	3.544388	3.544388	3.544388	3.544388
3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.0	3.544388	3.544388	3.544388	3.544388	3.544388	3.544388
3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.54	3.0	3.544388	3.544388	3.544388	3.544388	3.544388	3.544388

Chapter 5

Result Analysis

1) Visualizing the errors of all the models we tested out.

```
error_table2 = error_table.drop(["RMSE_train", "RMSE_test"], axis=1, errors='ignore')

error_table2.plot(x = "Model", kind = "bar", figsize = (25, 8), grid = True, fontsize = 15)


plt.title("Train and Test MAPE of all Models", fontsize = 20)
plt.ylabel("Error Values", fontsize = 10)
plt.xticks(rotation=60)
plt.legend(bbox_to_anchor=(1, 1), fontsize = 10)
plt.show()
```

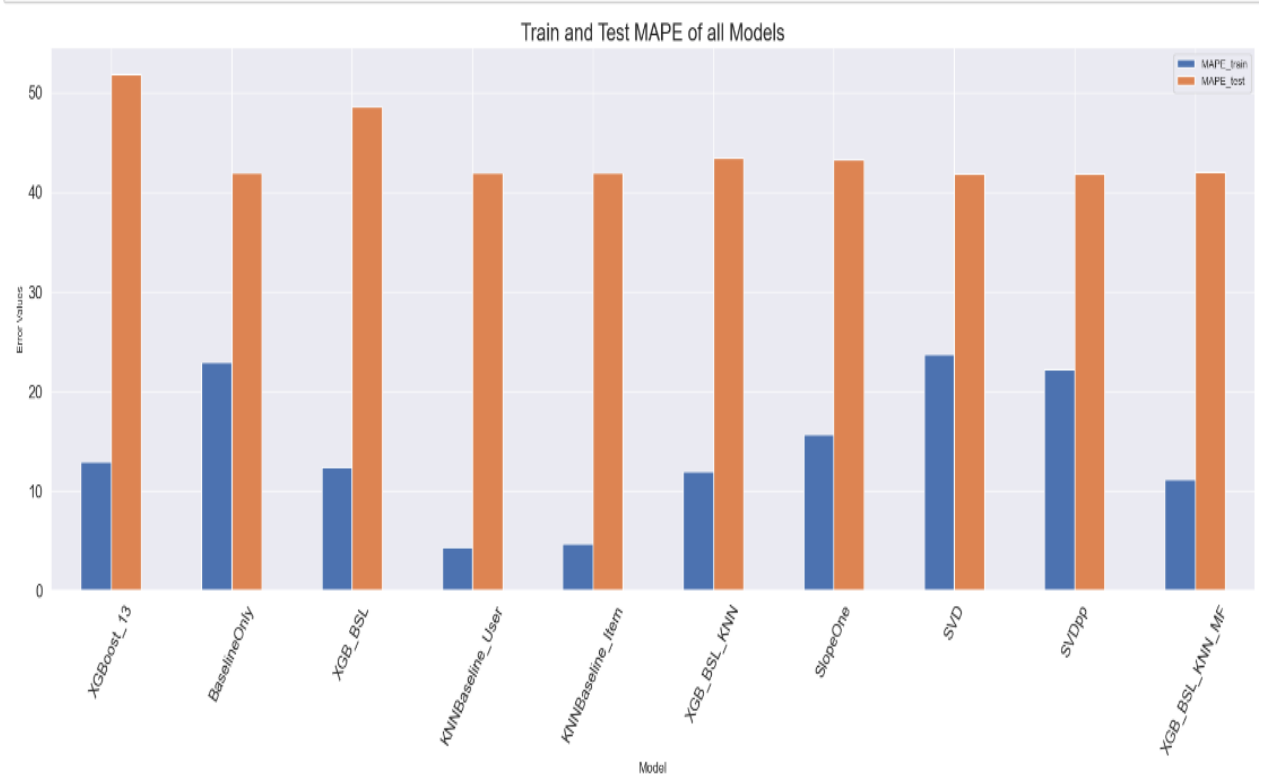


Fig:5.1

Observation---

Here we can see that BaselineOnly, KNNBaseline_User, KNNBaseline_item, SVD & SVDPP are showing less error compared to other algorithms.

2) Tabular Values of Errors.

```
error_table.drop(["Train_MAPE", "Test_MAPE"], axis = 1, errors='ignore')
```

	Model	RMSE_train	MAPE_train	RMSE_test	MAPE_test
0	XGBoost_13	0.502215	12.950331	1.254294	51.879038
1	BaselineOnly	0.774666	22.902067	1.064094	41.916075
2	XGB_BSL	0.479044	12.404606	1.189257	48.540174
3	KNNBaseline_User	0.190759	4.398733	1.063424	41.925632
4	KNNBaseline_Item	0.196855	4.723899	1.063250	41.899063
5	XGB_BSL_KNN	0.462617	11.990072	1.094762	43.473622
6	SlopeOne	0.586102	15.684078	1.087078	43.258430
7	SVD	0.788802	23.734543	1.063174	41.881904
8	SVDpp	0.747518	22.247005	1.062807	41.884178
9	XGB_BSL_KNN_MF	0.434563	11.208273	1.081923	42.033389

Fig:5.2

Chapter 6

Generating Recommendation for Users

We are using SVDPP to generate atleast 10 recommendated movies for various users.

```
# Testing the recommendations made by SVDpp Algorithm

from collections import defaultdict

def Get_top_n(predictions, n=10):

    # First map the predictions to each user.
    top_n = defaultdict(list)
    for uid, mid, true_r, est, _ in predictions:
        top_n[uid].append((mid, est))

    # Then sort the predictions for each user and retrieve the k highest ones.
    for uid, user_ratings in top_n.items():
        user_ratings.sort(key=lambda x: x[1], reverse=True)
        top_n[uid] = user_ratings[:n]

    return top_n
```

Also creating instance of SVDPP & saving the training predictions.

```
# Creating instance of svd_pp

svd_pp = SVDpp(n_factors = 10, lr_all = 0.006, verbose=True)
svd_pp.fit(trainset)
predictions = svd_pp.test(testset)
```

```
# Saving the training predictions

train_pred = svd_pp.test(trainset.build_anti_testset())
top_n = Get_top_n(train_pred, n=10)
```

Print the recommended items for each user & saving the sampled user id list to help generate movies.

```
# Print the recommended items for each user

def Generate_Recommendated_Movies(u_id, n=10):

    recommend = pd.DataFrame(top_n[u_id], columns=["Movie_Id", "Predicted_Rating"])
    recommend = recommend.merge(movies, how="inner", left_on="Movie_Id", right_on="movieId")
    recommend = recommend[["Movie_Id", "title", "genres", "Predicted_Rating"]]

    return recommend[:n]
```

```
# Saving the sampled user id list to help generate movies

sampled_user_id = list(top_n.keys())
```

And finally, generating recommendation using random user_Id.

```
# Generating recommendation using the user_Id

test_id = random.choice(sampled_user_id)
print("The user Id is : ", test_id)
Generate_Recommendated_Movies(test_id)
```

```
The user Id is :  170
```

	Movie_Id	title	genres	Predicted_Rating
0	842	Streetcar Named Desire, A	Drama	4.215233
1	8476	Guardians of the Galaxy	Action|Adventure|Sci-Fi	4.015073
2	1504	Saving Private Ryan	Action|Drama|War	3.997707
3	681	Philadelphia Story, The	Comedy|Drama|Romance	3.988467
4	7011	Inglourious Basterds	Action|Drama|War	3.959492
5	906	12 Angry Men	Drama	3.941057
6	911	Once Upon a Time in the West (C'era una volta ...	Action|Drama|Western	3.940156
7	136	Crumb	Documentary	3.936797
8	1703	Player, The	Comedy|Crime|Drama	3.920382
9	692	Apartment, The	Comedy|Drama|Romance	3.904746

Chapter 7

Deployment

Deploying a movie recommendation system involves creating an API for a trained machine learning model and choosing a deployment framework or technology.

- **Streamlit** is used to create a user-friendly interface where users input their preferences (like user ID or movie genres) and view personalized movie recommendations.
- **Flask** serves as the backend API that handles these inputs, interacts with the recommendation model (trained using collaborative filtering or other techniques), processes the data, and returns the recommended movies to Streamlit for display.

After completing the SVD++ movie recommendation model in our project, we saved it as a file, which Streamlit uses to generate personalized movie recommendations in the frontend interface. After saving the **svdpp.pkl** model file, the next steps involve setting up our development environment in VS Code and integrating the model into your movie recommendation system application.

How it Works:

- User Input: Users just give their ID to get personalized movie suggestions.
- Recommendation Generation: The system suggests top movies for the user based on their ID using collaborative filtering.
- Display: Users can see recommended movies with titles, genres, and expected ratings.
- Poster Image Fetching: We fetch movie posters from IMDb API to make recommendations.

Set Up the Development Environment

- **Install** Python and Pip

Ensure you have Python and pip installed. You can download Python from python.org which comes with pip.

- Create a Virtual Environment

Create a virtual environment to manage dependencies

```
C:\Users\Bibek Debnath\Desktop\Main Project> python -m venv venv
source venv/bin/activate  # On Windows, use `venv\Scripts\activate`
```

➢ Install Required Packages

```
pip install Flask pandas requests scikit-surprise
```

Install the necessary packages

- Flask: A micro web framework for Python
- pandas: A powerful data analysis and manipulation library.
- scikit-surprise: A Python library for building and analyzing recommender systems.
- requests: A simple HTTP library for making requests to external APIs.

➢ **Code Implementation Overview**

Load SVD++ Model: Load the trained SVD++ model from the pickled file (svdpp_model.pkl)

Function to Get Top N Recommendations

```python
# Function to get top N recommendations
def get_top_n(predictions, n=10):
    top_n = defaultdict(list)
    for uid, mid, true_r, est, _ in predictions:
        top_n[uid].append((mid, est))

    for uid, user_ratings in top_n.items():
        user_ratings.sort(key=lambda x: x[1], reverse=True)
        top_n[uid] = user_ratings[:n]

    return top_n
```

Generating Predictions and Top N Recommendations

```python
# Route for the index page
@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        user_id = int(request.form['user_id'])
        if user_id in sampled_user_ids:
            recommendations = generate_recommendated_movies(user_id, top_n, movie_info)
            return render_template('index.html', recommendations=recommendations)
        else:
            return render_template('index.html', error="User ID not found in the dataset.")
    else:
        return render_template('index.html')

if __name__ == '__main__':
    app.run(debug=True)
```

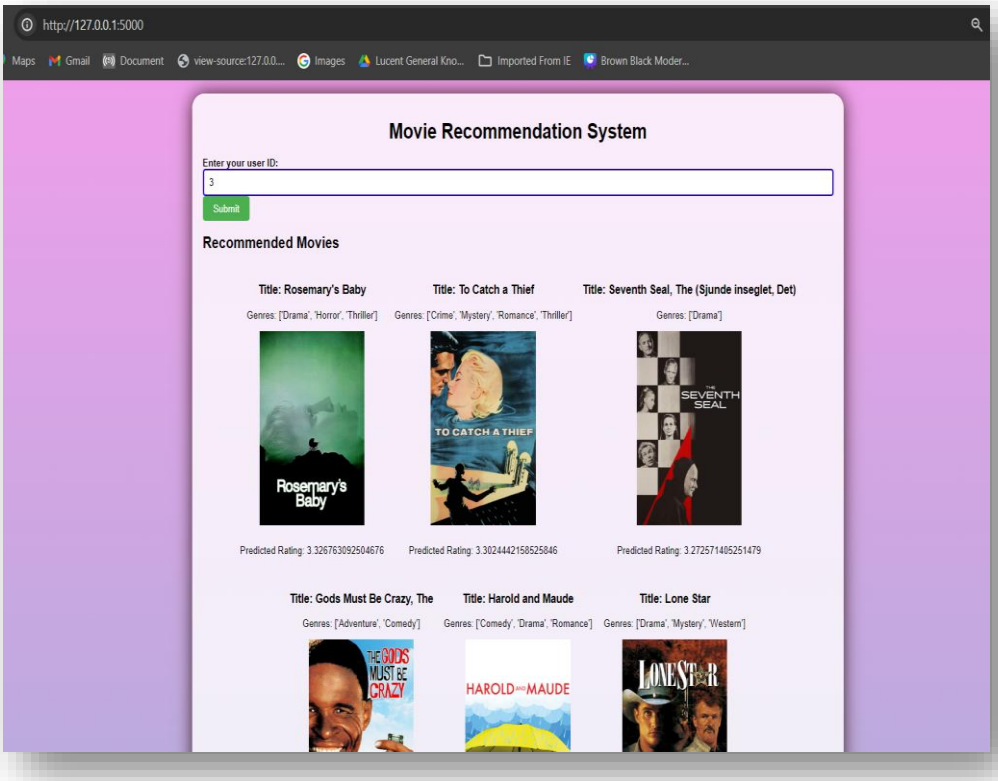
Flask route handling **G**ET and **P**OST requests for the main is the code for the index.html page that works with the Flask application. This template will display movie recommendations based on user input give recommendation.

Fetch movies details from **TMDb API** using movie title

```python
# Function to fetch movie details from TMDb API using movie title
def fetch_movie_details_by_title(movie_title):
    api_key = "8265bd1679663a7ea12ac168da84d2e8&language=en-US"
    url = f"https://api.themoviedb.org/3/search/movie?api_key={api_key}&language=en-US&query={movie_title}&page=1&include_adult=false"
    try:
        response = requests.get(url)
        response.raise_for_status()  # Raise an exception for HTTP errors
        movie_results = response.json()['results']
        if movie_results:
            return movie_results[0]  # Returning details of the first matching movie
        else:
            return None
    except requests.exceptions.RequestException as e:
        print(f"Error fetching movie details: {e}")
```

Run the flask app.py

```
S C:\Users\Bibek Debnath\Desktop\Main Project> python -u "c:\Users\Bibek Debnath\Desktop\Main Project\app.py"
processing epoch 0
processing epoch 1
processing epoch 2
processing epoch 3
processing epoch 4
```



Chapter 8

Challenges

The challenges in implementing a collaborative filtering movie recommendation system:

1. **Data Quality and Quantity:**
   - Sparse Data: Users typically rate only a small subset of all available movies, leading to sparse matrices which can make accurate recommendations harder.
   - Cold Start Problem: Difficulty in recommending movies for new users or movies with few ratings.
2. **Scalability:**
   - Handling large datasets efficiently can be challenging, especially as the number of users and movies grows.
   - Real-time recommendations may require efficient algorithms and data structures to maintain responsiveness
3. **Cold Start Issues:**
   - Dealing with new users or items that have insufficient data for accurate recommendations.
   - Implementing strategies such as popularity-based recommendations or hybrid approaches until enough data is gathered.

Conclusion

In this project, we learned how important recommendation systems are and the different types that exist. We also learned how matrix factorization can improve these systems. We built a movie recommendation system using user-user similarity, movie-movie similarity, global averages, and matrix factorization. These ideas can be used for other systems that match users with items. We made recommendations based on similarity and collaborative filtering. We predicted movie ratings based on users' past ratings and measured accuracy using RMSE and MAPE.

The aim was to provide personalized movie suggestions to users based on their preferences and the preferences of similar users. Our system implemented both user-based and item-based collaborative filtering methods, and we evaluated their performance to determine the most effective approach.

In conclusion, collaborative filtering is a powerful technique for developing movie recommendation systems, providing valuable personalized suggestions by leveraging user behavior patterns. The project highlights the potential and challenges of these methods, laying a foundation for further advancements and refinements in recommendation systems.

Future Scope

1. **Hybrid Filtering:**
   - **C**ontent-Based Filtering: Incorporating movie metadata (like genre, director, actors) to enhance recommendations based on similarities between movies.
   - **C**ollaborative Filtering: Utilizing user ratings and preferences to recommend movies similar to those liked by other users.
2. **User Registration/Login**:
   - Allowing users to create accounts and log in to personalize their experience.
   - Storing user information and preferences in a database for tailored recommendations.
3. **Rating System:**
   - Allowing users to rate movies they have watched.
   - Incorporating user ratings to refine and improve the accuracy of recommendations over time.
4. **Feedback Mechanism:**
   - Allowing users to provide feedback on recommendations (like/dislike) to further refine the system.
   - Incorporating feedback loops to continuously improve the accuracy and relevance of recommendations.

References

1] Hirdesh Shivhare, Anshul Gupta and Shalki Sharma, "Recommender system using fuzzy c means clustering and genetic algorithm based weighted similarity measure" Communication and Control. IEEE International Conference on Computer, 2015.

[2] Manoj Kumar, D.K. Yadav, Ankur Singh and Vijay Kr. Gupta. "A Movie Recommender System: MOVREC, International Journal of Computer Applications (0975-8887) Volume 124 No.3, 2015.

[3] RyuRi Kim, Ye Jeong Kwak, HyeonJeong Mo, Mucheol Kim, Seungmin Rho,Ka Lok Man, Woon Kian Chong "Trustworthy Movie Recommender System with Correct Assessment and Emotion Evaluation". Proceedings of the International MultiConference of Engineers and Computer Scientists Vol II, 2015.

[4] Zan Wang, Xue Yu*, Nan Feng, Zhenhua Wang, "An Improved Collaborative Movie Recommendation System using Computational Intelligence" Journal of Visual Languages & Computing. Volume 25, Issue 6, 2014.

[5] A. Roy and S. Ludwig, "Genre based hybrid filtering for movie recommendation engine", Journal of Intelligent Information Systems, vol. 56, no. 3, pp. 485-507, 2021. N. Shahabi and F. Najian, "A New Strategy in Trust-Based Recommender System using K Means Clustering", International Journal of Advanced Computer Science and Applications, vol. 8, no. 9, 2017.

[6] P. Sharma and L. Yadav, "MOVIE RECOMMENDATION SYSTEM USING ITEM BASED COLLABORATIVE FILTERING", International Journal of Innovative Research in Computer Science & Technology, vol. 8, no. 4, 2020.

[7] Zan Wang, Xue Yu*, Nan Feng, Zhenhua Wang (2014), “An Improved Collaborative Movie Recommendation System using Computational Intelligence”,Journal of Visual Languages & Computing,Volume 25, Issue 6.

[8] Debadrita Roy, Arnab Kundu, (2013), “Design of Movie Recommendation System by Means of Collaborative Filtering”, International Journal of Emerging Technology and Advanced Engineering, Volume 3, Issue 4.