**Recommendation System in Python**

The ability to generate value for businesses by leveraging data and applying pertinent programming abilities is the fundamental component of both Data Science (DS) and Artificial Intelligence (AI). The way individuals can now access and enjoy products and services from the comfort of their homes with just a few clicks has been transformed by industry leaders like Netflix, Amazon, and Uber Eats. These platforms have used recommendation algorithms to improve the user experience. Users are catered to by these systems, which provide an abundance of customized options that are carefully crafted to suit their individual interests and tastes. Within this framework, Python is a vital resource that provides an adaptable and strong environment for creating and implementing state-of-the-art recommendation systems. There are a lot of applications where websites collect data from their users and use that data to predict the likes and dislikes of their users. This enables people to suggest the material that interests them. Recommender systems are a means of making suggestions for products and concepts that align with a user’s particular perspective.

**Recommendation System in Python**

Python Recommendation Systems employs a data-driven methodology to offer customers tailored recommendations. It uses user data and algorithms to forecast and suggest goods, services, or content that a user is probably going to find interesting. These systems are essential in applications where users may become overwhelmed by large volumes of information, such as social media, streaming services, and e-commerce. Building recommendation systems is a common use for Python because of its modules and machine learning frameworks. The two main kinds are content-based filtering (which takes into account the characteristics of products and user profiles) and collaborative filtering (which generates recommendations based on user behaviour and preferences). Hybrid strategies that integrate the two approaches are also popular. These kinds of systems improve user experiences, boost user involvement, and propel corporate expansion.

Recommender System is of different types:

* **Content-Based Recommendation:** It is supervised machine learning used to induce a classifier to discriminate between interesting and uninteresting items for the user.
* **Collaborative Filtering:** Collaborative Filtering recommends items based on similarity measures between users and/or items. The basic assumption behind the algorithm is that users with similar interests have common preferences.

**Content-Based Recommendation System**

Content-based systems recommend items to the customer similar to previously high-rated items by the customer. It uses the features and properties of the item. From these properties, it can calculate the similarity between the items.

In a content-based recommendation system, first, we need to create a profile for each item, which represents the properties of those items. The user profiles are inferred for a particular user. We use these user profiles to recommend the items to the users from the catalog.

**Item profile**

In a content-based recommendation system, we need to build a profile for each item, which contains the important properties of each item. For Example, If the movie is an item, then its actors, director, release year, and genre are its important properties, and for the document, the important property is the type of content and set of important words in it.

Let’s have a look at how to create an item profile. First, we need to perform the TF-IDF vectorizer, here TF (term frequency) of a word is the number of times it appears in a document and The IDF (inverse document frequency) of a word is the measure of how significant that term is in the whole corpus.

**TF-IDF Vectorizer**

* **Term Frequency(TF)** : Term frequency, or TF for short, is a key idea in information retrieval and natural language processing. It displays the regularity with which a certain term or word occurs in a text corpus or document. TF is used to rank terms in a document according to their relative value or significance.
* **Inverse-document Frequency(IDF):** The measure known as Inverse Document Frequency (IDF) is employed in text analysis and information retrieval to evaluate the significance of phrases within a set of documents. IDF measures how uncommon or unique a term is in the corpus. To compute it, take the reciprocal of the fraction of documents that include the term and logarithmize it. Common terms have lower IDF values, while rare terms have higher values. IDF is an essential part of the TF-IDF (Term Frequency-Inverse Document Frequency) method, which uses it to assess the relative importance of terms in different documents. To improve information representation and retrieval from massive text datasets, IDF is used in tasks including document ranking, categorization, and text mining.

A numerical statistic called Term Frequency-Inverse Document Frequency (TF-IDF) is employed in information retrieval and natural language processing. The term’s significance within a document is assessed in relation to a group of documents (the corpus). TF emphasizes terms with greater frequencies by measuring a term’s frequency of occurrence in a document. IDF evaluates a term’s rarity within the corpus, emphasizing terms that are distinct. A weighted score is produced for each term in a document by multiplying TF and IDF together to compute TF-IDF.

**User profile**

The user profile is a vector that describes the user preference. During the creation of the user’s profile, we use a utility matrix that describes the relationship between user and item. From this information, the best estimate we can decide which item the user likes, is some aggregation of the profiles of those items.

**Advantages and Disadvantages**

* **Advantages:**
  + No need for data on other users when applying to similar users.
  + Able to recommend to users with unique tastes.
  + Able to recommend new & popular items
  + Explanations for recommended items.
* **Disadvantages**:
  + Finding the appropriate feature is hard.
  + Doesn’t recommend items outside the user profile.

**Collaborative Filtering**

Collaborative filtering is based on the idea that similar people (based on the data) generally tend to like similar things. It predicts which item a user will like based on the item preferences of other similar users.

Collaborative filtering uses a user-item matrix to generate recommendations. This matrix contains the values that indicate a user’s preference towards a given item. These values can represent either explicit feedback (direct user ratings) or implicit feedback (indirect user behavior such as listening, purchasing, watching).

* **Explicit Feedback:** The amount of data that is collected from the users when they choose to do so. Many of the times, users choose not to provide data for the user. So, this data is scarce and sometimes costs money.  For example, ratings from the user.
* **Implicit Feedback:** In implicit feedback, we track user behavior to predict their preference.

**Example:**

* Consider a user x, we need to find another user whose rating are similar to x’s rating, and then we estimate x’s rating based on another user.

|  | **M\_1** | **M\_2** | **M\_3** | **M\_4** | **M\_5** | **M\_6** | **M\_7** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **A** | 4 |  |  | 5 | 1 |  |  |
| **B** | 5 | 5 | 4 |  |  | 5 |  |
| **C** |  |  |  | 2 | 4 |  |  |
| **D** |  | 3 |  |  |  |  | 3 |

* Let’s create a matrix representing different user and movies:
* Consider two users x, y with rating vectors rx and ry. We need to decide a similarity matrix to calculate similarity b/w sim(x,y). THere are many methods to calculate similarity such as: Jaccard similarity, cosine similarity and pearson similarity. Here, we use centered cosine similarity/ pearson similarity, where we normalize the rating by subtracting the mean:

|  | **M\_1** | **M\_2** | **M\_3** | **M\_4** | **M\_5** | **M\_6** | **M\_7** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **A** | 2/3 |  |  | 5/3 | -7/3 |  |  |
| **B** | 1/3 | 1/3 | -2/3 |  |  |  |  |
| **C** |  |  |  | -5/3 | 1/3 | 4/3 |  |
| **D** |  | 0 |  |  |  |  | 0 |

* Here, we can calculate similarity: For ex: sim(A,B) = cos(rA, rB) = 0.09 ; sim(A,C) = -0.56. sim(A,B) > sim(A,C).

**Rating Predictions**

Let rx be the vector of user x’s rating. Let N be the set of k similar users who also rated item i. Then we can calculate the prediction of user x and item i by using following formula:

rxi=∑y∈NSxyryi∑y∈NSxy  Sxy=sim(x,y)rxi​=∑y∈N​Sxy​∑y∈N​Sxy​ryi​​Sxy​=sim(x,y)

**Advantages and Disadvantages**

* **Advantages:**
  + No need for the domain knowledge because embedding are learned automatically.
  + Capture inherent subtle characteristics.
* **Disadvantages**:
  + Cannot handle fresh items due to cold start problem.
  + Hard to add any new features that may improve quality of model

**Implementation of Recommendation System**

**Importing Libraries**

# Importing Libraries

import numpy as np

import pandas as pd

import sklearn

import matplotlib.pyplot as plt

import seaborn as sns

import warnings

warnings.simplefilter(action='ignore', category=FutureWarning)

The Python environment for data analysis and visualization is initialized using this line of code. First, it imports essential libraries for data processing and visualization, including NumPy, Pandas, scikit-learn, Matplotlib, and Seaborn. It also sets up the code to suppress future warnings, so that cautions about upcoming library changes don’t clog the output and create a messier, less productive workspace. These preparatory actions create the framework for effective data exploration and analysis with the imported tools.

**Loading Datasets**

#loading rating dataset

ratings = pd.read\_csv("ratings.csv")

print(ratings.head())

**Output:**

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

# loading movie dataset

movies = pd.read\_csv("movies.csv")

print(movies.head())

**Output:**

movieId title\  
0 1 Toy Story (1995)   
1 2 Jumanji (1995)   
2 3 Grumpier Old Men (1995)   
3 4 Waiting to Exhale (1995)   
4 5 Father of the Bride Part II (1995)   
  
 genres   
0 Adventure|Animation|Children|Comedy|Fantasy   
1 Adventure|Children|Fantasy   
2 Comedy|Romance   
3 Comedy|Drama|Romance   
4 Comedy

Two datasets are imported into this code to do a movie recommendation study. User ratings for movies are included in the first dataset, “ratings.csv,” which is kept in a Pandas DataFrame named ratings. The second dataset, called “movies.csv,” is put into a Pandas DataFrame called “movies” and contains movie metadata like names and genres. In order to give a preliminary overview of the data and lay the groundwork for further analysis or recommendation system development, the code displays the first few rows of each DataFrame.

**Statistical Analysis of Ratings**

n\_ratings = len(ratings)

n\_movies = len(ratings['movieId'].unique())

n\_users = len(ratings['userId'].unique())

print(f"Number of ratings: {n\_ratings}")

print(f"Number of unique movieId's: {n\_movies}")

print(f"Number of unique users: {n\_users}")

print(f"Average ratings per user: {round(n\_ratings/n\_users, 2)}")

print(f"Average ratings per movie: {round(n\_ratings/n\_movies, 2)}")

**Output:**

Number of ratings: 100836  
Number of unique movieId's: 9724  
Number of unique users: 610  
Average ratings per user: 165.3  
Average ratings per movie: 10.37

This code computes and reports a number of crucial statistics for a movie ratings dataset. It counts the number of unique movie IDs (n\_movies) and user IDs (n\_users) as well as the total number of ratings (n\_ratings). These metrics provide important information about the properties of the dataset, including its size and the variety of people and movies inside it. To give a more complete picture of the distribution of ratings throughout the dataset, it also calculates and shows the average number of ratings for each user and each movie. Understanding the size and user interaction of the dataset requires knowledge of this information.

**User Rating Frequency**

user\_freq = ratings[['userId', 'movieId']].groupby( 'userId').count().reset\_index()

user\_freq.columns = ['userId', 'n\_ratings']

print(user\_freq.head())

**Output:**

userId n\_ratings  
0 1 232  
1 2 29  
2 3 39  
3 4 216  
4 5 44

The movie ratings dataset’s user-specific statistics are computed and shown in this code segment. After classifying the data according to user IDs, it calculates the total number of ratings each user has submitted and saves the results in a new DataFrame named user\_freq. With ‘userId’ denoting the user ID and ‘n\_ratings’ the number of ratings the user has contributed, the columns are suitably labeled. To facilitate additional user-based analysis and the creation of recommendation systems, this user-level frequency information is crucial for comprehending user engagement and activity inside the rating dataset. The first few rows of this DataFrame are shown for a brief summary of user-specific rating counts by the print(user\_freq.head()) line.

**Movie Rating Analysis**

# Find Lowest and Highest rated movies:

mean\_rating = ratings.groupby('movieId')[['rating']].mean()

# Lowest rated movies

lowest\_rated = mean\_rating['rating'].idxmin()

movies.loc[movies['movieId'] == lowest\_rated]

# Highest rated movies

highest\_rated = mean\_rating['rating'].idxmax()

movies.loc[movies['movieId'] == highest\_rated]

# show number of people who rated movies rated movie highest

ratings[ratings['movieId']==highest\_rated]

# show number of people who rated movies rated movie lowest

ratings[ratings['movieId']==lowest\_rated]

## the above movies has very low dataset. We will use bayesian average

movie\_stats = ratings.groupby('movieId')[['rating']].agg(['count', 'mean'])

movie\_stats.columns = movie\_stats.columns.droplevel()

To determine which movies in the dataset have the lowest and highest ratings, this algorithm analyzes movie reviews. It determines the average ratings for every film, making it possible to identify which ones have the lowest and greatest average ratings. Subsequently, the algorithm accesses and presents the information about these films from the’movies’ dataset. It also sheds light on the popularity and audience involvement of the movie by displaying the number of users who rated both the highest and lowest-ranked ones. This gives insights into user engagement. Bayesian averages may offer more accurate quality ratings for films with a small number of ratings.

**User-Item Matrix Creation**

# Now, we create user-item matrix using scipy csr matrix

from scipy.sparse import csr\_matrix

def create\_matrix(df):

N = len(df['userId'].unique())

M = len(df['movieId'].unique())

# Map Ids to indices

user\_mapper = dict(zip(np.unique(df["userId"]), list(range(N))))

movie\_mapper = dict(zip(np.unique(df["movieId"]), list(range(M))))

# Map indices to IDs

user\_inv\_mapper = dict(zip(list(range(N)), np.unique(df["userId"])))

movie\_inv\_mapper = dict(zip(list(range(M)), np.unique(df["movieId"])))

user\_index = [user\_mapper[i] for i in df['userId']]

movie\_index = [movie\_mapper[i] for i in df['movieId']]

X = csr\_matrix((df["rating"], (movie\_index, user\_index)), shape=(M, N))

return X, user\_mapper, movie\_mapper, user\_inv\_mapper, movie\_inv\_mapper

X, user\_mapper, movie\_mapper, user\_inv\_mapper, movie\_inv\_mapper = create\_matrix(ratings)

A user-item matrix is a basic data structure in recommendation systems, and it is created by the code that is given. This is how it operates:

* To find the number of unique users and unique videos in the dataset, N and M are computed.
* There are four dictionaries produced:
  + **user\_mapper**: Maps distinct user IDs to indexes (user ID 1 becomes index 0 for example).
  + **movie\_mapper**: Converts distinct movie IDs into indices (movie ID 1 becomes index 0 for example).
  + **user\_inv\_mapper**: Reverses user\_mapper and maps indices back to user IDs.
  + **movie\_inv\_mapper**: Reverses movie\_mapper by mapping indices to movie IDs.
* To map the real user and movie IDs in the dataset to their matching indices, the lists user\_index and movie\_index are generated.
* A sparse matrix X is created using the SciPy function csr\_matrix. The user and movie indices that correspond to the rating values in the dataset are used to generate this matrix. The form of it is (M, N), where M denotes the quantity of distinct films and N denotes the quantity of distinct consumers.

To put it another way, this code makes it easy to do calculations and create recommendation systems based on the structured representation of user ratings for movies in the data.

**Movie Similarity Analysis**

"""

Find similar movies using KNN

"""

from sklearn.neighbors import NearestNeighbors

def find\_similar\_movies(movie\_id, X, k, metric='cosine', show\_distance=False):

neighbour\_ids = []

movie\_ind = movie\_mapper[movie\_id]

movie\_vec = X[movie\_ind]

k+=1

kNN = NearestNeighbors(n\_neighbors=k, algorithm="brute", metric=metric)

kNN.fit(X)

movie\_vec = movie\_vec.reshape(1,-1)

neighbour = kNN.kneighbors(movie\_vec, return\_distance=show\_distance)

for i in range(0,k):

n = neighbour.item(i)

neighbour\_ids.append(movie\_inv\_mapper[n])

neighbour\_ids.pop(0)

return neighbour\_ids

movie\_titles = dict(zip(movies['movieId'], movies['title']))

movie\_id = 3

similar\_ids = find\_similar\_movies(movie\_id, X, k=10)

movie\_title = movie\_titles[movie\_id]

print(f"Since you watched {movie\_title}")

for i in similar\_ids:

print(movie\_titles[i])

**Output:**

Since you watched Grumpier Old Men (1995)  
Grumpy Old Men (1993)  
Striptease (1996)  
Nutty Professor, The (1996)  
Twister (1996)  
Father of the Bride Part II (1995)  
Broken Arrow (1996)  
Bio-Dome (1996)  
Truth About Cats & Dogs, The (1996)  
Sabrina (1995)  
Birdcage, The (1996)

The provided code defines a function, “find\_similar\_movies,” which uses the k-Nearest Neighbors (KNN) algorithm to identify movies that are similar to a given movie. The function takes inputs such as the target movie ID, a user-item matrix (X), the number of neighbors to consider (k), a similarity metric (default is cosine similarity), and an option to show distances between movies. The function begins by initializing a blank list to hold the IDs of films that are comparable. It takes the target movie’s index out of the movie\_mapper dictionary and uses the user-item matrix to acquire the feature vector that goes with it. Next, the KNN model is configured using the given parameters.

The distances and indices of the k-nearest neighbors to the target movie are calculated once the KNN model has been fitted. Using the movie\_inv\_mapper dictionary, the loop retrieves these neighbor indices and maps them back to movie IDs. Since it matches the desired movie, the first item in the list is eliminated. The code ends with a list of related movie titles and the title of the target film, suggesting movies based on the KNN model.

**Movie Recommendation with respect to Users Preference**

Create a function to recomment the movies based on the user preferences.

def recommend\_movies\_for\_user(user\_id, X, user\_mapper, movie\_mapper, movie\_inv\_mapper, k=10):

df1 = ratings[ratings['userId'] == user\_id]

if df1.empty:

print(f"User with ID {user\_id} does not exist.")

return

movie\_id = df1[df1['rating'] == max(df1['rating'])]['movieId'].iloc[0]

movie\_titles = dict(zip(movies['movieId'], movies['title']))

similar\_ids = find\_similar\_movies(movie\_id, X, k)

movie\_title = movie\_titles.get(movie\_id, "Movie not found")

if movie\_title == "Movie not found":

print(f"Movie with ID {movie\_id} not found.")

return

print(f"Since you watched {movie\_title}, you might also like:")

for i in similar\_ids:

print(movie\_titles.get(i, "Movie not found"))

The function accepts the following inputs: dictionaries (user\_mapper, movie\_mapper, and movie\_inv\_mapper) for mapping user and movie IDs to matrix indices; the user\_id for which recommendations are desired; a user-item matrix X representing movie ratings; and an optional parameter k for the number of recommended movies (default is 10).

* It initially filters the ratings dataset to see if the user with the given ID is there. It notifies the user that the requested person does not exist and ends the function if the user does not exist (the filtered DataFrame is empty).
* The code, if it exists, designates the movie that has received the highest rating from that particular user. It finds the movieId of this movie and chooses it based on the highest rating.
* With information from the movies dataset, a dictionary called movie\_titles is created to map movie IDs to their titles. The function then uses find\_similar\_movies to locate films that are comparable to the movie in the user-item matrix that has the highest rating (denoted by movie\_id). It gives back a list of comparable movie IDs.
* The code searches the movie titles dictionary for the title of the highest-rated film, and if the film is not found, it sets the title to “Movie not found.” When a movie title is retrieved as “Movie not found,” it means that the highest-rated film (based on movie\_id) is not present in the dataset. If the movie is located, the customer is presented with recommendations for other movies based on the highest rated film. The list of comparable movie IDs is iterated over, and the titles are printed. When a movie isn’t discovered in the dataset, the default message is “Movie not found.”
* The function handles situations where the user or movie doesn’t exist in the dataset and is intended to suggest movies for a particular user based on their highest-rated film. The code calls the function with the necessary parameters and sets the user\_id to a specific user to show how to utilize the method.

**Reccomment the movies**

user\_id = 150 # Replace with the desired user ID

recommend\_movies\_for\_user(user\_id, X, user\_mapper, movie\_mapper, movie\_inv\_mapper, k=10)

**Output:**

Since you watched Twelve Monkeys (a.k.a. 12 Monkeys) (1995), you might also like:  
Pulp Fiction (1994)  
Terminator 2: Judgment Day (1991)  
Independence Day (a.k.a. ID4) (1996)  
Seven (a.k.a. Se7en) (1995)  
Fargo (1996)  
Fugitive, The (1993)  
Usual Suspects, The (1995)  
Jurassic Park (1993)  
Star Wars: Episode IV - A New Hope (1977)  
Heat (1995)

user\_id = 415 # Replace with the desired user ID

recommend\_movies\_for\_user(user\_id, X, user\_mapper, movie\_mapper, movie\_inv\_mapper, k=10)

**Output**:

Since you watched Pulp Fiction (1994), you might also like:  
Silence of the Lambs, The (1991)  
Shawshank Redemption, The (1994)  
Seven (a.k.a. Se7en) (1995)  
Forrest Gump (1994)  
Usual Suspects, The (1995)  
Braveheart (1995)  
Fight Club (1999)  
Fargo (1996)  
Terminator 2: Judgment Day (1991)  
Reservoir Dogs (1992)

**Conclusion**

In conclusion, developing a Python recommendation system allows for the creation of tailored content recommendations that improve user experience and take into account user preferences. Through the utilization of collaborative filtering, content-based filtering, and hybrid techniques, these systems are able to offer customized recommendations to consumers for content, movies, or items. These systems use sophisticated methods such as closest neighbors and matrix factorization to find hidden patterns in item attributes and user behavior. Recommendation systems are able to adjust and get better over time thanks to the combination of machine learning and data-driven insights. In the end, these solutions are essential for raising consumer satisfaction, improving user engagement, and propelling corporate expansion in a variety of industries.