

✓ Project: Amazon Product Recommendation System

Marks: 60

Welcome to the project on Recommendation Systems. We will work with the Amazon product reviews dataset for this project. The dataset contains ratings of different electronic products. It does not include information about the products or reviews to avoid bias while building the model.

Context:

Today, information is growing exponentially with volume, velocity and variety throughout the globe. This has led to information overload, and too many choices for the consumer of any business. It represents a real dilemma for these consumers and they often turn to denial.

Recommender Systems are one of the best tools that help recommending products to consumers while they are browsing online. Providing personalized recommendations which is most relevant for the user is what's most likely to keep them engaged and help business.

E-commerce websites like Amazon, Walmart, Target and Etsy use different recommendation models to provide personalized suggestions to different users. These companies spend millions of dollars to come up with algorithmic techniques that can provide personalized recommendations to their users.

Amazon, for example, is well-known for its accurate selection of recommendations in its online site. Amazon's recommendation system is capable of intelligently analyzing and predicting customers' shopping preferences in order to offer them a list of recommended products. Amazon's recommendation algorithm is therefore a key element in using AI to improve the personalization of its website. For example, one of the baseline recommendation models that Amazon uses is item-to-item collaborative filtering, which scales to massive data sets and produces high-quality recommendations in real-time.

Objective:

You are a Data Science Manager at Amazon, and have been given the task of building a recommendation system to recommend products to customers based on their previous ratings for other products. You have a collection of labeled data of Amazon reviews of products. The goal is to extract meaningful insights from the data and build a recommendation system that helps in recommending products to online consumers.

Dataset:

The Amazon dataset contains the following attributes:

- **userId:** Every user identified with a unique id
- **productId:** Every product identified with a unique id
- **Rating:** The rating of the corresponding product by the corresponding user
- **timestamp:** Time of the rating. We **will not use this column** to solve the current problem

✓ Please read the instructions carefully before starting the project.

This is a commented Jupyter IPython Notebook file in which all the instructions and tasks to be performed are mentioned. Read along carefully to complete the project.

- Blanks '___' are provided in the notebook that needs to be filled with an appropriate code to get the correct result. Please replace the blank with the right code snippet. With every '___' blank, there is a comment that briefly describes what needs to be filled in the blank space.
- Identify the task to be performed correctly, and only then proceed to write the required code.
- Fill the code wherever asked by the commented lines like "# Fill in the blank" or "# Complete the code". Running incomplete code may throw an error.
- Remove the blank and state your observations in detail wherever the mark down says 'Write your observations here:____'
- Please run the codes in a sequential manner from the beginning to avoid any unnecessary errors.
- You can the results/observations derived from the analysis here and use them to create your final report.

Sometimes, the installation of the surprise library, which is used to build recommendation systems, faces issues in Jupyter. To avoid any issues, it is advised to use **Google Colab** for this project.

Let's start by mounting the Google drive on Colab.

```
from google.colab import drive
drive.mount('/content/drive')
```

Mounted at /content/drive

Installing surprise library

```
!pip install surprise
```

```
Collecting surprise
  Downloading surprise-0.1-py2.py3-none-any.whl.metadata (327 bytes)
Collecting scikit-surprise (from surprise)
  Downloading scikit_surprise-1.1.4.tar.gz (154 kB)
    154.4/154.4 kB 363.0 kB/s eta 0:00:00
Installing build dependencies ... done
Getting requirements to build wheel ... done
Preparing metadata (pyproject.toml) ... done
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from scikit-surprise->surprise) (1.4.2)
Requirement already satisfied: numpy>=1.19.5 in /usr/local/lib/python3.10/dist-packages (from scikit-surprise->surprise) (1.25.2)
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.10/dist-packages (from scikit-surprise->surprise) (1.13.1)
Downloading surprise-0.1-py2.py3-none-any.whl (1.8 kB)
Building wheels for collected packages: scikit-surprise
  Building wheel for scikit-surprise (pyproject.toml) ... done
  Created wheel for scikit-surprise: filename=scikit_surprise-1.1.4-cp310-cp310-linux_x86_64.whl size=2357246 sha256=bddd9bf220421935cb0
  Stored in directory: /root/.cache/pip/wheels/4b/3f/df/6acbf0a40397d9bf3ff97f582cc22fb9ce66adde75bc71fd54
Successfully built scikit-surprise
Installing collected packages: scikit-surprise, surprise
Successfully installed scikit-surprise-1.1.4 surprise-0.1
```

Importing the necessary libraries and overview of the dataset

```
import warnings # Used to ignore the warning given as output of the code
warnings.filterwarnings('ignore')

import numpy as np # Basic libraries of python for numeric and dataframe computations
import pandas as pd

import matplotlib.pyplot as plt # Basic library for data visualization
import seaborn as sns # Slightly advanced library for data visualization

from collections import defaultdict # A dictionary output that does not raise a key error

from sklearn.metrics import mean_squared_error # A performance metrics in sklearn
```

Loading the data

```
# Import the dataset
df = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/ratings_Electronics (1).csv', header = None) # There are no headers in the data file

df.columns = ['user_id', 'prod_id', 'rating', 'timestamp'] # Adding column names

df = df.drop('timestamp', axis = 1) # Dropping timestamp

df_copy = df.copy(deep = True) # Copying the data to another DataFrame
```

As this dataset is very large and has 7,824,482 observations, it is not computationally possible to build a model using this. Moreover, many users have only rated a few products and also some products are rated by very few users. Hence, we can reduce the dataset by considering certain logical assumptions.

Here, we will be taking users who have given at least 50 ratings, and the products that have at least 5 ratings, as when we shop online we prefer to have some number of ratings of a product.

```
# Get the column containing the users
```

```

# Get the column containing the users
users = df.user_id

# Create a dictionary from users to their number of ratings
ratings_count = dict()

for user in users:

    # If we already have the user, just add 1 to their rating count
    if user in ratings_count:
        ratings_count[user] += 1

    # Otherwise, set their rating count to 1
    else:
        ratings_count[user] = 1

# We want our users to have at least 50 ratings to be considered
RATINGS_CUTOFF = 50

remove_users = []

for user, num_ratings in ratings_count.items():
    if num_ratings < RATINGS_CUTOFF:
        remove_users.append(user)

df = df.loc[ ~ df.user_id.isin(remove_users)]

# Get the column containing the products
prods = df.prod_id

# Create a dictionary from products to their number of ratings
ratings_count = dict()

for prod in prods:

    # If we already have the product, just add 1 to its rating count
    if prod in ratings_count:
        ratings_count[prod] += 1

    # Otherwise, set their rating count to 1
    else:
        ratings_count[prod] = 1

# We want our item to have at least 5 ratings to be considered
RATINGS_CUTOFF = 5


remove_users = []

for user, num_ratings in ratings_count.items():
    if num_ratings < RATINGS_CUTOFF:
        remove_users.append(user)

df_final = df.loc[~ df.prod_id.isin(remove_users)]

# Print a few rows of the imported dataset
df_final.head()

```



	user_id	prod_id	rating
1310	A3LDPF5FMB782Z	1400501466	5.0
1322	A1A5KUIIIHFF4U	1400501466	1.0
1335	A2XIOXRRYX0KZY	1400501466	3.0
1451	AW3LX47IHPFRL	1400501466	5.0
1456	A1E3OB6QMBKRYZ	1400501466	1.0

✎ Exploratory Data Analysis

✎ Shape of the data

```
# Check the number of rows and columns and provide observations
rows, columns = df_final.shape
print("No of rows: ", rows)
print("No of columns: ", columns)
```

```
↗ No of rows: 65290
  No of columns: 3
```

this dataset has 65,290 product ratings from amazon users, with 3 columns giving us a efficient base for understanding customers and exploring their patterns to give better product recommendations.

Double-click (or enter) to edit

▼ Data types

```
# Check Data types and provide observations
df_final.dtypes
```

```
↗ user_id    object
  prod_id    object
  rating    float64
  dtype: object
```

with these object and float64 dtypes they naturally can lead to sparse matrix representation which is fine beacuse of the surprise library we will use to implement standard recommendation models

▼ Checking for missing values

```
# Check for missing values present and provide observations
missing_values = df_final.isnull().sum()

print(missing_values)
```

```
↗ user_id    0
  prod_id    0
  rating     0
  dtype: int64
```

When examining our dataset for missing values, we found that we have complete data for working with our models!

▼ Summary Statistics

```
# Summary statistics of 'rating' variable and provide observations
df_final['rating'].describe()
```

```
↗ count    65290.000000
  mean      4.294808
  std       0.988915
  min       1.000000
  25%       4.000000
  50%       5.000000
  75%       5.000000
  max       5.000000
  Name: rating, dtype: float64
```

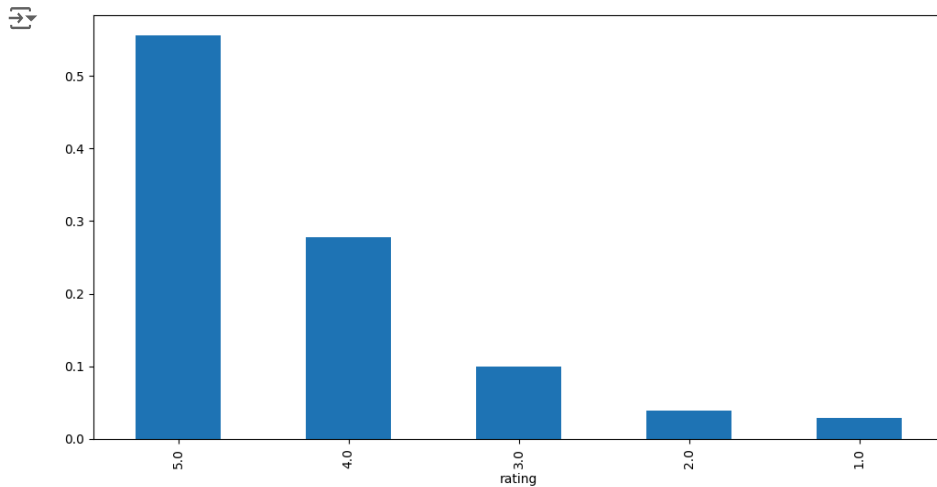
- ratings range from 1-5
- average rating is about 4.29, overall customer satisfaction is high
- 75% of all ratings are 4 stars and higher
- there might be a positive bias, a high concentration of 5 star ratings could mean rating inflation.

▼ Checking the rating distribution

```
# Create the bar plot and provide observations
plt.figure(figsize = (12, 6))
```

```
df_final['rating'].value_counts(1).plot(kind = 'bar');
```

```
plt.show()
```



```
# This is formatted as code
```

1. strong positive bias a product with 4 stars might actually be underperforming, further test will answer correctly
2. Because negative ratings are rare, they might carry extra significance.

✓ Checking the number of unique users and items in the dataset

```
# Number of total rows in the data and number of unique user id and product id in the data
```

```
print('The number of observations in the final data = ', len(df_final))
print('Number of unique USERS in Raw data = ', df_final['user_id'].nunique())
print('Number of unique ITEMS in Raw data = ', df_final['prod_id'].nunique())
```

```
The number of observations in the final data = 65290
Number of unique USERS in Raw data = 1540
Number of unique ITEMS in Raw data = 5689
```

- Average ratings per user: $65,290 / 1,540 = 42.4$
- Each user has rated about 42 products, which is a good level of user engagement and a good amount for collaborative filtering.
- Average ratings per product: $65,290 / 5,689 = 11.5$
- Each product has received about 11-12 ratings, a decent amount for recommendations.
- Theoretical maximum ratings: $1,540 * 5,689 = 8,761,060$ vs. Actual ratings: 65,290
- indicates a sparse dataset

✓ Users with the most number of ratings

```
# Top 10 users based on the number of ratings
most_rated = df_final.groupby('user_id').size().sort_values(ascending = False)[:10]
most_rated
```

```

user_id
ADLVFFE4VBT8    295
A30XHLG6DIBRW8  230
A10DOGXKEYECQQ8 217
A36K2N527TXXJN  212
A25C2M3QF9G7OQ  203
A680RUE1FD08B   196
A22CW0ZHY3NJH8  193
A1UQBFCERIP7VJ  193
AWPODHOB4GFWL   184
A3LGT6UZL99IW1  179
dtype: int64

```

- The highest number of **ratings by a user is 295** which is far from the actual number of products present in the data. We can build a recommendation system to recommend products to users which they have not interacted with.

Now that we have explored and prepared the data, let's build the first recommendation system.

✓ Model 1: Rank Based Recommendation System

```
df_final.head()
```

```

      user_id  prod_id  rating
1310  A3LDPF5FMB782Z  1400501466    5.0
1322  A1A5KUIIIHFF4U  1400501466    1.0
1335  A2XIOXRRYX0KZY  1400501466    3.0
1451  AW3LX47IHPFRL  1400501466    5.0
1456  A1E3OB6QMBKRYZ  1400501466    1.0

```

✓ Steps:

- Create the **final_rating DataFrame**
- Complete the code to create the function **top_n_products**
- Recommending **top 5 products with 50 minimum interactions based on popularity**
- Recommending **top 5 products with 100 minimum interactions based on popularity**

```

# Calculate the average rating for each product
average_rating = df_final.groupby('prod_id')['rating'].mean()

# Calculate the count of ratings for each product
count_rating = df_final.groupby('prod_id')['rating'].count()

# Create a dataframe with calculated average and count of ratings
final_rating = pd.DataFrame({'average_rating': average_rating, 'count_rating': count_rating})

# Sort the dataframe by average of ratings in the descending order
final_rating = final_rating.sort_values('average_rating', ascending=False)

# See the first five records of the "final_rating" dataset
final_rating.head()

```

```

      average_rating  count_rating
prod_id
B00LGQ6HL8         5.0            5
B003DZJQQI         5.0           14
B005FDXF2C         5.0            7
B00I6CVPVC         5.0            7
B00B9KOCYA         5.0            8

```

```
# Defining a function to get the top n products based on the highest average rating and minimum interactions
```

```
def top_n_products(final_rating, n, min_interaction):

    # Finding products with minimum number of interactions
    recommendations = final_rating[final_rating['count_rating'] >= min_interaction]

    # Sorting values with respect to average rating
    recommendations = recommendations.sort_values('average_rating', ascending=False)

    return recommendations.index[:n]
```

✓ Recommending top 5 products with 50 minimum interactions based on popularity

```
top_5_products = top_n_products(final_rating, n=5, min_interaction=50)
print(top_5_products)
```

```
Index(['B001TH7GUU', 'B003E55ZUU', 'B0019EHU8G', 'B006W8U2MU', 'B000QUUFRW'], dtype='object', name='prod_id')
```

✓ Recommending top 5 products with 100 minimum interactions based on popularity

```
top_5_products_100 = top_n_products(final_rating, n=5, min_interaction=100)
print(top_5_products_100)
```

```
Index(['B003E55ZUU', 'B000N99BBC', 'B002WE6D44', 'B007WTAJTO', 'B002V88HFE'], dtype='object', name='prod_id')
```

We have recommended the **top 5** products by using the popularity recommendation system. Now, let's build a recommendation system using **collaborative filtering**.

✓ Model 2: Collaborative Filtering Recommendation System

In this type of recommendation system, we do not need any information about the users or items. We only need user item interaction data to build a collaborative recommendation system. For example -

1. **Ratings** provided by users. For example, ratings of books on goodread, movie ratings on imdb, etc.
2. **Likes** of users on different facebook posts, likes on youtube videos.
3. **Use/buying** of a product by users. For example, buying different items on e-commerce sites.
4. **Reading** of articles by readers on various blogs.

Types of Collaborative Filtering

- Similarity/Neighborhood based
 - User-User Similarity Based
 - Item-Item similarity based
- Model based

✓ Building a baseline user-user similarity based recommendation system

- Below, we are building **similarity-based recommendation systems** using cosine similarity and using **KNN to find similar users** which are the nearest neighbor to the given user.
- We will be using a new library, called `surprise`, to build the remaining models. Let's first import the necessary classes and functions from this library.

```
# To compute the accuracy of models
from surprise import accuracy

# Class is used to parse a file containing ratings, data should be in structure - user ; item ; rating
from surprise.reader import Reader

# Class for loading datasets
from surprise.dataset import Dataset

# For tuning model hyperparameters
from surprise.model_selection import GridSearchCV

# For splitting the rating data in train and test datasets
from surprise.model_selection import train_test_split

# For implementing similarity-based recommendation system
from surprise.prediction_algorithms.knns import KNNBasic

# For implementing matrix factorization based recommendation system
from surprise.prediction_algorithms.matrix_factorization import SVD

# for implementing K-Fold cross-validation
from surprise.model_selection import KFold

# For implementing clustering-based recommendation system
from surprise import CoClustering
```

Before building the recommendation systems, let's go over some basic terminologies we are going to use:

Relevant item: An item (product in this case) that is actually **rated higher than the threshold rating** is relevant, if the **actual rating is below the threshold then it is a non-relevant item**.

Recommended item: An item that's **predicted rating is higher than the threshold** is a **recommended item**, if the **predicted rating is below the threshold then that product will not be recommended to the user**.

False Negative (FN): It is the **frequency of relevant items that are not recommended to the user**. If the relevant items are not recommended to the user, then the user might not buy the product/item. This would result in the **loss of opportunity for the service provider**, which they would like to minimize.

False Positive (FP): It is the **frequency of recommended items that are actually not relevant**. In this case, the recommendation system is not doing a good job of finding and recommending the relevant items to the user. This would result in **loss of resources for the service provider**, which they would also like to minimize.

Recall: It is the **fraction of actually relevant items that are recommended to the user**, i.e., if out of 10 relevant products, 6 are recommended to the user then recall is 0.60. Higher the value of recall better is the model. It is one of the metrics to do the performance assessment of classification models.

Precision: It is the **fraction of recommended items that are relevant actually**, i.e., if out of 10 recommended items, 6 are found relevant by the user then precision is 0.60. The higher the value of precision better is the model. It is one of the metrics to do the performance assessment of classification models.

While making a recommendation system, it becomes customary to look at the performance of the model. In terms of how many recommendations are relevant and vice-versa, below are some most used performance metrics used in the assessment of recommendation systems.

✓ Precision@k, Recall@ k, and F1-score@k

Precision@k - It is the **fraction of recommended items that are relevant in top k predictions**. The value of k is the number of recommendations to be provided to the user. One can choose a variable number of recommendations to be given to a unique user.

Recall@k - It is the **fraction of relevant items that are recommended to the user in top k predictions**.

F1-score@k - It is the **harmonic mean of Precision@k and Recall@k**. When **precision@k and recall@k both seem to be important** then it is useful to use this metric because it is representative of both of them.

✓ Some useful functions

- Below function takes the **recommendation model** as input and gives the **precision@k**, **recall@k**, and **F1-score@k** for that model.
- To compute **precision and recall**, **top k** predictions are taken under consideration for each user.
- We will use the precision and recall to compute the F1-score.

```
def precision_recall_at_k(model, k = 10, threshold = 3.5):
    """Return precision and recall at k metrics for each user"""

    # First map the predictions to each user
    user_est_true = defaultdict(list)

    # Making predictions on the test data
    predictions = model.test(testset)

    for uid, _, true_r, est, _ in predictions:
        user_est_true[uid].append((est, true_r))

    precisions = dict()
    recalls = dict()
    for uid, user_ratings in user_est_true.items():

        # Sort user ratings by estimated value
        user_ratings.sort(key = lambda x: x[0], reverse = True)

        # Number of relevant items
        n_rel = sum((true_r >= threshold) for (_, true_r) in user_ratings)

        # Number of recommended items in top k
        n_rec_k = sum((est >= threshold) for (est, _) in user_ratings[:k])

        # Number of relevant and recommended items in top k
        n_rel_and_rec_k = sum(((true_r >= threshold) and (est >= threshold))
                               for (est, true_r) in user_ratings[:k])

        # Precision@K: Proportion of recommended items that are relevant
        # When n_rec_k is 0, Precision is undefined. Therefore, we are setting Precision to 0 when n_rec_k is 0

        precisions[uid] = n_rel_and_rec_k / n_rec_k if n_rec_k != 0 else 0

        # Recall@K: Proportion of relevant items that are recommended
        # When n_rel is 0, Recall is undefined. Therefore, we are setting Recall to 0 when n_rel is 0

        recalls[uid] = n_rel_and_rec_k / n_rel if n_rel != 0 else 0

    # Mean of all the predicted precisions are calculated.
    precision = round((sum(prec for prec in precisions.values()) / len(precisions)), 3)

    # Mean of all the predicted recalls are calculated.
    recall = round((sum(rec for rec in recalls.values()) / len(recalls)), 3)

    accuracy.rmse(predictions)

    print('Precision: ', precision) # Command to print the overall precision

    print('Recall: ', recall) # Command to print the overall recall


    print('F_1 score: ', round((2*precision*recall)/(precision+recall), 3)) # Formula to compute the F-1 score
```

Hints:

- To compute **precision and recall**, a **threshold of 3.5** and **k value of 10** can be considered for the recommended and relevant ratings.
- Think about the performance metric to choose.

Below we are loading the **rating dataset**, which is a **pandas DataFrame**, into a **different format called** `surprise.dataset.DatasetAutoFolds`, which is required by this library. To do this, we will be **using the classes Reader and Dataset**.

```
df_final.head()
```



	user_id	prod_id	rating
1310	A3LDPF5FMB782Z	1400501466	5.0
1322	A1A5KUIIIHFF4U	1400501466	1.0
1335	A2XIOXRRYX0KZY	1400501466	3.0
1451	AW3LX47IHPFRL	1400501466	5.0
1456	A1E3OB6QMBKRYZ	1400501466	1.0

```
# Instantiating Reader scale with expected rating scale
reader = Reader(rating_scale = (0, 5))

# Loading the rating dataset
data = Dataset.load_from_df(df_final[['user_id', 'prod_id', 'rating']], reader)

# Splitting the data into train and test datasets
trainset, testset = train_test_split(data, test_size = 0.2, random_state = 42)
```

Now, we are **ready to build the first baseline similarity-based recommendation system** using the cosine similarity.

✓ Building the user-user Similarity-based Recommendation System


- Initialize the KNNBasic model using `sim_options` provided, `Verbose=False`, and setting `random_state=1`
- Fit the model on the training data
- Use the `precision_recall_at_k` function to calculate the metrics on the test data
- Provide your observations on the output

```
# Declaring the similarity options
sim_options = {'name': 'cosine',
               'user_based': True}

# Initialize the KNNBasic model using sim_options provided, Verbose = False, and setting random_state = 1
sim_user_user = KNNBasic(sim_options=sim_options, verbose=False, random=1)

# Fit the model on the training data
sim_user_user.fit(trainset)

# Let us compute precision@k, recall@k, and f_1 score using the precision_recall_at_k function defined above
precision_recall_at_k(sim_user_user)
```




```
RMSE: 1.0012
Precision: 0.855
Recall: 0.858
F_1 score: 0.856
```

The shows decent performance with high precision, recall, and F_1 scores. Balanced precision and recall shows its good at avoiding irrelevant recommendations, But with the RMSE at 1.0012 the models learning ability leaves room for improvement.

Let's now **predict rating for a user with** `userId=A3LDPF5FMB782Z` **and** `productId=1400501466` as shown below. Here the user has already interacted or watched the product with productId '1400501466' and given a rating of 5 which is denoted by the parameter `r_ui`.

```
# Predicting rating for a sample user with an interacted product
sim_user_user.predict("A3LDPF5FMB782Z", "1400501466", r_ui = 5, verbose = True)

 user: A3LDPF5FMB782Z item: 1400501466 r_ui = 5.00 est = 3.40 {'actual_k': 5, 'was_impossible': False}
Prediction(uid='A3LDPF5FMB782Z', iid='1400501466', r_ui=5, est=3.4, details={'actual_k': 5, 'was_impossible': False})
```

the user actually rated the product 5 and the model predicted it at 3.40. I believe we have to be mindful with such a positive bias with this data with over 75% of the ratings are 4 stars and higher. This cluster model prediction may not YET fully capture the nuances of this positive biased dataset.

Below is the function to find the **list of users who have not seen the product with product id "1400501466"**.

```
def n_users_not_interacted_with(n, data, prod_id):
    users_interacted_with_product = set(data[data['prod_id'] == prod_id]['user_id'])
    all_users = set(data['user_id'])
    return list(all_users.difference(users_interacted_with_product))[:n] # where n is the number of elements to get in the list

n_users_not_interacted_with(5, df_final, '1400501466')

→ ['A33ZYFE8XMKKR1',
   'AKSQNMIOU01H',
   'A26CPEEWB2WKRE',
   'A3BACUOZV1M0WM',
   'A21S26XYPGXJZX']
```

- It can be observed from the above list that **user "A2U0HALGF2X77Q" has not seen the product with productId "1400501466"** as this user id is a part of the above list.

Below we are predicting rating for userId=A2U0HALGF2X77Q and prod_id=1400501466 .

```
# Predicting rating for a sample user with a non interacted product
sim_user_user.predict("A2U0HALGF2X77Q", "1400501466", verbose = True)

→ user: A2U0HALGF2X77Q item: 1400501466 r_ui = None est = 5.00 {'actual_k': 1, 'was_impossible': False}
Prediction(uid='A2U0HALGF2X77Q', iid='1400501466', r_ui=None, est=5, details={'actual_k': 1, 'was_impossible': False})
```

With introduction of a user that hasn't used this product and with only 1 neighbor the model predicted the highest rating, keeping in touch with the positive bias of this dataset. This seems close to a cold-start scenario, at the current moment this might be better to present as a exploratory recommendation at best.

✓ Improving Similarity-based Recommendation System by tuning its hyperparameters

Below, we will be tuning hyperparameters for the KNNBasic algorithm. Let's try to understand some of the hyperparameters of the KNNBasic algorithm:

- **k** (int) – The (max) number of neighbors to take into account for aggregation. Default is 40.
- **min_k** (int) – The minimum number of neighbors to take into account for aggregation. If there are not enough neighbors, the prediction is set to the global mean of all ratings. Default is 1.
- **sim_options** (dict) – A dictionary of options for the similarity measure. And there are four similarity measures available in surprise -
 - cosine
 - msd (default)
 - Pearson
 - Pearson baseline

```
# Setting up parameter grid to tune the hyperparameters
param_grid = {'k': [20, 30, 40, 50],
              'min_k': [1, 2, 3],
              'sim_options':{'name': ['cosine', 'msd', 'pearson', 'pearson_baseline'],
                             'user_based': [True, False]}}

# Performing 3-fold cross-validation to tune the hyperparameters
gs = GridSearchCV(KNNBasic, param_grid, measures = ['rmse'], cv = 3, n_jobs = -1)

# Fitting the data
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'])

→ 0.9736636234080479
{'k': 50, 'min_k': 3, 'sim_options': {'name': 'cosine', 'user_based': True}}
```

Once the grid search is **complete**, we can get the **optimal values for each of those hyperparameters** as shown above.

Now, let's build the **final model by using tuned values of the hyperparameters**, which we received by using **grid search cross-validation**.

```
# Using the optimal similarity measure for user-user based collaborative filtering
sim_options = {'name': 'cosine', 'user_based': True}

# Creating an instance of KNNBasic with optimal hyperparameter values
sim_user_user_optimized = KNNBasic(sim_options = sim_options, k=50, min_k=3, random_state=1, verbose=False)

# Training the algorithm on the train set
sim_user_user_optimized.fit(trainset)

# Let us compute precision@k and recall@k also with k=10
precision_recall_at_k(sim_user_user_optimized)

RMSE: 0.9553
Precision: 0.862
Recall: 0.834
F_1 score: 0.848
```

Optimized KNNBasic model demonstrates improved performance, particularly in its ability to recall relevant items. However, the persistent high precision and recall in the context of a positively biased dataset suggest that while the model is functioning well, it might be benefiting from the skewed nature of the ratings.

Steps:

- **Predict rating for the user with** `userId="A3LDPF5FMB782Z"`, **and** `prod_id= "1400501466"` **using the optimized model**
- **Predict rating for** `userId="A2UOHALGF2X77Q"` **who has not interacted with** `prod_id = "1400501466"`, **by using the optimized model**
- **Compare the output with the output from the baseline model**

```
# Use sim_user_user_optimized model to recommend for userId "A3LDPF5FMB782Z" and productId 1400501466
sim_user_user_optimized.predict("A3LDPF5FMB782Z", "1400501466", verbose=True);
```

```
user: A3LDPF5FMB782Z item: 1400501466 r_ui = None est = 3.40 {'actual_k': 5, 'was_impossible': False}
```

```
# Use sim_user_user_optimized model to recommend for userId "A2UOHALGF2X77Q" and productId "1400501466"
sim_user_user_optimized.predict("A2UOHALGF2X77Q", "1400501466", verbose=True)
```

```
user: A2UOHALGF2X77Q item: 1400501466 r_ui = None est = 4.29 {'was_impossible': True, 'reason': 'Not enough neighbors.'}
Prediction(uid='A2UOHALGF2X77Q', iid='1400501466', r_ui=None, est=4.292024046561495, details={'was_impossible': True, 'reason': 'Not enough neighbors.'})
```

The Optimized model doesn't show improvement for users who have interacted with the product, For non-interacted items, the optimized model is more conservative, defaulting to the dataset average instead of making potentially overconfident predictions.

Identifying similar Users to a given User (nearest neighbors)

We can also find out **similar users to a given user** or its **nearest neighbors** based on this KNNBasic algorithm. Below, we are finding the 5 most similar users to the first user in the list with internal id 0, based on the `msd` distance metric.

```
# 0 is the inner id of the user
sim_user_user_optimized.get_neighbors(0, 5)
```

```
[6, 7, 17, 26, 32]
```

Implementing the recommendation algorithm based on optimized KNNBasic model

+ Code + Text

Below we will be implementing a function where the input parameters are:

- data: A **rating** dataset
- user_id: A user id **against which we want the recommendations**
- top_n: The **number of products we want to recommend**
- algo: the algorithm we want to use **for predicting the ratings**
- The output of the function is a **set of top_n items** recommended for the given user_id based on the given algorithm

```
def get_recommendations(data, user_id, top_n, algo):

    # Creating an empty list to store the recommended product ids
    recommendations = []

    # Creating an user item interactions matrix
    user_item_interactions_matrix = data.pivot(index = 'user_id', columns = 'prod_id', values = 'rating')

    # Extracting those product ids which the user_id has not interacted yet
    non_interacted_products = user_item_interactions_matrix.loc[user_id][user_item_interactions_matrix.loc[user_id].isnull()].index.tolist()

    # Looping through each of the product ids which user_id has not interacted yet
    for item_id in non_interacted_products:

        # Predicting the ratings for those non interacted product ids by this user
        est = algo.predict(user_id, item_id).est

        # Appending the predicted ratings
        recommendations.append((item_id, est))

    # Sorting the predicted ratings in descending order
    recommendations.sort(key = lambda x: x[1], reverse = True)

    return recommendations[:top_n] # Returing top n highest predicted rating products for this user
```

Predicting top 5 products for userID = "A3LDPF5FMB782Z" with similarity based recommendation system

```
# Making top 5 recommendations for user_id "A3LDPF5FMB782Z" with a similarity-based recommendation engine
recommendations = get_recommendations(df_final, "A3LDPF5FMB782Z", 5, sim_user_user_optimized)

# Building the dataframe for above recommendations with columns "prod_id" and "predicted_ratings"
pd.DataFrame(recommendations, columns = ['prod_id', 'predicted_ratings'])
```

```
↗
```

	prod_id	predicted_ratings
0	B00005LENO	5
1	B000067RT6	5
2	B00006HSML	5
3	B00006I53X	5
4	B00006I5J7	5

For user-user Collaborative Filtering, The consistent 5-star predictions align with the strong positive bias we've observed in the dataset (where 75% of ratings are 4 stars and higher).

✓ Item-Item Similarity-based Collaborative Filtering Recommendation System

- Above we have seen **similarity-based collaborative filtering** where similarity is calculated **between users**. Now let us look into similarity-based collaborative filtering where similarity is seen **between items**.

```
# Declaring the similarity options
sim_options = {'name': 'cosine',
               'user_based': False}

# KNN algorithm is used to find desired similar items
sim_item_item = KNNBasic(sim_options=sim_options, random_state = 1, verbose = False)

# Train the algorithm on the train set, and predict ratings for the test set
sim_item_item.fit(trainset)

# Let us compute precision@k, recall@k, and f_1 score with k = 10
precision_recall_at_k(sim_item_item)
```

```
↗
```

```
RMSE: 0.9950
Precision: 0.829
Recall: 0.759
F_1 score: 0.792
```

The item-item approach shows lower performance across all metrics compared to both the baseline and optimized user-user models. Despite the decrease, the precision and recall values are still relatively high, which likely reflects the positive bias

Let's now **predict a rating for a user with** `userId = A3LDPF5FMB782Z` and `prod_id = 1400501466`. Here the user has already interacted or watched the product with productId "1400501466".

```
# Predicting rating for a sample user with an interacted product
sim_item_item.predict("A3LDPF5FMB782Z", "1400501466", verbose = True)

user: A3LDPF5FMB782Z item: 1400501466 r_ui = None est = 4.27 {'actual_k': 22, 'was_impossible': False}
Prediction(uid='A3LDPF5FMB782Z', iid='1400501466', r_ui=None, est=4.2727272727272725, details={'actual_k': 22, 'was_impossible': False})
```

The model used 22 similar items to make the prediction, which is a substantial number and suggests a more robust prediction. The prediction for A3LDPF5FMB782Z seems more reliable due to the higher number of neighbors used.

Below we are **predicting rating for the** `userId = A2U0HALGF2X77Q` and `prod_id = 1400501466`.

```
# Predicting rating for a sample user with a non interacted product
sim_item_item.predict("A2U0HALGF2X77Q", "1400501466", verbose = True)

user: A2U0HALGF2X77Q item: 1400501466 r_ui = None est = 4.00 {'actual_k': 1, 'was_impossible': False}
Prediction(uid='A2U0HALGF2X77Q', iid='1400501466', r_ui=None, est=4.0, details={'actual_k': 1, 'was_impossible': False})
```

Only 1 similar item was used, which might lead to a less reliable prediction. But the model still made a prediction based on a single similar item, unlike the user-user model which defaulted to the global mean.

✓ Hyperparameter tuning the item-item similarity-based model

- Use the following values for the `param_grid` and tune the model
 - `'k': [10, 20, 30]`
 - `'min_k': [3, 6, 9]`
 - `'sim_options': {'name': ['msd', 'cosine']}`
 - `'user_based': [False]`
- Use `GridSearchCV()` to tune the model using the 'rmse' measure
- Print the best score and best parameters

```
# Setting up parameter grid to tune the hyperparameters
param_grid = {
    'k': [10, 20, 30],
    'min_k': [3, 6, 9],
    'sim_options': {'name': ['msd', 'cosine'],
                    'user_based': [False]},
}

# Performing 3-fold cross validation to tune the hyperparameters
gs = GridSearchCV(KNNBasic, param_grid, measures=['rmse'], cv=3, n_jobs=-1)

# Fitting the data
gs.fit(data)

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

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

0.975490454519289
{'k': 30, 'min_k': 6, 'sim_options': {'name': 'msd', 'user_based': False}}
```

Once the **grid search** is complete, we can get the **optimal values for each of those hyperparameters as shown above**.

Now let's build the **final model** by using **tuned values of the hyperparameters** which we received by using grid search cross-validation.

```
# Using the optimal similarity measure for item-item based collaborative filtering
sim_options = {'name': 'msd', 'user_based': False}

# Creating an instance of KNNBasic with optimal hyperparameter values
sim_item_item_optimized = KNNBasic(sim_options=sim_options, k=30, min_k=6, random_state = 1, verbose = False)

# Training the algorithm on the train set
sim_item_item_optimized.fit(trainset)

# Let us compute precision@k and recall@k, f1_score and RMSE
precision_recall_at_k(sim_item_item_optimized)

RMSE: 0.9576
Precision: 0.839
Recall: 0.88
F1 score: 0.859
```

The optimized item-item model shows significant improvements in RMSE, recall, and F1 score compared to the baseline item-item model. Performance is now comparable to the user-user model

Steps:

- Predict rating for the user with `userId="A3LDPF5FMB782Z"`, and `prod_id= "1400501466"` using the optimized model
- Predict rating for `userId="A2UOHALGF2X77Q"` who has not interacted with `prod_id = "1400501466"`, by using the optimized model
- Compare the output with the output from the baseline model

```
# Use sim_item_item_optimized model to recommend for userId "A3LDPF5FMB782Z" and productId "1400501466"
sim_item_item_optimized.predict("A3LDPF5FMB782Z", "1400501466", verbose=True)

user: A3LDPF5FMB782Z item: 1400501466 r_ui = None est = 4.67 {'actual_k': 22, 'was_impossible': False}
Prediction(uid='A3LDPF5FMB782Z', iid='1400501466', r_ui=None, est=4.67427701674277, details={'actual_k': 22, 'was_impossible': False})

# Use sim_item_item_optimized model to recommend for userId "A2UOHALGF2X77Q" and productId "1400501466"
sim_item_item_optimized.predict("A2UOHALGF2X77Q", "1400501466", verbose=True)

user: A2UOHALGF2X77Q item: 1400501466 r_ui = None est = 4.29 {'was_impossible': True, 'reason': 'Not enough neighbors.'}
Prediction(uid='A2UOHALGF2X77Q', iid='1400501466', r_ui=None, est=4.292024046561495, details={'was_impossible': True, 'reason': 'Not enough neighbors.'})
```

The higher prediction compared to the baseline model (4.67 vs 4.27) indicates that the optimization has led to more confidence in high ratings, And for who have not interacted it shows a more conservative approach compared to the baseline model, preferring to default to the global mean rather than make a potentially unreliable prediction because `was_impossible: True` reason: "Not enough neighbors."

Identifying similar items to a given item (nearest neighbors)

We can also find out **similar items** to a given item or its nearest neighbors based on this **KNNBasic algorithm**. Below we are finding the 5 most similar items to the item with internal id 0 based on the `msd` distance metric.


```
sim_item_item_optimized.get_neighbors(0, 5)

[29, 53, 67, 106, 151]
```

Predicting top 5 products for `userId = "A1A5KUIIIHFF4U"` with similarity based recommendation system.

```
# Making top 5 recommendations for user_id A1A5KUIIIHFF4U with similarity-based recommendation engine.
recommendations = get_recommendations(df_final, "A1A5KUIIIHFF4U", 5, sim_item_item_optimized)

# Building the dataframe for above recommendations with columns "prod_id" and "predicted_ratings"
pd.DataFrame(recommendations, columns = ['prod_id', 'predicted_ratings'])
```



	prod_id	predicted_ratings
0	1400532655	4.292024
1	1400599997	4.292024
2	9983891212	4.292024
3	B00000DM9W	4.292024
4	B00000J1V5	4.292024

This value is very close to the dataset's average rating of ~4.29

Now as we have seen **similarity-based collaborative filtering algorithms**, let us now get into **model-based collaborative filtering algorithms**.

✓ Model 3: Matrix Factorization

Model-based Collaborative Filtering is a **personalized recommendation system**, the recommendations are based on the past behavior of the user and it is not dependent on any additional information. We use **latent features** to find recommendations for each user.


✓ Singular Value Decomposition (SVD)

SVD is used to **compute the latent features** from the **user-item matrix**. But SVD does not work when we **miss values** in the **user-item matrix**.

```
# Using SVD matrix factorization. Use random_state = 1
svd = SVD(random_state = 1)

# Training the algorithm on the train set
svd.fit(trainset)

# Use the function precision_recall_at_k to compute precision@k, recall@k, F1-Score, and RMSE
precision_recall_at_k(svd)
```




```
RMSE: 0.8882
Precision: 0.853
Recall: 0.88
F_1 score: 0.866
```

SVD emerges as the best-performing model among those we've tested, providing a good balance of accuracy (low RMSE) and relevance (high F1 score).

Let's now predict the rating for a user with `userId = "A3LDPF5FMB782Z"` **and** `prod_id = "1400501466"`.

```
# Making prediction
svd.predict("A3LDPF5FMB782Z", "1400501466", r_ui = 5, verbose = True)
```




```
user: A3LDPF5FMB782Z item: 1400501466 r_ui = 5.00 est = 4.08 {'was_impossible': False}
Prediction(uid='A3LDPF5FMB782Z', iid='1400501466', r_ui=5, est=4.081406749810685, details={'was_impossible': False})
```

This is closer to the actual rating compared to the user-user model (which predicted 3.40) but less than the item-item model (which predicted 4.67)

Below we are predicting rating for the `userId = "A2U0HALGF2X77Q"` **and** `productId = "1400501466"`.

```
# Making prediction
svd.predict("A2U0HALGF2X77Q", "1400501466", verbose = True)
```



```
user: A2U0HALGF2X77Q item: 1400501466 r_ui = None est = 4.16 {'was_impossible': False}
Prediction(uid='A2U0HALGF2X77Q', iid='1400501466', r_ui=None, est=4.156510633154892, details={'was_impossible': False})
```

Unlike the neighborhood-based models which defaulted to the global mean for this non-interacted item, the SVD model provides a unique prediction.

✓ Improving Matrix Factorization based recommendation system by tuning its hyperparameters

Below we will be tuning only three hyperparameters:

- **n_epochs**: The number of iterations of the SGD algorithm.
- **lr_all**: The learning rate for all parameters.
- **reg_all**: The regularization term for all parameters.

```
# Set the parameter space to tune
param_grid = {'n_epochs': [10, 20, 30], 'lr_all': [0.001, 0.005, 0.01],
              'reg_all': [0.2, 0.4, 0.6]}

# Performing 3-fold gridsearch cross-validation
gs_ = GridSearchCV(SVD, param_grid, measures = ['rmse'], cv = 3, n_jobs = -1)

# Fitting data
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'])

0.8987448095773636
{'n_epochs': 20, 'lr_all': 0.01, 'reg_all': 0.2}
```

Now, we will **the build final model** by using **tuned values** of the hyperparameters, which we received using grid search cross-validation above.

```
# Build the optimized SVD model using optimal hyperparameter search. Use random_state = 1
svd_optimized = SVD(n_epochs=20, lr_all=0.01, reg_all=0.2, random_state=1)

# Train the algorithm on the train set
svd_optimized.fit(trainset)

# Use the function precision_recall_at_k to compute precision@k, recall@k, F1-Score, and RMSE
precision_recall_at_k(svd_optimized)

RMSE: 0.8808
Precision: 0.854
Recall: 0.878
F_1 score: 0.866
```

The improved RMSE, combined with maintained high precision and recall, suggests that users are likely to receive more accurate and relevant recommendations.

Let's now predict a rating for a user with `userId = "A3LDPF5FMB782Z"` and `productId = "1400501466"` with the optimized model.

✓ Steps:

- **Predict rating for the user with `userId="A3LDPF5FMB782Z"`, and `prod_id= "1400501466"` using the optimized model**
- **Predict rating for `userId="A2U0HALGF2X77Q"` who has not interacted with `prod_id ="1400501466"`, by using the optimized model**
- **Compare the output with the output from the baseline model**

```
# Use svd_algo_optimized model to recommend for userId "A3LDPF5FMB782Z" and productId "1400501466"
svd_optimized.predict("A3LDPF5FMB782Z", "1400501466", r_ui = 5, verbose = True)

user: A3LDPF5FMB782Z item: 1400501466 r_ui = 5.00 est = 4.13 {'was_impossible': False}
Prediction(uid='A3LDPF5FMB782Z', iid='1400501466', r_ui=5, est=4.128589011282042, details={'was_impossible': False})

# Use svd_algo_optimized model to recommend for userId "A2U0HALGF2X77Q" and productId "1400501466"
svd_optimized.predict("A2U0HALGF2X77Q", "1400501466", verbose = True)

user: A2U0HALGF2X77Q item: 1400501466 r_ui = None est = 4.11 {'was_impossible': False}
Prediction(uid='A2U0HALGF2X77Q', iid='1400501466', r_ui=None, est=4.105349444662267, details={'was_impossible': False})
```

The optimized SVD model shows improved performance over both the baseline SVD and the neighborhood-based models. It provides consistent, personalized predictions that reflect the dataset's characteristics while avoiding extreme values.

✓ Conclusion and Recommendations

Write your conclusion and recommendations here

Throughout this project, we've explored various collaborative filtering approaches to build a recommendation system for Amazon product ratings. We started with neighborhood-based methods (user-user and item-item) and progressed to matrix factorization using SVD. Each step of this journey has provided valuable insights into the nature of our dataset and the strengths and weaknesses of different recommendation techniques.

One of the most striking characteristics of our dataset was the strong positive bias, with an average rating of about 4.29 and 75% of ratings being 4 stars or higher. Initially, we were concerned that this bias might skew our recommendations, but through curiosity increased the relevance threshold to 4.5 and that drastically reduced the models performance, so i returned it to 3.5 and realized that the positive bias is an intrinsic feature of user behavior. These models provided a solid baseline but struggled with cold start problems and sometimes defaulted to global means for users or items with limited interactions. The optimization of these models improved their performance, but they still had limitations in handling sparse data.

The introduction of the SVD model marked a significant improvement. Its ability to capture latent features allowed for more nuanced and personalized recommendations, even for user-item pairs without direct interactions. The baseline SVD model outperformed the neighborhood-based approaches, and with hyperparameter tuning, we achieved even better results.

Recommendations:

Implement the Optimized SVD Model: Given its superior performance and ability to handle various scenarios, I recommend implementing the optimized SVD model as your primary recommendation engine.