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 lead 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

Note: The code has some user defined functions that will be usefull while making recommendations and measure model performance, you can use these functions or can create your own functions.

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.

In [139…
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.m
ount("/content/drive", force_remount=True).
```

Installing surprise library

In [140…
```
!pip install surprise
```

```
Requirement already satisfied: surprise in /usr/local/lib/python3.10/dist-packages
(0.1)
Requirement already satisfied: scikit-surprise in /usr/local/lib/python3.10/dist-pack
ages (from surprise) (1.1.3)
Requirement already satisfied: joblib>=1.0.0 in /usr/local/lib/python3.10/dist-packag
es (from scikit-surprise->surprise) (1.3.2)
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.10/dist-packag
es (from scikit-surprise->surprise) (1.23.5)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.10/dist-package
s (from scikit-surprise->surprise) (1.11.3)
```

Importing the necessary libraries and overview of the dataset

In [141…
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from collections import defaultdict
from sklearn.metrics import mean_squared_error
```

Loading the data

- Import the Dataset
- Add column names ['user_id', 'prod_id', 'rating', 'timestamp']
- Drop the column timestamp
- Copy the data to another DataFrame called **df**

In [143…

```
df = pd.read_csv('/content/drive/MyDrive/MIT Data Science & AI Course Notes/Week 9: Re
df.columns = ['user_id', 'prod_id', 'rating', 'timestamp']
df.drop('timestamp')
df_copy = df.copy(deep = True)
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-143-b4f1a112df99> in <cell line: 3>()
      1 df = pd.read_csv('/content/drive/MyDrive/MIT Data Science & AI Course Notes/W
eek 9: Recommendation Systems Project/ratings_Electronics.csv')
      2 df.columns = ['user_id', 'prod_id', 'rating', 'timestamp']
----> 3 df.drop('timestamp')
      4 df_copy = df_copy(deep = True)

/usr/local/lib/python3.10/dist-packages/pandas/util/_decorators.py in wrapper(*args,
**kwargs)
    329                     stacklevel=find_stack_level(),
    330                 )
--> 331             return func(*args, **kwargs)
    332 
    333         # error: "Callable[[VarArg(Any), KwArg(Any)], Any]" has no

/usr/local/lib/python3.10/dist-packages/pandas/core/frame.py in drop(self, labels, ax
is, index, columns, level, inplace, errors)
   5397                 weight  1.0     0.8
   5398         """
-> 5399         return super().drop(
   5400             labels=labels,
   5401             axis=axis,

/usr/local/lib/python3.10/dist-packages/pandas/util/_decorators.py in wrapper(*args,
**kwargs)
    329                     stacklevel=find_stack_level(),
    330                 )
--> 331             return func(*args, **kwargs)
    332 
    333         # error: "Callable[[VarArg(Any), KwArg(Any)], Any]" has no

/usr/local/lib/python3.10/dist-packages/pandas/core/generic.py in drop(self, labels,
axis, index, columns, level, inplace, errors)
   4503         for axis, labels in axes.items():
   4504             if labels is not None:
-> 4505                 obj = obj._drop_axis(labels, axis, level=level, errors=error
s)
   4506 
   4507         if inplace:

/usr/local/lib/python3.10/dist-packages/pandas/core/generic.py in _drop_axis(self, la
bels, axis, level, errors, only_slice)
   4544                 new_axis = axis.drop(labels, level=level, errors=errors)
   4545             else:
-> 4546                 new_axis = axis.drop(labels, errors=errors)
   4547             indexer = axis.get_indexer(new_axis)
   4548 

/usr/local/lib/python3.10/dist-packages/pandas/core/indexes/base.py in drop(self, lab
els, errors)
   6932         if mask.any():
   6933             if errors != "ignore":
-> 6934                 raise KeyError(f"{list(labels[mask])} not found in axis")
   6935             indexer = indexer[~mask]
   6936         return self.delete(indexer)

KeyError: "['timestamp'] not found in axis"
```

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.

In [144…

```
# 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
```

In [145…

```
# 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)]
```

In [146…

```
# 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
```

In [147…

```
# 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)]
df_final
```

Out[147]:

	user_id	prod_id	rating	timestamp
1309	A3LDPF5FMB782Z	1400501466	5.0	1336003200
1321	A1A5KUIIIHFF4U	1400501466	1.0	1332547200
1334	A2XIOXRRYX0KZY	1400501466	3.0	1371686400
1450	AW3LX47IHPFRL	1400501466	5.0	1339804800
1455	A1E3OB6QMBKRYZ	1400501466	1.0	1350086400
...	...	...	...	...
7824422	A34BZM6S9L7QI4	B00LGQ6HL8	5.0	1405555200
7824423	A1G650TTTHEAL5	B00LGQ6HL8	5.0	1405382400
7824424	A25C2M3QF9G7OQ	B00LGQ6HL8	5.0	1405555200
7824425	A1E1LEVQ9VQNK	B00LGQ6HL8	5.0	1405641600
7824426	A2NYK9KWFMJV4Y	B00LGQ6HL8	5.0	1405209600

65290 rows × 4 columns

In [149…

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

Out[149]:

	user_id	prod_id	rating	timestamp
1309	A3LDPF5FMB782Z	1400501466	5.0	1336003200
1321	A1A5KUIIIHFF4U	1400501466	1.0	1332547200
1334	A2XIOXRRYX0KZY	1400501466	3.0	1371686400
1450	AW3LX47IHPFRL	1400501466	5.0	1339804800
1455	A1E3OB6QMBKRYZ	1400501466	1.0	1350086400

Exploratory Data Analysis

Shape of the data

Check the number of rows and columns and provide observations.

In [150…

```
# Check the number of rows and columns and provide observations
df_final.shape
```

Out[150]:
```
(65290, 4)
```

Write your observations here: There are 65290 rows and 4 columns

Data types

In [151...
```
# Check Data types and provide observations
df_final.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 65290 entries, 1309 to 7824426
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   user_id    65290 non-null  object 
 1   prod_id    65290 non-null  object 
 2   rating     65290 non-null  float64
 3   timestamp  65290 non-null  int64  
dtypes: float64(1), int64(1), object(2)
memory usage: 2.5+ MB
```

Write your observations here:__ The user_id and prod_id columns are currently objects, while the rating column is made up of floats

Checking for missing values

In [152...
```
# Check for missing values present and provide observations
df_final.isna().sum()
```

Out[152]:
```
user_id      0
prod_id      0
rating       0
timestamp    0
dtype: int64
```

Write your observations here:__ There don't seem to be any missing values in the data

Summary Statistics

In [153...
```
# Summary statistics of 'rating' variable and provide observations
df_final['rating'].describe()
```

Out[153]:
```
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
```

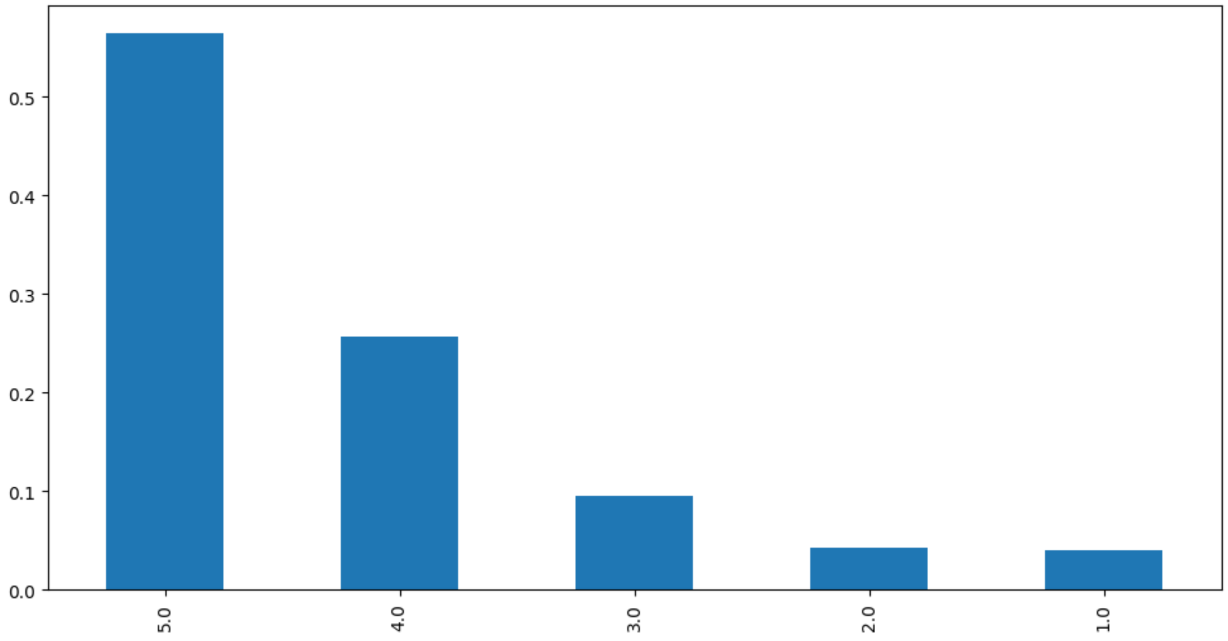
Write your observations here:__ The average rating is ~4.29, with the standard deviation being a little under 1 (0.988). It's clear that most ratings are 5's and 4's by looking at the 50% and 75%

distributions. Because our mean is lower than the median and our upper quartile and max equal to the median, our data is left skewed.

Checking the rating distribution

In [154…

```
# Create the bar plot and provide observations
plt.figure(figsize = (12, 6))
df['rating'].value_counts(1).plot(kind = 'bar')
plt.show()
```



Write your observations here:__ This confirms our earlier observations by looking at df.describe(). The vast majority of the ratings given are 5 stars (over 50%) with 4 star ratings being second at about 20%. 2 and 3 star ratings are the least common, both below 10%. This tells us the distribution is left skewed.

Checking the number of unique users and items in the dataset

In [155…

```
# Number of total rows in the data and number of unique user id and product id in the
print('Number of unique users: ', df_final['user_id'].nunique())
print('Number of unique products: ', df_final['prod_id'].nunique())
```

```
Number of unique users:  1540
Number of unique products:  5689
```

Write your observations here:___ There are 1,540 unique users, and 5,689 unique products.

Users with the most number of ratings

In [156…

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

Out[156]:
```
user_id
ADLVFFE4VBT8      295
A3OXHLG6DIBRW8    230
A1ODOGXEYECQQ8    217
A36K2N527TXXJN    212
A25C2M3QF9G7OQ    203
A680RUE1FDO8B     196
A22CW0ZHY3NJH8    193
A1UQBFCERIP7VJ    193
AWPODHOB4GFWL     184
A3LGT6UZL99IW1    179
dtype: int64
```

Write your observations here:___ The highest number of ratings by a user is 295 ratings, but we have way more unique products in the data, so we'll be able to successfully build a model to recommend new products to this users and all the others.

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

Model 1: Rank Based Recommendation System

In [157…
```python
# Calculate the average rating for each product
average_rating = df_final.groupby('prod_id').mean()['rating']
# Calculate the count of ratings for each product
rating_count = df_final.groupby('prod_id').count()['rating']
# Create a dataframe with calculated average and count of ratings
final_rating = pd.DataFrame({'average_rating':average_rating, 'rating_cc
# Sort the dataframe by average of ratings in the descending order
final_rating = final_rating.sort_values(by='average_rating', ascending=False)

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

```
<ipython-input-157-c1faef6baf60>:2: FutureWarning: The default value of numeric_only
in DataFrameGroupBy.mean is deprecated. In a future version, numeric_only will defaul
t to False. Either specify numeric_only or select only columns which should be valid
for the function.
  average_rating = df_final.groupby('prod_id').mean()['rating']
```

Out[157]:

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

In [158…
```python
# Defining a function to get the top n products based on the highest average rating ar
def get_top_n_products(final_rating, n, min_interaction):
```

```
# Finding products with minimum number of interactions
  recommendations = final_rating[final_rating['rating_count'] > 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

In [159…

```
list(get_top_n_products(final_rating, 5, 50))
```

Out[159]:

```
['B001TH7GUU', 'B003ES5ZUU', 'B0019EHU8G', 'B006W8U2MU', 'B000QUUFRW']
```

Recommending top 5 products with 100 minimum interactions based on popularity

In [160…

```
list(get_top_n_products(final_rating, 5, 100))
top_5_100_products = get_top_n_products(final_rating, 5, 50)
```

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

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.

In [161…

```
# 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
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.

In [162…

```
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 Precisi

        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

        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
```

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`**.**

In [163...

```
# 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

In [164...

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

# Initialize the KNNBasic model using sim_options declared, Verbose = False, and setti
```

```
algo_knn_user = KNNBasic(sim_options=sim_options, verbose=False, random_state = 42)

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

# Let us compute precision@k, recall@k, and f_1 score using the precision_recall_at_k
precision_recall_at_k(algo_knn_user)
```

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

Write your observations here:__ We want to minimize false positives (products recommended as popular with similar users but aren't, and false negatives (products predicted as unpopular with similar users that are popular). We want to **maximize the F_1 score, and it's already high at about 0.856**. Recall is also rather high, at 85.8%. This indicates that among all relevant products, **85.8% of them are correctly recommended**. Precision is also **high at 85.5%**, indicating that a high number of our predictions are **true positives**.

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.

In [165…
```
# Predicting rating for a sample user with an interacted product
algo_knn_user.predict('A3LDPF5FMB782Z', '1400501466', r_ui = 5, verbose = True)
df_final.max()
```

```
user: A3LDPF5FMB782Z item: 1400501466 r_ui = 5.00   est = 3.40   {'actual_k': 5, 'was
_impossible': False}
```
Out[165]:
```
user_id      AZOK5STV85FBJ
prod_id         B00LGQ6HL8
rating                 5.0
timestamp       1406073600
dtype: object
```

Write your observations here:__ The prediction for the user's rating for this product is 3.40, and the actual rating is 5.

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

In [166…
```
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
```

In [167…
```
# Find unique user_id where prod_id is not equal to "1400501466"
n_users_not_interacted_with(5, df_final, '1400501466')
```

Out[167]:
```
['A1WG97A0EFHYXN',
 'A3S3R88HA0HZG3',
 'A2BMZRO0H7TFCS',
 'A2EN82VBJT44QP',
 'A3TPNC3TKGCCEI']
```

- It can be observed from the above list that **user "A2UOHALGF2X77Q" 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=A2UOHALGF2X77Q` and `prod_id=1400501466`.

In [168…

```
# Predicting rating for a sample user with a non interacted product
algo_knn_user.predict('A2UOHALGF2X77Q', '1400501466', r_ui = 5, verbose = True)
```

```
user: A2UOHALGF2X77Q item: 1400501466 r_ui = 5.00   est = 5.00   {'actual_k': 1, 'was
_impossible': False}
```

Out[168]:

```
Prediction(uid='A2UOHALGF2X77Q', iid='1400501466', r_ui=5, est=5, details={'actual_
k': 1, 'was_impossible': False})
```

Write your observations here:__ Here, the predicted rating for this user-item pair is 5 based on this user-user similarity model.

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

In [169…

```
# Setting up parameter grid to tune the hyperparameters
param_grid = {'k': [10, 20, 30], 'min_k': [3, 6, 9],
              'sim_options': {'name': ['msd', 'cosine', 'pearson', 'pearson_baseline']
                              'user_based': [True], 'min_support':[2, 4]}}
# 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'])
```

```
0.9809803791744282
{'k': 20, 'min_k': 6, 'sim_options': {'name': 'msd', 'user_based': True, 'min_suppor
t': 2}}
```

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

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

In [170…

```
# Using the optimal similarity measure for user-user based collaborative filtering
sim_options = {'name': 'cosine',
               'user_based': True, 'min_support': 2}
# Creating an instance of KNNBasic with optimal hyperparameter values
sim_user_user_optimized = KNNBasic(sim_options = sim_options, k = 30, min_k = 3, rando
# Training the algorithm on the trainset
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: 0.9616
Precision:  0.848
Recall:  0.891
F_1 score:  0.869
```

Write your observations here:__ Precision is a little lower, but recall's noticeably higher. The F_1 score is also slightly higher, and the RMSE is lower, which confirms the improvement.

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**

In [171…

```
# Use sim_user_user_optimized model to recommend for userId "A3LDPF5FMB782Z" and produ
sim_user_user_optimized.predict('A3LDPF5FMB782Z', '1400501466', r_ui = 5, verbose = Tr
```

```
user: A3LDPF5FMB782Z item: 1400501466 r_ui = 5.00   est = 4.29   {'was_impossible': T
rue, 'reason': 'Not enough neighbors.'}
```

Out[171]:

```
Prediction(uid='A3LDPF5FMB782Z', iid='1400501466', r_ui=5, est=4.292024046561495, det
ails={'was_impossible': True, 'reason': 'Not enough neighbors.'})
```

In [172…

```
# Use sim_user_user_optimized model to recommend for userId "A2UOHALGF2X77Q" and produ
sim_user_user_optimized.predict('A2UOHALGF2X77Q', '1400501466', r_ui = 5, verbose = Tr
```

```
user: A2UOHALGF2X77Q item: 1400501466 r_ui = 5.00   est = 4.29   {'was_impossible': T
rue, 'reason': 'Not enough neighbors.'}
```

Out[172]:

```
Prediction(uid='A2UOHALGF2X77Q', iid='1400501466', r_ui=5, est=4.292024046561495, det
ails={'was_impossible': True, 'reason': 'Not enough neighbors.'})
```

Write your observations here:__ In both cases the model says that there weren't enough neighbors, and defaulted to the mean which was at 4.29.

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.

In [173…
```
# 0 is the inner id of the above user
sim_user_user_optimized.get_neighbors(0, 5)
```

Out[173]:
```
[54, 71, 94, 105, 113]
```

Implementing the recommendation algorithm based on optimized KNNBasic model

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

In [174…
```
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',

    # Extracting those product ids which the user_id has not interacted yet
    non_interacted_products = user_item_interactions_matrix.loc[user_id][user_item_int

    # 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
```

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

In [175…
```
# Making top 5 recommendations for user_id "A3LDPF5FMB782Z" with a similarity-based re
recommendations = get_recommendations(df_final, 'A2UOHALGF2X77Q', 5, algo_knn_user)
```

In [176…
```
# Building the dataframe for above recommendations with columns "prod_id" and "predict
pd.DataFrame(recommendations, columns = ['prod_id', 'predicted_ratings'])
```

Out[176]:

	prod_id	predicted_ratings
0	1400501466	5
1	B00000DM9W	5
2	B00000JDF6	5
3	B00000K4KH	5
4	B00001ZWXA	5

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**.

In [ ]:
```
# Declaring the similarity options
sim_options = {'name': 'cosine',
               'user_based': False
}
# KNN algorithm is used to find desired similar items. Use random_state=1
sim_item_item = KNNBasic(sim_options = sim_options, random_state = 1, verbose = False)
# Train the algorithm on the trainset, 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)
```

Write your observations here:__ We have high precision at 83.8%, indicating a lot of predictions are true positives. F_1 score is high 0.84, which indicates this model performs well. Recall is also good at 84.5%

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

In [ ]:
```
# Predicting rating for a sample user with an interacted product
sim_item_item.predict('A3LDPF5FMB782Z', '1400501466', r_ui = 5, verbose=True)
```

Write your observations here:__ The estimate is close, it predicts the rating to be at 4.27 when the actual rating was 5.

Below we are **predicting rating for the** `userId = A2UOHALGF2X77Q` **and** `prod_id = 1400501466`.

In [ ]:
```
# Predicting rating for a sample user with a non interacted product
sim_item_item.predict('A2UOHALGF2X77Q', '1400501466', r_ui = 5, verbose=True)
```

In this case, the model predicts a rating of 4.0 for the product the user hasn't seen.

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

In [ ]:
```
# 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], 'min_support': [2,4]}}
# 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'])
```

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.

Use the best parameters from GridSearchCV to build the optimized item-item similarity-based model. Compare the performance of the optimized model with the baseline model.

In [ ]:
```
# Using the optimal similarity measure for item-item based collaborative filtering
sim_options = {'name': 'msd',
               'user_based': False, 'min_support': 2}
# Creating an instance of KNNBasic with optimal hyperparameter values
sim_item_item_optimized = KNNBasic(sim_options=sim_options, k = 20, min_k = 6, random_
# Training the algorithm on the trainset
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)
```

Write your observations here:__ All metrics have improved with the exception of precision, which has lowered only slightly.

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**

In [ ]:
```
# Use sim_item_item_optimized model to recommend for userId "A3LDPF5FMB782Z" and produ
sim_item_item_optimized.predict('A3LDPF5FMB782Z', '1400501466', r_ui = 5, verbose = Tr
```

In [ ]:
```
# Use sim_item_item_optimized model to recommend for userId "A2UOHALGF2X77Q" and produ
sim_item_item_optimized.predict('A2UOHALGF2X77Q', '1400501466', r_ui = 5, verbose = Tr
```

Write your observations here:__ The model assigns the global mean of 4.3 for these predictions as there are 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.

In [ ]:
```
sim_item_item_optimized.get_neighbors(0, 5)
```

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

Hint: Use the get_recommendations() function.

In [ ]:
```
# Making top 5 recommendations for user_id A1A5KUIIIHFF4U with similarity-based recomm
recommendations = get_recommendations(df_final, 'A1A5KUIIIHFF4U', 5, sim_item_item)
```

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

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

Model 3: Model-Based Collaborative Filtering - 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**.

In [ ]:
```
# Using SVD matrix factorization. Use random_state = 1
svd = SVD(random_state = 1)
# Training the algorithm on the trainset
svd.fit(trainset)
# Use the function precision_recall_at_k to compute precision@k, recall@k, F1-Score, a
precision_recall_at_k(svd)
```

Write your observations here:___ This model performs well across the board, with RMSE being the lowest of any we've seen, and the recall being one of the highest at 88%

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

In [ ]:
```
# Making prediction
svd.predict('A3LDPF5FMB782Z', '1400501466', r_ui = 5, verbose = True)
```

Write your observations here:___ The model predicts 4.08

Below we are predicting rating for the `userId = "A2UOHALGF2X77Q"` and `productId = "1400501466"`.

In [ ]:
```
# Making prediction
svd.predict('A2UOHALGF2X77Q', '1400501466', r_ui = 5, verbose = True)
```

Write your observations here:___ The model predicts a rating of 4.16.

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.

In [ ]:
```
# Set the parameter space to tune
param_grid = {
    'n_epochs': [10, 20, 30],
    'lr_all': [0.001, 0.005, 0.01],
    'reg_all': [0.02, 0.1, 0.2, 0.5]
}
# 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'])
```

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

In [ ]:
```
# Build the optimized SVD model using optimal hyperparameter search. Use random_state=
svd_optimized = SVD(n_epochs = 20, lr_all = 0.01, reg_all = 0.2, random_state = 1)
# Train the algorithm on the trainset
svd_optimized.fit(trainset)
# Use the function precision_recall_at_k to compute precision@k, recall@k, F1-Score, a
precision_recall_at_k(svd_optimized)
```

Write your observations here:_ The model has slightly improved in all areas except for the F1 score. RMSE is slightly lower, precision is slightly higher, recall is actually a little lowere, and the F1 score is unchanged.

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**

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

In [ ]:
```
# Use svd_algo_optimized model to recommend for userId "A2UOHALGF2X77Q" and productId
svd_optimized.predict('A2UOHALGF2X77Q', '1400501466', r_ui = 5, verbose = True)
```

The model predicts slightly higher values at 4.13 and 4.11 (for an unseen product).

Conclusion and Recommendations

Write your conclusion and recommendations here

We used:

- Rank-Based Recommendation Model using Averages
- User-User-Similarity Based Collaborative Filtering
- Item-Item-Similarity Based Collaborative Filtering
- Model-based (matrix factorization) collaborative filtering

We didn't use clustering-based recommendation. SVD's great performance may be in part due to an ability to account for latent factors with matrix factorization.

Although all the models did well, here are the final results:

- Optimized SVD has the lowest RMSE by a significant margin
- User-User collaborative filtering has the highest precision, with Optimized SVD being a close second.
- Optimized User-User collaborative filtering has the highest F1 score, again with SVD being a close second
- Optimized User-User has the highest precision, again with optimized SVD being a close second.

We will go with **Optimized User-User collaborative filtering** because it performed well and has the highest F1 score, but we could just as easily go with **Optimized SVD** because it had great performance in all of the four metrics.

- It could be worth looking into a clustering-based recommendation model, and also potentially a content-based model.