Project: Amazon Product Recommendation System

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.

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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=Tr
```

Installing surprise library

```
!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-packages (from surprise) (1.1.3)
Requirement already satisfied: joblib>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-surprise->surprise) (1.
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.10/dist-packages (from scikit-surprise->surprise) (1.
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.10/dist-packages (from scikit-surprise->surprise) (1.1
```

Importing the necessary libraries and overview of the dataset

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

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

# Basic library for data visualization
import matplotlib.pyplot as plt

# Slightly advanced library for data visualization
import seaborn as sns

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

# A performance metrics in sklearn
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**

```
# Importing and reading the dataset and adding column names
rating = pd.read_csv('/content/drive/MyDrive/ratings_Electronics.csv', names=['user_id', 'prod_id', 'rating', 'timestamp'])
```

```
rating.head()
```

	user_id	prod_id	rating	timestamp
0	AKM1MP6P0OYPR	0132793040	5.0	1365811200
1	A2CX7LUOHB2NDG	0321732944	5.0	1341100800
2	A2NWSAGRHCP8N5	0439886341	1.0	1367193600
3	A2WNBOD3WNDNKT	0439886341	3.0	1374451200
4	A1GI0U4ZRJA8WN	0439886341	1.0	1334707200

```
rating = rating.drop('timestamp', axis = 1)
```

```
df = rating.copy()
```

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
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.

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

```
(65290, 3)
```

Observations

- The data now has 65,290 rows and 3 columns, which is considerably smaller and will be easier/faster to use.

Data types

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

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 65290 entries, 1310 to 7824427
Data columns (total 3 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
dtypes: float64(1), object(2)
memory usage: 2.0+ MB
```

Observations

- The user id and product id are both object, whereas the rating is a float.
- The user id contains both alphanumeric characters, whereas the product id contains numbers.

Checking for missing values

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

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

Observations

- There are no missing values present

Summary Statistics

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

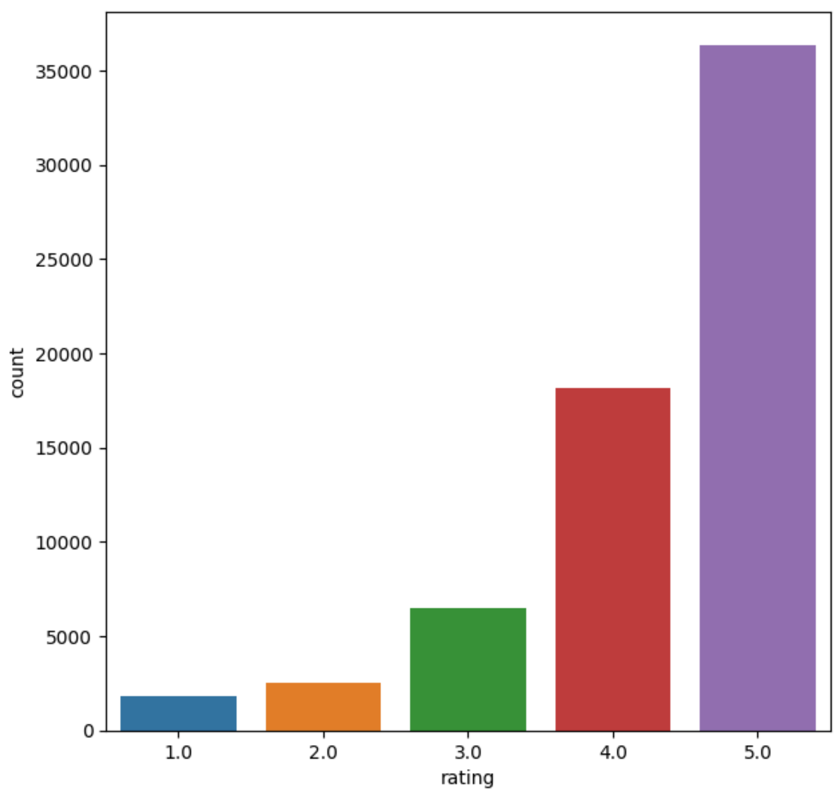
```
count    65290.000000
mean         4.294808
std          0.988915
min          1.000000
25%          4.000000
50%          5.000000
75%          5.000000
max          5.000000
Name: rating, dtype: float64
```

Observations

- The rating has a minimum score of 1 and a maximum score of 5
- A majority of the ratings are 5
- The standard deviation is around 1, meaning most of the scores fall between 3-5, and scores like 1 are outliers.

Checking the rating distribution

```
# Create the bar plot and provide observations
plt.figure(figsize =(7, 7))
sns.countplot(x = 'rating', data = df_final)
plt.show()
```



Observations

- A majority of the ratings were 5 and 4, with anything lower being rarely rated.
- With more than half the ratings being a 5, it seems that people will rarely rate anything less than 4 unless they really didn't like it.

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
nrows, ncols = df_final.shape
print("There are", nrows, "total rows in the data.")
```

```
There are 65290 total rows in the data.
```

```
# Number of unique user ids
df_final['user_id'].nunique()
```

```
1540
```

```
# Number of unique product ids
df_final['prod_id'].nunique()
```

```
5689
```

Observations

- There were a total of 1540 users in this set, meaning some rated different products.
- There were a total of 5689 products that were rated.

Users with the most number of ratings

```
# Top 10 users based on the number of ratings
n = 10
t10 = df_final['user_id'].value_counts().head(10).to_frame('num_ratings')

t10.reset_index(inplace = True)
t10.index = t10.index + 1
t10.head(10)
```

	index	num_ratings
1	ADLVFFE4VBT8	295
2	A3OXHLG6DIBRW8	230
3	A1ODOGXEYECQQ8	217
4	A36K2N527TXXJN	212
5	A25C2M3QF9G7OQ	203
6	A680RUE1FDO8B	196
7	A1UQBFCERIP7VJ	193
8	A22CW0ZHY3NJH8	193
9	AWPODHOB4GFWL	184
10	AGVWTYW0ULXHT	179

```
b10 = df_final['user_id'].value_counts().tail(10).to_frame('num_ratings')
b10.reset_index(inplace = True)
b10.index = b10.index + 1
b10.head(10)
```

	index	num_ratings
1	A2U3OEIK1CUPIK	4
2	A18NDN2CIG2TKR	4
3	A1RCGK44YXNBBB	3
4	A2CL818RN52NWN	3
5	A2J4XMWKR8PPD0	3
6	A16CVJUQOB6GIB	2
7	A2BGZ52M908MJY	2
8	A3DL29NLZ7SXXG	1
9	AP2NZAALUQKF5	1
10	A3MV1KKHX51FYT	1

Observations

- The user ADLVVFFE4VBT8 had the most number of ratings at 295.
- The next highest user had 230.
- Additionally, only 3 users in this dataset rated a product once, meaning the vast majority rated multiple products.

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

Model 1: Rank Based Recommendation System

```
# 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
calc_avg_count = pd.DataFrame({'avg_rating':average_rating, 'rating_count':count_rating})
# Sort the dataframe by average of ratings in the descending order
final_rating = calc_avg_count.sort_values(by = ['avg_rating'], ascending = False)
# See the first five records of the "final_rating" dataset
final_rating.head()
```

	avg_rating	rating_count
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_prod(data, n, min_interaction):
# Finding products with minimum number of interactions
    top_n = data[data['rating_count'] >= min_interaction]
# Sorting values with respect to average rating
    top_n = top_n.sort_values(by='avg_rating', ascending= False)
    return top_n.index[:n]
```

Recommending top 5 products with 50 minimum interactions based on popularity

```
top5_50 = top_n_prod(calc_avg_count, 5, 50)
top5_50
```

```
Index(['B001TH7GUU', 'B003ES5ZUU', 'B0019EHU8G', 'B006W8U2MU', 'B000QUUFRW'], dtype='object', name='prod_id')
```

Recommending top 5 products with 100 minimum interactions based on popularity

```
top5_100 = top_n_prod(calc_avg_count, 5, 100)
top5_100
```

```
Index(['B003ES5ZUU', '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

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`.**

```
# Instantiating Reader scale with expected rating scale
reader = Reader(rating_scale=(0,5))
# Loading the rating dataset
data = Dataset.load_from_df(df[['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

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

# Initialize the KNNBasic model using sim_options declared, Verbose = False, and setting random_state = 1
algo_knn_user = KNNBasic(sim_options = sim_options, verbose = False, random_state = 1)

# 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 function defined above
precision_recall_at_k(algo_knn_user)
```

```
RMSE: 1.1067
Precision:  0.847
Recall:  0.718
F_1 score:  0.777
```

Observations

- The precision is .847, meaning we may be losing some ratings that are relevant.
- The recall score is at .718 which is decent, but could be improved
- The F_1 score is at .777, meaning there is a good balance between the precision and recall

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.

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

```
user: A3LDPF5FMB782Z item: 1400501466 r_ui = 5.00   est = 3.33   {'actual_k': 6, 'was_impossible': False}
Prediction(uid='A3LDPF5FMB782Z', iid='1400501466', r_ui=5, est=3.3333333333333335, details={'actual_k': 6,
'was_impossible': False})
```

Observation

- The model predicted that the user would rate the item as a 3.33, while the actual value is 5.

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

```
# Find unique user_id where prod_id is not equal to "1400501466"
user_not_seen = df_final[df_final.prod_id != '1400501466']
user_not_seen
```

	user_id	prod_id	rating
2082	A2ZR3YTMEEIIZ4	1400532655	5.0
2150	A3CLWR1UUZT6TG	1400532655	5.0
2162	A5JLAU2ARJ0BO	1400532655	1.0
2228	A1P4XD7IORSEFN	1400532655	4.0
2363	A341HCMGNZCBIT	1400532655	5.0
...	...	...	...
7824423	A34BZM6S9L7QI4	B00LGQ6HL8	5.0
7824424	A1G650TTTHEAL5	B00LGQ6HL8	5.0
7824425	A25C2M3QF9G7OQ	B00LGQ6HL8	5.0
7824426	A1E1LEVQ9VQNK	B00LGQ6HL8	5.0
7824427	A2NYK9KWFMJV4Y	B00LGQ6HL8	5.0

65284 rows × 3 columns

```
'A34BZM6S9L7QI4' in user_not_seen['user_id']
```

```
False
```

- It can be observed from the above list that **user "A34BZM6S9L7QI4" has not seen the product with productId "1400501466"** as this userId is a part of the above list.

Below we are predicting rating for `userId=A34BZM6S9L7QI4` and `prod_id=1400501466`.

```
# Predicting rating for a sample user with a non interacted product
algo_knn_user.predict('A34BZM6S9L7QI4', '1400501466', verbose = True)
```

```
user: A34BZM6S9L7QI4 item: 1400501466 r_ui = None   est = 3.00   {'actual_k': 1, 'was_impossible': False}
Prediction(uid='A34BZM6S9L7QI4', iid='1400501466', r_ui=None, est=3.0, details={'actual_k': 1, 'was_impossible': False})
```

Observations

- We can see that the predicted rating is 3.00 for this product.

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], 'min_k': [3],
              'sim_options': {'name': ['msd', 'cosine', 'pearson'],
                              'user_based': [True]}
              }
# Performing 3-fold cross-validation to tune the hyperparameters
gs = GridSearchCV(KNNBasic, param_grid, measures = ['rmse', 'mae'], 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'])
```

```
1.055376411743726
{'k': 40, '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**.

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': 'msd',
               'userbased': True}
# Creating an instance of KNNBasic with optimal hyperparameter values
similarity_algo_optimized_user = KNNBasic(sim_options = sim_options, k = 40, min_k = 3, verbose = False)
# Training the algorithm on the trainset
similarity_algo_optimized_user.fit(trainset)
# Let us compute precision@k and recall@k also with k =10
precision_recall_at_k(similarity_algo_optimized_user, 10 , 3.5)
```

```
RMSE: 1.0539
Precision:  0.841
Recall:  0.721
F_1 score:  0.776
```

Observations

- The RMSE is now a bit lower than in the previous model.
- The precision has slightly increase, whereas the recall has increased.
- The F1 score has slightly decresed.

Steps:

- **Predict rating for the user with `userId="A3LDPF5FMB782Z"`, and `prod_id= "1400501466"` using the optimized model**
- **Predict rating for `userId="A34BZM6S9L7QI4"` 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
similarity_algo_optimized_user.predict('A3LDPF5FMB782Z', '1400501466', r_ui = 5, verbose = True)
```

```
user: A3LDPF5FMB782Z item: 1400501466 r_ui = 5.00   est = 4.74   {'actual_k': 6, 'was_impossible': False}
Prediction(uid='A3LDPF5FMB782Z', iid='1400501466', r_ui=5, est=4.73758865248227, details={'actual_k': 6, 'was_impossible':
False})
```

```
# Use sim_user_user_optimized model to recommend for userId "A34BZM6S9L7QI4" and productId "1400501466"
similarity_algo_optimized_user.predict('A34BZM6S9L7QI4', '1400501466', verbose = True)
```

```
user: A34BZM6S9L7QI4 item: 1400501466 r_ui = None   est = 4.26   {'was_impossible': True, 'reason': 'Not enough neighbors.'}
Prediction(uid='A34BZM6S9L7QI4', iid='1400501466', r_ui=None, est=4.263416620322555, details={'was_impossible': True,
'reason': 'Not enough neighbors.'})
```

Observations

- The predicted rating is 4.74 and 4.26

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 above user
similarity_algo_optimized_user.get_neighbors(0,5)
```

```
[5, 8, 19, 22, 25]
```

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

```
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()].ir

    # 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
rec = get_recommendations(df_final, 'A3LDPF5FMB782Z', 5, similarity_algo_optimized_user)
```

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

	prod_id	predicted_ratings
0	B000053HC5	5
1	B00005LENO	5
2	B00005N6KG	5
3	B000067RT6	5
4	B00006I53X	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**.

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

# Train the algorithm on the trainset, and predict ratings for the test set
#algo_knn_item.fit(trainset)
# Let us compute precision@k, recall@k, and f_1 score with k = 10
#precision_recall_at_k(algo_knn_item, 10)
```

Observations

- Unable to fit trainset due to exceeding ram usage

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".

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

Observations

- Unable to predict due to missing training set

Below we are **predicting rating for the `userId = A34BZM6S9L7QI4` and `prod_id = 1400501466`**.

```
# Predicting rating for a sample user with a non interacted product
# algo_knn_item.predict('A34BZM6S9L7QI4', '1400501466')
```

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
#three_fold = GridSearchCV(KNNBasic, param_grid, measures=['rmse'], cv = 3, n_jobs = -1)
# Fitting the data
#three_fold.fit(data)
# Find the best RMSE score
#print(three_fold.best_score['rmse'])
# Find the combination of parameters that gave the best RMSE score
#print(three_fold.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.

```
# 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_optimized = KNNBasic(sim_options=sim_options, k = 30, min_k = 3, random_state = 1, verbose = False)
# Training the algorithm on the trainset
# sim_item_optimized.fit(trainset)
# Let us compute precision@k and recall@k, f1_score and RMSE
# precision_recall_at_k(sim_item_optimized)
```

Observations

- Optimized options should increase precision, recall and the F1 score
- Increase in precision and recall would increase F1

Steps:

- **Predict rating for the user with `userId="A3LDPF5FMB782Z"`, and `prod_id= "1400501466"` using the optimized model**
- **Predict rating for `userId="A34BZM6S9L7QI4"` 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_optimized.predict('A3LDPF5FMB782Z', '1400501466', r_ui = 5, verbose = True)
```

```
# Use sim_item_item_optimized model to recommend for userId "A34BZM6S9L7QI4" and productId "1400501466"
#sim_item_optimized.predict('A34BZM6S9L7QI4', '1400501466', verbose = True)
```

Observations

- Unable to observe due to inability to fit trainset

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_optimized.get_neighbors(0, 5)
```

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

Hint: Use the get_recommendations() function.

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

```
# Building the dataframe for above recommendations with columns "prod_id" and "predicted_ratings"
#pd.DataFrame(rec, 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**.

```
# 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, and RMSE
precision_recall_at_k(svd)
```

```
RMSE: 0.9867
Precision:  0.842
Recall:  0.704
F_1 score:  0.767
```

Observations

- The Precision and Recall are relatively high
- The F1 score is a bit higher than the other model
- The RMSE is slightly lowwer than the other models

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 = 4.11   {'was_impossible': False}
Prediction(uid='A3LDPF5FMB782Z', iid='1400501466', r_ui=5, est=4.113771760275886, details={'was_impossible': False})
```

Observations

- The model predicts the rating to be 4.11

Below we are predicting rating for the `userId = "A34BZM6S9L7QI4"` and `productId = "1400501466"`.

```
# Making prediction
svd.predict('A34BZM6S9L7QI4', '1400501466', r_ui = 5, verbose = True)
```

```
user: A34BZM6S9L7QI4 item: 1400501466 r_ui = 5.00   est = 4.27   {'was_impossible': False}
Prediction(uid='A34BZM6S9L7QI4', iid='1400501466', r_ui=5, est=4.265322068241108, details={'was_impossible': False})
```

Observations

- The model predicts the rating to be 4.27

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', 'mae'], cv=3, n_jobs=-1)
```

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

```
# Best RMSE score
```

```
# Build the optimized SVD model using optimal hyperparameter search. Use random_state=1
svd_algo_optimized = SVD(n_epochs=30, lr_all=0.005, reg_all = 0.2)
# Train the algorithm on the trainset
svd_algo_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_algo_optimized)
```

```
RMSE: 0.9817
Precision:  0.845
Recall:  0.714
F_1 score:  0.774
```

Observations

- The RMSE is now a bit lower.
- The precision is slightly lower while the recall is a bit higher
- The F1 Score has increased, meaning the model is more optimized.

Steps: