Recommendation_Systems_Learner_Notebook_Full_Code (1)

December 10, 2024

1 Project: Amazon Product Recommendation System

2 Marks: 40

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.

[ ]:
```
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=True).
```

[ ]:
```
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) (1.1.4)
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) (1.26.4)
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.10/dist-
packages (from scikit-surprise->surprise) (1.13.1)
```

2.1 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
```

2.1.1 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**

[ ]:
```
# Import the dataset
df = pd.read_csv(r"/content/drive/MyDrive/MIT IDSS/Data Science and Machine␣
 ↪Learning: Making Data-Driven Decisions/ratings_Electronics.csv", header =␣
 ↪None) # There are no headers in the data file
# Adding appropriate names to the columns
df.columns = ['user_id', 'prod_id', 'rating', 'timestamp']
# Dropping the timestamp column since its irrelevant
df = df.drop('timestamp', axis = 1)
# Saving a copy of the DataFrame that will remain unmodified
df_original = df.copy(deep = True)
df.head(10)
```

[ ]:

	user_id	prod_id	rating
0	AKM1MP6P0OYPR	0132793040	5.0
1	A2CX7LUOHB2NDG	0321732944	5.0
2	A2NWSAGRHCP8N5	0439886341	1.0
3	A2WNBOD3WNDNKT	0439886341	3.0
4	A1GI0U4ZRJA8WN	0439886341	1.0
5	A1QGNMC6O1VW39	0511189877	5.0
6	A3J3BRHTDRFJ2G	0511189877	2.0
7	A2TY0BTJOTENPG	0511189877	5.0
8	A34ATBPOK6HCHY	0511189877	5.0
9	A89DO69P0XZ27	0511189877	5.0

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 a list of all the users
users = df.user_id
# Initializing a dictionary to store the amount of ratings per user
ratings_count = dict()

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

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

[ ]:
```
# We want our users to have at least 50 ratings to be considered
RATINGS_CUTOFF = 50
# Empty list of users that will be removed
remove_users = []

# Using .items() each key-value pair of the dictionary is explored.
# In this case, the key is collected as user and the value as num_ratings
for user, num_ratings in ratings_count.items():
    if num_ratings < RATINGS_CUTOFF:
        remove_users.append(user)

# Keep only the users not in the DataFrame remove_users
df = df.loc[ ~ df.user_id.isin(remove_users)]
```

[ ]:
```
# Get the column containing the products
prods = df.prod_id
# Initializing a dictionary to store the amount of ratings per product
ratings_count = dict()

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

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

[ ]:
```
# We want our item to have at least 5 ratings to be considered
RATINGS_CUTOFF = 5
# Empty list of products that will be removed
remove_products = []

# Using .items() each key-value pair of the dictionary is explored.
# In this case, the key is collected as product and the value as num_ratings
```

```
for product, num_ratings in ratings_count.items():
    if num_ratings < RATINGS_CUTOFF:
        remove_products.append(product)

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

[ ]:
```
# Print a few rows of the imported dataset
df_final.head(10)
```

[ ]:

	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
1485	A19N3S7CBSU6O7	1400501466	5.0
2082	A2ZR3YTMEEIIZ4	1400532655	5.0
2150	A3CLWR1UUZT6TG	1400532655	5.0
2162	A5JLAU2ARJ0BO	1400532655	1.0
2228	A1P4XD7IORSEFN	1400532655	4.0

2.2 Exploratory Data Analysis

2.2.1 Shape of the data

2.2.2 Check the number of rows and columns and provide observations.

[ ]:
```
# Check the number of rows and columns in the DataFrame
rows, columns = df_final.shape
print(f"There are {rows} rows and {columns} columns.")
```

```
There are 65290 rows and 3 columns.
```

2.2.3 Data types

[ ]:
```
# Check the datatpyes
df_final.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 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 and the product id are both type object. - The rating is considered a float. It could probably be turned into an integer.

2.2.4 Checking for missing values

[ ]:
```
# Check for missing values present
df_final.isnull().sum()
```

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

There are no missing values.

2.2.5 Summary Statistics

[ ]:
```
print("Values of rating:", sorted(df_final["rating"].unique()))
```

```
Values of rating: [1.0, 2.0, 3.0, 4.0, 5.0]
```

[ ]:
```
# Summary statistics of 'rating' variable
df_final['rating'] = df_final['rating'].astype(int)
df_final.describe().T
```

[ ]:

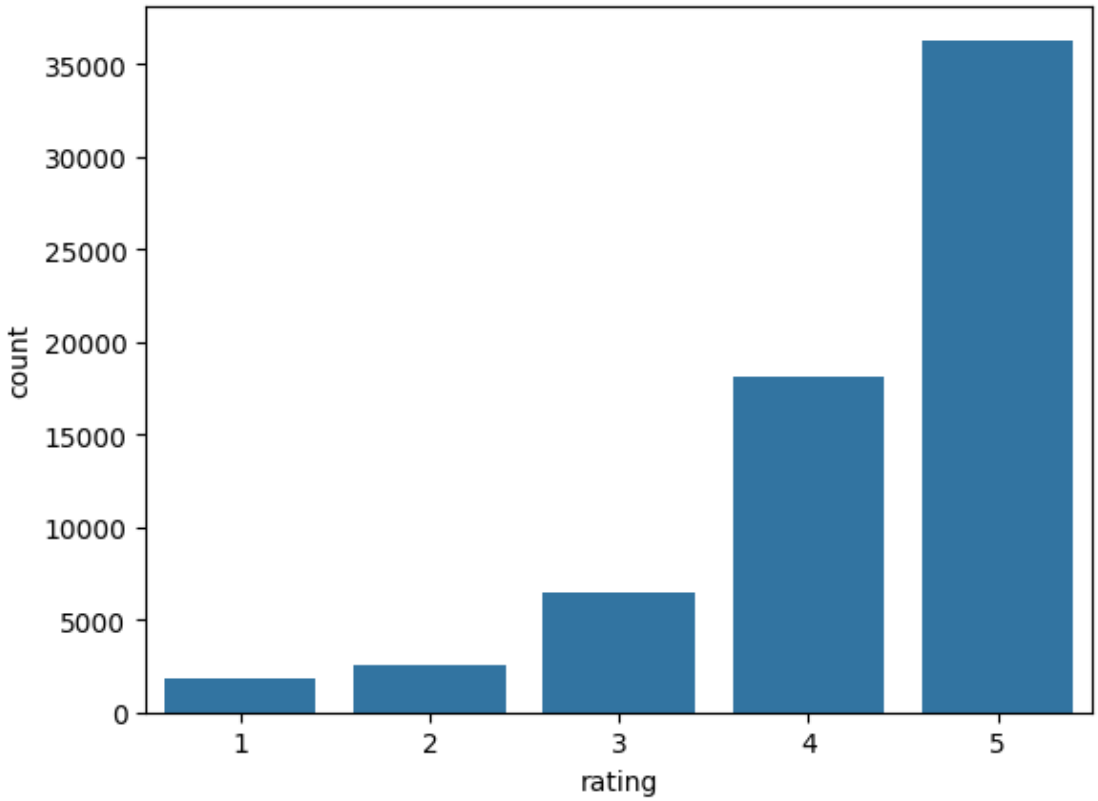
	count	mean	std	min	25%	50%	75%	max
rating	65290.0	4.294808	0.988915	1.0	4.0	5.0	5.0	5.0

More than 75% of the ratings are between 4 and 5. Very few users have given low ratings.

2.2.6 Checking the rating distribution

[ ]:
```
# Create the bar plot and provide observations
print(df_final.rating.value_counts(normalize=True))
sns.countplot(data=df_final, x="rating")
plt.show()
```

```
rating
5    0.556211
4    0.277638
3    0.099265
2    0.038520
1    0.028366
Name: proportion, dtype: float64
```



More than 35000 of the ratings are a 5, which represent 55.6% of all the ratings - Only 15% of the ratings are a 3 or less

2.2.7 Checking the number of unique users and products in the dataset

[ ]:
```
# Check the number of unique rows and columns in the DataFrame
print(f"There are {df_final.user_id.nunique()} unique users and {df_final.
  ↪prod_id.nunique()} unique products.")
```

```
There are 1540 unique users and 5689 unique products.
```

2.2.8 Users with the most number of ratings

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

[ ]:
```
user_id
ADLVFFE4VBT8      295
A3OXHLG6DIBRW8    230
A1ODOGXEYECQQ8    217
A36K2N527TXXJN    212
```

```
A25C2M3QF9G7OQ    203
A680RUE1FDO8B     196
A22CW0ZHY3NJH8    193
A1UQBFCERIP7VJ    193
AWPODHOB4GFWL     184
A3LGT6UZL99IW1    179
dtype: int64
```

Observations:

- The maximum number of reviews from a user is 295. This means that there is definiftely some products that have not been rated by all users.
- Thrrefore, it is pertinent to use Reinforcemnt Learning to achieve matrix completion through different methods.

2.3 Recommendation systems

2.4 Model 1: Rank Based Recommendation System

[ ]:
```
df_final.head()
```

[ ]:

	user_id	prod_id	rating
1310	A3LDPF5FMB782Z	1400501466	5
1322	A1A5KUIIIHFF4U	1400501466	1
1335	A2XIOXRRYX0KZY	1400501466	3
1451	AW3LX47IHPFRL	1400501466	5
1456	A1E3OB6QMBKRYZ	1400501466	1

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

[ ]:

prod_id	avg_rating	rating_count
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['rating_count'] >␣
      ↪min_interaction]

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

         return recommendations.index[:n]
```

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

```
[ ]: top_n_products(final_rating, 5, 50)
```

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

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

```
[ ]: top_n_products(final_rating, 5, 100)
```

```
[ ]: Index(['B003ES5ZUU', 'B000N99BBC', 'B007WTAJTO', 'B002V88HFE', 'B004CLYEDC'],
     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.**

2.5 Model 2: Collaborative Filtering Recommendation System

2.5.1 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.

2.5.2 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.

2.5.3 Some useful functions

- The function below 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_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 = 1)
```

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

2.5.4 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
sim_user_user = KNNBasic(sim_options=sim_options, verbose=False, random_state =␣
 ↪1)

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

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

```
RMSE: 1.0260
Precision:  0.844
Recall:  0.862
F_1 score:  0.853
```

Observations: - The Root Mean Squared Error is 1.026 which is a measure of how distant the predictions are from the actual value. - Precision of 84.4%, which means out of all the recommended products, 84.4% are relevant, which is decent for a rudimentary model. - Recall of 86.2%, which means out of all the relevant products, 86.2% are recommended, which is also decent for a rudimentary model. - F1 score is 85.3%. It indicates that most recommended products were relevant and the books that are relevant were recommended decently correctly. - Tuning the hyperparameters through GridSearchCV can improve the results.

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
sim_user_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})
```

Observations: - The actual rating was 5.00, and the predicted rating was 3.33 which is very inaccurate. The model has to be tuned.

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"
not_seen_this_product = df_final[df_final.prod_id != "1400501466"]["user_id"].
 ↪unique()
print("Is product A34BZM6S9L7QI4 in this list?", "A34BZM6S9L7QI4" in␣
 ↪not_seen_this_product)
not_seen_this_product
```

```
Is product A34BZM6S9L7QI4 in this list? True
```

[ ]:
```
array(['A2ZR3YTMEEIIZ4', 'A3CLWR1UUZT6TG', 'A5JLAU2ARJ0BO', …,
       'A215WH6RUDUCMP', 'A38C12950IM24P', 'A2J4XMWKR8PPD0'], dtype=object)
```

- It can be observed from the above list that **user "A34BZM6S9L7QI4" 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=A34BZM6S9L7QI4` and `prod_id=1400501466`.

[ ]:
```
# Predicting rating for a sample user with a non interacted product
sim_user_user.predict("A34BZM6S9L7QI4", "1400501466", verbose = True)
```

```
user: A34BZM6S9L7QI4 item: 1400501466 r_ui = None   est = 1.99   {'actual_k': 2,
'was_impossible': False}
```

[ ]:
```
Prediction(uid='A34BZM6S9L7QI4', iid='1400501466', r_ui=None,
est=1.991150442477876, details={'actual_k': 2, 'was_impossible': False})
```

Observations: - Based on this user-user similarity-based model, the estimated rating from this user to this product would be 1.99. - Let's see if we get a similar preicted rating if we tune the hyperparameters through GridSearchCV

2.5.5 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': [10, 20, 30], 'min_k': [3, 6, 9],
              'sim_options': {'name': ["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'])
```

```
0.9795880993941418
{'k': 30, 'min_k': 3, 'sim_options': {'name': 'cosine', 'user_based': True,
'min_support': 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**.

[ ]:
```
# 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,␣
 ↪verbose=False, random_state=1)

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

# Let us compute precision@k and recall@k with k=10.
```

```
precision_recall_at_k(sim_user_user_optimized)
```

```
RMSE: 0.9846
Precision:  0.836
Recall:  0.895
F_1 score:  0.864
```

Observations: - After tuning the hyperparameters, the Root Mean Squared Error improved from 1.0260 to 0.9846. - Both the precision and the recall have slightly worsened. - The F1 score has improved from 0.853 to 0.864. - Let's observe if the model predicts better after the tuning of the hyperparameters.

2.5.6 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
sim_user_user_optimized.predict("A3LDPF5FMB782Z", "1400501466", r_ui = 5,␣
↪verbose = True)
```

```
user: A3LDPF5FMB782Z item: 1400501466 r_ui = 5.00   est = 4.30
{'was_impossible': True, 'reason': 'Not enough neighbors.'}
```

[ ]:
```
Prediction(uid='A3LDPF5FMB782Z', iid='1400501466', r_ui=5,
est=4.296427477408486, details={'was_impossible': True, 'reason': 'Not enough
neighbors.'})
```

[ ]:
```
# Use sim_user_user_optimized model to recommend for userId "A34BZM6S9L7QI4"␣
↪and productId "1400501466"
sim_user_user_optimized.predict("A34BZM6S9L7QI4", "1400501466", verbose = True)
```

```
user: A34BZM6S9L7QI4 item: 1400501466 r_ui = None   est = 4.30
{'was_impossible': True, 'reason': 'Not enough neighbors.'}
```

[ ]:
```
Prediction(uid='A34BZM6S9L7QI4', iid='1400501466', r_ui=None,
est=4.296427477408486, details={'was_impossible': True, 'reason': 'Not enough
neighbors.'})
```

Observations: - The model does not find enough neighbors for the predictions for both cases (previously seen and not). It does give a prediction of 4.30 anyways but it likely comes from a rudimentary built-in model as a backup for the sake of proposing a value.

2.5.7 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
sim_user_user_optimized.get_neighbors(0, k=5)
```

[ ]:
```
[17, 38, 80, 99, 154]
```

2.5.8 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()].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	B001ENW61I	5
1	B002WE4HE2	5
2	B002WE6D44	5
3	B003D5MY5I	5
4	B0052SCU8U	5

Observations: - The same rating of approximately 5 was suggested for the user with user_id of A3LDPF5FMB782Z for 5 different products.

2.5.9 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: 1.0147
Precision:  0.826
Recall:  0.853
F_1 score:  0.839
```

Observations: - The Root Mean Squared Error is 1.015 which is a measure of how distant the predictions are from the actual value. - Precision of 82.6%, which means out of all the recommended products, 82.6% are relevant, which is decent for a rudimentary model. - Recall of 85.3%, which means out of all the relevant products, 85.3% are recommended, which is also decent for a rudimentary model. - F1 score is 83.9%. It indicates that most recommended products were relevant and the books that are relevant were recommended decently correctly.

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
sim_item_item.predict("A3LDPF5FMB782Z", "1400501466", r_ui = 5, verbose = True)
```

```
user: A3LDPF5FMB782Z item: 1400501466 r_ui = 5.00   est = 4.30   {'actual_k':
20, 'was_impossible': False}
```

[ ]:
```
Prediction(uid='A3LDPF5FMB782Z', iid='1400501466', r_ui=5, est=4.3,
details={'actual_k': 20, 'was_impossible': False})
```

Observations: - The estimated value 4.30 differs from the actual rating of 5. However it clearly shows an improvement from the user-user untuned model which predicted 3.3 for the same user and product combination.

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

[ ]:
```
# Predicting rating for a sample user with a non interacted product
sim_item_item.predict("A34BZM6S9L7QI4", "1400501466", verbose = True)
```

```
user: A34BZM6S9L7QI4 item: 1400501466 r_ui = None   est = 4.00   {'actual_k': 4,
'was_impossible': False}
```

[ ]:
```
Prediction(uid='A34BZM6S9L7QI4', iid='1400501466', r_ui=None, est=4.0,
details={'actual_k': 4, 'was_impossible': False})
```

Observations: - The estimated value is 4.00 for an item that has never been bought by the user. This prediction seems more reasonable than the 1.99 suggested by the user-user untuned model based on the barplot of the given ratings.

2.5.10 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
grid_obj = GridSearchCV(KNNBasic, param_grid, measures=['rmse'], cv=3)

# Fitting the data
grid_obj.fit(data)

# Best RMSE score
print(grid_obj.best_score['rmse'])

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

```
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
```

```
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
```

```
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the msd similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
Computing the cosine similarity matrix…
Done computing similarity matrix.
0.9747414118847249
{'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.

2.5.11 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_item_optimized = KNNBasic(sim_options=sim_options , k=30 , min_k=9 ,␣
 ↪random_state = 1, verbose = False)

# 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)
```

```
RMSE: 0.9766
Precision:  0.828
Recall:  0.896
F_1 score:  0.861
```

Observations: - After tuning the hyperparameters, the Root Mean Squared Error improved from 1.0147 to 0.9766. - The precision has slightly increased from 82.6% to 82.8%. - The recall has significantly improved from 85.3% to 89.6%. - The F1 score has also improved from 83.9% to 86.1%. - The tuned model clearly does a better job than the rudimentary model.

2.5.12 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_item_optimized.predict("A3LDPF5FMB782Z", "1400501466", r_ui = 5,␣
 ↪verbose = True)
```

```
user: A3LDPF5FMB782Z item: 1400501466 r_ui = 5.00   est = 4.62   {'actual_k':
20, 'was_impossible': False}
```

[ ]:
```
Prediction(uid='A3LDPF5FMB782Z', iid='1400501466', r_ui=5,
est=4.617647058823531, details={'actual_k': 20, 'was_impossible': False})
```

Observations: - The model predicted decently correctly the rating for an item previously bought

by this user (estimated: 4.62, actual: 5.00). This is the ebst prediction yet, beating the 3.3 from the user-user untuned model and the 4.3 from the item-item untuned model.

[ ]:
```
# Use sim_item_item_optimized model to recommend for userId "A34BZM6S9L7QI4"
 ↪and productId "1400501466"
sim_item_item_optimized.predict("A34BZM6S9L7QI4", "1400501466", verbose = True)
```

```
user: A34BZM6S9L7QI4 item: 1400501466 r_ui = None   est = 4.30
{'was_impossible': True, 'reason': 'Not enough neighbors.'}
```

[ ]:
```
Prediction(uid='A34BZM6S9L7QI4', iid='1400501466', r_ui=None,
est=4.296427477408486, details={'was_impossible': True, 'reason': 'Not enough
neighbors.'})
```

Observations: - For a new item, the model could not find enough neightbors to appropriately make a prediction. Nevertheless, a predicition of 4.30 is still shown. This prediction is made with a built-in rudimentary model for the sake of proposing something, but it does not represent a prediction from this sophisticated model.

2.5.13 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, k=5)
```

[ ]:
```
[2, 4, 9, 12, 13]
```

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.
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	4.296427
1	1400599997	4.296427
2	9983891212	4.296427
3	B00000DM9W	4.296427
4	B00000J1V5	4.296427

Observations: - The same rating of approximately 4.3 was suggested for the user with user_id of A1A5KUIIIHFF4U for 5 different products.

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

2.5.14 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.

2.5.15 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: 0.9104
Precision:  0.837
Recall:  0.88
F_1 score:  0.858
```

Observations: - The Root Mean Squared Error is 0.9104 which is a measure of how distant the predictions are from the actual value. - Precision of 83.7%, which means out of all the recommended products, 83.7% are relevant, which is decent for a rudimentary model. - Recall of 88.0%, which means out of all the relevant products, 88.0% are recommended, which is also decent for a rudimentary model. - F1 score is 85.8%. It indicates that most recommended products were relevant and the books that are relevant were recommended decently correctly.

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 = 4.09
{'was_impossible': False}
```

[ ]:
```
Prediction(uid='A3LDPF5FMB782Z', iid='1400501466', r_ui=5,
est=4.094312160755627, details={'was_impossible': False})
```

Observations: - The SVD model predicted 4.09 for an item that has been previously rated with a 5.0 by this user. This untuned model is doing worse than other models previously introduced.

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

[ ]:
```
# Making prediction
svd.predict("A34BZM6S9L7QI4", "1400501466", verbose = True)
```

```
user: A34BZM6S9L7QI4 item: 1400501466 r_ui = None   est = 4.05
{'was_impossible': False}
```

[ ]:
```
Prediction(uid='A34BZM6S9L7QI4', iid='1400501466', r_ui=None,
est=4.051141373805704, details={'was_impossible': False})
```

Observations: - The SVD model predicted 4.05 for an item that has neve been bought by this user. This untuned SVD model has given the highest rating yet for this user-product combination with a rating of 4.05.

2.5.16 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.

[ ]:
```
# 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'])
```

```
0.900292798410359
{'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 trainset
svd_optimized=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: 0.9014
Precision:  0.841
Recall:  0.88
F_1 score:  0.86
```

Observations: - The Root Mean Squared Error is 0.9014 which is a measure of how distant the predictions are from the actual value. - Precision of 84.1%, which means out of all the recommended products, 84.1% are relevant, which is decent for a rudimentary model. - Recall of 88.0%, which means out of all the relevant products, 88.0% are recommended, which is also decent for a rudimentary model. - F1 score is 86.0%. It indicates that most recommended products were relevant and the books that are relevant were recommended decently correctly.

2.5.17 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 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 = 4.09
{'was_impossible': False}
```

[ ]:
```
Prediction(uid='A3LDPF5FMB782Z', iid='1400501466', r_ui=5,
est=4.090441179731308, details={'was_impossible': False})
```

Observations: - The predicted rating of 4.09 is far from the actual one of 5.0. Some other models have done a better job at this.

[ ]:
```
# Use svd_algo_optimized model to recommend for userId "A34BZM6S9L7QI4" and␣
 ↪productId "1400501466"
svd_optimized.predict("A34BZM6S9L7QI4", "1400501466", verbose = True)
```

```
user: A34BZM6S9L7QI4 item: 1400501466 r_ui = None   est = 4.06
{'was_impossible': False}
```

```
[ ]: Prediction(uid='A34BZM6S9L7QI4', iid='1400501466', r_ui=None,
     est=4.06446285064312, details={'was_impossible': False})
```

Observations: - The predicted rating of 4.06 has been the highest prediction in all models all-around for this combination of user-product.

```
[ ]: # Making top 5 recommendations for user_id A1A5KUIIIHFF4U with similarity-based␣
      ↪recommendation engine.
     recommendations = get_recommendations(df_final, "A1A5KUIIIHFF4U", 5,␣
      ↪svd_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  B00007E89K           4.315606
     1  B003S5SOLG           4.263806
     2  B0009YDP7W           4.263234
     3  B000MP831G           4.261569
     4  B002M3SOCE           4.257270
```

2.5.18 Conclusion and Recommendations

- First and foremost, a model will never predict accurately the human behavior. Indeed, even a model predicts upon predictions (consumers will act similarly, historiy can predict the future).That being said, a sophisticated model can absolutely be a helpful tool for resources allocation.
- The first model *Rank Based Recommendation System* could be helpful because peer pressure plays a factor. E.g. "Netflix top 10 trending movies". No one wants to miss what everyone is watching. Namely, this type of model could be useful for frequent and/or compulsive clients that navigate to Amazon's webpage out of boredom.
- The model that performed the best in terms of Root Mean Squared Error is the Tuned Singular Value Decomposition model with a RMSE of 0.9014.
- The model that performed the best in terms of Precision is the Untuned User-User similarity-based model with a precision of 84.4%.
- The model that performed the best in terms of Recall is the Tuned Item-Item similarity-based model with a recall of 89.6%.
- The model that performed the best in terms of F1 Score is the Tuned User-User similarity-based model with a F1 score of 86.4%.
- Evidently, it is not straightforward to determine which is the best model since it depends on the application. The model that best predicted the rating for a product that had been previously rated by a given user is the Tuned Item-Item similarity-based model, predicting 4.62 for a product that was rated with a 5.0.
- The model that best predicted the rating for a product that had never been rated by a given user is the Tuned Singular Value Decomposition model, predicting 4.09. Other models could not find enough neighbors or gave an extremly low value.

- Overall, I consider that the best model is the Tuned Singular Value Decomposition model because it has the lowest RMSE, while keeping a high precision, recall and F1 score. Moreover, the 5 recommendations suggested by this model show very close yet different ratings. This is realistic and accurate.