## Recommender System

import pandas as pd  
import numpy as np  
import scipy  
import scipy.stats  
from sklearn.metrics import mean\_squared\_error

#### Reading the dataset and preprocessing

rating\_data = pd.read\_csv('ml-1m/ratings.dat', header=None, sep='::')  
rating\_data.columns = ['UserID', 'ItemID', 'Rating', 'Timestamp']  
rating\_data.drop(columns=['Timestamp'], axis=1, inplace=True)

rating\_data

UserID ItemID Rating  
0 1 1193 5  
1 1 661 3  
2 1 914 3  
3 1 3408 4  
4 1 2355 5  
... ... ... ...  
1000204 6040 1091 1  
1000205 6040 1094 5  
1000206 6040 562 5  
1000207 6040 1096 4  
1000208 6040 1097 4  
  
[1000209 rows x 3 columns]

item\_cnt = max(rating\_data['ItemID']) + 1  
users\_cnt = max(rating\_data['UserID']) + 1  
item\_cnt, users\_cnt

(3953, 6041)

rating\_data.shape

(1000209, 3)

## Training testing split

rating\_train = rating\_data.sample(frac=0.8, random\_state=200)  
rating\_test = rating\_data.drop(rating\_train.index)

matrix = np.zeros(shape=(users\_cnt, item\_cnt))  
matrix.shape

(6041, 3953)

for row in rating\_train.itertuples():  
 matrix[row.UserID][row.ItemID] = row.Rating  
matrix

array([[0., 0., 0., ..., 0., 0., 0.],  
 [0., 0., 0., ..., 0., 0., 0.],  
 [0., 0., 0., ..., 0., 0., 0.],  
 ...,  
 [0., 0., 0., ..., 0., 0., 0.],  
 [0., 0., 0., ..., 0., 0., 0.],  
 [0., 3., 0., ..., 0., 0., 0.]])

## Using Item-item based

we have adopted item-item based collaborative filtering for this assignment. In order to avoid repeatitive computation of item-item similarity, we calculate similarity between each item and store it in a matrix which is pickled and later can be loaded into RAM any time needed. For similarity we are using pearson correlation which handles dissimilarity and simmilarity between users in a better way as pearson correlation is between [-1,-1] where -1 indicates severely dissimilar and 1 indicates highly similar. Pearson correlation is defined as

def pearson\_coorelation(x, y):  
 return scipy.stats.pearsonr(x, y)

item\_similarity = np.zeros(shape=(item\_cnt, item\_cnt))  
for i in range(item\_cnt):  
 print(i)  
 for j in range(i + 1, item\_cnt):  
 item1 = np.array(matrix[:, i])  
 item2 = np.array(matrix[:, j])  
 ids\_take = np.intersect1d(np.nonzero(item1), np.nonzero(item2))  
 item1 = item1[ids\_take]  
 item2 = item2[ids\_take]  
 if len(item1) > 1:  
 item\_similarity[i][j] = item\_similarity[j][i] = pearson\_coorelation(item1, item2)[0]

## Collaborative Filtering

import pandas as pd  
import numpy as np  
import scipy  
import math  
import scipy.stats  
from sklearn.metrics import mean\_squared\_error  
import textract  
import os  
import pickle  
from pathlib import Path  
import numpy as np  
import random  
import pandas as pd  
from itertools import combinations  
from datetime import datetime  
root = Path(".")

### Reading the movie-rating dataset and loading the item similarity matrix

my\_path = root / "Pickled\_files" / "item\_similarity\_matrix"  
dbfile = open(my\_path, 'rb')   
item\_sim = pickle.load(dbfile)  
dbfile.close()

rating\_data = pd.read\_csv('ml-1m/ratings.dat', header=None, sep='::')  
rating\_data.columns = ['UserID', 'ItemID', 'Rating', 'Timestamp']  
rating\_data.drop(columns=['Timestamp'], axis=1, inplace=True)

rating\_data

UserID ItemID Rating  
0 1 1193 5  
1 1 661 3  
2 1 914 3  
3 1 3408 4  
4 1 2355 5  
... ... ... ...  
1000204 6040 1091 1  
1000205 6040 1094 5  
1000206 6040 562 5  
1000207 6040 1096 4  
1000208 6040 1097 4  
  
[1000209 rows x 3 columns]

item\_cnt = max(rating\_data['ItemID']) + 1  
users\_cnt = max(rating\_data['UserID']) + 1  
item\_cnt, users\_cnt

(3953, 6041)

rating\_data.shape

(1000209, 3)

### Splitting the dataset into test and train

rating\_train = rating\_data.sample(frac=0.8, random\_state=200)  
rating\_test = rating\_data.drop(rating\_train.index)

### Populating the rating matrix

matrix = np.zeros(shape=(users\_cnt, item\_cnt))  
matrix.shape

(6041, 3953)

for row in rating\_train.itertuples():  
 matrix[row.UserID][row.ItemID] = row.Rating  
matrix

array([[0., 0., 0., ..., 0., 0., 0.],  
 [0., 0., 0., ..., 0., 0., 0.],  
 [0., 0., 0., ..., 0., 0., 0.],  
 ...,  
 [0., 0., 0., ..., 0., 0., 0.],  
 [0., 0., 0., ..., 0., 0., 0.],  
 [0., 3., 0., ..., 0., 0., 0.]])

colab\_baseline = matrix.copy()

item\_sim.shape

(3953, 3953)

### Calculating User, Item and Global Average for baseline prediction

algo\_stats = list()  
actual\_by\_pred = dict()  
global\_avg , cnt = 0 , 0  
item\_avg = [0] \* item\_cnt  
user\_avg = [0] \* users\_cnt  
item\_sz = [0] \* item\_cnt  
user\_siz = [0] \* users\_cnt

for users in range(1 , users\_cnt) :   
 for items in range(1 , item\_cnt) :   
 if matrix[users][items] > 0 :   
 global\_avg += matrix[users][items]  
 cnt += 1  
 item\_avg[items] += matrix[users][items]  
 item\_sz[items] += 1  
 user\_avg[users] += matrix[users][items]  
 user\_siz[users] += 1

global\_avg = global\_avg / cnt

We are using Item-Item based collaborative filtering for our prediction. Here, we explore the relationship between the pair of items (the user who bought Y, also bought Z). We find the missing rating with the help of the ratings given to the other items by the user. We get the top K similar items for every item which will help us in predicting the rating a user might give for any particular Item.

def get\_sim\_items(itemID) :   
 sim\_userID = list()  
 for item in range(1 , item\_sim.shape[0]) :   
 if item == itemID :   
 continue  
 sim\_userID.append((item\_sim[itemID][item] , item))  
 sim\_userID.sort(key=lambda y: -y[0])  
 return sim\_userID

sim\_top\_k = dict()  
for i in range(1 , item\_cnt) :   
 sim\_top\_k[i] = get\_sim\_items(i)

## Prediction using collaborative filtering

we use the items (already rated by the user) that are most similar to the missing item to generate rating. We hence try to generate predictions based on the ratings of similar products. We compute this using a formula which computes rating for a particular item using weighted sum of the ratings of the other similar products.

def predict(user\_id , item\_id , K) :   
 sim\_items = sim\_top\_k[item\_id]  
 pred\_rating , sum\_sim = 0 , 0  
 taken = 0  
 for (item\_s , itemID) in sim\_items :  
 if item\_s < 0 or math.isnan(item\_s):   
 continue  
 if taken == K :  
 break  
 if matrix[user\_id][itemID] > 0 :   
 taken += 1  
 sum\_sim += item\_s   
 pred\_rating += item\_s \* matrix[user\_id][itemID]  
 if sum\_sim == 0 :   
 return sum\_sim  
 a = pred\_rating / sum\_sim  
 return pred\_rating / sum\_sim

#### Combining baseline with collaborative filtering

For baseline approach to handle strict and lineant raters we have prediction :

where = global average rating = user average - global average = item average - global average

In combination with collaborative filtering we have predicted rating as average of colab filtering rating and baseline approach

def predict\_baseline(user\_id , item\_id) :   
 if user\_siz[user\_id] == 0 :  
 user\_bias = -global\_avg  
 else :   
 user\_bias = (user\_avg[user\_id] / user\_siz[user\_id]) - global\_avg  
 if item\_sz[item\_id] == 0 :   
 item\_bias = -global\_avg  
 else :  
 item\_bias = (item\_avg[item\_id] / item\_sz[item\_id]) - global\_avg  
 return global\_avg + user\_bias + item\_bias

start\_prediction\_time = datetime.now()  
for user in range(1 , users\_cnt) :   
 unrated\_items = np.where(matrix[user] == 0)[0]  
 for item in unrated\_items:  
 if item == 0 :   
 continue  
 colab\_rat = predict(user , item , 5)  
 matrix[user][item] = colab\_rat  
 colab\_baseline[user][item] = (colab\_rat + predict\_baseline(user, item)) / 2  
end\_prediction\_time = datetime.now()  
print("Time for prediction" , (end\_prediction\_time - start\_prediction\_time).total\_seconds())

Time for prediction 235.235813

def rmse\_matrix(mat, test\_data):  
 y\_actual = list(test\_data.Rating)  
 y\_pred = list()  
 y\_pred\_baseline = list()  
 y\_pred\_comb = list()  
 for id , row in test\_data.iterrows() :   
 uid , itid = row['UserID'] , row['ItemID']  
 y\_pred.append(mat[uid][itid])  
 baseline = predict\_baseline(uid , itid)  
 y\_pred\_baseline.append(baseline)  
 y\_pred\_comb.append((mat[uid][itid] + baseline) / 2)  
 return (mean\_squared\_error(y\_actual , y\_pred) , mean\_squared\_error(y\_actual , y\_pred\_comb))

rmse\_matrix(matrix , rating\_test)

(1.1557051129397395, 0.9212280220671454)

my\_path = root / "Pickled\_files" / "pred\_matrix\_collab"  
dbfile = open(my\_path, 'wb')   
pickle.dump(matrix,dbfile)  
dbfile.close()

my\_path = root / "Pickled\_files" / "pred\_basline"  
dbfile = open(my\_path, 'wb')   
pickle.dump(colab\_baseline,dbfile)  
dbfile.close()

## Singular Value Decomposition

Singular value decomposition is a very popular linear algebra technique to break down a matrix into the product of a few smaller matrices. we can use SVD to discover relationship between items. A recommender system can be build easily from this.

where U is a m x m matrix, S is a diagonal matrix of m x r and V is a r x r matrix. Singular value decomposition gets its name from the diagonal entries on , which are called the singular values of matrix. Based on the percentage of singular values we need to retain, we can do a dimensionality reduction and take top K diagonal elements of S. This is called the reduced SVD.

In essence, we are removing several rows on U that the corresponding singular values in S are small, before we use it to compute the similarity. This would likely make the prediction more accurate as those less important features of a item are removed from consideration. Once we have the item-item similarity using the reduced SVD, the method to predict a rating is similar to collaborative filtering where we give importance to top K similar items

class SVD\_recommender:  
 def \_\_init\_\_(self, retained\_energy) -> None:  
 self.read\_dataset()  
 self.train\_test\_split()  
 self.generate\_user\_item\_matrix()  
  
 self.U, self.S, self.V\_T = la.svd(self.matrix)  
 self.SVD\_transform(retained\_energy)  
 self.calculate\_item\_similarity()  
   
 self.sim\_top\_k = [None for \_ in range(self.item\_cnt + 1)]  
 for i in range(1, self.item\_cnt):   
 self.sim\_top\_k[i] = self.get\_sim\_items(i)  
  
 def read\_dataset(self):  
 self.rating\_data = pd.read\_csv('ml-1m/ratings.dat', header=None, sep='::')  
 self.rating\_data.columns = ['UserID', 'ItemID', 'Rating', 'Timestamp']  
 self.rating\_data.drop(columns=['Timestamp'], axis=1, inplace=True)  
   
 self.item\_cnt = max(self.rating\_data['ItemID']) + 1  
 self.users\_cnt = max(self.rating\_data['UserID']) + 1  
  
 def train\_test\_split(self):  
 self.rating\_train = self.rating\_data.sample(frac=0.8, random\_state=200)  
 self.rating\_test = self.rating\_data.drop(self.rating\_train.index)  
  
 def generate\_user\_item\_matrix(self):  
 self.matrix = np.zeros(shape=(self.users\_cnt, self.item\_cnt))  
 for row in self.rating\_train.itertuples():  
 self.matrix[row.UserID][row.ItemID] = row.Rating  
  
 def get\_dim(self, S, retained\_energy):   
 sum\_sq = sum(S \*\* 2)  
 cur\_sum, cur\_d = 0, 0  
 for val in S:  
 if cur\_sum >= sum\_sq \* retained\_energy:   
 return cur\_d   
 cur\_sum += val \*\* 2  
 cur\_d += 1  
 return cur\_d  
  
 def similarity(self, A, B):  
 return (1.0 / (1.0 + la.norm(A - B)))  
  
 def SVD\_transform(self, retained\_energy):   
 red\_dim = self.get\_dim(self.S, retained\_energy)  
  
 transformed\_S = np.diag(self.S[:red\_dim])  
 transformed\_U = self.U[:, :red\_dim]  
 transformed\_V\_T = self.V\_T[:red\_dim, :]  
  
 self.SVD\_matrix = np.dot(transformed\_U, transformed\_S)  
 self.SVD\_matrix = np.dot(self.SVD\_matrix, transformed\_V\_T)  
   
 def calculate\_item\_similarity(self):  
 self.item\_similarity = np.zeros(shape=(self.item\_cnt, self.item\_cnt))  
 transformed\_item\_matrix = list()  
 for items in range(1, self.item\_cnt):   
 transformed\_item\_matrix.append(self.SVD\_matrix[items, :].T)   
  
 for item1 in range(1, self.item\_cnt):   
 for item2 in range(item1 + 1, self.item\_cnt):   
 sim = self.similarity(transformed\_item\_matrix[item1 - 1], transformed\_item\_matrix[item2 - 1])  
   
 self.item\_similarity[item1][item2] = self.item\_similarity[item2][item1] = sim  
   
 def get\_sim\_items(self, itemID):   
 sim\_userID = list()  
 for item in range(1, self.item\_similarity.shape[0]):   
 if item == itemID:   
 continue  
 sim\_userID.append((self.item\_similarity[itemID][item], item))  
 sim\_userID.sort(key=lambda y: -y[0])  
 return sim\_userID  
  
 def predict\_SVD(self, userID, itemID, top\_K):  
 similarity\_sum = 0   
 rating\_sum, cnt = 0, 0  
   
 for (item\_s, items) in self.sim\_top\_k[itemID]:  
 if cnt == top\_K:   
 break  
 if self.matrix[userID][items] == 0:   
 continue  
 similarity\_sum += item\_s  
 rating\_sum += item\_s \* self.matrix[userID][items]  
 cnt += 1  
 if similarity\_sum == 0:  
 return 0  
   
 return (rating\_sum / similarity\_sum)  
   
 def get\_top\_k(self, k=3):  
 start\_prediction\_time = datetime.now()  
 for users in range(1, self.users\_cnt):   
 unrated\_items = np.where(self.matrix[users] == 0)[0]  
 for items in unrated\_items:  
 if items == 0:  
 continue  
 assert(self.matrix[users][items] == 0)  
 self.matrix[users][items] = self.predict\_SVD(users, items, k)  
 end\_prediction\_time = datetime.now()  
 print("Time for prediction" , (end\_prediction\_time - start\_prediction\_time).total\_seconds())  
  
  
 def get\_rmse(self):  
 y\_actual = list(self.rating\_test.Rating)  
 y\_pred = list()  
 for id, row in self.rating\_test.iterrows():   
 uid, itid = row['UserID'], row['ItemID']  
 y\_pred.append(self.matrix[uid][itid])  
 return mean\_squared\_error(y\_actual, y\_pred) \*\* .5

s\_recom = SVD\_recommender(0.9)  
s\_recom.get\_top\_k()  
s\_recom.get\_rmse()

Time for prediction 247.880419

1.1767304502722231

my\_path = root / "Pickled\_files" / "SVD90"  
dbfile = open(my\_path, 'wb')   
pickle.dump(s\_recom.matrix,dbfile)  
dbfile.close()

s\_recom = SVD\_recommender(1.0)  
s\_recom.get\_top\_k()  
s\_recom.get\_rmse()

Time for prediction 233.878857

1.188154215448053

my\_path = root / "Pickled\_files" / "SVD"  
dbfile = open(my\_path, 'wb')   
pickle.dump(s\_recom.matrix,dbfile)  
dbfile.close()

## CUR Decomposition

CUR decomposition is similar to SVD in a way that it also decomposes a given matrix into three different matrices C, U and R where R represents the diagonal matrix but it differs from SVD in the method through which C, U and R are formed Formally, a CUR matrix approximation of a matrix A is three matrices C, U, and R such that C is made from columns of A, R is made from rows of A, and that the product CUR closely approximates A. Usually the CUR is selected to be a rank-k approximation, which means that C contains k columns of A, R contains k rows of A, and U is a k-by-k matrix. Each row and column of A has some probability assigned to it which is equal to sum of squares of all entries in that row divided by sum of squares of all elements in the matrix. K rows and and K columns are sampled from the original matrix with respect to the probability that any row/column is picked proportional to sum of squares of entries it has. Once the matrix is decomposed, the method of predicting the ratings is similar to SVD.

class CUR:  
 def \_\_init\_\_(self, retained\_energy) -> None:  
 self.read\_dataset()  
 self.train\_test\_split()  
 self.generate\_user\_item\_matrix()  
 self.U, self.S, self.V\_T = self.CUR\_decomp(500)  
 self.CUR\_transform(retained\_energy)  
 self.calculate\_item\_similarity()  
   
 self.sim\_top\_k = [None for \_ in range(self.item\_cnt + 1)]  
 for i in range(1, self.item\_cnt):   
 self.sim\_top\_k[i] = self.get\_sim\_items(i)  
  
 def CUR\_decomp(self , red\_dim):   
 C = np.zeros((self.matrix.shape[0], red\_dim))  
 R = np.zeros((red\_dim, self.matrix.shape[1]))  
 sum\_of\_squares = np.sum(self.matrix \*\* 2)  
 probability\_colwise = np.sum(self.matrix \*\* 2, axis=0) / sum\_of\_squares  
 col\_ids = np.arange(self.matrix.shape[1])  
  
 taken\_cols = 0  
 taken\_col\_list = list()  
 dup\_col\_list = list()  
   
 while(taken\_cols < red\_dim) :   
 cur\_p = np.random.choice(col\_ids, p = probability\_colwise)  
 if cur\_p not in taken\_col\_list :   
 C[:, taken\_cols] = self.matrix[:, cur\_p] / np.sqrt(probability\_colwise[cur\_p] \* red\_dim)  
 taken\_cols += 1  
 taken\_col\_list.append(cur\_p)  
 dup\_col\_list.append(1)  
 else :   
 get\_id = taken\_col\_list.index(cur\_p)  
 dup\_col\_list[get\_id] += 1  
   
 C = np.multiply(C, np.sqrt(dup\_col\_list))  
  
 sum\_of\_squares\_row = np.sum(self.matrix \*\* 2, axis = 1)  
 probability\_row\_wise = sum\_of\_squares\_row / sum\_of\_squares  
 col\_ids = np.arange(self.matrix.shape[0])  
 taken\_row = 0  
 taken\_row\_list = list()  
 dup\_row\_list = list()  
  
 while(taken\_row < red\_dim) :   
 cur\_p = np.random.choice(col\_ids, p = probability\_row\_wise)  
 if cur\_p not in taken\_row\_list :   
 R[taken\_row, :] = self.matrix[cur\_p, :] / np.sqrt(probability\_row\_wise[cur\_p] \* red\_dim)  
 taken\_row += 1  
 taken\_row\_list.append(cur\_p)  
 dup\_row\_list.append(1)  
 else :   
 get\_id = taken\_row\_list.index(cur\_p)  
 dup\_row\_list[get\_id] += 1  
   
 R = np.multiply(R.T, np.sqrt(dup\_row\_list)).T  
  
 Sigma = np.zeros((red\_dim, red\_dim))  
 for i, I in enumerate(taken\_row\_list):   
 for j, J in enumerate(taken\_col\_list):  
 Sigma[i, j] = self.matrix[I, J]  
   
 cur\_U, cur\_S, cur\_V\_T = la.svd(Sigma, red\_dim)  
 new\_S = np.zeros((red\_dim, red\_dim))  
 for i in range(red\_dim):  
 new\_S[i, i] = cur\_S[i]  
  
 cur\_S = new\_S   
 for cols in range(red\_dim):  
 if cur\_S[cols, cols] >= 1:  
 cur\_S[cols, cols] = 1 / cur\_S[cols, cols]  
 else:   
 cur\_S[cols, cols] = 0  
   
 cur\_S\_sq = np.dot(cur\_S, cur\_S)  
 U = np.dot(cur\_V\_T.T, np.dot(cur\_S\_sq, cur\_U.T))  
 print(U.shape)  
 return C, U, R  
  
 def read\_dataset(self):  
 self.rating\_data = pd.read\_csv('ml-1m/ratings.dat', header=None, sep='::')  
 self.rating\_data.columns = ['UserID', 'ItemID', 'Rating', 'Timestamp']  
 self.rating\_data.drop(columns=['Timestamp'], axis=1, inplace=True)  
   
 self.item\_cnt = max(self.rating\_data['ItemID']) + 1  
 self.users\_cnt = max(self.rating\_data['UserID']) + 1  
  
 def train\_test\_split(self):  
 self.rating\_train = self.rating\_data.sample(frac=0.8, random\_state=200)  
 self.rating\_test = self.rating\_data.drop(self.rating\_train.index)  
  
 def generate\_user\_item\_matrix(self):  
 self.matrix = np.zeros(shape=(self.users\_cnt, self.item\_cnt))  
 for row in self.rating\_train.itertuples():  
 self.matrix[row.UserID][row.ItemID] = row.Rating  
  
 def get\_dim(self, S, retained\_energy):   
 sum\_sq = 0  
 val\_list = list()  
 for i in range(S.shape[0]) :   
 sum\_sq += S[i, i] \*\* 2  
 val\_list.append(S[i, i])  
 cur\_sum, cur\_d = 0, 0  
 for val in val\_list:  
 if cur\_sum >= sum\_sq \* retained\_energy:   
 return cur\_d   
 cur\_sum += val \*\* 2  
 cur\_d += 1  
 return cur\_d  
  
 def similarity(self, A, B):  
 return (1.0 / (1.0 + la.norm(A - B)))  
  
 def CUR\_transform(self, retained\_energy):   
 red\_dim = self.get\_dim(self.S, retained\_energy)  
  
 transformed\_S = np.diag(self.S[:red\_dim])  
 transformed\_U = self.U[:, :red\_dim]  
 transformed\_V\_T = self.V\_T[:red\_dim, :]  
 new\_S = np.zeros((transformed\_S.shape[0] , transformed\_S.shape[0]))  
 print(new\_S.shape)  
 for i in range(transformed\_S.shape[0]) :   
 new\_S[i, i] = transformed\_S[i]  
 transformed\_S = new\_S  
 print(transformed\_S.shape , transformed\_U.shape , transformed\_V\_T.shape)  
 self.SVD\_matrix = np.dot(transformed\_U, transformed\_S)  
 self.SVD\_matrix = np.dot(self.SVD\_matrix, transformed\_V\_T)  
   
 def calculate\_item\_similarity(self):  
 self.item\_similarity = np.zeros(shape=(self.item\_cnt, self.item\_cnt))  
 transformed\_item\_matrix = list()  
 for items in range(1, self.item\_cnt):   
 transformed\_item\_matrix.append(self.SVD\_matrix[items, :].T)   
  
 for item1 in range(1, self.item\_cnt):   
 for item2 in range(item1 + 1, self.item\_cnt):   
 sim = self.similarity(transformed\_item\_matrix[item1 - 1], transformed\_item\_matrix[item2 - 1])  
   
 self.item\_similarity[item1][item2] = self.item\_similarity[item2][item1] = sim  
   
 def get\_sim\_items(self, itemID):   
 sim\_userID = list()  
 for item in range(1, self.item\_similarity.shape[0]):   
 if item == itemID:   
 continue  
 sim\_userID.append((self.item\_similarity[itemID][item], item))  
 sim\_userID.sort(key=lambda y: -y[0])  
 return sim\_userID  
  
 def predict\_SVD(self, userID, itemID, top\_K):  
 similarity\_sum = 0   
 rating\_sum, cnt = 0, 0  
   
 for (item\_s, items) in self.sim\_top\_k[itemID]:  
 if cnt == top\_K:   
 break  
 if self.matrix[userID][items] == 0:   
 continue  
 similarity\_sum += item\_s  
 rating\_sum += item\_s \* self.matrix[userID][items]  
 cnt += 1  
 if similarity\_sum == 0:  
 return 0  
   
 return (rating\_sum / similarity\_sum)  
   
 def get\_top\_k(self, k=3):  
 start\_prediction\_time = datetime.now()  
 for users in range(1, self.users\_cnt):   
 unrated\_items = np.where(self.matrix[users] == 0)[0]  
 for items in unrated\_items:  
 if items == 0:  
 continue  
 assert(self.matrix[users][items] == 0)  
 self.matrix[users][items] = self.predict\_SVD(users, items, k)  
 end\_prediction\_time = datetime.now()  
 print("Time for prediction" , (end\_prediction\_time - start\_prediction\_time).total\_seconds())  
  
 def get\_rmse(self):  
 y\_actual = list(self.rating\_test.Rating)  
 y\_pred = list()  
 for id, row in self.rating\_test.iterrows():   
 uid, itid = row['UserID'], row['ItemID']  
 y\_pred.append(self.matrix[uid][itid])  
 return mean\_squared\_error(y\_actual, y\_pred) \*\* .5

cur\_recom = CUR(0.9)  
cur\_recom.get\_top\_k()  
cur\_recom.get\_rmse()  
  
my\_path = root / "Pickled\_files" / "CUR90"  
dbfile = open(my\_path, 'wb')   
pickle.dump(cur\_recom.matrix,dbfile)  
dbfile.close()

Time for prediction 223.776183

print(cur\_recom.get\_rmse())

1.1851374361914413

cur\_recom = CUR(1.0)  
cur\_recom.get\_top\_k()  
  
  
my\_path = root / "Pickled\_files" / "CUR100"  
dbfile = open(my\_path, 'wb')   
pickle.dump(cur\_recom.matrix,dbfile)  
dbfile.close()

Time for prediction 220.19741

print(cur\_recom.get\_rmse())

1.1781062652249525

### Evaluation Metrics

rating\_data = pd.read\_csv('ml-1m/ratings.dat', header=None, sep='::')  
rating\_data.columns = ['UserID', 'ItemID', 'Rating', 'Timestamp']  
rating\_data.drop(columns=['Timestamp'], axis=1, inplace=True)

rating\_data

UserID ItemID Rating  
0 1 1193 5  
1 1 661 3  
2 1 914 3  
3 1 3408 4  
4 1 2355 5  
... ... ... ...  
1000204 6040 1091 1  
1000205 6040 1094 5  
1000206 6040 562 5  
1000207 6040 1096 4  
1000208 6040 1097 4  
  
[1000209 rows x 3 columns]

item\_cnt = max(rating\_data['ItemID']) + 1  
users\_cnt = max(rating\_data['UserID']) + 1  
item\_cnt, users\_cnt

(3953, 6041)

rating\_train = rating\_data.sample(frac=0.8, random\_state=200)  
rating\_test = rating\_data.drop(rating\_train.index)

rating\_test

UserID ItemID Rating  
0 1 1193 5  
2 1 914 3  
8 1 594 4  
9 1 919 4  
13 1 2918 4  
... ... ... ...  
1000201 6040 1080 4  
1000202 6040 1089 4  
1000204 6040 1091 1  
1000207 6040 1096 4  
1000208 6040 1097 4  
  
[200042 rows x 3 columns]

matrix = np.zeros(shape=(users\_cnt, item\_cnt))  
matrix.shape

(6041, 3953)

for row in rating\_data.itertuples():  
 matrix[row.UserID][row.ItemID] = row.Rating  
actual\_matrix = matrix  
actual\_matrix

array([[0., 0., 0., ..., 0., 0., 0.],  
 [0., 5., 0., ..., 0., 0., 0.],  
 [0., 0., 0., ..., 0., 0., 0.],  
 ...,  
 [0., 0., 0., ..., 0., 0., 0.],  
 [0., 0., 0., ..., 0., 0., 0.],  
 [0., 3., 0., ..., 0., 0., 0.]])

### Precision at top K

Precision at k is the propotion of recommended items in the top k that are relevant

def topKPrecision(actual\_mat,pred\_mat,test\_data,K):  
 pred\_ui\_data = dict()  
 for row in test\_data.itertuples():  
 if pred\_ui\_data.get(row.UserID) == None:  
 pred\_ui\_data[row.UserID] = [(row.ItemID,pred\_mat[row.UserID,row.ItemID])]  
 else:  
 pred\_ui\_data[row.UserID].append((row.ItemID,pred\_mat[row.UserID,row.ItemID]))  
 cnt = 0  
 tot = 0  
 for user in pred\_ui\_data.keys():  
 pred\_ui\_data[user] = sorted(pred\_ui\_data[user],key=lambda x: x[-1],reverse = True)  
 if(len(pred\_ui\_data[user]) >= K):  
 cnt += 1  
 tp = 0  
 for item,pr in pred\_ui\_data[user][:K]:  
 if actual\_mat[user,item] >= 3.5:  
 tp += 1  
 tot += tp/K  
 if cnt > 0:  
 return tot/cnt  
 else:  
 print("Error : very large K")

### Spearman Coefficient

Instead of calculating correlation over the raw item score, we calculate it over the rank of the items ordered in the set. It the correlation of the ranks of items between two different sets.

def spearmanCoeff(actual\_mat,pred\_mat,test\_data):  
 pred\_ui\_data = dict()  
 actual\_ui\_data = dict()  
 for row in test\_data.itertuples():  
 if pred\_ui\_data.get(row.UserID) == None:  
 pred\_ui\_data[row.UserID] = [(row.ItemID,pred\_mat[row.UserID,row.ItemID]),]  
 actual\_ui\_data[row.UserID] = [(row.ItemID,actual\_mat[row.UserID,row.ItemID]),]  
 else:  
 pred\_ui\_data[row.UserID].append((row.ItemID,pred\_mat[row.UserID,row.ItemID]))  
 actual\_ui\_data[row.UserID].append((row.ItemID,actual\_mat[row.UserID,row.ItemID]))  
   
 cnt = 0  
 tot = 0  
 for user in pred\_ui\_data.keys():  
 pred\_ui\_data[user] = sorted(pred\_ui\_data[user],key=lambda x: x[-1],reverse = True)  
 actual\_ui\_data[user] = sorted(actual\_ui\_data[user],key=lambda x: x[-1],reverse = True)  
   
 pred\_ranking = dict()  
 actual\_ranking = dict()  
 for i in range(len(pred\_ui\_data[user])):  
 pred\_ranking[pred\_ui\_data[user][i][0]] = i+1  
 actual\_ranking[actual\_ui\_data[user][i][0]] = i+1  
 x = []  
 y = []  
 for key in pred\_ranking.keys():  
 x.append(pred\_ranking[key])  
 y.append(actual\_ranking[key])  
 if len(x) > 1 and len(y) > 1 :  
 cnt += 1  
 tot += pearsonr(x, y)[0]  
 if cnt > 0:  
 return tot/cnt  
 else:  
 print("Error : ....")

my\_path = root / "Pickled\_files" / "pred\_matrix\_collab"  
dbfile = open(my\_path, 'rb')   
pred\_matrix = pickle.load(dbfile)  
dbfile.close()

topKPrecision(actual\_matrix,pred\_matrix,rating\_test,10)

0.642648067688087

spearmanCoeff(actual\_matrix,pred\_matrix,rating\_test)

0.04138696913650746

my\_path = root / "Pickled\_files" / "pred\_basline"  
dbfile = open(my\_path, 'rb')   
pred\_matrix = pickle.load(dbfile)  
dbfile.close()

topKPrecision(actual\_matrix,pred\_matrix,rating\_test,10)

0.7666590441344756

spearmanCoeff(actual\_matrix,pred\_matrix,rating\_test)

0.3635990667830773

my\_path = root / "Pickled\_files" / "SVD"  
dbfile = open(my\_path, 'rb')   
pred\_matrix = pickle.load(dbfile)  
dbfile.close()

topKPrecision(actual\_matrix,pred\_matrix,rating\_test,10)

0.6103361536702476

spearmanCoeff(actual\_matrix,pred\_matrix,rating\_test)

0.009268828494600173

my\_path = root / "Pickled\_files" / "SVD90"  
dbfile = open(my\_path, 'rb')   
pred\_matrix = pickle.load(dbfile)  
dbfile.close()

topKPrecision(actual\_matrix,pred\_matrix,rating\_test,10)

0.610884975989023

spearmanCoeff(actual\_matrix,pred\_matrix,rating\_test)

0.008314632507753952

my\_path = root / "Pickled\_files" / "CUR90"  
dbfile = open(my\_path, 'rb')   
pred\_matrix = pickle.load(dbfile)  
dbfile.close()

topKPrecision(actual\_matrix,pred\_matrix,rating\_test,10)

0.6075691744797624

spearmanCoeff(actual\_matrix,pred\_matrix,rating\_test)

0.0010459932683929657

my\_path = root / "Pickled\_files" / "CUR100"  
dbfile = open(my\_path, 'rb')   
pred\_matrix = pickle.load(dbfile)  
dbfile.close()

topKPrecision(actual\_matrix,pred\_matrix,rating\_test,10)

0.6119826206265719

spearmanCoeff(actual\_matrix,pred\_matrix,rating\_test)

0.0033905042404267717

### Tabulated Summary

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Recommender System Technique | RMSE | Precision on top K | Spearman Correlation | Prediction Time (in seconds) |
| Collaborative Filtering | 1.1557051129397395 | 0.642648067688087 | 0.04138696913650746 | 235.235813 |
| Collaborative Filtering with baseline | 0.9212280220671454 | 0.7666590441344756 | 0.3635990667830773 | 235.235813 |
| SVD | 1.188154215448053 | 0.6103361536702476 | 0.009268828494600173 | 233.878857 |
| SVD with 90% energy | 1.1767304502722231 | 0.610884975989023 | 0.008314632507753952 | 247.880419 |
| CUR | 1.1781062652249525 | 0.6119826206265719 | 0.0033905042404267717 | 220.19741 |
| CUR with 90% energy | 1.1851374361914413 | 0.6075691744797624 | 0.0010459932683929657 | 223.776183 |
|  |  |  |  |  |

Group Members

1. Milind Jain – 2020A7PS0153H
2. M Bhargav – 2020A7PS0025H
3. Ashwin Naveen Pugalia – 2020A7PS1080H
4. Shubh Badjate – 2020A7PS0028H