Movie Recommender System

I. ABOUT RECOMMENDER SYSTEMS

Users of various video hosting platforms and ecommerce platforms might have a preference towards specific items. In order to increase item accessability and improve overall user experience it is imperative that users are suggested to view certain items depending on the items they have already viewed, bought or purchased. Apart from improving user experience this might have commercial implications as well e.g promoting certain items which might otherwise may not have much viewership.

In the retail market this might entail to suggesting items that are of direct relevance to items that have been bought. As an example consider that someone has bought bread. Then it is expected that the user might buy butter as well. The system might then recommend butter. This is a simple example of a recommender system. In this context the machine learning model used may be the Apriori model which learns the conditional probabilty $P(buy\ butter|bread\ has\ been\ bought)$.

In video or movie hosting platforms, a viewer might be recommended movies based on his/her inclination towards viewing movies of particular genres. In this case one might look for movies that are similar to movies that have already been watched or rated. Following this idea one might start by definining a distance function between two movies. Movies that are close to the movies that have been already watched, in this sense, may then be recommended. These are the so called item based filtering models. One may also consider a distance between users that depend on their age, occupation etc. One might then recommended movies that have been watched by user which are close to the user under consideration. These are the user based filtering problems. Approches that combine both user and movie features are the Collaborative filtering models.

Recommender system are therefore learning models that learn the taste of an user depending on the rating or viewing history.

II. DATA DESCRIPTION AND EXPLORATION

The data set we will be analysing and modelling is popularly known as the movie lens data set available at the Movie lens website. MovieLens is run by GroupLens, a research lab at the University of Minnesota that develops new experimental tools and interfaces for data exploration and recommendation. It is non-commercial, and free of advertisements.

The dataset contains four importable data files which we will analyse shortly. It also contains five random train-test split of the user ratings. We will be using these to obtain a measure of performance for the models. The initial analysis will require the following libraries.

```
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt
from datetime import datetime
from datetime import timedelta
import time
from sklearn.svm import SVR
```

The files were located in Downloads of the home folder. The path has to be appropriately chosen for a different user.

```
query=pd.read_csv("~/Downloads/ml-100k/u.data",header=None,delimiter="\t",\
        names=["user id", "item id","rating","timestamp"])
genre=pd.read_csv("~/Downloads/ml-100k/u.genre",header=None,delimiter="|")
user=pd.read_csv("~/Downloads/ml-100k/u.user",header=None,delimiter="|")
item=pd.read_csv("~/Downloads/ml-100k/u.item",header=None,delimiter="|",\
        encoding='ISO-8859-1')
```

A. The Item file

The **item** file consists of a list of movies along with descriptions. The useful information are only the date of release and the genre. The imdb links given are broken and therefore cannot be used for webscrapping the imdb ratings. The release date will be useful in the latter part of the analysis as will be seen shortly.

```
with pd.option_context("display.max.rows",4,"display.max.columns",10):
        print(item)
```

```
        0                                        1            2    3   \
0       1                          Toy Story (1995)  01-Jan-1995 NaN
1       2                          GoldenEye (1995)  01-Jan-1995 NaN
...   ...                                       ...          ...  ..
1680  1681                      You So Crazy (1994)  01-Jan-1994 NaN
1681  1682  Scream of Stone (Schrei aus Stein) (1991)  08-Mar-1996 NaN

                                                     4  ...  19  20  21  22  23
0     http://us.imdb.com/M/title-exact?Toy%20Story%2...  ...   0   0   0   0   0
1     http://us.imdb.com/M/title-exact?GoldenEye%20(...  ...   0   0   1   0   0
...                                                 ...  ...  ..  ..  ..  ..  ..
1680  http://us.imdb.com/M/title-exact?You%20So%20Cr...  ...   0   0   0   0   0
1681  http://us.imdb.com/M/title-exact?Schrei%20aus%...  ...   0   0   0   0   0

[1682 rows x 24 columns]
```

The release date column will be converted to a pandas datetime data type. It can then be seen that one of the movies does not have a valid release date.

```
item[2]=pd.to_datetime(item[2])
with pd.option_context("display.max.rows",10,"display.max.columns",10):
        print(item[item[0]==267])
```

```
       0        1    2    3    4  ...  19  20  21  22  23
266  267  unknown NaT NaN  NaN  ...   0   0   0   0   0
```

```
[1 rows x 24 columns]
```

The missing data is populated with the date when it was first rated. This can be found by listing the ratings for the movie and looking for the earliest date when it was rated. The date of the first rating can be found from the query file as follows.

```
with pd.option_context("display.max.rows",10,"display.max.columns",10):
        print(query[query["item id"]==267])
```

```
       user id  item id  rating  timestamp
2172       130      267       5  875801239
3781         5      267       4  875635064
7245       268      267       3  875742077
12475      297      267       3  875409139
14756      319      267       4  875707690
15292        1      267       4  875692955
49295      532      267       3  875441348
93523      833      267       1  875655669
99723      422      267       4  875655986
```

The smallest timestamp is then converted to a date time object. It turns out that the date corresponding to the lowest timestamp is “1997-09-28”. Thus we set the release date to this value.

```
item.iloc[266,2]=pd.to_datetime("1997-09-28")
item.drop([1,3,4],axis=1,inplace=True)
item[2]=item[2].apply(lambda x: datetime.timestamp(x))
```

```
       0       1          2   3    4   ...  19  20  21  22  23
266  267  unknown 1997-09-28 NaN  NaN  ...   0   0   0   0   0

[1 rows x 24 column
```

The irrelevant columns are then dropped and the whole release column is converetd into a timestamp.

```
with pd.option_context("display.max.rows",4,"display.max.columns",10):
        print(item)
```

```
        0              2    5    6    7   ...  19   20   21   22   23
0         1  788898600.0    0    0    0   ...   0    0    0    0    0
1         2  788898600.0    0    1    1   ...   0    0    1    0    0
...     ...          ...   ..   ..   ..   ...  ..   ..   ..   ..   ..
1680   1681  757362600.0    0    0    0   ...   0    0    0    0    0
1681   1682  826223400.0    0    0    0   ...   0    0    0    0    0

[1682 rows x 21 columns]
```

B. The Query File

This is a collection of ratings given by various users, the item rated and the time at which the rating are contained in this data frame.

```
with pd.option_context("display.max.rows",4,"display.max.columns",10):
        print(query)
```

```
        user id  item id  rating  timestamp
0           196      242       3  881250949
1           186      302       3  891717742
...         ...      ...     ...        ...
99998        13      225       2  882399156
99999        12      203       3  879959583
```

```
[100000 rows x 4 columns]
```

An exploration of the dataframe will show that all users have one or more rating and all movies have been rated atleast once. The timestamp column gives the time at which the rating was given. To this data frame we will append a column which we will call the ‘lapse’. This will indicate the age of the movie at the time of the rating.

$$lapse = time\ of\ rating - date\ of\ release \tag{1}$$

The following code performs this task. The process is also parallelizable. We therefore include the corresponding code that can be run on the GPU.

```
a=[]
for i in range(len(query)):
        a.append(query.iloc[i,3]-item.iloc[query.iloc[i,1]-1,1])
query["lapse"]=a
del a
with pd.option_context("display.max.rows",10,"display.max.columns",10):
        print(query)
```

```
#GPU Code
@cuda.jit
def lapse(A,B,C,D):
        row=cuda.grid(1)
        if row<len(B):
                A[row]=B[row]-C[D[row]-1]

A=cp.zeros(len(query))
B=cp.asarray(query.iloc[:,3].values)
D=cp.asarray(query.iloc[:,1].values)
C=cp.asarray(item.iloc[:,1].values)

lapse[200,500](A,B,C,D)
query["lapse"]=cp.asnumpy(A)
del A,B,C,D
```

```
with pd.option_context("display.max.rows",4,"display.max.columns",10):
        print(query)
```

```
       user id  item id  rating  timestamp        lapse
0          196      242       3  881250949   27206749.0
1          186      302       3  891717742   39660742.0
...        ...      ...     ...        ...          ...
99998       13      225       2  882399156   33366156.0
99999       12      203       3  879959583  185755383.0

[100000 rows x 5 columns]
```

An important point to note here is that the 'lapse' column contains entries which are negative. This means that the rating was given before the movie was released. This possibility arises if the movie was rated seeing the trailer for example. One way to hande this is to correct for this by defining a corrected lapse by either shifting the release date of the movie to the date when it was first rated or by shifting the lapse column by an amount equal to the smallest element of the lapse column corresponding to a movie. We will however not do either of this and will model the negative values in an appropriate way.

1. Creating the rating matrix

The rating matrix is a matrix whose rows are the users and the columns are the movies. The entries of the matrix are the ratings. The matrix is evidently quite sparse. The following code creates the rating matrix from the query table. The corresponding GPU code is also given.

```
rating=np.zeros((943,1682))
for i in range(len(query)):
        rating[query.iloc[i,0]-1,query.iloc[i,1]-1]=query.iloc[i,2]
rating
```

```
#GPU Code
@cuda.jit
def rating_func(A,B,C,D):
        row=cuda.grid(1)
        if row<len(B):
                A[B[row]-1,C[row]-1]=D[row]

A=cp.zeros((943,1682))
B=cp.asarray(query.iloc[:,0].values)
C=cp.asarray(query.iloc[:,1].values)
D=cp.asarray(query.iloc[:,2].values)

rating_func[200,500](A,B,C,D)
rating=cp.asnumpy(A)
del A,B,C,D
rating
```

```
array([[5., 3., 4., ..., 0., 0., 0.],
       [4., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [5., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 5., 0., ..., 0., 0., 0.]])
```

III. THE USER FILE

The **user** file contains information about users' age, location, and occupation. This will be particularly useful in recommending movies according to users features.

```
with pd.option_context("display.max.rows",4,"display.max.columns",10):
        print(user)
```

```
       0   1  2            3      4
0      1  24  M   technician  85711
1      2  53  F        other  94043
..   ...  .. ..          ...    ...
941  942  48  F    librarian  78209
942  943  22  M      student  77841

[943 rows x 5 columns]
```

The 4th column here corresponds to the pin codes. This have been used to obtain information about the state of residence of the user in the US. The information has been obtained by web scrapping the website https://www.zipdatamaps.com/. The corresponding code can be found in the appendix (D). The final DataFrame was stored in a csv file. We work by importing the saved file.

```
user=pd.read_csv("~/Downloads/ml-100k/user_state.csv",header=None)
with pd.option_context("display.max.rows",10,"display.max.columns",10):
        print(user)
```

```
       0   1  2            3      4           5
0      1  24  M   technician  85711     Arizona
1      2  53  F        other  94043  California
..   ...  .. ..          ...    ...         ...
941  942  48  F    librarian  78209       Texas
942  943  22  M      student  77841       Texas

[943 rows x 6 columns]
```

The age column here is a continuous variable. We will disctetize it using the **cont_to_disctrete** function given in appendic (C). The file is then saved as **user_full.csv** We will work with this modified version of the user data frame.

```
user=pd.read_csv("~/Downloads/ml-100k/user_full.csv",header=None)
with pd.option_context("display.max.rows",4,"display.max.columns",10):
        print(user)
```

```
       0      1  2           3      4           5
0      1  20-25  M  technician  85711     Arizona
1      2  50-55  F       other  94043  California
..   ...    ... ..         ...    ...         ...
941  942  45-50  F   librarian  78209       Texas
942  943  20-25  M     student  77841       Texas

[943 rows x 6 columns]
```

IV. WEIGHT ACCORDING TO MOVIE AGE

To understand what we mean by weight according to movie age. Let us start by analysing the distribution of the lapse column. In other words we would like to find the distribution of the age of rated movies at the time of the rating. A more clear explanation can be given as follows. Given a time window we would like to find the number of ratings whose timestamp lies within this window.

```
X=query.copy()
X.sort_values(by="timestamp",inplace=True,ascending=False)
plt.hist(X["lapse"]);
plt.xlabel("age")
plt.ylabel("no. of ratings")
```

The histogram here shows the distribution of movie ages at the time of rating. Clearly there is a bias towards rating recent movies i.e movies whose ages are less or those which have been recently released. An exploration of the distribution of all existing movies (not only the rated movies) in different age intervals show that there are more newer movies in the sample. So, the question is whether there is actually a bias towards rating newer.

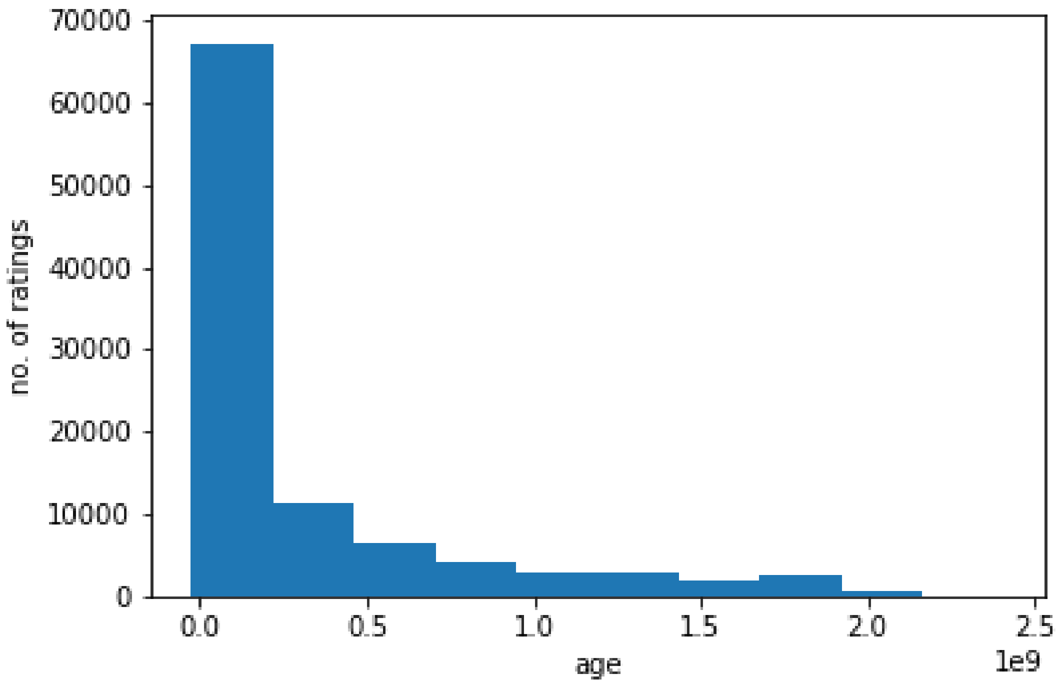
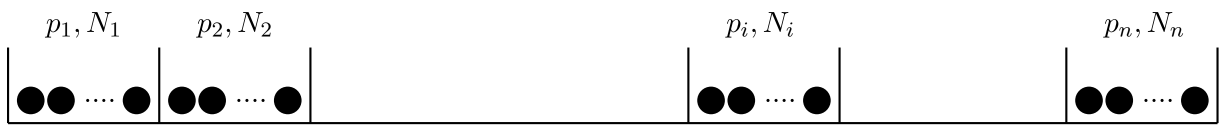


FIG. 1

To find this out lets consider the following model. Assume, that there are n boxes each representating an age interval. The ith box is assumed to contain N_i balls (representing N_i movies in ith age interval). Assume that the probalility of choosing a movie of age in the ith interval is p_i. Then the probability of choosing any movie in the i th interval is $N_i p_i$. This, on being equated to the obtained distribution of ratings should give us a sample estimate of p_i. If all the p_is are equal then there is no preference for rating newer movies. But there



is an issue here. N_i's in general change with time because the ages of movies change with times. Moreover there could be new movies releasing. So, in general the distribution of movie ages would be dynamic. In order to circumvent this issue we try to find a sequence of rating events where the N_i remains constant. The final analysis will be done with the longest such sequence. The age interval is taken to be one year.

The following code code finds the movie age distribution at all the rating events i.e at any given rating event the no. movies available for rating is found. This includes movies that are yet to release that is movies whose release date is one year after the date of the rating event (we assume that the earliest that a trailer can come out, is one year before the release). This allows for incorporating trailers as well. The available movies are then distributed according to age. Whenever the distribution changes the index is stored in the array change_loc.

```
movie_pop_time=np.zeros((78,len(X)))
i=0
change_loc=[0]
for x in X["timestamp"].values:
        Y=item[item[2]<=(x+31536000)]
        current_date=x
        count=np.array([0 for i in range(78)])
        #date=[893286638]
        for y in Y[2].values:
                if (current_date-y)<0:
                        count[0]=count[0]+1
                else:
                        count[int((current_date-y)/31536000)+1]=\
                                count[int((current_date-y)/31536000)+1]+1
        if i !=0:
                if np.array_equal(count,movie_pop_time[:,i-1],equal_nan=False)==False:
                        change_loc.append(i-1)
                        #date.append(x)
        movie_pop_time[:,i]=count
        i=i+1
        print(("steps:{}/{}").format(i,len(X)),end="\r")
del Y,current_date,count,i
```

```
steps:100000/100000
```

```
#GPU Code
A=cp.zeros((len(X),78))
B=cp.asarray(X["timestamp"].values)
C=cp.asarray(item[2].values)
```

```
@cuda.jit
def movie_pop(A,B,C):
        row,column=cuda.grid(2)
        if row<A.shape[0] and column<78:
                A[row,column]=counter(C,B[row],column)

@cuda.jit(device=True)
def counter(Y,x,i):
        count=0
        if i==0:
                for y in Y:
                        #print(i,y)
                        if (x-y)<0 and y<=(x+31536000):
                                count=count+1
        else:
                for y in Y:
                        #print(i,y)
                        if (x-y)>=0 and int((x-y)/31536000)+1==i and y<=(x+31536000):
                                count=count+1
        return(count)

movie_pop[(6250,5),(16,16)](A,B,C)
movie_pop_time=np.transpose(cp.asnumpy(A))
del A,B,C
change_loc=[0]
for i in range(100000):
        if i !=0:
                if np.array_equal(movie_pop_time[:,i],movie_pop_time[:,i-1],\
                        equal_nan=False)==False:
                        change_loc.append(i-1)
```

The longest sequence of rating events over which the distribution does not change is found by taking the difference of consecutive elements of the change_loc array. The routine returns the location of the change in the change_loc array and the length of the sequence of ratings

```
length=0
loc=0
for i in range(len(change_loc)-1):
        if (change_loc[i+1]-change_loc[i])>length:
                length=change_loc[i+1]-change_loc[i]
                loc=i
print(loc,length)
```

```
82 7292
```

The following gives the index of the ratings, in the ratings data frame, between which the distribution does not change.

```
change_loc[82],change_loc[83]
```

```
(64986,72278)
```

```
movie_rating_count=[0 for i in range(78)]
for x in X["lapse"].values[64986+1:72278+1]:
if x<0:
        movie_rating_count[0]=movie_rating_count[0]+1

if x>=0 and int(x/31536000)<79:

        movie_rating_count[int(x/31536000)+1]=movie_rating_count[int(x/31536000)+1]+1
```

```
movie_age_weight=[]
for i in range(1,69):
        movie_age_weight.append(movie_rating_count[i]/movie_pop_time[i,64986+1])
```

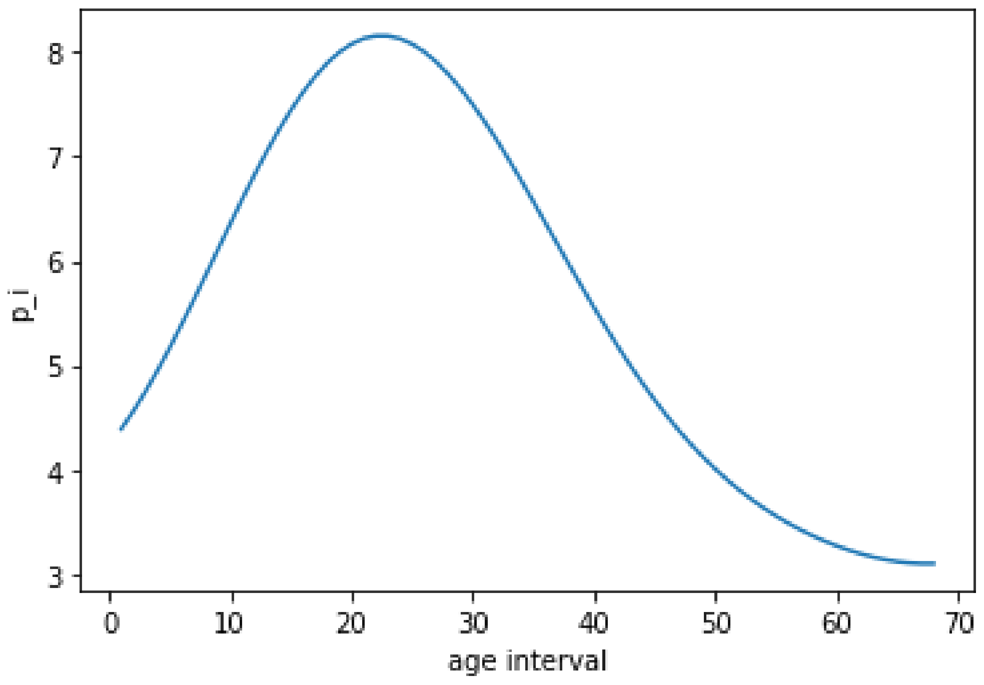


FIG. 2

```
model=SVR(degree=10)
model.fit(np.array([i for i in range(1,69)]).reshape(-1,1),movie_age_weight[:68])
movie_age_weight_fit=model.predict(np.array([i for i in range(1,69)]).reshape(-1,1))
plt.plot([i for i in range(1,69)],movie_age_weight_fit)
```

V. K NEAREST NEIGHBOUR (KNN)

The idea behind the K nearest Neigbour models is to define a distance function between users depending on user features or between movies depending on movie genres. Based on this we can divide this model into two classes item based or user based.

A. User Based

Let us assume that we have two users u_i and u_j who has been represented on a vector space of user features. Based on this we can define a distance function between them. Usually this is the cosine distance $1 - \frac{(u_i,u_j)}{||u_i||||u_j||}$. If u_i is close to u_j then we can recommend the movies rated by u_j to u_i. As is apparent from the data set the users have several features. Let us assume that each of these features can be represent on a vector space V_i. To represent all these features together we construct the product space $\bigotimes V_i$. The function **feature_matrix** is used to construct the matrix, user against feature for each feature and then these matrices are concatenated along the columns.

```
X=feature=hpgpu.feature_matrix_construct(features=[user[1].unique(),user[2].unique(),\
        user[3].unique(),user[5].unique()],columns_index=[1,2,3,5],df=user)
```

We will build a model with these features and evaluate it over the five train test splits.

```
neigh = NearestNeighbors(n_neighbors=40,metric="cosine")
neigh.fit(X)
neigh.kneighbors([X[10,:]])
```

```
(array([[0.  , 0.  , 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.5 , 0.5 , 0.5 ,
0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 ,
0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 ,
0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 ]]),
array([[ 10, 718,  17, 856, 759, 555, 166, 277, 416, 154, 701, 858, 150,
420, 148, 423, 900, 164, 436, 708, 860, 449, 687, 169,  78, 651,
504, 891, 493, 489, 884, 101, 668,   1, 459, 456, 119, 329, 128,
332]]))
```

This is a sample output for the user with user id 11. We will take all the nearest neighbours, that is users at distance zero from user 11 and recommend all those movies that has been rated 5 by them. We can then find the number of matches between this list and list of movie ratings of user 11 in the test set. The ratio of no. of matches to the total number movie‘s in the test set gives an accuracy score.

We estimate the total accuracy score i.e the ratio of total no. of matches to the total no. of movies in the test set over all users. The results for each of the five train test split is tabulated below (table I). The scores roughly show a two percent match. Note that query here does not refer to the full query file but the train set only and query_test refers to the test set only.

split 1	split 2	split 3	split 4	split 5
0.04565	0.05327	0.06690	0.05184	0.05652

TABLE I: Performance of user based filtering

```
for k in range(len(user)):
        print(k,end="\r")
        Y=neigh.kneighbors([X[k,:]])
        reco=[]
        for i in range(len(Y[0][0])):
                if Y[0][0][i]==0:
                reco.append(Y[1][0][i])

        mreco=[]
        tot=0
        deno_tot=0
        for x in reco:
                if x !=k:
                tot=tot+(len(query[query["user id"]==x+1].\
                merge(query_test[query_test["user id"]==11],\
                how="inner", left_on="item id",right_on="item id")))
                deno_tot=deno_tot+len(query_test[query_test["user id"]==x+1])
print(tot/deno_tot)
```

B. Item Based

In the item based approach the starting point is the movie feature matrix. Here we have only the movie genre information. Given a particular user the distance of all the movies from the set of movies that has been already rated is found via KNN. The un rated movies are then sorted according to these distances, in descending order. The first n no. of movies are recommended, where n is the no. of data points, for the particular user, in the test set.

The ratio of the no. of recommendation that have matched with those in the test set to the total no of movies in the test set can be taken to be the measure of accuracy. The numbers for each train test split is given in table II.

```
# Generating the movies vs genre matrix
genre_movie_embedding=item.iloc[:,2:].values
```

```
# Filtering out those item that have already been rated
def item_based(user_id,query,query_test,item):
#print(genre_movie_embedding)
        genre_movie_embedding_train=[]
        for y in query[query["user id"]==user_id]["item id"].values:
                genre_movie_embedding_train.append(genre_movie_embedding[y-1,:])

        neigh = NearestNeighbors(n_neighbors=1,metric="cosine")
        neigh.fit(genre_movie_embedding_train)



# Glink is a class of graph nodes and edge weights put in a tuple (,). The float
#comparison operators are polymorhed into this class therefore these data structures
#can be sorted by any sorting algorithm based on the second
#entry of the tuple (,....).



        distance=[]
        for i in range(len(item)):
                if item.iloc[i,0] not in query[query["user id"]==user_id]\
                                                ["item id"].values:
                        distance.append(glink((item.iloc[i,0],neigh.kneighbors\
                                ([genre_movie_embedding[i,:]])[0][0,0])))



        distance=np.sort(distance)
        reco=[]
        for i in range(50):
                reco.append(distance[i].link[0])
```

split 1	split 2	split 3	split 4	split 5
0.0337	0.02565	0.02245	0.0198	0.02205

TABLE II: Performance of item based filtering

```
        match=0
        for x in reco:
                if x in query_test[query_test["user id"]==user_id]["item id"].values:
                        match=match+1
        return(match,len(query_test[query_test["user id"]==user_id]))
```

```
temp1=0
temp2=0
for x in user[0].values:
        print(x,end="\r")
        out=item_based(user_id=x,query=query,query_test=query_test,item=item)
        temp1=temp1+out[0]
        temp2=temp2+out[1]
# print(out[0],out[1])
print(temp1/temp2)
```

VI. LATENT CLASS MODELS, MATRIX FACTORISATION AND COLLABORATIVE FILTERING

To understand why matrix factorisation is applicable to recommender systems, consider the following contingency table [2].

		Mental Health Status			
Parental Socio econ. Status		Well 1	Mild Symptoms 2	Moderate Symptoms 3	Impaired 4
High	1	64	94	58	46
	2	57	94	54	40
	3	57	105	65	60
	4	72	141	77	94
	5	36	97	54	78
Low	6	21	71	54	71

The simplest model to reproduce this contingency table is to assume independence i.e the probability that an ibservation lies in i th row and j th column is $p_i^X p_j^Y$, where X, Y are the row and column variables and i, j are the states of these variables. This is usually the null-hypothesis,

$$H_0 : p_{ij}^{X,Y} - p_i^X p_j^Y = 0, \tag{2}$$

where $p_i^X = \sum_j p_{ij}^{X,Y}$ and $p_j^Y = \sum_i p_{ij}^{X,Y}$. The number of degree of freedom of this model can be calculated as follows. The full contigency table has $6 \times 4 - 1$ independent variables, because of the contraint $\sum_{i,j} p_{ij} = 1$. The model fitting variables p_i, p_j are 10 in number. There are however two contraints $\sum_i p_i^X = 1$ and $\sum_j p_j^Y = 1$ which reduces the number to 8. Thus the number of degrees of freedom is $23 - 8 = 15$. The χ^2 test gives the goodness of fit to be 45.99 and likelihood ratio to be 47.42. To understand these numbers let us write down the observed matrix

```
table=np.array([[64,94,58,46],[57,94,54,40],[57,105,65,60],[72,141,77,94],\
        [36,97,54,78],[21,71,54,71]])
print(table)
```

```
[[ 64  94  58  46]
 [ 57  94  54  40]
 [ 57 105  65  60]
 [ 72 141  77  94]
 [ 36  97  54  78]
 [ 21  71  54  71]]
```

The modelled table can then be obtained as

```
import helper_contingency as hpc
table_model=hpc.xy_independent(table)
```

```
[[ 48.45421687  95.01445783  57.13493976  61.39638554]
 [ 45.31024096  88.84939759  53.42771084  57.4126506 ]
 [ 53.07771084 104.08072289  62.58674699  67.25481928]
 [ 71.01686747 139.25783133  83.73975904  89.98554217]
 [ 49.00903614  96.10240964  57.78915663  62.09939759]
 [ 40.13192771  78.69518072  47.32168675  50.85120482]]
```

The metrics can then be calculated as,

$$\chi^2 = \sum_{i,j} \frac{(\mathcal{O}_{ij} - \mathcal{E}_{ij})^2}{\mathcal{E}_{ij}} \tag{3}$$

$$G^2 = 2\sum_{ij} \mathcal{O}_{ij} \log\left(\frac{\mathcal{O}_{ij}}{\mathcal{E}_{ij}}\right), \tag{4}$$

where $\mathcal{O}_{ij}$ and $\mathcal{E}_{ij}$ are observed and expected values, while G^2 refers to the likelihood ratio.

```
hpc.chi_square(table,table_model)
```

```
45.985258802994935
```

```
hpc.chi_square(table,table_model)
```

```
47.41784679189713
```

The next level of complexity is to assume a latent explanatory variable Z such that,

$$p^{X,Y}(i|j) = \sum_k p^{X,Z}(i|k)p^{Z,Y}(k|j), \tag{5}$$

i.e the variables X, Y are conditionally independent given X. The model can also be expressed in the following alternatve forms.

$$p_{ij}^{X,Y} = p^{X,Y}(i|j)p_j^Y = \sum_k p^{X,Z}(i|k)p^{Z,Y}(k,j) = \sum_k p^{X,Z}(i,k)p^{Z,Y}(j|k) \tag{6}$$

Let us say that this is the alternate hypothesis,

$$H_1 : p^{X,Y}(i|j) - \sum_k p^{X,Z}(i|k)p^{Z,Y}(k|j) = 0 \tag{7}$$

Suppose the latent variable has two states. Then the number of model parameter are $6 \times 2 + 4 \times 2$. Moreover there are four contraints for sum over the states of each variable X, Y, Z. Thus the number of degrees of freedom is $24 - (20 - 4) = 8$. The χ^2 test gives the goodness of fit to be 2.74 and likelihood ratio to be 2.75. To understand these numbers we will do a non-negative matrix factorization of the **table** matrix, $TABLE_{6\times4} = U_{6\times2}V_{2\times4}$, based on the Kullback-Liebler divergence [4].

```
import helper_cpu as hpcpu
table_model=hpcpu.nmf_kl(R=table,steps=1000,dim=2)
print(table_model)
```

The **table_model** array, i.e the learned matrix, comes out to be

```
[[ 62.21921827,  98.17523399,  56.26254683,  45.34300092],
 [ 59.20664449,  92.04034684,  52.54698996,  41.20601871],
 [ 58.21053671, 105.25934345,  62.26144074,  61.26867911],
 [ 70.02616893, 139.03034306,  83.80254715,  91.14094087],
 [ 36.07729539,  93.13297021,  58.60873949,  77.1809949 ],
 [ 21.26013622,  74.36176245,  48.51773585,  72.86036549]]
```

The metrics when calculated with this modelled table gives χ^2 to be 2.7430685865425657 and likelihood ratio to be 2.7462974898227053. The important point to note here is that the expression for the model essentially takes the form of a matrix multiplication and the problem of fitting the model is that of a Non-negative Matrix Factorization (NMF) of the matrix p_{ij}.

In the case of Recommender system the rating matrix works as a proxy for p_{ij}. Though it is not a probability matrix it works fine provided we do not impose any normalisation contraints as in the case of p_{ij} In fact it is expected that an ordinary matrix factorization (and not non-negative) should also work for the rating matrix. Having understood this we will go over to the actual description of the problem in hand.

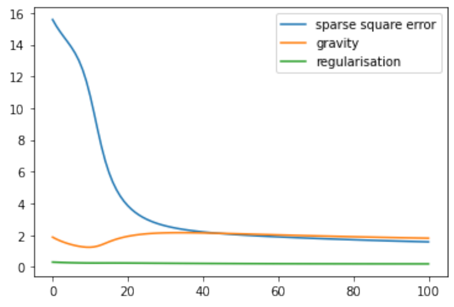
A. Collaborative Filtering

In the case of Collaborative filtering we assume that we only have the rating matrix R (say) and nothing else. It is then at our discretion to specify the number of states (D say) of a latent explanatory variable. The problem then reduces to a matrix factorization problem [3].

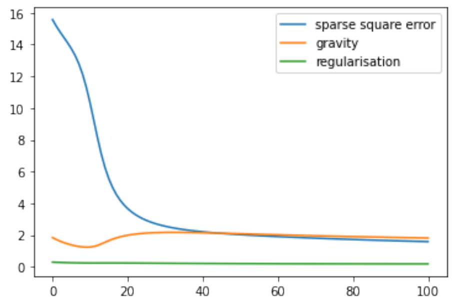
$$R_{M\times N} = U_{M\times D}V_{D\times N}, \tag{8}$$

where M, N are the number of users and movies respectively. The matrix R is in general sparse. However the learned matrix $\tilde{R}$ (say) is non-sparse. It is then on the basis of these non-zero rating in $\tilde{R}$ that new recommendations can be given. An important point to note here is that the latent variables could be anything, and does not need to be specified. They can be a combination of features that movies posses, genre, actors etc. or features that a user might posses, age, salary etc. Since the latent variables can in general be interpreted as a mixture of user and item feactures, it is called a Collaborative filtering model. We therefore analyse each of the train test split of the query data frame. We also device a way to analyse the perfomance of the model. In particular we consider a user and take n movies (from among those that have not been rated by the user already) with highest rating (obtained from the learned rating matrix $\tilde{R}$). The number n is taken such that it matches the number of ratings (for the particular user) in the test set. We then compare the recommended movies and the movies in the test set to see the number of natches. The ratio of the number of natches to the total length of the user's test set gives a evaluation metric. These number calculated over all users is tabulated in table III.

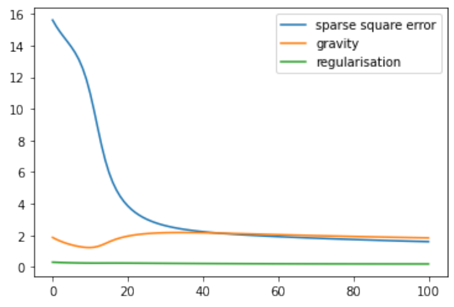
```
learned_rating=cp.asnumpy(fact_reg_sparse(steps=100,rating=rating,dim=30,gravity=1,\
weight=0.02,learn_rate=20)[0])
```



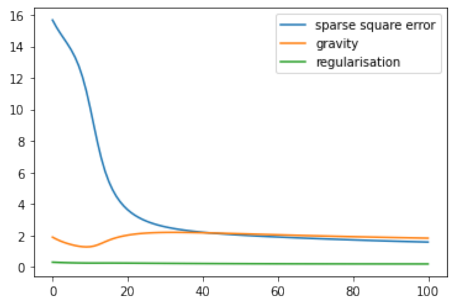
(a)



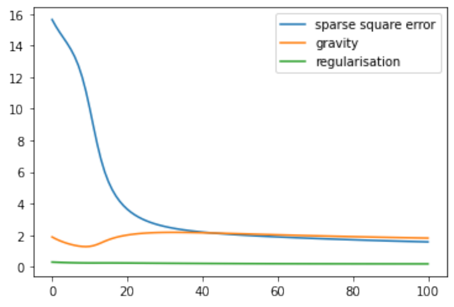
(b)



(c)



(d)



(e)

FIG. 3: Sparse Mean Square Error vs No. of Steps for Collaborative Filtering

```
temp1=0
temp2=0
for user_id in user[0].values:
        print(user_id,end="\r")
        X=pd.DataFrame(columns=["item id","score"])
        X["item id"]=[int(i+1) for i in range(rating.shape[1])]
        X["score"]=learned_rating[user_id-1,:]
        reco=recommendation(user_id=user_id,rank_matrix=X,\
                length=len(query_test[query_test["user id"]==user_id]))
        Y=pd.merge(reco,query_test[query_test["user id"]==user_id],\
                how="inner",left_on="item id",right_on="item id")
        temp1=temp1+len(query_test[query_test["user id"]==user_id])
        temp2=temp2+len(Y)
print(temp2/temp1)
```

TABLE III: Performance of Collaborative Filtering

split 1	split 2	split 3	split 4	split 5
0.32155	0.2946	0.265	0.26785	0.2749

B. Feature Based Collaborative Filtering

In the feature based collabaorative filtering models one of the matrices U or V is assumed to be already known. For example the matrix movue against genre is already known. V can therefore be taken to be the transpose this matrix. The learning model then learns U only. The rest of the analysis is similar to the previous case.

1. *Item based Collaborative filtering*

In this case we will assume that the movie genre matrix is known.

```
genre_movie_embedding=item.iloc[:,2:].values
learned_rating=hpgpu.fact_reg_sparse(learn_rate=100,rating=rating,steps=200,\
choice=1,v=np.transpose(genre_movie_embedding),weight=.02,gravity=1)[0]
learned_rating=cp.asnumpy(learned_rating)
```

2. *User based Collaborative Filtering*

In this case the U matrix is assumed to be known and equal to the user against user feature matrix. As in the case of KNN models the user feature matrix is constructed using

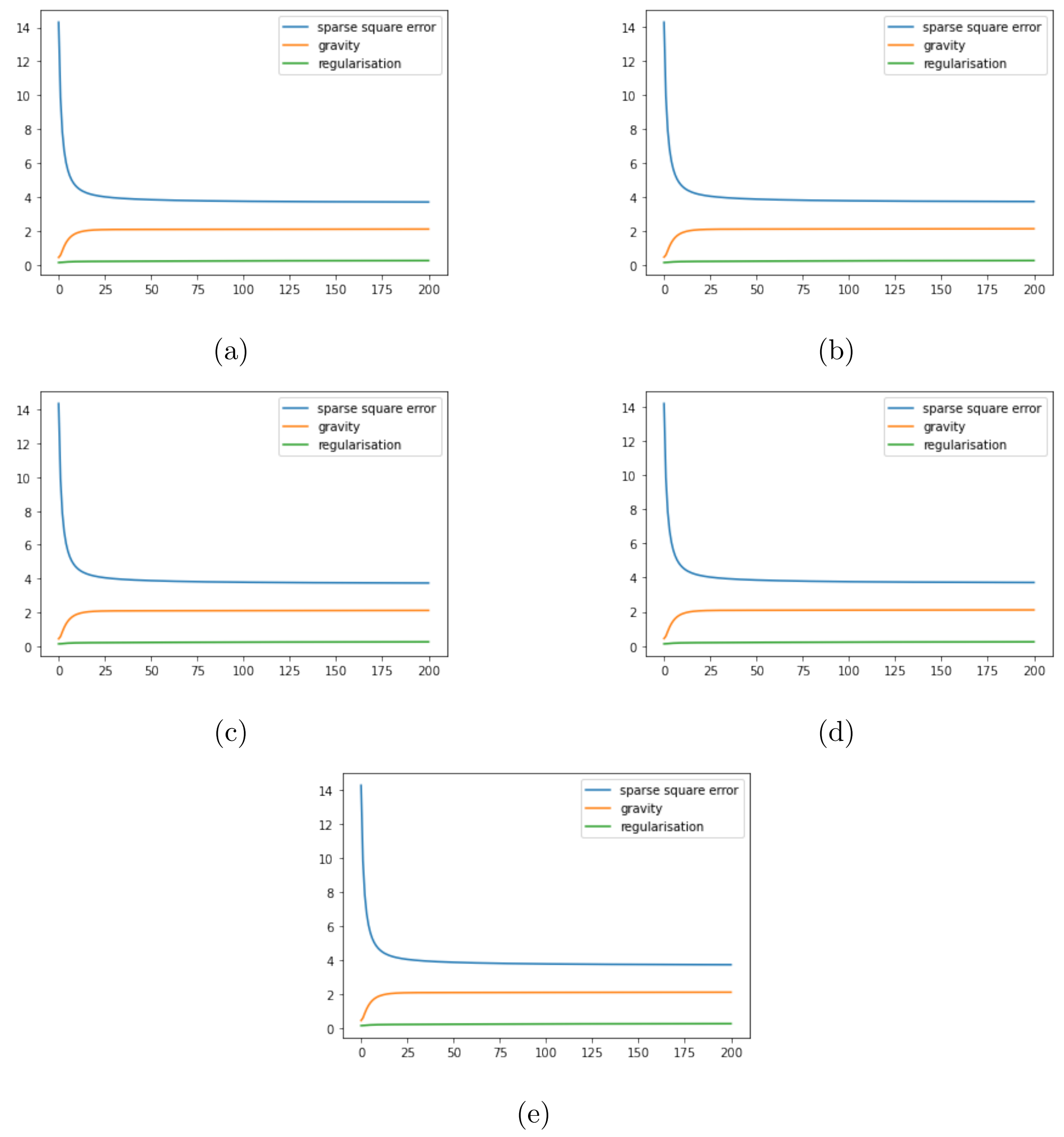


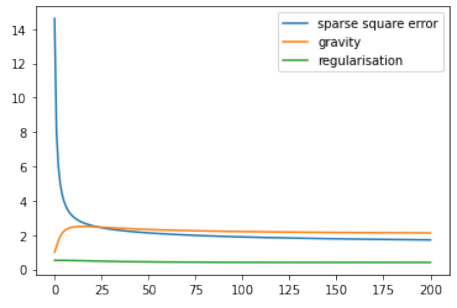
FIG. 4: Sparse Mean Square Error vs No. of Steps for Item Based Collaborative Filtering

split 1	split 2	split 3	split 4	split 5
0.13695	0.10655	0.0991	0.10145	0.10515

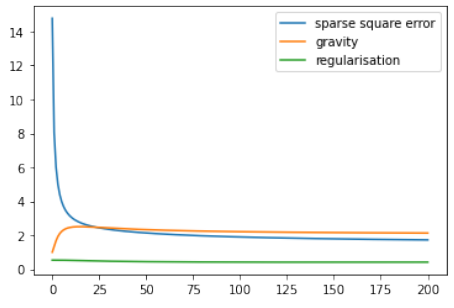
TABLE IV: Performance of item based collaborative filtering

the functions in appendix (C).

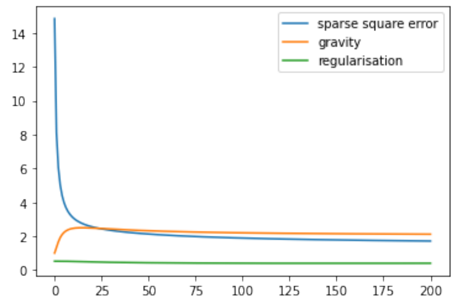
```
feature=hpgpu.feature_matrix_construct(features=[user[1].unique(),user[2].unique(),\
        user[3].unique(),user[5].unique()],columns_index=[1,2,3,5],df=user)

learned_rating=hpgpu.fact_reg_sparse(learn_rate=1000,rating=rating,steps=200,\
choice=2,u=feature,weight=.02,gravity=1)[0]
learned_rating=cp.asnumpy(learned_rating)
```



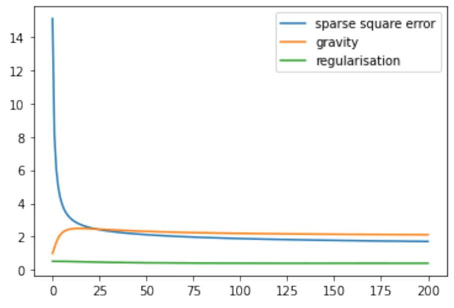
(a)



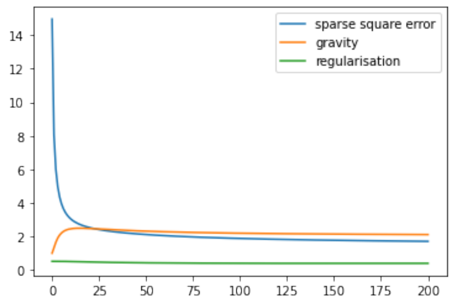
(b)



(c)



(d)



(e)

FIG. 5: Sparse Mean Square Error vs No. of Steps for User based Collaborative Filtering with Age and Occupation as the features

split 1	split 2	split 3	split 4	split 5
0.36165	0.3073	0.27195	0.2688	0.21225

TABLE V: Performance of user based collaborative filtering

VII. DISCUSSION

Some of the issues and possible improvements have been discussed in this section. We start with the movie weight by age parameter. It has alsready been discussed that the weight by movie age gives an estimate of the bias towards rating newer movies. The analysis clearly shows that there is a bias towards rating a movie that is about 20 years old. The main motive behind calculating this was to provide additional weight to the final ratings. In particular if we assume that this bias is independent of any other features user or movie then it is expected that simply by defining a new rating as $rating \times$ *movie weight by age* one could incude the bias into the model. However it was seen that by doing so most of the recommended movies are usually 20 years old. Thus this naive model simply does not work and one might require include movie age non-trivially into the model. This problem has been addressed before during the Netflix Competition. However we wish to provide further insights into the issue in the future.

Let us go over to issue in relation to matrix factorization. There are several ways the rating matrix be factorized e.g optimising for mean square error, regulated error, non negative matrix factorisation for square error, non negative matrix factorisation for Kullback Liebler divergence. After trying each of these methhods the regulated error was chosen. We provide the reason for it here. While implementing non negative matrix factorisation without regularisation the movies that are 'supposed to be recommended' had perfectly good scores around 5. However the movies that are not supposed to be recommended increase unboundedly thus giving overwhelmingly more weightage to these thus giving wrong recommendations. Ideally a solution to this is a bounded matrix factorisation where an extra condition, that the elements of $\tilde{R}$ is bounded between 0 and 5, is imposed. However very few algorithms exist for this problem ([5, 6] and references therein). It would interesting to device such an an algorithm and check if the results improve. The factorisation based on regulated error partially implements this strategy because the gravity term and the regularisation term tends to keep the elemenys of the matrices close to zero. This is concluded from the form the extra terms in eq. (B1)

Appendix A: Matrix Routines and Errors

In this appendix we will write down various helper functions that will be helpful throughout the project. We first discuss some function that will be used on matrices. The **sparser** function takes as input a sparse matrix. It then creates a matrix of the same dimension as the one that was given such that there are ones at the positions of non zero elements of the original matrix and zeros else where. It is useful in writing gradient descent algorithms applied on sparse matrices to be written as matrix operations rather than having to loop over individual elements. The function is implemented on the GPU. The **sparse_rep** function takes as input a sparse matrix and creates a list of tuples (i, j) where (i, j) are the locations of the non zero elements of the original matrix. Both these functions can be found in the file **helper.py**

Appendix B: Gradient Descent and Optimization algorithms

1. Matrix Factorization for regulated square error

The Matrix factorisation algorith that we will be using will essentially optimize a regulated or penalised l^2 error [3]. Let the set of non zero elements of the rating matrix R be denoted by Ω and $|\Omega|$ the no. of non zero elements.

$$L(U,V) = \frac{1}{|\Omega|}\sum_{i,j\ \in\ \Omega}\big(R_{ij} - \sum_k U_{ik}V_{kj}\big) + \frac{\lambda_r}{M}Tr(U^TU) + \frac{\lambda}{N}Tr(V^TV) + \frac{\lambda_g}{M\times N}Tr(UV), \tag{B1}$$

where λ_r is the regularisation coefficient and λ_g the gravity. The update steps are then given by:

$$V \to V + \lambda_l\Bigg(-\frac{2}{|\Omega|}U^TP.(R-UV) + \frac{2\lambda_g}{M\times N}U^TUV + \frac{2}{N}V\Bigg) \tag{B2}$$

$$V \to V + \lambda_l\Bigg(-\frac{2}{|\Omega|}P.(R-UV)V^T + \frac{2\lambda_g}{M\times N}UVV^T + \frac{2}{M}U\Bigg), \tag{B3}$$

where λ_l is the learning rate. Note that $P.(R - UV)$ is not a matrix operation. It just an element wise multiplication of P, the sparser matrix, whith $R - UV$. The algorith is implemented via the **fact_reg_sparse** function in **helper.py**.

Appendix C: Feature Manipulation routines

In this section we provide some of the functions used for feature engineering. The **cont_to_discrete** function converts a continuous feature age e.g into a discretised feature. This is essential if we want to model age by matrix factorisation. The **product_space** function gives constructs the basis vectors of product of several vector spaces. The **feature_matrix_construct** constructs the vectorized version of a categorical variable. The **recommendation** function takes the learned_matrix $\tilde{R}$ as input and ranks the unrated movies. The highest rated movies are then recommended. The codes can be found in the file **helper.py**

Appendix D: Web Scrapping the State Information

```
import requests
from bs4 import BeautifulSoup
state=[]
i=0
for x in user[4].values:
    webadd="https://www.zipdatamaps.com/"+str(x)
    print(i,end='\r')
    i=i+1
    r = requests.get(webadd)
    soup = BeautifulSoup(r.content, 'html.parser')
    s = soup.find('table', class_="table table-striped table-bordered table-hover\
             table-condensed")
    content = s.find_all('tr')
    for line in content:
            if "Zip Code State" in line.text:
                    print(line.text[14:])
        state.append(line.text[14:])
```

```
            break
user[5]=state
del state
user.to_csv(r"~/Downloads/ml-100k/user_state.csv",index=False,header=False)
```

[1] Movie Lens Website, “ The movielens dataset,”

[2] Allan L. McCutcheon, Jacques A. Hagenaars, ed. “Applied Latent Class Analysis.,” Cambridge University Press (2002).

[3] Google Machine Learning Course,

[4] Lee, D., Seung, H. “Learning the parts of objects by non-negative matrix factorization.” Nature 401, 788–791 (1999).

[5] S. Jiang, K. Li and R. Y. Da Xu, “Magnitude Bounded Matrix Factorisation for Recommender Systems,” in IEEE Transactions on Knowledge and Data Engineering, vol. 34, no. 4, pp. 1856-1869, 1 April 2022.

[6] Kannan, R., Ishteva, M. and Park, H. “Bounded matrix factorization for recommender system” Knowl Inf Syst 39, 491–511 (2014).