Name- Jatin Karthik

MAtriculation nr. -313301

Lab 10 Q2 and Q3

Exercise 2: Implementing basic matrix factorization (MF) technique

for recommender systems

In this task you are required to implement a matrix factorization (MF) technique for recommender systems. You are given a rating matrix Rn×m and you have to learn latent matrices P n×k and Qk×m, where n is the number of users, m is the number of items and k the latent dimensions. You can solve the MF problem by implementing Stochastic Gradient Descent (SGD) learning algorithms (Algorithm LearnLatentFactors on slide 29). Measure the prediction quality (the RMSE score) on the validation and test dataset. You can set 10%/10% of ratings aside for validation and testing.

In [7]:
```python
# I import all these libraries here but i use the scikit learn only in task 3
import numpy as np
import tensorflow as tf
from tensorflow import keras
import pandas as pd
import seaborn as sns
from pylab import rcParams
import string
import re
import matplotlib.pyplot as plt
import math
from matplotlib import rc
from google.colab import drive
from sklearn.model_selection import train_test_split
from collections import Counter, defaultdict
from sklearn.metrics import accuracy_score
import matplotlib.ticker as ticker
from math import sqrt


from sklearn.metrics import mean_squared_error


rcParams['figure.figsize'] = 14, 8

RANDOM_SEED = 3116

np.random.seed(RANDOM_SEED)
```

In [8]:
```python
drive.mount("/content/drive")
```
```
Mounted at /content/drive
```

In [9]:
```python
#loading the main dataset of userId and ratings
all_data = pd.read_csv('/content/drive/MyDrive/Recommender/u.data', sep=r"\s+", names=["userId","itemId","rating","timestamp"])
all_data
```

Out[9]:

	userId	itemId	rating	timestamp
0	196	242	3	881250949
1	186	302	3	891717742
2	22	377	1	878887116
3	244	51	2	880606923
4	166	346	1	886397596
...	...	...	...	...
99995	880	476	3	880175444
99996	716	204	5	879795543
99997	276	1090	1	874795795
99998	13	225	2	882399156
99999	12	203	3	879959583

100000 rows × 4 columns

In [84]:
```python
# loading the item part of the movie lens dataset
item_data = pd.read_csv('/content/drive/MyDrive/Recommender/u.item', sep="|", names=["movieid", "movietitle","releasedate","videoreleasedate",
              "IMDbURL","unknown","Action","Adventure", "Animation",
              "Children's","Comedy", "Crime", "Documentary","Drama","Fantasy",
              "Film-Noir","Horror","Musical", "Mystery","Romance","Sci-Fi",
              "Thriller", "War", "Western"], encoding='latin-1')
```

In [85]:
```python
ratings_df = all_data.pivot(
    index='userId',
    columns='itemId',
    values='rating'
)
ratings_df
```

Out[85]:

itemId	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	...	1643	1644	1645	1646	1647	1648
userId																																															
1	5.0	3.0	4.0	3.0	3.0	5.0	4.0	1.0	5.0	3.0	2.0	5.0	5.0	5.0	5.0	5.0	3.0	4.0	5.0	4.0	1.0	4.0	4.0	3.0	4.0	3.0	2.0	4.0	1.0	3.0	3.0	5.0	4.0	2.0	1.0	2.0	2.0	3.0	4.0	3.0	...	NaN	NaN	NaN	NaN	NaN	NaN
2	4.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2.0	NaN	NaN	4.0	4.0	NaN	NaN	NaN	NaN	3.0	NaN	NaN	NaN	NaN	NaN	4.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	4.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN
5	4.0	3.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	4.0	NaN	NaN	NaN	3.0	NaN	NaN	4.0	3.0	NaN	NaN	NaN	4.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	4.0	...	NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
939	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	5.0	NaN	NaN	NaN	NaN	NaN	5.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN
940	NaN	NaN	NaN	2.0	NaN	NaN	4.0	5.0	3.0	NaN	NaN	4.0	NaN	3.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN
941	5.0	NaN	NaN	NaN	NaN	NaN	4.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	4.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN
942	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	5.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN
943	NaN	5.0	NaN	NaN	NaN	NaN	NaN	NaN	3.0	NaN	4.0	5.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	4.0	4.0	4.0	NaN	NaN	4.0	4.0	NaN	NaN	4.0	NaN	NaN	NaN	NaN	NaN	NaN	3.0	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN

943 rows × 1682 columns

In [86]:
```python
# filling all the NA values with 0
ratings = ratings_df.fillna(0).values
```

Checking how much sparsity exists in Movie lens

In [87]:
```python
sparsity = float(len(ratings.nonzero()[0]))
sparsity /= (ratings.shape[0] * ratings.shape[1])
sparsity *= 100
print('{:.2f}%'.format(sparsity))
```
```
6.30%
```

Splitting the dataset into 80% train, 10% validation and 10%test dataset

In [88]:
```python
train, val, test = np.split(ratings_df.sample(frac=1, random_state=3116), [int(.8*len(ratings_df)), int(.9*len(ratings_df))])
train_list=np.array(train)
validate_list=np.array(val)
tets_list=np.array(test)
```

In [89]:
```python
#converting thme into array type
train=train.replace(np.nan,0)
train_list=np.array(train)
np.any(np.isnan(train_list))

val=val.replace(np.nan,0)
validate_list=np.array(val)

test=test.replace(np.nan,0)
test_list=np.array(test)
```

In [90]:
```python
# defining a RMSE function which gives mean sqaured error comparing the ground truth and predicted values

def rmse(prediction, ground_truth):
    prediction = prediction[ground_truth.nonzero()].flatten()
    ground_truth = ground_truth[ground_truth.nonzero()].flatten()
    return sqrt(mean_squared_error(prediction, ground_truth))
```

In [ ]:

Defining a class which initialises a function, normalises the P and Q matrices and implementing stochastic gradient descent

In [93]:
```python
class Recommender:



  def __init__(self, n_epochs=2, n_latent_features=3, lmbda=0.1, learning_rate=0.001):
    self.n_epochs = n_epochs
    self.n_latent_features = n_latent_features
    self.lmbda = lmbda
    self.learning_rate = learning_rate

  def predictions(self, P, Q):
    return np.dot(P.T, Q)

  def fit(self, X_train, X_val):
    m, n = X_train.shape

    #Normalising the dataset of movie lens and scaling it such that specific users don't always rate high or low

    self.P=np.random.normal(scale=1./self.n_latent_features, size=(self.n_latent_features, m))
    self.Q=np.random.normal(scale=1./self.n_latent_features, size=(self.n_latent_features, n))

    # self.P = 3 * np.random.rand(self.n_latent_features, m)
    # self.Q = 3 * np.random.rand(self.n_latent_features, n)

    self.train_error = []
    self.val_error = []

    users, items = X_train.nonzero()
    #users, items = X_train.shape
    #print(users, items)

#Stochastic gradient descent
    for epoch in range(self.n_epochs):
        for u, i in zip(users, items):
            error = X_train[u, i] - self.predictions(self.P[:,u], self.Q[:,i])
            self.P[:, u] += self.learning_rate * (error * self.Q[:, i] - self.lmbda * self.P[:, u])
            self.Q[:, i] += self.learning_rate * (error * self.P[:, u] - self.lmbda * self.Q[:, i])

        train_rmse = rmse(self.predictions(self.P, self.Q), X_train)
        val_rmse = rmse(self.predictions(self.P, self.Q), X_val)
        self.train_error.append(train_rmse)
        self.val_error.append(val_rmse)

    print("the final test rmse with epochs=", self.n_epochs,"latent dimensions=", self.n_latent_features,"lambda=", self.lmbda,"alpha=" ,self.learning_rate,"parameters is",val_rmse)
    return self

  def predict(self, X_train, user_index):
    y_hat = self.predictions(self.P, self.Q)
    predictions_index = np.where(X_train[user_index, :] == 0)[0]
    return y_hat[user_index, predictions_index].flatten()
```

Q) **optimize the hyper-parameters i.e. λ regularization constant, α learning rate, k latent dimensions**

Keeping the epochs to 25 as fixed and varying the alpha, lambda and latent dimensions i.e the hyperparameter optimisation.

Combinations used:- alpha=[0.0001,0.001,0.01,0.1] lambda=[0.01,0.1] latent dimensions=[3,5,8]

In [94]:
```python
recommender1 = Recommender()
recommender1.__init__(n_epochs=25, n_latent_features=3, lmbda=0.01, learning_rate=0.0001)
recommender1.fit(train_list, validate_list)
```
```
the final test rmse with epochs= 25 latent dimensions= 3 lambda= 0.01 alpha= 0.0001 parameters is 3.7085368315574447
```
Out[94]: `<__main__.Recommender at 0x7f2b39579190>`

In [ ]:
```python
recommender2 = Recommender()
recommender2.__init__(n_epochs=25, n_latent_features=3, lmbda=0.1, learning_rate=0.0001)
recommender2.fit(train_list, validate_list)
```
```
the validation rmse with epochs= 25 latent dimensions= 3 lambda= 0.1 alpha= 0.0001 parameters is 2.278171679475042
```
Out[ ]: `<__main__.Recommender at 0x7fad481c5790>`

In [41]:
```python
recommender3 = Recommender()
recommender3.__init__(n_epochs=25, n_latent_features=5, lmbda=0.01, learning_rate=0.0001)
recommender3.fit(train_list, validate_list)
```
```
the final test rmse with epochs= 25 latent dimensions= 5 lambda= 0.01 alpha= 0.0001 parameters is 4.058583280050861
```
Out[41]: `<__main__.Recommender at 0x7f2b3ae65ad0>`

In [ ]:
```python
recommender4 = Recommender()
recommender4.__init__(n_epochs=25, n_latent_features=5, lmbda=0.1, learning_rate=0.0001)
recommender4.fit(train_list, validate_list)
```
```
the validation rmse with epochs= 25 latent dimensions= 5 lambda= 0.1 alpha= 0.0001 parameters is 2.891815751568864
```
Out[ ]: `<__main__.Recommender at 0x7fad481638d0>`

In [ ]:
```python
recommender5 = Recommender()
recommender5.__init__(n_epochs=25, n_latent_features=5, lmbda=0.1, learning_rate=0.001)
recommender5.fit(train_list, validate_list)
```
```
the validation rmse with epochs= 25 latent dimensions= 5 lambda= 0.1 alpha= 0.001 parameters is 1.3784116075020219
```
Out[ ]: `<__main__.Recommender at 0x7fad481c5310>`

In [ ]:
```python
recommender6 = Recommender()
recommender6.__init__(n_epochs=25, n_latent_features=5, lmbda=0.01, learning_rate=0.001)
recommender6.fit(train_list, validate_list)
```
```
the validation rmse with epochs= 25 latent dimensions= 5 lambda= 0.01 alpha= 0.001 parameters is 1.4689259939462378
```
Out[ ]: `<__main__.Recommender at 0x7fad48184310>`

In [ ]:
```python
recommender7 = Recommender()
recommender7.__init__(n_epochs=25, n_latent_features=5, lmbda=0.01, learning_rate=0.01)
recommender7.fit(train_list, validate_list)
```
```
the validation rmse with epochs= 25 latent dimensions= 5 lambda= 0.01 alpha= 0.01 parameters is 1.2090601952853912
```
Out[ ]: `<__main__.Recommender at 0x7fad48184dd0>`

In [ ]:
```python
recommender8 = Recommender()
recommender8.__init__(n_epochs=25, n_latent_features=5, lmbda=0.1, learning_rate=0.01)
recommender8.fit(train_list, validate_list)
```
```
the validation rmse with epochs= 25 latent dimensions= 5 lambda= 0.1 alpha= 0.01 parameters is 1.147437249408723
```
Out[ ]: `<__main__.Recommender at 0x7fad481c5f10>`

In [ ]:
```python
recommender11 = Recommender()
recommender11.__init__(n_epochs=25, n_latent_features=8, lmbda=0.01, learning_rate=0.0001)
recommender11.fit(train_list, validate_list)
```
```
the validation rmse with epochs= 25 latent dimensions= 8 lambda= 0.01 alpha= 0.0001 parameters is 4.804520272102495
```
Out[ ]: `<__main__.Recommender at 0x7fad4818ee50>`

In [ ]:
```python
recommender12 = Recommender()
recommender12.__init__(n_epochs=25, n_latent_features=8, lmbda=0.1, learning_rate=0.0001)
recommender12.fit(train_list, validate_list)
```
```
the validation rmse with epochs= 25 latent dimensions= 8 lambda= 0.1 alpha= 0.0001 parameters is 4.038553881960411
```
Out[ ]: `<__main__.Recommender at 0x7fad482cbb10>`

In [ ]:
```python
recommender13 = Recommender()
recommender13.__init__(n_epochs=25, n_latent_features=8, lmbda=0.01, learning_rate=0.001)
recommender13.fit(train_list, validate_list)
```
```
the validation rmse with epochs= 25 latent dimensions= 8 lambda= 0.01 alpha= 0.001 parameters is 1.9484016280136647
```
Out[ ]: `<__main__.Recommender at 0x7fad481c6e50>`

In [ ]:
```python
recommender14 = Recommender()
recommender14.__init__(n_epochs=25, n_latent_features=8, lmbda=0.1, learning_rate=0.01)
recommender14.fit(train_list, validate_list)
```
```
the validation rmse with epochs= 25 latent dimensions= 8 lambda= 0.1 alpha= 0.01 parameters is 1.1972023532061669
```
Out[ ]: `<__main__.Recommender at 0x7fad4818e610>`

In [ ]:
```python
recommender15 = Recommender()
recommender15.__init__(n_epochs=25, n_latent_features=8, lmbda=0.01, learning_rate=0.01)
recommender15.fit(train_list, validate_list)
```
```
the validation rmse with epochs= 25 latent dimensions= 8 lambda= 0.01 alpha= 0.01 parameters is 1.3090677151549854
```
Out[ ]: `<__main__.Recommender at 0x7fad4818ef90>`

In [ ]:
```python
recommender17 = Recommender()
recommender17.__init__(n_epochs=25, n_latent_features=8, lmbda=0.1, learning_rate=0.1)
recommender17.fit(train_list, validate_list)
```
```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:46: RuntimeWarning: overflow encountered in multiply
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:45: RuntimeWarning: overflow encountered in multiply
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-87-08c097a0cdb2> in <module>()
      1 recommender17 = Recommender()
      2 recommender17.__init__(n_epochs=25, n_latent_features=8, lmbda=0.1, learning_rate=0.1)
----> 3 recommender17.fit(train_list, validate_list)

<ipython-input-71-94263e2183f9> in fit(self, X_train, X_val)
     46             self.Q[:, i] += self.learning_rate * (error * self.P[:, u] - self.lmbda * self.Q[:, i])
     47 
---> 48         train_rmse = rmse(self.predictions(self.P, self.Q), X_train)
     49         val_rmse = rmse(self.predictions(self.P, self.Q), X_val)
     50         self.train_error.append(train_rmse)

<ipython-input-62-91c37d0a009d> in rmse(prediction, ground_truth)
      2     prediction = prediction[ground_truth.nonzero()].flatten()
      3     ground_truth = ground_truth[ground_truth.nonzero()].flatten()
----> 4     return sqrt(mean_squared_error(prediction, ground_truth))

/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_regression.py in mean_squared_error(y_true, y_pred, sample_weight, multioutput, squared)
    437     """
    438     y_type, y_true, y_pred, multioutput = _check_reg_targets(
--> 439         y_true, y_pred, multioutput
    440     )
    441     check_consistent_length(y_true, y_pred, sample_weight)

/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_regression.py in _check_reg_targets(y_true, y_pred, multioutput, dtype)
     93     """
     94     check_consistent_length(y_true, y_pred)
---> 95     y_true = check_array(y_true, ensure_2d=False, dtype=dtype)
     96     y_pred = check_array(y_pred, ensure_2d=False, dtype=dtype)
     97 

/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py in check_array(array, accept_sparse, accept_large_sparse, dtype, order, copy, force_all_finite, ensure_2d, allow_nd, ensure_min_samples, ensure_min_features, estimator)
    798 
    799         if force_all_finite:
--> 800             _assert_all_finite(array, allow_nan=force_all_finite == "allow-nan")
    801 
    802     if ensure_min_samples > 0:

/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py in _assert_all_finite(X, allow_nan, msg_dtype)
    114             raise ValueError(
    115                 msg_err.format(
--> 116                     type_err, msg_dtype if msg_dtype is not None else X.dtype
    117                 )
    118             )

ValueError: Input contains NaN, infinity or a value too large for dtype('float64').
```

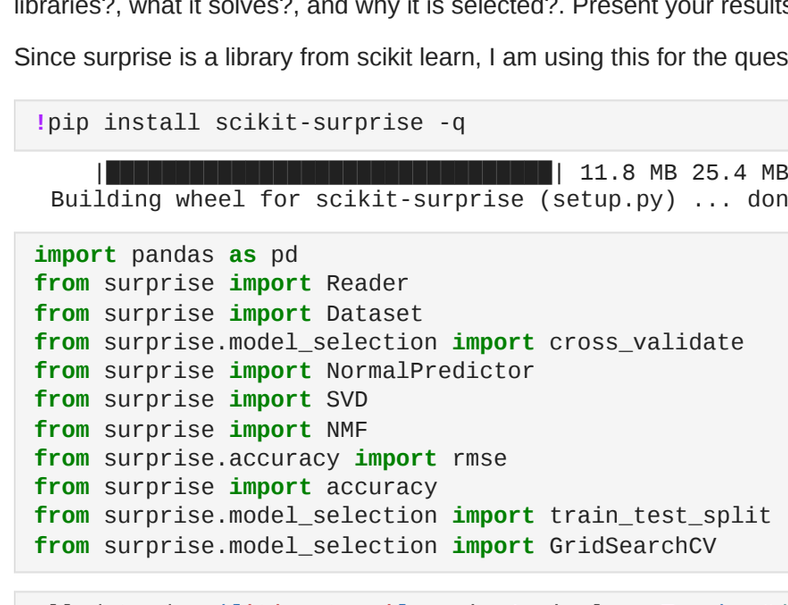
For the above, I get an error because the values are too large for float, so I stopped optimisation search after this.

Q) **Compute the validation RMSE**

The lowest rmse is achieved for recommender8 which is 1.1478 with n_latent_features=5, lmbda=0.1, learning_rate=0.01 hyperparameters, so I plot only that.

Plotting only the best train and validation rmse graph:

In [ ]:
```python
plt.plot(range(recommender8.n_epochs), recommender8.train_error, marker='o', label='Training Data');
plt.plot(range(recommender8.n_epochs), recommender8.val_error, marker='v', label='Validation Data');
plt.xlabel('Number of Epochs');
plt.ylabel('RMSE');
plt.legend()
plt.grid()
plt.show()
```



Now, looking at the various combinations of different learning rate, lambda and latent dimensions, we can clearly see that the rmse is minimum when n_latent_features=5, lmbda=0.1, learning_rate=0.01. Therefore, we shall calculate our test rmse finally on these set of hyperparameters.

We calculate our final rmse results on the test dataset from the optimal hyperparameters we get on the validation set

In [ ]:
```python
recommender_final = Recommender()
recommender_final.__init__(n_epochs=25, n_latent_features=5, lmbda=0.1, learning_rate=0.01)
recommender_final.fit(train_list, test_list)
```
```
the final test rmse with epochs= 25 latent dimensions= 5 lambda= 0.1 alpha= 0.01 parameters is 1.1209035594368975
```
Out[ ]: `<__main__.Recommender at 0x7fad47d18990>`

Q) **Compute the test RMSE**

So, the final test rmse is 1.12090

Exercise 3: Recommender Systems using matrix factorization sckitlearn

In this task you are required to use off-the-shelf libraries such as libmf or scikit-learn. You have to learn a matrix factorization model. Optimize the hyper parameters and perform a 3-fold cross validation. Compare your results with the results in task 1. List in detail which/how you used these libraries?, what it solves?, and why it is selected?. Present your results in form of plots and tables.

Since surprise is a library from scikit learn, I am using this for the question (and as i can use any off the shelf library). Please find another implementation of library at the end)

In [2]:
```python
!pip install scikit-surprise -q
```
```
     |████████████████████████████████| 11.8 MB 25.4 MB/s
  Building wheel for scikit-surprise (setup.py) ... done
```

In [3]:
```python
import pandas as pd
from surprise import Reader
from surprise import Dataset
from surprise.model_selection import cross_validate
from surprise import NormalPredictor
from surprise import SVD
from surprise import NMF
from surprise.accuracy import rmse
from surprise import accuracy
from surprise.model_selection import train_test_split
from surprise.model_selection import GridSearchCV
```

In [10]:
```python
all_data.drop(['timestamp'], axis=1, inplace=True)  #drop the timestamp column
reader = Reader(rating_scale=(1, 5)) #create a reader to load the dataset following
data = Dataset.load_from_df(all_data[['userId', 'itemId', 'rating']], reader)
```

Explaining how this methodology works:- Answers which, how questions??

Surprise library reduces the user-item interaction into the lower dimensional space latent matrix. However, there is no dimensionality reduction technique like SVD or PCA applied under the hood.

This method estimates the latent factor matrix and the bias termed minimizing the loss between the original explicit rating and the reconstructed prediction rating. Therefore, that is why it is called approach model-based.

To make a predicted rating, we can multiply the estimated latent factors matrix and add the bias term for the user or item like the below figure.

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T p_u$$

Where the u is denoting user index,

and i is denoting the item index.

r(hat)ui is the predicted rating of the user u on the item i

mu is the mean of all ratings

b is the bias term (baseline) for the user or item

qi is the item latent factor matrix

pu is the user latent factor matrix

Since we don't know the information about the user, then the term bu and pu will be assumed to be zero. Thus, the predicted rating of the new user will be the mean of all ratings plus the bi term, which means if we don't know the user, we will recommend them with the product with a high baseline term that we learned from the data. The same applies to the item terms bi and qi . If it's the new user and new item, we will have the predicted rating equal to the mean of all ratings.

The loss function used in the surprise library is:-

$$\sum_{r_{ui} \in R_{train}} \left(r_{ui} - \hat{r}_{ui}\right)^2 + \lambda \left(b_i^2 + b_u^2 + ||q_i||^2 + ||p_u||^2\right)$$

why is this library selected?

Provides various ready-to-use prediction algorithms such as baseline algorithms, neighborhood methods, matrix factorization-based (SVD, PMF, SVD++, NMF), and many others. Also, various similarity measures (cosine, MSD, pearson…) are built-in.

Makes it easy to implement new algorithm ideas. Provide tools to evaluate, analyse and compare the algorithms' performance.

Cross-validation procedures can be run very easily using powerful CV iterators (inspired by scikit-learn excellent tools), as well as exhaustive search over a set of parameters.

In [11]:
```python
benchmark = []
rmse_list=[]
# Iterate over the below two kinds of algorithms
# I tried out two most themost used methodologies in Matrix factorisation which are SVD and NMF


algorithms = [SVD(), NMF()]

print ("Attempting: ", str(algorithms), '\n\n\n')

for algorithm in algorithms:
    print("Starting: " ,str(algorithm))
    # Perform 3 fold cross validation and only use RMSE as performance metric
    results = cross_validate(algorithm, data, measures=['RMSE'], cv=3, verbose=False)
    # results = cross_validate(algorithm, data, measures=['RMSE','MAE'], cv=3, verbose=False)
    rmse_list.append(results['test_rmse'])

    # Get results & append algorithm name
    tmp = pd.DataFrame.from_dict(results).mean(axis=0)
    tmp = tmp.append(pd.Series([str(algorithm).split(' ')[0].split('.')[-1]], index=['Algorithm']))
    benchmark.append(tmp)
    print("Done: " ,str(algorithm), "\n\n")

print ('\n\tDONE\n')
```
```
Attempting:  [<surprise.prediction_algorithms.matrix_factorization.SVD object at 0x7f851ecc6ed0>, <surprise.prediction_algorithms.matrix_factorization.NMF object at 0x7f8520b195d0>]


Starting:  <surprise.prediction_algorithms.matrix_factorization.SVD object at 0x7f851ecc6ed0>
Done:  <surprise.prediction_algorithms.matrix_factorization.SVD object at 0x7f851ecc6ed0>


Starting:  <surprise.prediction_algorithms.matrix_factorization.NMF object at 0x7f8520b195d0>
Done:  <surprise.prediction_algorithms.matrix_factorization.NMF object at 0x7f8520b195d0>



	DONE
```

In [12]:
```python
rmse_list
```

Out[12]:
```
[array([0.94345155, 0.94563508, 0.94394977]),
 array([0.97427259, 0.97688421, 0.97477224])]
```

Plotting a table of the test rmse and its time of running against both the selected algorithms.

In [13]:
```python
surprise_results = pd.DataFrame(benchmark).set_index('Algorithm').sort_values('test_rmse')
surprise_results
```

Out[13]:

Algorithm	test_rmse	fit_time	test_time
SVD	0.944345	4.103123	0.379911
NMF	0.975310	4.988396	0.347002

In [14]:
```python
# smaller grid for lesser running time
param_grid = {
    "n_epochs": [10, 20, 40, 75],
    "lr_all": [0.002, 0.005],  #alpha--learning_rate
    "reg_all": [0.02]          #lambda
}
gs = GridSearchCV(SVD, param_grid, measures=["rmse", "mae"], refit=True, cv=3)

gs.fit(data)

training_parameters = gs.best_params["rmse"]

print("BEST RMSE: \t", gs.best_score["rmse"])
print("BEST MAE: \t", gs.best_score["mae"])
print("BEST params: \t", gs.best_params["rmse"])
```
```
BEST RMSE:      0.9432340469672061
BEST MAE:       0.7440505288284266
BEST params:    {'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0.02}
```

In [ ]:

In [15]:
```python
results_df = pd.DataFrame.from_dict(gs.cv_results)
results_df
```

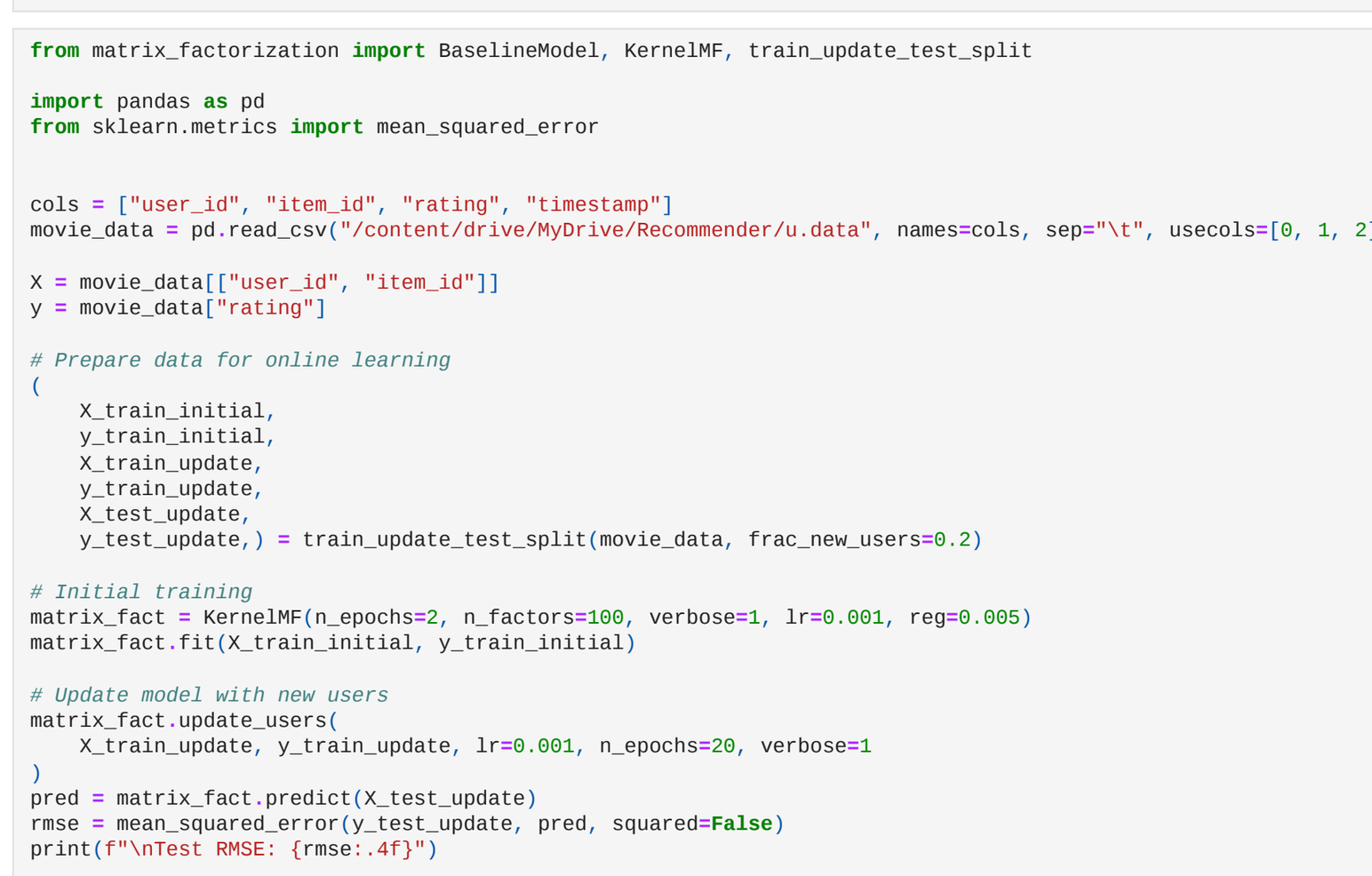
Out[15]:

	split0_test_rmse	split1_test_rmse	split2_test_rmse	mean_test_rmse	std_test_rmse	rank_test_rmse	split0_test_mae	split1_test_mae	split2_test_mae	mean_test_mae	std_test_mae	rank_test_mae	mean_fit_time	std_fit_time	mean_test_time	std_test_time	params	param_n_epochs
0	0.966533	0.971441	0.966463	0.968146	0.002330	7	0.768117	0.774073	0.767973	0.770054	0.002842	8	2.063159	0.042983	0.350125	0.097387	{'n_epochs': 10, 'lr_all': 0.002, 'reg_all': 0...	10
1	0.951074	0.956494	0.949415	0.952327	0.003023	4	0.752128	0.758245	0.750975	0.753783	0.003190	4	2.002056	0.014635	0.409148	0.093396	{'n_epochs': 10, 'lr_all': 0.005, 'reg_all': 0...	10
2	0.954160	0.959733	0.953373	0.955755	0.002831	5	0.755430	0.761845	0.755379	0.757551	0.003036	5	3.967993	0.009103	0.356832	0.128275	{'n_epochs': 20, 'lr_all': 0.002, 'reg_all': 0...	20
3	0.941657	0.946199	0.941846	0.943234	0.002098	1	0.742100	0.747307	0.742745	0.744051	0.002317	1	3.964984	0.015152	0.276456	0.008239	{'n_epochs': 20, 'lr_all': 0.005, 'reg_all': 0...	20
4	0.944309	0.949785	0.943442	0.945845	0.002808	2	0.745675	0.751149	0.744345	0.747057	0.002944	2	7.916635	0.027692	0.417499	0.082382	{'n_epochs': 40, 'lr_all': 0.002, 'reg_all': 0...	40
5	0.965181	0.968984	0.960363	0.964843	0.003528	6	0.759202	0.762270	0.753200	0.758224	0.003767	6	8.123780	0.303961	0.329781	0.092928	{'n_epochs': 40, 'lr_all': 0.005, 'reg_all': 0...	40
6	0.951410	0.956716	0.946420	0.951516	0.004204	3	0.747356	0.753371	0.744804	0.748510	0.003592	3	15.086792	0.231853	0.275119	0.000643	{'n_epochs': 75, 'lr_all': 0.002, 'reg_all': 0...	75
7	0.979605	0.986884	0.976519	0.981003	0.004346	8	0.768026	0.773235	0.766807	0.769356	0.002788	7	14.749054	0.117120	0.401874	0.098910	{'n_epochs': 75, 'lr_all': 0.005, 'reg_all': 0...	75

plotting a graph for rmse vs each of the three splits for test dataset only. We can see that even though the graph is slightly increasing; the increase in itself is rather small

In [17]:
```python
rmse_split_cols=[0.946623,0.942259,0.949082]
plt.xlabel("3 fold splits")
plt.ylabel("rmse per split")
plt.plot([1,2,3],rmse_split_cols)
```

Out[17]: `[<matplotlib.lines.Line2D at 0x7f851a884650>]`



Comparison of results from task 2 against task 3

The test rmse achieved using the hand made function of matrix factorisation gave a RMSE of 1.12 whereas using the surprise library we got better results of 0.9434. Hence, the library does implement the algorithm in a better fashion than the crude algorithm developed from scratch.

implementation of the same question as above but with a different library

In [ ]:
```python
pip install matrix_factorization
```

In [19]:
```python
from matrix_factorization import BaselineModel, KernelMF, train_update_test_split

import pandas as pd
from sklearn.metrics import mean_squared_error


cols = ["user_id", "item_id", "rating", "timestamp"]
movie_data = pd.read_csv("/content/drive/MyDrive/Recommender/u.data", names=cols, sep="\t", usecols=[0, 1, 2], engine="python")

X = movie_data[["user_id", "item_id"]]
y = movie_data["rating"]

# Prepare data for online learning
(
    X_train_initial,
    y_train_initial,
    X_train_update,
    y_train_update,
    X_test_update,
    y_test_update,) = train_update_test_split(movie_data, frac_new_users=0.2)

# Initial training
matrix_fact = KernelMF(n_epochs=2, n_factors=100, verbose=1, lr=0.001, reg=0.005)
matrix_fact.fit(X_train_initial, y_train_initial)

# Update model with new users
matrix_fact.update_users(
    X_train_update, y_train_update, lr=0.001, n_epochs=20, verbose=1
)
pred = matrix_fact.predict(X_test_update)
rmse = mean_squared_error(y_test_update, pred, squared=False)
print(f"\nTest RMSE: {rmse:.4f}")

# Get recommendations
user = 200
items_known = X_train_initial.query("user_id == @user")["item_id"]
matrix_fact.recommend(user=user, items_known=items_known)
```
```
Epoch  1 / 2  -  train_rmse: 1.072446827939706
Epoch  2 / 2  -  train_rmse: 1.038233436500082
Epoch  1 / 20  -  train_rmse: 1.0793743768018684
Epoch  2 / 20  -  train_rmse: 1.06775766358668
Epoch  3 / 20  -  train_rmse: 1.0584399074213695
Epoch  4 / 20  -  train_rmse: 1.0508079862307615
Epoch  5 / 20  -  train_rmse: 1.0444439275567512
Epoch  6 / 20  -  train_rmse: 1.0390420157363378
Epoch  7 / 20  -  train_rmse: 1.0343913974239423
Epoch  8 / 20  -  train_rmse: 1.030312068426762
Epoch  9 / 20  -  train_rmse: 1.026711453812834
Epoch  10 / 20  -  train_rmse: 1.023484240074798
Epoch  11 / 20  -  train_rmse: 1.0205624522412469
Epoch  12 / 20  -  train_rmse: 1.0178983991763628
Epoch  13 / 20  -  train_rmse: 1.015440139837755
Epoch  14 / 20  -  train_rmse: 1.0131650761413826
Epoch  15 / 20  -  train_rmse: 1.0110349517624047
Epoch  16 / 20  -  train_rmse: 1.0090326205335547
Epoch  17 / 20  -  train_rmse: 1.0071402058401868
Epoch  18 / 20  -  train_rmse: 1.0053438633703515
Epoch  19 / 20  -  train_rmse: 1.0036265623893441
Epoch  20 / 20  -  train_rmse: 1.001984527421633

Test RMSE: 1.0306
```

Out[19]:

	user_id	item_id	rating_pred
1116	200	1128	4.122772
1157	200	801	4.083705
60	200	134	4.077005
48	200	181	4.053374
115	200	127	4.030001
252	200	12	4.029805
87	200	168	4.026652
859	200	119	4.024970
26	200	275	4.009629
679	200	697	4.005560

References:- 1) https://towardsdatascience.com/a-complete-guide-to-recommender-system-tutorial-with-sklearn-surprise-keras-recommender-5e52e8ceace1

2) https://githubhelp.com/Quang-Vinh/matrix-factorization

3)http://www.quuxlabs.com/blog/2010/09/matrix-factorization-a-simple-tutorial-and-implementation-in-python/

4)https://towardsdatascience.com/comprehensive-data-explorations-with-matplotlib-a388be12a355

In [ ]: