Experiment No. 8
Implementation of a Regression based recommendation system
Date of Performance:
Date of Submission:
Marks:
Sign:

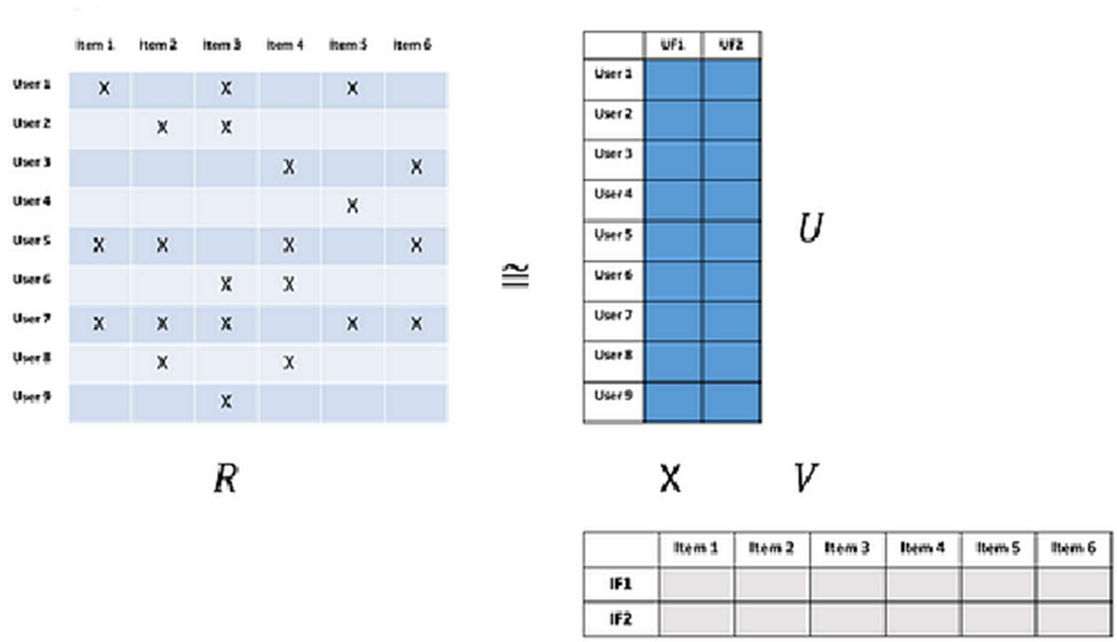
Aim: Implementation of a Regression based recommendation system.

Objective: Able to design a regression based recommendation system with respective implementation.

Theory: Often, matrix factorization is applied in the realm of dimensionality reduction, where we are trying to reduce the number of features while still keeping the relevant information. This is the case with principal component analysis (PCA) and the very similar singular value decomposition (SVD).

Essentially, can we take a large matrix of user/item interactions and figure out the latent (or hidden) features that relate them to each other in a much smaller matrix of user features and item features? That's exactly what ALS is trying to do through matrix factorization.

As the image below demonstrates, let's assume we have an original ratings matrix RR of size MxNMxN, where MM is the number of users and NN is the number of items. This matrix is quite sparse, since most users only interact with a few items each. We can factorize this matrix into two separate smaller matrices: one with dimensions MxKMxK which will be our latent user feature vectors for each user (U)(U) and a second with dimensions KxNKxN, which will have our latent item feature vectors for each item (V)(V). Multiplying these two feature matrices together approximates the original matrix, but now we have two matrices that are dense including a number of latent features KK for each of our items and users.



In order to solve for UU and VV, we could either utilize SVD (which would require inverting a potentially very large matrix and be computationally expensive) to solve the factorization more precisely or apply ALS to approximate it. In the case of ALS, we only need to solve one feature vector at a time, which means it can be run in parallel! (This large advantage is probably why it is the method of choice for Spark). To do this, we can randomly initialize UU and solve for VV. Then we can go back and solve for UU using our solution for VV.

Implementation:

```
import numpy as np
import pandas as pd
y = pd.read_excel('./ex8_movies.xlsx', sheet_name
= 'y', header=None)
y.head()
r = pd.read_excel('./ex8_movies.xlsx',
sheet_name='R', header=None)
r.head()
X = pd.read_excel('./movie_params.xlsx',
sheet_name='X', header=None)
X.head()
theta = pd.read_excel('./movie_params.xlsx',
sheet_name='theta', header=None)
theta.head()
for i in range(len(r.columns)):
    r[i] = r[i].replace({True: 1, False: 0})
r.head()
movies = open('./movie_ids.txt',
'r').read().split("\n")[:-1]
movies
def costfunction(X, y, r, theta, Lambda):
    predictions = np.dot(X, theta.T)
    err = predictions-y
    J = 1/2 * np.sum((err**2) * r)
    reg_x = Lambda/2 * np.sum(np.sum(theta**2))
    reg_theta = Lambda/2 * np.sum(np.sum(X**2))
```

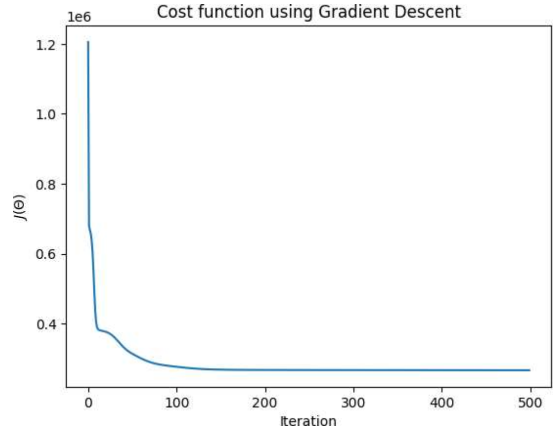
```
    grad = J + reg_x + reg_theta
    return J, grad
J, grad = costfunction(X, y, r, theta, 0)
m, n = y.shape[0], y.shape[1]
ymean = np.zeros((m, 1))
ymean
ynorm = np.zeros((m, n))
a = pd.DataFrame([[3,4,5], [7,3,5], [7,4,6], [1,2,3]])
np.sum(a, axis=1)
np.sum(y, axis=1)
ymean = np.sum(y, axis=1)/np.sum(r, axis=1)
ynorm = np.sum(y, axis=1)*np.sum(r, axis=1) -
ymean
def normalizeRatings(y, r):
    ymean = np.sum(y, axis=1)/np.sum(r, axis=1)
    ynorm = np.sum(y, axis=1)*np.sum(r, axis=1) -
ymean
    return ymean, ynorm
np.sum(y, axis=1)/np.sum(r, axis=1)
np.sum(y, axis=1)*np.sum(r, axis=1) - ymean
def gradientDescent(X, y, r, theta, Lambda,
num_iter, alpha):
    J_hist = []
    for i in range(num_iter):
        cost, grad = costfunction(X, y, r, theta,
Lambda)
        X = X - alpha*(np.dot(np.dot(X, theta.T) - y,
theta) + Lambda*X)
        theta = theta - alpha*(np.dot((np.dot(X,
theta.T) - y).T, X) + Lambda*theta)
        #print(cost)
        J_hist.append(cost)
    return X, theta, J_hist
x_up, theta_up, J_hist = gradientDescent(X, y, r,
theta, 0, 5, 0.03)
my_ratings = np.zeros((1682,1))

# Create own ratings
my_ratings[5] = 5
my_ratings[50] = 1
my_ratings[9] = 5
my_ratings[27]= 4
my_ratings[58] = 3
my_ratings[88]= 2
my_ratings[123]= 4
my_ratings[165] = 1
my_ratings[187]= 3
my_ratings[196] = 2
my_ratings[228]= 4
my_ratings[258] = 5
my_ratings[343] = 4
my_ratings[478] = 1
my_ratings[511]= 4
my_ratings[690] = 5
my_ratings[722]= 1
my_ratings[789]= 3
my_ratings[832] = 2
my_ratings[1029]= 4
my_ratings[1190] = 2
my_ratings[1245]= 5
my_r = np.zeros((1682,1))
for i in range(len(r)):
    if my_ratings[i] !=0:
        my_r[i] = 1
my_r
y1 = np.hstack((my_ratings, y))
r1 = np.hstack((my_r, r))
ymean, ynorm = normalizeRatings(y1, r1)
num_users = y1.shape[1]
num_movies = y1.shape[0]
num_features = 10

X1= np.random.randn(num_movies, num_features)
Theta1 = np.random.randn(num_users,
num_features)
Lambda=10

x_up, theta_up, J_hist = gradientDescent(X1, y1,
r1, Theta1, 10, 500,0.001)
import matplotlib.pyplot as plt
plt.figure()
plt.plot(J_hist)
plt.xlabel("Iteration")
plt.ylabel("$J(\Theta)$")
plt.title("Cost function using Gradient Descent")
p = np.dot(x_up, theta_up.T)
p[:, 0]+ ymean
my_predictions = p[:, 0] + ymean
my_predictions = pd.DataFrame(my_predictions)
df =
pd.DataFrame(np.hstack((my_predictions,np.array(
movies)[:,np.newaxis])))
df.head()
df.sort_values(by=[0],ascending=False,inplace=Tr
ue)
df.head(10)
```

Output:



	0	1
813	5.00899174417492	814 Great Day in Harlem, A (1994)
1499	5.008863681768815	1500 Santa with Muscles (1996)
1200	5.00750219922489	1201 Marlene Dietrich: Shadow and Light (1996)
1188	5.0063394140439215	1189 Prefontaine (1997)
1292	5.001841760627783	1293 Star Kid (1997)
1652	5.001830463490426	1653 Entertaining Angels: The Dorothy Day Stor...
1535	5.00046215214084	1536 Aiqing wansui (1994)
1598	5.000328136887362	1599 Someone Else's America (1995)
1121	4.999218218796543	1122 They Made Me a Criminal (1939)
1466	4.998626979267996	1467 Saint of Fort Washington, The (1993)

Conclusion:

Regression-based recommendation systems predict user-item ratings using regression models. By analyzing historical interactions and item features, these systems learn to estimate user preferences accurately. Despite being computationally intensive, regression models offer fine-grained predictions and handle sparse data well. Their ability to capture complex relationships between users and items makes them valuable for personalized recommendation tasks, enhancing user experience and satisfaction.