EE219 PROJECT 3
Collaborative Filtering

Guanchu Ling 904590047
Yingchao Tang 404592020
Yang Guo 304681050

Introduction:

In this project, collaborative filtering is used for predicting the user preferences of different movies. The original dataset contains user ratings of movies ranging from 0.5 to 5.0, with increment 0.5. It contains 100,004 ratings across 9125 movies. The ratings were created by 671 users between January 09, 1995 and October 16, 2016. Every user has rated at least 20 movies, but not necessarily all movies. By building a recommendation algorithm that predicts how users rate movies, it is possible to fill out the missing data not yet created by real users, and we can use these results to make recommendations to what the users may potentially like.

The collaborative filtering technique we use in this project is Alternating Least Squares. It performs a special factorization job to decompose the data matrix R into two matrices U and V. Then the product of the two matrices, UV, are calculated. The UV matrix is a good approximation to R at known data point, and the unknown data points are thus predicted.

Question 1

The basic idea of the predicting algorithm is to run the matrix factorization. But first we have to build the data matrix R. Generally, matrix R contains user ratings with user IDs on rows and movie IDs on columns. By doing some simple analysis on the original data in file “ratings.csv”, we noticed that the IDs of all 671 users are consecutive from 1 to 671. So, the user IDs can be directly used as the row indices of the R matrix. However, the IDs of 9125 movies are not necessarily consecutive, which poses the problem that we cannot use them as the column indices. The way we solve it is by

building an extra correspondence relationship that maps the movie IDs to their actual column indices in the R matrix. In other words, we use an integer array to store this correspondence: the indices of the array represent the IDs of the movies; the entries in the array represent the actual column indices in R matrix; if certain movie ID (i.e. certain index of the correspondence array) is not included in the dataset, the entry in the array is set to be -1, representing an invalid entry. Now we can use the movie IDs to quickly locate the column index in the R matrix; this operation is done in constant time.

Before building the R matrix, we iterate through the whole dataset to count the actual number of movies involved. Then we use this number and the number of users to define the dimension of R matrix. The entries in R matrix is initially set to be 0, then we iterate the original dataset again to fill the data points with known data. In the meantime, we build another 0-1 matrix called weight matrix. The weight matrix is used for indicating whether a data point is known or unknown, thus simplifying the implementation of the factorization step.

For the factorization job, we use the wnmfrule function from Matrix Factorization Toolbox in Matlab. Because in default, we do not need to pass the weight matrix into the function; it would generate the weight matrix from R matrix. But we have already built the weight matrix before calling the function, so a little modification must be made to its implementation to let it use the weight matrix we pass into. In order to avoid confusion, we changed the name of the function to be “factorize”.

After the factorization, the total least squared error could be calculated. The results with different k value (number of clusters) for this question are listed below:

```
Max number of iterations: 500
The total squared error for k=10 is: 32144.4086
The total squared error for k=50 is: 6403.456
The total squared error for k=100 is: 1370.905
```

```
Max number of iterations: 1000
The total squared error for k=10 is: 30860.6592
The total squared error for k=50 is: 5232.8248
The total squared error for k=100 is: 732.8457
```

```
Max number of iterations: 2000
The total squared error for k=10 is: 30248.6885
The total squared error for k=50 is: 4653.2603
The total squared error for k=100 is: 444.1897
```

As we can see from the above results, larger k value leads to better results, because the number of clusters has a close relation with the precision of the factorized matrix. With larger k values, more clusters are used in the algorithm, thus more information is extracted from R matrix.

The number of iterations is also an important parameter. As in the results, larger number of iterations leads to better results. This is because the fitting of the factorized matrices is improved if the number of iteration is increased. However, the runtime of the factorization process is approximately linear with respect to the number of iterations, so the value of this parameter should be selected depending on actual computing capability to keep runtime under acceptable limit. We can see that the benefit of increasing the k value is far better than that of increasing the number of iterations, so we prefer choosing a higher k value with less iterations, which can maintain high precision while make the algorithm rather fast.

Question 2

In this question, we are going to use 10-fold cross validation to test the algorithm. For each time, we randomly choose 10% data points as the testing data and the other 90% as the training data. After all predicted data points are calculated, it will be compared with the actual data and we can obtain the total absolute error.

The method of splitting the dataset into 10 parts is by assign an index to every known data point, then use this to create a random ordering of the index. By sequentially selecting 10% indices in the random ordering, we can randomly get 10% data points without overlap.

The results are shown below:

Error information (50 iterations & k = 10):	
Average error for each fold:	Average value :
Error for fold-1 : 0.8554132	0.863946
Error for fold-2 : 0.8636819	
Error for fold-3 : 0.8665677	Highest value :
Error for fold-4 : 0.8694112	0.875774
Error for fold-5 : 0.8757741	
Error for fold-6 : 0.8584238	Lowest value :
Error for fold-7 : 0.8663630	0.855413
Error for fold-8 : 0.8582418	
Error for fold-9 : 0.8599843	
Error for fold-10 : 0.8655998	

Error information (50 iterations & k = 50):	
Average error for each fold:	Average value :
Error for fold-1 : 0.8539255	0.857167
Error for fold-2 : 0.8597144	
Error for fold-3 : 0.8535093	Highest value :
Error for fold-4 : 0.8626315	0.867208
Error for fold-5 : 0.8578069	
Error for fold-6 : 0.8541038	Lowest value :
Error for fold-7 : 0.8581539	0.849584
Error for fold-8 : 0.8550341	
Error for fold-9 : 0.8495837	
Error for fold-10 : 0.8672079	

Error information (50 iterations & k = 100):	
Average error for each fold:	Average value :
Error for fold-1 : 0.8237915	0.833340
Error for fold-2 : 0.8291833	
Error for fold-3 : 0.8317440	Highest value :
Error for fold-4 : 0.8430870	0.843087
Error for fold-5 : 0.8370268	
Error for fold-6 : 0.8360567	Lowest value :
Error for fold-7 : 0.8323288	0.823791
Error for fold-8 : 0.8304181	
Error for fold-9 : 0.8287944	
Error for fold-10 : 0.8409721	

The above results are obtained by setting the number of iteration to be 50. As we can see, the errors of each fold are reasonably evenly distributed, which means that the algorithm provides very stable recommendation at this number of iteration. As the k value increases, the average error decreases at a rather slow rate because the fitting of the model is slightly improved.

However, if the number of iteration is set too high, the model will have serious overfitting, leading to bad results like below (e.g. for 500 iterations):

Error information (500 iterations & k = 10):	
Average error for each fold:	Average value :
Error for fold-1 : 3176.1110	332.889786
Error for fold-2 : 0.9887022	
Error for fold-3 : 0.9862393	Highest value :
Error for fold-4 : 144.3456	3176.111029
Error for fold-5 : 1.004665	
Error for fold-6 : 1.059884	Lowest value :
Error for fold-7 : 1.192874	0.986239
Error for fold-8 : 0.9867793	

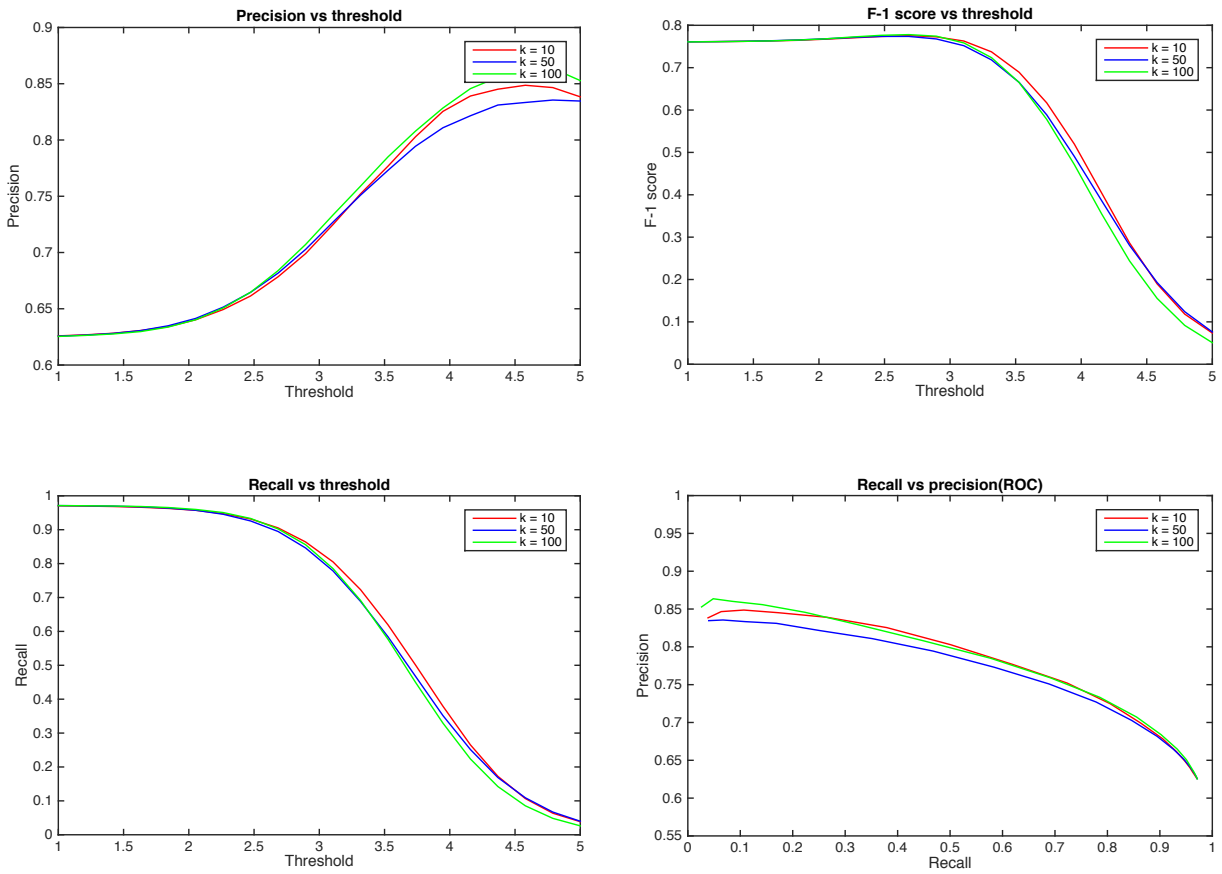
```
Error for fold-9 : 0.9880146
Error for fold-10 : 1.234112
```

Now the average error for each fold becomes very unstable; most error is contributed by fold-1 and fold-4. Therefore, to maintain the precision of the algorithm, we cannot use too many iterations for the factorization process.

Question 3

Before we move further down to finally implementing the recommendation algorithm for the users, let's see how we could use the rating data to classify the preference of users. Based on the actual rating data, we assume that the user likes the movie if the rating he/she gave for the movie is higher than 3, else the user just dislike the movie. And in question 2, we already got the predicted rating matrix based on the 10-fold cross validation. And now in the weighted predication matrix, each entry represents the rating of a movie from a certain user based on our collaborative filtering method. And now we want to set a preference threshold for the predicted rating data, classify the user preference base on this and compare with 3-partitioned actual rating data to see how the preference classification works as for different threshold values. And in question 2 we do the prediction based on three different number of clusters, so in this part we also separate the evaluation procedure for three different number of clusters.

Here are the results:



Precision, Recall, F-1 verse threshold value and ROC curve

In the first two plot we can see that, as expected, when the rating threshold is high and near 5, we got large precision but small recall. When the threshold is around 5, the movies recommended by the system or the system think that the user would like must also have higher actual rating if our algorithm works normally, so the precision would be good. But it's not necessarily a good sign. As we can see the recall in this situation is suppressed compared with lower rating threshold, which is resulted from that the high threshold also filtered some movies that the user might like (or actually like based on the actual rating data), thus the recall would be small and vise versa for the low threshold case.

As for some practical application of the measuring or precision and recall, we would prefer a more general case to evaluate the performance of recommendation without putting more or less weight on each of them. So sometimes we would rather use the F-1 score to evaluate the performance (the higher F-1 score, the better). As for the third plot, we can see that that F-1 score reaches its peak value at around the threshold of 2.7.

So, we would pick 2.7 as the optimized threshold for the user preference classification.

Also, ROC curve can be intuitive for the recommendation accuracy. And we can get the similar result: the higher the precision, the lower the recall. As for the different number of clusters used, the impact on the precision and recall is not very large except when the threshold is higher than 3, we can see that 100-clusters factorization gives higher precision while 10-clusters factorization gives higher recall. The precision and recall for three different number of clusters around our optimized point (around 2.7) do not vary obviously.

In this part, we explored the recommendation precision and recall based on different threshold values. And we got an optimized threshold value of 2.7.

Question 4

Part (a)

As we know, in the statistic field, when we are trying to evaluate some product based on a pool of survey results, a more precise way is putting unbalanced weight on each of the survey result. For positive feedback, higher the rating, higher the weight, and same manner for the negative feedback, lower the rating, higher the weight as for punishment. In our case, we want to see if this mechanism will give us higher predication accuracy as well. What we did is using the rating as the weight matrix instead of simple 0-1 weight matrix. And we do the same factorization process after switching the rating and weight matrix to see the outcome.

Here are the results:

```
The total least squared error for k=10 is: 90.4792
The total least squared error for k=50 is: 149.1514
The total least squared error for k=100 is: 107.1042
```

We can see that compared with the mean squared error we got in question 1, the

predication accuracy is significantly improved. And the interesting part is that, after doing the unbalanced weight, the total least square error is reduced by 30 – 300 times as for different number of clusters, and the error suppression is much more obvious for the 10-clustered case than the 100-clustered case. Now 10-clustered unbalanced weighted factorization gives as a much more accurate rating prediction.

Part (b)

In the similar manner as what we do in linear regression, we can improve the regression accuracy by adding an extra regularization term with a tunable constant to the penalty function. Here in our matrix factorization case, we can do the same thing to avoid singular solution. In our case, we optimized the factorization model for different k parameter (number of clusters) as well as the regularization constant lambda.

Here are the results:

```
      Error information (k = 10, lambda = 0.01):
Average error for each fold:
0.864300, 0.864967, 0.859825, 0.855778, 0.850094,
0.858656, 0.855245, 0.870970, 0.862336, 0.871776

Average value of average absolute error : 0.861394
Highest value of average absolute error : 0.871776
Lowest value of average absolute error : 0.850094
```

```
      Error information (k = 10, lambda = 0.10):
Average error for each fold:
0.872204, 0.867475, 0.867082, 0.861324, 0.857418,
0.855387, 0.860522, 0.867395, 0.862492, 0.869129

Average value of average absolute error : 0.864043
Highest value of average absolute error : 0.872204
Lowest value of average absolute error : 0.855387
```

```
      Error information (k = 10, lambda = 1.00):
Average error for each fold:
0.871601, 0.871308, 0.870664, 0.863050, 0.859038,
0.863015, 0.859085, 0.882976, 0.862905, 0.868775

Average value of average absolute error : 0.867242
Highest value of average absolute error : 0.882976
Lowest value of average absolute error : 0.859038
```

```
      Error information (k = 50, lambda = 0.01):
Average error for each fold:
0.867034, 0.859142, 0.863589, 0.870886, 0.863341,
0.854443, 0.867816, 0.876931, 0.874466, 0.873675

Average value of average absolute error : 0.867132
Highest value of average absolute error : 0.876931
Lowest value of average absolute error : 0.854443
```

```
      Error information (k = 50, lambda = 0.10):
Average error for each fold:
0.881868, 0.879968, 0.887318, 0.870264, 0.874968,
0.863950, 0.873655, 0.883825, 0.883789, 0.885401

Average value of average absolute error : 0.878501
Highest value of average absolute error : 0.887318
Lowest value of average absolute error : 0.863950
```

```
      Error information (k = 50, lambda = 1.00):
Average error for each fold:
0.926370, 0.927919, 0.925603, 0.918606, 0.918853,
0.908012, 0.921508, 0.937606, 0.932163, 0.940772
```

```
Average value of average absolute error : 0.925741
Highest value of average absolute error : 0.940772
Lowest value of average absolute error : 0.908012
```

```
      Error information (k = 100, lambda = 0.01):
Average error for each fold:
0.858120, 0.852327, 0.855543, 0.853245, 0.845503,
0.851750, 0.858062, 0.868700, 0.856372, 0.868197

Average value of average absolute error : 0.856782
Highest value of average absolute error : 0.868700
Lowest value of average absolute error : 0.845503
```

```
      Error information (k = 100, lambda = 0.10):
Average error for each fold:
0.878066, 0.882632, 0.880752, 0.880112, 0.876278,
0.875009, 0.870929, 0.897429, 0.889617, 0.894802

Average value of average absolute error : 0.882563
Highest value of average absolute error : 0.897429
Lowest value of average absolute error : 0.870929
```

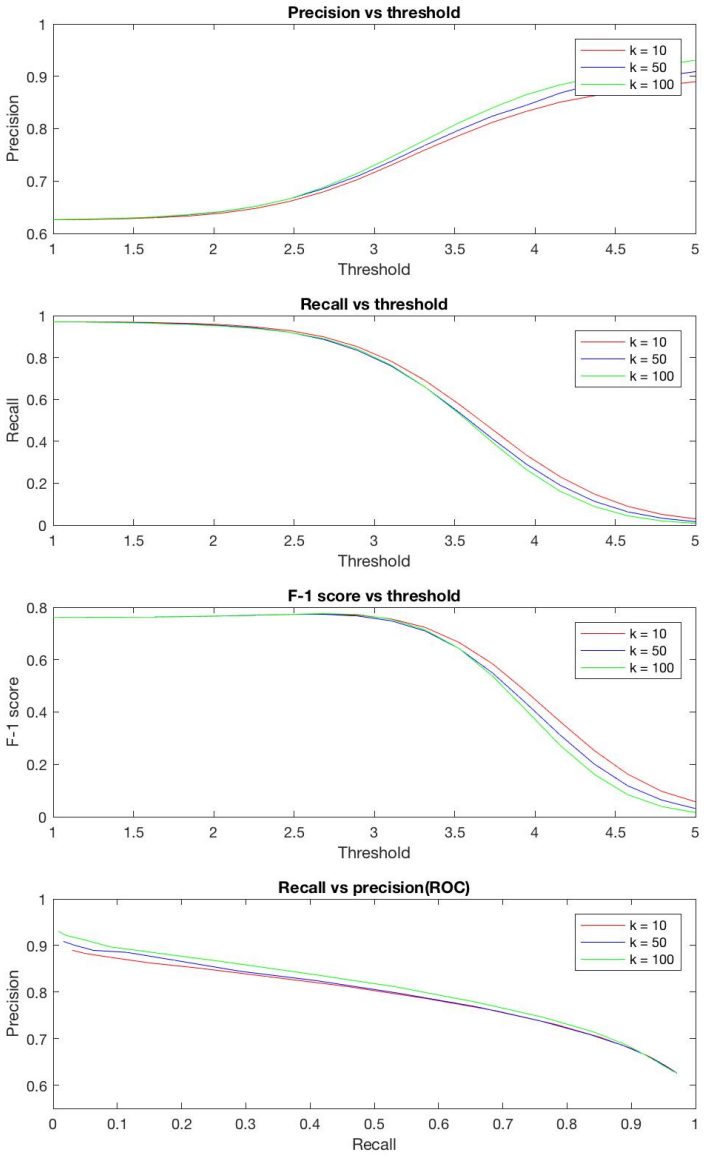
```
      Error information (k = 100, lambda = 1.00):
Average error for each fold:
0.938600, 0.944955, 0.947457, 0.935303, 0.937769,
0.934471, 0.943576, 0.951768, 0.947955, 0.957892

Average value of average absolute error : 0.943975
Highest value of average absolute error : 0.957892
Lowest value of average absolute error : 0.934471
```

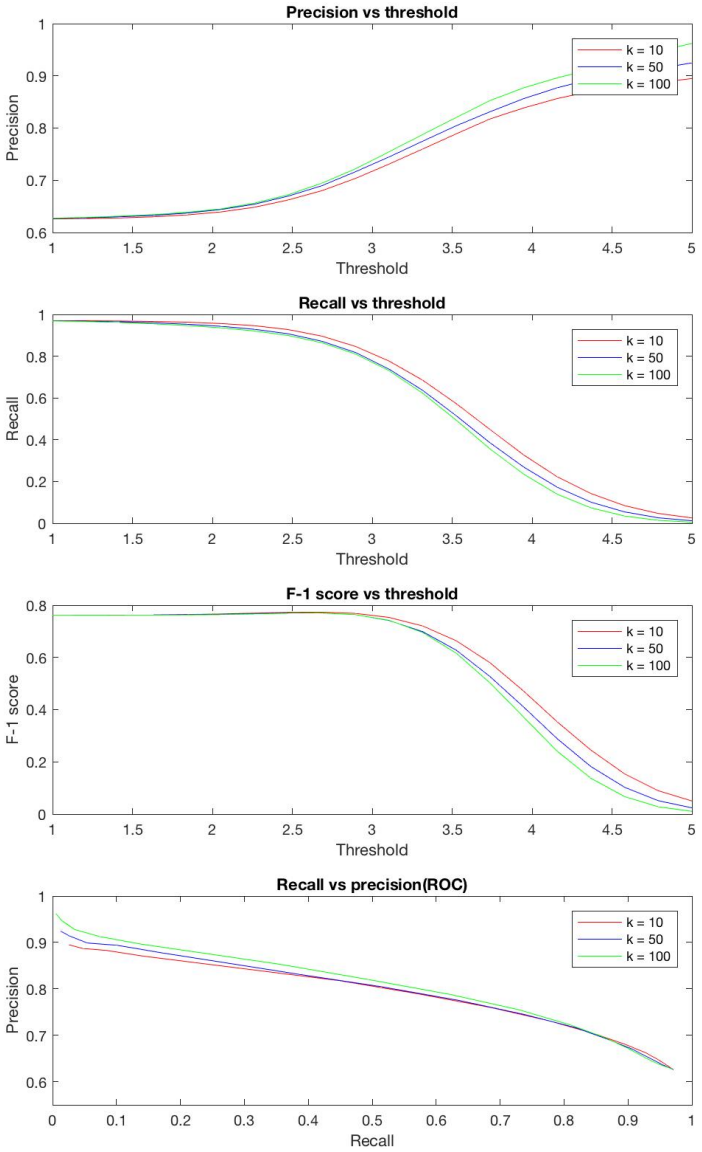
From the numerical results above we could see that the absolute error compared with

the non-regularized factorization is slight suppressed, especially for the 50-clustered one. And the 100-clustered factorization with regularization term of 0.01 gives the lowest average absolute error of 0.857.

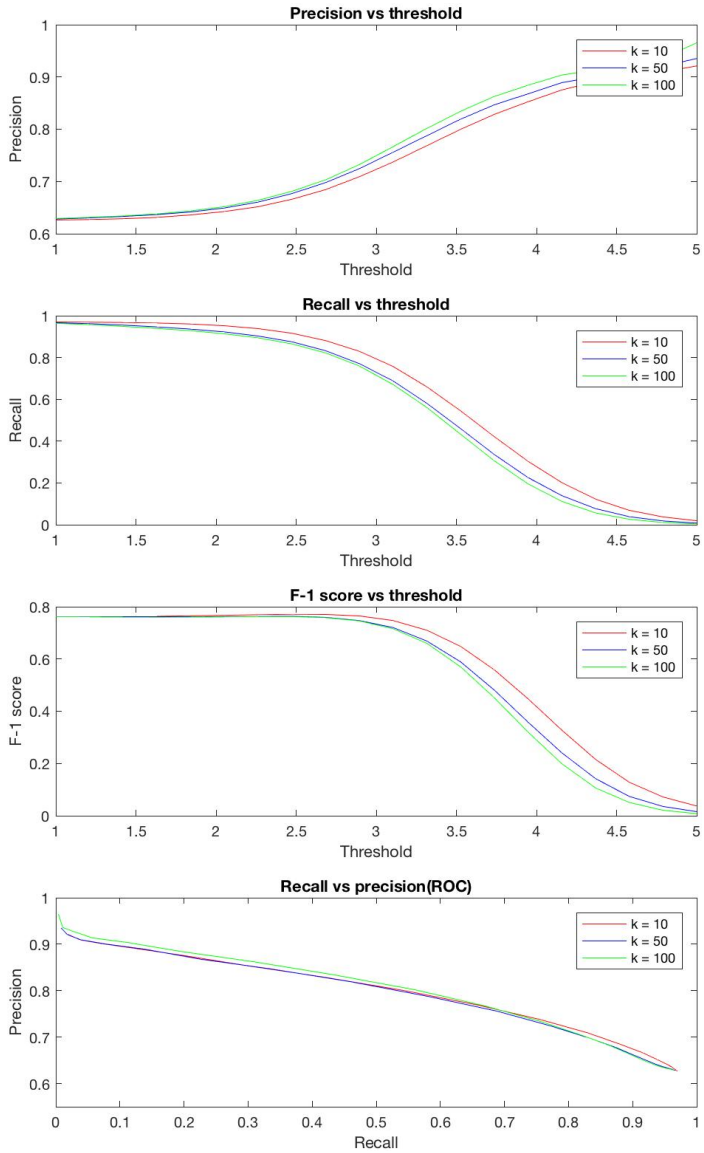
And also:



Precision & Recall over threshold and ROC (lamda = 0.01)



Precision & Recall over threshold and ROC (lamda = 0.1)



Precision & Recall over threshold and ROC (lamda = 1.0)

From the graphical results above we could see that, as for the preference classification with the regularized factorization, precision is improved with higher regularization constant, while recall is higher with smaller regularization constant. And for our previous optimized threshold around 2.7, the peak f1-score is slightly reduced after regularization.

Question 5

For this part, first we calculate predicted R matrix by building R as a 0-1 matrix where 1 is when a rating is available and 0 otherwise. For the weight matrix, we use the actual ratings as the weights to fill it. After calculating the product of U and V matrix, the weight matrix is again applied to eliminate those data points unknown. Then we can sort the ratings for every user into descending order to get the top L movies for every user.

The next step is to execute the above for every fold and find the precision. Precision is defined as:

$$Precision = \frac{\#\ of\ true\ positive}{\#\ of\ true\ positive + \#\ of\ false\ positive}$$

First, we try to recommend 5 movies for every user. While calculating the precision we ignore the movies whose ratings are not present in the actual R matrix. For different values of k, the precision results are:

```
Precision for k=10, L=5 : 0.9976
Precision for k=50, L=5 : 0.9988
Precision for k=100, L=5 : 0.9988
```

From the results, we can see that even for very small k values, the precision is reasonably high. This is because the number of movies recommended to each user is rather small (L = 5). Since we just simply pick the top 5 rated movies for the user while each user has rates at least 20 movies, it has big chance that the recommended movie is liked by the user.

Next is to calculate the Hit rate and False alarm rate for the entire algorithm. Hit rate is the fraction of the recommended test movies actually liked by the user; it is calculated as:

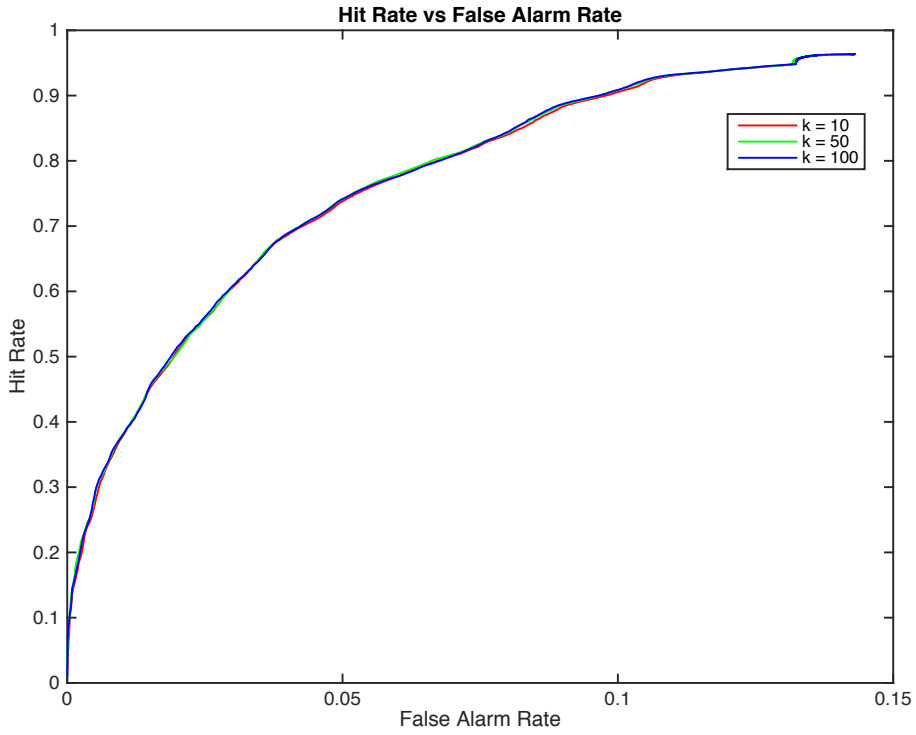
$$Hit\ Rate = \frac{\#\ of\ true\ positive}{\#\ of\ true\ positive + \#\ of\ false\ negative}$$

False alarm rate is the fraction of the recommended test movies not actually liked by the user; it is calculated as:

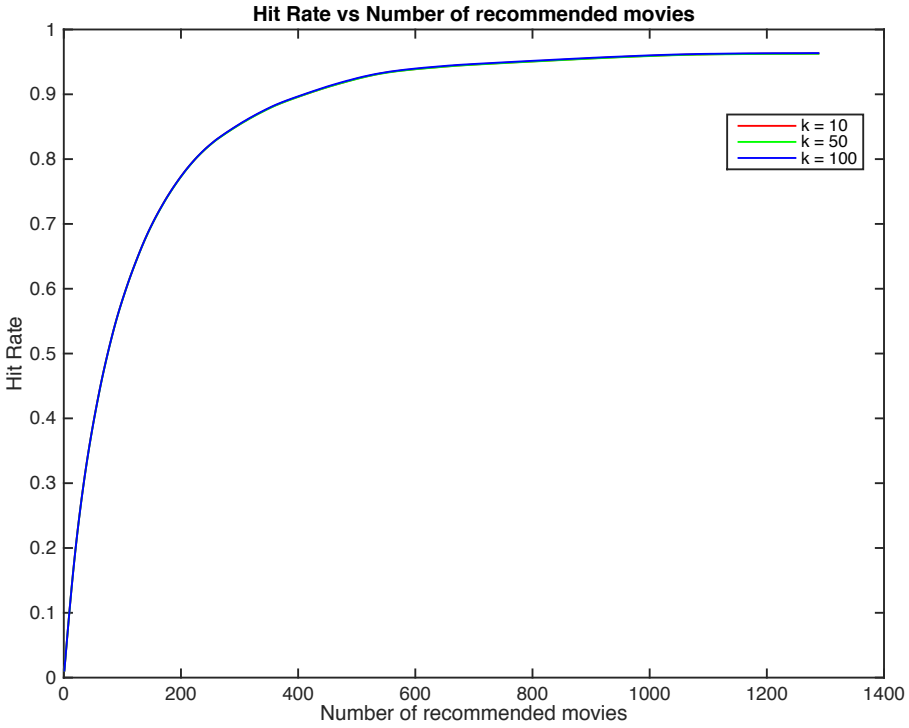
$$False\ Alarm\ Rate = \frac{\#\ of\ false\ positive}{\#\ of\ false\ positive + \#\ of\ true\ negative}$$

We calculate different values of Hit Rate and False Alarm Rate by increasing the L from 1 onwards. While calculating for different values of L, we reuse the same recommendation matrix for to avoid refactorization and increase speed.

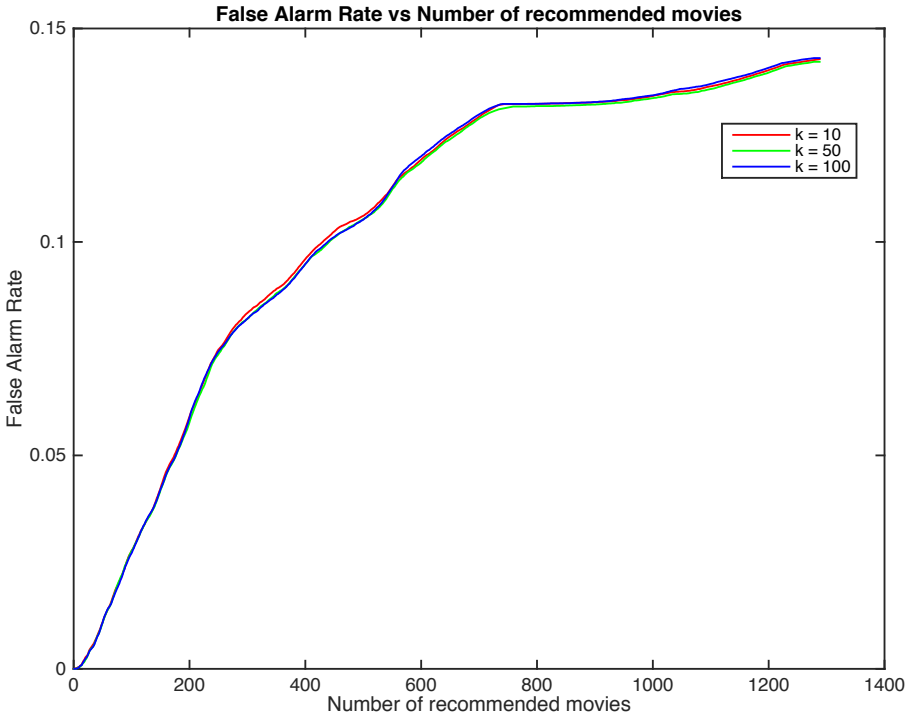
Once we have the recommendation matrix, we can count the numbers of True Positives and False Positives, then calculate the hit rate and false alarm rate. The results are shown below:



Hit Rate vs False Alarm Rate



Hit Rate vs Number of Recommended Movies



False Alarm Rate vs Number of Recommended Movies

We can see from the plots that hit rate and false alarm rate have a positive correlation. For different k values, the plots are largely overlapped, which means that k value does

not affect the result of the algorithm too much.

As we increase the value of L (maximum number of movies recommended to users), more movies actually liked by the user will be recommended to the user, causing the hit rate to go up. But in the meantime, the chance of recommending movies not liked by the user will also go up, leading to the increase of false alarm rate. Luckily, the false alarm rate remains at a very low level (lower than 0.15), compared to the hit rate (as high as 0.96). We can conclude that this recommendation system is properly functioning.

Reference:

1. The Non-Negative Matrix Factorization Toolbox in MATLAB
   https://sites.google.com/site/nmftool/
2. Yifeng, Li, Alioune, Ngom. The non-negative matrix factorization toolbox for biological data mining[R], 2013.