Predicting User Ratings for Music

Juan Llanes
Department of Computer Science
Georgia Institute of Technology
Atlanta, GA 30318

Myoungho Park
School of Industrial and Systems Engineering
Georgia Institute of Technology
Atlanta, GA 30318

Sourabh Bajaj
Department of Computer Science
Georgia Institute of Technology
Atlanta, GA 30318

Abstract

Recommendation systems have become one of the prominent area in machine learning with the growth in the internet based content providers such as Pandora and Netflix. Recommending systems have become really important in order to provide custom experience to the users. Rating other items based on ratings of one users and ratings by other users is of critical importance in these systems. In this paper we hope to predict user ratings accurately based on a hybrid approach that uses collaborative filtering at the core, and then exploits the hierarchy in music tags to accurately predict ratings of various items using linear regression between several predicted ratings. This combination provides accurate results of ratings of music items that haven't been rated by the user before. And root mean square error is found to be better then content based matching approaches.

1 Problem Statement

In this paper, the problem of learning to predict user ratings of musical items will be tackled. This problem was first proposed in the 2011 KDD Cup, a competition sponsored by Yahoo!. Predicting user ratings of new musical items based on previous rating history of the user itself, as well as from ratings derived from other users, is of critical importance in recommendation based systems that present a user with new content that best matches his/her taste. For example, Pandora and Netflix are such systems whose value is derived from showing the user new items that match the user preferences for certain media content. Of course, satisfying users by presenting them with media they like encourages them to return to the site, which implies more business and thus more money for the company presenting the content. A win-win situation.

2 Datasets

2.1 Main Yahoo! Dataset

The entire dataset is comprised of 300 million ratings of 600K musical items. Musical items can be tracks, artists, albums, and genre. These musical items are arranged in a hierarchy (i.e. a track belongs to an artist, which belongs to an album, which belongs to a genre. shown in Fig. 1), and a user may rate any musical item. The fact that there exist a hierarchy in the data provides more information for predicting a user rating for a new item.

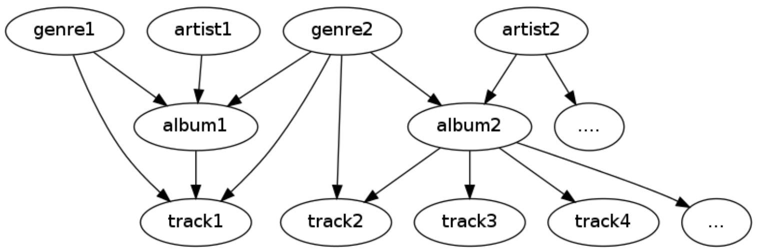


Figure 1: Heirarchy present in the dataset.

2.2 Datasets used for experiments

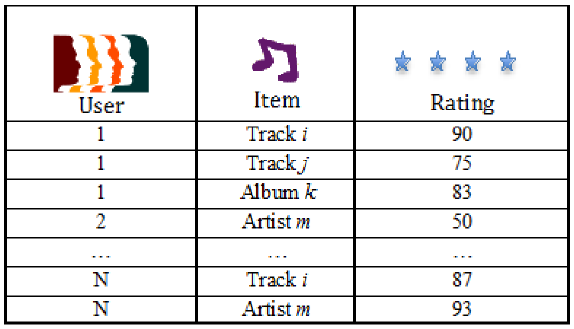
The dataset comes in form of training, validation and testing sets as it was part of the competition and these were provided in the contest guidelines itself. We try and maintain the same consistency so that it is feasible to compare with the results of other teams that participated in the competition. The ratings are in denominations of 10 from 0 to 100.

- **Toy data set :** In order to improve our development process, we write our algorithms and test them (while we are writing the code) against examples worked out by hand to ensure correctness. This is just a sample example we created on the fly, it is not related to the dataset.
- **Sample Dataset :** Once our algorithm is proven correct using Toy Dataset, we proceed to evaluate its effectiveness on this larger dataset. This was a sample provided in the contest itself for testing of algorithms.
- **Complete Dataset :** Our final learning will be done on this larger dataset using a more powerful computer.

Dataset	Users	Music Items	Ratings	Sparsity
Toy Data Set	4	5	15	75.0 %
Sample Dataset	44	9,300	11,000	2.69%
Complete Dataset	1,000,990	600,000	300,000,000	0.04%

2.3 Issue with the Dataset

The training file of the main dataset is of size 6.5 GB. For this reason, we cannot work on this large dataset in our personal laptops. Therefore, we resorted to a GaTech computer cluster called Gekko, which has 132 GB of memory, to work on this larger dataset. Note that we had to approximate the Complete data set using a random filtering approach (explained later).



User	Item	Rating
1	Track i	90
1	Track j	75
1	Album k	83
2	Artist m	50
...	...	...
N	Track i	87
N	Artist m	93

Figure 2: Input data format to the SVD algorithm

3 Implementation

We implemented two algorithms to solve this problem. The first one is a combination of a content based and a collaborative filtering approach using batch gradient descent. The second algorithm we implemented is a latent factor model SVD that as the previous algorithm, it finds a measure of similarity among users and between musical items.

3.1 Gradient Descent

As a first attempt, we implemented the content-based and collaborative filtering approach using gradient descent discussed in [4]. The intuition behind this method is explained below, and the math derivations are explained after that. If we had a set of features $F = [f_1, f_2, f_3, \cdots, f_n]$ for each musical item, which each could be interpreted as how much guitar (f_1), piano (f_2), rock(f_3), etc. a musical item has, this information would be useful if used in combination with a set of weights $\theta = [\theta_1, \theta_2, \theta_3, \cdots, \theta_n]$, where each user has a θ vector representing how much they like each of the features presented above (i.e. user1 may like rock and guitar a lot, but he may like piano very little). Thus, for each user, we have a θ vector that we can multiply with a feature vector corresponding to a particular song to yield a rating the user will give to that particular song:

$$Rating = [f_1, f_2, f_3, \cdots, f_n] * [\theta_1, \theta_2, \theta_3, \cdots, \theta_n]^T$$

The issue is we are not given any features for musical items nor any θ weights for the users, thus we find them by first initializing them to some small values, and then estimating them using gradient descent. Gradient descent will find a set of features derived from the actual content of the musical items, as well as a set of weights for each user defining how much they value each of the features.

3.1.1 Mathematical Formulation

1. Content-based filtering method
   (a) Assume we are given a set of features for each musical item $X_1, \cdots, X_n$.
   (b) Optimize/obtain the weights θ for each user minimizing the RMSE.

   Given $x^{(1)}, \cdots, x^{(n_m)}$, estimate $\theta^{(1)}, \cdots, \theta^{(n_u)}$:

$$\underset{\theta^{(1)},\cdots,\theta^{(n_u)}}{min} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} \left(\theta_k{}^{(j)}\right)^2$$

2. Collaborative-filtering method
   (a) Assume we are given a set of user weights for each user $\theta_1, \cdots, \theta_n$.
   (b) Optimize/obtain the weights $X_i, \cdots, X_n$ for each user minimizing the RMSE.

   Given $\theta^{(1)}, \cdots, \theta^{(n_u)}$, estimate $x^{(1)}, \cdots, x^{(n_m)}$:

$$\underset{x^{(1)},\cdots,x^{(n_m)}}{min} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_m} \sum_{k=1}^{n} \left(x_k{}^{(j)}\right)^2$$

3. Notice that we are not given either one of the quantities assumed to have in order to solve this problem, thus we randomly initialize this data in our algorithm. As an optimization to avoid optimizing back and forth, the following approach was followed. Optimize both the user weights θ and the Features at the same time. Minimize $x^{(1)}, \cdots, x^{(n_u)}$, and $\theta^{(1)}, \cdots, \theta^{(n_m)}$ simulatiously :

$$J_{(\theta^{(1)},\cdots,\theta^{(n_m)},x^{(1)},\cdots,x^{(n_u)})} = \frac{1}{2} \sum_{(i,j):r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} \left(x_k{}^{(j)}\right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} \left(\theta_k{}^{(j)}\right)^2$$

$$\underset{\substack{\theta^{(1)},\cdots,\theta^{(n_u)} \\ x^{(1)},\cdots,x^{(n_m)}}}{min} J_{(\theta^{(1)},\cdots,\theta^{(n_m)},x^{(1)},\cdots,x^{(n_u)})}$$

4. The final update rule for each feature and user weight θ is found by finding the derivative of the above formulas with respect to the features X_k and user vector $\theta^{(j)}$.

   (a) Initiallize $x^{(1)}, \cdots, x^{(n_m)}$ and $\theta^{(1)}, \cdots, \theta^{(n_u)}$ to small values.

   (b) Minimize $J_{(\theta^{(1)}, \cdots, \theta^{(n_m)}, x^{(1)}, \cdots, x^{(n_u)})}$ using gradient desent.

$$x_k{}^i = x_k{}^i - \alpha \left(\sum_{(i,j):r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) \theta_k{}^{(j)} + \lambda x_k{}^{(i)} \right)$$

$$\theta_k{}^i = \theta_k{}^i - \alpha \left(\sum_{(i,j):r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k{}^{(j)} + \lambda \theta_k{}^{(i)} \right)$$

3.1.2 Drawbacks

The issue we encountered using this batch gradient descent update rules is that they are extremely slow for our dataset. The reason for the failure is that gradient decent has a complexity of $O(N^5)$ for solving this problem using the math above, which is infeasible to achieve given our current computing capabilities. For instance, on the "Sample Dataset," which is comprised of 44 users with 9300 unique items, and approximately 11,000 ratings, it took 17 hours for this batch update rule to run one iteration of the gradient descent algorithm when using the learning rate of 0.1 (see Appendix A for code implementing this update rule). On the "toy data set" we needed approximately 15 iterations to obtain rates that were close to be correct (see results section).
For this reason, we searched for a faster method discussed below to approximate the user ratings. The gradient descent provides, however, a basic and intuitive understanding of how users can "collaborate" to yield a rating for a user that has never seen a musical item before; an idea that is further exploited in the next algorithm.

3.2 SVD

The second algorithm we implemented is a Singular Value Decomposition based collaborative filtering algorithm. We use SVD based Collaborative Filtering Algorithm in order to capture latent relationships between users and items that allow us to compute the predicted likeliness of a certain item by a user. We noticed on a similar contest sponsored by Netflix that the SVD based collaborative filtering algorithm was relatively successful; therefore, we decided to try it on this problem.
SVD is used for finding the Eigen vectors of the user-item matrix and then selecting the top K Eigen vectors. The number of eigenvectors K is found by iterating and using the value that gives the best error on the validation set. The intuition here is that retaining the k eigenvectors associated with the largest variance in the data will express the most interesting dynamics present in the data. The reason this method helps in predicting the ratings is that in the 600,000 dimensional item space or the 1,000,990 dimensional user space, the data is very sparse, and so it is difficult to extract useful information. Using the first K Eigen vectors, we can map this data into a K dimensional space (observed to be around 150 in our case). The data is easier to analyze in this lower dimensional space since we can cluster similar users and items together much more easily in this Eigen space. Now that we have gained some intuition, we give the steps for implementing the algorithm.

1. Data Representation Define the original user-item matrix, R, of size mxn, which includes the ratings of m users on n items. $r_{i,j}$ refers to the rating of user $user_i$ on item $item_j$. Preprocess user-item matrix R in order to impute the missing data. The preprocessing is described as follows:

   (a) Compute the average of each row, r_i, where $i = 1, 2, \ldots, m$, and the average of each column, c_j, where $j = 1, 2, \ldots, n$, from the user-item matrix, R.

   (b) Replace all missing values with the corresponding column average, c_j, which leads to a new filled-in matrix, R_f.

(c) Subtract the corresponding row average, r_i, from R_f, and obtain the row centered matrix A i.e the row mean of A is 0.

2. Low-rank approximation Computes the SVD of A and keeps only the first K diagonal entries from matrix S to obtain a $k \times k$ matrix, S_k. Similarly, matrices U_k and V_k of size $m \times k$ and $k \times n$ are generated. The reduced or reconstructed matrix is denoted as A_k.
3. Prediction generation : The predicted rating for user $user_i$ on item $item_j$ is given by:

$$pr_{i,j} = r_i + U_k \sqrt{S_k^T(i)} \sqrt{S_k} V_k^T(j)$$

The second part of the equation gives the corresponding element of the reduced matrix A_k. The prediction is generated by adding the mean of the appropriate row, r_i, to this element.

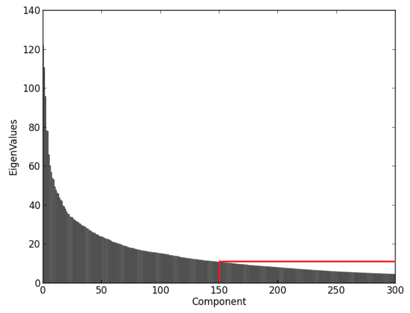


Figure 3: Eigenvalues, which were emperically found using the validation set. Line at K = 150

Figure 3 above shows the eigenvalues of the sparse matrix SVD decomposition. Specifically, this shows the variance present in the data along each of the eigenvector component. We only show the first 300 eigenvalues instead of all the possible 1,000,990 ones since they become rather small and unimportant for feature extraction after this point. We empirically selected the number K=150 eigenvectors using the validation set to find the number k that yielded the lowest error.

4 Method

4.1 Processing text files

The raw dataset is presented in text files, which are difficult to read directly into a matrix. Moreover, the latter text files amount to 6GB of data, and they include some information that is not used in our algorithm. To exclude the information we do not need, we parse the text files into a csv file that has the following columns: user id, item id, and the rating.
Using this method, the processed data can be compressed down to 4GB worth of data.

4.2 Sampling Users

Because we did not have enough computer processing and storage power needed to process the entire 300M ratings dataset, we filtered this data by randomly picking 5000 users from the 1M users. Using this 5000 users, we then apply the SVD algorithm to find the eigenvectors that represent the variance along all components in the data. We then iterate picking incremental number of eigenvectors k, and at each iteration, we find the error in the validation set using a value of k. We keep the k that minimizes the error. We run the above procedure ten times and kept the average k out of all the k that minimized the error. In our case K=150.

4.3 Creating the sparse matrix

Given that we have 1M users and around 600K items, creating a matrix with items as rows and users and columns is wasteful and impossible given our computer power capabilities. Thus, we used a sparse matrix representation of the data. To avoid wasting space on empty entries, the sparse matrix

has the following format: 3 columns [row, col, data]; row count is the number of actual ratings present in the original matrix. Furthermore, we always use unsigned values as the ratings are always positive.
The sparse matrix is created using python's numpy library. We then use the sparsesvd library in python to implement SVD and find the relevant eigenvalues and eigenvectors. The number of eigenvectors to be used are found by iterating over the possible eigenvectors and picking the number of them that give the best error on the validation set.
When finding the error during testing, we avoid multiplying the matrices obtained using SVD (U,S, and V) and instead, we only multiply relevant rows and columns for the rating that is being predicted at that instant in time. The latter allows for an efficient memory usage, although it is slightly more computationally intensive. This is a space-time tradeoff.

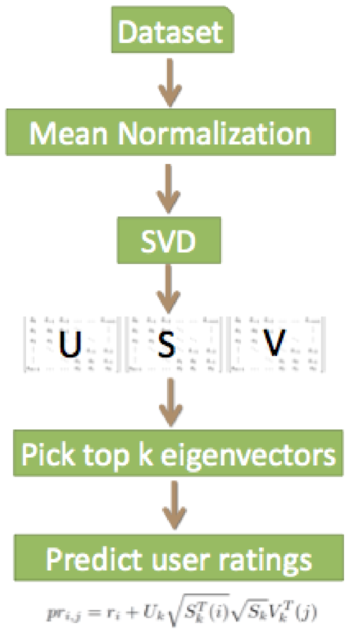


Figure 4: Flow diagram for the complete experiment

5 Results

The criteria used in the KDD Cup challenge for evaluating the algorithm was the Root Mean Square Error (RMSE).

$$Error = \sqrt{\frac{\sum_{i=1}^{n}(PredictedRating - ActualRating)^2}{N}}$$

This means that, in the test set, we will compare the predicted rating using the SVD algorithm versus what the actual value was. We compare our SVD algorithm against [1], who evaluated their algorithm on the same testing dataset we evaluated our results against.
We also compared our two implementations using the toy data set to ensure correctness, but the optimized gradient descent approach did not scale up to the complete data set.

	SVD	Gradient Descent	Bian[1]
Toy DataSet	1.23	1.12	-
Complete DataSet	25.96	No Result	31.00

We observe the RMSE to be 25.96 in the collaborative filtering approach compared to the error of 31.00 observed using the similarity based technique implemented in [1]. The batch gradient descent algorithm never finished on the Sample data set, and thus we did not run the complete data set on it.

6 Discussion

As it can bee seen from results section, our SVD algorithm performed better than the algorithm implemented in [1]. A lower RMSE implies that SVD algorithm was able to make better predictions of unknown ratings once it was trained. We believe one reason for this is that the original matrix is too sparse with a density of 0.04%. What this means is that most of matrix if filled with empty entries that contain no ratings at all, and so trying to use similarity based methods like they used in [1] may give more incorrect ratings because their algorithm fails to find correlation among users. We empirically selected the number k = 150 of eigenvectors by using the validation set as discussed in the SVD algorithm section. Indeed, this selection of k corresponds with the intuition provided in Figure 3 that the components of most variance are the first 150 eigenvectors corresponding to the observed first 150 eigenvalues in the graph. The method we implemented first condenses all the information using the sparse matrix technique discussed in the "data processing" section. Second, the SVD method we implemented finds similarities among users (i.e. if $user_1$ and $user_2$ give similar ratings to several songs, then we look into $user_1$ rating of an unknown song for $user_2$ when predicting $user_2$ rating of this unknown song), and also takes into account the similarities between different items(similarity between two different tracks).

7 Future Improvement

We had planned to implement a weighting scheme that exploited the hierarchy of the data. Time constraints did not allow for implementing this technique. However, we did do the mathematical derivation and the intuition behind it, which can readily be added to our current svd approach.

7.1 Using hierarchy and multiple matrices

In order to use the information contained in the hierarchy of the data, we need to have 4 other matrices like the one above, but each with rows corresponding to items belonging to genre, albums, artists, or tracks respectively. Having this 5 matrices (the 4 specified here plus the general one containing all musical items), given a particular item (i.e. a track) and a user, we can get the ratings contained in each of the 5 matrices, which we then can combine using a weighting scheme to decide on the final rating. Once we have calculated the 5 matrices and estimated the weights, we can proceed to classify data outside the training set.
Classification example:
Given a track1 (T1) for which user1 (U1) has no ratings, we will predict what rating the user would give to this track by:

1. Obtain the rating (R1) from the general matrix.
2. Obtain which genre (G1), album (A1), and artist (AT1) this track belongs to using the hierarchy in the data.
3. Obtain ratings R2, R3, R4, R5 by plugging G1, A1, AT1, T1 values into the 4 other matrices.
4. Combine all the ratings using 6 weights and yield the final rating.
   $Rfinal = w0 + w1(R1) + w2(R2) + w3(R3) + w4(R4) + w5(R5)$.

7.2 Weighting scheme

1. Query the $user_i$ rating for $item_j$ from the full matrix and this is X_f
2. Use the hierarchical information from other four matrixes (Genre, Artists, Album, and Track). $X_{Genre}, X_{Artist}, X_{Album}, X_{Track}$ Ratings by $user_i$ from matrices of Genre, Artists, Album and Track, respectively, corresponding to the dependency on $item_j$.
3. Define h_j as a hierarchical distance from $Item_j$ to other Nodes on the dependency tree above.
4. $W_j = \frac{K((X_f - X_j)h_j)}{\sum 5_{i=1} K((X_f - X_i)h_i)}$. Gaussian Kernel $K(x) = \frac{1}{\sqrt{2pi}} e^{-\frac{x^2}{2}}$
5. Properties

- Using the Guassian Kernel, nodes with the heirarchy level being queried gets higher weight.
- The Guassuian Kernel makes the weights for the farther nodes small.

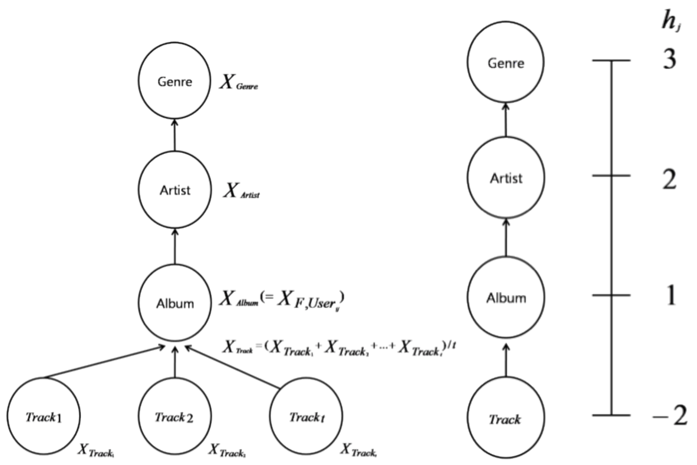


Figure 5: Weighting Scheme

8 Acknowledgments

We would like to thank Prof. Song for his guidance and Prof. Balch for allowing us to use the machine Gekko for our experiments. We also want to thank the T.A.s for their help with the project and the presentations.

References

[1] Joy Bian, Diana Palsetia and Kalika Vaidya, *Predicting User Ratings within Yahoo! Music Data*. Northwestern University, 2012.

[2] Diyi Yang, Tianqi CHen, Weinan Zhang and Yong Yu, *Collaborative Filtering with Short Term Preference Mining*. Proceedings of the 35th international ACM SIGIR conference, 2012.

[3] McFee, Brian, *The million song dataset challenge*. Proceedings of the 21st international conference companion on World Wide Web. ACM, 2012

[4] Ng, Andrew, *Machine Learning lecture notes and videos*. Coursera, Machine Learning, 2012

[5] Christopher M. Bishop *Pattern Recognition and Machine Learning*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. 2006.

[6] Vozalis M, Markos A, and Margaritis K.G. *Evaluation of standard SVD-based techniques for Collaborative Filtering*, In Proc. 9th Hellenic European Research on Computer Mathematics and its Applications Conference, 2009.

Appendix A Code for Optimized Gradient Descent

```
import numpy as np
import pandas
import Reading as rd

def collaborative_filtering_batch(Y, num_f, num_iterations):
   # Constants
   alpha = 0.01
   landa = 0.01

   # Extract data
   Nm = Y.shape[0]
   Nu = Y.shape[1]

   # 1- Init Xs and Os
   X = np.random.rand(Nm, num_f)
   O = np.random.rand(num_f, Nu)

   # 2- Minimize J
   x_k = 0 # row entry
   o_k = 0 # col entry

   for ite in range(0,num_iterations):
      print 'iteration: ' + str(ite)
      for k in range(0, num_f):
         print 'k: ' + str(k)

         x_k = k
         o_k = k

         for j in range(0,Nu):
            print "j: " + str(j)

            for i in range(0,Nm):
               #print "i: " + str(i)

               # 1- Calculate sum
               sum_total = 0
               for jj in range(0,Nu):
                  if (np.isnan(Y[i,jj])):
                     continue
                  sum_total += np.dot((np.dot(O[:,jj].T , X[i]) -
                      Y[i,jj]) , O[o_k,jj]) + landa*X[i,x_k]
               # Update X
               X[i,x_k] = X[i,x_k] - alpha * sum_total

               # 2- Calculate sum
               sum_total = 0
               for ii in range(0,Nm):
                  if (np.isnan(Y[ii,j])):
                     continue
                  sum_total += np.dot((np.dot(O[:,j].T, X[ii]) -
                      Y[ii,j]), X[ii,x_k]) + landa*O[o_k,j]

               # Update X
               O[o_k,j] = O[o_k,j] - alpha * sum_total

      na_mat = np.dot(X,O)
      np.savetxt('GD_output/'+str(ite)+'_X.txt', X)
      np.savetxt('GD_output/'+str(ite)+'_O.txt', O)
      np.savetxt('GD_output/'+str(ite)+'_na_mat.txt', na_mat)

   return X,O
```

Appendix B Code for SVD Collaborative Filtering

```
"""
Author : Sourabh Bajaj
Date : sourabhbajaj90@gmail.com
"""

#Python Imports
# import datetime as dt
# import copy
# import pickle
# import itertools

#3rd Party Imports
import numpy as np
import pandas
import scipy.sparse as sp
import scipy.stats.stats as st
import scipy
from sparsesvd import sparsesvd
import matplotlib.pyplot as plt
# Import other files


def SVD():
   na_ratemat = np.loadtxt('./Data/Filter1/user_item_max_rating222.csv',
       dtype='uint32', delimiter=',')
   users = list(set(na_ratemat[:, 0]))
   items = list(set(na_ratemat[:, 1]))
   print "Data Read"
   row_len = len(items)
   col_len = len(users)
   users = dict(zip(users, range(col_len)))
   items = dict(zip(items, range(row_len)))
   row = []
   col = []
   print "Rwo and Col"
   for i in range(len(na_ratemat[:, 2])):
      row.append(items[na_ratemat[i, 1]])
      col.append(users[na_ratemat[i, 0]])
   print row_len
   print col_len
   print "Creating the sparse matrix"
   na_ratemat = na_ratemat[:, 2]

   mean_ratemat = np.mean(na_ratemat)
   na_ratemat = na_ratemat - mean_ratemat
   std_ratemat = np.std(na_ratemat)
   na_ratemat = na_ratemat / std_ratemat

   na_ratemat = sp.csc_matrix((na_ratemat, (row, col)), shape=(row_len,
       col_len))
   U, s, V = sparsesvd(na_ratemat, 300)
   na_ratemat = np.dot(U.T, np.dot(np.diag(s), V))

   plt.clf()
   # plt.plot(s)
   plt.vlines(range(len(s)), np.zeros(len(s)), s)
   # plt.hist(s, bins=10)
   # plt.plot(Xtest[:, -1],'r')
   # plt.legend( ('Learner Predict', 'Actual Value') )
   plt.xlabel('Component')
   plt.ylabel('EigenValues')
   # plt.show()
   plt.savefig( 'EigenValues.png', format='png' )

   return na_ratemat, users, items, mean_ratemat, std_ratemat


def _validation():
   na_ratemat, users, items, mean_ratemat, std_ratemat = SVD()

   print "Validating the results"
   error = 0
   predicted_vals = []
   actual_vals = []
   count = 0
   s_input_file = './Data/Raw/validationIdx1.txt'
   ip_file = open(s_input_file, 'r')

   user = 0
   item = 0
   flag = False
   flag2 = False

   for line in ip_file.readlines():
      line = line.split('|')
      if len(line) == 2:
         u1 = int(line[0])
         try:
            user = users[u1]
            flag = True
         except:
            flag = False
            continue
      elif len(line) == 1 and flag:
         rating_line = line[0].split('\t')
         i1 = int(rating_line[0])
         rating = int(rating_line[1])
         try:
            item = items[i1]
            flag2 = True
         except:
            flag2 = False
         if flag2:
            predicted = (na_ratemat[item, user] * std_ratemat) +
                mean_ratemat
            if predicted > 100:
               predicted = 100
            if predicted < 0:
               predicted = 0
            # error = error + (predicted - rating) * (predicted - rating)
            predicted_vals.append(predicted)
            actual_vals.append(rating)
            # print predicted, rating
            count = count + 1

   print "Finished Creating User-Item-Rating CSV"
   ip_file.close()

   m1 = max(predicted_vals)
   m2 = min(predicted_vals)

   print m1, m2

   predicted_vals = np.array(predicted_vals)
   actual_vals = np.array(actual_vals)
   print predicted_vals[:20]
   print m1 - m2
```

```
    predicted_vals = ((predicted_vals - m2)*100) / (m1 - m2)
    print predicted_vals[:20]
    # print zip(predicted_vals, actual_vals)
    error = predicted_vals - rating
    error = sum(error * error)

    print "Least square error : ", error
    print "Count : ", count
    RMSE = np.sqrt(error / count)
    print "RMSE : ", RMSE


def _validation_original(df_rating, filename):

    print "Validating the results"
    error = 0
    predicted_vals = []
    actual_vals = []
    count = 0
    file = open(filename, 'r')

    for line in file.readlines():
        user = -1
        item = -1

        line = line.split('|')
        if len(line) == 2:
            user = line[0]
        elif len(line) == 1:
            rating_line = line[0].split('\t')
            item = rating_line[0]
            rating = int(rating_line[1])
            try:
                predicted = df_rating[item].ix[user]
                error = error + (predicted - rating)*(predicted - rating)
                predicted_vals.append(predicted)
                actual_vals.append(rating)
                count = count+1
            except:
                pass

    file.close()

    print "Least square error : ", error
    print "Count : ", count
    RMSE = np.sqrt(error/count)
    print "RMSE : ", RMSE

    print zip(predicted_vals, actual_vals)

    return error

def _original_method(k=1000):

    na_ratemat = np.loadtxt('./Data/user_item_max_rating222.csv',
        dtype='uint32', delimiter=',')
    print "Loaded the file"
    users = list(set(na_ratemat[:, 0]))
    items = list(set(na_ratemat[:, 1]))

    df_rate_mat = np.zeros((len(users), len(items)))
    df_rate_mat[:] = np.NAN

    df_rate_mat = pandas.DataFrame(df_rate_mat, index=users,
        columns=items)
    for i in range(len(na_ratemat[:, 2])):
        df_rate_mat[na_ratemat[i, 1]].ix[na_ratemat[i, 0]] = na_ratemat[i,
            2]
    # file = open('./Data/user_item_max_rating222.csv', 'r')
    # for line in file.readlines():
    #     line = line.split(',')

    #     df_rate_mat[line[1]].ix[line[0]] = int(line[2])

    # file.close()
    na_ratemat = 0
    print "Created the dataframe"

    na_mat = df_rate_mat.values

    column_mean = st.nanmean(na_mat, axis=0)
    temp = np.isnan(na_mat) * column_mean
    na_mat[np.isnan(na_mat)] = 0
    na_mat = na_mat + temp

    row_mean = st.nanmean(na_mat, axis=1)
    na_mat = (na_mat.T - row_mean).T
    print "Will take SVD now"
    U, s, V = np.linalg.svd(na_mat)
    print s.shape
    s = s[:k]
    U = U[:, :k]
    V = V[:k, :]
    na_mat = np.dot(np.dot(U, scipy.linalg.diagsvd(s, k, len(V))), V)

    na_mat = (na_mat.T + row_mean).T

    na_mat[na_mat < 0] = 50
    na_mat[na_mat > 100] = 50
    print "Done with the readings"
    df_rate_mat = pandas.DataFrame(na_mat, index=df_rate_mat.index,
        columns=df_rate_mat.columns)

    error = _validation_original(df_rate_mat, './Data/validationIdx1.txt')
    print "Done"

def main():
    _validation()
    # _original_method()


if __name__ == '__main__':
    main()
```

```
"""
Author : Sourabh Bajaj
Date : sourabhbajaj90@gmail.com
"""

#Python Imports
# import datetime as dt
# import copy
# import pickle
import csv
# import itertools

#3rd Party Imports
```

```
import numpy as np
# import pandas as pd


def _create_user_rating_csv(s_input_file, s_output_file):

   #Input and Output Files
   ip_file = open(s_input_file, 'r')
   op_file = csv.writer(open(s_output_file, 'wb'), delimiter=',')

   #Reading and outputing the data
   print "Begin Creating User-Rating CSV"

   for line in ip_file.readlines():
      line = line.split('|')
      if len(line) == 2:
         op_file.writerow([line[0], line[1][:-1]])

   #Done
   print "Finished Creating User-Rating CSV"
   ip_file.close()


def _user_item_rating_csv(s_input_file, s_output_file):

   #Input and Output Files
   ip_file = open(s_input_file, 'r')
   op_file = csv.writer(open(s_output_file, 'wb'), delimiter=',')
   user = 0

   #Reading and outputing the data
   print "Begin Creating User-Item-Rating CSV"

   for line in ip_file.readlines():
      line = line.split('|')
      if len(line) == 2:
         user = line[0]
      if len(line) == 1:
         rating_line = line[0].split('\t')
         op_file.writerow([user, rating_line[0], rating_line[1]])

   #Done
   print "Finished Creating User-Item-Rating CSV"
   ip_file.close()


def _artist_genre_csv(s_input_file, s_output_file):

   #Input and Output Files
   ip_file = open(s_input_file, 'r')
   op_file = csv.writer(open(s_output_file, 'wb'), delimiter=',')

   #Reading and outputing the data
   print "Begin Creating CSV from list"

   for line in ip_file.readlines():
      op_file.writerow([line[:-1]])

   #Done
   print "Finished Creating CSV from list"
   ip_file.close()


def _album_list_csv(s_input_file, s_output_file):
   #Input and Output Files
   ip_file = open(s_input_file, 'r')
   op_file = csv.writer(open(s_output_file, 'wb'), delimiter=',')

   #Reading and outputing the data
   print "Begin Creating CSV from list"

   for line in ip_file.readlines():
      line = line.split('|')
      op_file.writerow([line[0]])

   #Done
   print "Finished Creating CSV from list"
   ip_file.close()


def _read_list(s_input_file):
   l_list = np.loadtxt(s_input_file, dtype='uint32', delimiter=',')
   return l_list


def _user_csv(s_input_file, s_output_file):

   #Output File
   na_user = np.loadtxt(s_input_file, dtype='uint32', delimiter=',')
   op_file = csv.writer(open(s_output_file, 'wb'), delimiter=',')

   #Reading and outputing the data
   print "Begin Creating CSV from list"

   for user in na_user[:, 0]:
      op_file.writerow([user])

   #Done
   print "Finished Creating CSV from list"


def _max_users(s_input_file, s_output_file1, s_output_file2):
   na_user_item = np.loadtxt(s_input_file, dtype='uint32', delimiter=',')
   indices = na_user_item[:, 1].argsort()
   indices = indices[::-1]
   na_user_item = na_user_item[indices]

   na_user_item = na_user_item[:5000, :]

   op_file1 = csv.writer(open(s_output_file1, 'wb'), delimiter=',')
   op_file2 = csv.writer(open(s_output_file2, 'wb'), delimiter=',')

   #Reading and outputing the data
   print "Begin Creating CSV from list"

   for i in range(5000):
      op_file1.writerow([na_user_item[i, 0]])
      op_file2.writerow([na_user_item[i, 0], na_user_item[i, 1]])

   #Done
   print "Finished Creating CSV from list"


def _max_item_users(s_input_file, s_input_file2, s_output_file):
   #Input and Output Files
   ip_file = open(s_input_file, 'r')
   users = _read_list(s_input_file2)
   users.sort()
   op_file = csv.writer(open(s_output_file, 'wb'), delimiter=',')
   user = 0
   #Reading and outputting the data
   print "Begin Creating User-Item-Rating CSV"

   print type(users[0])
   flag = False
   for line in ip_file.readlines():
      line = line.split('|')
      if len(line) == 2:
         user = int(line[0])
         if user in users:
            flag = True
         else:
            flag = False
      if len(line) == 1 and flag:
         rating_line = line[0].split('\t')
         op_file.writerow([user, rating_line[0], rating_line[1]])

   #Done
   print "Finished Creating User-Item-Rating CSV"
   ip_file.close()


if __name__ == '__main__':
   path = './Data/'

   # s_input_file_UR = path + 'Raw/trainIdx1.txt'
   # # s_input_file_UR = path + 'Sample/trainIdx1.firstLines.txt'
   # s_output_file_UR = path + 'Processed/user_rating.csv'
   # _create_user_rating_csv(s_input_file_UR, s_output_file_UR)

   # s_input_file_Ar = path + 'Raw/artistData1.txt'
   # s_output_file_Ar = path + 'Processed/artist.csv'
   # _artist_genre_csv(s_input_file_Ar, s_output_file_Ar)

   # s_input_file_G = path + 'Raw/genreData1.txt'
   # s_output_file_G = path + 'Processed/genre.csv'
   # _artist_genre_csv(s_input_file_G, s_output_file_G)

   # s_input_file_Al = path + 'Raw/albumData1.txt'
   # s_output_file_Al = path + 'Processed/album.csv'
   # _album_list_csv(s_input_file_Al, s_output_file_Al)

   # s_input_file_T = path + 'Raw/trackData1.txt'
   # s_output_file_T = path + 'Processed/track.csv'
   # _album_list_csv(s_input_file_T, s_output_file_T)

   # s_input_file_UIR = path + 'Raw/trainIdx1.txt'
   # # s_input_file_UIR = path + 'Sample/trainIdx1.firstLines.txt'
   # s_output_file_UIR = path + 'Processed/user_item_rating.csv'
   # _user_item_rating_csv(s_input_file_UIR, s_output_file_UIR)

   s_input_file_U = path + 'Sample/trainIdx1.firstLines.txt'
   s_input_file_U_1 = path + 'MaxFilter/user.csv'
   s_output_file_U = path + 'MaxFilter/user_item_max_rating.csv'
   _max_item_users(s_input_file_U, s_input_file_U_1, s_output_file_U)
\end{appendices}
```