Your Name Xin Qian

Your Andrew ID xinq

# Homework 4

## 0. Statement of Assurance

You must certify that all of the material that you submit is original work that was done only by you. If your report does not have this statement, it will not be graded.

Yes, all of the material that I submitted is my original work.

## 1. Corpus Exploration (10)

Please perform your exploration on the training set.

### 1.1 Basic statistics (5)

| **Statistics** | |
|---|---|
| the total number of movies | 5392 (movies with a rating：5353) |
| the total number of users | 10916 (users with a rating：10858) |
| the number of times any movie was rated '1' | 53852 |
| the number of times any movie was rated '3' | 260055 |
| the number of times any movie was rated '5' | 139429 |
| the average movie rating across all users and movies | 3.38056747773 |

| For user ID **4321** | |
|---|---|
| the number of movies rated | 73 |
| the number of times the user gave a '1' rating | 4 |
| the number of times the user gave a '3' rating | 28 |
| the number of times the user gave a '5' rating | 8 |
| the average movie rating for this user | 3.15068493151 |

| For movie ID **3** | |
|---|---|
| the number of users rating this movie | 84 |
| the number of times the user gave a '1' rating | 10 |
| the number of times the user gave a '3' rating | 29 |
| the number of times the user gave a '5' rating | 1 |
| the average rating for this movie | 2.52380952381 |

### 1.2 Nearest Neighbors (5)

| | **Nearest Neighbors** |
|---|---|
| Top 5 NNs of user 4321 in terms of dot product similarity | 980, 155, 411, 169, 262 |
| Top 5 NNs of user 4321 in terms of cosine similarity | 8249, 7415, 9303, 8527, 7474 |
| Top 5 NNs of movie 3 in terms of dot product similarity | 1873, 4491, 5216, 3386, 1904 |
| Top 5 NNs of movie 3 in terms of cosine similarity | 3877,4557, 3804, 4680, 452 |

## 2. Rating Algorithms (50)

### 2.1 User-user similarity (10)

| **Rating Method** | **Similarity Metric** | **K** | **RMSE** | **Runtime(sec)** |
|---|---|---|---|---|
| Mean | Dot product | 10 | 1.00045809652 | 212.814899921 |
| Mean | Dot product | 100 | 1.0030868235 | 217.178015947 |
| Mean | Dot product | 500 | 1.03649673677 | 234.067832947 |
| Mean | Cosine | 10 | 1.04927073478 | 237.320639133 |
| Mean | Cosine | 100 | 1.04972857606 | 238.204774141 |
| Mean | Cosine | 500 | 1.06280019458 | 242.362897873 |
| Weighted | Cosine | 10 | 1.04902177879 | 243.237136126 |
| Weighted | Cosine | 100 | 1.0491750564 | 253.073585033 |
| Weighted | Cosine | 500 | 1.06143498335 | 279.876321793 |

## 2.2 Movie-movie similarity (10)

| Rating Method | Similarity Metric | K | RMSE | Runtime(sec) |
|---|---|---|---|---|
| Mean | Dot product | 10 | 1.02040776163 | 98.8140439987 |
| Mean | Dot product | 100 | 1.04683518601 | 101.887119055 |
| Mean | Dot product | 500 | 1.10883088024 | 112.398873091 |
| Mean | Cosine | 10 | 1.02565347723 | 106.61107111 |
| Mean | Cosine | 100 | 1.06191102228 | 108.902889013 |
| Mean | Cosine | 500 | 1.11528538719 | 115.316734791 |
| Weighted | Cosine | 10 | 1.01304374478 | 121.650727034 |
| Weighted | Cosine | 100 | 1.05470596369 | 124.992744923 |
| Weighted | Cosine | 500 | 1.09984570008 | 133.49823904 |

## 2.3 Movie-rating/user-rating normalization (10) PCC-based normalization

| Rating Method | Similarity Metric | K | RMSE | Runtime(sec) |
|---|---|---|---|---|
| ~~Mean~~ | ~~Dot product (same to cosine similarity)~~ | ~~10~~ | ~~1.05233613555~~ | ~~259.169136047~~ |
| ~~Mean~~ | ~~Dot product~~ | ~~100~~ | ~~1.05239875621~~ | ~~263.152855873~~ |
| ~~Mean~~ | ~~Dot product~~ | ~~500~~ | ~~1.06458391599~~ | ~~279.288192987~~ |
| Mean | Cosine | 10 | 1.05233613555 | 259.169136047 |
| Mean | Cosine | 100 | 1.05239875621 | 263.152855873 |
| Mean | Cosine | 500 | 1.06458391599 | 279.288192987 |
| Weighted | Cosine | 10 | 1.05209425787 | 275.688817978 |
| Weighted | Cosine | 100 | 1.05192781855 | 279.877198935 |
| Weighted | Cosine | 500 | 1.06335284489 | 294.373552799 |

Add a detailed description of your normalization algorithm.

The normalization process strictly follows the page 22 from lecture 10.

Initial Vector: $\vec{x} = (x_1, \cdots, x_n)$

Centering: $\bar{x} := \frac{1}{n}\sum_{j=1}^{n} x_j, \quad x_j' := x_j - \bar{x}, \forall j$

Normalization: $\| x' \| := \sqrt{\sum_{j=1}^{n} (x_j')^2}, \quad \vec{z} := \frac{x'}{\| x' \|}$

Equivalently: $z_j := \frac{x_j - \bar{x}}{\sqrt{\sum_{j'=1}^{n} (x_{j'} - \bar{x})^2}}, \forall j$

Clearly: $\bar{z} = 0, \quad \| \vec{z} \|^2 = \sum_{j=1}^{n} z_j^{\,2} = 1$

1. Read from train.csv the original user-matrix sparse matrix M
2. M is updated to (M.todense() subtracts the row mean vector of M)
3. M_scale = Normalize M by row using L2 norm

4. Normalized user_similarity_matrix = M_scale .* M_scale'

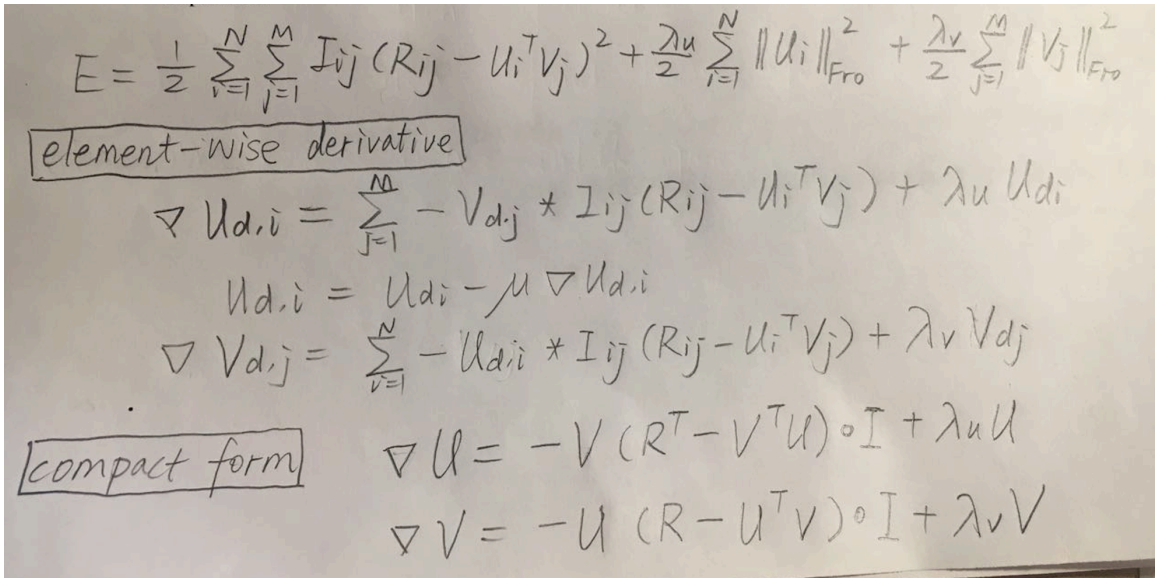
Note that after normalization, there's no distinct difference between dot product and cosine similarity.

## 2.4 Matrix Factorization (20)

a. Briefly outline your optimization algorithm for PMF

I'm using batch gradient descent for updating.

Writing in element-wise form and compact matrix form is as follows,




Initial weights for user matrix and movie matrix follows Gaussian with zero mean and unit variance. Each iteration takes 15s to finish. For each iteration, we update the entire U and V matrix. The learning rate is adjusted dynamically. If it hits the gradient ascent direction, learning_rate /=2, otherwise learning_rate *=1.25.

My parameter setting generally follows the paper experiment section, where given a learning_rate is set to 1e-4 due to the batch gradient descent nature. Since the number of users is twice the amount of the movie, we follow the ratio of sigma_u and sigma_v = 1:10. sigma_square_u = 10 sigma_square_v = 100

As an alternative, we could do mini-batch gradient descent, please look for further analysis below in implementation session.

b. Describe your stopping criterion

As optimization takes up around ~1hr to get an acceptable RMSE, I was thinking of a smarter early stopping criterion, several choices available include:

1) When objective function update value absolute difference smaller than a threshold
2) When objective function update difference percentage smaller than a threshold => specify a percentage threshold, smaller by x% or y total, then stop
3) When RMSE update less than certain percentage
4) After a fixed number of iterations, say 400

I chose 4) as a result, as we are sweeping over different settings of latent factor dimensions, I would also like to see how different dimensions affect the convergence speed/quality.

| Num of Latent Factors | RMSE | Runtime(sec)* | Num of Iterations |
|---|---|---|---|
| 2 | 0.962425319385 | 4823.12958192 | 400 |
| 5 | 0.939549025626 | 4975.18186402 | 400 |
| 10 | 0.951329107942 | 5291.97232985 | 400 |
| 20 | 0.978069483797 | 5451. 68090081 | 400 |
| 50 | 1.04464483264 | 5893.95524311 | 400 |

## 3. Analysis of results (20)

For experiment 1, weighted mean is done with normalization based on similarity measure.

$$w(u,u_i) = \frac{\cos(u,u_i)}{\sum_1^k cos(u,u_i)} \quad w(u,u_i) = \frac{dotp(u,u_i)}{\sum_1^k dotp(u,u_i)}$$

Experiment 1 shows an increasing RMSE along with increasing K. An optimal setting of K for this collection would be around 10, which means your ratings are largely represented by the 10 most similar users in the collection.

Dot product similarity over users is more effective in terms of RMSE than cosine similarity. This is reasonable in the following example.

Suppose there are users A, B and C. Suppose A rated two movies p and q both as 5. B is a hot user who rates a lot of movies as 5 stars, including q.C is a cold user who rates only one movie q as 5.

Dotp (A, B) = Dotp (A, C) while Dotp (A, B) < Dotp (A, C). Without given more information, it is more suitable to choose B as A's nearest neighborhood than C. Since C is a cold start user, his user profile vector is sparse, using this vector to make prediction suffers from data sparseness. He could not give many useful info for predicting a movie-

user pair. This also reveals the shortcoming of memory-based CF algorithms, when data are sparse and the common movies are few for most user pairs.

Dot product is also slightly efficient in computation than cosine similarity (without normalization, neat math equation).

Experiment 2 shows again an increasing RMSE along with increasing K. An optimal setting of K for this collection would be below 10, as when K=10, RMSEexperiment1 < RMSEexperiment2, which means a movie might need less than 10 neighbors to represent itself. A reasonable guess of optimal K is 5. It is reasonable as new movies are famous for its novelty, not how similar they are to many other existing movies.

This validated the two observations in experiment 2 that gives slightly larger error. Both of them set K=500. Note the dataset has only 5000 users. Intuitively, it is unlikely to have as many as 500 similar movies. The K value is set overly high to introduce more bias instead of improving on accuracy.

In experiment 2, cosine similarity with K=10, weighted mean gives the best performance. Dot product does not normalize over movie profile as some movies are inherently popular and controversial (have many ratings). These movies are dominant to become neighbors of many less popular movies in dot product similarity. We would prefer cosine similarity.

Comparing the running time between experiment 1 and experiment 2, the running time for model-based item-item similarity approach is around ½ of the user-user similarity approach. Its because the item-item similarity matrix is around ¼ the size of the user-user similarity matrix. Searching for kNN movies requires less time.

Discuss the complete set of experimental results, comparing the algorithms to each other. Discuss your observations about the various algorithms, i.e., differences in how they performed, what worked well and didn't, patterns/trends you observed across the set of experiments, etc. Try to explain why certain algorithms or approaches behaved the way they did.

In experiment 3, PCC-based method is more stable with different K settings. I was guessing PCC-based method groups similar users into more intra dense clusters. Within the cluster, the users are intrinsically similar (similar rating behaviors, similar movie subset) rather than superficially similar (rated the same movie with same rating). This cluster density difference and intrinsic similarity allows more flexible K.

I didn’t choose movie-rating standardization. Popular movies do not introduce movie bias. A higher average rating is an indication of the underlying movie quality. Therefore,

I was trying to solve the problem of generous users, i.e. user bias. I applied user-rating standardization.

Even though the motivation perfectly makes sense, and I was expecting PCC-based method to output the previous experiment 1 and experiment 2 method, the empirical result turns out PCC-based method is slightly worse than experiment 1. My explanation is that PCC-based method filter out user bias in suggesting KNN. For example, A and B are similar under Pearson's correlation coefficient. But when calculating a specific movie rating (t for user A), we should add a linear transform (factor back the eliminated user generosity bias) userB's rating based on his generosity (mean difference between B and A). rating (userA, t) would more likely to be 4 instead of 5. The linear transformation is missing. Of course this assumes there's no dependency between user A and user B's rating for movie t (an exception is that user A and user B watched movie t together and discussed to agree with each other's comment on this movie t).

| | p | q | r | s | t |
|---|---|---|---|---|---|
| User A | 2 | 3 | 4 | 3 | ? |
| User B | 3 | 4 | 5 | 4 | 5 |

In general, PCC has the problem that users with few rated items in common will have very high similarities. In specific, the Pearson correlation over a single pair of ratings is undefined (division by 0); over 2 pairs it is 1. This is not a problem in our normalization process after imputation (subtract by 3). But normalizing a user profile will make the user-movie matrix denser. Computing user-user similarity takes slightly longer time from the denser user-movie matrix.

In experiment 4, I did not do the imputation. Since we are only utilizing training instances to minimizing the objective function. Imputation is a general way of treating unobserved user-item pairs, which with the existence of the identity matrix, does not contribute to the objective function.

Generally, we would like to set num_latent_factor as 5. As it yields the minimum RMSE under same amounts of iteration/time. What num_latent_factor suggests is the underlying generative class of user-movie pair. As we could see, 50 is too large as to user-movie categories count. Increasing num_latent_factor slightly increases over training time. The iteration time is proportional to matrix size, while in mini-batch gradient descent we could implement it in a partial way proportional to mini-batch size.

## 4. The software implementation (5)

A side note: the software/codes can be automatically run on unix.andrew.cmu.edu or linux.gp.cs.cmu.edu, (as tested, assuming Python scientific computation library SciPy and NumPy are automatically installed) Please find the README.txt file for instructions on running the code. Please also find the xinq-test-clusters.txt file as the final document clustering result on the test set. Thanks for your support!

1. A description of what you did to preprocess the dataset to make your implementations easier or more efficient.

For experiment 1 and 2, I read in the sparse user-movie matrix and calculated the user-user/movie-movie similarity matrix in memory. For diagonal element, the value is reset to matrix global minimum as we want to exclude the user/movie itself in its k nearest neighbor. Calculating k nearest neighborhood online is an expensive operation, which requires pairwise similarity and might cause duplicate calculation. This preprocessing make the implementation more efficient.

2. A description of major data structures (if any); any programming tools or libraries that you used;

The software is developed under Python 2.7.10 with external SciPy and NumPy library for simple linear algebra operations and sparse data representation. There are 4 modules, userUserSimilarityPrediction.py, itemItemSimilarityPrediction.py, PCCSimilarityPrediction.py and PMFSimilarityPrediction.py, which corresponds to the four required experiments.

Reading the train.csv file once could keep the user-movie matrix (sparse) in memory. Besides, I calculated along axis=1, the user rating average for a given movie and along axis=0, the movie rating average for a given user. This solves the cold start problem.

In experiment 1, for every user, we are expected to lookup the user-movie matrix to get its k nearest neighbors. My implementation is dataset-oriented memory-based approach. There are 30000 movie-user pair to predict, if we do memory-based approach, iterating/looping over each user pairs, then each user has its kNN computed for around 6 times (30000*10000/(10000*10000/2)=6). We might not want to compute 30000 user kNN without caching, as user-user similarity is reciprocal, computing similarity (A, B) implies similarity(B, A). Caching helps but it would also make the implementation be somewhat similar to a model-based approach, after each similarity calculation, we throw

away the ratings. Therefore, for both experiment, I implemented a memory-based user-user/movie-movie similarity matrix.

A corner case treatment for cold-start users is if a user (movie) does not occur in the training data, then in the sparse user-movie matrix, the user/movie is represented by an empty (all zero) vector. It will not have any specific nearest neighbor as the dot product or cosine similarity is 0. For these cases, I use the average rating for the movie across all users to predict this movie-user pair. More often than not, none of a user's KNN gives any rating for a particular movie, for this case, I use again the average rating for the movie across all users to predict the rating. The corner case treatment for cold-start movies is also the analogous.

3. Strengths and weaknesses of your design, and any problems that your system encountered;

The data structure allows read once, write once from/to disk which reduces File system IO as to improve on calculation efficiency.

For PMF-based approach, I derived a compact matrix form derivative (gradient on U and V) to speed up the batch gradient descent process. Batch gradient descent is the most intuitive way of optimization. However, the partial update nature of mini-batch gradient descent in terms of convergence speed and computational complexity might improve on program efficiency. As an afterthought, I tested over minibatch gradient descent, with a batch size 10000. The convergence is not advantageously faster but slightly faster than batch gradient descent. A possible reason is because I did not use compact matrix form update for mini-batch instances. Doing element-wise update can be slow in Python as there's the for loops. For each iteration, batch gradient descent takes ~15s to finish while mini-batch gradient descent takes ~10s to finish. As objective function is minimizing, the difference in terms of training time diverges from each other.

Below is the number of iterations (times) taken to reach a specific objective function value level. From the table we could see batch gradient descent converges slower than mini-batch gradient descent converges equally fast. Given the total number of training instances as 820367. An optimal mini-batch size would be 10000.

| Objective function value | 500000 | 450000 | 420000 | 40000 |
|---|---|---|---|---|
| Batch Gradient Descent | 311.104706049s | 537.968626022s | 979.846630096s | 1680.37237096s |
| Mini-Batch Gradient Descent | 220.343675852s | 438.224372864s | 850.751818895s | 1590.83728886s |