11-641 Search Engine Homework #5 Report

Name: Yitong Zhou ID: yitongz

1 Experiments and Analysis

1.1 Corpus Exploration

Corpus Statistics:

Total number of movies: 5392
Total number of users: 10916
Number of times any movie was rated '1': 57887
Number of times any movie was rated '3': 278902
Number of times any movie was rated '5': 149542
Average movie rating across all users and movies: 3.38029

User ID 1234576:

Number of movies rated: 50
Number of times the user gave '1': 1
Number of times the user gave '3': 14
Number of times the user gave '5': 7
Average movie rating for this user: 3.56

Movie ID 4321:

Number of users rating this movie: 18
Number of times the movie was rated '1': 2
Number of times the movie was rated '3': 9
Number of times the movie was rated '5': 2
Average rating for this movie: 2.94444

1.2 Basic rating algorithms

The basic algorithm all follows the standard two steps to predict the new user movie pair's rating:
Step (1) Use Pearson Correlation to find the K nearest items (users/movies);
Step (2) Weight sum the K items score towards the certain user/movie.

And the detailed similarity description is as follows:

(1) Basic User-User Similarity

Similarity Metric: Pearson Correlation

For given to-be-predicted $< u_1, m_1 >$ pair, for each user u_i, the similarity with u_1 equals to:

$$sim(u_1, u_i) = \frac{\sum_i (r_{u1,i} - \bar{r}_{u1})(r_{ui,i} - \bar{r}_{ui})}{\sqrt{\sum_j (r_{u1,j} - \bar{r}_{u1})^2} \sqrt{\sum_j (r_{ui,j} - \bar{r}_{ui})^2}}$$

(2) Basic Movie-Movie Similarity

Similarity Metric: Pearson Correlation

For given to-be-predicted $< u_1, m_1 >$ pair, for each movie m_i, the similarity with $m_1$equals to:

$$sim(m_1, m_i) = \frac{\sum_i (r_{m1,i} - \bar{r}_{m1})(r_{mi,i} - \bar{r}_{mi})}{\sqrt{\sum_j (r_{m1,j} - \bar{r}_{m1})^2} \sqrt{\sum_j (r_{mi,j} - \bar{r}_{mi})^2}}$$

(3) Rating Normalized Movie-Movie Similarity

Similarity Metric: Pearson Correlation

For given to-be-predicted $< u_1, m_1 >$ pair, for each movie m_i, the similarity with $m_1$equals to:

$$sim(m_1, m_i) = \frac{\sum_i (r_{m1,i} - \bar{r}_{m1})(r_{mi,i} - \bar{r}_{mi})}{\sqrt{\sum_j (r_{m1,j} - \bar{r}_{m1})^2} \sqrt{\sum_j (r_{mi,j} - \bar{r}_{mi})^2}}$$

Rating Normalization: given the to-be-predicted $< u_1, m_1 >$ pair, for each user u_i belongs to m_1's rating list, do the following normalization:

$$R_{normalized} = R_{u_i} - (\mu \bar{R}_{u_i} + (1 - \mu)\bar{R})$$

($\bar{R}$ is the total average rating of the corpus, equals to 3.38029 and μ is a smoothing variable equals to 0.95)

And after weight summing all the ratings, I use the following formula to restore the score back to [1,5] range:

$$R_{predicted} = R_{predicted\_normalized} + (\mu \bar{R}_{u_1} + (1 - \mu)\bar{R})$$

The reason to add a μ is because sometimes, if a UserList or MovieList only has one non-zero rating for a given vector, the score-average will always be zero, making this method not quite effective in predicting these numbers. By adding a smooth variable, the RSME can be slightly improved and more importantly we can assure almost every one will get different treats instead of being the same.

(4) One more interesting thing:

Actually I also find a very interesting similarity, which is called dice coefficient[1]. For two vector X Y, the formula is as follows:

$$Sim(X, Y) = \frac{2|X \cap Y|}{|X| + |Y|}$$

The similarity equals to 2 times of total common elements of X Y divides the total element number of X and Y. Which in our case is quite a suitable metric. For two users, if their all rate a same movie with the same score, then $|X \cap Y|$ should increase one. I tested it in small query, using algorithm 1, and the performance is around 0.93 which is better than cosine similarity metric only a little worse than Pearson correlation. I should try it more in the feature, but if I use this metric, it will be quite hard to normalize and integerate other features into the model, so I finally only consider it an interesting finding.

Also please see 2.2 Architecture (1) architecture (2)process for detailed step description of the basic algorithm implementation.

[1] http://en.wikipedia.org/wiki/Dice's_coefficient

1.3 Self-developed rating algorithm

(1) Idea Derivation

My self-developed algorithm is based on the observation that either User-User and Movie-Movie similarity viewed the user-movie score pair from one side and almost does not consider the other's effect except the rating normalization at the last step. On the other hand, the Pearson Correlation, which in math is actually the expectation of co-variance, whose range is within [-1,1] could not only be viewed as a metric for similarity, but also viewed as a metric for possibility of correctly representing the query pair (by make an absolute, the range becomes [0,1]). That is to say, I view the recommendation problem as follows: with m users and n movies forming an $m \times n$ pair space, while predicting a new pair < x, y> is equivalent to calculate the expected score for this new pair given each pair existing in the space could all have a probability to have the same rating with the space:

$$\begin{aligned} Rating_{predicted} &= E\big(Rating(< x_{new}, y_{new} >)\big) \\ &= \sum P\big(< x_i, y_j > \big| < x_{new}, y_{new} >\big) Rating\big(< x_i, y_j >\big) \\ &= \sum \frac{P(x_i)P\big(y_j\big)}{P(< x_{new}, y_{new} >)} Score\big(< x_i, y_j >\big) \\ &\big(i, j \; is \; any \; pair \; exists \; in \; current \; train \; dataset\big) \end{aligned}$$

Here I assume that for one user and one movie, they are independent to each other, they do not affect each other's probability of correctly representing the future rating result. So actually if we separately calculate out the Pearson correlation between x_{new} and x_i , y_{new} and y_j, we could assume it to be an metric for the P(x) and P(y) probability. Also I consider $P(< x_{new}, y_{new} >)$ a constant, which means I assume that all pairs have equal probability to appear.

So finally we can form a model of "**top K pairs**". Instead of select out the top K user-users or the top K movie-movies, I first calculate both pairs, and then make them together form a sub-space of existing scoring pairs, then sort out the top pairs and finally use that pair set to approximate new user-movie pair's rating.

(2) Detailed Description:

The detailed algorithm is described as follows (to improve results, I also applied many heuristic techniques):

Similarity metric for User-User: $z_i \cdot z_j$ (Pearson correlation similar to the basic algorithms)

Similarity metric for Movie-Movie: $z_i \cdot z_j + 0.8\frac{x_i \cdot x_j}{|x_i||x_j|}$ (Pearson correlation plus a weight of cosine similarity)

Step 1: For one predicting pair of user, movie, calculate the top K user-user pairs on the basis of Pearson correlation;

Step 2: For one predicting pair of user, movie, calculate the top K movie-movie pairs on the basis of a combination of Pearson correlation and cosine similarity.

Step 3: For each top K users, check each movie of each user, when the movie is also in the top movie list, multiply their similarity scores and put this pair into a list for collecting top user-movie pairs;
Step 4: Sort the top user-movie pair list for top K/2 pairs.
Step 5: For each <user,movie> pair in the list, normalize the rating with the following formula:

$$r_{normalized} = \frac{r_{original} - \frac{avg_{user} + avg_{movie}}{2}}{\sqrt{var_{user}\, var_{movie}}}$$

Step 6: Use the multiplied similarity metric as the weight and the normalized ratings as the basis to calculate the weighted sum as the final normalized score for given user and movie.
Step 7: Use the following formula to restore the score into [1,5] range:

$$r = r_{sum\_normalized} * \sqrt{var_{newuser} var_{newmovie}} + (avg_{newuser} + avg_{newmovie})/2$$

(3)Heuristic Improvement
1) I round a final predicted score to integer if the number is less than 0.1 away from the nearest integer.

2) I tuned the weight of the sum of Pearson correlation and Cosine similarity to achieve better performance.

3) I only introduce the weight sum of similarity in Movie-Movie top items choosing, because by practice such smoothing method while applied in User-User top items choosing, can only make the result worse.

(4) What works and what does not
My original purpose of this algorithm is to expand the possible similarity space of kNN algorithm so that the prediction can be based on more evidence. It turns out to improve over the three basic algorithms but not at a very distinct level even though I adapted many trivial optimizations.

One of the most striking things during exploration is that if I normalize the ratings by further dividing the variance, the result will become much worse instead of becoming better. And the smoothing method, which could be quite effective in Movie-Movie finding sub-algorithm, actually does not work well in User-User findings.

If consider the previous problem I encountered in the UserUserNorm algorithm (that normalizing ratings harm the performance), I think maybe for this data set the assumption that there is an obvious bias over movies can be true, but the bias between users could be a much complicated.

1.4 Experiment Result

	Algorithm	RMSE	Runtime	K
Big Dataset	User-User	0.990143685437	91s	1000
	Movie-Movie	1.02789697032	37s	1000
	Movie-Movie-Norm	0.960577340711	46s	1000
	User-User-Norm*	1.01420357885	158s	1000
	Self-Defined Top K Pairs	0.948447537518	3387s*	1000
Small Dataset	User-User	0.922568645874	13s	1000
	Movie-Movie	0.94265494572	2s	1000
	Movie-Movie-Norm	0.890394383037	2s	1000
	Self-Defined Top K Pairs	0.885510235135	78s	1000

* User-User-Norm algorithm is quite lousy that I do not include it into my final codes and only carried out a big dataset experiment.
* The long running time of my self-defined top K pairs is partially due to I do not implement the 2nd level cache and need-for-calculation id filtering for this algorithm, because I keeps changing the structure and ideas of my self-defined algorithm that I need a flexible design to keeps changing every detail of the algorithm implementation.

(1) General Performance

As we could see, the performance rank is quite stable from best to worst as: Self-Defined Top K Pairs, Movie-Movie-Norm, User-User, User-User-Norm, Movie-Movie. Of all the algorithms, my self-defined algorithm has the best performance since I keep iterating the implementation until I see an advantage over other algorithms.

However, the algorithm performed best from a full point of view should be the Movie-Movie-Norm algorithm, since it achieves a second best RMSE with almost he least of running time, which is very impressive.

And the most unexpected result is the failure of User-User-Norm, actually I tried every possible different normalization methods for the ratings, and the result just walks towards worse instead of having any improvement. Later I will discuss further the possible cause for such failure.

(2) Time Cost

Expect my self-defined algorithm is extremely slow compared to others (because it does not have the second level cache support and also does not filter the least needed id sets for calculation.), other programs all perform with quite reasonable time consuming. Since in the total number of movies is only half that of users, it is unsurprising to see that in large data result, the time consuming difference is also about 2 times.

(3) Movie-Movie vs. User-User

Before normalization, Movie-Movie algorithm's performance is very bad and User-User algorithm's performance seems already has been quite good even without any rating normalization. But after normalization, Movie-Movie algorithm gains significant improvement while User-User algorithm gets even worth result. This problem is actually very interesting to look. I think the Movie-Movie algorithm

improvement may result from the strong bias between movies. Some movies are very touching, every one rates high while others quite lousy, and no one gives any credit. Rating normalizing could actually be very effective that finally uses this feature to gain better performance. On the other hand, the bias lies in users may be more complicated, that not simply bias over all movies but maybe bias over certain types of movies. So simple rating normalization just is not quite enough to eliminate such bias effects and I do believe we need further exploration to clear this problem.

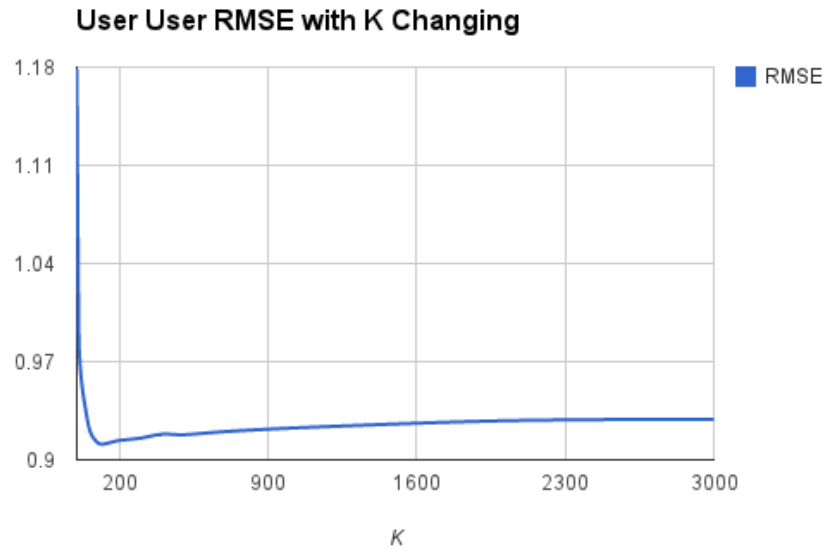
(5) Top K Pairs

My top K Paris seem to be better, but its high run time disadvantage is also obvious. Although from the time complexity point of view, it is still a o(nxn) algorithm but with larger basic numbers without using two level caches. And even use cashes theoretically I think it should still be about 2.5~4 times slower than other algorithms, so theoretically it can be an improvement, but in practical use I would rather choose the Movie-Movie-Norm algorithm.

(6) RMSE with k changing

I did an experiment with k changing in algorithm 1 -----User User algorithm.

K	RMSE	Time
1	1.178316491	15
10	0.990876221	8
50	0.928998945	8
100	0.912095883	9
200	0.913799447	9
300	0.915448906	8
400	0.918266511	9
500	0.917854668	9
700	0.920151446	9
1000	0.922568646	9
2000	0.92785273	9
3000	0.928998945	9



It is quite interesting to see that actually K is not as large as good. Which means kNN algorithm is both more accurate and time saving compared to directly use a matrix to calculate all the similarity scores. And I also tried other algorithms and found the best fit K are quite different, with algorithm 1 around 100, algorithm 2 and 3 around 1000 and algorithm 4 around 800. Which means if in real system usage, as long as we receives new data or adjusted the previous model, we had better again tune the k to achieve a better performance.

(7) Some conclusions

- Running time and RMSE can be improved at the same time
- Optimize the original lousy algorithm can sometimes gain much bigger improvement.
- The normalization theory seems to be powerful, but it does not always help in practice. Sometimes basic and simple ideas can be more useful.
- K value's choice can also be a tricky part. Too large K values means no real sense of recognizing important items among all, while too small K always leads to unstable predictions.

2 Software Implementation

2.1 Pre-processing

(1) In-memory Movie/User Dictionary

I read all training set into memory and store each user movie pair for duplicate copies ----- one column copy and one row copy. I use two tree maps to implement id-list mapping. So every time I need either a movie or user vector, the retrieval time complexity is always O(logn) and all are proceeded in memory.

(2) Need-for-calculation filtering

I read query files one more time before formally begin the calculation process. And use two maps to store the following information:

1) For each user appears in the query file, record all the ids of movies in the query file that occurs with him/her.
2) For each movie appears in the query file, record all the ids of users in the query file that occurs with the movie.

I do this is based on the analysis that not every user or movie is important while calculating the top K item. For example, in user-user similarity case, for given movie id and user id in the query file, we only care about those users actually occur in the current movie's score list, if not, even we pick them as top users, their contribution to the final sum weight of the score would be 0.

But also, if we filter each user separately to generate a need-for-calculation id set would be too time expensive. So I use a compromise strategy to get all the relevant movies at once and then iterate every item stored in these movie lists to get a sub set of users. This could effectively reduce time consumption by 40%~66%.

2.2 Architecture

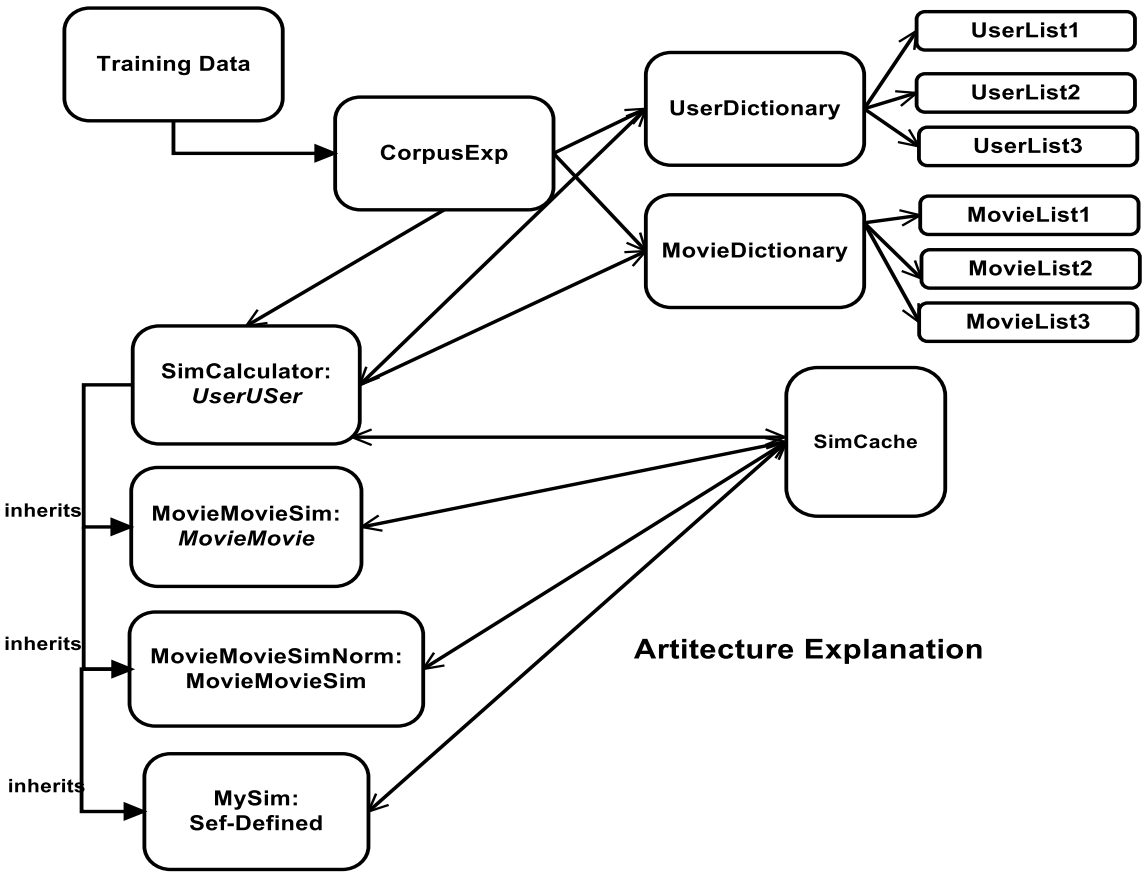
(1) Architecture:

The basic design idea of my software is to user 3-level in-memory storage to accelerate the data processing as fast as possible:

(1) The 1st level in-memory storage is the Movie dictionary and User dictionary, which separately make duplicate copies of the matrix by storing all the rows and all the columns, all in memory. The access to these data, whether through user_id or movie_id is extremely fast.

(2) The 2nd level in-memory storage (also the 1st level cache) stores all already calculated Top K similarity lists into a map, so that for each unique user or movie's list, we only need to calculate once.

(3) The 3rd level in-memory storage (also the 2nd level cache) stores a symmetric similarity score into a partially formed similarity score list. That is to say, if we have calculated u1 with u2, we never need to store u1's cache list again, because it will only be calculated once, and next time we will use the 1st level cache. So we only need to store u2's similarity score of u1. And if we use a map of id keys and list pointers values, the partial stored cache list can be used as the partially finished similarity score list for the certain item as long as it is to be calculated.



(2) Process:
My basic program process is as following:
Step 1: CorpusExp Class reading the training data set file and load all data into Movie Dictionary and User Dictionary.
Step 2: A Calculator Class (SimCalculator, MovieMovieSim, MovieMovieSimNorm, MySim) is called and begins the calculation.
Step 3: First go over the query file for one time, for each user appears in the query file, record all the ids of movies in the query file that occurs with him/her; for each movie appears in the query file, record all the ids of users in the query file that occurs with the movie.
Step 4: Read the query file for a second time, and for each line calculates the score and prints out.
Step 5: Calculation needs two list: a Top K items list with similarity scores and an item list for weight sum score prediction. First check in-memory cache to see the Top K item list whether already stored in cache, if yes, read it and directly go to score summing process; if not, begin Top K items calculation.
Step 6: For each Top K items calculation, first begin a need-for-calculation id filtering process, if the similarity already occurred in cache, do not calculate; if the id does not appear in the step 3's map's correspondent id list, do not calculate.

Step 7: Begins Top K items calculation, for each similarity result, store a cache for future use, e.g. If we finish calculate sim(a1,a2), then we store a sim(a2,a1) (give the same score to the cached list for a2).

Step 8: After finishing calculating, partial sort the similarity score list by score, resize the list to K length, and then sort the list by ID ascending.

Step 9: Put the top K list into corresponding id's map, go on through to calculate the score and move on to the next line of data.

2.3 Major Data Structure

(1) MovieDictionary
This is singleton static in-memory storage of the train dataset. I use a <movie_id, MovieList pointer> map to store the data, and supports iterator.

(2) UserDictionary
This is singleton static in-memory storage of the train dataset. I use a <user_id, UserList pointer> map to store the data, and supports iterator. Together with MovieDictionary, the original training data are stored in-memory for twice, every row into a map once and every column into a map once.

(3) ScorePair (id, score, score_norm)
This is a basic struct for both rating calculation and similarity score calculation, composed of three elements: id, score, score_norm. The id could be user_id or

movie_id, depends on which class is using it. And the score could be used to store similarity score or ratings. The score_norm is designed to store normalized scores. I deliberately make this as a duplicate element because sometimes, we may need the normalized score and the original score at the same time. Such design will be much more flexible. All the basic d

(4) MovieList
This is a rating list of a given movie_id, composed of ScorePair elements with user_id and rating score. It supports iterator, normalization, sort by ID and statistical numbers like average and variance. It is mapped into MovieDictionary through the movie_id.

(5) UserList
This is a rating list of a given user_id, composed of ScorePair elements with movie_id and rating score. It supports iterator, normalization, and statistical numbers like average and variance. Interestingly, during the formation of this list, it is originally generated in ascending ID order, so there is no need to implement a sort function.

(6) Map in SimCalculator Class (First level cache)
This is implemented as a map in SimCalculator Class to directly store the already calculated top k items for a given user/movie. Every time we need calculate scores, we first check whether there is already cached data, if not any, then we carry out the getTopK() task.

(7) SimCache (Second level cache)
The first level cache used to store the symmetric similarity scores. Every time we need to calculate the top k movie or top k users, we first read this memory to reuse the scores we have already calculated before. And after calculation of a score, we also insert calculated similarity into this map to make program run faster in the future.

2.4 Tools and Libraries
Valgrind: a memory leak detector
GCC: G++ compiler
Make: compile tools
Sublime Text2: a nice text editor

2.5 Strength, Weakness and Problems
(1) Strength

- Written in C++, very high efficiency, event most complicated algorithm can be completed within 33%~50% time compared to Java program;
- Make multiple similarity calculation classes all inherits the same original calculation class and efficient reduce duplicate codes.
- Only use std lib to maintain as much compatible as possible.

(2) Weakness

- Due to the variety of C++ compilers, my program cannot guarantee its safe running under different environments, really need further study to improve the ability to write compatible codes over different environments.
- Do not implement second level cache for my self-designed algorithm and making it really slow to run. Actually, one of my design decision to not to implement cache storage for the 4th algorithm is because previously I try hard to make the differences of ids of movie and user as little as possible. Only comprehend one is primary key and another is second key but does not really use different data types. The advantage to do so is it saves half of the duplicate codes, making my implementation quite light and simple. But the tricky problem is it becomes very hard for me to introduce cache storage for both user list and movie list ---- I have to rebuild an entire cache class to support such function. So I just decided not to do so, since algorithm 4 is changing so fast that I can hardly guarantee my architecture won't change swiftly.

(3) Problems

- The first tricky problem is about whether should we make an absolute value out the Pearson Correlation. It seems that making -1 correlation into 1 is not quite reasonable in math, however, doing such things can truly improve the performance of the recommendation. I am still not quite sure what causes this problem. One of my guess that also reversed, vectors with -1 correlation can still share the same expectation of rating with the +1 correlation vector, making it possible for us to use weight sum scores to estimate the predicted scores.
- The second is a debugging problem, I always get memory leak problems and finally found a tool called valgrind to detect and debug them, which is really really helpful.

3 How To Run

Coded under Ubuntu 12.04, I use g++ to compile the source code and the flags are:

-g –O2.

Only use std lib, but I do expect the source code may not compile in other Unix/ Linux systems, since the compiling standards are slightly different.

You can directly type the following command to run the code under Code directory:
sh run.sh –m [Model#(from 1 to 4)]
Every time, the script will call make to recompile. And the there will be a result file generated in the directory.

Also –s [none | -u USERID | -m MOVIEID] will show you the statistical data.
-h will show the help information.

The DATA.txt format: (The directory recognition is start from the Code directory)

Train=../data/download_sample/training_set
query=../data/queries.txt
k=10

Examples:
~/Code/: sh run.sh -m 1 // run the first algorithm model
~/Code/: sh run.sh –m 4 // run the fourth algorithm model
~/Code/: sh run.sh –s –m 4321 // get the statistical information of movie 4321