CS174: MACHINE LEARNING

PROJECT FINAL WRITE-UP

Personalized Song Recommender System

Michael Lau, Binjie Li, Beitong Zhang

March 15, 2015

1 INTRODUCTION

The aim of our project was to create a personalized song recommender system for a dataset taken from last.fm. More specifically, we endeavored to use user listening history to predict future behavior, which is a natural problem for any online music providing service.

First, we implemented a naive collaborative filtering neighborhood model, which provided surprisingly good benchmark results. Then we implemented a form of the latent factor model, which yielded results with much lower accuracy. Finally, we utilized SVD to reduce the dimensionality of the neighborhood model so that it can run more efficiently, which is a crucial requirement for online recommender systems.

2 DATASET AND DATA PREPROCESSING

We are using a real-world dataset [1] [3] containing the listening histories for 1000 different users from *Last.fm*. The size of the raw dataset is nearly **3GB** which is a really big dataset and surely quite difficult for us to analyze. Therefore, we first did analysis on the records to reduce the size of the dataset while keeping its underlying feature distribution as much as possible.

Figure 2.1 shows the history size distribution in our original dataset and we can tell that the size ranges greatly among different users. For those users with small history sizes, we believe that they are not good samples for our analysis based on two reasons. First, if we only know a little about a user's history, then it would be difficult to find similar users and infer his taste in music. Secondly, we aim to recommend new songs to each user, so for the user with a small

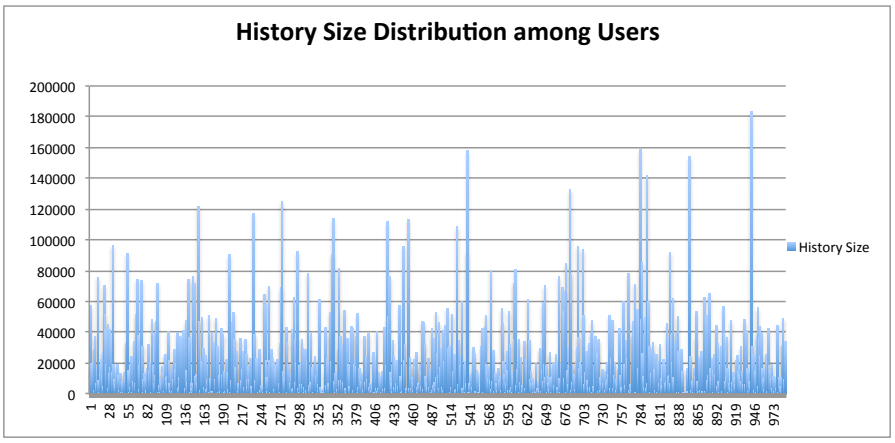

Figure 2.1: History Size Distribution among Users

history size, we won't have enough test data to predict the new songs he will listen to. We decided to filter users with history sizes smaller than 10K, leaving us with only 542 "valid" users out of 1K as our sample set.

In the raw dataset file, we have more than 100K artists and 1 million songs. Since putting them all in the user-item matrix is not computationally feasible, we analyzed the popularity of artists and songs and chose the most popular m artists and n songs to construct our dictionaries. For now, we've chosen $m = 3000$ and $n = 10000$.

We wrote a 300-line *Java* program to analyze the raw datafile and do the above two main things. The format of one history record in the raw datafile is as follows:

<timestamp, user_id, artist_id, artist_name, song_id, song_name>

After our program, we output two dictionary files for both artists and songs. Also, for each user, we create two vectors v_a and v_s corresponding to user-artists and user-songs:

$$v_a =< a_1, a_2 .... a_n >$$
$$v_s =< s_1, s_2 .... s_n >$$

where a_i and s_j are the times a user listens to the artist i and song j.

3 Neighborhood Model

3.1 Similarity

We determined the similarity of users x and y using simply the cosine of their vectors:

$$sim(x, y) = \cos(\vec{x}, \vec{y}) = \frac{\vec{x} \cdot \vec{y}}{||\vec{x}||_2 \times ||\vec{y}||_2} \tag{3.1}$$

As each user had three associated vectors, each pair of users had three associated similarities, based on the songs they had listened to, the artists they had listened to, and their user profiles.

Each of these similarities was given a weight in their composition to form the overall user-user similarity.

To optimize the weights, we iterated through different weight ratios, finding that a ratio of 100: 10 : 1, for the artist, song, and profile vectors respectively, was sufficient, with higher values for the artist and song vectors yielding only very marginal increases in accuracy.

3.2 Recommendations

The goal is to provide user x with a set of songs that he or she will probably like. This set, X_R, must be derived from the set $X_W = S - X_\ell$, where S is the set of all known songs and X_ℓ is the set of songs x has listened to.

For each song $S_i \in X_W$, we project a recommendation score r_{xi} based on the number of times other users U have listened to S_i, weighted by their similarity to x:

$$r_{xi} = \sum_{u \in U} sim(x,u) \times \frac{\ell_{ui}}{\sum_{j=1}^{|S|} \ell_{uj}} \tag{3.2}$$

ℓ_{ui} is the number of times user u listened to S_i. It is normalized by sum of all the times u has listened to any song, essentially giving the percent that S_i constitutes u's history.

This function was based off of one for a standard dataset, where a user's relationship with an item is determined by a rating, not a history. These ratings were normalized by subtracting the average of all the user's ratings [2].

$$r_{xi} = \bar{r}_x + k \sum_{u \in U} sim(x,u) \times (r_{ui} - \bar{r}_u) \tag{3.3}$$

Ratings and histories differ in their range of values, so normalizing using the original function was not effective. Also, because we are not predicting a rating, we do not need to incorporate the user's average rating or the normalization constant k.

3.3 Evaluation

Using the first n songs that X listened to, X_n, we found recommendation scores for all the songs that X hadn't listened to in the training set. The songs with the top k recommendations scores constitute the set $R \subseteq X - X_n$.

Our goal is to predict X_p, X's k most listened to songs in the test set minus the training set, i.e. their 'future' behavior: $X_p \subseteq X - X_n$.

We define accuracy rate z as:

$$z = \frac{\sum_{r \in R} r \in X_p}{m} \tag{3.4}$$

3.4 Results

As shown in Figure 3.1, the neighborhood model yielded surprisingly good results. To put them into perspective, after a user has listened to 3000 songs, our model accurately predicts

three of the five songs they will listen to the most that they have not yet listened to. With this kind of information, a music recommender system can provide real utility for the average user.

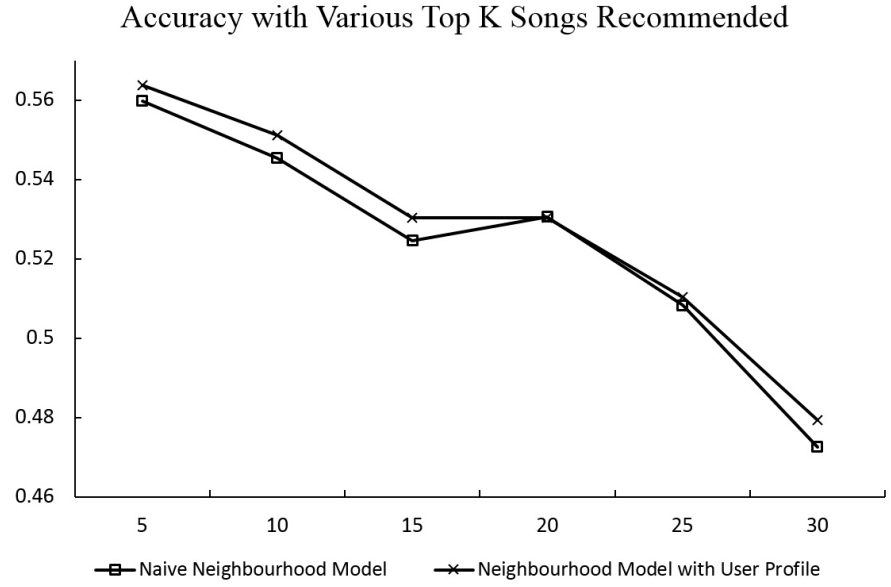

Figure 3.1: Accuracy with Various Top K Songs Recommended

There is a significant caveat for these results in that the sample size is only a little more than 500 users. When looking at the user profiles, it is clear that the users are actually very similar, which is when the neighborhood model works the best. In a real world application, the dataset would be much larger, and the similarity between users might not be so high.

4 Latent Factor Model

Our second approach, the Latent Factor Model [5] [4], predicts the "ratings" of items which is unrated in our training set. In the Neighborhood Model, our recommendations were based on the assumption that similar users will listen to similar songs. As mentioned in Sec. refsec:similarity, the similarity we used was the *cosine distance* of different users' listening histories, formatted as vectors. In the Latent Factor Model, we try to predict the "rating" of a user to all the songs which hasn't been rated and recommend songs with high "rating". The primary distinction from the Neighborhood Model is that we do not directly calculate the similarity between users, but instead elicit latent associations between users and items in the entire listening history matrix to indirectly find which songs to recommend.

4.1 General Idea

We assume that ratings are deeply influenced by a set of factors that are specific to the domain, e.g. the amount of characters in a novel, scene complexity in a movie. Since these factors

are in general very ambigious, it's hard to manually classify them and estimate their impact on the ratings. Hence, the Latent Factor Model uses mathematical techniques, e.g. matrix factorization, to infer these latent factors from the existing rating data. Given a particular user u and an item i, we can get two vectors, each representing how much interest user u has in these latent features and how related is the item i to these latent features. Such decomposition is executed on a partially defend rating matrix and we can use the inner product of these two vectors to predict unknown ratings. Since these two vectors are based on known ratings in the training data, it is crucial that the rating we have can reflect the users' interest well. Unfortunately, in our case the only data we have is how many times user u has listened to a song i, which is only a kind of implicit feedback which can reflect users' interest in songs. Hence, the results we get are not as good as the Neighborhood Model, and we will give detailed analysis later.

4.2 Learning Algorithm

In our model, as given in the Recommender Systems Handbook [5], we predict a rating according to the following rule:

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T p_u \tag{4.1}$$

Here $r_u i$ denotes an unknown rating in our data. μ is the average of the rating data, and b_i, b_u are the bias for user u and item i. These three factors are called *baseline prediction* because for example, it would predict a higher rating of items to a user if he in past always rated higher compared to other users. $q_i^T p_u$ is the inner product of user feature matrix and item feature matrix. We then loop through all the known ratings in the training set and compute the prediction error between the existent rating and the predicted rating as:

$$\hat{e}_{ui} = r_{ui} - \hat{r}_{ui} \tag{4.2}$$

To train the model, we define the loss function as:

$$\min_{b_*, q_*, p_*} \sum_{(u,i)\in\mathcal{K}} (r_{ui} - \mu - b_i - b_u - q_i^T p_u)^2 + \lambda(b_i^2 + b_u^2 + ||q_i||^2 + ||p_u||^2) \tag{4.3}$$

which is a squared error function with a regularization term. We use stochastic gradient descent to find the convergence since the quadratic function is guaranteed a global minimum. In every loop, for each known rating tuple, we update its corresponding parameters b_i, b_u, p_u, q_i in the opposite direction of the gradient.

4.3 Results

The accuracy rate evaluated here is the same as the one in the Neighborhood Model and we get the following result with different number of latent features:

As shown in Table 4.1, the results are relatively low compared to our earlier success with the Neighborhood Model. Here we list several main reasons why we think the Latent Factor Model is not a good choice in our case.

Table 4.1: Accuracy with various number of latent features

FeatureNumber	2	8	16	64	128
Accuracy	8.11%	9.37%	9.41%	8.98%	9.04%

We believe the main reason is that the Latent Factor Model works well on a dataset with explicit feedback, for example, the users' ratings for various items. Traditionally, a high rating reflects that user has interest in that item and lower rating means dislike. Based on such training data, the predicted rating would also reflects the user's taste or interests. Here our rating data is actually the number of times the song or artist has been listened to, and we try to predict the top songs the user will listen to the most in the future. The first problem is, in the training data, the number of times a song has been listened to can differ a lot from the user's actual interest, especially when the times are few. For example, for a song the user listened to only once, we cannot claim the user dislikes the song, because we believe the user would only repeat his favorite songs a lot of times. Hence, listening few times doesn't mean very low rating in the standard form. Unfortunately, such instances with very few listening times account for a very large proportion in our known data (as shown in the Figure 4.1 where tag 11 stands for the ratings above 10), and we believe this is the reason why the results we predict are not so accurate.

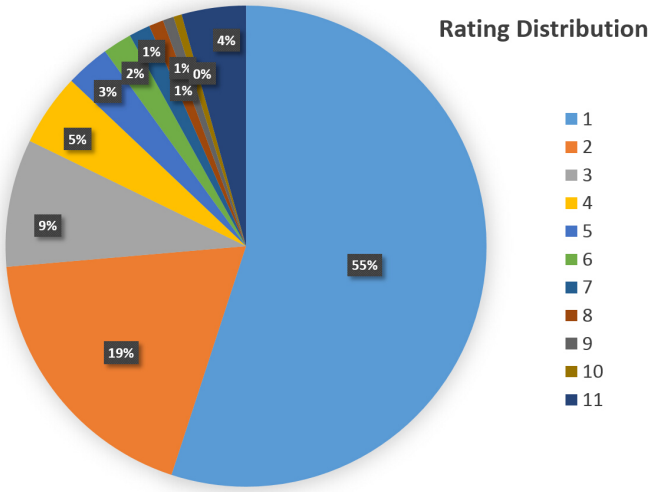

Figure 4.1: Distribution of the Ratings

During our presentation, Professor Torresani suggested we delete the ratings with small values, which is the data with few listening times, since such history is mostly noise and doesn't represent user's interest. We tried this idea and got slightly better results(as shown in the Table 4.2), but the improvement is negligible. We think excluding such data in our training stage is helpful, but the property of our "rating" data remains the same. For example, the residual rating distribution in our case is still quite different from the standard rating data, which is generally a Gaussian Distribution.

Table 4.2: Accuracy with various number of latent features(Improved Version)

FeatureNumber	2	8	16	64	128
Accuracy	10.96%	9.94%	9.82%	9.65%	9.26%

Secondly, we think negative samples should be considered in our case. As mentioned before, in our training stage, we use only the known rating data and ignore all zeros since we treat them as unobserved ratings. Actually, such zero ratings are meaningful to us. For example, if a song is very popular among other users but this user has never listened to it, we can infer that this user doesn't like this song. But the challenge here is how to rate such a negative sample, since our ratings have a large range and it is unclear how to differentiate whether a user dislikes a song or is simply ignorant of it. Another reason why we need negative sample is that in the Neighborhood Model, we calculate the relative distance with users and such zeros are considered because we calculate the difference in vectors. We haven't yet tried to implement this in our algorithm, and we shall treat it as our future work.

There are other reasons we think might cause the problem here, for example, the size of our user data is not large and diverse enough. Since users are quite similar, the Neighborhood Model should perform far better than the LFM.

5 SVD + Neighborhood Approach

After trying the LMF method, we decided to use SVD in another way. We started by using the built-in SVD algorithm in Matlab to factorize our original user-item matrix A.

$$SVD(A) = U \sum V^{\mathrm{T}} \tag{5.1}$$

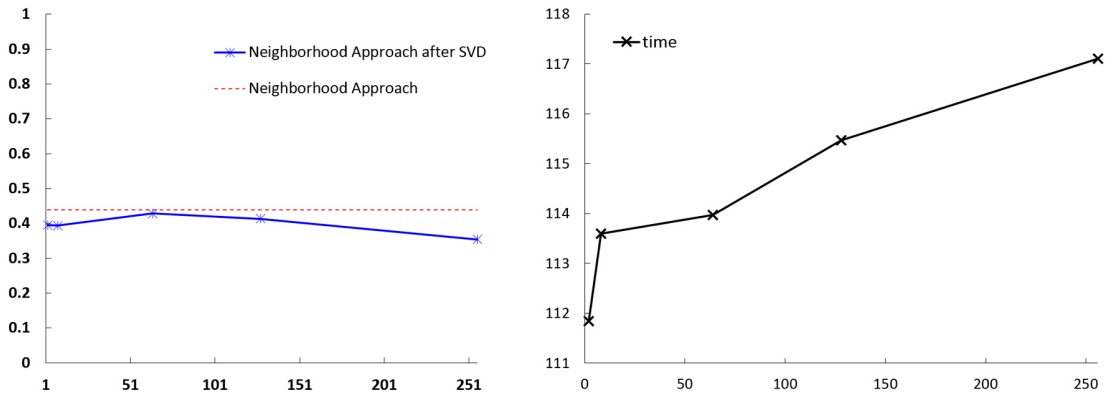

Figure 5.1: Performance of SVD+Neighborhood Approach

We then reduced the dimensions of A by selecting the first k columns of matrix U, thus getting the top k components of the listening history. This new matrix U' became our new user- item matrix. The neighborhood approach in Section 3 is used again on U'. We ran this

program with different remaining dimensions k and also calculate the running time of our program (SVD+neighborhood approach).

We can see that in the left of Figure 5.1, the accuracy of our prediction after using SVD is a little lower than the one using original data. However, the accuracy is still acceptable. The running time grows when we use a larger dimension k. For all of the dimensions, the running times are much better than in the original method (218 seconds). Thus, we have found a much faster recommender. This is very meaningful for a larger, real-world dataset that will be used for an online recommender system.

6 Conclusion

Facing a non-standard dataset, we first found a way to reprocess our data to fit the input of a traditional rating based recommender system. We then implemented two recommender systems with different models: the Neighborhood Model and the Latent Factor Model. Our results show that, though it is a simpler model, due to our uncommon data, the accuracy using Neighborhood Model is much better than the Latent Factor Model. We analyzed different possible reasons why the Latent Factor Model doesn't work well, which mainly centered around problems with our dataset. Furthermore, we tried different ways to improve the accuracy of both models, and made marginal improvements in both cases. We also found a way to apply SVD on the Neighborhood Model and achieve a better time-efficiency with an acceptable precision level.

Though it is slightly disappointing that our more sophisticated technique performed worse, the results from both methods had significant value. The metric we chose for evaluating accuracy was such that even a result of ten percent has a clear value for the user. If we can recommend just one song that will become one of a user's ten most frequently listened to songs, then we have succeeded. This is the result of only one term's worth of work, and it is clear that with a larger dataset and some more refinement to our techniques, the methods we have developed form the basis of a successful recommender system.

7 Implementation Details

7.1 Data Preprocessing

The folder **DataProcess** contains two *Java* files we wrote to process the original data. *DataProcess.java* does the job mentioned in Section 2. *dataProcessLFM.java* turns the user-song matrix, which is the output of the *DataProcess.java*, into a line based <user,item,rating> tuples as the input of our LFM algorithm.

7.2 Neighborhood Model

Our implementation of the Neighborhood Model required four functions in *Matlab*: *simUser*, *recommendations*, *accuracy*, and *test*. In addition we used a matlab script for implementing the tests necessary for analyzing our methods.

The simUser function simply implemented the cosine similarity function described above. The *recommendations* function recommends songs for a particular user, given a set of other users and all their listening histories. The *accuracy* function finds the accuracy for the recommendations for one user. Finally, the *test* function iterates through the data set, finding the recommendations and accuracy for each user. All of these functions were written by us.

7.3 Latent Factor Model

We wrote the LFM algorithm in Python based on the equations given in [5]. We use stochastic gradient descent to find the convergence since the quadratic function is guaranteed a global minimum. In every loop, for each known rating tuple (each line in the output file of *dataProcessLFM.java*), we update its corresponding parameters b_i, b_u, p_u, q_i in the opposite direction of the gradient. The training step would use *pickle* module to output a *lfm_model.pkl* file containing the information of the parameters of the model we get. The prediction step uses the model we trained to output a prediction file which contains the predicted rating matrix. We also wrote *Matlab* scripts to calculate the accuracy based on our evaluation rules mentioned in Section 3.3.

7.4 SVD + Neighborhood Model

For this part, we have almost the same *Matlab* code as the naive neighborhood model. We call the built-in SVD function in the *Test_Script.m* in the folder **SVD+Neighborhood**. As mentioned in Section 5, we use the similarity in the subspace to run our Neighborhood Model.

References

[1] last.fm music recommendation dataset. `http://last.fm`. Accessed: 2015-02-17.

[2] Gediminas Adomavicius and Alexander Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *Knowledge and Data Engineering, IEEE Transactions on*, 17(6):734–749, 2005.

[3] O. Celma. *Music Recommendation and Discovery in the Long Tail*. Springer, 2010.

[4] Yehuda Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 426–434. ACM, 2008.

[5] Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B Kantor. *Recommender systems handbook*, volume 1. Springer, 2011.