GAMES RECOMMENDATION SYSTEM BASED ON TIME PLAYED

Storm Ross, Yixiong Fan

New York University

ABSTRACT

In this project, we built recommendation systems for the popular digital distribution platform Steam. First, we transformed the hours users spent playing a game to an implicit rating for that game. Next, we built several recommendation models to train the data set we got. Finally, we evaluated them and compared their performance on the test data set.

***Index Terms*—** Steam, Games recommendation system, Play time, Collaborative filter, Matrix factorization

1. INTRODUCTION

For our project we built recommendation systems for the popular digital distribution platform Steam. Officially released and launched in 2003 by video game developer Valve Corporation and has become the largest digital distribution platform for PC gaming with 12.5 million concurrent users.

Unlike typical recommendation systems that rely on the users giving an explicit rating to items they have purchased we used the hours users spent playing a game as an implicit rating for that game.

2. DATA

The data that we are using to build these recommendation systems is a kaggle data set with users and the items they have bought through Steam as well as time spent using each item. After putting this data into a sparse user-item matrix we got a 11360 users by 3600 games user-item matrix with about 70000 non-zero elements(playtime). A few things that we notice about the data is that a significant proportion of the users in the data set have either one item purchased (roughly 40%) or have playtime on only one game (over 50%).

To address this issue we preformed preprocessing on the data to eliminate the users with only on game from the user-item matrix. Doing so won't hamper the effectiveness of our recommendation systems.

3. METRICS FOR EVALUATING MODEL PERFORMANCE

The metrics that we used to evaluate the performance of this recommendation system will be Mean Square Error (MSE) and Mean Absolute Error (MAE).

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{u,i}\left(r_{u,i} - p_{u,i}\right)^2}$$

$$\mathrm{MAE} = \frac{1}{n}\sum_{u,i}\left|r_{u,i} - p_{u,i}\right|$$

Where $r_{u,i}$ and $p_{u,i}$ are the rating and prediction of the rating user u gave item i respectively.

4. FEATURE ENGINEERING

For the problem of building recommendation system the only feature of concern would be the "ratings". In our case those would be the amount time each user spent playing a particular game they owned. An issue we encountered with these play-times was that the distribution of play-time hours were fairly spread out with values ranging from 0.1 to 11754.0.

To address this issue of the playtime having values so spread out we simply took the $\log_{10}$ of the hours. After doing so we can see in Figure 1 that the distribution of the playtime starts to resemble that of a normal distribution. But for our purposes this transformation of the data is not suitable enough for what we need to build the recommendation systems since they require the "ratings" to be strictly positive and for unrated items for a user are considered an implicit zero. To fix this we simply converted the playtime from hours to minutes and the added one before taking the log. Now as you can see in Figure 2 we still have a distribution that looks like a normal distribution and none of "rating" are zero or negative.

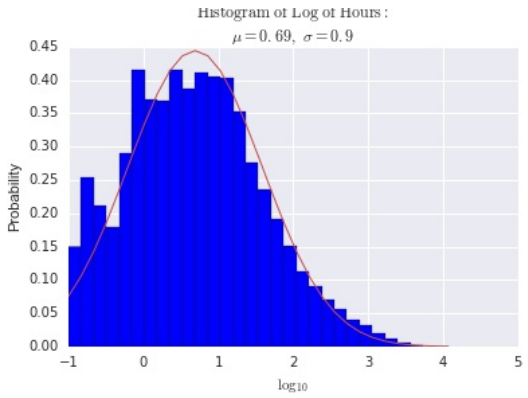

Fig. 1. Histogram of the $\log_{10}$ of playtime hours

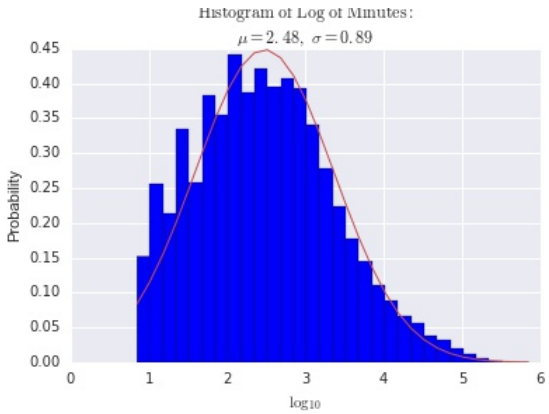

Fig. 2. Histogram of the $\log_{10}$ of playtime minutes plus one

5. MODELS

Before we begin describing the models, we'll first describe the structure of the code we used to hold and build models around the data. To store the data we used nested python dictionaries to hold the ratings with the first level of keys being the user IDs and the values being dictionaries that then hold the users games as keys and "rating" as the values. We chose this structure because it made transposing the user item matrix and making implicit zeros in the matrix easier during training (the pop dictionary method). For a more in depth explanation of the code we used to make the following models, we have provided comments in the scripts on our github

5.1. Baseline

5.1.1. Model Overview

Reading online we found an article called Collaborative Filtering Recommender Systems[1] which described the following baseline algorithm for recommender systems:

$$b_{u,i} = \mu + b_u + b_i$$

$$b_u = \frac{1}{|I_u| + \beta_u} \sum_{i \in I_u} (r_{u,i} - \mu)$$

$$b_i = \frac{1}{|U_i| + \beta_i} \sum_{u \in U_i} (r_{u,i} - b_u - \mu)$$

Where μ is average of all ratings in the matrix, β_u, β_i are regularization terms, and $b_{u,i}$ is the baseline prediction for user u's rating of item i.

5.1.2. Tuning and Performance

For our purposes we have set the regularization for both items and users equal so that there is only one parameter to tune for the performance of this baseline model. From Figure 3 we came to conclusion that the best regularization parameter would be around 5.

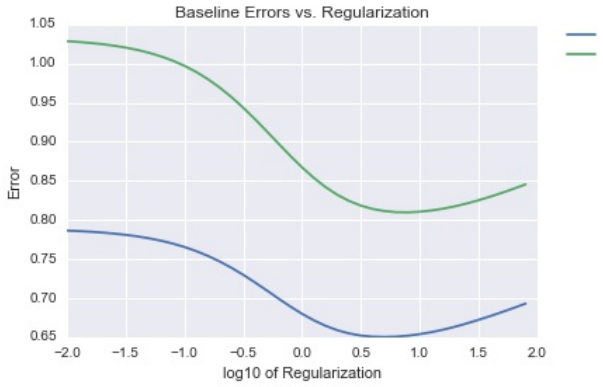

Fig. 3. Baseline Errors versus Regularization

5.2. User Based Collaborative Filter

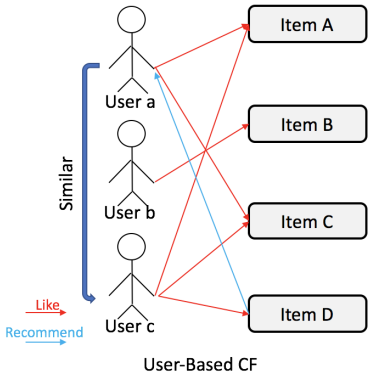

Fig. 4. User Based Collaborative Filter

5.2.1. *Model Overview*

User based collaborative filters attempt to predict the user will give an item by computing a weighted average of the ratings of that item given by users that are found to be similar. The predictions can be formulated as:

$$p_{u,i} = \bar{r}_u + \frac{\sum_{u' \in N} s\left(u,\ u'\right)\left(r_{u',i} - \bar{r}_u\right)}{\sum_{u' \in N} \left|s\left(u,\ u'\right)\right|}$$

Where N is the set of users that are similar to user u and have rated item i. Also s is the similarity function which for this model we used the cosine similarity function:

$$s\left(u, v\right) = \frac{\sum_i r_{u,i} r_{v,i}}{\sqrt{\sum_i r_{u,i}^2}\sqrt{\sum_i r_{v,i}^2}}$$

Here we found in the reference paper[1] that there are two kind of user based systems, in some systems like original GroupLens, all the users are considered as neighbors; in other systems, the neighbors are chosen from all the users who have rated the item that we are trying make a prediction. For example, if we want to know whether we should recommend item (game) M to user A, first, we need to find the ratings for item M from other users. Then, in those users, we are supposed to choose the top k similar users to user A. And using their ratings on M, we predict the rating of user A for item M.

And in our project, we choose to use the second method to build our user based system, since our data set is too sparse. If we use the first idea, we have two options to deal with the absent rating from top k similar users for item M. One is to ignore them, which is the same thought as second method. The other is to assume the absent ratings are 0, which will make the recommend probability pretty low, even negative predictions. It may cause the error become really high. In the Figure 5, we compared the two method versus different k. Method 1 is using all the users (except itself) as valid neighbors. Method 2 is using all the users who share the same object item with the object user as valid neighbors, which is what we choose in our project.

5.2.2. *Tuning*

For user based collaborative filters the only parameter that needs tuning is k which is the number similar users. From Figure 6 we came to the conclusion that around 1000 neighbors to yield the best results.

5.3. Item Based Collaborative Filter

5.3.1. *Model Overview*

Item based collaborative filter are very similar to user based collaborative filters in the sense that they both require you to calculate weighted means based on a similarity score among group of neighbors. Similar to user based collaborative filters

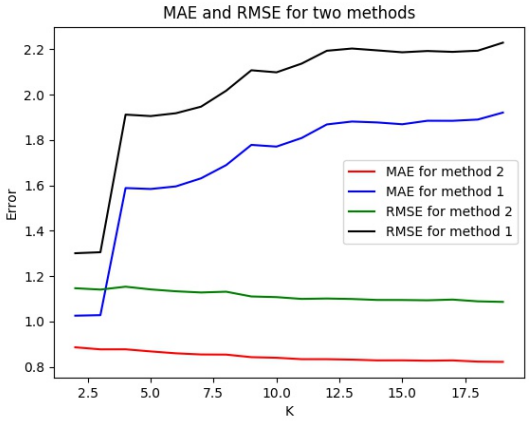

Fig. 5. Errors for Two Neighborhood Methods

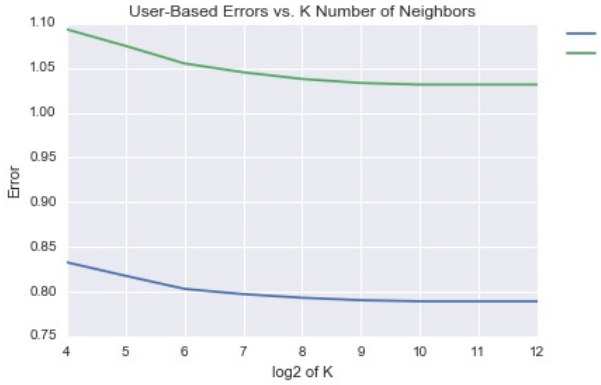

Fig. 6. User Based Errors versus K Number of Neighbors

this model requires that for a given user-item pair that we find the group of items that have been rated by the user in the user-item pair. To make use of fact that user may rate things differently we use the following similarity function we read in a paper form the GroupLens Research Group[2]:

$$s\left(i, j\right) = \frac{\sum_{u \in U}\left(r_{u,i} - \bar{r}_u\right)\left(r_{u,j} - \bar{r}_u\right)}{\sqrt{\sum_{u \in U}\left(r_{u,i} - \bar{r}_u\right)^2}\sqrt{\sum_{u \in U}\left(r_{u,j} - \bar{r}_u\right)^2}}$$

The only problem with using this similarity function is that it is possible to get negative similarity scores but fortunately the article Collaborative Filtering Recommender Systems[1] describes the following prediction formulation that involves using the baseline predictions:

$$p_{u,i} = b_{u,i} + \frac{\sum_{j \in S} s\left(i,\ j\right)\left(r_{u,j} - b_{u,i}\right)}{\sum_{j \in S} \left|s\left(i,\ j\right)\right|}$$

Here S is the set of items j that have been rate by user u and similar to item i.

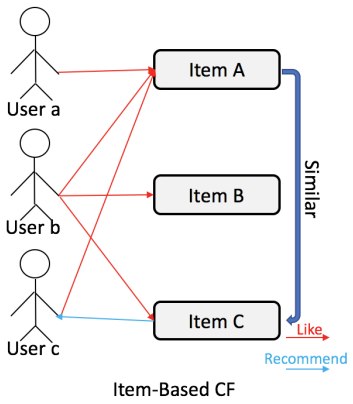

Fig. 7. Item Based Collaborative Filter

The reason that we used this similarity function and prediction with the baseline is because it performs better than using the regular cosine similarity function and the following prediction function:

$$p_{u,i} = \frac{\sum_{j \in S} s\,(i,\ j)\,(r_{u,j})}{\sum_{j \in S} |s\,(i,\ j)|}$$

This difference in performance can be seen in Figure 8

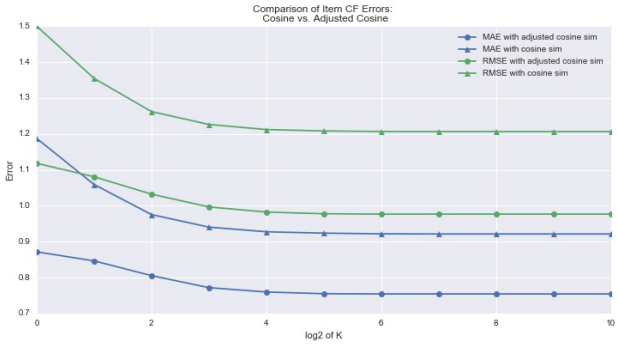

Fig. 8. Comparison of using cosine function versus adjusted cosine function

5.3.2. Tuning

Now unlike the user based filter our item based filter uses the baseline model to help make its predictions so the parameters that we have to tune are k, the number of neighboring items, and the regularization parameter for the baseline model. From Figures 9 and 10 we found that using around 500 neighbors and a regularization parameter of about 1 to get the best predictions.

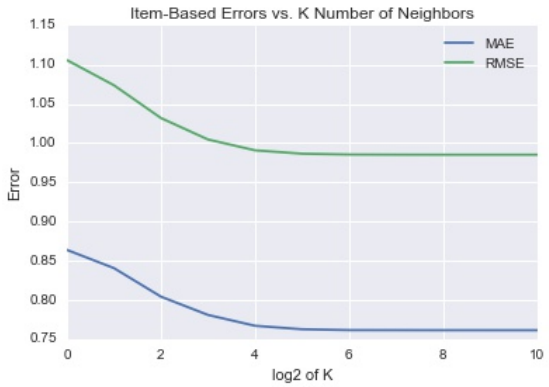

Fig. 9. Item Based Errors versus K Number of Neighbors

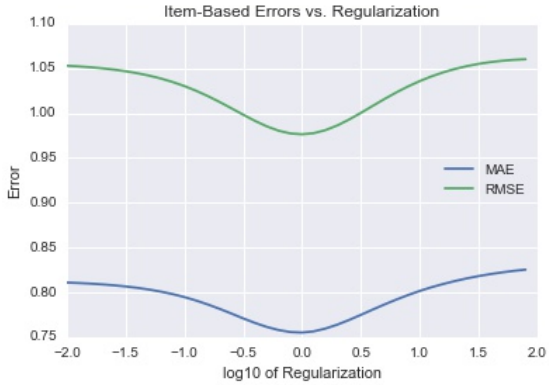

Fig. 10. Item Based Errors versus Regularization

5.4. Matrix Factorization: Alternating Least Squares

5.4.1. Model Overview

The idea behind matrix factorization is construct two matrices of the shape $m \times k$ and $k \times n$ where m and n are the sizes of the user base and item catalog and k is the number latent factors that we believe influences a user's rating of an item. And the product of these two matrices should create a matrix that can be used to make predictions for all possible user-item pairs. The main trick of matrix factorization is not try and optimize the two factor matrices to exactly replicate the rating matrices but rather to get the predictions of ratings that were explicitly given. There are two main ways to optimize the two matrix factors the first of which is alternating least squares where we try and minimize the following objective function:

$$J = \sum_{u,i \in S} \left(r_{u,i} - h_{u,} \cdot g_{,i}\right)^2 + \beta \sum_{u} h_{u,}^2 + \beta \sum_{,i} g_i^2$$

Where $h_{u,}$ is the latent factor vector of user u and $g_{,i}$ is the latent factor vector of item i. From this one can see that the prediction for a single user-item pair $u,\ i$ is:

$$p_{u,i} = h_{u,}g_{,i}$$

The alternating least squares involves holding one of the matrix factors constant and solving the least squares problem for the other. We found the solutions to theses least square problems on this blog on matrix factorization methods[3]:

$$h_{u,} = r_{u,}G^T \left(G^T G + \beta I\right)^{-1}$$

$$g_{,i} = r_{,i}H \left(HH^T + \beta I\right)^{-1}$$

Where G is the user latent factor matrix (with shape $m \times k$), H is the item latent factor matrix (with shape $k \times n$), and I is the identity matrix of shape $k \times k$. These solutions are used as updating rules for the alternating least square rules in getting the optimal matrix factors.

5.4.2. Tuning

For the ALS method of matrix factorization we need to tune k number of latent factors and the regularization parameter. From Figures 11 12 we found that using about 10 latent factors and a regularization parameter of about 0.01 to yield the best results.

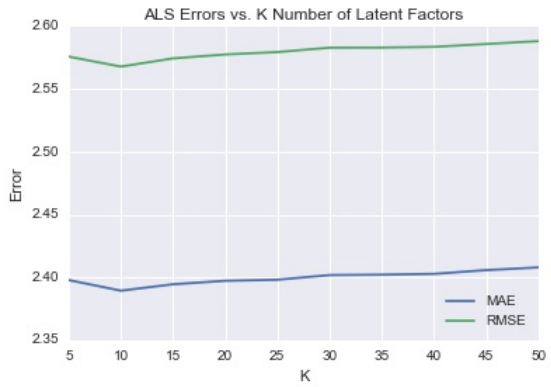

Fig. 11. ALS Errors versus K Number of Latent Factors

Fig. 12. ALS Errors versus Regularization

5.5. Matrix Factorization: Stochastic Gradient Descent

5.5.1. Model Overview

Like the ALS method from before SGD attempts to minimize the loss function described in the previous model overview. This time we get the following update rules for an user-item pair $u,\ i$:

$$h_{u,} = h_{u,} + \eta \left(\left(r_{u,i} - p_{u,i}\right) g_{,i} - \beta h_{u,}\right)$$

$$g_{,i} = g_{,i} + \eta \left(\left(r_{u,i} - p_{u,i}\right) h_{u,} - \beta g_{,i}\right)$$

5.5.2. Tuning

For the SGD method we have to perform parameter tuning on the number latent factors k and the regularization parameter β. From Figures 13 14 we found that using about 20 latent factors and a regularization parameter of about 0.1 to yield the best results.

Fig. 13. SGD Errors versus K Number of Latent Factors

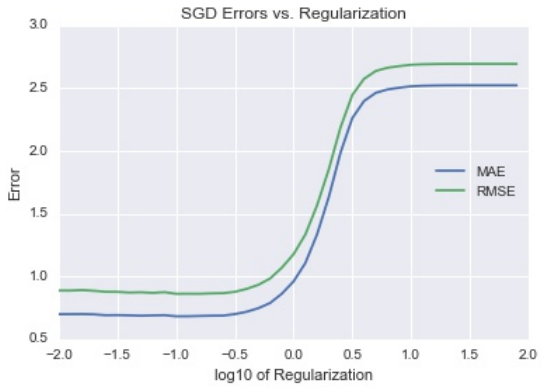

Fig. 14. SGD Errors versus Regularization

6. PERFORMANCE AND COMPARISON

Before we get into comparing the models let us take a quick look at how fast the two different matrix factorization methods converge to their solutions in Figure 15. There you can see that although having a higher training error the ALS method arrives to its solution much faster that the SGD method.

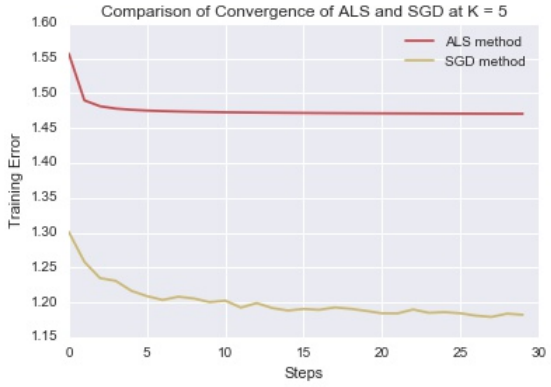

Fig. 15. Comparison between the Convergence of the different Matrix Factorization Methods

From Figure 16 we came to conclusion that the baseline model performed the best with the SGD method of Matrix Factorization being a close second

7. NEXT STEPS

From preprocessing we had to exclude some users from the overall data set before we did the training split because they had only one game. Matrix factorization has a method of introducing new users to into model without having to retrain the model. This method is referred to as "folding in" in the Collaborative Filtering Recommender Systems[1] survey:

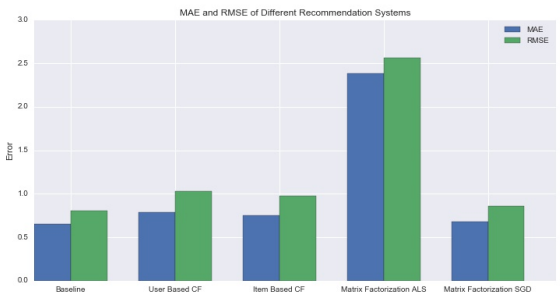

Fig. 16. Comparison between the Performance of the different Recommendation Systems

$$h_{u,} = G^T r_{u,}$$

Where $r_{u,}$ is the rating vector of the new user u and $h_{u,}$ is their latent factor vector. Since these users have only one game each folding them in to the user latent factor matrix should not have too much of an effect on the predictive power of the factor matrices.

If we were to continue and delve deeper into this topic of Recommendation Systems we could possibly look into the cold start problem that is prevalent in this topic. We could possibly improve the performance our recommender systems with the help other data sets such as user preferences or attributes about the games in the current item catalog.

We will also look into why the ALS method performed so terribly compared to the rest of the methods. Perhaps find some way to incorporate the baseline model into the prediction like we did for the Item-based Collaborative Filter.

8. REFERENCES

[1] Michael D Ekstrand, John T Riedl, Joseph A Konstan, et al., "Collaborative filtering recommender systems," *Foundations and Trends® in Human–Computer Interaction*, vol. 4, no. 2, pp. 81–173, 2011.

[2] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl, "Item-based collaborative filtering recommendation algorithms," in *Proceedings of the 10th international conference on World Wide Web*. ACM, 2001, pp. 285–295.

[3] "Explicit matrix factorization: Als, sgd, and all that jazz," `https://blog.insightdatascience.com/explicit-matrix-factorization-als-sgd-and-all-that-jazz-b00e4d9b21ea`, Accessed: 2016-03-16.