# Achieving Better Predictions with Collaborative Neighborhood

Edison Alejandro García, *garcial@stanford.edu*
Stanford Machine Learning - CS229

***Abstract*—Collaborative Filtering (*CF*) is a popular method that uses machine learning to predict interest overlap between users based on their behavior such as common ratings of items, or common purchasing and browsing patterns [1]. Having an accurate algorithm can mean the difference between better automated predictions or hours of manual post-processing data. We propose a method that improves the quality of collaborative filtering approaches up to a factor of four percent from our baselines. Because of its modular design our method can be tweaked for higher quality or faster prediction. This flexibility makes it a good choice in a wide variety of practical applications**

## I. Introduction

In today's Recommendation Systems (*RS*) a vast amount of predictions are being computed off and on-line every day. A prediction typically undergoes a number of pre and post-processing procedures before it is finally ready to provide meaningful suggestions to existing and potential customers. Thus *CF* applications are a central tool in retail and e-commerce everyday business.

One factor that limits the effectiveness of deriving interest overlap between users in *CF* is data *sparsity*: there are often many more items in the system than any user is able to rate [1]. Three main challenges a *RS* like Amazon[1] needs to solve [2], these are:

1) Producing high quality recommendations
2) Performing many recommendations per second for millions of customers and products
3) Achieving high coverage in the face of data sparsity

CF works by matching preferences or properties between two or more distinct subjects and makes recommendations based on the learned data. This approach has been shown to produce accurate results but at the cost of performance degradation and slower running time [4]. In this paper we present a method that uses Averaging (*Avg*), Latent Factors with Biases (*LFB*) and Singular Vector Decomposition (*SVD*), in order to improve new subject preferences predictions with good computational performance. The input is dataset MovieLens [3] for which we transform it into a sparse matrix $X \in \Re^{u \times i}$ where u is the number of users and i is the number of items. The output is a non sparse matrix $X' \in \Re^{u \times i}$ with the predicted ratings for all missing values in $X$.

[1]http://www.amazon.com

## II. Related Work

Collaborative filtering builds a model from a user's past behavior, activities, or preferences and makes recommendations to the user based upon similarities to other users [4]. There is lot of research in this area and one of our favorite sources was ”*A Netflix Case Study*” [5]. One of the top models used is Singular Value Decomposition. *SVD* analysis offers among others an interesting way to compare users and find nearest neighbors. It is suggested to compare users based on their feature weights because it may be more accurate. By only comparing feature weights and individual feature importance rather than individual document ratings, we can aggregate each document in the analysis [6]. Moreover, Berry et al. (1995) point out that the reduced orthogonal dimensions resulting from *SVD* are less noisy than the original data and capture the latent associations between the terms and documents [2]. *SVD* is one of the favorites models in *RS*. It was shown in our experiment that *SVD* was the overall best performer in this project.

Another well known algorithm is Nearest Neighbor (*NN*), it's underlying idea is that if a record falls in a particular neighborhood where a class label is predominant it is because the record is likely to belong to that very same class [4]. We implemented the method described in [6] and we were surprised by the fact that it was not part of the *Netflix Prize* winner solution. Furthermore, similar to *NN* we implemented *K-means* as discussed during the course. *K-means* has three drawbacks [4]:

1) Assumes prior knowledge of the data in order to choose the appropriate k
2) Final clusters are very sensitive to the selection of the initial centroids
3) It can produce empty cluster

In this project, point one was time consuming during the implementation phase in the project.

## III. Datasets and Features

The dataset of choice was MovieLens ”recommended for education and development” [3]. From the set, we used the ”ratings” matrix and built the data matrix with users in rows and items in columns. MovieLens has more data than we used in our project, like similarity between items, which are definitely useful for some state of the art Recommendation Systems.

The dimensions of the small dataset were: *100000* ratings with *9000* movies and *700* users. We have chosen the latter because it was built for *CF* problems and has a number of ratings with size that fits in our current system. Moreover, it allow us to:

1) Having more time to learn other *CF* solutions
2) faster iterations and learning over the data
3) faster code debugging cycles
4) Spending longer time in vectorizing the code so experiments run faster
5) Having more time to experiment with our pipeline

For experiment purposes we split the dataset into training (*50%*), validation (*20%*) and test (*30%*). Find in Appendix the code on how we split the dataset.

The biggest challenge with the dataset was the sparsity of the data matrix. The sparsity was *98.5%* for which it translates to *6.200.000* missing ratings. One of the main drawbacks of our training, validation and test dataset selection is that it increased the matrix sparsity because of the random selection.

## IV. Methods

*CF* has received considerable amount of attention in the scientific community. There are many models, algorithms and pipelines implemented that try to predict better user item ratings, improve runtime performance, transform from offline to online processing and so on. We propose a set of orthogonal improvements to the *Averaging* and *LFB* algorithms that can be combined at will. The strengths and weaknesses of individual algorithms are discussed and their contributions are experimentally evaluated in section V-C below. We further identify the most relevant combinations as the ones with the most favorable *RMSE* trade-off. Nevertheless we compare the different path starting from the *baselines* which are *Averaging* and *LFB*. Next we introduce the *baselines* and four main approaches that are build on top of our *baselines*.

### A. Baselines

They were our way to address the cold start. The algorithms that run on top of the *baselines* are not for initializing missing values, with the exception of *SVD*. We defined two approaches with the same underlying idea of averaging, in their own way, user-item ratings.

*1) Averaging:* This is the simplest of all models and algorithms in this project. We assign to all missing entries in the input matrix X the same rating, which is calculated as follows:

$$\forall(u,i) \in (X \in \Re^{m \times n}) \ where \ X(u,i) = nil, \ \exists \ X' = X \mid$$

$$X'(u,i) = \frac{\frac{\sum_{u=1}^{m} X(u,:)}{m} + \frac{\sum_{i=1}^{n} X(:,i)}{n}}{2} \tag{1}$$

Eq (1) says that for all users and items in X and the value of X at row *u* and column *i* is missing then there exists a copy X' of X such that the value of X' at row *u* and column *i* is the overall average in X.

*2) Latent Factors with Biases:* Matrix factorization models map both users and items to a joint latent factor space of dimensionality, such that user-item interactions are modeled as inner products in that space. The latent space tries to explain ratings by characterizing both products and users on factors automatically inferred from user feedback [4]. It is a very similar approach as *SVD* but our goal was to find P and Q such that it minimizes the Sum of Squared Errors (*SSE*) as in Eq (2).

$$min_{p,q} = \sum_{(u,i) \in R} (r_{ui} - q_i \cdot p_u)^2 \tag{2}$$

To obtain P and Q we computed the *SVD* of X where missing values in X were substituted with zero, and finally P = U and Q = S * V'. Despite of the fact that what we wanted was to minimized the Sum of Squared Errors (*SSE*) as shown in Eq (2), given P and Q, we took a different approach. Instead we assigned to all missing user ratings the following formula:

$$r_{ui} = \mu + b_u + b_i \tag{3}$$

Where $r_{ui}$ is the rating of user *u* for item *i*, $\mu$ is the item mean value over all users, $b_u$ is the bias value for user u and $b_i$ is the bias for item i.

### B. Singular Value Decomposition

One of the most applied matrix factorization techniques is *SVD*. The latter is a widely used technique to decompose a matrix into several component matrices, exposing many of the useful and interesting properties of the original matrix like *rank*, nullspace, orthogonal basis of column and row space.

$$X = U \cdot S \cdot V^{\top} \tag{4}$$

In an item recommendation system the resulting decomposition matrices from Eq(4) means:

- *U*: Users-to-concept affinity matrix
- *S*: The diagonal elements of S represent the 'expressiveness' of each concept in the data
- *V*: Items-to-concept similarity matrix

The *User-Item* ratings matrix can be imputed by choosing the *k* highest eigenvalues in *S*. By imputing we are deliberately selecting the items with highest concept expressiveness as well as significantly reducing *user-item* ratings matrix space. The biggest difficulty is how to choose the value of *k*. This can be done by an offline study of *S*, where we plotted the diagonal matrix curve values and look for the value at the *knee* of the curve. Furthermore, after making the choice of the highest *k* eigenvalues we can impute *U* and *V* by the *k* number of columns respectively.

In Eq(5) we present another implemented way to choose the value of *k* eigenvalues by imputing *S* using the energy of the diagonal matrix. The approach to decide which was the best energy value was the same as for choosing *k* in our latter approach.

$$K = i\ s.t. \left\{ \forall (r,c) \in S | r == c,\ (\sum_{i}^{m} S(r,c))/i \geq energy \right\} \tag{5}$$

As shown in Eq(6), by reconstructing back to *X'* we obtain a *user-item* ratings matrix solely representing the most significant items concepts. The *SVD* approach is significant because it addresses the need to present a more accurate and noiseless matrix.

$$X' = U_k \cdot S_k \cdot V_k{}^{\top} \tag{6}$$

## C. K-means

*K-Means* clustering is a partitioning method. The function partitions the dataset of N items into k disjoint subsets $S_j$ that contain $N_j$ items so that they are as close to each other as possible according a given distance measure. Each cluster in the partition is defined by its $N_j$ members and by its centroid $\lambda_j$ [4]. *K-means* is an iterative process to minimize the distortion function Eq (7).

$$J(c,u) = \sum_{i=1}^{m} \parallel x^{(i)} - \mu_{c^{(i)}} \parallel^2 \tag{7}$$

## D. Nearest Neighborhood (NN)

The main idea of this method is to use a concept vector that describes the affinity to different concepts and then find the most similar vectors in the search space to predict the missing values by averaging the ratings from all these neighbors. We implemented the method described in [6] :

For the concept vector we applied *SVD*, reduced the number of eigenvectors to k and then get the product

$$U_k \cdot S_k \tag{8}$$

to have the affinity of the users or items to concept weighted by the importance of that category.

To get the similarity between these vectors we look at the angle between the vectors by calculating the cosine distance doing a dot product between the vectors and normalize it:

$$cos\Theta = \frac{U \bullet V}{\|U\| \|V\|} \tag{9}$$

Fortunately to calculate these dot products between all vector tuples, we can efficiently do a matrix multiplication

$$(U_k \cdot S_k) \cdot (U_k \cdot S_k)^{\top} \tag{10}$$

and normalize it to get a symmetric matrix of all cosine distances between each two vectors. Then we sort the cosines in each row and keep track of the order of the positions. Now we can easily just pick the N vectors with the highest cosine distance to get the N nearest neighbors, so we can take the mean of the N neighbors for the missing rating. Intuitively this makes sense for users as well as items, because users with similar preferences will probably rate a item similar, or items having similar affinities to certain genres will be rated similar by users.

## E. Sparse Coding (SC)

One of the most successful implementations of *SC* was originated by Bergeaud et al in [7]. In short, it is a technique for representing signals by a linear combination of a small number of dimensions. Despite of the fact that sparse coding is mostly applied in the image processing area, it can be applied in the same way to *CF*. Furthermore in the item domain the dimensions are called atoms and they typically capture some basic local shape such users or items neighborhoods. The set of atoms used for sparse coding together forms a dictionary *U*. A matrix *X* is thus approximated by

$$X' = U \cdot Z \tag{11}$$

where $Z$ is a sparse assignment matrix.

The *User-Item* matrix is decomposed into small patches and each patch $x$ is reconstructed independently. Notably, given an over-complete dictionary, each patch can be represented in many ways. The biggest difficulty is thus to find a balance between sparsity and approximation error. Among the first solutions to this problem was Matching Pursuit [7] which relies on iterative greedy choices. Initializing with an atom assignment vector $z^0 = 0$ and a residual $r^0 = x$, the algorithm selects the atom $u_d^t$ that maximally reduces the current residual:

$$d^* = \arg\max_d |u_d^{\top} \cdot r^t| \tag{12}$$

The assignment vector and residual are subsequently updated as follows:

$$z_{d^*}^{t+1} = z_{d^*} + u_{d^*}^{\top} \cdot r^t \tag{13}$$

$$r^{t+1} = r^t - (u_{d^*}^{\top} \cdot r^t) \cdot u_{d^*} \tag{14}$$

When Matching Pursuit is applied to the *CF* problem, missing values are disregarded when initializing and recomputing the residual Eq(14). This way the algorithm tries to explain only the known parts of the *user-item* matrix. The reconstructed matrix is then used to fill the unknown user values. Researchers have proposed many improvements to the basic algorithm, one example being the iterative version [8], whereby the matrix is repeatedly coded and reconstructed until convergence.

### F. Goals

There are several criteria how to measure the quality of a Recommendation System.

1) Time to calculate the missing values (predictions),
2) Error between artificially removed ratings and their predicted counterparts
3) Measure how satisfied the user is with the predicted recommendations.

Point number three has some psychological aspect in it and could only be determined by a user study. Therefore we're just measuring the time in seconds and the *Root Mean Square Error* (*RMSE*) of the test dataset. Since our focus is not on effectively updating the matrix with changed rating or new users, but creating the initial prediction, we concentrate on having a balance between error and computation time. The goal is to learn what is the pipeline learning from data and achieve better predictions by not penalizing running time or expecting results in matter of hours.

## V. Experiments/Results/Discussion

### A. Experimental Setup

As first part, our constant experiments parameters are: Intel i7 4 cores and a single processor at 2.5 GHz with 16GB RAM, Mac OS and Matlab *R2016b*. The System Under Test (*SUT*) is defined in Figure 1, with the path that returned the lowest *RMSE*:

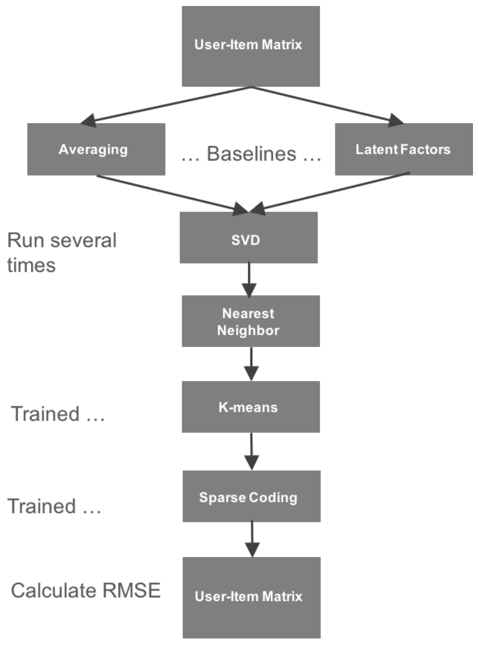

Figure 1. Prediction pipeline system.

The best pipeline was obtained by first performing a grid search on the primary factors and then creating a series of different combinations. The process was repeated every time an algorithm was run in a different order. As an example let's say we run the pipeline with *Averaging*, *SVD* and *NN*. First, we found the best primary factors for *SVD*. Then after running *Averaging* and *SVD* together we looked for the best configurations in *NN* to be applied after *Averaging* and *SVD*. Then, we switched *NN* and *SVD* places and did the same process again.

*1) Primary Factors:* These are summarized in Table I and are the best parameters values for the pipeline presented in Figure 1. Nevertheless the full list of *Primary Factors* and their possible *levels* can be found in the source code.

| Primary Factor | Description | Level |
|---|---|---|
| SVD_K | Number of chosen Eigenvalues | $k \leq .01$ |
| KM_C_U | Number of User clusters | 30 |
| KM_C_I | Number of Item clusters | 200 |
| NN_K_U | Eigenvalues Nearest Neighbors Users | 5 |
| NN_N_U | Number of Users Nearest Neighbors | 30 |
| NN_K_I | Eigenvalues Nearest Neighbors Items | 5 |
| NN_N_I | Number of Items Nearest Neighbors | 5 |
| SC_sigma | Stop correlation of residual and best atom | .5 |
| SC_sigma_min | Minimal correlation of residual | .01 |
| SC_neighbors | Patch sizes used in the matrix decom. | 16 |

Table I
Experimental parameter values

The primary factor for *KM*, *NN* and *SC* were learned and chosen in a cross validation grid search approach. For *SVD* it was done by performing a plot analysis, as described in section IV.

*2) Secondary Factors:* Those consists of hardware configuration. We have tested our optimal solution on better computers and we have seen an improvement on the final *running time*. The hardware configuration was definitely and an influencing factor when running the pipeline online.

### B. Quality Measures

When dealing with CF, there is no perfect solution to benchmark against. We therefore used the most objective measure, i.e. the error to the original matrix missing values. The *error* is thus computed as the *RMSE* of the computed missing values versus the original matrix missing values. The *time* was measured in seconds. However, standard deviation $\sigma$ for both error and time are not presented because we obtained the same *RMSE* value when replicated over the same experimental path. Despite of the fact that *Secondary Factors* do change our performance values, we are not interested in quantifying their results.

### C. Results

Figure 2 shows the results of applying different algorithm version of our method to *CF* with *98.5%* missing values. Our *baseline Averaging* approach corresponds to versions number 1 to 5 and *baseline LFB* from versions 6 to 10. We have chosen a set of ten versions to present from the whole set of possible algorithm and parameter combinations in our methods. The latter ten results showed clear changes in the pipeline performance. The y-axis marks the optimal front of pipeline performance with respect to *RMSE* and *Running Time* respectively with lower numbers meaning better results. The approaches were sorted with respect to the *baselines* and *RMSE* in descending order. We can clearly see how different combinations affect the *RMSE* and runtime performances.

| Version | Pipeline | *RMSE* | Time(sec) |
|---|---|---|---|
| 1 | A | 0.9459 | 0.360 |
| 2 | A + S | 0.9323 | 1.080 |
| 3 | A + xS | 0.9210 | 6.071 |
| 4 | A + xS + KM + xS | 0.9143 | 90.541 |
| 5 | A + xS + NN + KM + SC + xS | 0.9130 | 102.232 |
| 6 | L | 1.0140 | 18.059 |
| 7 | L + S | 1.0056 | 20.453 |
| 8 | L + xS | 0.9960 | 25.325 |
| 9 | L + xS + KM + xS | 0.9898 | 128.979 |
| 10 | L + xS + NN + KM + SC + xS | 0.9870 | 127.932 |

Table II
(A) Averaging; (L) Latent Factor with Biases; (S) SVD; (xS) x times SVD; (NN) Nearest Neighborhood; (SC) Sparse Coding; (KM) K-Means

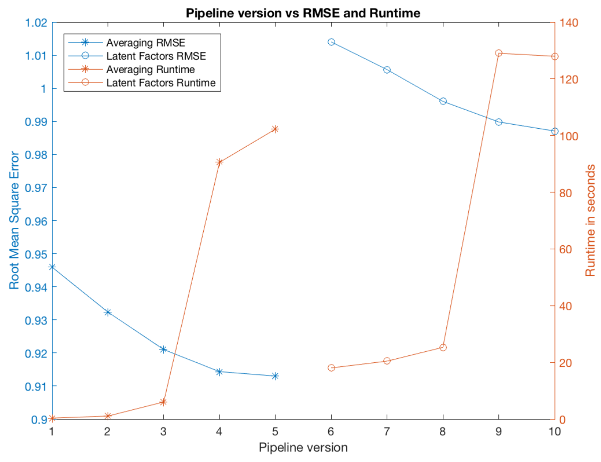

Figure 2. Pipeline version performance plot from table II

Improvement between the optimal and the baseline is about four percent *4%* and can be seen from Figure 2 and Table II. Version number 5 is the one with lowest *RMSE* and therefore with better predictions. The latter is composed by *Averaging* a series of *SVD*'s (user based), *NN*, *K-means*, *SC* and a final set of *SVD*'s (item based). It is worth noticing that the inclusion of *SVD* algorithm in any of the version always had a positive impact on *RMSE* with a low drop in running time performance. Furthermore, an important point from Figure 2 is that whenever the *K-Means* was introduced in any path, running time made a big jump up. Finally, we can see that *NN* and *SC* did not provide any improvement on top of any of the presented paths.

## VI. Conclusions

We have developed a novel approach to solve the *Collaborative Filtering* problem by extending two *baseline* algorithms. Our contribution includes developing a method for computing a *Nearest Neighborhood user-item* matrix reconstruction. Dynamically computing the cosine distances between users and items to later have a more accurate user-items recommendations based on neighbors centroid values. Based on all the research done for this project we can argue that the pipeline results are not optimal. However, we need to consider that we did not take into account many other data features state of the art solutions introduced in their systems to achieve better performance. Finally, our method allows for a flexible configuration of the graph to better match the demand of various practical applications based on recommendation predictions or time performance.

## Acknowledgements

The authors thank Michael for his time and project ideas and friends for their valuable project feedback.

## Appendix

Split Dataset into Training, Validation and Test sets

```
% get all indices in X where we have rating
rated_indexes = find(X ~= nil);
total_nil_indexes = numel(rated_indexes);
% number of training rows
number_train_rows = round(total_nil_indexes *
    prc_trn);
% number of validation rows
number_val_rows = round(total_nil_indexes *
    prc_trn);

% rand select training, validation and test rows
rp = randperm(total_nil_indexes);
idexes_train = rated_indexes(rp(1:
    number_train_rows));
idexes_val = rated_indexes(rp(number_train_rows+1:
    number_train_rows+number_val_rows));
idexes_test = rated_indexes(rp(number_train_rows+
    number_val_rows+1:end));

% Build training and testing matrices
[m,n] = size(X);
X_train = ones(m,n) * nil;
X_train(idexes_train) = X(idexes_train);  % add
    known training ratings
X_val = ones(m,n) * nil;
X_train(idexes_val) = X(idexes_val);  % add known
    validation ratings
X_test = ones(m,n) * nil;
X_test(idexes_test) = X(idexes_test);  % add known
     test ratings
```

## References

[1] S. Amer-Yahia, V. Markl, A. Halevy, A. Doan, G. Alonso, D. Kossmann, and G. Weikum, “Databases and web 2.0 panel at vldb 2007,” *SIGMOD Rec.*, vol. 37, pp. 49–52, March 2008. [Online]. Available: http://doi.acm.org/10.1145/1374780.1374794

[2] B. M. Sarwar, G. Karypis, J. A. Konstan, and J. T. Riedl, “Application of dimensionality reduction in recommender system – a case study,” in *IN ACM WEBKDD WORKSHOP*, 2000.

[3] F. M. Harper and J. A. Konstan, “The movielens datasets: History and context,” *ACM Trans. Interact. Intell. Syst.*, vol. 5, no. 4, pp. 19:1–19:19, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2827872

[4] B. S. Francesco Ricci, Lior Rokach and P. B. Kantor, *Recommender Systems Handbook*. USA: Springer US, 2011, vol. 1. [Online]. Available: http://www.springer.com/us/book/9780387858203

[5] X. Amatriain and J. Basilico, *Recommender Systems in Industry: A Netflix Case Study*, F. Ricci, L. Rokach, and B. Shapira, Eds. Boston, MA: Springer US, 2015. [Online]. Available: http://dx.doi.org/10.1007/978-1-4899-7637-6_11

[6] M. H. Pryor, “The Effects of Singular Value Decomposition on Collaborative Filtering,” Dartmouth College, Computer Science, Hanover, NH, Tech. Rep. PCS-TR98-338, June 1998. [Online]. Available: http://www.cs.dartmouth.edu/reports/TR98-338.ps.Z

[7] F. Bergeaud and S. Mallat, “Matching pursuit of images,” *Image Processing, International Conference on*, vol. 1, p. 53, 1995.

[8] M. jalal Fadili, J. luc Starck, and F. Murtagh, “Inpainting and zooming using sparse representations,” *The Computer Journal*, vol. 52, pp. 64–79, 2009.