United We Stand: Combinations for Collaborative Filtering

Mariyana Koleva
ETH Zurich
Switzerland
Email: kolevam@student.ethz.ch

Karim Labib
ETH Zurich
Switzerland
Email: labibk@student.ethz.ch

João Lourenço Ribeiro
ETH Zurich
Switzerland
Email: ljoao@student.ethz.ch

***Abstract*—Collaborative filtering consists of making predictions about some unknown preferences of a user based on available data from a set of users. There exist several philosophies for tackling this problem, and algorithms from each different philosophy pick up distinct properties from the data. In this paper, we study whether we can obtain better predictions by combining results from algorithms based on two different approaches: latent-factor and neighborhood-based models. We also study the effect of adding a learning rate heuristic to some algorithms.**

I. Introduction

Recommender systems have gained popularity in the last decade due to their immediate market applications. The huge selection of products allow online retailers to attract a wide-range of clients. However, it also means that users can no longer be expected to browse through everything available until they find what they like the most. Good recommendations are thus key for many leading online ventures, notably Netflix [1], and Amazon [2].

In this paper, we concentrate on the collaborative filtering approach to recommender systems, which relies on previous user experiences (ratings), rather than predefined user profiles. Our problem and data are largely based on the Netflix challenge [3], [4], [5].

Our main contributions are two-fold. We first combine latent factor models and neighbourhood models to produce a robust recommender system which optimizes the Root Mean Squared Error (RMSE) of 1176953 predicted ratings from **Base score** to **Best score without BD**. Secondly, we show that the addition of a more complex learning rate heuristic on two latent factor models improves the score significantly, to **best score**.

In Section II we describe the idea and the implementation of our systems. We present the results from the local evaluation of the models and from the validation set on Kaggle in Section III. Finally, in Section IV we draw conclusions from the analysis of the various methods and reported results.

II. Models and Methods

The dataset on which the work was based is a collection of *(user, movie, rating)* tuples. For training, we have 1176953 ratings available. The online evaluation on Kaggle is performed on another 1176953 ratings. We were able to see the prediction accuracy on 50% of the online dataset, and we aimed to improve it.

A. Data Model

The provided dataset can be seen as an $N \times M$ matrix, with N being the number of users ($N =$ 10000), and M the number of movies ($M = 1000$). If every user had rated every movie, the matrix would have 10^7 entries. However, in our case, the matrix is quite sparse. From here stems the difficulty of the problem – we attempt to predict more than a million ratings based on a representative subset.

There are two main views which yield different methods for tackling this problem.

Latent factor models: These models attempt to explain movie ratings by users as being influenced by a set of unknown (also called *latent*) factors.

One approach characterizes users and movies by low-dimensional feature vectors in a latent feature space. The missing rating of a given movie by a given user is modeled as the dot product between the corresponding user and movie feature vectors. As examples of this approach, we discuss Singular Value Decomposition (SVD) and Stochastic Gradient Descent (SGD).

It is also possible to predict missing ratings through neural networks which are composed by visible and latent units. We focus on Restricted Boltzmann Machines (RBM).

Neighborhood-based models: Another approach is to view the ratings as features of either the users (user-based) or the movies (item-based). In the user-based (item-based) perspective, we predict a missing rating of movie j by user i from the ratings of similar users (movies) to i (j), according to some distance function. We review implementations of both these ideas: user-based and item-based k-Nearest Neighbors (kNN).

B. Algorithms

We used three latent-factor and two neighborhood-based algorithms:

SVD: For this algorithm, we view the rating matrix as the product of a user matrix and a movie matrix, which are both on the latent factors' feature space. SVD is a natural method for performing this decomposition [4]. The main issue we face is that SVD works on full matrices only. Hence, we need to impute the missing values before we can proceed. If rating r_{ij} is missing, we impute the value $\lambda\text{AvgUser}[i] + (1-\lambda)\text{AvgFilm}[j]$, where $\lambda \in [0,1]$, and AvgUser$[i]$ and AvgFilm$[j]$ are the average ratings of user i and movie j, respectively.

We experimented with different values for λ and numbers of latent features. We concluded that the lowest RMSE on a probe set was accomplished using 13 features and $\lambda = 0.25$.

Nevertheless, we expect the results achieved with SVD to be suboptimal due to the large number of missing values, and the inaccuracy introduced by the imputation of those values. We use this algorithm as a baseline and we try to improve on it.

SGD: Instead of using the standard SVD, which requires a full matrix, we can use only the available ratings in a Stochastic Gradient Descent method. The regularized (to avoid over-fitting) objective function to minimize is as follows:

$$\min_{p,q,b_u,b_m} \sum_{(i,j)\in T} (r_{ij} - (b_u(i) + b_m(j) + p_i^T \cdot q_j))^2 + \lambda(\|p_i\|^2 + \|q_j\|^2 + b_u(i)^2 + b_m(j)^2). \tag{1}$$

T is the set of observed ratings (pairs (i,j) of user i rating movie j), r_{ij} is the known rating of user i to movie j, and b_u and b_m are the bias vectors of users and movies respectively. It has been pointed out in [4] that the bias vectors lead to higher accuracy. Intuitively, these biases help us as some user tend to consistently rate higher than others. In order to predict an unknown rating of user i to movie j we compute $r_{pred} = b_u(i) + b_m(j) + p_i^T \cdot q_j$. The parameters are modified during the algorithm by looping through all ratings in the training set for several epochs. The learning rules are

$$\begin{aligned} p_i &\leftarrow p_i + \gamma(e_{ij}q_j - \lambda p_i)\\ q_j &\leftarrow q_j + \gamma(e_{ij}p_i - \lambda q_j)\\ b_u(i) &\leftarrow \gamma(e_{ij} - \lambda b_u(i))\\ b_m(j) &\leftarrow \gamma(e_{ij} - \lambda b_m(j)). \end{aligned} \tag{2}$$

The algorithm was very sensitive to hyper-parameters, so those were optimized on both a local validation set and the online Kaggle score.

Restricted Boltzmann Machines (RBM's): RBM's are neural networks consisting of a layer of visible units and a layer of hidden units. Each unit has a stochastic binary random variable associated with it. There may be undirected connections between hidden and visible units, but no hidden-hidden or visible-visible connections are allowed. These networks are a restricted version of Boltzmann machines (hence the name), introduced in [6].

Recently, RBM's have found applications in collaborative filtering, as illustrated in [3]. A very nice overview of RBM's for collaborative filtering can be found in [7]. Our approach is based on these two works, although there are some differences.

We first build an RBM with 5000 visible units v_i^k, for $k \in \{1,\dots,5\}$ and $i \in \{1,\dots,1000\}$, and L hidden units $h_1,\dots,h_L$, where L is a hyper-parameter of the algorithm. Furthermore, for each hidden unit h_j we add connections to every visible unit v_i^k. Each connection between h_j and v_i^k has an associated weight W_{ij}^k. Each hidden unit h_j has an associated bias b_j and each visible unit v_i^k has an associated bias b_i^k. Finally, each unit can be in state 0 or 1.

The probability that a given unit is turned on (i.e. its associated state is 1) is a logistic function of the states of the units it is connected to.

To train our RBM, we divide the users into small batches. For each user u in a batch, we encode its ratings as follows: $v_i^k = 1$ if and only if user u gave rating k to movie i. After going through each batch, we update the weights and biases. We omit a detailed description of the training procedure and of the learning rules, which can be found in [7]. Some quantities needed for the updates are hard to compute exactly, but they can be approximated efficiently via Gibbs sampling (see [8]). In our algorithm we use a single step of Gibbs sampling. According to [7], this yields a good approximation already.

Note that we train only a single RBM. In [3], the method presented consists of building a different RBM for each user, which all share the same weights and biases. In particular, they keep only connections from hidden units to visible units corresponding to observed ratings of a particular user.

The prediction algorithm is the same in our method and in [3] and [7]. Namely, for a fixed user u, we set $v_i^k = 1$ if and only if user u ranked movie i with rating k. We then compute $\hat{p}_j = p(h_j = 1|V)$ for $j = 1,\dots,L$. If we want to predict the rating of movie i, we compute

$$p(v_i^k = 1|\hat{p}) = \frac{\exp(b_i^k + \sum_{j=1}^L \hat{p}_j W_{ij}^k)}{\sum_{l=1}^5 \exp(b_i^l + \sum_{j=1}^L \hat{p}_j W_{ij}^l)}, \tag{3}$$

for $k = 1,\dots,5$. This yields a probability distribution over the possible ratings for movie i. Our prediction is the expected value of this distribution, $\sum_{k=1}^5 k \cdot p(v_i^k = 1|\hat{p})$.

We make use of the built-in BernoulliRBM algorithm in the *sklearn* Python library to create our RBM and train it. Furthermore, we modified the source code to add a heuristic for the learning rate of the RBM, which improved its performance.

User-based k-Nearest Neighbors: User based neighbourhood algorithms consist of two steps. First, the algorithm computes the similarity between different users through some similarity measure. Second, to predict the rating of user u for a particular movie m, the ratings of the most similar k users who have rated movie m are aggregated to estimate the required rating. The similarity measure, which according to [7] was found to be the most accurate, was the Pearson correlation measure. We denote the Pearson correlation between two users u_1 and u_2 by $w(u_1, u_2)$ and it is calculated as follows:

$$\frac{\sum_{j \in C}(R(u_1, j) - \overline{R}_C(u_1)) \cdot (R(u_2, j) - \overline{R}_C(u_2))}{\sqrt{\sum_{j \in C}(R(u_1, j) - \overline{R}_C(u_1))^2}\sqrt{\sum_{j \in C}(R(u_2, j) - \overline{R}_C(u_2))^2}}, \tag{4}$$

where C is the set of co-rated items between users u_1 and u_2, $R(u, j)$ is the rating of user u to movie j, and $\overline{R}_C(u)$ is the average rating of user u on the set of movies C. The higher the correlation, the more similar the two users. This measure can be shown [7] to be equivalent to the cosine based similarity of the centered rating matrix (averages of users are subtracted from the corresponding ratings). This allows for easier and faster computation of correlation between every pair of users using *sklearn*. The best k was chosen by calculating the error of the algorithm on a validation set for different values of k. Finally, to predict the unknown rating of a user u for a particular movie i, we compute an averaged weighted sum of the ratings of the k nearest neighbors to user u as follows:

$$R(u, i) = \overline{R}(u) + \frac{\sum_{u_k \in N(u)} w(u, u_k)(R(u_k, i) - \overline{R}(u_k))}{\sum_{u_k \in N(u)} w(u, u_k)}, \tag{5}$$

where $N(u)$ is the set of the k most similar users who rated item i.

***Item-based k-Nearest Neighbors**:* For this algorithm, we consider neighborhoods of movies, instead of users. Computing the neighborhood of a movie and the predictions can be done in an analogous fashion to user-based kNN. Again, we make use of the Pearson correlation measure. The size of the neighborhood of a movie is a hyper-parameter that is optimally chosen using a local validation set.

C. The bold driver heuristic

A significant improvement to the running time and efficiency of our algorithm came from incorporating the bold driver heuristic for the learning rate in both SGD and RBM. This technique works as follows: Suppose we start with learning rate λ. After each epoch we compute the current error on the validation set and compare it with the error of the previous epoch. If the error increased from one epoch to the other, we decrease λ to $d\lambda$, where $d \in [0, 1]$. On the other hand, if the error decreased, we increase λ to $(1+c)\lambda$, where $c \geq 0$. Intuitively, the bold driver heuristic adapts the learning rate to the surface we are trying to maneuver. The values c and d are hyper-parameters.

III. Results

In this section we report the most interesting results achieved during our experiments, particularly for the SGD and RBM algorithms. Our goal was to understand how the different hyper-parameters affect the prediction accuracy, and what the best combination of algorithms is. We describe the validation protocol and results. In Section IV we derive conclusions.

A. Parameter optimization and validation

The general protocol for training and testing followed the steps below:

1) Choose a random subset of the train samples for a local validation set if not already chosen. This was in most cases 5% of the data, which amounts to 58847 samples. All algorithms trained on the same data.
2) Train each individual algorithm on the leftover training dataset. In some cases, such as in the item-based kNN, the validation set was used to automatically choose the optimal k in the predefined range. In the iterative algorithms, like SGD and RBM, we used the validation set as an early indication on whether the error is really falling with every epoch.
3) Each algorithm generates predictions for the local validation set and the test set obtained from Kaggle.
4) The predictions from all algorithms are combined using linear regression. The regressor is trained on the predictions generated for the local validation set. The coefficients which fit the results best are then used to generate a final prediction.

B. SGD Bold Driver and c parameter validation

The accuracy of the SGD algorithm for matrix factorization is strongly linked to the number of iterations of the algorithms. The parameter c defines the number of latent factors of the learned matrices in the SGD algorithm. Figures 1 and 2 show how the different setups affect the number of epochs required to achieve optimal errors, and their values

C. RBM Bold Driver

As the bold driver heuristic improved the convergence speed of SGD, we also implemented it for RBM and validated its effect. Figure 3 shows how number of epochs and accuracy are affected by the bold driver heuristic. To find the optimal hyper-parameters we used *grid search* on number of hidden units and learning rate, ranging between 20 and 101 and 0.03 and 0.1, respectively. The optimal error was achieved with learning rate 0.05 and 80 hidden units.

D. Comparison between baseline and optimal algorithms

In this section we report the RMSE error achieved on our local validation set, the public test set on Kaggle for each of the individual algorithms, and the blended result of the different algorithms.

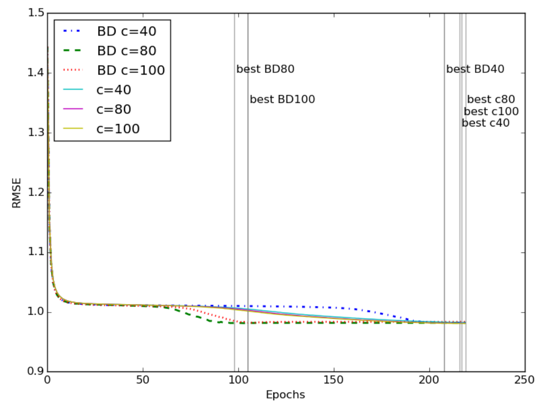

Figure 1. RMSE decreases with epochs for the SGD algorithm with varying c, with and without the bold driver heuristic. Optimal error 0.98066 was achieved after 217 epochs with C100, and 0.9811 was achieved after 98 epochs with the BD C80 setup. The vertical lines mark the epochs when the minimum error was achieved.

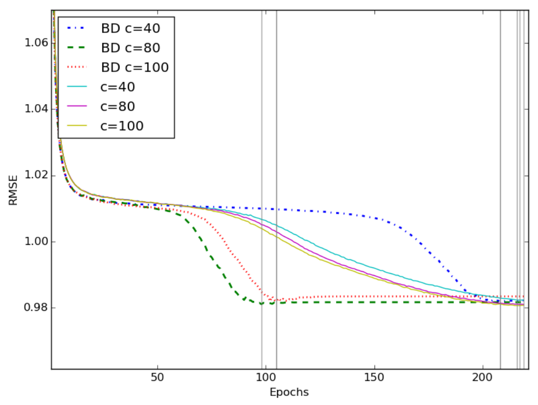

Figure 2. Zoomed-in version of Figure 1. Dashed graphs show results for algorithm with bold driver heuristic. Those converge much faster than the original algorithm at the cost of a 0.0004 error increase.

Algorithm	RMSE (validation)	RMSE (Kaggle Public Testset)
SVD (K=13)	1.00249	1.00082
SGD (c=80)	0.98151	0.98011
RBM (hidden=80)	0.98887	0.98814
kNN-item (k=50)	1.00260	0.99885
kNN-user (k=100)	1.00876	1.00569

We included all algorithms mentioned in the table when doing linear regression, along with some more variations. For example, we included RBM results obtained with 50 and 100 hidden states. We also included SGD results for $c = 100$. Performing linear regression on all predictions achieved a 10 fold cross validation error of 0.9420984 (Error achieved on each of the folds [0.9782558, 0.9193776, 0.94058739, 0.93060165, 0.9327, 0.98069694, 0.93532443, 0.91192913, 0.95785112, 0.93366013]). When submitting these predictions, we achieved a score of 0.96928, which outperforms each of the individual algorithms.

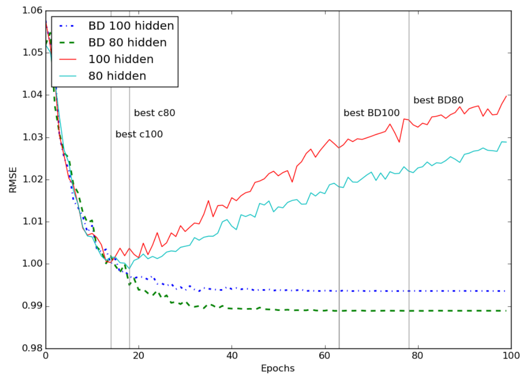

Figure 3. Effect of bold driver heuristic on RMSE error of RBM with 80 and 100 hidden units. The optimal error achieved without Bold Driver is 0.99883 with 100 units after 18 epochs, and 0.98885 with Bold Driver after 78 epochs.

IV. Conclusions

The first aim of our experiments was to investigate whether we could achieve better overall accuracy by combining different algorithms which are known to perform well on the collaborative filtering problem. The intuition behind this is that different approaches to collaborative filtering capture distinct properties of the data, and so they can “help” each other. The fact that the linear regression of predictions computed by our latent-factor and neighborhood-based algorithms outperforms all individual predictions confirms the validity of our idea.

Our second contribution was investigating the bold driver heuristic. From the results of Section III, we can easily see the positive impact of this heuristic. For SGD, adding the heuristic leads to a significant increase in convergence speed for all values of c considered, especially when $c = 80$ or 100 (convergence speed doubles), with very small associated error increase. The impact of the bold driver heuristic on RBM is even greater. Setups with the bold driver heuristic see their error continue to decrease for 50 or more additional epochs, achieving an improvement of 0.01 compared to the original algorithm. The difference between the behavior of the bold driver heuristic in SGD and RBM can be due to differences between the objective functions of both algorithms. It may be the case that the objective function of RBM is resistant to fixed learning rate methods, but is also well-behaved enough to be easily maneuvered by a slightly more complex method, like the bold driver.

References

[1] X. Amatriain and J. Basilico, “Netflix recommendations: beyond the 5 stars (part 1),” *Netflix Tech Blog*, vol. 6, 2012.

[2] G. Linden, B. Smith, and J. York, “Amazon. com recommendations: Item-to-item collaborative filtering,” *IEEE Internet computing*, vol. 7, no. 1, pp. 76–80, 2003.

[3] R. Salakhutdinov, A. Mnih, and G. Hinton, “Restricted boltzmann machines for collaborative filtering,” in *Proceedings of the 24th international conference on Machine learning*. ACM, 2007, pp. 791–798.

[4] Y. Koren, R. Bell, and C. Volinsky, “Matrix factorization techniques for recommender systems,” 2009.

[5] S. Funk, “Netflix update: Try this at home, 2006,” *URL http://sifter. org/˜ simon/journal/20061211. html*, 2011.

[6] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, “A learning algorithm for boltzmann machines,” *Cognitive science*, vol. 9, no. 1, pp. 147–169, 1985.

[7] G. Louppe, “Collaborative filtering: Scalable approaches using restricted boltzmann machines,” Master’s thesis, Université de Liège, Liège, Belgique, 2010.

[8] G. E. Hinton, “Training products of experts by minimizing contrastive divergence,” *Neural computation*, vol. 14, no. 8, pp. 1771–1800, 2002.