STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LV**, Number 4, 2010

IMPLEMENTATION OF A RECOMMENDER SYSTEM USING COLLABORATIVE FILTERING

PRODAN ANDREI-CRISTIAN

Abstract. Nowadays, consumers have a lot of choices. Electronic retailers offer a great variety of products. Because of this, there is a need for Recommender Systems. These systems aim to solve the problem of matching consumers with the most appealing products for them. They do this by analyzing either the products information details (Content Based methods) or users social behavior (Collaborative Filtering). This paper describes the Collaborative Filtering technique in more detail. It then presents one of the best methods for CF: the Matrix Factorization technique. Next, it presents two algorithms used for matrix factorization. Then, the paper describes the implementation details of a framework created by us, called *Rho*, that uses Collaborative Filtering. In the end, we present some results obtained after experimenting with this framework.

1. Introduction

This paper is organized in two parts. The first part (sections 1 and 2) presents some theoretical aspects about *Recommender Systems*. It describes what they are, who uses them, a couple of examples and some mathematical background. In terms of building a user profile, the paper describes the Collaborative Filtering technique.

In the second part (section 3), the paper describes an implementation of a framework (named *Rho* and implemented by us) which can be used to model a Recommender System based on Collaborative Filtering techniques. Using the algorithms described in section 2.1, the presentation continues with the implementation details. Finally, in section 4, we comment on some results obtained after experimenting with this framework. In the end, section 5 we state our conclusions along with some ideas on how to further expand the capabilities of such systems.

Received by the editors: August 30, 2010.

2010 *Mathematics Subject Classification.* 68P20, 15A18, 15A23.

1998 *CR Categories and Descriptors.* H.3.3 [**Information Systems**]: Information storage and retrieval – *Information search and retrieval*; G.1.2 [**Mathematics of Computing**]: Numerical Analysis – *Approximation*.

Key words and phrases. collaborative filtering, Recommender Systems, svd.

We continue with this section by giving an overview of what Recommender Systems are, a general view on how do they work and some examples, used by several big companies.

Nowadays, consumers have a lot of choices. Electronic retailers offer a great variety of products. Because of this, there is a need for Recommender Systems. These systems aim to solve the problem of matching consumers with the most appropriate products.

Recommender Systems can be used on products such as books, movies, music, restaurants and TV shows. Many customers will view the same movie or purchase the same item. Every item has a couple of characteristics like genre or subject, and users can express their preferences (like of dislike) regarding them. Also, customers give feedback on products, indicating how much they liked it, so data about which product appeals to which customer, is available. Companies can analyze this data and recommend products to their customers.

Essentially, Recommender Systems compare the user's profile to some reference characteristics, and try to predict the "rating" that a user would give to an item they had not yet considered.

There are mainly two forms of data collection needed for building a users profile:

(1) **Explicit data**: ask a user to rate an item on a scale, present two items to a user and asking him/her to choose the best one, ask a user to create a list of items that he/she likes, ask a user to rank a collection of items from favorite to least favorite;
(2) **Implicit data**: observing the items that a user views in an online store, analyze item/user viewing times, keep a record of the items that a user purchases online, obtaining a list of items that a user has listened to or watched on his/her computer, analyzing the user's social network and discovering similar likes and dislikes.

The Recommender System compares the collected data to similar and non similar data collected from others and calculates a list of recommended items for the user.

The following issues represent the main challenges that must be considered when implementing a Recommender System.

- **The cold-start problem**: Recommender Systems must be capable of matching the characteristics of an item against relevant features in the user's profile. In order to do this, it must first construct a sufficiently-detailed model of the user's tastes and preferences. The cold start problem implies that the user has to dedicate an amount of effort to contribute to the construction of their user profile before the system can start providing any intelligent recommendations.

- **Sparsity**: In any Recommender System, the number of ratings already obtained is usually very small compared to the number of ratings that need to be predicted. Effective prediction of ratings from a small number of examples is important. Also, the success of the collaborative Recommender System depends on the availability of a critical mass of users.
- **Scalability**: Recommender Systems are usually designed to work on very large data sets. Therefore the scalability of the methods employed by them systems is crucial.

It is worth mentioning that big companies like Amazon, Google, Yahoo, Netflix, last.fm make use of Recommender Systems.

2. Designing a Recommender System using Collaborative Filtering

Collaborative filtering is a term coined by the developers of Tapestry, the first Recommender System ([1]). The underlying assumption of CF approach is that those who agreed in the past tend to agree again in the future. For example, a collaborative filtering or recommendation system for music tastes could make predictions about which music a user should like given a partial list of that user's tastes (likes or dislikes). These predictions are specific to the user, but use information gathered from many users.

The problem of collaborative filtering (CF) is defined in [2], as follows. The problem can be modeled by the random triplet (U, I, R), where:

- U taking values from $\{1, ..., N\}$ is the user identifier (N is the number of users),
- I taking values from $\{1, ..., M\}$ is the item identifier (M is the number of items), and
- R taking values from $X \subset \mathbb{R}$ is the rating value. Typical rating values can be binary ($X = \{0, 1\}$), integers from a given range (for example, $X = \{1, 2, 3, 4, 5\}$), or real numbers of a closed interval (for example, $X = [10, 10]$).

A realization of (U, I, R) denoted by (u, i, r) means that user u rated item i with value r. The goal is to estimate R from (U, I) such that the *root mean squared error* of the estimate,

$$RMSE = \sqrt{E\{(\hat{R} - R)^2\}} \tag{1}$$

is minimal, where $\hat{R}$ is the square estimate of R and E denotes the mean.

In practice, the distribution of (U, I, R) is not known: we are only given a finite sample, $\mathcal{T}' = \{(u_1, i_1, r_1), (u_2, i_2, r_2), ..., (u_t, i_t, r_t)\}$, generated by it.

The sample $\mathcal{T}'$ can be used for training predictors. We assume sampling without replacement in the sense that $(userID, itemID)$ pairs are unique in the sample, which means that users do not rate items more than once. Let us introduce the notation $\mathcal{T} = \{(u,i) : \exists r : (u,i,r) \in \mathcal{T}'\}$ for the set of $(userID, itemID)$ pairs. Note that $|\mathcal{T}'| = |T|$, and typically $|\mathcal{T}| \ll N \cdot M$, because most of the users rate only a small subset of the entire set of items. The sample can be represented as a partially specified matrix denoted by $\mathbf{R} \in \mathbb{R}^{N \times M}$, where the matrix elements are known in positions $(u,i) \in \mathcal{T}$, and unknown in positions $(u,i) \notin T$. The value of the matrix $\mathbf{R}$ at position $(u,i) \in \mathcal{T}$, denoted by r_{ui}, stores the rating of user u for item i. For clarity, we use the term (u,i)-th rating in general for $r_u i$, and (u,i)-th training example if $r_{ui} : (u,i) \in \mathcal{T}$.

The goal of this CF setup is to create such predictors that aim at minimizing the error (1). In practice, we cannot measure the error because the distribution of (U, I, R) is unknown, but we can estimate the error on a validation set. Let us denote the validation set by $\mathcal{V}' \subset [1, ..., N] \times [1, ..., M] \times X$, assuming sampling without replacement as defined above, and we further assume the uniqueness of $(userID, itemID)$ pairs across $\mathcal{T}'$ and $\mathcal{V}'$. We define $\mathcal{V} = \{(u,i) : \exists r : (u,i,r) \in \mathcal{V}'\}$. The assumptions ensure that $\mathcal{T} \cap \mathcal{V} = \emptyset$. If both the training set $\mathcal{T}$ and validation set $\mathcal{V}'$ are generated from the same distribution the estimate of RMSE can be calculated as:

$$\hat{RMSE} = \sqrt{\frac{1}{|\mathcal{V}|} \sum_{(u,i) \in \mathcal{V}} (\hat{r_{ui}} - r_{ui})^2} \tag{2}$$

2.1. **Matrix Factorization for Collaborative Filtering.** The idea behind Matrix Factorization (MF) is quite simple. We want to approximate matrix R (the ratings matrix) as the product of two matrices:

$$\mathbf{R} \approx \mathbf{PQ}, \tag{3}$$

where $\mathbf{P}$ is an $N \times K$ matrix and $\mathbf{Q}$ is a $K \times M$ matrix. We call P the user feature matrix and Q the item feature matrix. K is the number of features in the given factorization. $\mathbf{Q}$ and $\mathbf{P}$ typically contain real numbers, even when R contains only integers.

One way to do this is to use a techniques called *Singular Value Decomposition* (SVD). This technique is a matrix factorization technique commonly used for producing *low-rank* approximations of the initial matrix. Given an $m \times n$ matrix A, with rank r, the singular value decomposition, $SVD(A)$, is defined as:

$$SVD(A) = U \times S \times V^T, \tag{4}$$

where, where U is an $m \times m$ unitary matrix over $\mathbb{R}$, the matrix S is an $m \times n$ diagonal matrix with nonnegative real numbers on the diagonal, and V^T, an $n \times n$ unitary matrix over $\mathbb{R}$, denotes the conjugate transpose of V.

Getting back to collaborative filtering, the task is to factorize the R (rating matrix) according to SVD. Once the $m \times n$ ratings matrix R is decomposed and reduced into three SVD component matrices with k features U_k, S_k, and V_k, prediction can be generated from it by computing the cosine similarities (dot products) between m pseudo-customers $U_k \cdot \sqrt{S_k}^T$ and n pseudo-products $\sqrt{S_k} \cdot {V_k}^T$. In particular, the prediction score $P_{i,j}$ for the i-th customer on the j-th product by adding the row average r_i to the similarity. Formally,

$$P_{i,j} = r + U_k \cdot \sqrt{S_k}^T(i) \cdot \sqrt{S_k} \cdot {V_k}^T \tag{5}$$

Once the SVD decomposition is done, the prediction generation process involves only a dot product computation, which takes $O(1)$ time, since k is a constant.

However, this is unfeasible for very big and sparse matrices. An alternative to this is proposed by [2] and presented next.

2.2. **Background on the ISMF and RISMF algorithms.** In this section we give an overview of the theoretical aspects of two recommendation algorithms used in the framework implemented by us and described in section 3. These algorithms (denoted ISMF and RISMF [2]) use the following matrix factorization technique.

The notations are the same used in section 2. Let p_{uk} denote the elements of $\mathbf{P} \in \mathbb{R}^{N \times K}$, and q_{ki} the elements of $Q \in \mathbb{R}^{K \times M}$. Further, let $\mathbf{p_u}$, denote a row (vector) of $\mathbf{P}$, and $\mathbf{q_i}$, a column (vector) of $\mathbf{Q}$. Then:

$$\hat{r_{ui}} = \sum_{k=1}^{K} p_{uk} q_{ki} = \mathbf{p_u q_i}, \tag{6}$$

$$e_{ui} = r_{ui} - \hat{r_{ui}}, (u,i) = r_{ui} - \mathbf{p_u q_i} \in (T),$$

$${e_{ui}}' = \frac{1}{2} {e_{ui}}^2, \tag{7}$$

$$SSE = \sum_{(u,i)\in\mathcal{T}} {e_{ui}}^2 = \sum_{(u,i)\in\mathcal{T}} \left(r_{ui} - \sum_{k=1}^{K} p_{uk} q_{ki} \right)^2$$

$$SSE' = \frac{1}{2} SSE = \sum_{(u,i)\in\mathcal{T}} {e_{ui}}',$$

$$RMSE = \sqrt{\frac{SSE}{|\mathcal{T}|}},$$

$$(\mathbf{P}*, \mathbf{Q}*) = \arg\min_{\mathbf{(P*,Q*)}} SSE' = \arg\min_{\mathbf{(P*,Q*)}} SSE = \arg\min_{\mathbf{(P*,Q*)}} RMSE \tag{8}$$

Here:

- $\hat{r_{ui}}$ denotes how the u-th user would rate the i-th item, according to the model;
- e_{ui} denotes the training error measured at the (u,i)-th rating;
- SSE denotes the sum of squared training errors.

Equation 8 states that the optimal $\mathbf{P}$ and $\mathbf{Q}$ minimize the sum of squared errors only on the known elements of $\mathbf{R}$.

In order to minimize $RMSE$, which is in this case equivalent to minimizing SSE', we apply a simple incremental gradient descent method to find a local minimum of SSE', where one gradient step intends to decrease the square of prediction error of only one rating, or equivalently, either ${e_{ui}}'$ or ${e_{ui}}^2$.

For the incremental gradient descent method, suppose we are at the (u,i)-th training example, r_{ui}, and its approximation $\hat{r_{ui}}$ is given.

We compute the gradient of ${e_{ui}}'$ and we obtain:

$$\nabla {e_{ui}}' = \left(\frac{\partial {e_{ui}}'}{\partial \mathbf{p_u}}, \frac{\partial {e_{ui}}'}{\partial \mathbf{q_i}} \right) \tag{9}$$

$$\frac{\partial {e_{ui}}'}{\partial \mathbf{p_{uk}}} = -e_{ui} \cdot \mathbf{q_{ki}} \tag{10}$$

$$\frac{\partial {e_{ui}}'}{\partial \mathbf{q_{ki}}} = -e_{ui} \cdot \mathbf{p_{uk}} \tag{11}$$

We update the weights in the direction opposite to the gradient:

$$p_{uk} \leftarrow p_{uk} + \gamma \cdot e_{ui} \cdot q_{ki} \tag{12}$$

$$q_{ki} \leftarrow q_{ki} + \gamma \cdot e_{ui} \cdot p_{uk} \tag{13}$$

That is, we change the weights in $\mathbf{P}$ and $\mathbf{Q}$ to decrease the square of actual error, thus better approximating r_{ui}. Here γ is the learning rate.

When the training has been finished, each value of $\mathbf{R}$ can be computed easily using Eq. 6 , even at unknown positions. In other words, the model ($\mathbf{P*}$, $\mathbf{Q*}$) provides a description of how an arbitrary user would rate any item.

This method is called **ISMF**, that is incremental simultaneous MF, according to [2] due to its distinctive incremental and simultaneous weight updating to other MF methods.

2.3. **Improving the ISMF algorithm.** The matrix factorization presented in the previous section can overfit for users with few (no more than K) ratings: assuming that the feature vectors of the items rated by the user are linearly independent and $\mathbf{Q}$ does not change, there exists a user feature vector with zero training error. Thus, there is a potential for overfitting, if γ and the extent of the change in $\mathbf{Q}$ are both small. A common way to avoid overfitting is to apply regularization by penalizing the square of the Euclidean norm of weights. Penalizing the weights results in a new optimization problem:

$$\begin{aligned} {e_{ui}}' &= \frac{{e_{ui}}^2 + \lambda \cdot \mathbf{p_u} \cdot {\mathbf{p_u}}^T + \lambda \cdot \mathbf{q_i}^T \cdot \mathbf{q_i}}{2}, \\ SSE' &= \sum_{(u,i)\in\mathcal{T}} {e_{ui}}', \\ (\mathbf{P}*, \mathbf{Q}*) &= \arg\min_{(\mathbf{P},\mathbf{Q})} SSE'. \end{aligned} \tag{14}$$

Here $\lambda \geq 0$ is the regularization factor. Note that minimizing SSE'' is no longer equivalent to minimizing SSE, unless $\lambda = 0$, in which case we get back to the **ISMF**. The authors call this MF variant RISMF, that stands for regularized incremental simultaneous MF.

Similar to the ISMF approach, we compute the gradient of ${e_{ui}}'$:

$$\begin{aligned} \frac{\partial {e_{ui}}'}{\partial \mathbf{p_{uk}}} &= -e_{ui} \cdot \mathbf{q_{ki}} + \lambda * p_{uk}, \\ \frac{\partial {e_{ui}}'}{\partial \mathbf{q_{ki}}} &= -e_{ui} \cdot \mathbf{p_{uk}} + \lambda * p_{ki} \end{aligned} \tag{15}$$

We update the weights in the direction opposite to the gradient:

$$p_{uk} \leftarrow p_{uk} + \gamma \cdot (e_{ui} \cdot q_{ki} - \lambda * p_{uk}) \tag{16}$$

$$q_{ki} \leftarrow q_{ki} + \gamma \cdot (e_{ui} \cdot p_{uk} - \lambda * p_{ki}) \tag{17}$$

The training algorithm is for training the data can be found in [2]:

3. The Rho framework

This section presents a small framework implemented by us, that uses the algorithms mentioned in the previous chapter and which he called *Rho*. It can be used to train a model, analyze the results and provide recommendations for a user. Starting with the overall architecture, in which the main components of the software are presented, we then show the parameters supported by the framework on each of the four components and how to use each component.

The purpose of *Rho* is to provide a framework for Recommender Systems research, having a couple of tools for training, analyzing the results and making recommendations. It is formed of four components: **Trainer**, **BatchRunner**, **Analyzer**, **Recommender**. A diagram showing the interaction (inputs and outputs) between the components of the system is presented in Figure 1.

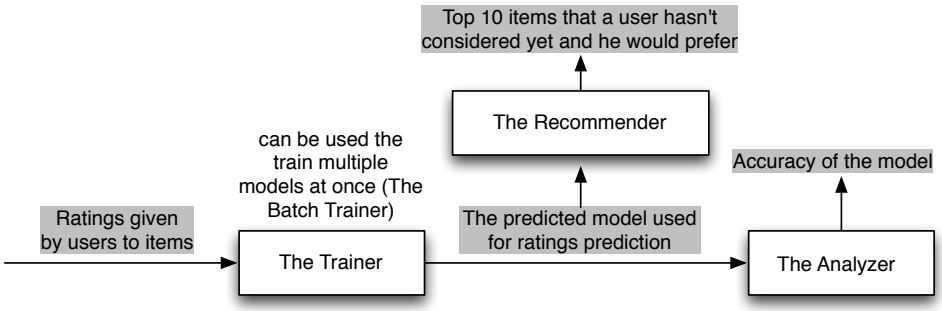

Figure 1. The components in the Rho framework

Next, we will analyze each component and explain its functionality.

3.1. **Training the model with the Trainer.** The main purpose of Rho is to allow making recommendations using matrix factorization techniques. Since it's unfeasible to factorize big sparse matrixes, the proposed algorithms uses some machine learning techniques. Discovering the model using machine learning assumes two phases: 1) train the model on a training dataset, 2) test the model on a test dataset, with the information gathered during training. As its name suggests, the Trainer component is used to train the model, on a

training dataset. The script located in `trainer.py` runs the effective training algorithm. The parameters can be changed in that file.

For now, Rho supports the 2 algorithms that were described earlier in sections 2.2 and 2.3 respectively. The parameters used to tune the algorithm are written in a Python hash format, which is easy to understand and follow. They are the following: *algorithm type* (ISMF or RISMF), *minimum improvement required to continue current feature*, *learning rate*, *regularization factor*, *number of features(factors)* to use, *initialization value for features*, *max epochs per feature*, *minimum number of epochs*, *number of items in entire training set*, *number of users in entire training set*, *number of ratings in entire training set*, *path to training dataset*, *path to test dataset*.

After running a training round, the results (the items features and user features vectors corresponding to $\mathbf{P}*$ and $\mathbf{Q}*$ respectively) are stored within the results filed in the following format:
`Features_[DAY]-[MONTH]-[YEAR]_[HOUR]-[MINUTE].txt` where,
`[DAY], [MONTH], [YEAR], [HOUR], [MINUTE]` refer to the current date of the system. The data is serialized using the `cPickle` python library, and contains `Numpy` vectors (see section 3.5 for details).

Further more, if the user choses the option to store additional results about that training round, things like RMSE, and the parameters with which the algorithm had ran, he can do that by enabling the `RECORD_RESULTS_TO_SQL` option. This stores the results in a sqlite3 database (it's format is the same as Table 4.2).

We can find the corresponding files for the user and item feature vectors by having a look at the date field.

3.2. **Analyzing the results with the `Analyzer`.** The Analyzer is a script which analyzes the results. We can use it in order to measure the efficiency of some of the tests we ran. This means, the script loads up the model, and tries to predict the ratings in the test file (learning the $\mathbb{P}, \mathbb{Q}$ models and predicting the ratings using Eq. 6). The level of acceptance, when analyzing weather a prediction was good or bad, can be adjusted using the `tolerance` parameter ($rating \in [predicted\_rating - tolerance, predicted\_rating + tolerance]$). For example, if the rating is 4.0, the predicted rating is 3.7, and the tolerance is 0.5, the prediction is considered to be a success. The parameters are also expressed in python hashes and can be configured within the script.

If no training/test files are specified, the script will load all the results existing in the database described earlier, and test them. The tolerance would be the same and can be modified within the script. This is useful when trying to analyze all the experiments carried so far.

3.3. **Carrying multiple experiments using the `BatchTrainer`.** We have found that running one experiment at a time can be a tedious and boring operation unless we really must do that. Most experiments usually involve changing some parameters and re-running the algorithm, whose speed can vary between a couple of seconds to tens of minutes.

The `BatchTrainer` overcomes this problem by allowing us to describe experiments, and ultimately creating workflows for running multiple algorithms sequentially.

We are able to state multiple parameters for different experiments in the `BatchTrainer.py` script (their meaning is the same as those described in section 3.1). The script will run the experiments one after the other, registering the results.

3.4. **A recommendation service using the `Recommender`.** `Recommender.py` provides a service for querying for user preferences. Given a user id and based on a model, the program returns a list of 10 items, that it "considers" the user would rate as high.

The parameters are the user feature file in the model learned and the user feature file in the model learned.

For example, when asking, "*What items should I recommend to user Alice ?*", the recommender would respond with a list of item ids and the corresponding predicted rating.

3.5. **Technologies used.** The implementation of *Rho* is done using Python version 2.6. The reason for using Python is that we wanted to model and test the different parameters quickly, rather then optimizing the algorithm for speed. For efficiently storing and working with the arrays and matrices, the `NumPy` library [3] was used. The code for this framework can be found in [4].

4. Results

This section presents the results obtained by us when running the implementation described in the previous section. It starts by presenting the dataset (the Movie Lens dataset [5]). Then it describes the attempt to get the correct γ and λ parameters, for minimizing the RMSE error on the models. We also analyzed how does the algorithms performs relative to the number of training epochs or features. Also we were interested in the time needed to run the experiments and the eventual correlation between it and the RMSE evolution. At the end of the chapter, we present a more comprehensive table with many values that have been obtained during the experiment.

$\lambda \setminus \gamma$	0.005	0.007	0.01	0.015	0.02
0.005	0.9333	0.8815	0.8462	0.7097	**0.7011**
0.007	0.9333	0.8815	0.8463	0.7168	0.7026
0.01	0.9333	0.8816	0.8465	0.7197	0.7027
0.015	0.9333	0.8819	0.8470	0.7263	0.7057

TABLE 1. Different RMSEs for λ - regularization factor and γ - learning rate

4.1. **Datasets.** The experiments presented in this article have been carried out using the MovieLens database [5]. MovieLens data sets were collected by the GroupLens Research Project at the University of Minnesota.

This data set consists of:

- 100,000 ratings (1-5) from 943 users on 1682 movies.
- Each user has rated at least 20 movies.

Users and items are numbered consecutively from 1. The data is randomly ordered. This is a tab separated list of user `id | item id | rating | timestamp`.

Regarding the tests dataset: `u1.base` and `u1.test` through `u5.base` and `u5.test` are 80%/20% splits of the u data into training and test data. Each of `u1, ..., u5` have disjoint test sets; this if for 5 fold cross validation (where you repeat your experiment with each training and test set and average the results).

4.2. **Experiments and results.** It has been observed that RSIMF (with regularization factor usually performs better than ISMF). As described throughout this paper, the idea is to minimize the RMSE error in order to obtain better results.

First we wanted to see what are the best learning rates and regularization factors. For that we have tried a couple of tests with different values for the two parameters, which we presented in Table 1. In order to obtain a reasonable training we have used **20 features** and **50 epochs**. We noticed that not every feature was trained 50 times. If the improvement between two epochs is not grater than 0.0001, we move on to training the next feature. The smaller RMSE obtain was 0.7011 which we have achieved for $\gamma = 0.02$ and $\lambda = 0.005$.

The running time for these experiments was about 70962 seconds, about 19.71 hours. The medium training is 0.98 hours. We show this on Figure 2. We notice that the time to train the models which yielded best RMSE is significantly longer than that used to train models with lower RMSE.

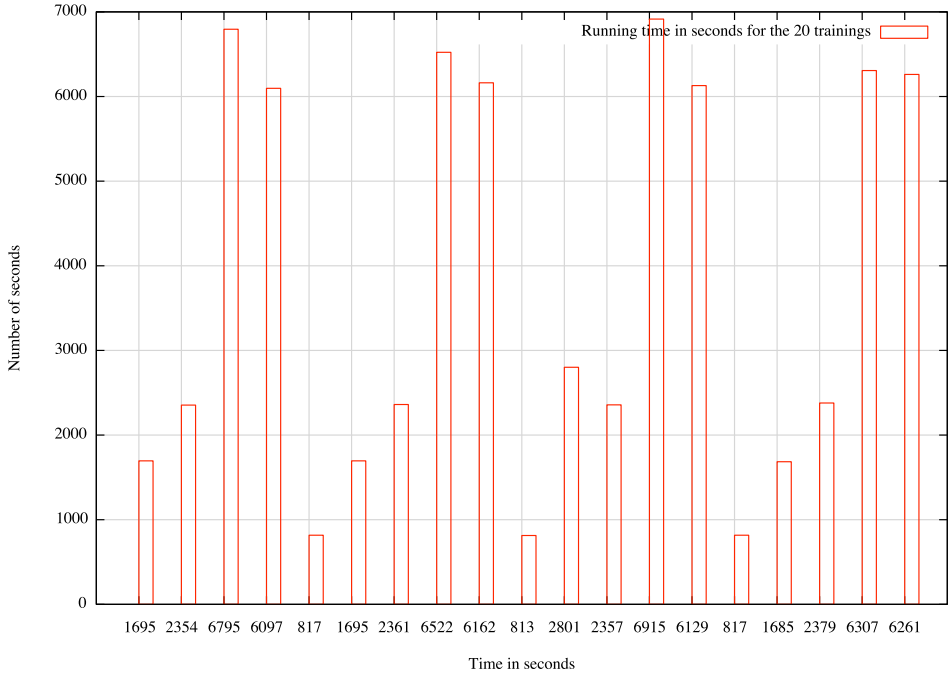

FIGURE 2. Running times for the 20 trainings. First bar corresponds to value (1,1) in Table 1, second bar - (1, 2) and so on.

We have also noticed in Figure 3 a certain periodicity on the RMSE. The values plotted here are from the same experiment presented in Table 1 and follows the same rule as Figure 2.

Table 2 offers a complete overview over the experiments which have been ran.

5. Conclusions and further work

In this thesis, we analyzed a couple of alternatives for building *Recommender Systems* with emphasis on Collaborative Filtering (CF), namely some Matrix Factorization (MF) techniques. Some of the biggest challenges imposed by such systems are scalability and sparsity. We find that often Recommender Systems have to deal with thousands of users and products and potentially hundreds of million of ratings, which result in big matrices with very few ratings - 98-99% sparse - in the case of CF and implicitly MF). It is unfeasible to factorize those matrices through classical linear algebra algorithms. In section 2.1 we describe two machine learning algorithms (named ISMF and RISMF

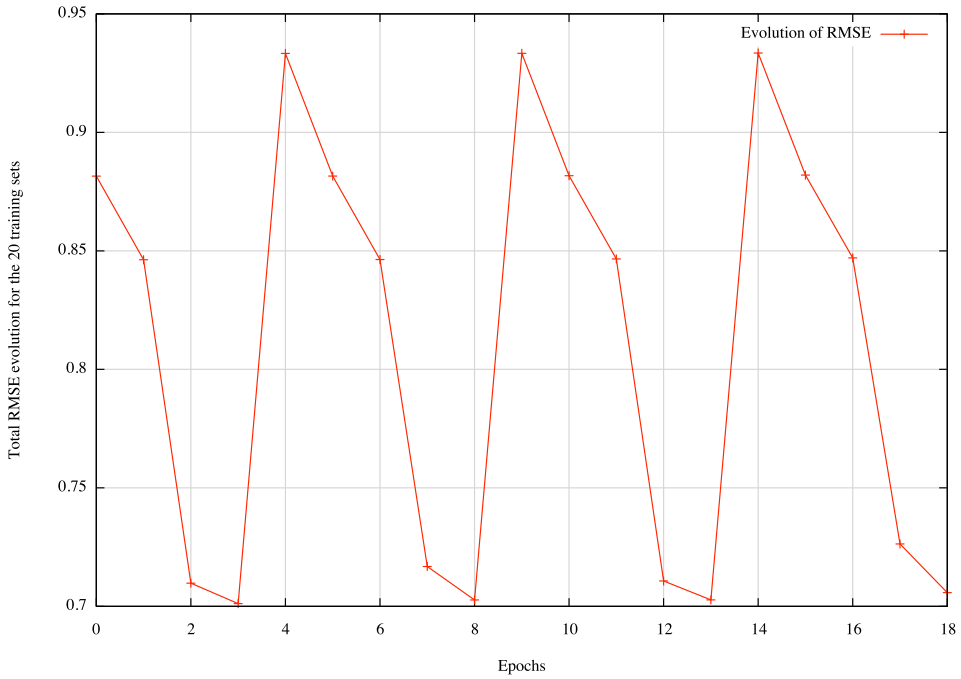

FIGURE 3. Total RMSE evolution on 20 trainings. First bar corresponds to value (1,1) in Table 1, second bar - (1, 2) and so on.

described in [2]) that do this. They overcome these problems by "guessing" the ratings (missing) values.

Further more, in order to do some experiments with these algorithms, we created a framework which allows training models based on the above mentioned algorithms, store and analyze the results in an easy to follow manner. The framework also can be used to query recommendations for best 10 items for a certain user.

Some of the results have been discussed on section 4. The best RMSE used on the MovieLends database (100.000 ratings), was 0.7011 with the RMSIF algorithm.

There are many aspects that can be improved when building Recommender Systems. From a prediction accuracy perspective, authors have tried many other techniques like SVD++ [6] or building a complex model which includes both a *k-Nearest Neighborhood* approach combined with a matrix factorization [7]. These techniques also take into account things like implicit feedback and even time ("temporal effects"). It would be interesting to enrich the existing

alg	epochs	features	time(sec)	RMSE	training DS	test DS
RISMF	50	20	826	0.9332	dataset/u1.base	dataset/u1.test
RISMF	50	20	1695	0.8815	dataset/u1.base	dataset/u1.test
RISMF	50	20	2354	0.8462	dataset/u1.base	dataset/u1.test
RISMF	50	20	6795	0.7097	dataset/u1.base	dataset/u1.test
RISMF	50	20	6097	0.7011	dataset/u1.base	dataset/u1.test
RISMF	50	20	817	0.9332	dataset/u1.base	dataset/u1.test
RISMF	50	20	1695	0.8815	dataset/u1.base	dataset/u1.test
RISMF	50	20	2361	0.8463	dataset/u1.base	dataset/u1.test
RISMF	50	20	6522	0.7168	dataset/u1.base	dataset/u1.test
RISMF	50	20	6162	0.7026	dataset/u1.base	dataset/u1.test
RISMF	50	20	813	0.9333	dataset/u1.base	dataset/u1.test
RISMF	50	20	2801	0.8816	dataset/u1.base	dataset/u1.test
RISMF	50	20	2357	0.8465	dataset/u1.base	dataset/u1.test
RISMF	50	20	6915	0.7107	dataset/u1.base	dataset/u1.test
RISMF	50	20	6129	0.7027	dataset/u1.base	dataset/u1.test
RISMF	50	20	817	0.9334	dataset/u1.base	dataset/u1.test
RISMF	50	20	1685	0.8819	dataset/u1.base	dataset/u1.test
RISMF	50	20	2379	0.8470	dataset/u1.base	dataset/u1.test
RISMF	50	20	6307	0.7263	dataset/u1.base	dataset/u1.test
RISMF	50	20	6261	0.7057	dataset/u1.base	dataset/u1.test

TABLE 2. Table format for storing training rounds results

framework with those algorithms or ideas from them. The users of the framework could either analyze their performances in terms of speed and accuracy.

Because of the large amounts of data (which keeps growing) it starts to be very hard to store it on single computing unit and perform complicated calculations. There may be useful to distribute the computations across multiple computers. Porting these algorithms on distributed frameworks like *hadoop* [**?**] (which uses the MapReduce programming model) would be beneficial. Another way would be to parallelize the factorization operations, on a single computer. In this case we could take advantage of the multicore processors.

Another thing worth mentioning is to make use of "ensemble methods" [**?**] to combine the predicted results of multiple recommender algorithms, using linear regression or other blending algorithms.

REFERENCES

[1] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35:61–70, 1992.

[2] Gábor Takács, István Pilászy, Bottyán Németh, and Domonkos Tikk. Scalable collaborative filtering approaches for large recommender systems. *J. Mach. Learn. Res.*, 10:623–656, 2009. ISSN 1533-7928. URL http://portal.acm.org/citation.cfm?id=1577069.1577091.

[3] Open Source Library. Numpy library. ***. URL http://numpy.scipy.org/.

[4] Cristian Prodan. The rho framework. 2010. URL http://github.com/christian/Rho.

[5] MOVIELENS-DATA. *MovieLens dataset.* URL http://www.grouplens.org/node/73.

[6] R. M. Bell and Y. Koren. Scalable collaborative filtering with jointly derived neighborhood interpolation weights. In *ICDM '07: Proceedings of the 2007 Seventh IEEE International Conference on Data Mining*, pages 43–52, Washington, DC, USA, October 2007. IEEE Computer Society. ISBN 0-7695-3018-4. doi: 10.1109/ICDM.2007.90. URL http://dx.doi.org/10.1109/ICDM.2007.90.

[7] Yehuda Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *Proc. Int. Conf. on Knowledge Discovery and Data Mining*, pages 426–434, 2008.

Babeş-Bolyai University, Faculty of Mathematics and Computer Science, Cluj-Napoca, Romania

E-mail address: prodan.cristian@gmail.com