Factorization-Based Large Scale Recommendation Algorithms

PhD thesis

István Pilászy

Under the supervision of
Dr. Domonkos Tikk, Dr. Tadeusz P. Dobrowiecki

Budapest University of Technology and Economics, Hungary
Department of Measurement and Information Systems

Contents

Acknowledgements

1 Preliminaries
   1.1 List of acronyms
   1.2 Notation
   1.3 Introduction
   1.4 Problem definition

2 Literature survey
   2.1 Performance measures
   2.2 Collaborative filtering methods
   2.3 Matrix factorization methods for rating-based CF
      2.3.1 Incremental gradient descent method for matrix factorization
      2.3.2 Alternating least squares (ALS) for matrix factorization
      2.3.3 NSVD1 for matrix factorization
   2.4 Content-based filtering methods
   2.5 New CF methods from the Netflix Prize contest
   2.6 Datasets
      2.6.1 The Netflix Prize dataset
      2.6.2 The MovieLens dataset
      2.6.3 The Jester dataset
      2.6.4 Analysis of statistical significance of results

3 Improving on the accuracy of MF
   3.1 Improving the performance of a single model
      3.1.1 Parameter optimization
      3.1.2 Introduction of parameters
      3.1.3 Evaluation
      3.1.4 Evaluation on Jester
      3.1.5 Evaluation on MovieLens
      3.1.6 Conclusion
   3.2 Parameter optimization on multiple models
   3.3 Support-based regularization
   3.4 Resetting the user matrix improves performance
   3.5 Hybrid MF+NSVD1 method
      3.5.1 Matrix factorization and NSVD1 as neural networks
      3.5.2 Combining the two neural networks
CONTENTS

3.5.3 Related works ................................................. 60
3.5.4 Results ......................................................... 61
3.6 RMSE values reported by other authors ......................... 62
3.7 Summary .......................................................... 63

4 Improving on the speed of matrix factorization ......... 65
4.1 Retraining user features for handling new ratings .... 65
4.1.1 Experiments ..................................................... 66
4.2 Fast and accurate MF by parameter optimization .... 70
4.3 Applications of the Sherman-Morrison formula ....... 72
4.3.1 ALS on explicit feedback datasets ....................... 72
4.3.2 ALS on implicit feedback datasets ....................... 72
4.3.3 Handling new ratings of users with ALS .......... 74
4.3.4 Backward and forward feature selection .......... 75
4.4 Applications of kernel ridge regression ................. 76
4.4.1 KRR on explicit feedback datasets .................... 76
4.4.2 Handling new ratings of users with KRR .......... 77
4.5 Summary .......................................................... 78

5 Content-based filtering ........................................ 79
5.1 Using NSVD1 for content-based filtering .................. 79
5.2 Related work ....................................................... 81
5.3 Experiments ........................................................ 82
5.3.1 The Netflix Prize dataset .................................. 82
5.3.2 Results ............................................................ 83
5.4 Summary .......................................................... 86

6 Explaining recommendations ................................ 89
6.1 Explaining recommendations of ALS ....................... 89
6.2 Explaining recommendations using dual formulation of RR 84
6.3 Explaining recommendations of BRISMF ................. 90
6.3.1 Comparing primal and dual formulation .......... 93
6.4 Experiments ........................................................ 93
6.4.1 Examining primal formulation for gradient methods 93
6.5 Summary .......................................................... 96

Summary .............................................................. 97

Further research ..................................................... 101

Bibliography .......................................................... 103
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Most important publicly available CF datasets for explicit feedback</td>
</tr>
<tr>
<td>3.1</td>
<td>Minimum and maximum RMSE using different initial PRNG seeds for parameter</td>
</tr>
<tr>
<td></td>
<td>optimization, in specific loops</td>
</tr>
<tr>
<td>3.2</td>
<td>A very accurate matrix factorization for the Netflix Prize with different</td>
</tr>
<tr>
<td></td>
<td>$K$ values</td>
</tr>
<tr>
<td>3.3</td>
<td>Probe10 and Quiz RMSE of MF models (rows 1–4) and their combinations</td>
</tr>
<tr>
<td></td>
<td>(rows 5–10)</td>
</tr>
<tr>
<td>3.4</td>
<td>Probe10 and Quiz RMSE of SBRISMFs (rows 1–3) and their combinations (rows</td>
</tr>
<tr>
<td></td>
<td>4–8)</td>
</tr>
<tr>
<td>3.5</td>
<td>Examining the effect of retraining user features</td>
</tr>
<tr>
<td>3.6</td>
<td>Results of hybrid MF+NSVD1 matrix factorizations (rows 1–10) and their</td>
</tr>
<tr>
<td></td>
<td>combinations (other rows)</td>
</tr>
<tr>
<td>3.7</td>
<td>Comparison of Quiz RMSE values of reported MF based methods. I also</td>
</tr>
<tr>
<td></td>
<td>indicate the date of publication</td>
</tr>
<tr>
<td>4.1</td>
<td>Examining the effect of retraining user features</td>
</tr>
<tr>
<td>4.2</td>
<td>Examining the addition of new users on BRISMF#250.</td>
</tr>
<tr>
<td>4.3</td>
<td>Examining the addition of new ratings.</td>
</tr>
<tr>
<td>4.4</td>
<td>Probe10 RMSE of the second training step, when using only a subset of</td>
</tr>
<tr>
<td></td>
<td>users</td>
</tr>
<tr>
<td>4.5</td>
<td>Number of epochs in the first training step and Probe10 RMSE of the second</td>
</tr>
<tr>
<td></td>
<td>training step, when using only the first $N$ ratings of movies.</td>
</tr>
<tr>
<td>4.6</td>
<td>RMSE and running time of fast and accurate MFs.</td>
</tr>
<tr>
<td>4.7</td>
<td>Average running time of recomputing a $K$-dimensional user feature vector,</td>
</tr>
<tr>
<td></td>
<td>when the number of ratings of the user is in an interval</td>
</tr>
<tr>
<td>5.1</td>
<td>Probe10 RMSE of movie-NSVD1 with different metadata</td>
</tr>
<tr>
<td>5.2</td>
<td>X10 RMSE of bias predictors, when $N$ ratings from each skipped movie</td>
</tr>
<tr>
<td></td>
<td>are added to the training set</td>
</tr>
<tr>
<td>6.1</td>
<td>Comparing different explanation methods for the same model.</td>
</tr>
<tr>
<td>6.2</td>
<td>How many times a prediction for a movie from a set of similar movies is</td>
</tr>
<tr>
<td></td>
<td>mostly explained by a rating on a movie from the same set</td>
</tr>
</tbody>
</table>
List of Figures

2.1 The train-test split and the naming convention of Netflix Prize dataset, after Bell et al. 3 37
3.1 Effect of the number of users on Probe10 RMSE and on the optimal number of training epochs. 44
3.2 Comparing BRISMF with 4 and 8 parameters using parameter optimization. X-axis: the number of MF evaluations (loops). Y-axis: the best RMSE result until that loop. 45
3.3 Comparing parameter optimization on BRISMF with 8 parameters, using 1/10 of the training set / full training set, for various $K$ values. 46
3.4 The multilayer perceptron equivalent for the general MF scheme 57
3.5 The multilayer perceptron equivalent for NSVD1 58
3.6 The multilayer perceptron for hybrid MF + NSVD1 learning 59

List of Algorithms

1 Training algorithm for BRISMF 30
2 Training algorithm for NSVD1 34
3 Simple parameter optimization algorithm 42
4 Algorithm for retraining user features 66
5 Algorithm to recompute $p_u$ in $O(K^2 \cdot n_u)$ time for explicit feedback datasets. 73
6 Training algorithm for movie-NSVD1 80
First, I would like to thank my advisor, Domonkos Tikk. He always let me know about the relevant call-for-papers, helped me in funding the large costs of travelling to distant conferences: I especially thank him to be able to participate at the ACM KDD’07-Netflix Workshop, and at the ACM RecSys’09 Conference, both in the U.S. He was also very helpful in reviewing my papers, even if this required long Skype calls in the evening, midnight, or weekend, or he was due to submit another paper. His hard working style was also very inspiring.

I would like to thank my second advisor, Tadeusz P. Dobrowiecki for helping me starting my academic career.

I would like to thank the Technical University of Budapest (now Budapest University of Technology and Economics), for teaching me so many courses with strong maths.

I would like to thank Gábor Takács, for inviting me to join team Gravity, which participated very successfully in the Netflix Prize competition. The discussions with him were always very interesting and inspiring, no matter whether it was about collaborative filtering, the broader machine learning, or anything else.

I am also thankful to Bottyán Németh for finding that addictive Netflix Prize competition.

I thank the reviewers of JMLR for their valuable comments and requests.

I thank Netflix for organizing the Netflix Prize contest, and generously releasing a dataset of more than 100 million ratings for the community. The leaderboard of the competition with instant feedback was a very good idea, which made this contest so addictive.

Last, but not least, I would like to thank my whole family, but especially my wife for having the power of making me write this dissertation.
Chapter 1

Preliminaries

1.1 List of acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Long form</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALS</td>
<td>Alternating Least Squares</td>
<td>✓</td>
</tr>
<tr>
<td>BRISMF</td>
<td>Biased Regularized Incremental Simultaneous MF</td>
<td>✓</td>
</tr>
<tr>
<td>CBF</td>
<td>Content-Based Filtering</td>
<td>✓</td>
</tr>
<tr>
<td>CF</td>
<td>Collaborative Filtering</td>
<td>✓</td>
</tr>
<tr>
<td>GBFS</td>
<td>Greedy Backward Feature Selection</td>
<td>✓</td>
</tr>
<tr>
<td>GFFS</td>
<td>Greedy Forward Feature Selection</td>
<td>✓</td>
</tr>
<tr>
<td>ID</td>
<td>Identifier</td>
<td>✓</td>
</tr>
<tr>
<td>IGD</td>
<td>Incremental Gradient Descent</td>
<td>✓</td>
</tr>
<tr>
<td>IoC</td>
<td>Improvement over Cinematch</td>
<td>✓</td>
</tr>
<tr>
<td>ISMF</td>
<td>(like BRISMF, without B and R)</td>
<td>✓</td>
</tr>
<tr>
<td>KRR</td>
<td>Kernel Ridge Regression</td>
<td>✓</td>
</tr>
<tr>
<td>LS</td>
<td>Least Squares</td>
<td>✓</td>
</tr>
<tr>
<td>MAE</td>
<td>Mean Absolute Error</td>
<td>✓</td>
</tr>
<tr>
<td>MF</td>
<td>Matrix Factorization</td>
<td>✓</td>
</tr>
<tr>
<td>MMMF</td>
<td>Maximum Margin Matrix Factorization</td>
<td>✓</td>
</tr>
<tr>
<td>NB</td>
<td>Neighbor</td>
<td>✓</td>
</tr>
<tr>
<td>NDCG</td>
<td>Normalized Discounted Cumulative Gain</td>
<td>✓</td>
</tr>
<tr>
<td>NN</td>
<td>Neural Network</td>
<td>✓</td>
</tr>
<tr>
<td>NP</td>
<td>Netflix Prize</td>
<td>✓</td>
</tr>
<tr>
<td>NSVD1</td>
<td>(It is a method name, no long form available)</td>
<td>✓</td>
</tr>
<tr>
<td>PLSA</td>
<td>Probabilistic Latent Semantic Analysis</td>
<td>✓</td>
</tr>
<tr>
<td>PRNG</td>
<td>Pseudo Random Number Generator</td>
<td>✓</td>
</tr>
<tr>
<td>RBM</td>
<td>Restricted Boltzmann Machine</td>
<td>✓</td>
</tr>
<tr>
<td>RISMF</td>
<td>(like BRISMF, without B)</td>
<td>✓</td>
</tr>
<tr>
<td>RMSE</td>
<td>Root Mean Squared Error</td>
<td>✓</td>
</tr>
<tr>
<td>RR</td>
<td>Ridge Regression</td>
<td>✓</td>
</tr>
<tr>
<td>SMF</td>
<td>Sherman-Morrison Formula</td>
<td>✓</td>
</tr>
<tr>
<td>SSE</td>
<td>Sum of Squared Error</td>
<td>✓</td>
</tr>
<tr>
<td>SVD</td>
<td>Singular Value Decomposition</td>
<td>✓</td>
</tr>
<tr>
<td>WRR</td>
<td>Weighted Ridge Regression</td>
<td>✓</td>
</tr>
</tbody>
</table>
CHAPTER 1. PRELIMINARIES

The last column denotes whether the acronym is widely used.

1.2 Notation

Generally, I use the following notations:

- Column vectors are denoted by bold lower-case letters, e.g. $\mathbf{a}, \mathbf{b}, \mathbf{p}_u, \mathbf{q}_i$, etc.
- Matrices are denoted by bold upper-case letters, e.g. $\mathbf{A}, \mathbf{P}, \mathbf{Q}$, etc.
- Letter or number in subscript: an element of a vector or matrix. E.g.: $\mathbf{p}_u$ is the transpose of the $u$-th row of $\mathbf{P}$. Another example: $p_{uk}$ is the $(u, k)$-th element of the matrix $\mathbf{P}$.
- Letter or number in superscript: distinction. When I need multiple $\eta$ parameters, with similar but different meaning, I add superscript. For example, $\eta^b$ is different from $\eta$. Here $b$ is not a variable holding a positive integer: it is a letter not to be substituted. In this dissertation there will be no raising to a power of a variable.
- Superscript “hat” denotes the prediction of the given quantity, so $\hat{x}$ is the prediction of $x$.

In the followings I list notations which are used in many places in this dissertation. Each of them will be defined where used first.

- $N$: number of users
- $M$: number of movies or items.
- $u, v \in \{1, \ldots, N\}$: indices for users.
- $i, j \in \{1, \ldots, M\}$: indices for movies or items.
- $r_{ui}$: the rating of user $u$ on item $i$.
- $d_{ui}$: the date when user $u$ made a rating on item $i$.
- $\hat{r}_{ui}$: the prediction of $r_{ui}$.
- $\mathbf{R}$: the matrix of $r_{ui}$ values (for both explicit and implicit ratings), i.e. the matrix of ratings.
- $\mathcal{R}$: for explicit feedback: the set of $(u, i)$ indices of $\mathbf{R}$ where a rating is provided; for implicit feedback: the set of all indices of $\mathbf{R}$.
- $\mathcal{R}^+$: only for implicit feedback, the set of $(u, i)$ indices of $\mathbf{R}$ where the feedback is non-zero (user $u$ watched item $i$).
- $e_{ui} = r_{ui} - \hat{r}_{ui}$: the error of prediction on the $(u, i)$-th rating.
- $l_{ui}$: the training loss associated with the $(u, i)$-th rating.
- $L$: the training loss associated on the training set.
1.2. NOTATION

\( n_u \): user support, i.e. number of ratings of user \( u \). Definition: \( n_u = |\{i : (u, i) \in \mathcal{R}\}|. \)

\( n_i \): item support, i.e. number of ratings of item \( i \). Definition: \( n_i = |\{u : (u, i) \in \mathcal{R}\}|. \)

\( K \): the number of features (factors) of matrix factorization.

\( k \): index for features (factors).

\( \mathbf{P} \in \mathbb{R}^{N \times K} \): the user feature matrix.

\( \mathbf{p}_u \in \mathbb{R}^{K \times 1} \), the transpose of the \( u \)-th row of \( \mathbf{P} \): the user feature vector.

\( p_{uk} \): the \((u, k)\)-th element of \( \mathbf{P} \)

\( \mathbf{b} \in \mathbb{R}^{N \times 1} \): the user biases

\( b_u \): the \( u \)-th user bias, i.e. the \( u \)-th element of \( \mathbf{b} \).

\( \mathbf{Q} \in \mathbb{R}^{M \times K} \): the item feature matrix (movie feature matrix),

\( \mathbf{q}_i \in \mathbb{R}^{K \times 1} \), the transpose of the \( i \)-th row of \( \mathbf{Q} \): the item feature vector.

\( q_{ik} \): the \((i, k)\)-th element of \( \mathbf{Q} \).

\( \mathbf{c} \in \mathbb{R}^{M \times 1} \): the item biases

\( c_i \): the \( i \)-th item bias, i.e. the \( i \)-th element of \( \mathbf{c} \).

\( w^p, w^q, w^\mathbf{p}, w^\mathbf{q} \): when \( \mathbf{P}, \mathbf{Q}, \mathbf{b} \) and \( \mathbf{c} \) are initialized uniform random, these numbers specify the minimum and maximum values of initialization, respectively.

\( A_u \): used for ridge regression, denoting the covariance matrix of input (considering user \( u \) in context).

\( B_u \): like \( A_u \), but contains also the regularization term (increased diagonal).

\( d_u \): used for ridge regression, denoting the input-output covariance vector.

\( \mathbf{I} \): denotes the identity matrix of the appropriate size.

\( \lambda \): regularization parameter.

\( \eta \): learning rate for gradient methods.

\( C \): dimension of the vector space of the vectorial representation of user or movie (item) metadata (depends on the context). Vectors in that space are usually sparse.

\( l \): index for the metadata features \((l \in \{1, \ldots, C\}\)
1.3 Introduction

Recommender systems attempt to profile user preferences over items, and model the relation between users and items. The task of recommender systems is to recommend items that fit user’s tastes, in order to help the user in selecting/purchasing items from an overwhelming set of choices. Such systems have great importance in applications such as e-commerce, subscription based services, information filtering, etc. Recommender systems providing personalized suggestions greatly increase the likelihood of a customer making a purchase compared to unpersonalized ones. Personalized recommendations are especially important in markets where the variety of choices is large, the taste of the customer is important, and last but not least the price of the items is modest. Typical areas of such services are mostly related to art (especially books, movies, music), fashion, food and restaurants, gaming and humor.

With the burgeoning of web based businesses, an increasing number of web based merchant or rental services use recommender systems. Some of the major participants of e-commerce web, like Amazon and Netflix, successfully apply recommender systems to deliver automatically generated personalized recommendation to their customers. The importance of a good recommender system was recognized by Netflix, which led to the announcement of the Netflix Prize (NP) competition to motivate researchers to improve the accuracy of the recommender system of Netflix.

There are two basic strategies that can be applied when generating recommendations. Content-based approaches profile users and items by identifying their characteristic features, such us demographic data for user profiling, and product information/descriptions for item profiling. The profiles are used by algorithms to connect user interests and item descriptions when generating recommendations. However, it is usually laborious to collect the necessary information about items, and similarly it is often difficult to motivate users to share their personal data to help create the database for the basis of profiling.

Therefore, the alternative approach, termed collaborative filtering (CF), which makes use of only past user activities (for example, transaction history or user satisfaction expressed in ratings), is usually more feasible. CF approaches can be applied to recommender systems independently of the domain. CF algorithms identify relationships between users and items, and make assumptions using this information to predict user preferences.

**Netflix Prize** Netflix initiated the Netflix Prize competition in October 2006, to improve their recommender system called Cinematch, which was developed in many years. The goal of the competition is to predict users’ ratings on given movies. Predictions are evaluated in terms of RMSE (Root Mean Squared Error). Interestingly, after 3 weeks the competition began, Cinematch’s performance score has been surpassed. This means that for someone, only 3 weeks were enough to create something better compared to Cinematch, which incorporated many man-years. Netflix offered 50,000 USD Progress Prizes at the anniversaries of the competition, and also a 1,000,000 USD Grand Prize for the best team, which gives 10% better RMSE score than Cinematch. The competition ended in July 26, 2007, with a 10.09% improvement.

The competition involved a lot of research effort on CF algorithms. Due to the competition, the field of CF has completely changed, and former methods are now completely obsolete. Before the competition, many researches focused on Pearson-correlation-based
1.3. INTRODUCTION

k-nearest neighbors methods. However, during the competition it turned out that matrix factorization-based approaches are much better than the aforementioned methods, in training time, prediction time and accuracy.

I got involved into this competition in November 2006, as a member of team Gravity. We had been leading the competition between January 2007 and April 2007, and were always at the top.

Outline First part of my propositions deals with Netflix Prize related algorithms that seek for better RMSE score.

The second part deals with how matrix factorization algorithms can be speed up: how to speed up the model building process; how to efficiently handle new users, or new ratings of users.

The third part deals with how textual description of movies can be used to give recommendations to users, especially recommending new movies, i.e. movies that no one has rated (yet). Again, the focus is on getting the lowest possible error of predictions. I point out that if a movie has 10 ratings, then this information is more predictive than the textual description of the movie.

The fourth part presents a method to explain predictions of recommendation algorithms.

Whenever I use the personal pronoun I, I refer to my own achievements. Whenever I use the personal pronoun we, I refer to the achievements of the team Gravity, i.e. those achievements are not mine. Sometimes I refer to the achievements of the team by citing a paper in third person, e.g. Takács et al. proposed the following: .... These sentences are not to be treated as my achievements, since they are not in first person.

Why to seek after a better RMSE In the Netflix Prize competition, even very small differences were decisive in obtaining prizes. For example, in October 2, 2007, team When Gravity and Dinosaurs Unite missed only 0.0005 RMSE behind the first team. In July 26, 2009, the first two team had exactly the same RMSE score, but team The Ensemble –which I was a member of– submitted 20 minutes later than the first team, BellKor’s Pragmatic Chaos, and so they won the Grand Prize.

Another important reason for seeking low RMSEs is that although a score of 0.9000 and 0.8950 are both far from zero, but this small improvement is huge, if we compare the methods to a hypothetical score, for example to 0.8500. Obviously there is a limit on improving the scores, and apparently the limit is closer to 0.8500 than to zero. Thus, even small improvements –e.g. 0.0010– means that we are getting much closer to the limit in modelling users’ behavior. We may think at that limit as the maximum what we can squeeze from the data, and all other variances are just noise.

Moreover, Koren pointed out that even small improvements in RMSE yield huge differences in top-N recommendations, where users are recommended with items they are most likely to like.

I also experienced, that when a matrix factorization model is used to calculate movie similarities, a model with better RMSE (e.g. 0.9300 versus 0.9000) yields much more sensible most-similar-pairs, intuitively.
1.4 Problem definition

I define the problem of collaborative filtering (CF) in the following setting. The problem can be modeled by the random triplet \((U, I, R)\), where

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

A realization of \((U, I, R)\) denoted by \((u, i, r)\) means that user \(u\) rated item \(i\) with value \(r\). Let \(P(U = u, I = i, R = r)\) denote the probability of this event. The probabilities can be arranged into a \(N \times M \times \mathcal{X}\) table. I assume that ratings are deterministic, i.e.

\[
P(R = r | U = u, I = i) = 1. \tag{1.1}
\]

Note that the \(P(U = u, I = i)\) is the probability of giving a rating on item \(i\) by user \(u\).

The goal is to estimate \(R\) from \((U, I)\) such that the root mean squared error of the estimate,

\[
\text{RMSE} = \sqrt{\mathbb{E}\{(R - \hat{R})^2\}}, \tag{1.2}
\]

is minimal, where the expectation is taken over \((U, I, R)\). \(\hat{R}\) is the deterministic estimate of \(R\). I briefly discuss other possible evaluation metrics in CF in Section 2.1.

Assume that we are given a finite i.i.d. sample of \((U, I, R)\) realizations:

\[
\mathcal{R}' = \{(u_1, i_1, r_1), (u_2, i_2, r_2), \ldots, (u_t, i_t, r_t)\}.
\]

The sample \(\mathcal{R}'\) can be used for training predictors, or to estimate the distribution of \((U, I, R)\). Let us introduce the notation \(\mathcal{R} = \{(u, i) : \exists r : (u, i, r) \in \mathcal{R}'\}\) for the set of (user ID, item ID) pairs. The sample can be represented as a partially specified matrix denoted by \(R \in \mathbb{R}^{N \times M}\), where the matrix elements are known in positions \((u, i) \in \mathcal{R}\), and unknown in positions \((u, i) \notin \mathcal{R}\). The value of the matrix \(R\) at position \((u, i) \in \mathcal{R}\), denoted by \(r_{ui}\), stores the rating of user \(u\) for item \(i\). For clarity, I use the term \((u, i)\)-th rating in general for \(r_{ui}\), and \((u, i)\)-th training example if \(r_{ui} : (u, i) \in \mathcal{R}\).

The goal of CF is to create such predictors that aim at minimizing the error [12]. 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, for example, by randomly partitioning the above sample into a smaller training set and a validation set. Let the validation set be denoted by \(\mathcal{V}'\). Let \(\mathcal{V} = \{(u, i) : \exists r : (u, i, r) \in \mathcal{V}'\}\). The estimate of RMSE can be calculated as

\[
\hat{\text{RMSE}} = \sqrt{\frac{1}{|\mathcal{V}'|} \sum_{(u, i, r) \in \mathcal{V}'} (\hat{r}_{ui} - r_{ui})^2}. \tag{1.3}
\]

\[1\]In general, superscript “hat” denotes the prediction of the given quantity, so \(\hat{x}\) is the prediction of \(x\).
1.4. PROBLEM DEFINITION

For better readability, from now on I omit the “hat” from the RMSE, recalling that I always calculate the estimate of the error.

In practice usually additional assumptions are made:

• The $\mathcal{R}'$ sample above does not contain repeated ratings (sampling without replacement). This makes the estimation of RMSE somewhat biased. In other situations, users remember which items they rated, and they never re-rate items. For such situations modeling the generation of events by the $(U, I, R)$ triplet may neglect important details of the rating generation process.

• Since $\mathcal{R}'$ does not contain repeated ratings, $|\mathcal{R}'| = |\mathcal{R}|$, $|\mathcal{V}| = |\mathcal{V}'|$ and $\mathcal{R} \cap \mathcal{V} = \emptyset$.

• In many cases, the date of the rating is also available, which will be denoted by $d_{ui}$.

• Usually the partitioning into training set and the validation set is not random, which makes the estimation of RMSE even more biased. For example, in the Netflix Prize dataset, each user has roughly the same number of ratings in the validation set, maybe because each user is equally important for the Netflix company. Moreover, in the Netflix dataset, ratings in the validation set are newer than in the training set.

When I predict a given rating $r_{ui}$ by $\hat{r}_{ui}$ I refer to the user $u$ as active user and to the item $i$ as active item.
CHAPTER 1. PRELIMINARIES
Chapter 2

Literature survey

The first works on collaborative filtering dates back to the early 1990s. Goldberg et al. [18] presented the Tapestry system which filtered e-mails using collaborative filtering. The GroupLens system described by Resnick et al. [42] allowed users to rate articles in a 1–5 scale, after having read them. Breese et al. [14] divided the CF methods into two groups: memory-based approaches operate on the entire database of ratings to give a recommendation to a user, while model based approaches first build a model of the database, which is then used for recommendations.

Over the last broad decade many CF algorithms have been proposed that approach the problem by different techniques, including similarity/neighborhood based approaches [12, 46], personality diagnosis [34], Bayesian networks [14], restricted Boltzmann machines (RBM) [43], and various matrix factorization techniques [13, 22, 47, 51]. Breese et al. [14] surveyed in detail the major CF approaches of the early years, Rashid et al. [40] gave a short description of most recent methods, and Adomavicius and Tuzhilin [1] also investigated in their survey the possible extensions.

There is an emerging importance of CF and recommender systems, both in the industry and in the research community. There are many companies in the world offering recommender systems. On the other hand, the number of conferences and workshops dedicated to focus only on recommender systems is also numerous. I list some of the most popular ones:

- Netflix Prize workshops: both in 2007 and 2008, at the KDD conference a workshop was dedicated to the Netflix Prize and recommender systems.
- Recsys’07, Recsys’08 and Recsys’09: The first, second and third ACM Int’l Conference on Recommender Systems. This is the most popular conference on recommender systems, for example, in 2009 only 17% of papers were accepted.

Clearly, recommender systems is a very popular and living research area.

The research field of recommender systems is very broad and covers many aspects:

Collaborative Filtering (CF):

- based on explicit feedback: given the matrix of ratings expressing user opinion, try to predict missing ratings.
  Example: Netflix Prize competition.
- based on implicit feedback: given the shopping history of users (user behav-
ior), try to predict which item is the most likely to be bought by the user.

Example: amazon.com recommendations.

**Content-based filtering (CBF)**: given the metadata about items, and user feedback (explicit or implicit), try to predict which items the user likes. Users do not profit from each others ratings.

Example: teaching our mailer agent to filter spam, where items are mails, metadata is the email itself.

**Hybrid CF-CBF systems**: try to merge both information sources.

Example: teaching our mailer agent to filter spam, but also fetching some spam filtering rules from a central server (and also uploading our manual separation of spam and not-spam e-mails).

Another example: blend the output of a CF and a CBF system. For old movies, use CF, for new movies, use CBF.

**User interface** of the recommender system. In the simplest and most common case, a list of recommended items is displayed to users. However, there are conversational approaches, where the recommender system asks few questions to users, and refines the recommendations based on the answers. These systems usually incorporates domain expert knowledge.

**Explaining recommendations**: Recommender systems may be capable of explaining why they “think” a particular item meets the user’s taste. Many kinds of explanation exist, depending on the underlying recommendation algorithm, for example: explanation can be based on the similar items found in the user’s history, or the list of similar users, who have already rated the recommended product positively, or in a movie recommender system, the list of actors the user likes.

**Preserving privacy** in recommender systems.

Example: there are recommender systems (e.g. [2], [65]) that do not need to collect the users’ ratings, only some information extracted from the ratings, for example the errors of the predictions.

**Robust recommender systems**: they are resistant against manipulation, e.g. when a small collaborative group of users try to promote or demote a product.

**Using context** for recommendations. There may be many contextual parameters that should influence the recommended items. For example, when a user is reading a web page which describes an item, the recommender system should recommend items that are more-or-less related to that item. Another example: on a news portal, users may need different news-recommendations when browsing at a workplace or at home.

**Recommender systems in social networks**: Traditional recommender systems have no information about the explicit connection between users. Two users are similar, if they have similar taste. However, in social networks, the connection between users may be exploited.

**Explicit feedback vs. implicit feedback** There are two types of user feedback: explicit feedback and implicit feedback. In case of explicit feedback, users explicitly express their opinion by rating items. Usually they rate only a very small subset of items. In this case, $|R| \ll NM$.

Another type of user feedback is implicit feedback: users do not explicitly express their opinion by ratings. They watch/buy/rent certain items, from which we can con-
clude that they like those items (positive feedback). The rest of the items are not watched/bought/rented, from which we may conclude that they do not like those items (negative feedback), or they just did not know the existence of an item. In this case, all \((u, i)\) pairs are rated either positively or negatively, thus \(|\mathcal{R}| = NM\).

This two types of user feedback needs to be handled differently. Most of this dissertation except a few chapters deals with explicit feedback.

Datasets for explicit feedback Many publicly available benchmarking datasets exists for recommender systems’ researchers, most of them are for evaluating the accuracy of collaborative filtering or content-based filtering algorithms. In Table 2.1 I summarized the most important sets.

<table>
<thead>
<tr>
<th>Name</th>
<th># of users</th>
<th># of items</th>
<th># of ratings</th>
<th>sparsity</th>
<th>date</th>
<th>items</th>
</tr>
</thead>
<tbody>
<tr>
<td>EachMovie</td>
<td>72916</td>
<td>1628</td>
<td>2811983</td>
<td>1/42</td>
<td>1997</td>
<td>movies</td>
</tr>
<tr>
<td>Movielens-100k</td>
<td>943</td>
<td>1682</td>
<td>100000</td>
<td>1/16</td>
<td>1999</td>
<td>movies</td>
</tr>
<tr>
<td>Movielens-1M</td>
<td>6040</td>
<td>3900</td>
<td>1000000</td>
<td>1/24</td>
<td>2003</td>
<td>movies</td>
</tr>
<tr>
<td>Jester</td>
<td>73421</td>
<td>100</td>
<td>4136360</td>
<td>1/1.78</td>
<td>2003</td>
<td>jokes</td>
</tr>
<tr>
<td>Book-crossing</td>
<td>278858</td>
<td>271379</td>
<td>1149780</td>
<td>1/65818</td>
<td>2005</td>
<td>books</td>
</tr>
<tr>
<td>Dating agency</td>
<td>135359</td>
<td>220970</td>
<td>17359346</td>
<td>1/1723</td>
<td>2006</td>
<td>people</td>
</tr>
<tr>
<td>Netflix</td>
<td>480189</td>
<td>17770</td>
<td>100480507</td>
<td>1/85</td>
<td>2006</td>
<td>movies</td>
</tr>
<tr>
<td>Movielens-10M</td>
<td>71567</td>
<td>10681</td>
<td>10000000</td>
<td>1/76</td>
<td>2009</td>
<td>movies</td>
</tr>
</tbody>
</table>

\(^{a}\)Withdrawn

\(^{b}\)http://www.grouplens.org/

\(^{c}\)http://www.ieor.berkeley.edu/goldberg/jester-data/

\(^{d}\)http://www.informatik.uni-freiburg.de/cziegler/BX/

\(^{e}\)http://www.occamslab.com/petricek/data/

\(^{f}\)http://www.netflixprize.com/

Interestingly, most of the publicly available CF datasets deals with explicit feedback on movies. For implicit feedback data, there are much less public datasets. However, one can easily convert explicit feedback datasets into implicit feedback data, by converting rated items to 1, and not rated items to 0, for each user.

Fields related to collaborative filtering In general, CF methods deal with ratings arranged into a matrix, where rows correspond to users, columns correspond to movies. There are many other practical problems, where the efficient handling of large matrices is crucial.

In the field of text mining, a set of text documents are often represented in a so-called term-document matrix, where rows corresponds to words, columns corresponds to documents, and an element of the matrix indicates whether a word occurs in a document or not. Deerwester et al. \(^{16}\) successfully applied singular value decomposition for handling term-document matrices (SVD will be described on page \(^{23}\)). SVD (which is special matrix factorization technique for fully specified matrices) is also a well-known method in Collaborative Filtering. Sarwar et al. \(^{47}\) applied pure SVD for the Movielens-100k dataset. They concluded that filling missing ratings by movie-average is better than user-average.
Gorrell and Webb\cite{20} applied an incremental variant of SVD for text mining. Simon Funk (this is a pseudonym, his real name is Brandyn Webb), a co-author of that paper, published in the very beginning of the NP competition his approach \cite{67}, which is essentially the same as described in that paper. This illustrates how different fields of data mining and text mining can benefit from each others’ advances.

Another interesting application of matrix factorization is the reconstruction of images, when some pixels are missing.\footnote{http://www.netflixprize.com/community/viewtopic.php?id=1528}

Srebro\cite{52} summarizes four practical applications of matrix factorization:

- Signal reconstruction: for example, by analyzing psychometric data, we can reconstruct the underlying characteristics of people, that explains their answers to a series of questions.
- Lossy compression: the factorized form of the matrix is usually more compact than the original.
- Understanding structure: the relationship between columns (or rows) of a factor matrix reflects the relationships between items (or users).
- Prediction: missing values can be predicted by multiplying the factor matrices.

\subsection{2.1 Performance measures}

There are many characteristics of recommendation algorithms:

- accuracy: the system should recommend items that users like. This is the most important characteristic.
- computational complexity: running time and memory requirements (both random access memory and disk storage).
- handling new ratings: the time between the arrival of a new rating, and when it will effect recommendations
- scalability: can the system handle millions or billions of items or users?
- generality: is the system general, or highly fine-tuned for a specific recommendation problem.

Usually recommendation to a user is divided into two phases: generating a prediction on each item the user does not know about, and then picking up the items with the highest predicted value. In the first phase, we talk about prediction error, while in the second phase, we talk about ranking error.

There are performance measures for both the first and the second phase. For prediction performance, these are the most widely accepted measures:

\textbf{RMSE} Root Mean Squared Error, as defined in \cite{132}.

\textbf{MAE} Mean Absolute Error:

\begin{equation}
\text{MAE} = \frac{1}{|V|} \sum_{(u,i) \in V} |\hat{r}_{ui} - r_{ui}|. \tag{2.1}
\end{equation}

For ranking performance, the most important measures are the followings:

\textbf{NDCG} Normalized Discounted Cumulative Gain: This is a rather complex measure.

It is normalized, so its value is between 0 and 1. Not only the highest ranked item matters, but the highest \emph{n} items, that is why it is cumulative. If an item should
2.1. PERFORMANCE MEASURES

be ranked first, but it is third, then this is a higher error, than if an item should
be ranked 10th, but it is ranked 12th. That is why it is discounted. The formal
definition of NDCG is the following \cite{25}: Suppose that user $u$ would like to receive
a recommendation of $K$ items, in the form of a vector $v \in \{1, \ldots, M\}^K$ containing
different item ids. Let $v_k$ denote the $k$-th element of $v$. Then the DCG and NDCG
of $v$ is defined as follows:

$$DCG(v) = \sum_{k=1}^{K} \frac{2^{r_{u,v_k}} - 1}{\log_2(i + 1)}$$  \hspace{1cm} (2.2)

$$NDCG(v) = \max_w \frac{DCG(v)}{DCG(w)}$$  \hspace{1cm} (2.3)

Where $w \in \{1, \ldots, M\}^K$ contains different item ids, and $r_{ui}$ is the users’ preference
(explicit or implicit) on items. To maximize the DCG of a recommendation, the
most preferred items should be ranked first. NDCG is the normalized variant of
DCG: the DCG value is divided by the DCG of the best possible ordering. The
NDCG on a test set could be computed by giving a recommendation to each user
(recommending only items $u$ has not rated), and then averaging the computed
NDCG values.

**relative ranking** This is also a normalized measure, proposed by Koren \cite{27} and Hu
et al. \cite{23}. The recommendation algorithm has to order all the recommendable
items. For each user, the set of items are divided into two parts: relevant and
non-relevant. Assume that the recommendable items (e.g. not rated by the user)
are indexed by $i$ ranging from 1 to $M$. Let $r_{ui} \in \{0, 1\}$ denote whether an item is
relevant to the user ($p_{ui} = 1$) or not ($p_{ui} = 0$). Let $\hat{r}_{ui} \in \mathbb{R}$ denote the prediction
of relevancy, and let the list of items being recommended to user $u$ be ordered by
$\hat{r}_{ui}$. Then the percentile ranking for a relevant $(u, i)$ is defined by:

$$\text{rank}_{ui} = \frac{\sum_{j: (r_{uj} = 0 \land \hat{r}_{uj} > \hat{r}_{ui})} 1}{\sum_{j: r_{uj} = 0} 1}$$  \hspace{1cm} (2.4)

For a test set of implicit ratings, $\text{rank}_{ui}$ is computed for each positive rating, and
then averaged.

The goal is to rank relevant items high, but the order between two relevant items
is not important. The relative ranking measure calculates the average position
of relevant items, and divides this number by the number of all recommendable
items. For example, if we have 100 items, only 3 of them are relevant, and the
ranking algorithm puts them at the 1st, 20th and 54th position, then the relative
ranking performance is $(1 + 20 + 54)/(3 \cdot 100) = 0.25$.

The relative ranking performance measure is more directly related to the user’s
need, than RMSE. However, Koren \cite{27} pointed out, that improving the RMSE yields
improvement on relative ranking performance. Weimer et al. \cite{68} proposed a method to
directly optimize NDCG, which scales up to the Netflix Prize dataset. However, they
concluded that optimizing RMSE instead of NDCG during training yields better NDCG
results on test set. Hu et al. \cite{23} proposed a method for implicit feedback that optimizes
RMSE during training, and then relative ranking performance was used to evaluate on the test set. In their experiments, better training RMSE yielded better
relative ranking on the test set.
CHAPTER 2. LITERATURE SURVEY

From these experiments we may conclude that RMSE is a practically useful performance measure, while being very simple.

2.2 Collaborative filtering methods

As mentioned at the beginning of this section, Breese et al. divide CF methods into two groups: “Memory-based algorithms operate over the entire user database to make predictions. In contrast, model-based collaborative filtering uses the user database to estimate or learn a model, which is then used for predictions.” However, the boundary between these two groups is not clear, for example many authors proposed methods, where they built some model from the data, and then this model was used to assign importance (weight) to parts of the database.

Neighbor-based methods The most commonly used memory-based methods are neighbor-based methods, which can be subdivided into two parts: item-based and user-based neighbor methods.

To predict a given rating, user-based approaches find users who are similar to the active user and have rated the active item, and then calculate from their ratings the prediction for the active (user,item) pair. This approach is very intuitive. Item based approaches are similar, the only difference is that the role of users and items are interchanged.

Assuming the item neighbor method, an unknown matrix element $r_{ui}$ can be estimated as

$$
\hat{r}_{ui} = \frac{\sum_{j:(u,j) \in R} s_{ij} f_{ij}(r_{uj})}{\sum_{j:(u,j) \in R} s_{ij}},
$$

where $s_{ij}$ is the similarity between items $i$ and $j$, and $f_{ij}$ is the function that predicts the rating of the $i$th movie from the rating of the $j$th. The answer of the system is the similarity-weighted average of movie-to-movie sub-predictions.

Model-based methods The most important part of model-based approaches are matrix factorization methods, which will be detailed in the next section. However, there are other important approaches.

Co-clustering based approaches –proposed for CF first by Ungar and Foster– try to cluster both users and items. At the end of training, we have $n$ user clusters, $m$ item clusters, and $n \cdot m$ co-clusters, by pairing all user and item clusters. Ratings falling in the same co-cluster are predicted with the same value.

Double centering is another very simple approach. It tries to predict a rating by

$$
\hat{r}_{ui} = b_u + c_i
$$

where $c_i$ is the item-bias and $b_u$ is the user offset. There are many ways to compute $b_u$ and $c_i$. For example, one can apply incremental (stochastic) gradient descent. Another way is to set $b$ to zero, then compute only $c$, and then compute $b$.

Restricted Boltzmann machines (RBM), a class of two-layer undirected graphical models is another important model-based approach, which was proven to be efficient in the Netflix Prize contest.
2.3 Matrix factorization methods for rating-based collaborative filtering

Matrix factorization is one of the most often applied techniques for CF problems, and it is the most popular method in the Netflix Prize contest. Numerous MF variants have been already published and were validated against the NP dataset as well. In this section I give an overview of MF variants that are known to be effective for rating-based CF (explicit feedback). I remark that incremental gradient descent and alternating least squares MF approaches are the most popular in the Netflix Prize community, since these approaches minimize RMSE in a straightforward way while being effective for problems of such large.

The idea behind MF techniques is very simple. Suppose we want to approximate the matrix $R$ as the product of two smaller matrices:

$$\hat{R} = P Q^T,$$

where $P$ is an $N \times K$ and $Q$ is a $M \times K$ matrix. We call $P$ the user feature matrix and $Q$ the item feature matrix, and $K$ is the number of features (factors) in the given factorization.

If we consider the matrices as linear transformations, the approximation can be interpreted as follows: matrix $Q$ is a transformation from $\mathbb{R}^M$ into $\mathbb{R}^K$, and matrix $P$ is a transformation from $\mathbb{R}^K$ into $\mathbb{R}^N$, and $R$ is a transformation from $\mathbb{R}^M$ into $\mathbb{R}^N$. Typically, $K \ll N$ and $K \ll M$, therefore the intermediate $K$-dimensional vector space acts as a bottleneck in the approximation of $R$ as a sequence of two linear transformations.

In other words, the number of parameters to describe $R$ can be reduced from $|R|$ to $NK + MK$, if $K$ is small. However, we will see examples, when $NK + MK$ is larger than $|R|$, for example, when $K$ is greater than the average number of ratings per user. For the NP problem this is the case when $K > 201$. A good learning algorithm should be able to handle these cases as well, since even for relatively small values of $K$ (for example $K = 40$) there will be many users who have less than $K$ ratings. A technique called “regularization” helps avoiding overfitting by keeping model weights close to zero. Note that $P$ and $Q$ typically contain real numbers, even when $R$ contains only integers.

The above equation can be rewritten in the following form for each element of $R$:

$$\hat{r}_{ui} = p^T_u q_i = \sum_{k=1}^{K} p_{uk} q_{ik},$$

where $p_u$ is the transpose of the $u$-th row of $P$, the user feature vector; $q_i$ is the transpose of the $i$-th row of $Q$, the item feature vector, $p_{uk}$ is the $(u,k)$-th element of $P$, and $q_{ik}$ is the $(i,k)$-th element of $Q$.

**Singular value decomposition (SVD)** The most commonly understood MF method is singular value decomposition, which finds the optimal fixed-rank approximation of a fully observed matrix $R$ in the following form:

$$\hat{R} = U \Sigma V^T$$

Where the optimality means that the Frobenius norm of $R - \hat{R}$ is minimized.
**Definition** Frobenius norm of a matrix $A$ with $N$ rows and $M$ columns is defined as:

$$\|A\|_F = \sqrt{\sum_{i=1}^{N} \sum_{j=1}^{M} a_{ij}^2}$$

The columns of $V$ are orthonormal, the columns of $U$ are orthonormal and $\Sigma$ is a non-negative diagonal matrix.

The problems with singular value decomposition are that it requires a fully observed matrix, which is not the case for explicit feedback, and does not do any regularization. For CF problems, Sarwar et al. [45, 47] applied SVD for explicit feedback CF. They filled missing values with the movie-average. This method is impractical for large datasets like the Netflix dataset. They concluded, that $K = 14$ gave the best results. Later I will remark that many new MF approaches proposed during the NP competition are different: larger $K$ implies better accuracy. I think that sticking at $K = 14$ is due to a lack of regularization.

**Other latent factor models** Matrix factorization approaches are considered as latent factors models, because each user and item gets associated with factors that are discovered by the learning algorithm. Latent factor models have been extensively studied in the field of CF: probabilistic latent semantic analysis [22], restricted Boltzmann machines [43], maximum margin matrix factorization [51].

### 2.3.1 Incremental gradient descent method for matrix factorization

In this section I point out that many variations may exist of incremental gradient descent methods. Basically, these methods iterate through the available ratings, and changes model parameters such that the actual prediction error is decreased. Although at this point the algorithms may seem obvious, it is not the case, which will be demonstrated in the followings. Here is the pseudo-code of one variant:

1. foreach epoch $e$
2.   foreach user $u$
3.     foreach item $i$ rated by $u$
4.       calc $err = R - P Q^T$
5.     foreach feature index $k$
6.       update $p_{uk}$ and $q_{ik}$ to decrease $err_{ui}$
7.   endfor
8. endfor
9. endfor
10. endfor

Later I will provide a more detailed description of the algorithm, but now the focus is on the large number of possible alternatives, which are more or less useful. The pseudo-code can be summarized as follows: it iterates through all users, and then queries all items rated by user $u$. It calculates the error of the prediction of the model (only the error on the $(u, i)$-th rating matters), and then updates all model parameters of the active user and item, namely the $p_{uk}$ and $q_{ik}$ variables for each $k$, to better predict the
2.3. MATRIX FACTORIZATION METHODS FOR RATING-BASED CF

actual rating. The iteration through all the ratings is repeated many times (the number of epochs), until the prediction error on a given test set is decreasing.

Although it seems very straightforward, it contains many decision points:

- Should we iterate over all items, or just items rated by the user? This is very straightforward in this algorithm: use only existing ratings. However, some previous CF algorithm (e.g. [45]) applied the well-known SVD method on the whole R. In that case, the unknown ratings were replaced by a baseline prediction.

- the order of iteration: should we order ratings by (u,i) or by (i,u)? We have found (60) that amongst many possible orderings, iterating over users in an arbitrary order, and for each user, taking the ratings in an increasing chronological order, that is, starting from the oldest and ending with the newest yields the best RMSE. In the followings, I refer to this ordering as the (u,du)-ordering, although the order of users are arbitrary, i.e. they need not to be ordered exactly according to their numeric index, any permutation is sufficient. The (u,du)-ordering is even better than ordering only by (du).

- the point of error calculation: In the above pseudocode, error is calculated before updating the two K-dimensional vector. However, there may be other variants, for example: error is calculated before updating puk or qik for a specific k, or errors are calculated before iterating through all ratings of u (updating in a batch mode).

- updating variables: Even the variable update process can be made more complicated than above: we can collect the changes going to be made to puk and qik, and then update them at some point when the changes are summarized.

Simon Funk’s variant applies the following order of the cycles: (k,e,u,i). He calculated the error in the innermost loop, and updated puk and qik immediately.

The initialization process can be quite different for different variants of the above algorithm. For example, Simon Funk’s variant is fine with initializing P and Q with the same small constant value. However, the above pseudocode is not: it requires at least one of P and Q to be initialized full-rank (it depends on the exact implementation of model parameters update, which we will see later). However, if the error calculation is moved into the innermost loop, this restriction can be dropped.

**Biased regularized incremental simultaneous matrix factorization (BRISMF)**

During the Netflix Prize competition, many variants were proposed which base on incremental (stochastic) gradient descent.

BRISMF is also an MF variant proposed by team Gravity [57]. It is a simple, fast and accurate method. In the followings, I describe this method step-by-step by making small changes to simpler methods (ISMF → RISMF → BRISMF). BRISMF will serve as a starting point for many of my proposed methods, that is why I go into the details.

**Basic incremental gradient descent based MF method (ISMF)** As I mentioned earlier, R contains many unknown ratings which cannot be represented by zero. For
CHAPTER 2. LITERATURE SURVEY

In this case, the approximation of $\mathbf{R}$ can be defined as follows.

$$
\hat{r}_{ui} = \sum_{k=1}^{K} p_{uk} q_{ik} = \mathbf{p}_u^T \mathbf{q}_i 
$$

(2.6)

$$
e_{ui} = r_{ui} - \hat{r}_{ui} \quad \text{for } (u, i) \in \mathcal{R}
$$

(2.7)

$$
l_{ui} = \frac{1}{2} e_{ui}^2
$$

(2.8)

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

(2.9)

$$
L = \frac{1}{2} \text{SSE} = \sum_{(u, i) \in \mathcal{R}} l_{ui}
$$

$$
\text{RMSE} = \sqrt{\text{SSE}/|\mathcal{R}|}
$$

$$
(P^*, Q^*) = \arg \min_{(P, Q)} \text{SSE} = \arg \min_{(P, Q)} L = \arg \min_{(P, Q)} \text{RMSE}
$$

(2.10)

Here (2.6) describes how the predictions are calculated by MF: $\hat{r}_{ui}$ denotes how the $u$-th user would rate the $i$-th item, according to the MF model. In (2.7), $e_{ui}$ denotes the training error measured at the $(u, i)$-th rating. In (2.8), $l_{ui}$ denotes the training loss associated with the $(u, i)$-th rating, which will be important later on. SSE is the sum of square of training errors (prediction errors on training set) for all known ratings. $L$ is the training loss, which is the sum of all training losses measured on each training example. Eq. (2.9) states that the optimal $P$ and $Q$ minimize the sum of squared training errors.

In order to minimize RMSE, which is in this case equivalent to minimizing $L$, ISMF (and RISMF and BRISMF) applies a simple incremental gradient descent method to find a local minimum of RMSE (or $L$), where one gradient step intends to decrease the square of prediction error of only one rating ($e_{ui}^2$, or equivalently $l_{ui}$).

Minimizing RMSE can be seen as a weighted low-rank approximation of $\mathbf{R}$. Weighted low-rank approximations try to minimize the objective function $\text{SSE}_w = \sum_{u=1}^{N} \sum_{i=1}^{M} w_{ui} \cdot e_{ui}^2$, where $w_{ui}$-s are predefined non-negative weights. For collaborative filtering problems, $w_{ui}$ is 1 for known ratings, and 0 for unknown ratings. Srebro and Jaakkola showed that when the rank of $((w_{ui})$) is 1, all local minima are global minima. However, when it is greater than 1—as in the case of collaborative filtering—this statement does not hold any more, which was shown by them via counterexamples. This means that a gradient method can find only a local minimum. However, I pointed out by some experiments that different random initializations of $P$ and $Q$ lead to almost the same result, thus, sticking in a local minima is usually not a problem.

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

We compute the gradient of $l_{ui}$:

$$
\frac{\partial}{\partial \mathbf{p}_u} l_{ui} = -e_{ui} \cdot \mathbf{q}_i, \quad \frac{\partial}{\partial \mathbf{q}_i} l_{ui} = -e_{ui} \cdot \mathbf{p}_u.
$$

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

$$
\mathbf{p}_u' = \mathbf{p}_u + \eta \cdot e_{ui} \cdot \mathbf{q}_i, \quad \mathbf{q}_i' = \mathbf{q}_i + \eta \cdot e_{ui} \cdot \mathbf{p}_u.
$$

(2.11)
2.3. MATRIX FACTORIZATION METHODS FOR RATING-BASED CF

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

When the training has been finished, each value of $R$ can be computed easily using (2.6), even at unknown positions. In other words, the model ($P^*$ and $Q^*$) provides a description of how an arbitrary user would rate any item.

Adding regularization (RISMF) 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 $Q$ does not change, there exists a user feature vector with zero training error. Thus, there is a potential for overfitting, if $\eta$ and the extent of the change in $Q$ are both small. A common way to avoid overfitting is to apply regularization by penalizing the square of the Euclidean norm of weights. This is often used in machine learning methods, for example support vector machines and ridge regression apply that. It is common also in neural networks, where it is termed as weight decay [17]. Penalizing the weights results in a new optimization problem:

$$
\hat{r}_{ui} = \sum_{k=1}^{K} p_{uk} q_{ik} = p_u^T q_i
$$

$$
e_{ui} = r_{ui} - \hat{r}_{ui} \quad \text{for} \ (u, i) \in \mathcal{R}
$$

$$
l_{ui} = \frac{1}{2} \left( e_{ui}^2 + \lambda \cdot p_u^T p_u + \lambda \cdot q_i^T q_i \right)
$$

$$
L = \sum_{(u,i) \in \mathcal{R}} l_{ui} = \frac{1}{2} \sum_{(u,i) \in \mathcal{R}} \left( e_{ui}^2 + \lambda \cdot p_u^T p_u + \lambda \cdot q_i^T q_i \right)
$$

$$(P^*, Q^*) = \arg \min_{(P,Q)} L \quad (2.12)
$$

Here $\lambda \geq 0$ is the regularization factor. Note that minimizing $L$ is no longer equivalent to minimizing SSE, unless $\lambda = 0$, in which case we get back the basic method.

Note that in the above formula of $L$, the penalization of $p_u$ is proportional to $n_u$, the number of items rated by user $u$. Similarly, the penalization of $q_i$ is proportional to $n_i$, the number of ratings on item $i$. This may seem somewhat strange compared to the “traditional” way of regularization, and will be discussed on page 32, after introducing ALS based MF.

For the RISMF approach the gradient of $l_{ui}$ is the following:

$$
\frac{\partial}{\partial p_u} l_{ui} = -e_{ui} \cdot q_i + \lambda \cdot p_u, \quad \frac{\partial}{\partial q_i} l_{ui} = -e_{ui} \cdot p_u + \lambda \cdot q_i. \quad (2.13)
$$

Similarly to ISMF, we update the weights in the direction opposite to the gradient:

$$
p'_u = p_u + \eta \cdot (e_{ui} \cdot q_i - \lambda \cdot p_u), \quad (2.14)
$$

$$
q'_i = q_i + \eta \cdot (e_{ui} \cdot p_u - \lambda \cdot q_i) \quad (2.15)
$$

Besides penalizing the magnitude of weights, ISMF also applies early stopping: when the prediction error on a validation set stops decreasing, it stops the learning procedure. Thus, $P^*$ and $Q^*$ differs from eq. (2.12), because of the early stopping. Note that the matrices are initialized randomly. If both $P$ and $Q$ are initialized with a constant value,
that is, both are rank 1, the weight update will not increase the rank, which is equivalent
to the $K = 1$ case. Random initialization is a simple way to avoid this, allowing matrices
to be full-rank.

Takács et al. [55] observed, that the learning curve of epochs (RMSE on the validation
set as a function of the number of epochs) is first decreasing, and after reaching the
optimum, it starts increasing, regardless of the value of $\lambda$, that is the learning curve has
only one local minimum, which is also the global minimum (this is not to be confused
with that there are many local minima of $P$ and $Q$: only the number of epochs has
a global optimum, given the above training method). This experimental observation
justifies the usage of early stopping, which is also a kind of model regularization besides
the penalization of weights.

Takács et al. [55] pointed out that RISMF differs from Funk’s MF in a few important
aspects. It updates each feature simultaneously and initialize the matrix randomly.
Simon Funk’s approach learns each feature separately for a certain number of epochs.
His approach converges slower than ISMF (RISMF), and it needs to iterate over $R$
more times. Both methods use regularization and early stopping to prevent overfitting.
Another important difference is the ordering of ratings: $(u, i)$ for Funk’s MF, and $(u, d_{ui})$
in RISMF. However, this is only important for datasets where the test ratings are newer
than the training ratings, like in the case of the Netflix dataset. For such datasets where
the date is missing, BRISMF iterates in a random order.

**Adding biases: BRISMF** One may argue that some users tend to rate all items
higher or lower than the average. The same may hold for items: some items can be
very popular. Although MF can reconstruct the original matrix exactly when $K$ is large
enough and $\lambda = 0$, this is not the case when overfitting is to be avoided. There is a
straightforward way to extend RISMF to be able to directly model this phenomenon,
by extending MF with biases for users and items, which leads to BRISMF.

The bias feature idea in the context of matrix factorization was mentioned first by
Paterek [32]. Many other authors have also used some method to account for biases. For
example, Sarwar et al. [45] subtracted user average before training. Simon Funk applied
double centering, first by subtracting the movie average, then the user average of the
residuals. The advantage of BRISMF and Paterek’s approach is that these parameters
are trained with other model parameters simultaneously.

BRISMF is introduced in [57]. It extends RISMF by adding biases. This can be
considered in two different but mathematically equivalent ways. The first way fixes the
first column of $P$ and the second column of $Q$ to the constant value of 1, that is, they
are initialized to 1 and not updated. The second way is to introduce two new vectors,$b \in \mathbb{R}^{N \times 1}$ and $c \in \mathbb{R}^{M \times 1}$, and modify the formula of predictions:

$$ r_{ui} = b_u + c_i + p_u^T q_i, $$

where $c_i$ is the item bias (related to the average rating of item), and $b_u$ is the user bias
by which a user offsets his ratings for all movies. In the followings, I use this latter
version, as it is more intuitive.

Paterek’s variant and BRISMF share some common features, but there are important
differences which makes BRISMF better:

- **BRISMF** updates all features simultaneously, while Paterek followed Simon Funk’s
  approach to update feature weights.
Both methods have biases. However, in Paterek’s article it is not clear when the biases and when the features are trained. BRISMF trains all variables of the active prediction (biases and features) simultaneously.

BRISMF orders ratings by users, and then by date, while Paterek does not give any details about the ordering of ratings.

This simple extension speeds up the training phase and yields a more accurate model with better generalization performance. Since BRISMF is always superior to RISMF in terms of both the accuracy and the running time, it is our recommended basic MF method.

The task of finding an appropriate $P$, $Q$, $b$ and $c$ can be defined as an optimization problem:

$$
e_{ui} = r_{ui} - \hat{r}_{ui} \text{ for } (u, i) \in R$$

$$l_{ui} = (e_{ui}^2 + \lambda p_u^T p_u + \lambda q_i^T q_i + \lambda b_u^2 + \lambda c_i^2)/2$$

$$L = \sum_{(u, i) \in R} l_{ui}$$

$$(P^*, Q^*, b^*, c^*) = \arg \min_{P, Q, b, c} L$$

(2.16)

The gradient of $l_{ui}$ is the following:

$$\frac{\partial}{\partial p_u} l_{ui} = -e_{ui} \cdot q_i + \lambda p_u,$$  \hspace{1cm} (2.17)

$$\frac{\partial}{\partial q_i} l_{ui} = -e_{ui} \cdot p_u + \lambda q_i,$$  \hspace{1cm} (2.18)

$$\frac{\partial}{\partial b_u} l_{ui} = -e_{ui} + \lambda b_u,$$  \hspace{1cm} (2.19)

$$\frac{\partial}{\partial c_i} l_{ui} = -e_{ui} + \lambda c_i.$$  \hspace{1cm} (2.20)

Weight must be updated in the direction opposite to the gradient:

$$p_u' = p_u + \eta \cdot (e_{ui} \cdot q_i - \lambda p_u),$$

$$q_i' = q_i + \eta \cdot (e_{ui} \cdot p_u - \lambda q_i),$$

$$b_u' = b_u + \eta \cdot (e_{ui} - \lambda b_u),$$

$$c_i' = c_i + \eta \cdot (e_{ui} - \lambda c_i).$$  \hspace{1cm} (2.21)

For the complete training algorithm, see Algorithm 1.

### 2.3.2 Alternating least squares (ALS) for matrix factorization

Alternating least squares is a commonly used method in the field of Machine Learning to factorize matrices. It can be summarized as follows: first it initializes $Q$ randomly, fixes that, and computes $P$ using ridge regression (RR). Then it fixes $P$, and computes $Q$ using RR. These two steps are repeated until convergence, thus the algorithm alternates between recomputing $P$ and $Q$. ALS has been proposed for the Netflix Prize problem by team BellKor [6].
**Input:** \( R \): dataset, 
\( \eta \): learning rate, 
\( \lambda \): regularization factor, 
\( K \): number of features.

**Output:** \( P^*, Q^*, b^*, c^* \): the user and item feature matrices and biases, 
\( n^* \): the optimal number of epochs

1. Partition \( R \) into two sets: \( T \) (training set), \( V \) (validation set).
2. Initialize \( P, Q, b^* \) and \( c^* \) with small random numbers.
3. \( n = 0 \)

**loop** until the terminal condition is met. One epoch:

4. \( n = n + 1 \)

5. for each \( (u, i) \) element of \( T \) ordered by \( (u, d_{ui}) \):

6. \[ \hat{r}_{ui} = b_u + c_i + \sum_{k=1}^{K} p_{uk} q_{ik}; \]

7. \[ e_{ui} = r_{ui} - \hat{r}_{ui}; \]

8. /* update the variables according to eqs. (2.17)–(2.20) */

9. for each \( k \in \{1, \ldots, K\} \)

10. \[ p'_{uk} = p_{uk} + \eta \cdot (e_{ui} \cdot q_{ik} - \lambda p_{uk}); \]

11. \[ q'_{ik} = q_{ik} + \eta \cdot (e_{ui} \cdot p_{uk} - \lambda q_{ik}); \]

12. \[ p_{uk} = p'_{uk}; \]

13. \[ q_{ik} = q'_{ik}; \]

14. end

15. \[ b_u = b_u + \eta \cdot (e_{ui} - \lambda b_u); \]

16. \[ c_i = c_i + \eta \cdot (e_{ui} - \lambda c_i); \]

17. end

18. calculate the RMSE on \( V \), using eq. (1.3);

19. if the RMSE on \( V \) was better than in any previous epoch:

20. Let \( P^* = P, Q^* = Q, b^* = b, c^* = c \) and \( n^* = n \);

21. end

22. terminal condition: RMSE on \( V \) does not decrease.

23. end

**Algorithm 1**: Training algorithm for BRISMF
The recomputation of $P$ is performed by solving a separate least squares problem for each user. We can rewrite the following overall cost function,

$$
\sum_{(u,i) \in \mathcal{R}} (r_{ui} - p_u^T q_i)^2 + \lambda p_u^T p_u + \lambda q_i^T q_i,
$$

(2.22)

into a different form:

$$
N \sum_{u=1}^N \left( \sum_{i:(u,i) \in \mathcal{R}} (r_{ui} - p_u^T q_i)^2 + \lambda p_u^T p_u + \lambda q_i^T q_i \right).
$$

(2.23)

The outer sum is taken over each user. If $q_i$-s are fixed, then only $p_u$-s are to be optimized, which can be done separately for each user. For the $u$-th user, this leads to the following equivalent cost function:

$$
\left( \sum_{i:(u,i) \in \mathcal{R}} (r_{ui} - p_u^T q_i)^2 \right) + n_u \lambda p_u^T p_u,
$$

(2.24)

where $n_u$ is the number of ratings of user $u$, i.e. $n_u = \sum_{i:(u,i) \in \mathcal{R}} 1$. Note that the constant term $\lambda q_i^T q_i$ is removed. This is a least squares problem without any linear inequality constraints, which can be solved by ridge regression as follows: using the notations of [6], let the matrix $Q[u] \in \mathbb{R}^{n_u \times K}$ denote the restriction of $Q$ to the items rated by user $u$, the vector $r_u \in \mathbb{R}^{n_u \times 1}$ denote the ratings given by the $u$-th user to the corresponding items, and let

$$
A_u = Q[u]^T Q[u] = \sum_{i:(u,i) \in \mathcal{R}} q_i q_i^T,
$$

$$
d_u = Q[u]^T r_u = \sum_{i:(u,i) \in \mathcal{R}} r_{ui} q_i.
$$

(2.25)

Then ridge regression recomputes $p_u$ as

$$
p_u = (\lambda n_u I + A_u)^{-1} d_u.
$$

(2.26)

After recomputing all $p_u$, the algorithm recomputes all $q_i$, using the above formulas, but interchanging the role of users and items.

In the recomputation of $P$, two operations are dominant:
1. the computation of $A_u$ requires $O(K^2 \cdot n_u)$ time, and
2. the inversion that requires $O(K^3)$ time.

The recomputation of $p_u$ requires $O(K^2 \cdot n_u + K^3)$ time. Thus, the recomputation of $P$ requires $O(\sum_{u=1}^N (K^2 \cdot n_u + K^3))$, which is equal to $O(K^2 \cdot |\mathcal{R}| + N \cdot K^3)$, as it has been noted by Takács et al. [58], Hu et al. [23]. Similarly, the recomputation of $Q$ requires $O(K^2 \cdot |\mathcal{R}| + M \cdot K^3)$. According to [6, 23], the number of recomputations needed ranges between 10 and a "few tens".

Bell and Koren [6] also proposed a non-negative variant of ALS ("to further avoid overfitting"), where the least squares problem is constrained such that the resulting $p_u$ and $q_i$ must be non-negative.

It has been pointed out that larger $K$ yields more accurate predictions both in BRISMF [55, 58] and in ALS [23, 70], using this form of regularization.
Different formulas for regularization In many ridge regression problems, the regularization term (in the above formulas: $\lambda p_u^T p_u$) does not depend on the number of examples, while in the above formulas it does, as it is added with each rating. However, many “traditional” regression problems are just separate problems, where the number of examples is fixed, and the optimal regularization is usually determined by trial-and-error. In case of CF, many regression tasks have to be solved simultaneously. Zhou et al. \cite{70} investigated the “traditional” way of regularization, i.e. they used the following cost function for recomputing $p_u$:

$$
\left( \sum_{i \in \{u,i\} \in R} (r_{ui} - p_u^T q_i)^2 \right) + \lambda p_u^T p_u.
$$

(2.27)

They concluded, that the “traditional” variant leads to poor result. I remark, that the non-traditional cost function was explicitly proposed for the NP problem first by Salakhutdinov et al. \cite{43}. Although Simon Funk proposed his update formulas sooner, there is a lack of the definition of cost function.

I experimented with ALS based MF on the Netflix Prize dataset, using different forms of regularization: where it is proportional to $n_u$, $\sqrt{n_u}$ or it is a constant term.

When using a constant term for regularization, as I increased $K$ from 10 to 20 and 50, the accuracy deteriorated. When using the $\sqrt{n_u}$ variant, the accuracy improved. When using the $n_u$ variant, the accuracy improved, and the improvements were larger. With a proper regularization we can expect that by increasing $K$ the performance not worsens.

Possible explanations for the success of this special form of regularization are as follows:

- If we mix the ratings of two random users with the same number of ratings, then we need to double the regularization for the doubled-user. Users have a long history of ratings, and their taste changes over time. We may consider a user with a 2-year long rating history as a mix of two users with 1-1 year long histories. If in each year the users give roughly the same number of ratings (while she is a Netflix subscriber), then the regularization needs to be multiplied by the number of years, or equivalently, by the number of ratings of the user. Even though the above speculation is over-simplified, we may assume, that after watching –let’s say– 10 ratings (no matter how much time elapses between them), the user’s taste ($p_u$) changes by an important amount. That can explain why the above form of regularization is useful for users.

- For gradient based methods, the $n_u$ variant provides the simplest formula. It is like weight decay in neural networks: after each training step, we apply the same weight decay. If the regularization term does not depend on $n_u$, then for each gradient step, it must be divided by $n_u$. This would result in very different step-sizes of the gradient learning. For example, consider a user $u$ with only 1 rating, and another user $v$ with 200 ratings. Each regularization step of $u$ would be 200 times larger than $v$’s. This may call for small learning rates, which would result in slow convergence.

- With the $n_u$ variant, the difference in the norm of user feature vectors will be smaller between rare and frequent raters, than with the constant term variant. Thus, the rare-raters have more influence on the item feature vectors, which will
end up in better user feature vectors for rare-raters (where actually the majority of users belong to). Recall that in the Netflix Prize dataset each user has roughly the same number of ratings in the test set, which is special amongst CF datasets.

### 2.3.3 NSVD1 for matrix factorization

NSVD1 was proposed by Paterek [32] for the Netflix Prize contest. There is no resolution of the acronym. The idea is to replace the usual prediction rule \( r_{ui} = b_u + c_i + \mathbf{p}_u^T \mathbf{q}_i \) with the following:

\[
r_{ui} = b_u + c_i + \mathbf{p}_u^T \mathbf{q}_i, \quad \text{where} \quad \mathbf{p}_u = n_u^{-0.5} \sum_{j: (u,j) \in \mathcal{R}} \mathbf{w}_j
\]

(2.28)

In this way, \( \mathbf{p}_u \) is not a free variable anymore: we have a second set of item feature vectors (\( \mathbf{w}_j \)), and \( \mathbf{p}_u \) depends on what movies the user rated, but not on how they are rated. Thus, the free parameters of the model are: two sets of item feature vectors (\( \mathbf{q}_i \) and \( \mathbf{w}_j \)) and user and item biases (\( b_u \) and \( c_i \)). The rationale behind this formula is the following: if \( u \) watches a movie, this gives information about the user’s taste, although this information is not as valuable as an explicit rating. The information is captured by the \( \mathbf{w}_j \) variables. In practice, this approach works, which means that the assumption that users randomly choose items for making explicit ratings does not hold.

Paterek described only the prediction formula, but not the training algorithm. However, the naive implementation of gradient descent is very ineffective for this model [57]. If all \( \mathbf{w}_j \)-s are updated at each training example, then the training will be very slow [57]. Takács et al. [57] suggested an efficient training algorithm for NSVD1, the complete details are described in Algorithm 2. I will rewrite the algorithm in a more general form in Section 5.1. The training algorithm is similar to BRISMF’s, and it operates on a \((u, d_{ui})\)-ordered database: for user \( u \), it computes \( \mathbf{p}_u \). Then this user feature vector \( \mathbf{p}_u \) is updated with each rating of \( u \) as it were normal (just like in BRISMF). After processing all rating of \( u \), the algorithm distributes the change of \( \mathbf{p}_u \) amongst the \( \mathbf{w}_j \) vectors, and forgets \( \mathbf{p}_u \).

### 2.4 Content-based filtering methods

Pure content-based filtering (CBF) methods try to recommend items to users based only on the available metadata about items (and sometimes about users). This means that a different classifier / regression model is built for each user separately [1]. Adomavicius and Tuzhilin [1] gives a survey about content-based filtering methods: most of the content-based filtering methods are applied on textual data, and therefore they have roots in text mining. For example, various text classification algorithms and document similarity calculation formulas exist that have been explored for news or web page recommendation.

The Netflix Prize contest pointed out that when we have accurate algorithms and plenty of ratings for movies, then movie metadata cannot provide additional information about movies [8]. An interesting question is whether movie metadata is helpful in the prediction of new or rarely rated movies.
In the followings I survey some CBF-related articles. Most of the CBF methods are pure CBF methods, where the problem is which one of the plenty of machine learning algorithms to choose. For example, Pazzani and Billsus\cite{33} surveys CBF methods by the underlying machine learning methods, which includes decision trees, nearest neighbour classification, Rocchio’s method, linear classifiers, naive Bayes.

Melville et al.\cite{31} suggested the following CF+CBF approach: take the matrix of ratings, apply a pure CBF method and fill the missing ratings with that method. Then, based on the fully-filled matrix, apply Pearson-correlation based user-neighbour approach, with the following modifications: the active user is neighbor of itself (thus the prediction of the content-based approach serves as a baseline); users with few ratings or few common ratings are given lower weight, and the Pearson-correlation is measured on the original matrix. The idea can be summarized as follows: the principal assumption is that CBF will make similar prediction errors on similar users. Based on the measured errors on similar users, we can estimate the prediction error on the active user, and offset the prediction by this term. They evaluated on the EachMovie dataset using MAE metric. First, they evaluated their approach on new movies (the first rater problem). The MAE of a pure CBF approach and their approach was 1.060 and 1.023 resp. Then they evaluated their approach using regular train/test splitting. They concluded that their CF+CBF approach outperforms both pure CF and pure CBF. The MAE values were 0.962, 1.002 and 1.059 respectively. However, user-neighbor methods can be very inaccurate, and in the Netflix Prize competition it turned out that it is easy to obtain good results by blending two not so accurate but complementary predictors. On the other hand, it is hard to obtain good results by blending a very accurate and an inaccurate method. Since their approach is user-neighbor based and needs to fill the whole matrix of ratings, it is impractical for the Netflix Prize dataset.

Vozalis and Margaritis\cite{66} presented an MF approach that incorporates demographic information and ratings to enhance plain CF algorithms. They apply SVD on the matrix.

Algorithm 2: Training algorithm for NSVD1
2.5. NEW CF METHODS FROM THE NETFLIX PRIZE CONTEST

of demographic data about users, and on the matrix of metadata about items. The feature vectors extracted by the first SVD can be used to calculate similarity between users (and the 2nd SVD for items). They also run SVD on the matrix of ratings, where missing ratings are filled with some baseline prediction. Then they apply two different neighbor methods on top of these SVDs, one is based on the similarities computed by the demographic/metadata SVD, the other is based on SVD of ratings. The predictions of these neighbor methods are then combined.

Basilico et al. suggested the following formula for prediction (rewritten for better readability, [4]):

\[
\hat{r}_{ui} = \sum_{(v,j) \in R} \alpha_{vj} \cdot \text{sim}_u^{(1)} \cdot \text{sim}_{ij}^{(2)},
\]

where \(\text{sim}_u^{(1)}\) and \(\text{sim}_{ij}^{(2)}\) are the predefined user and movie similarity measures that tell how similar are the metadata of two users or items; here \(\alpha_{vj}\) is to be optimized. One prediction requires \(O(|R|)\) time, which makes this approach unpractical.

The popular TiV o system also applies both CF and CBF approaches [2]. They extend the list of recommended movies by CF with movies recommended by a CBF algorithm. They also conclude that in general, CF provides better movie recommendations than CBF.

2.5 New CF methods from the Netflix Prize contest

In the Netflix Prize contest, many top teams published their approaches. The rules of the contest also required that the winner of the Grand Prize and the Progress Prizes must publish its solution. During the 3 years of the contest, there was a clear change of scope:

**First year** The goal is to cope with the huge dataset (100 times larger than Movielens-1M). Invention of new matrix factorization approaches [67, 69, 52, 55, 43]. Restricted Boltzmann Machines [43] and new neighbor methods [7, 8]. Invention of asymmetric factorization methods (NSVD1, NSVD2) [32]. Application of linear blending and few basic non-linear blending approaches [55, 32, 10]. Running one method on the residual of other method [10].

**Second year** Further enhancement of matrix factorization methods (using biases, regularization) [57, 54, 58, 27], asymmetric factor models [27]. Invention of hybrid methods: sum the output of two different methods, and train them simultaneously [57, 27, 11]. Invention of a new fast neighbor method which operates on MF result [57]. Exhaustive investigation of factorized neighborhood methods [27, 63]. Methods for exploiting the time of ratings: ordering the examples by date [57, 58], introduction of a parameter that reflects user’s mood on a given day [33], introduction of a vector that reflects user’s mood on a given day [11, 28].

**Third year** Further enhancement of hybrid methods [33]. Sophisticated non-linear blending techniques [48, 35, 26, 54].
A clear conclusion of the Netflix Prize contest is that the best RMSEs can be achieved by excessive blending of many predictors made by many people. For example, the first Progress Prize of the contest was won by 3 people and more than 100 predictors. The second Progress Prize was won by 5 people and \( \sim 200 \) predictors. The Grand Prize was won by 7 people and more than 600 predictors.

2.6 Datasets

In Takács et al. [60] we give a thorough description about the Netflix Prize dataset, which I adopt in this section. From now on, unless not mentioned otherwise explicitly, the reported RMSE results are on the Netflix Prize dataset.

2.6.1 The Netflix Prize dataset

Netflix generously provided a dataset for the Netflix Prize competition which is 100 times larger than any previous dataset. The database contains 100,480,507 ratings made by 480,189 users on 17,770 ratings. Each rating is a \((u, i, r_{ui}, d_{ui})\) quadruple representing that user \(u\) rated item \(i\) as \(r_{ui}\) on date \(d_{ui}\), where \(d_{ui} \in D\) the ordered set of possible dates. The ratings \(r_{ui}\) are integers from 1 to 5, where 1 is the worst, and 5 is the best. Besides the ratings, the date of the ratings and the title and release year of movies are provided. The data were collected between October, 1998 and December, 2005 and reflect the distribution of all ratings received by Netflix during this period [12]. The collected data was released in a train-test setting in the following manner (see also Figure 2.1).

Netflix selected a random subset of users from their entire customer base with at least 20 ratings in the given period. A Hold-out set was created from the 9 most recent ratings of the users, consisting of about 4.2 million ratings. The remaining data formed the Training set. The ratings of the Hold-out set were split randomly with equal probability into three subsets of equal size: Quiz, Test and Probe. The Probe set was added to the Training set and was released with ratings. The ratings of the Quiz and Test sets were withheld as a Qualifying set to evaluate competitors. The Quiz/Test split of the Qualifying set was unknown to the public during the competition. I remark that the date based partition of the entire NP dataset into train-test sets reflects the original aim of recommender systems, which is the prediction of future interest of users from their past ratings/activities.

As the aim of the competition was to improve the prediction accuracy of user ratings, Netflix adopted RMSE as evaluation measure. The goal of the competition was to reduce by at least 10 percent the RMSE on the Test set, relative to the RMSE achieved by Cinematch. The contestants had to submit predictions for the Qualifying set. The organizers returned the RMSE of the submissions on the Quiz set, which was also reported on a public leaderboard [12].

There are some interesting characteristics of the data and the set-up of the competition that pose a difficult challenge for prediction:

- The distribution over the time of the ratings of the Hold-out set is quite different from the Training set. As a consequence of the selection method, the Hold-out set contains a significantly different time distribution compared to the Training set.

[http://www.netflixprize.com/leaderboard](http://www.netflixprize.com/leaderboard)
2.6. DATASETS

![Diagram of data split and naming convention](image)

Figure 2.1: The train-test split and the naming convention of Netflix Prize dataset, after Bell et al. [5]

does not reflect the skewness of the movie-per-user, observed in the much larger Training set. Therefore the Qualifying set contains approximately equal number of queries for often and rarely rating users.

- The designated aim of the release of the Probe set is to facilitate unbiased estimation of RMSE for the Quiz/Test sets despite of the different distributions of the Training and the Hold-out sets. In addition, it permits off-line comparison of predictors before submission.

- Users’ activity at rating is skewed. To put this into numbers, ten percent of users rated 16 or fewer movies and one quarter rated 36 or fewer. The median is 93. Some very active users rated more than 10,000 movies. A similar biased property can be observed for movies: The most-rated movie, Miss Congeniality was rated by almost every second user, but a quarter of titles were rated fewer than 190 times, and a handful were rated fewer than 10 times [5].

- The variance of movie ratings is also very different. Some movies are rated approximately equally by the user base (typically well), and some partition the users. The latter ones may be more informative in predicting the taste of individual users.

In the experiments I evaluate the presented methods on a randomly selected 10% subset of the Probe set, which is termed as Probe10 [57]. Unless I explicitly mention, from now on the RMSE values refer to the Probe10 RMSE. During the competition the Probe10 had a great advantage: the rules of Netflix Prize competition allowed only 1 submission daily, which limited the number of the Quiz RMSE calculations, and typically the difference between the Probe10 and Quiz RMSE values was < 0.0003, and
only sometimes it was 0.0010. Now that the contest is over, and the qualifying
answers are available, the importance of Probe10 is smaller.

When Quiz RMSE values are reported I also mention the percentage of improvement
over Cinematch (IoC).

I performed tests on two different configurations:

- an average single processor laptop, with a 2 GHz Intel Pentium M (Dothan) CPU.
- an Intel Core 2 Quad Q9300 CPU, at 3.3GHz, using only 1 core.

When I report running times, I specify which one of the configurations is used.

2.6.2 The MovieLens dataset

The 1M MovieLens dataset contains cca. 1 million ratings from 6,040 users on 3,900
movies. As in the case of the Netflix dataset, ratings are made on a 5 star scale, and the
rating records are also quadruples containing the timestamp of the rating. Demographic
data provided with the ratings are not used in my settings. Since there is no standard
train-test split of the dataset, I applied a simple random split to generate a 90%–10%
train-test setting.

2.6.3 The Jester dataset

The Jester dataset contains 4,136,360 ratings from 73,421 users on 100 jokes. Users
rated jokes on the continuous $[-10, +10]$ range. Ten percent of the jokes (called the
gauge set, which users were asked to rate) are densely rated, others, more sparsely.
Two thirds of the users have rated at least 36 jokes, and the remaining ones have
rated between 15 and 35 jokes. The average number of ratings per user is 46, so it
is a particularly dense dataset compared to NP and MovieLens. Goldberg et al. created their train-test split by a random division of a subset of 18,000 users into two
disjoint sets. For my experiments this split is inappropriate since the my goal is to give
predictions for users who have already provided ratings. Therefore, here I also use a
random 90%–10% train-test setting.

We compared BRISMF with other algorithms found in the literate on all of the
above 3 datasets, and have found that it is favorable both in terms of running time and
accuracy.

2.6.4 Analysis of statistical significance of results

An important question about test results is whether a method with better RMSE is
really better or only by chance. In other words: is it worth to measure the RMSE by 4
digits, or only 3 digits should be used?

I assume the prediction errors have normal distribution. In the followings, assume
the Probe10 set of the Netflix dataset as a test set, and denote it by $V$. Let $e_{ui} = r_{ui} - \hat{r}_{ui}$
denote the error of the $(u, i)$-th prediction. Calculating the RMSE of predictions is an
estimation of the standard deviation $\sigma$ of $e_{ui}$, assuming i.i.d. sampled ratings.

\footnote{Available at \url{http://archive.ics.uci.edu/ml/machine-learning-databases/netflix/}}
\footnote{Available at \url{http://www.grouplens.org/node/73}}
\footnote{Available at \url{http://goldberg.berkeley.edu/jester-data/}}
Let \( \text{SSE} \) denote the sum of square of prediction errors: \( \text{SSE} = \sum_{(u,i) \in V} c_{ui}^2 \), and let \( n = |V| = 140840 \) denote the number of examples in the test set. Then the following holds \([24, p.351]\):  
\[
P \left( \frac{\text{SSE}}{\chi_{1-\alpha/2}^2(n-1)} < \sigma^2 < \frac{\text{SSE}}{\chi_{\alpha/2}^2(n-1)} \right) = 1 - \alpha
\]
where \( \chi_p^2(n-1) \) is the \( p \)-th quantile of the chi-square distribution with degrees of freedom \( n-1 \).

If we set \( \alpha \) to 0.01, and \( \text{SSE} \) is exactly 140840, i.e. RMSE is exactly 1, then:

\[
P(0.9903 < \sigma^2 < 1.0098) = 0.99.
\]

Taking the square root it yields:

\[
P(0.9952 < \sigma < 1.0049) = 0.99.
\]
In other words: if RMSE is 1 on Probe10, then we can be 99% sure that the standard deviation of the predictor is between 0.9952 and 1.0049, which is a huge difference.

Next, we examined whether an improvement on one test set turns into an improvement on another test set. We performed a thorough analysis to check how reliable the Probe10 results are. For this, we will show that if a given method has better RMSE compared to another method for a particular subset of the Probe set, then it has the same performance gain on other subsets of the Probe set. To do this, we partitioned the Probe set into 10 subsets, and we ran 10 different methods using them. Therefore, in total we had 100 runs. We denote the test RMSE of the \( x \)-th method on the \( y \)-th test set by \( m_{xy} \). To summarize the results, we calculated the average test RMSE for each method, denoted by \((\overline{m}_1, \ldots, \overline{m}_{10})\), and a difficulty offset for each test set, denoted by \((o_1, \ldots, o_{10})\), defined as

\[
o_y = \frac{1}{10} \sum_{x=1}^{10} m_{xy} - \overline{m}_x.
\]

The test RMSE of the 100 runs are approximated as \( m_{xy} = \overline{m}_x + o_y \). The standard deviation of this approximation is 0.000224 RMSE score, and the maximal deviation is less than 0.0007 RMSE score, which means that \( m_{xy} \) is well approximated, thus we can assign a difficulty offset to each test set. Consequently, Probe10 results are reliable: although different test sets can yield highly different results, but the relation between different methods are preserved: if a method is better on one test set, it will be usually better on another test set. The only thing not advised is to compare different methods on different test sets. In that case, the pessimistic estimations of statistical significance analysis should be applied. On the other hand, if we are comparing different methods, then the reliability of the difference of the RMSEs is of interest.

Conclusion: it is worth to measure the RMSE by 4 digits. 3 are too few, 5 are too much.
Chapter 3

Improving on the accuracy of matrix factorization

3.1 Improving the performance of a single model

All of the presented methods have many predefined parameters that can greatly influence the accuracy. Sometimes a few experiments are enough to set them well, sometimes we need to apply parameter optimization to find the best settings.

In this section I propose a very simple parameter optimization algorithm which is able to optimize the parameters of the matrix factorization method. Moreover, I propose the introduction of different learning rates and regularization factors for the matrix factorization model. In this way the number of optimizable parameters are increased. However, this modification allows to boost the prediction accuracy of the matrix factorization model, which I will demonstrate experimentally.

I proposed the introduction of many parameters in [55, 56, 58]. I described the parameter optimization in [58].

3.1.1 Parameter optimization

Algorithm describes the proposed simple algorithm for parameter optimization.

The algorithm takes another algorithm, the MF training algorithm, as an input parameter. The training algorithm takes the learning parameters as input, and outputs the RMSE of the generated model on the validation set. For example, the training algorithm could be BRISMF, the two parameters are the learning rate and regularization factor.

At one iteration the parameter optimization algorithm chooses one variable to optimize: it evaluates the training algorithm $n$ times, assigns $n$ different random values to that parameter. Then it picks up the best values, and keeps the parameter optimization going on. Usually, iteration is stopped manually (this parameter optimization algorithm is an anytime algorithm).

Note that after each iteration, the performance is not worse than after the previous iteration. However, the algorithm is highly prone to stuck in local minima.

For implementations, it is very important to use the random seed variable: if for some purposes we should stop the training procedure and then rerun, the algorithm will
**Input:** $L$: number of parameters, 
$A(p_1, \ldots, p_L) \rightarrow \mathbb{R}$: the training algorithm as a callable function, 
$P_1, \ldots, P_L$: set of allowed values for each parameter, 
$n$: number of evaluations in each iteration, 
$S$: random seed

**Output:** $p_1, \ldots, p_L$: the found parameter values

1. Initialize a pseudo random number generator with the seed $S$.
2. Randomly initialize parameters $p_1, \ldots, p_L$.
3. loop forever (iteration is stopped manually).
4. Randomly choose $l$ such that $1 \leq l \leq L$.
5. /* $p_l$ will be optimized */
6. Randomly choose $n$ different elements of $P_l$:
7. $u_1 \in P_1, \ldots, u_n \in P_l$
8. $u_0 = p_l$ /* backup the current value of the $p_l$ */
9. for each $i \in \{0, 1, \ldots, n\}$:
10. $p_l = u_i$
11. /* run the training algorithm and evaluate it on the validation set: */
12. $a_i = A(p_1, \ldots, p_L)$
13. end
14. $j = \arg\min_{i=0}^L a_i$
15. $p_l = u_j$ /* Assign the best value to $p_l$. */
16. end

**Algorithm 3:** Simple parameter optimization algorithm
run exactly in the same way. If we cache the results of previous evaluations on hard disk, we can continue the process from the last stop.

3.1.2 Introduction of parameters

In case of BRISMF, I introduced many parameters, namely:

- different learning rate and regularization factor
  - for user and item feature vector ($\eta^p, \eta^q, \lambda^p, \lambda^q$);
  - for user and item bias ($\eta^{pb}, \eta^{qb}, \lambda^{pb}, \lambda^{qb}$);
- minimum and maximum weights in the uniform random initialization of $P$ and $Q$: $w^p, w^q, w^u, w^t$;

In this way, the update rules of BRISMF (equation (2.21) on page 29) becomes the following:

$$
\begin{align*}
\mathbf{p}'_u &= \mathbf{p}_u + \eta^p \cdot (e_{ui} \cdot \mathbf{q}_i - \lambda^p \cdot \mathbf{p}_u), \\
\mathbf{q}'_i &= \mathbf{q}_i + \eta^q \cdot (e_{ui} \cdot \mathbf{p}_u - \lambda^q \cdot \mathbf{q}_i), \\
b_u &= b_u + \eta^{pb} \cdot (e_{ui} - \lambda^{pb} b_u), \\
c_i &= c_i + \eta^{qb} \cdot (e_{ui} - \lambda^{qb} c_i)
\end{align*}
$$

Note that while the original formulation of BRISMF had 2 parameters, now it has 8 parameters (plus the parameters of weight initialization).

**Speeding up parameter optimization** I subsample the matrix $R$ for faster evaluation of parameter settings. I have experienced that movie-subsampling substantially increased the error, in contrast to user-subsampling, thus I did not perform the former. A possible reason for this is that in the Netflix prize dataset movies have much more ratings than users: users have 209 ratings on average, while movies have 5600 ratings on average. If we drop half of the movies, then an average user will have 105 instead of 209 ratings. On the other hand, if we drop half of the users, an average movie will have 2800 instead of 5600 ratings, which is much less loss of information: 2800 ratings are still enough to get an accurate model of the movie, each doubling of the number ratings brings less information than the previous doubling.

Interestingly, the larger the subsample is, the fewer iterations are required to achieve the optimal MF model. This is examined in Section 3.1.3. I pointed out that larger $K$ yields better RMSE [55, 58] (page 31). Thus, it is straightforward to use smaller $K$ values for faster evaluation, because it does not change the number of epochs substantially. We have two ways to speed up the parameter optimization process: subsampling users, and using small $K$ values.

3.1.3 Evaluation

In the following, I use a baseline BRISMF instance, termed as BRISMF#1, with the following parameter settings: $K = 40, \eta = 0.007, \lambda = 0.005, w^p = w^q = -w^u = -w^t = -0.01$. BRISMF#1 reaches its optimal Probe10 RMSE in the 10th epoch: 0.9056.

**Subsampling users**

On Figure 3.1 I demonstrate how the number of users (thus, the number of ratings) influences RMSE and the optimal number of training epochs in case of BRISMF#1.
RMSE varies between 0.9056 and 0.9677, and the number of epochs between 10 and 26. The smaller the subset of users used for training and testing, the larger the RMSE and the number of epochs. This means that the time-complexity of MF is sublinear in the number of ratings (see Section 3.1.1).

This experiment indicates, that subsampling the users will not speed up the parameter optimization process linearly, due to the slightly increased number of epochs, but the speedup is near-linear.

I remark that the number of ratings is proportional to the number of users; the ratio of them is the average number of ratings per user. This implies also that the time-complexity of MF is sublinear in the number of ratings. A possible explanation for the decreasing number of epochs is that the data set is redundant: from the viewpoint of movie features, iterating on 100% of users is almost equivalent to iterate on 50% of users twice.

![Figure 3.1: Effect of the number of users on Probe10 RMSE and on the optimal number of training epochs.](image)

Relation between results obtained on a subset with small $K$

To speed up the training process, I proposed to subsample users and use small $K$ values. In the followings, I refer to this as the simplified problem, while using all users and large $K$ values as the full problem.

An important question arises: when a parameter change improves result on the simplified problem, does it imply that is also improves result on the full problem?

If the answer is yes, then this is a fast way to find good parameter settings, otherwise it is not.

First, I evaluated the parameter optimization algorithm on BRISMF#1, optimizing it as a BRISMF with 4 parameters (different learning rate and regularization for users and movies: $\eta^p$, $\eta^q$, $\lambda^p$, $\lambda^q$), and as a BRISMF with 8 parameters (treating biases differently: $\eta^p$, $\eta^q$, $\lambda^p$, $\lambda^q$, $\eta^{pb}$, $\eta^{qb}$, $\lambda^{pb}$, $\lambda^{qb}$), using 11% of the users, and $K = 10$.

I run the optimization algorithm for 200 loops, using $n = 1$, that is in each loop only one new variable assignment is evaluated. One MF run required 38 seconds on average
(on an Intel Core2 Quad Q9300 CPU, at 3.3GHz). Results are depicted on Figure 3.2. Clearly, the 8 parameters variant has better performance, and the sudden improvement at the beginning tends to slow down very quickly in both variants.

![Figure 3.2: Comparing BRISMF with 4 and 8 parameters using parameter optimization. X-axis: the number of MF evaluations (loops). Y-axis: the best RMSE result until that loop.](image)

The parameter optimization algorithm requires a pseudo random number generator (PRNG). I evaluated whether the initial seed $S$ of the PRNG has effect on the results, using 20 different random seeds: $S \in \{10, 11, \ldots, 29\}$, which involves $2 \cdot 20 \cdot 200 = 8000$ MF run in total. I examined the minimum and maximum RMSE in each of the 200 cycles, across different PRNG initializations. Results are summarized in Table 3.1. We can see, that the 8 parameters MF always outperforms the 4 parameters MF, and the difference between maximum and minimum RMSE values is much smaller, that is it is less sensitive to the initial seed of the PRNG.

### Table 3.1: Minimum and maximum RMSE using different initial PRNG seeds for parameter optimization, in specific loops

<table>
<thead>
<tr>
<th>loop</th>
<th>4 parameters MF</th>
<th>8 parameters MF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>minimum</td>
<td>maximum</td>
</tr>
<tr>
<td>1</td>
<td>0.9304</td>
<td>0.9304</td>
</tr>
<tr>
<td>10</td>
<td>0.9277</td>
<td>0.9303</td>
</tr>
<tr>
<td>20</td>
<td>0.9270</td>
<td>0.9282</td>
</tr>
<tr>
<td>50</td>
<td>0.9263</td>
<td>0.9282</td>
</tr>
<tr>
<td>70</td>
<td>0.9262</td>
<td>0.9278</td>
</tr>
<tr>
<td>100</td>
<td>0.9254</td>
<td>0.9269</td>
</tr>
<tr>
<td>150</td>
<td>0.9254</td>
<td>0.9269</td>
</tr>
<tr>
<td>200</td>
<td>0.9247</td>
<td>0.9269</td>
</tr>
</tbody>
</table>

Last, I examined whether the improvements found by the parameter optimization algorithm on the subset of users with small $K$ turns into improvement on the whole
database, using larger $K$ values. I run the parameter optimization for BRISMF#1 on 11% of the data with $K = 10$, using one seed for 200 cycles, and then evaluated the 200 found parameter settings using 100% of the data and $K \in \{10, 20, 40, 100\}$. Results are depicted on Figure 3.3. The average number of epochs for these $5 \cdot 200$ MF evaluation was: 35, 27, 27, 26, 26 respectively. Note that these numbers are different from what is depicted on Figure 3.1 because the parameter optimization process changed parameters (decreased learning rates), which increased the number of epochs.

![Figure 3.3](image)

Figure 3.3: Comparing parameter optimization on BRISMF with 8 parameters, using 1/10 of the training set / full training set, for various $K$ values.

Clearly, there is strong correlation amongst 100%-trained results (correlation is 0.98—0.99), and a moderate correlation between the 11%-trained result and the 100%-trained results (correlation is 0.78—0.84).

Thus, both subsampling the users and using small $K$ values are good idea to speed up the parameter optimization process:

- using 1/9 of the training data, the average number of epochs was 35, as opposed to 27 of using the whole dataset. This is a $7 \times$ speedup, which compensates the side effect that the optimal parameter setting is different due to the subsampling.

- using small $K$ values: there is a strong correlation between results obtained with small and large $K$ values. Using $K = 10$ instead of $K = 40$ is a $4 \times$ speedup.

A very accurate MF found by parameter optimization

Using the above parameter optimization algorithm with some refinements (shrinking the set of allowed values for each parameter manually at some point) it found the following parameters (for the Netflix Prize dataset):

$w_p^w = -0.01$, $w_p^b = -0.006$, $w_q^w = -0.010$, $w_q^b = 0.020$, $\eta_p = 0.008$, $\eta_p^b = 0.016$, $\eta_q = 0.015$, $\eta_q^b = 0.007$, $\lambda_p = 0.048$, $\lambda_q = 0.008$, $\lambda_p^b = 0.019$, $\lambda_q^b = 0$, $G = 0$. In the followings I will use 3 variants of the above parameters later:

- BRISMF#250: $K = 250$,
3.1. IMPROVING THE PERFORMANCE OF A SINGLE MODEL

- BRISMF#1000: $K = 1000$.
- BRISMF#4000: $K = 4000$.

Table 3.2 summarizes the results obtained using these parameters and different $K$ values. Note that with $K = 1000$, we can achieve 6.04% improvement over Netflix’s Cinematch algorithm.

Table 3.2: A very accurate matrix factorization for the Netflix Prize with different $K$ values

<table>
<thead>
<tr>
<th>Model</th>
<th>$K$</th>
<th>Epochs</th>
<th>Probe10</th>
<th>Quiz</th>
<th>IoC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>12</td>
<td>0.9079</td>
<td>0.9071</td>
<td>4.66%</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>13</td>
<td>0.9057</td>
<td>0.9052</td>
<td>4.86%</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>13</td>
<td>0.9023</td>
<td>0.9020</td>
<td>5.19%</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>13</td>
<td>0.9009</td>
<td>0.9008</td>
<td>5.32%</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td>14</td>
<td>0.8992</td>
<td>0.8993</td>
<td>5.48%</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>14</td>
<td>0.8979</td>
<td>0.8980</td>
<td>5.61%</td>
</tr>
<tr>
<td></td>
<td>125</td>
<td>14</td>
<td>0.8975</td>
<td>0.8976</td>
<td>5.65%</td>
</tr>
<tr>
<td>BRISMF#250</td>
<td>250</td>
<td>14</td>
<td>0.8961</td>
<td>0.8962</td>
<td>5.80%</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>14</td>
<td>0.8948</td>
<td>0.8950</td>
<td>5.93%</td>
</tr>
<tr>
<td>BRISMF#1000</td>
<td>1000</td>
<td>14</td>
<td>0.8938</td>
<td>0.8939</td>
<td>6.04%</td>
</tr>
<tr>
<td></td>
<td>2000</td>
<td>14</td>
<td>0.8928</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRISMF#4000</td>
<td>4000</td>
<td>14</td>
<td>0.8919</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.1.4 Evaluation on Jester

In the following, I use a baseline BRISMF instance, termed as BRISMF#3, with the following parameter settings: $K = 10, \eta = 0.001, \lambda = 0.01, w^p = -w^q = w^u = -w^\tau = -0.01$. BRISMF#3 reaches its optimal test RMSE in the 10th epoch: 4.1400. Like to the Netflix dataset, increasing the value of $K$ increases the accuracy of the factorization:

$$\text{test RMSE} \begin{array}{c|c|c|c|c}
K & 5 & 10 & 20 & 50 \\
\hline
4.2241 & 4.1400 & 4.1084 & 4.0907 \\
\end{array}$$

Note that when we compare these results to the Netflix RMSE-s, these RMSE values should be divided by 5, since in the Netflix dataset ratings are on a 1 to 5 scale, while they are in a $-10$ to $10$ scale here. This also holds for improvements in the above table, for example, the RMSE of $K = 50$ is 0.0177 better than the RMSE of $K = 20$, but when we divide 0.0177 by 5, we got 0.0035.

I applied parameter optimization on BRISMF#3, and got the following parameterization:

- BRISMF#23: $K = 10, \eta = 0.00012, \lambda = 0.0219, w^p = -w^q = w^u = -w^\tau = -0.01$. The optimal test RMSE is reached in the 104th epoch: 4.0468. During the optimization, the number of training epochs were limited to 100. I tried to optimize the 2, 4 and 8 parameters variants, but there was only small difference between the results, the best result obtained with the 8-parameter variant was only 4.0757, which is only slightly better than the 2-parameter variant.
A possible explanation for the low learning rate found by the parameter optimization process: in the Netflix dataset, test ratings are newer (for each user) than training ratings, thus varying with learning rate and regularization we can control how much the old examples should be “forgotten” in favor of the newest ratings. In the Jester dataset, the date of the ratings are missing, test ratings have the same time distribution as train ratings, due to the random train-test split. Thus, the best factorization “remembers” each training example equally, which can be accomplished best in gradient methods by using as small learning rate as possible. However, the number of training epochs should not be too much, due to running time.

The lack of improvement of the 8 or 4 parameters variants can be explained in the same way: when we iterate over the Netflix database using \((u, \langle d_{ui}\rangle)\) ordering, movies and users have to be treated differently, if we assume that users’ taste change over time, but movies are stable. However, no such assumption can be made in the Jester dataset, due to the lack of dates.

Relation between results obtained using different subsets and different \(K\) values

I examined whether the improvements found by the parameter optimization algorithm using all users and \(K = 10\) turn into improvements on different subsets of the database (25% and 50%), using different \(K\) values (10,20,50).

Like in the Netflix dataset, there is strong correlation amongst 100%-trained results using different \(K\) values (correlation is 0.98 – 0.99). The same holds for results using 50% of users but different \(K\) values for training, and the results are similar when using only 25% of the users (correlation is 0.97 – 0.99). The correlation between 50% and 100% trained results (with the same \(K\) value) is much lower, 0.6. Results are similar between 25% and 50%, while between 25% and 100%, the correlation is approximately 0.

I checked why I got these disappointing results, and found that the optimal number of training epochs highly varied between different \(K\) values and subsets. Then I compared the test RMSE of 10th epoch of each of the models. In this way the zero correlation mentioned above changed to 0.89, and the 0.6 correlation changed to 0.94.

3.1.5 Evaluation on MovieLens

In the following, I use a baseline BRISMF instance, termed as BRISMF\#2, with the following parameter settings: \(K = 50, \eta = 0.01, \lambda = 0.02, w_E = -w_p^q = u_d = -w_I = -0.01.\) BRISMF\#2 reaches its optimal Probe10 RMSE in the 20th epoch: 0.8391.

I ran the simple parameter optimization algorithm for 200 cycles to optimize the parameters of BRISMF\#2. Using the 8 parameters variant I obtained the following RMSE result in the 133th optimization cycle: 0.8336 in the 72th training epoch. The parameters are the following:

- **BRISMF\#52**: \(\eta^p = \eta^q = 0.0048, \lambda^p = 0.04, \lambda^q = 0.03, \eta^{pb} = 0.0055, \eta^{qb} = 0.0025, \lambda^{pb} = 0.022, \lambda^{qb} = 0.04\)

Using the **CMA-ES\[21\]** algorithm, I obtained test RMSE = 0.8337 in the 156th evaluation. From this experiment, I conclude that the parameter optimization task is

\textsuperscript{1}I used the following implementation: [http://www.lri.fr/~hansen/cmaes_inmatlab.html](http://www.lri.fr/~hansen/cmaes_inmatlab.html)
simple enough to be solved by the proposed simple method, and there is no need for more sophisticated methods, like CMA-ES. With $K = 1000$ and $K = 2000$, I got RMSE = 0.8310. This means, that there is no point in increasing $K$ to very large values.

Using the 4 parameters variant, I obtained RMSE = 0.8337. Using the 2 parameters variant, I obtained RMSE = 0.8341, which is very close the 8-parameters variant. This means, that the introduction of 4 or 8 parameters is useless in this dataset.

**Relation between results obtained on a subset with small $K$**

I reran matrix factorizations computed during the optimization of BRISMF#52, but with different $K$ values (5,10,20,50,100). and using 25%, 50%, 75%, 100% of users. This means 200 $\times$ 5 $\times$ 4 matrix factorizations in total. Note that the parameter optimization was run for 200 cycles.

From these MFs, I generated 200-dimensional vectors holding the test RMSE of models. Then I computed the correlation between any pair of these 20 vectors.

First, I compared the 4 vectors of $K = 50$, and obtained the following correlations:

<table>
<thead>
<tr>
<th></th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>25%</td>
<td>1.00</td>
<td>0.99</td>
<td>0.97</td>
<td>0.95</td>
</tr>
<tr>
<td>50%</td>
<td>0.99</td>
<td>1.00</td>
<td>0.99</td>
<td>0.98</td>
</tr>
<tr>
<td>75%</td>
<td>0.97</td>
<td>0.99</td>
<td>1.00</td>
<td>0.99</td>
</tr>
<tr>
<td>100%</td>
<td>0.95</td>
<td>0.98</td>
<td>0.99</td>
<td>1.00</td>
</tr>
</tbody>
</table>

For other $K$ values, the 4 $\times$ 4 tables contained the same numbers.

Next I compared the 5 vectors of 100% subsampling. The correlation was 0.98 $-$ 0.99. Then I compared any of the 20 vectors. The minimum correlation was 0.92, which was between the $(K = 5,25\%)$ and $(K = 100,100\%)$ vectors.

Clearly, the parameter optimization process can be speed up both by using small $K$ and small subset of the dataset. However, since the database contains only 6040 users, there is no reason to further subsample it.

3.1.6 Conclusion

**Thesis 1.1** I suggested using 8 parameters in BRISMF ($\eta^p, \eta^q, \lambda^p, \lambda^q, \eta^{pb}, \eta^{qb}, \lambda^{pb}, \lambda^{qb}$) instead of 4 ($\eta^p, \eta^q, \lambda^p, \lambda^q$) or 2 ($\eta, \lambda$). I applied automatic parameter optimization to set the parameters. I suggested user-subsampling and using small $K$ values to speed up the optimization process. With the above considerations I achieved 6.04% improvement over Netflix’s Cinematch algorithm.

I also evaluated the proposed parameter optimization procedure on the MovieLens and Jester datasets. I concluded that the introduction of 4 or 8 parameters is useless for these datasets. I also evaluated whether user-subsampling and the usage of small $K$ values are useful to improve the speed of parameter optimization. I concluded that using small $K$ values is useful. Subsampling the users is useful on the MovieLens dataset, but not on the Jester dataset.

3.2 Parameter optimization on multiple models

A particular parameter setting can be advantageous because it produces a very accurate model, or it improves on a combination of some other models. In the previous section I
examined the optimization of accuracy.

In this section I will examine how the above parameter optimization method is able to generate models which “blends well” with another models, that is the target function of the optimization is now how much a generated model is able to improve the RMSE of the linear combination of some already existing models. I described the approach in [57, 58, 60].

The parameter optimization algorithm remains the same, but the target function is the RMSE of the blended result. To get better results, after each epoch of the active MF, I compute the blended RMSE. After performing 20 epochs, the algorithm returns the combination RMSE of the epoch with the best combination RMSE. While the learning curve of a simple model is also simple (U-shape, with 1 minimum), this does not hold for the combination: usually there are more than one local minima, i.e. the learning of an MF should not be stopped when the combination RMSE starts increasing, that is why I fixed the number of epochs to 20.

When the parameter optimization algorithm adjusted the parameters, I added the found MF to the combination, and started optimizing a new model. The following two models were found by the optimizer:

- **MLMF#200**: $K = 200, w_p = -0.034, w_q = -0.004, w = -0.021, w = -0.009, \eta = 0.019, \eta = 0.0385, \eta = 0.019, \eta = 0.0036, \lambda = 0.066, \lambda = 0.004, \lambda = 0.1652, \lambda = 0.0001$. Number of epochs: 6.

- **MLMF#80**: $K = 80, w_p = -0.029, w_p = 0.005, w_p = -0.011, w_p = 0.010, \eta = 0.014, \eta = 0.138, \eta = 0.006, \eta = 0.041, \lambda = 0.039, \lambda = 0.027, \lambda = 0.089, \lambda = 0.003$. Number of epochs: 11.

These parameterizations are very interesting, in my opinion: In MLMF#200, the value of $\lambda_p$ or $\lambda_{pb}$ are quite high. Note that also $\lambda_{pb} = 10 \times \lambda_q$, a huge difference. In MLMF#80, the value of $\lambda_p$ and $\lambda_q$ are high.

Table 3.3 summarizes the RMSE results of the models and the linear combinations, using ridge regression. Note that 1st and 5th rows of the table (like rows 2 and 8) are different: the 1st row shows the RMSE of the raw output of the predictor, while for the 5th row, the mean and variance of predictions were adjusted to coincide with Probe10, i.e. an appropriate linear transformation was applied on the output of the predictor.

The two MLMF-s together are able to improve the RMSE performance by $0.0011 + 0.0004$ on BRISMF#250, and by $0.0008 + 0.0005$ on BRISMF#4000.

**Evaluation on Jester**

I used a $K = 50$ variant of BRISMF#3, which I term here as BRISMF#53. The test RMSE of this method is 4.0298. When the mean and variance are adjusted, the RMSE is 4.0284. I run the parameter optimization algorithm to find a model that blends well with BRISMF#53. During the optimization, I used $K = 10$, and then I reran the best model with $K = 50$. The parameters of this model are the followings:

- **MLMF#53**: $K = 50, w_p = w_p = 0.009, w_p = w_p = 0.01, \eta = 0.001, \eta = 0.0006, \lambda = 0.029, \lambda = 0.01, \lambda = 0.001, \lambda = 0.0006, \lambda = 0.013, \lambda = 0.01$.

The RMSE of the combination of BRISMF#53 and MLMF#53 is 0.0094 better than the RMSE of BRISMF#53. Again, to compare it to the Netflix results, it should be divided by 5.
Table 3.3: Probe10 and Quiz RMSE of MF models (rows 1–4) and their combinations (rows 5–10)

<table>
<thead>
<tr>
<th>#</th>
<th>Model</th>
<th>Epoch</th>
<th>Probe10 RMSE</th>
<th>Quiz RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BRISMF#250</td>
<td>14</td>
<td>0.8961</td>
<td>0.8962</td>
</tr>
<tr>
<td>2</td>
<td>BRISMF#4000</td>
<td>14</td>
<td>0.8919</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>MLMF#200</td>
<td>6</td>
<td>0.9075</td>
<td>0.9072</td>
</tr>
<tr>
<td>4</td>
<td>MLMF#80</td>
<td>11</td>
<td>0.9260</td>
<td>0.9256</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td></td>
<td>0.8954</td>
<td>0.8955</td>
</tr>
<tr>
<td>6</td>
<td>1+3</td>
<td></td>
<td>0.8943</td>
<td>0.8943</td>
</tr>
<tr>
<td>7</td>
<td>1+3+4</td>
<td></td>
<td>0.8939</td>
<td>0.8938</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td></td>
<td>0.8915</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2+3</td>
<td></td>
<td>0.8907</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2+3+4</td>
<td></td>
<td>0.8902</td>
<td></td>
</tr>
</tbody>
</table>

Evaluation on MovieLens

I ran the parameter optimization algorithm to find a model that blends well with BRISMF#52. During the optimization, I used $K = 50$, and then I reran the best model with $K = 1000$ as well. The parameters are the following:

- MLMF#52: $K = 50, w_p^u = -0.014, w_q^u = 0.005, w_p^v = -0.009, w_q^v = 0.017, \eta_p^u = \eta_q^u = 0.0029, \lambda_p^u = \lambda_q^u = 0.035, \eta_p^b = 0.003, \eta_q^b = 0.0005, \lambda_p^b = 0.013, \lambda_q^b = 0.04.$

When the mean and variance of BRISMF#52 is adjusted, its RMSE is 0.8320 (instead of 0.8337). Blending this model with BRISMF#52, it can improve this RMSE by 0.0018.

When using a $K = 1000$ variant of both MFs, the improvement is still 0.0015.

Next, I ran the parameter optimization algorithm to find a model that improves on the combination of these 2 $K = 1000$ MFs. The found model can improve the RMSE by 0.0007.

Conclusion

**Thesis 1.2** I suggested finding automatic BRISMF parameterizations that improve on a combination of previously existing methods. I showed on the Netflix Prize dataset, that the proposed approach works: on BRISMF#250, two models (MLMF#200 and MLMF#80) can improve by 0.0011 + 0.0004 = 0.0015 RMSE points. This result allows to achieve the same performance as of BRISMF#1000, but within less than half of that time and memory requirement (1000 factors versus 250+200+80 factors).

I also evaluated the method on the Jester and MovieLens datasets. Again, the method can find models which blends well with a given model.

### 3.3 Support-based regularization

In Section 3.3 I introduced many new parameters to BRISMF: different learning rates and regularization factors for user / item features, bias feature / non-bias features. In this section I introduce further parameters: let each learning rate and regularization factor depend on the number of ratings of the active user (user support) and the active
item (item support). This will lead to an even more accurate model at the cost of using many parameters.

To support this idea, recall that in Section 2.3.2 I pointed out that the regularization term in equation 2.24 depends on \( n_u \), the user support. If this term is constant (equation 2.24), then the performance is worse. This naturally leads to the following question: is the addition of term \( n_u \lambda p_u p_a \) the best way of regularization? In this section I examine this question. A CF problem may require that the features of users or items with many ratings should be penalized less or more.

I described this approach in [59].

**Definition of SBRISMF** BRISMF has 8 important meta-parameters: \( \lambda^p, \eta^p, \lambda^q, \eta^q, \lambda^{pb}, \eta^{pb}, \lambda^{pf}, \eta^{pf} \). If we introduce more parameters, then we are able to get better performance, but the task of setting these parameters will become harder. In the followings, I introduce many-many parameters to get a very general method.

I replace \( \lambda^p \) with \( \lambda^p(u, i) \), \( \eta^p \) with \( \eta^p(u, i) \), and I also replace the other parameters \( (\lambda^q, \eta^q, \lambda^{pb}, \eta^{pb}, \lambda^{pf}, \eta^{pf}) \) with the appropriate functions.

The formulas of eq. (3.1) for the \((u, i)\)-th rating become the following:

\[
\begin{align*}
    p_u' &= p_u + \eta^p(u, i) \cdot (e_{ui} \cdot q_i - \lambda^p(u, i) \cdot p_u), \\
    q_i' &= q_i + \eta^q(u, i) \cdot (e_{ui} \cdot p_u - \lambda^q(u, i) \cdot q_i), \\
    b_u' &= b_u + \eta^{pb}(u, i) \cdot (e_{ui} - \lambda^{pb}(u, i) \cdot b_u), \\
    c_i' &= c_i + \eta^{pf}(u, i) \cdot (e_{ui} - \lambda^{pf}(u, i) \cdot c_i).
\end{align*}
\]  

(3.2)

To allow the learning rate and regularization to depend on the user and item support, I define the above functions in the following way: I give each function 7 parameters such that the value of the functions depend only on the number of ratings of active user and item. Let

\[
\begin{align*}
    \eta^p(u, i) &= p_1 + p_2 / \log n_u + p_3 / \sqrt{n_u} + p_4 / n_u + \\
                          &+ p_5 / \log n_i + p_6 / \sqrt{n_i} + p_7 / n_i, \\
    \eta^q(u, i) &= p_8 + p_9 / \log n_u + p_{10} / \sqrt{n_u} + p_{11} / n_u + \\
                          &+ p_{12} / \log n_i + p_{13} / \sqrt{n_i} + p_{14} / n_i,
\end{align*}
\]  

(3.3)

and the rest of the functions are defined similarly, thus we have 56 parameters in total \((p_1, \ldots, p_{56})\). Though at first sight this parameterization seems as an unnecessary over-complication, it can give a very accurate model, which blends well with other models. In the followings, I refer to this method as SBRISMF.

**Implementation details** Before optimizing the parameters of SBRISMF, I calculated the average value of \( \log n_u, \sqrt{n_u}, n_u, \log n_i, \sqrt{n_i} \) and \( n_i \), and modified the formula of (3.3) such that \( p_2 / \log n_u \) becomes \( p_2 / ((\log n_u) / \log n_u) \), etc. In this way, each parameter \( (p_1, p_2, \ldots, p_7) \) is multiplied with numbers of similar magnitude. If the value of \( f(u, i, [p_1, \ldots, p_7]) \) becomes negative or too large, I clamp it: the maximum value is also a parameter.

Optimizing the BRISMF method on the Probe10 dataset is highly prone to overfitting. To overcome this effect, I used a different Probe10 dataset, ran the optimization using that train-test splitting, and then applied the result on the “original” Probe10 dataset.
3.3. SUPPORT-BASED REGULARIZATION

Results

I found the following models using the parameter optimizer:

- SBRISMF#1000: an SBRISMF with 1000 features.
- SBRISMF#4000: the same as SBRISMF#2000, but $K = 4000$. The computation required more than 3 days.

See Table 3.4 for the RMSE values obtained with this approach. Note the 6th row of table: by adjusting the mean and variance of the predictions of SBRISMF#4000, we can achieve 6.53% improvement over Cinematch.

Table 3.4: Probe10 and Quiz RMSE of SBRISMFs (rows 1–3) and their combinations (rows 4–8)

<table>
<thead>
<tr>
<th>#</th>
<th>Model</th>
<th>Epochs</th>
<th>Probe10</th>
<th>Quiz</th>
<th>IoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SBRISMF#1000</td>
<td>8</td>
<td>0.8911</td>
<td>0.8912</td>
<td>6.33%</td>
</tr>
<tr>
<td>2</td>
<td>SBRISMF#2000</td>
<td>19</td>
<td>0.8910</td>
<td>0.8909</td>
<td>6.36%</td>
</tr>
<tr>
<td>3</td>
<td>SBRISMF#4000</td>
<td>18</td>
<td>0.8906</td>
<td>0.8906</td>
<td>6.39%</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
<td>0.8905</td>
<td>0.8903</td>
<td>6.42%</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td></td>
<td>0.8901</td>
<td>0.8898</td>
<td>6.47%</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td></td>
<td>0.8895</td>
<td>0.8893</td>
<td>6.53%</td>
</tr>
<tr>
<td>7</td>
<td>1+2</td>
<td></td>
<td>0.8891</td>
<td>0.8892</td>
<td>6.54%</td>
</tr>
<tr>
<td>8</td>
<td>1+2+3</td>
<td></td>
<td>0.8886</td>
<td>0.8886</td>
<td>6.60%</td>
</tr>
</tbody>
</table>

Evaluation on Jester

I evaluated the proposed method on the Jester dataset. I applied parameter optimization to optimize the parameters to get best performance. To prevent overfitting, I tuned the parameters using a different 90%-10% partitioning (there is no intersection between the original and this test set).

I used $K = 10$ during the parameter optimization, chose the best settings, and then ran SBRISMF on the original test set using these parameters, but different $K$ values. The obtained results are the followings:

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>20</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBRISMF</td>
<td>4.0641</td>
<td>4.0386</td>
<td>4.0228</td>
</tr>
<tr>
<td>BRISMF</td>
<td>4.0766</td>
<td>4.0476</td>
<td>4.0298</td>
</tr>
<tr>
<td>difference</td>
<td>0.0125</td>
<td>0.0090</td>
<td>0.0070</td>
</tr>
</tbody>
</table>

The SBRISMF results in this table are from the above mentioned parameterization, while the BRISMF results (the last line) are simple BRISMF models, using the parameters of BRISMF#23, except $K$, which is varied. The optimal number of epochs varied between 19 and 21, while in the case of BRISMF#23, it was always around 100. Interestingly, as $K$ grows, the difference between the results decreases. Both methods were optimized using $K = 10$. 
I also evaluated SBRISMF on the MovieLens dataset, using parameter optimization with $K = 50$ on a hold-out set. I obtained the following results, which were compared to BRISMF#52 with different $K$ values:

<table>
<thead>
<tr>
<th>$K$</th>
<th>10</th>
<th>20</th>
<th>50</th>
<th>100</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBIRSMF</td>
<td>0.8408</td>
<td>0.8362</td>
<td>0.8328</td>
<td>0.8319</td>
<td>0.8313</td>
</tr>
<tr>
<td>BRISMF</td>
<td>0.8429</td>
<td>0.8361</td>
<td>0.8336</td>
<td>0.8325</td>
<td>0.8317</td>
</tr>
<tr>
<td>difference</td>
<td>0.0021</td>
<td>-0.0001</td>
<td>0.0008</td>
<td>0.0006</td>
<td>0.0004</td>
</tr>
</tbody>
</table>

Again, as $K$ grows, the difference between the results decreases, apart from $K = 20$. The results indicate a moderate success of the method. Maybe this is due to the small number of users and or items: in the Netflix Prize dataset there is a much higher variation of the number of ratings of users and items. For example, in the MovieLens dataset, 15% most active users give the 50% of ratings. In the MovieLens dataset, the most popular 12% of movies provide 50% of the ratings, while the Netflix dataset is more skewed: 3.5% of movies provide 50% of the ratings.

### Conclusion

I remark that team BigChaos utilized my above idea of user- and item-dependent support for regularization, and applied it on BellKor’s TimeSVD++ model. That approach is a part of their Netflix Grand Prize winning solution [64, p.12], [28].

### Thesis 1.3

I suggested that the regularization and learning rate parameters of BRISMF should depend on the number of ratings of the active user and item. I proposed to replace each of these parameters with a function with 7 parameters. Again, parameters are optimized with an automatic parameter optimization algorithm. This method can achieve 6.53% improvement over Netflix’s Cinematch algorithm. I also tested the method on the Jester dataset with success, and on the MovieLens dataset with moderate success.

### 3.4 Resetting the user matrix improves performance

This method was discovered accidentally, but it is very simple, and despite its simplicity it is able to improve the performance of MF models by a huge amount in some cases. I described this approach in [57, 58, 60].

The algorithm modifies the training of BRISMF in the following way: when the performance on the validation set is optimal (i.e. we are in the best epoch), reset $P$ to its initial state, but do not reset $Q$, and keep the learning procedure go on until RMSE on the validation set improves.

I append letters “UM” to the modified methods, e.g. BRISMF#1 becomes BRISMF#1UM. Results are summarized in Table 3.5. I also indicated the required number of epochs before and after resetting $P$ (if applicable). Note, that the UM-trick improves the performance of BRISMF#250 and BRISMF#1000 on Quiz by 0.0027 and 0.0021 resp.

### Evaluation on Jester

I experimented with BRISMF#23, using different $K$ values. I obtained the following results:
3.4. Resetting the User Matrix Improves Performance

Table 3.5: Examining the effect of retraining user features

<table>
<thead>
<tr>
<th>Model</th>
<th>Epochs</th>
<th>Probe10</th>
<th>Quiz</th>
<th>IoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRISMF#1</td>
<td>10</td>
<td>0.9056</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRISMF#1UM</td>
<td>10+6</td>
<td>0.9053</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRISMF#250</td>
<td>14</td>
<td>0.8961</td>
<td>0.8962</td>
<td>5.80%</td>
</tr>
<tr>
<td>BRISMF#250UM</td>
<td>14+7</td>
<td>0.8937</td>
<td>0.8935</td>
<td>6.09%</td>
</tr>
<tr>
<td>BRISMF#1000</td>
<td>14</td>
<td>0.8938</td>
<td>0.8939</td>
<td>6.04%</td>
</tr>
<tr>
<td>BRISMF#1000UM</td>
<td>14+8</td>
<td>0.8921</td>
<td>0.8918</td>
<td>6.26%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>K</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>test RMSE</td>
<td>4.1595</td>
<td>4.0766</td>
<td>4.0476</td>
<td>4.0298</td>
</tr>
<tr>
<td>test RMSE of UM</td>
<td>4.1513</td>
<td>4.0683</td>
<td>4.0358</td>
<td>4.0176</td>
</tr>
</tbody>
</table>

The first learning step required \( \sim 100 \) epochs, while the second step \( \sim 30 \) epochs.

Clearly, the UM-method can improve on the accuracy of matrix factorizations on the Jester dataset as well.

Evaluations on MovieLens

I experimented with BRISMF#52, using different \( K \) values. I obtained the following results:

<table>
<thead>
<tr>
<th>K</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>50</th>
<th>100</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>test RMSE</td>
<td>0.8569</td>
<td>0.8429</td>
<td>0.8361</td>
<td>0.8336</td>
<td>0.8325</td>
<td>0.8317</td>
</tr>
<tr>
<td>test RMSE of UM</td>
<td>0.8570</td>
<td>0.8432</td>
<td>0.8382</td>
<td>0.8365</td>
<td>0.8355</td>
<td>0.8349</td>
</tr>
</tbody>
</table>

Clearly, this is not what we expected: the UM-variant is the worse. As we increase the dimension, the difference between the two factorizations increases.

I suspected that the inferior performance is due to the small number of users (6040 users, 3952 items). Thus I subsampled the users. If my assumption is right, the results should be even much worse. I used 25% of users, and obtained the following results:

<table>
<thead>
<tr>
<th>K</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>50</th>
<th>100</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>test RMSE</td>
<td>0.8802</td>
<td>0.8739</td>
<td>0.8714</td>
<td>0.8691</td>
<td>0.8683</td>
<td>0.8673</td>
</tr>
<tr>
<td>test RMSE of UM</td>
<td>0.8810</td>
<td>0.8780</td>
<td>0.8761</td>
<td>0.8741</td>
<td>0.8730</td>
<td>0.8722</td>
</tr>
</tbody>
</table>

Clearly, the results worsened, the gap became larger.

Next, I wanted to check whether the gap would become smaller if I increase the number of users. For this experiment I used the Netflix data and obtained the following results:

<table>
<thead>
<tr>
<th>U</th>
<th>K</th>
<th>20</th>
<th>50</th>
<th>100</th>
<th>250</th>
</tr>
</thead>
<tbody>
<tr>
<td>9011</td>
<td>0.9486 / 0.9610</td>
<td>0.9483 / 0.9729</td>
<td>0.9514 / 0.9803</td>
<td>0.9516 / 0.9802</td>
<td></td>
</tr>
<tr>
<td>18095</td>
<td>0.9407 / 0.9458</td>
<td>0.9408 / 0.9530</td>
<td>0.9409 / 0.9547</td>
<td>0.9414 / 0.9555</td>
<td></td>
</tr>
<tr>
<td>36234</td>
<td>0.9190 / 0.9168</td>
<td>0.9174 / 0.9192</td>
<td>0.9169 / 0.9197</td>
<td>0.9171 / 0.9199</td>
<td></td>
</tr>
<tr>
<td>91093</td>
<td>0.9089 / 0.9058</td>
<td>0.9053 / 0.9033</td>
<td>0.9036 / 0.9019</td>
<td>0.9019 / 0.9007</td>
<td></td>
</tr>
<tr>
<td>181421</td>
<td>0.9078 / 0.9063</td>
<td>0.9034 / 0.9014</td>
<td>0.9014 / 0.8993</td>
<td>0.8997 / 0.8978</td>
<td></td>
</tr>
<tr>
<td>480189</td>
<td>0.9057 / 0.9041</td>
<td>0.9000 / 0.8977</td>
<td>0.8979 / 0.8956</td>
<td>0.8961 / 0.8937</td>
<td></td>
</tr>
</tbody>
</table>
$U$ denotes the number of users in the subsample. Each cell contains two Probe10 RMSE values, the first is the original, the second is the UM-variant. Clearly, as we increase the number of users or decrease the value of $K$, the advantage of the UM-variant becomes larger. This may be the case on the MovieLens database as well.

**Why it works**

A possible explanation for why this method works is the following: during the training, the user features tend to overlearn sooner than the item features, because items have (much) more ratings. By resetting the user matrix, we delay the overlearning.

I experimented on the Netflix prize dataset with a different variant of the UM-method: I reset the item feature matrix instead of the user feature matrix, but that approach yielded worse RMSEs. The above assumption about overlearning would explain this experiment as well.

In the above experiments increasing the number of users amplifies the beneficial effect of this method. This is also in accordance with the above hypothesis about overlearning, because by increasing the number of users, items will have more ratings, thus they will overlearn later.

**Conclusion**

I remark that this idea has been reimplemented by team PragmaticTheory, and BRISMF-
#250UM and BRISMF#1000UM are part of their Netflix Grand Prize winning solution [38, p.39]. We (not me) have also successfully applied the idea on the training of NSVD1 [57].

**Thesis 1.4** I suggested a simple modification of the learning rule of BRISMF algorithm: in the best epoch, reset $P$ and keep the learning process going on until RMSE on a validation set improves. This modification can improve the accuracy of BRISMF-based matrix factorizations. On the Netflix Prize dataset: The RMSE performance of BRISMF#250 and BRISMF#1000 on Quiz can be boosted by 0.0027 and 0.0021 resp.

I also evaluated the method on the Jester dataset with success, and on the MovieLens without success. However, I pointed out, that this may be due to the small number of users in that dataset.

### 3.5 Hybrid MF+NSVD1 method

The linear blending of BRISMF and Paterek’s NSVD1 is much more accurate than these single methods, the difference can be 0.0034 in RMSE on the Netflix Prize dataset. This means, that these two approaches are highly complementary. An interesting question arises: can we integrate these methods into a single (but not necessarily simple) model?

In this section I propose a hybrid method that alloys BRISMF and Paterek’s NSVD1 in a single (but not simple) model. The idea is to represent both BRISMF and NSVD1 as a neural network, and then combine these two neural networks. The idea of hybrid method is described in [57]. The idea of treating a BRISMF as a neural network is described in [55, 60].
3.5. HYBRID MF+NSVD1 METHOD

3.5.1 Matrix factorization and NSVD1 as neural networks

Both matrix factorization (RISMF, BRISMF) and NSVD1 methods can be seen as learning neural networks. In this section I point out that the learning scheme of some MF variants and NSVD1 can directly be represented in the framework of neural networks (NN).

Matrix factorization as neural network The learning of the ISMF and RISMF models can be paired with the multi-layer perceptron depicted on Figure 3.4. The network has $N$ inputs, $M$ outputs and $K$ hidden neurons and an identity activation function in each neuron. The weight between the $u$-th input and the $k$-th hidden neuron corresponds to $p_{uk}$, and the weight between the $k$-th hidden neuron and the $i$-th output neuron corresponds to $q_{ik}$.

![Figure 3.4: The multilayer perceptron equivalent for the general MF scheme](image)

In the learning phase, an incremental learning method is used. For the $(u, i)$-th rating, I set the input $x$ so that $x_u$ is 1 and $x_{u'\neq u} = 0$, thus $z_k = p_{uk}$ holds. Let $\hat{r}_{ui} = y_i$ denote the $i$-th output of the network. I compute the error: $e_{ui} = r_{ui} - \hat{r}_{ui}$. This error is back-propagated from the $i$-th output neuron towards the input layer. This neural network (NN) with this special learning is equivalent to ISMF.

If we apply regularization, which is termed weight decay in the NN literature, we get the RISMF approach.

We can extend this NN to be equivalent to BRISMF: we can set the 1st column of $P$ and the second column of $Q$ to constant 1 to emulate biases. In the evaluation phase, we can set the input as in the learning phase, and $y_i$ ($i = 1, \ldots, M)$ predicts the active user’s rating on the $i$-th item. Note that the training of this neural network is different from multi-layer perceptron training using the backpropagation algorithm:

- only those outputs matters that corresponds to an existing rating of the active user.
- activation functions are linear (not sigmoid)
- the input is a sparse vector. The training algorithm should exploit this to be efficient.

Of course, the role of users and items can be interchanged, and if we do so, we get a different neural network. However, the above scheme directly corresponds to the MF learning with user-ordered ratings database (and if we process the outputs of the
network using time ordering, then to the (user,time)-ordered database), which is better suited for CF tasks, as it has been pointed out by Takács et al. [60].

It was shown that feed-forward neural networks with linear activation function can find the principal components of a dataset [3], since all local minima are global minima there. However, the problem setting of matrix factorization for collaborative filtering differs in two minor and one major issues: First, the goal is to factorize a non-squared matrix (the number of inputs of the neural network is different from the number of outputs), second, the weights of the neural network are penalized, thus, the extracted vectors need not be orthogonal. Third and principally, not all outputs are defined, since users rate only a small subset of the items. This latter point causes that we cannot expect local minima to be global minima, as it has been reflected on page 26.

I remark that Team BigChaos picked up the idea of representing MFs as NNs, and used \( \tanh(\cdot) \) as activation function, and it took part of their winning solution for the Netflix Progress Prize 2008 [62, p.8].

**NSVD1 as neural network** The learning of NSVD1 (without biases) can be paired with the learning of the neural network depicted on Figure 3.5 as follows: the neural network has the same structure as the NN for RISMF, except one modification: the input is not an \( N \)-dimensional binary vector indicating who is the active user, but it is an \( M \)-dimensional normalized binary vector that indicates which items are rated by the active user, and has length 1. User and item biases can be easily added.

In this way, this network directly enforces that users rating similar sets of items will have similar user feature vector \( \mathbf{p}_u \), even if they rate them differently: note that only the existence of rating matters, but not the actual value of rating.

**3.5.2 Combining the two neural networks**

Experiments showed us that MF is much more accurate than NSVD1, however, blending them linearly can further improve the accuracy by a large amount. A possible explanation for this is that while NSVD1 is not so accurate, it treats users differently, and in this way it is able to capture different aspects of the data.

An interesting question arises: could we create a hybrid method that is able to capture both or even new aspects of the data?

The are many ways to combine two methods (predictors), \( A \) and \( B \):
3.5. HYBRID MF+NSVD1 METHOD

- do linear (or nonlinear) blending with $A$ and $B$
- run $B$ on the residual of $A$
- run $A$ on the residual of $B$
- create a hybrid variant as I will propose in the followings.

I proposed a new scheme to combine the two methods, which is a NN depicted on Figure 3.6. For simplicity, I neglected the biases on the figure. This NN is the “vertical concatenation” of the NN for MF and the NN for NSVD1: the input and hidden layers are concatenated, the output layer is shared. The neural network has $N + M$ inputs, $K_1 + K_2$ neurons in the hidden layer, and $M$ outputs. For the active user $u$, the first $N$ dimensions of the input is the same as of MF, the last $M$ dimensions are the same as of NSVD1. The first $N$ input neurons are connected only with the first $K_1$ hidden neurons. Similarly, the last $M$ input neurons are connected with the last $K_2$ hidden neurons. The network predicts the ratings of the active user on any movies.

$$\hat{r}_{ui} = b_u + c_i + p_u^T q_i' + n_u^{-0.5} \left( \sum_{j:(u,j) \in R} w_j^T q_i'' \right)$$

(3.4)

I proposed different learning rates and regularizations for the MF and NSVD1 part.

Note that this method can be seen as running two methods on the residuals of “each other”: if we set the learning rates of $B$ to zero at the beginning, then train the network until convergence, then set the learning rates of $A$ to zero and $B$ to non-zero, and then further train the network until convergence, then we get (almost) exactly the same as running $B$ on the residual of $A$.

Since both methods are trained simultaneously, we can expect to extract new signals from the data. Moreover, we can expect to get better (not worse) accuracy than that of
a single MF, since in the worst case, we can set the learning rates of the NSVD1 part to zero (or near-zero), thus completely relying only on the MF part.

In the followings, let $p_u'' = n_u^{-0.5} \sum_{j:(u,j) \in \mathcal{R}} w_j$.

I proposed to use following prediction and update rules:

$$\hat{r}_{ui} = b_u + c_i + p_u'T q'_i + p_u''T q''_i$$

$$e_{ui} = r_{ui} - \hat{r}_{ui}$$

$$p'_u := p'_u + \eta p'(e_{ui} q'_i - \lambda p'_u)$$

$$q'_i := q'_i + \eta q'(e_{ui} p'_u - \lambda q'_i)$$

$$p''_u := p''_u + \eta p''(e_{ui} q''_i - \lambda p''_u)$$

$$q''_i := q''_i + \eta q''(e_{ui} p''_u - \lambda q''_i)$$

$$b_u := b_u + \eta b(e_{ui} - \lambda b_b)$$

$$c_i := c_i + \eta q (e_{ui} - \lambda q_c)$$

Note that the model of the hybrid MF + NSVD1 method can be converted into a matrix-factorization model:

$$\hat{r}_{ui} = p'_u q'_i, \quad \text{where}$$

$$p_u = (b_u; 1; p'_u; p''_u),$$

$$q_i = (1; c_i; q'_i; q''_i),$$

where “;” is the vector concatenation operator.

### 3.5.3 Related works

The hybrid MF+NSVD1 approach mentioned above was published on August 4, 2008 [57]. My colleague, Gábor Takács developed this idea further, and introduced modifications to simplify the setting of learning rates and regularizations. That further developed idea is described in [57]. He also applied the “UM”-method (Section 3.4) on both the MF and the NSVD1 parts.

Some kinds of the unification of MF and NSVD1 were invented in the Netflix Prize before or after the publication of [57]:

- Constrained Probabilistic MF approach by Salakhutdinov and Mnih [44] (December 04, 2007):

$$\hat{r}_{ui} = \left(p_u + n_u^{-1} \sum_{j:(u,j) \in \mathcal{R}} w_j\right)^T q_i$$

Salakhutdinov and Mnih invented the usability of implicit feedback data independently from Paterek [32], the inventor of NSVD1. They also note that this approach is especially useful for users with few ratings. Formerly the authors proposed Restricted Boltzmann Machines for CF [43], where they also explored one way to integrate implicit feedback data.

- Bell and Koren [9] (December, 2007):

$$\hat{r}_{ui} = \left(p_u + n_u^{-0.5} \sum_{j:(u,j) \in \mathcal{R}} w_j\right)^T q_i$$
3.5. HYBRID MF+NSVD1 METHOD

Reported Quiz RMSE: 0.8970.

- SVD++ by Koren [27] (August 24, 2008):

\[ \hat{r}_{ui} = b_u + c_i + \left( \mathbf{p}_u + n_u^{-0.5} \sum_{j: (u,j) \in \mathcal{R}} \mathbf{w}_j \right)^T \mathbf{q}_i \]

Some kind of hybrid approaches were also known in the competition, although the notion of creating hybrid approaches was not clear at that time. For example, creating from \( \hat{r}_{ui} = c_i \) and \( \hat{r}_{ui} = b_u \) the \( \hat{r}_{ui} = b_u + c_i \) predictor. Note that during the hybridization, the meaning of \( b_u \) changes from user average to user offset, while \( c_i \) remains the movie average (because movies have \( 27 \times \) as many ratings as users). Another example of a hybrid approach is BRISMF: it unifies \( \hat{r}_{ui} = b_u + c_i \) and \( \hat{r}_{ui} = \mathbf{p}_u^T \mathbf{q}_i \).

Hybrid approaches were also proposed by team BellKor, for example BellKor’s integrated model [27] extends SVD++ by two additional terms. Team PragmaticTheory proposed a very complex hybrid approach the prediction formula of which occupies a complete A4 page [38, p.18] and gives Quiz RMSE = 0.8713, which is the most accurate “single model” (i.e. no blending) of the competition.

The principle behind hybrid approaches is baselining: we have a baseline predictor with few parameters, and then extend this model with more sophisticated terms which can overfit more easily. In case of the hybrid MF+NSVD1 approach, biases serve as a baseline (\# of parameters: \( N + M \)), which is then extended with more parameters (\( 2KM \)) based on the implicit feedback of users (NSVD1), and then it is further extended by the \( r_{ui} = \mathbf{p}_u^T \mathbf{q}_i \) term which has \( K(N + M) \) parameters.

3.5.4 Results

I ran the parameter optimization algorithm to optimize the performance of the hybrid method. I also created an NSVD1 method by parameter optimization. Results are summarized in Table 3.6. The Hybrid methods in the table have the same parameters, only the value of \( K_1 \) and \( K_2 \) differs. Clearly, the Hybrid#1000-80 approach outperforms the combination of BRISMF#1000 and NSVD1#80, cf. rows 7, 8, 9 and 8+9 of the table. It also captures different aspects of the data, as the combination 7+8+9 is much better than 7 or 8+9, the improvement is 0.0022 in RMSE (compared to row 7).

Note that also SBRISMF and the hybrid approach blends well: row 10+7 has an IoC = 7.04%. The RMSE of their combination is 0.0039 better than the better method.

**Thesis 1.5** I suggested a hybrid method that efficiently alloys BRISMF and NSVD1. The idea is to sum the output the two methods, and train them simultaneously. This can be seen as training two methods on the residual of each other. I put BRISMF and NSVD1 in the framework of multi-layer perceptrons. I experimented with the Netflix Prize data, where the method (Hybrid#1000-80) achieved 6.61% improvement over Cinematch. I also pointed out that the approach blends well with the method of Thesis 1.3: the blending of this 6.61% and that 6.39% approaches can achieve 7.04% improvement over Cinematch.
Table 3.6: Results of hybrid MF+NSVD1 matrix factorizations (rows 1–10) and their combinations (other rows)

<table>
<thead>
<tr>
<th>#</th>
<th>Model</th>
<th>$K_1$</th>
<th>$K_2$</th>
<th>Probe10</th>
<th>Quiz</th>
<th>IoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hybrid#20-20</td>
<td>20</td>
<td>20</td>
<td>0.8959</td>
<td>0.8962</td>
<td>5.80%</td>
</tr>
<tr>
<td>2</td>
<td>Hybrid#80-20</td>
<td>80</td>
<td>20</td>
<td>0.8916</td>
<td>0.8924</td>
<td>6.20%</td>
</tr>
<tr>
<td>3</td>
<td>Hybrid#80-80</td>
<td>80</td>
<td>80</td>
<td>0.8899</td>
<td>0.8907</td>
<td>6.38%</td>
</tr>
<tr>
<td>4</td>
<td>Hybrid#160-80</td>
<td>160</td>
<td>80</td>
<td>0.8890</td>
<td>0.8899</td>
<td>6.46%</td>
</tr>
<tr>
<td>5</td>
<td>Hybrid#400-80</td>
<td>400</td>
<td>80</td>
<td>0.8882</td>
<td>0.8891</td>
<td>6.55%</td>
</tr>
<tr>
<td>6</td>
<td>Hybrid#400-400</td>
<td>400</td>
<td>400</td>
<td>0.8878</td>
<td>0.8887</td>
<td>6.59%</td>
</tr>
<tr>
<td>7</td>
<td>Hybrid#1000-80</td>
<td>1000</td>
<td>80</td>
<td>0.8876</td>
<td>0.8885</td>
<td>6.61%</td>
</tr>
<tr>
<td>8</td>
<td>BRISMF#1000</td>
<td>1000</td>
<td>80</td>
<td>0.8938</td>
<td>0.8939</td>
<td>6.04%</td>
</tr>
<tr>
<td>9</td>
<td>NSVD1#80</td>
<td>40</td>
<td>40</td>
<td>0.9342</td>
<td>0.9332</td>
<td>1.91%</td>
</tr>
<tr>
<td>10</td>
<td>SBRISMF#4000</td>
<td>4000</td>
<td>4000</td>
<td>0.8906</td>
<td>0.8906</td>
<td>6.39%</td>
</tr>
</tbody>
</table>

|     | 1               |       |       | 0.8959  | 0.8962 | 5.80% |
|     | 2               |       |       | 0.8914  | 0.8922 | 6.22% |
|     | 3               |       |       | 0.8898  | 0.8906 | 6.39% |
|     | 4               |       |       | 0.8889  | 0.8898 | 6.47% |
|     | 5               |       |       | 0.8881  | 0.8890 | 6.56% |
|     | 6               |       |       | 0.8877  | 0.8885 | 6.61% |
|     | 7               |       |       | 0.8875  | 0.8883 | 6.63% |
|     | 8               |       |       | 0.8933  | 0.8934 | 6.10% |
|     | 9               |       |       | 0.9342  | 0.9331 | 1.92% |
|     | 8+9             |       |       | 0.8901  | 0.8900 | 6.45% |
|     | 7+8+9           |       |       | 0.8856  | 0.8861 | 6.86% |
|     | 10              |       |       | 0.8895  | 0.8893 | 6.53% |
|     | 10+7            |       |       | 0.8840  | 0.8844 | 7.04% |
|     | 10+7+8          |       |       | 0.8838  | 0.8843 | 7.05% |
|     | 10+7+8+9        |       |       | 0.8832  | 0.8835 | 7.14% |

3.6 RMSE values reported by other authors

Finally, let me compare the accuracy of the proposed methods with other RMSE values reported for the Netflix Prize dataset. The comparison presented in Table 3.7 focuses on methods where Quiz RMSE values are available. The table shows that my presented MF methods compare favorably with other published ones.

The hybrid MF + NSVD1 approach and SVD++ are very similar. Not surprisingly, their accuracies are also similar: SVD++ with 200+200 features achieves 6.34% over Cinematch, while the hybrid method with 160+80 features has 6.46%.

The date of last two rows are marked by a star, because these results were regenerated for this dissertation.
Table 3.7: Comparison of Quiz RMSE values of reported MF based methods. I also indicate the date of publication.

<table>
<thead>
<tr>
<th>Source</th>
<th>Method’s name</th>
<th>Quiz</th>
<th>IoC</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paterek [32]</td>
<td>Basic + RSVD + RSVD2</td>
<td>0.9070</td>
<td>4.67%</td>
<td>2007/08/12</td>
</tr>
<tr>
<td>Salakhutdinov and Mnih [44]</td>
<td>PMF + PMF with a learnable prior + constrained PMF</td>
<td>0.8970</td>
<td>5.72%</td>
<td>2007/12/04</td>
</tr>
<tr>
<td></td>
<td>best stand-alone MF with adaptive user factors</td>
<td>0.8955</td>
<td>5.88%</td>
<td>2007/11/13</td>
</tr>
<tr>
<td>Koren [27]</td>
<td>SVD++ with K = 200 Integrated model with K = 200</td>
<td>0.8911</td>
<td>6.34%</td>
<td>2008/08/24</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.8868</td>
<td>6.79%</td>
<td>2008/08/24</td>
</tr>
<tr>
<td>this dissertation</td>
<td>stand-alone MF, BRISMF#1000 stand-alone MF with retrained features, BRISMF#250UM stand-alone MF with retrained features, BRISMF#1000UM MF with modified regularization, SBRISMF#4000 Hybrid MF+NSVD1 160+80 Hybrid MF+NSVD1 1000+80</td>
<td>0.8939</td>
<td>6.04%</td>
<td>2008/08/24 [58]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.8935</td>
<td>6.09%</td>
<td>2008/08/04 [57]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.8918</td>
<td>6.26%</td>
<td>2008/08/24 [58]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.8893</td>
<td>6.53%</td>
<td>2008/10/25 [59]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.8899</td>
<td>6.46%</td>
<td>2008/08/04* [57]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.8885</td>
<td>6.61%</td>
<td>2008/08/04* [57]</td>
</tr>
</tbody>
</table>

3.7 Summary

In this chapter I proposed many methods to improve the accuracy of BRISMF-based matrix factorizations. I proved experimentally that these algorithms outperform previous matrix factorization methods. Most of the methods are not only accurate but blend well with other methods. For example, the hybrid approach and the SBRISMF approach are very complementary.

I tested most of the approaches on the MovieLens and Jester datasets. The experiments showed that MovieLens and Netflix datasets are very similar, although MovieLens contains much less ratings.

I remark that two of my approaches were utilized by winning Netflix Prize contestants: the SBRISMF idea by Team BigChaos, and the UM approach by Team PragmaticTheory.
Chapter 4

Improving on the speed of matrix factorization

4.1 Retraining user features for handling new ratings

The incremental gradient descent weight update of RISMF has a serious drawback: item features change while we iterate through users. If the change is significant, user features updated in the beginning of an epoch may be inappropriate at the end of the epoch. I found two ways to solve this.

1. We can evaluate the test ratings of a user immediately after iterating through its ratings in the training set, and before starting to iterate through the next user’s ratings.
2. We can completely recompute the user features after the learning procedure. This method can serve as an efficient incremental training method for recommender systems. Algorithm 4 describes the method.

I described this idea in [58, 60].

The algorithm can be summarized as follows:

- Run the training algorithm of BRISMF, save \( Q \) and \( c \).
- Fix \( Q \) and \( c \), reset \( P \) and \( b \), and run BRISMF again. Let \( n^* \) denote the optimal number of epochs. This is the end of the algorithm.
- If user \( u \) gives a new rating, or she is a new user, then reset \( b_u \) and the \( u \)-th row of \( P \), fetch the ratings of that user only, and run the inner loop of BRISMF for that user with those ratings \( n^* \) times. Now \( p_u \) and \( b_u \) are refreshed.

Note that this method can efficiently incorporate into the model new users or new ratings of existing users without the necessity of retraining the entire model, which is very important for recommender systems.

I remark that the presented algorithm does not handle the addition of new items, and after the addition of many new ratings to the database, \( Q \) will be obsolete, thus the first training step should be re-run.

I also remark that Algorithm 4 is very similar to the “UM” approach proposed in Section 3.4 where after resetting the \( P \), the \( Q \) is trained, not fixed.
CHAPTER 4. IMPROVING ON THE SPEED OF MATRIX FACTORIZATION

Input: \( R \): training set, \( \eta \): learning rate, \( \lambda \): regularization factor

Output: \( P^*, Q^*, b^*, c^* \): the user and item feature matrices and biases,
\( n^* \): optimal number of epochs in the 2nd step

1. Partition \( R \) into two sets: \( T \) (training set), \( V \) (validation set)
2. First training step:
   3. Call Algorithm 1 (page 30), store the result in \( (P_1, Q_1, b_1, c_1, n_1^*) \)
4. Second training step: call Algorithm 1, but:
   5. Initialize randomly only \( P \) and \( b \),
   6. let \( Q \) and \( c \) be initialized with \( Q_1 \) and \( c_1 \) resp.
   7. Apply gradient descent only on \( P \) and \( b \), do not change \( Q \) or \( c \).
   8. Store the result in \( (P_2, b_2, n_2^*) \).
9. Let \( P^* = P_2 \), \( b^* = b_2 \), \( Q^* = Q_1 \), \( c^* = c_1 \), \( n^* = n_2^* \).
10. end

Algorithm 4: Algorithm for retraining user features

4.1.1 Experiments

I investigated the effectiveness of retraining user features. The experiments were run with three methods: BRISMF#1 (page 43), BRISMF#250 (page 46), BRISMF#1000 (page 46).

Results are summarized in Table 4.1. I append letter “U” to the method name indicating that it is trained by Algorithm 4. In case of BRISMF#1, the retraining of \( P \) increased the training error, while in case of BRISMF#250 and 1000, it decreased. One should focus in this table on the following: the “U” variants must not have a much worse performance than the original variants. This is true for all methods: in one case, it is slightly worse, in the other two cases, it is slightly better. This implies, that the “U” variant yields similar performance, thus, it can be useful for practical purposes, where—for example—a user provides a new rating, and not the whole training procedure should be re-run, only the active user’s ratings need to be processed.

<table>
<thead>
<tr>
<th>Model</th>
<th>Epochs</th>
<th>Probe10</th>
<th>Quiz</th>
<th>IoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRISMF#1</td>
<td>10</td>
<td>0.9056</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRISMF#1U</td>
<td>10+8</td>
<td>0.9072</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRISMF#250</td>
<td>14</td>
<td>0.8961</td>
<td>0.8962</td>
<td>5.80%</td>
</tr>
<tr>
<td>BRISMF#250U</td>
<td>14+8</td>
<td>0.8953</td>
<td>0.8954</td>
<td>5.89%</td>
</tr>
<tr>
<td>BRISMF#1000</td>
<td>14</td>
<td>0.8938</td>
<td>0.8939</td>
<td>6.04%</td>
</tr>
<tr>
<td>BRISMF#1000U</td>
<td>14+8</td>
<td>0.8936</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Using only a random subset of users I investigated the effect of retraining only \( P \), when BRISMF#250 was learnt on a subset of the database. The question in this case is: how reliable is a \( Q \) learnt only on a subset of the database.

First, I kept only the 40%, 60% or 80% of users and ran an MF algorithm and fixed the resulting \( Q \). Then I reset and learnt \( P \), first on the same subset of the database,
4.1. RETRAINING USER FEATURES FOR HANDLING NEW RATINGS

and then on the entire database. In all 3 cases, the difference between the two Probe10 RMSE results was less than 0.0013. Each Probe10 RMSE was less than 0.8970. Thus, we can conclude that the proposed retraining method can handle the addition of new users.

The results are summarized in Table 4.2.

- The first column tells how much percentage of the users were used in the first learning step.
- The next two columns tell the optimal number of epochs and Probe10 RMSE of the first learning step.
- The next two columns tell the optimal number of epochs and Probe10 RMSE of the second learning step (running on the same database).
- The next two columns tell the optimal number of epochs and Probe10 RMSE of the second learning step (running on the whole database).

<table>
<thead>
<tr>
<th>%</th>
<th>Epochs</th>
<th>Probe10</th>
<th>Epochs</th>
<th>Probe10</th>
<th>Epochs</th>
<th>Probe10</th>
</tr>
</thead>
<tbody>
<tr>
<td>40%</td>
<td>14</td>
<td>0.8982</td>
<td>8</td>
<td>0.8990</td>
<td>8</td>
<td>0.8998</td>
</tr>
<tr>
<td>60%</td>
<td>14</td>
<td>0.8972</td>
<td>8</td>
<td>0.8969</td>
<td>8</td>
<td>0.8969</td>
</tr>
<tr>
<td>80%</td>
<td>14</td>
<td>0.8959</td>
<td>8</td>
<td>0.8952</td>
<td>8</td>
<td>0.8964</td>
</tr>
<tr>
<td>100%</td>
<td>14</td>
<td>0.8961</td>
<td>8</td>
<td>0.8953</td>
<td>8</td>
<td>0.8953</td>
</tr>
</tbody>
</table>

Table 4.2: Examining the addition of new users on BRISMF#250.

Discarding the last few ratings of users    Second, I discarded the last $N_1$ ratings of each user and ran the same retraining procedure. In our experiments, $N_1$ was set to 0, 10, 20, 40. Obviously, the removal of ratings increased Probe10 RMSE significantly; the highest score was 1.0038, whereas Probe10 RMSE on the entire training database went up to only 0.8980, which means that the $Q$ calculated on the subset of the data differs slightly from the $Q$ calculated on the entire data set. Thus, the proposed retraining method can handle the addition of new ratings as well. These experiments verify the usability of user feature retraining method for handling new users or ratings. The results are summarized in Table 4.3. Columns have the same meaning as in Table 4.2

<table>
<thead>
<tr>
<th>$N_1$</th>
<th>Epochs</th>
<th>Probe10</th>
<th>Epochs</th>
<th>Probe10</th>
<th>Epochs</th>
<th>Probe10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14</td>
<td>0.8961</td>
<td>8</td>
<td>0.8953</td>
<td>8</td>
<td>0.8953</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>0.9252</td>
<td>8</td>
<td>0.9246</td>
<td>8</td>
<td>0.8956</td>
</tr>
<tr>
<td>20</td>
<td>13</td>
<td>0.9550</td>
<td>8</td>
<td>0.9547</td>
<td>8</td>
<td>0.8962</td>
</tr>
<tr>
<td>40</td>
<td>12</td>
<td>1.0043</td>
<td>8</td>
<td>1.0038</td>
<td>9</td>
<td>0.8980</td>
</tr>
</tbody>
</table>

Table 4.3: Examining the addition of new ratings.

Using only users with a pre-specified number of ratings    I created 10 partitions of users, based on their number of ratings. Each partition was made by specifying a lower and an upper limit on the number of ratings of users, such that the number of ratings associated with the partitions are roughly equal. For a partition, I run the training algorithm using only those ratings, and then saved the resulting $Q$. Then I
retrained $P$ using the whole database, while $Q$ remained fixed. Table 4.4 indicates the results. Although the number of ratings were almost equal in each partition the results are somewhat different. It indicates that the ratings of users with $\sim 500$ ratings are the most valuable (for estimating $Q$).

Table 4.4: Probe10 RMSE of the second training step, when using only a subset of users. I also indicate the percentage ratio of the number of ratings of the subset and the original training data.

<table>
<thead>
<tr>
<th>$\leq n_u &lt;$</th>
<th>RMSE</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.9338</td>
<td>10.06</td>
</tr>
<tr>
<td>97</td>
<td>0.9227</td>
<td>9.94</td>
</tr>
<tr>
<td>178</td>
<td>0.9194</td>
<td>9.98</td>
</tr>
<tr>
<td>263</td>
<td>0.9176</td>
<td>10.02</td>
</tr>
<tr>
<td>357</td>
<td>0.9178</td>
<td>9.98</td>
</tr>
<tr>
<td>463</td>
<td>0.9173</td>
<td>10.00</td>
</tr>
<tr>
<td>588</td>
<td>0.9182</td>
<td>10.00</td>
</tr>
<tr>
<td>744</td>
<td>0.9187</td>
<td>9.99</td>
</tr>
<tr>
<td>965</td>
<td>0.9195</td>
<td>10.01</td>
</tr>
<tr>
<td>1339</td>
<td>0.9221</td>
<td>10.01</td>
</tr>
</tbody>
</table>

Using only the first few ratings of a movie I subsampled the database in the following way: let $N$ be a predefined constant integer. I ordered the database of ratings by $(i, u)$, and for each $i$, kept only the first $N$ ratings. Then I reordered this subset by $(u, d_{ui})$, that is the original ordering of ratings. This database is used then for the first training step. I used the whole database for the second training step. I varied the number of epochs in the first training step to get optimal results, using these numbers: 6, 12, 24, 36, 48, 60. Table 4.5 indicates the results of the second training step and the number of epochs of the first training step.

Interestingly, only 100 ratings per movie is enough to get an RMSE comparable to Netflix’s Cinematch algorithm, which is 0.9514.

Evaluation on Jester I also evaluated this method on the Jester dataset, using the BRISMF#23 model (page 47), with different $K$ values. I obtained the following results:

<table>
<thead>
<tr>
<th>$K$</th>
<th>test RMSE</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>test RMSE</td>
<td>4.1595</td>
<td>4.0766</td>
<td>4.0476</td>
<td>4.0298</td>
<td></td>
</tr>
<tr>
<td>test RMSE of U</td>
<td>4.1713</td>
<td>4.0920</td>
<td>4.0641</td>
<td>4.0462</td>
<td></td>
</tr>
</tbody>
</table>

Here the second and third rows show the results of BRISMF#23 and BRISMF#23U with different $K$ values, respectively. The first learning step required $\sim 100$ epochs, while the second step only $\sim 30-40$ epochs. Although the accuracy of the “U” variant is worse by $0.0118 - 0.0165$ (to compare with Netflix results, it should be divided by 5, in my opinion it is acceptable), but note that the running time is greatly decreased when a user gives a new rating: there is no need to iterate over the full ratings database, only the active users ratings need to be queried.
4.1. RETRAINING USER FEATURES FOR HANDLING NEW RATINGS

Table 4.5: Number of epochs in the first training step and Probe10 RMSE of the second training step, when using only the first $N$ ratings of movies. I also indicate the percentage ratio of the number of ratings of the subset and the original training data.

<table>
<thead>
<tr>
<th>$N$</th>
<th>Epoch</th>
<th>RMSE</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>24</td>
<td>1.0063</td>
<td>0.16</td>
</tr>
<tr>
<td>20</td>
<td>48</td>
<td>0.9889</td>
<td>0.34</td>
</tr>
<tr>
<td>30</td>
<td>48</td>
<td>0.9764</td>
<td>0.51</td>
</tr>
<tr>
<td>40</td>
<td>48</td>
<td>0.9699</td>
<td>0.69</td>
</tr>
<tr>
<td>50</td>
<td>36</td>
<td>0.9654</td>
<td>0.87</td>
</tr>
<tr>
<td>100</td>
<td>36</td>
<td>0.9516</td>
<td>1.74</td>
</tr>
<tr>
<td>200</td>
<td>24</td>
<td>0.9418</td>
<td>3.21</td>
</tr>
<tr>
<td>500</td>
<td>24</td>
<td>0.9298</td>
<td>6.44</td>
</tr>
<tr>
<td>1000</td>
<td>24</td>
<td>0.9246</td>
<td>10.45</td>
</tr>
<tr>
<td>10000</td>
<td>12</td>
<td>0.9111</td>
<td>40.82</td>
</tr>
<tr>
<td>20000</td>
<td>12</td>
<td>0.9101</td>
<td>56.70</td>
</tr>
<tr>
<td>100000</td>
<td>12</td>
<td>0.9070</td>
<td>100.00</td>
</tr>
</tbody>
</table>

Evaluation on MovieLens  I also evaluated this method on the MovieLens dataset, using the BRISMF#52 model (page 55), with different $K$ values. I obtained the following results:

<table>
<thead>
<tr>
<th>$K$</th>
<th>test RMSE</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>50</th>
<th>100</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.8569</td>
<td>0.8429</td>
<td>0.8361</td>
<td>0.8336</td>
<td>0.8325</td>
<td>0.8317</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.8437</td>
<td>0.8375</td>
<td>0.8351</td>
<td>0.8342</td>
<td>0.8335</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The accuracy of the “U” variant is close to the simple method. Like on page 55 I examined the relation between the following 3 variables: the number of users; $K$; and the difference between test RMSE and test RMSE of the U variant. I concluded the same as on page 55: more users yield more favorable difference, and larger $K$ yields the opposite.

Remarks  I remark that a similar retraining approach was proposed by Rendle and Schmidt-Thieme [41] on 2008/10/25 (mine was published on 2008/08/24 [58]). The differences between mine and their approach are the followings:

- they also propose to handle new ratings of items: after recomputing the changed $p_u$-s, they propose to recompute changed $q_i$-s.
- they do not differentiate between $n^{*}_1$ and $n^{*}_2$, i.e. in the retraining phase they run their algorithm for exactly the same number of epochs as in the first phase.

Thesis 2.1 I proposed a modification for BRISMF that can efficiently handle new ratings of users. I pointed out by extensive experiments on the Netflix Prize dataset, that the method is efficient in terms of both speed and accuracy. As a result, only a small subset of the database is enough to get a reliable estimate of $Q$, and then fixing this matrix we can recompute $P$ for each user separately, using all of her ratings. I also evaluated the method on the MovieLens dataset with success, and on the Jester dataset with moderate success.
CHAPTER 4. IMPROVING ON THE SPEED OF MATRIX FACTORIZATION

4.2 Fast and accurate matrix factorizations by parameter optimization

From the scalability point of view, it is interesting and important to investigate the relationship between speed and accuracy. I ran numerous randomly parameterized MFs with \( K = 40 \), and collected the best accuracies in each epoch, and then optimized the parameters. Table 4.6 summarizes the results. One epoch takes 80 seconds (\( K = 40 \)), and the initialization takes an additional 40 seconds (loading the full database into the memory). Running times were measured on a 2 GHz Intel Pentium M (Dothan).

I described this idea in [58, 60].

An RMSE of 0.9071 can be achieved within 200 seconds (including the time to train with the 100 million available ratings and evaluate on the Probe10)! For a comparison: Netflix’s Cinematch algorithm can achieve Quiz RMSE 0.9514, so this fast solution achieves more than 5.6% improvement on Cinematch. I remark that running times differ on the Core 2 Quad PC (using 1 core): it is 3\times faster than the Pentium M Dothan.

Note that even with 1 epoch, Probe10 RMSE = 0.9179 can be achieved. This means, that the algorithm processes the database of ratings only once. Users processed first have a very inaccurate model, since \( Q \) contains only noise at the beginning of the epoch.

In Table 4.6, 1.1 epoch means that the model was trained for one epoch and then the ratings of the first 1/10 of users were used for another epoch.

<table>
<thead>
<tr>
<th>Epoch</th>
<th>Training Time (sec)</th>
<th>Probe10 RMSE</th>
<th>Quiz RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>120</td>
<td>0.9179</td>
<td>0.9188</td>
</tr>
<tr>
<td>1.1</td>
<td>128</td>
<td>0.9147</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>0.9071</td>
<td>0.9077</td>
</tr>
<tr>
<td>3</td>
<td>280</td>
<td>0.9057</td>
<td>0.9061</td>
</tr>
<tr>
<td>4</td>
<td>360</td>
<td>0.9028</td>
<td>0.9026</td>
</tr>
<tr>
<td>5</td>
<td>440</td>
<td>0.9008</td>
<td>0.9012</td>
</tr>
<tr>
<td>6</td>
<td>520</td>
<td>0.9002</td>
<td>0.9005</td>
</tr>
</tbody>
</table>

**Evaluation on Jester** I also evaluated this idea on the Jester dataset for the first 8 epochs, using \( K = 10 \). I obtained 8 models with the following results:

<table>
<thead>
<tr>
<th>Epoch</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.4804</td>
</tr>
<tr>
<td>2</td>
<td>4.3142</td>
</tr>
<tr>
<td>3</td>
<td>4.2300</td>
</tr>
<tr>
<td>4</td>
<td>4.1823</td>
</tr>
<tr>
<td>5</td>
<td>4.1546</td>
</tr>
<tr>
<td>6</td>
<td>4.1335</td>
</tr>
<tr>
<td>7</td>
<td>4.1158</td>
</tr>
<tr>
<td>8</td>
<td>4.1021</td>
</tr>
</tbody>
</table>

On page 54, BRISMF#23 with \( K = 10 \) reaches 4.0766 in the 100th epoch. With the same number of features, this approach can reach worse, but acceptable performance much faster.

**Evaluation on MovieLens** I also evaluated this idea on the MovieLens dataset, using \( K = 50 \). I obtained the following test RMSEs:
4.2. FAST AND ACCURATE MF BY PARAMETER OPTIMIZATION

Epoch | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 20
---|---|---|---|---|---|---|---|---|---
RMSE | 0.9618 | 0.8998 | 0.8755 | 0.8631 | 0.8556 | 0.8505 | 0.8470 | 0.8449 | 0.8374

Recall that BRISMF#52 reaches 0.8336 in the 72th training epoch. The best available performances in different epochs here are more different than on the Netflix dataset. This may be due to the small number of users. On the Netflix dataset, only 1 or 2 epochs are enough to set the item features appropriately, and then the next epoch can finalize the user features. On the MovieLens dataset, this is not possible because the number of users and items are small and almost equal. On page 43 I noted that subsampling the users increases the number of training epochs. That may be due to the redundancy in the dataset: doubling the number of users yields less than twice as much information about items, and the redundancy yields less training epochs for the item features.

When the number of users is so small like in the MovieLens database, then we need to iterate over the database many times, it is not possible to found such a good solution as in the case of Netflix dataset. However, in many situations, the above results can offer a good trade-off between training time and accuracy.

**Conclusion**

To my best knowledge, running times and accuracies on the Netflix dataset reported here are favorable compared to any other method published in the field of Collaborative Filtering. Though this statement might seem somewhat speculative since authors do not tend to publish running times, I can support it with the following arguments:

- features are trained simultaneously;
- the number of epochs is small;
- I use a gradient descent algorithm, which is the fastest if we can keep the number of required gradient steps low, which is exactly the case.

Note that given $n^*$, the number of epochs, there are $n^* \cdot |\mathcal{R}| \cdot K$ variable updates in $P$ and $Q$ during the training. The presented methods can achieve the favorable RMSEs while keeping the number of features ($K$) and the number of epochs ($n^*$) low; consequently they are also favorable in terms of time requirement.

I remark that the idea to get fast and accurate MFs has also been suggested later by a Netflix Prize contestant, Kulik [30]. Although his description is not clear, he suggest the following: have different $\eta$-s and $\lambda$-s for each epoch. First, optimize the performance only for the first epoch. Then save the model. Now optimize $\eta$-s and $\lambda$-s for the second epoch. The second epoch starts on top of the result of the first model. Do it similarly for additional epochs. He claims that with this approach, he can reach Quiz RMSE = 0.8840 in 8 minutes within 3 epochs, although very important details of his method are kept secret.

**Thesis 2.2** I proposed to optimize the parameters of BRISMF such that the best RMSE is achieved in the first few epochs. I pointed out by experiments on the Netflix Prize dataset, that only 200 seconds is enough to get a model in 2 epochs with Probe10 RMSE = 0.9071, on a Pentium M 2GHz (Dothan) CPU. I also tried the idea on the Jester and MovieLens datasets. The results are acceptable. On the MovieLens dataset, I concluded that if we had more users, the results would be more favorable.
4.3 Applications of the Sherman-Morrison formula

In this section I explore how the Sherman-Morrison formula (SMF) can be used to speed up alternating least squares based matrix factorizations. I also explored it in [36]. In this section both explicit and implicit feedback CF will be considered. I also propose methods for handling new ratings of users by SMF, and to speed up greedy forward / backward feature selection.

Definition Sherman-Morrison formula (SMF): suppose that $A \in \mathbb{R}^{n \times n}$ is an invertible matrix, $u, v \in \mathbb{R}^{n}$ are column vectors, and $1 + v^T A^{-1} u \neq 0$. Then the inverse of $A + uv^T$ can be computed as:

$$(A + uv^T)^{-1} = A^{-1} - \frac{(A^{-1} u)(v^T A^{-1})}{1 + v^T A^{-1} u}$$

This formula allows us to compute the inverse of a modified matrix in $O(n^2)$, when the modification is an addition of a dyadic product ($uv^T$) and the inverse of the original matrix is precomputed.

Note that when we extend $A$ with a new row and a new column, the inverse of the new $(n+1) \times (n+1)$ matrix can be computed in $O(n^2)$ time using the SMF (assuming that the above conditions are met): first we extend $A$ and $A^{-1}$ with the last row and last column of the $(n+1) \times (n+1)$ identity matrix, and then change the $n+1$-th column of the new $A$ and update the inverse with the SMF. Then we change the $n+1$-th row.

Deleting a row and column can be treated similarly. In the followings we investigate some recommender algorithms where the Sherman-Morrison formula is applicable. I remark the $1 + v^T A^{-1} u \neq 0$ will always hold in the followings, because $A$ will always be positive semidefinite, and $v = v$ will be true as well.

4.3.1 ALS on explicit feedback datasets

It has been noted in Section 2.3.2 (page 31), that the recomputation of $p_u$ requires $O(K^2 \cdot n_u + K^3)$ time. When $n_u \leq K$, then $O(K^3)$ is dominant. Now I show how to eliminate it. Recall that from eq. (2.25), we have $A_u = \sum_{i: (u,i) \in R} q_i q_i^T$. Algorithm 5 computes the inverse of $(\lambda n_u I + A_u)$ in $O(K^2 \cdot n_u)$ time (excluding the $O(K^3)$ term), starting with the regularization term $\lambda n_u I$, and then successively adding the $q_i q_i^T$ dyadic products.

For users with $n_u \geq K$ ratings, the application of Sherman-Morrison formula does not increase the running time (in the $O(\cdot)$ sense), while for users with $n_u < K$, it can greatly speed it up. To recompute $P$, the time complexity is reduced from $O(K^2 \cdot |R| + N \cdot K^3)$ to $O(K^2 \cdot |R|)$, yielding exactly the same $P$ as the original ALS. $Q$ can be treated similarly.

4.3.2 ALS on implicit feedback datasets

In the case of the implicit feedback, $R$ is fully filled, i.e. $|R| = NM$, as opposed to the explicit case, where only a small subset of its values is known, i.e. $|R| \ll NM$. In the case of NP competition, an average user rates only each 85th movie. If we consider the NP dataset as implicit feedback—substituting known ratings with 1 and
4.3. APPLICATIONS OF THE SHERMAN-MORRISON FORMULA

Input: $R$: matrix of ratings, $R$: training set.
$K$: number of features, $Q \in \mathbb{R}^{M \times K}$: the item feature matrix,
$\lambda$: regularization parameter, $u$: user id, $n_u$: number of ratings of $u$.

Output: $p_u$: the recomputed user feature vector

1. Let $I \in \mathbb{R}^{K \times K}$ be the identity matrix
2. Let $B_u \in \mathbb{R}^{K \times K} = \lambda n_u I$ // The covariance matrix and regularization term
3. Let $W_u \in \mathbb{R}^{K \times K} = \begin{cases} 0 \cdot I, & \text{if } \lambda = 0 \\ \frac{1}{\lambda n_u} \cdot I, & \text{otherwise} \end{cases}$ // The inverse of $B_u$
4. Let $d_u \in \mathbb{R}^{K \times 1} = 0$ // The input-output covariance vector
5. for each $i : (u, i) \in R$
   6. Let $B_u = B_u + q_i q_i^T$
   7. Let $W_u = W_u - \frac{(W_u q_i)(q_i^T W_u)}{1 + q_i^T W_u q_i}$ // Applying the Sherman-Morrison formula
   8. Let $d_u = d_u + q_i \cdot r_{ui}$
   9. end
10. Let $p_u = W_u d_u$
11. end

Algorithm 5: Algorithm to recompute $p_u$ in $O(K^2 \cdot n_u)$ time for explicit feedback datasets.

unknown ratings with 0, indicating which movies users rated and which not—then a recommendation algorithm has to deal with 85 times more data than at explicit feedback, i.e. $|R|/|R^+| = 85$, where $R^+$ denotes the elements of $R$ where implicit feedback exists (i.e. the user rated the movie).

Hu et al. suggested an elegant method to handle implicit feedback datasets, more specifically, for recommending IPTV movies for users based on their watching habits. The key idea behind their approach is to assign a preference (here: $r_{ui}$) and a confidence level $c_{ui}$ for each element of $R$. Preference typically takes values of 0 and 1, indicating whether a user watched a particular movie or not. Confidence level is related to the amount of time the user spent watching a particular movie. They define the optimal model as:

$$(P^*, Q^*) = \arg\min_{P, Q} \left( \sum_{(u, i) \in R} c_{ui} \cdot e_{ui}^2 \right) + \lambda \left( \sum_u p_u^T p_u \right) + \lambda \left( \sum_i q_i^T q_i \right),$$

where $e_{ui} = r_{ui} - p_u^T q_i$ is the prediction error. Note that $R$ contains all $(u, i)$ pairs regardless of whether user $u$ has watched movie $i$ or not. The authors use one restriction, namely if $u$ has not watched $i$, then $r_{ui} = r_0 = 0$ and $c_{ui} = c_0 = 1$, where $r_0$ and $c_0$ are predefined constants. Similarly to the rating-based case, the authors used ALS to solve the problem, but with the following two differences:

- handling confidence levels: the covariance matrix $A_u$ and covariance vector $d_u$ contains now a confidence-weighted sum.
- the sums are taken over all items (since all users rated all items).

The authors propose an efficient solution to the second issue: when recomputing $P$, they calculate an initial covariance matrix $A_0$ and a vector $d_0$ for a virtual user who watched
CHAPTER 4. IMPROVING ON THE SPEED OF MATRIX FACTORIZATION

no movies:

\[ A_0 = \sum_i c_0 \cdot q_i q_i^T, \quad d_0 = \sum_i c_0 \cdot r_0 \cdot q_i. \]

Recall that \( \mathcal{R}^+ \) denotes the set of \((u, i)\) values of positive feedback. In this case, let \(\mathcal{R}^+ = \{(u, i) : r_{ui} \neq r_0 \lor c_{ui} \neq c_0\}\). For user \(u\), they start with that precomputed covariance matrix, and for each movie watched by the user, they replace the did-not-watch-assumption. Then it yields:

\[ A_u = A_0 + \sum_{i: (u, i) \in \mathcal{R}^+} (-c_0 + c_{ui}) \cdot q_i q_i^T, \quad d_u = d_0 + \sum_{i: (u, i) \in \mathcal{R}^+} (-c_0 \cdot r_u + c_{ui} \cdot r_{ui}) \cdot q_i. \]

Then \(p_u\) is computed by RR:

\[ p_u = (\lambda I + A_u)^{-1} d_u. \] (4.1)

The authors note that the running time of recomputing \(P\) or \(Q\) are \(O(K^2 \cdot |\mathcal{R}^+| + K^3 \cdot N)\) and \(O(K^2 \cdot |\mathcal{R}^+| + K^3 \cdot M)\), resp. They show experimentally that larger \(K\) values yield better prediction performance.

I will show that the Sherman-Morrison formula can be used here to speed up the recomputation of both \(P\) and \(Q\) to \(O(K^2 \cdot |\mathcal{R}^+|)\). Here I present the solution for \(P\). For \(Q\), it can be derived analogously.

Note that \((\lambda I + A_0), d_0\) and \((\lambda I + A_0)^{-1}\) can be computed in \(O(K^2 \cdot M)\): I initialize a \(K \times K\) matrix with \(\lambda I\) and then add \(c_0 q_i q_i^T\) values for each \(i\), and also initialize the inverse of that (diagonal) matrix, and keep maintained by the Sherman-Morrison formula. Note that the regularization term is added at the beginning, not at the end, since the Sherman-Morrison formula cannot be applied for such modification; and the regularization term is the same for all users, as opposed to the explicit feedback case, where it is \(\lambda u_n I\).

For user \(u\), start with \(A_u := (\lambda I + A_0), d_u := d_0\) and \(W_u := (\lambda I + A_0)^{-1}\). Iterate over \(i : (u, i) \in \mathcal{R}^+\), and update \(A_u\) and \(d_u\) as

\[ A_u := A_u + (-c_0 + c_{ui}) \cdot q_i q_i^T, \quad d_u := d_u + (-c_0 \cdot r_u + c_{ui} \cdot r_{ui}) \cdot q_i. \]

\(W_u\) can be maintained with the SMF, since \(A_u\) is modified by the dyadic product \((-c_0 + c_{ui}) \cdot q_i q_i^T\). After the iteration, \(p_u\) can be computed: \(p_u = W_u d_u\). Thus, the time to recompute \(P\) is \(O(K^2 \cdot M + K^2 \cdot |\mathcal{R}^+|)\). Assuming that each movie has at least one non-zero rating, then \(M < |\mathcal{R}^+|\), and the time complexity reduces to \(O(K^2 \cdot |\mathcal{R}^+|)\).

Note that the confidence-weighted ALS is easy to evaluate on implicit feedback datasets: for \(r_{ui}\)'s in the test set, we can set \(c_{ui} = 0\) in the training set, thus no information about those ratings will be contained in the model. Note that SMF can be used here to speed up cross-validation: assuming \(Q\) is fixed, moving one example from the training set to the test set (or vice versa), the new \(p_u\) can be computed in \(O(K^2)\).

4.3.3 Handling new ratings of users with ALS

In real recommender systems users can rate items, and expect to be provided with instant recommendations based on all available information.

The Sherman-Morrison formula can be used to efficiently handle this for ALS-based recommender algorithms assuming \(Q\) is fixed. Let us assume that the sets of users
and items do not change. Furthermore, we may assume that there are more users than items. When user $u$ signs in to the service that uses the recommender system, we can calculate $A_u, W_u, d_u$ and $p_u$ based on her ratings. When $u$ rates a new item (or cancels a rating), we can update these variables and get the new $p_u$ value in $O(K^2)$ time. We can do the same for $q_i$ variables, however, for items with many ratings the benefit of instant recomputation of features may be negligible. Note that recomputing $p_u$ using the traditional way (i.e. inverting $A_u$, not using the SMF) requires $O(K^3)$ time.

Note that when a user changes one of its ratings, neither $A_u$ nor $W_u$ changes (recall equations (2.25)–(2.26)), only $d_u$. However, computing $W_u d_u$ requires $O(K^2)$ time in this case as well, unless we store the precomputed value of $W_u Q[u]^T$ (recall that $d_u = Q[u]^T r_u$), in which case it is reduces to $O(K)$ at the cost of increased memory requirement. A good trade-off may be to store $W_u Q[u]^T$ restricted to the last few ratings of the user, i.e. restricted to those ratings that are likely to be changed.

I remark that the Sherman-Morrison formula has already been proposed for recommendation systems to handle new data in a web recommendation system [53], where a square matrix of probabilities gets updated.

### 4.3.4 Backward and forward feature selection

The ridge regression of many predictors may improve the performance measured by RMSE. For example, team BellKor combined 107 predictors in [10]. When we have many predictors, it is crucial to be able to select the best performing ones. A straightforward way is to apply the greedy forward or backward feature selection, although there are other ways, for example, applying Lasso regression [61]. However, if the goal is to get the best possible RMSE with the fewest possible number of predictors, then greedy feature selection is more straightforward, and it also allows for a user to manually set a threshold on the number of predictors.

Suppose that we have $F$ different predictors (in the above example $F = 107$), and for notational simplicity I assume that RR with regularization parameter $\lambda$ is optimized on $R$, not on a separate validation set. Let $\hat{r}_{ui}(f)$ denote the prediction of the $f$-th predictor ($f \in \{1, \ldots, F\}$) for the ($u, i$)-th rating, and $\hat{r}_{ui} = (\hat{r}_{ui}(1), \ldots, \hat{r}_{ui}(f))^T \in \mathbb{R}^{F \times 1}$ denote the prediction vector for the ($u, i$)-th rating. Assume that the variables $B = \lambda I + \sum_{(u,i) \in R} \hat{r}_{ui} \hat{r}_{ui}^T$ and $d = \sum_{(u,i) \in R} \hat{r}_{ui} p_{ui}$ are precomputed.

The greedy forward feature selection (GFFS) works as follows: suppose we have selected $n$ out of the $F$ predictors. Now we examine, how each one of the rest $F - n$ predictors can further improve RMSE. We select the best predictor from these $F - n$ predictors, and extend the set of already selected $n$ predictors with the new one. Then the process is repeated with the $n + 1$ predictors, until the difference of the actual RMSE and the RMSE of using all predictors goes below a threshold. We begin with $n = 0$.

The greedy backward feature selection (GBFS) works in the opposite way: we begin with $n = F$, and exclude those predictions the removal of which minimally increases RMSE.

Both GFFS and GBFS requires to run linear regression for $1 + \ldots + F = O(F^2)$ times. Since RR requires $O(n^3)$ time for $n$ variables, a complete GFFS or GBFS requires $O(F^5)$ time. Now I show how the Sherman-Morrison formula can be applied to speed up the process to $O(F^4)$.

For GFFS, suppose that we have selected $n$ out of the $F$ predictors. Let $B[n] \in \mathbb{R}^{n \times n}$
and $d[n] \in \mathbb{R}^{n \times 1}$ denote the restriction of $B$ and $d$ to the selected predictors resp. Assume that $B[n]^{-1}$ is precomputed. To include a new predictor, we can extend $B[n]$ to $B[n+1]$ by adding a new row and a new column. Thus, the inverse of $B[n+1]$ can be computed in $O(F^2)$ using the SMF, instead of $O(F^3)$.

For GBFS, the same applies, except that $B[n]$ is not extended by a row and a column, but deleted. Thus, applying the Sherman-Morrison formula we can reduce the time complexity of the greedy backward or forward feature selection algorithms from $O(F^5)$ to $O(F^4)$, with the RMSE as performance measure.

**Thesis 2.3** I examined the applicability of the Sherman-Morrison formula for alternating least squares based matrix factorizations. I showed that by using the Sherman-Morrison formula ALS based algorithms can be speed up both on explicit and implicit feedback datasets. For explicit feedback, I improved from $O(M)$ the time complexity of the recomputation of the user feature matrix ($P$). For implicit feedback $O(K^2 \cdot |R^+| + K^3 \cdot N)$ is reduced to $O(K^2 \cdot |R^+|)$. For the item feature matrix $Q$ analog results hold. If a user provides a new rating, recomputing $p_u$ can be speed up from $O(K^3)$ to $O(K^2)$.

I also pointed out that SMF can also be applied to reduce the complexity of greedy feature selection algorithms on $F$ predictors from $O(F^5)$ to $O(F^4)$.

### 4.4 Applications of kernel ridge regression

Kernel ridge regression (KRR) has been applied for the NP dataset first by Paterek [32]. He used KRR to post-process the results of an MF: after determining $P$ and $Q$ with an MF algorithm, he applied KRR with a Gaussian kernel variant to recompute $p_u$-s.

In this section I propose the usage of KRR for users with $n_u < K$ ratings, using linear kernel function: $K(q_1, q_2) := q_1^T q_2$.

Recall that in equations (2.23)–(2.26), ALS recalculates $p_u$ in the following way:

$$p_u = (\lambda n_u I + Q[u]^T Q[u])^{-1} (Q[u]^T r_u).$$

The equivalent dual formulation involving the Gram matrix $Q[u]Q[u]^T$ is the following [32]:

$$p_u = Q[u]^T (\lambda n_u I + Q[u]Q[u]^T)^{-1} r_u. \quad (4.2)$$

I proposed the usage or linear KRR for ALS in [32].

#### 4.4.1 KRR on explicit feedback datasets

Note that the size of the Gram matrix is $n_u \times n_u$, as opposed to $K \times K$ of $Q[u]^T Q[u]$. To recompute $p_u$, the following operations are needed:

- calculate $\lambda n_u I + Q[u]Q[u]^T$, which is $O(K \cdot n_u^2)$.
- invert that matrix, which is $O(n_u^3)$.
- multiply by $r_u$ from right, which is $O(n_u^2)$.
- multiply by $Q[u]^T$ from left, which is $O(K \cdot n_u)$.

Thus, $p_u$ can be recomputed in $O(K \cdot n_u^2 + n_u^3)$ time. Assuming $n_u \leq K$, it is $O(K^2 \cdot n_u^2)$. Thus, we have two new algorithms to recompute $p_u$ (or $q_i$), which outperform in terms of computational complexity the original algorithm that has $O(K^2 \cdot n_u + K^3)$:
1. if \( n_u \leq K \): KRR, with \( O(K \cdot n_u^2) \)
2. if \( n_u \geq K \): Sherman-Morrison formula, with \( O(K^2 \cdot n_u) \)

Note that for users with very few ratings, the ratio \( K/n_u \) can be huge, which means that the application of KRR can greatly speed up the learning process in such a case. This allows to increase \( K \) while not worrying too much about the running time.

Note that the proposed method is not usable for implicit feedback datasets, where for all \( u \): \( n_u = M \) (unless \( K > M \), which would be unusual).

**Evaluation**

I compared the running time of using KRR and using the formula of eq. (2.26) on the Netflix Prize dataset. The experiments were performed on a Intel Core2 Quad Q9300 CPU on 3.3GHz, using only 1 core. I refer to the first method as KRR, and to the second method as RR.

First, I compared whether the two methods provide the same result or not. Although theoretically the two results should be the same, they could differ due to floating point computations. The experiments showed, that RMSE values are equal (up to 5 digits, I used floating point numbers with 32 bit precision).

Next, I partitioned the dataset based on the number of ratings of the users. I created one partition for users with \( 0 < n_u < 10 \), one partition for \( 10 \leq n_u < 20 \), and so on. I evaluated the running time of recomputing the user feature vectors in each partition. Since the number of users in each partition differs, the average time per user is of interest.

Table 4.7 summarizes the results: when the number of features is 100 (i.e. \( K = 100 \)), then to recompute the feature vector of a user, whose number of ratings are between 30 and 40, it takes \( 29.682 \cdot 10^{-4} \) seconds with RR, while KRR needs only \( 5.661 \cdot 10^{-4} \) seconds.

One can verify, that when \( n_u \) is close to \( K \), then the running time of RR and KRR are roughly equal. When \( n_u \ll K \), KRR is much faster than RR.

**4.4.2 Handling new ratings of users with KRR**

The Sherman-Morrison formula can be applied when we extend a matrix with one row or column. When we apply KRR, the regularized Gram matrix \( \lambda n_u I + Q[u]Q[u]^T \) is extended by one row and one column, thus, the SMF is applicable. The following steps are required to compute the new \( p_u \), according to (4.2):

- Compute the new row and column of the matrix: \( O(K \cdot n_u) \).
- Update the inverse of the Gram matrix: \( O(n_u^2) \).
- multiply by \( r_u \) from right: \( O(n_u^2) \).
- multiply by \( Q[u]^T \) from left: \( O(K \cdot n_u) \).

Total: \( O(K \cdot n_u + n_u^2) \).

Let us compare the time complexity of adding or removing one rating for users with \( n_u < K \) using the Sherman-Morrison formula (as proposed in Section 4.3.3) and KRR:

- Sherman-Morrison formula: \( O(K^2) \).
- KRR: \( O(K \cdot n_u + n_u^2) \), which reduces to \( O(K \cdot n_u) \), since \( n_u < K \).

Clearly, KRR is better if \( n_u < O(K) \).

**Thesis 2.4** I examined the applicability of linear kernel ridge regression for ALS based matrix factorizations. I showed, how linear kernel ridge regression can speed up the
CHAPTER 4. IMPROVING ON THE SPEED OF MATRIX FACTORIZATION

Table 4.7: Average running time of recomputing a $K$-dimensional user feature vector, when the number of ratings of the user is in an interval. First I report the running time of RR (ridge regression), then the running time of KRR, both in $10^{-4}$ seconds.

<table>
<thead>
<tr>
<th>$n_u$</th>
<th>RR (s)</th>
<th>KRR (s)</th>
<th>RR (s)</th>
<th>KRR (s)</th>
<th>RR (s)</th>
<th>KRR (s)</th>
<th>RR (s)</th>
<th>KRR (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-10</td>
<td>0.360</td>
<td>0.272</td>
<td>0.833</td>
<td>0.284</td>
<td>4.065</td>
<td>0.343</td>
<td>24.03</td>
<td>0.461</td>
</tr>
<tr>
<td>10-20</td>
<td>0.511</td>
<td>0.706</td>
<td>1.235</td>
<td>0.985</td>
<td>4.470</td>
<td>1.123</td>
<td>24.80</td>
<td>1.344</td>
</tr>
<tr>
<td>20-30</td>
<td>0.547</td>
<td>1.191</td>
<td>1.627</td>
<td>1.835</td>
<td>5.819</td>
<td>2.615</td>
<td>26.95</td>
<td>2.961</td>
</tr>
<tr>
<td>30-40</td>
<td>0.582</td>
<td>1.977</td>
<td>1.760</td>
<td>2.747</td>
<td>7.274</td>
<td>4.800</td>
<td>29.68</td>
<td>5.661</td>
</tr>
<tr>
<td>40-50</td>
<td>0.601</td>
<td>3.114</td>
<td>1.770</td>
<td>3.992</td>
<td>8.953</td>
<td>7.673</td>
<td>33.01</td>
<td>9.349</td>
</tr>
<tr>
<td>50-60</td>
<td>1.834</td>
<td>5.713</td>
<td>10.27</td>
<td>10.76</td>
<td>36.60</td>
<td>14.10</td>
<td>216.1</td>
<td>16.59</td>
</tr>
<tr>
<td>60-70</td>
<td>10.98</td>
<td>14.62</td>
<td>40.63</td>
<td>20.68</td>
<td>222.0</td>
<td>24.41</td>
<td></td>
<td></td>
</tr>
<tr>
<td>70-80</td>
<td>11.51</td>
<td>19.21</td>
<td>45.04</td>
<td>28.79</td>
<td>229.0</td>
<td>34.08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80-90</td>
<td>11.95</td>
<td>23.97</td>
<td>49.36</td>
<td>37.32</td>
<td>238.3</td>
<td>44.91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>90-100</td>
<td>11.71</td>
<td>29.66</td>
<td>53.70</td>
<td>47.36</td>
<td>248.1</td>
<td>58.17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100-110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>57.42</td>
<td>57.20</td>
<td>257.7</td>
<td>72.51</td>
</tr>
<tr>
<td>130-140</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>63.95</td>
<td>127.7</td>
<td>287.9</td>
<td>164.7</td>
</tr>
<tr>
<td>170-180</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>66.09</td>
<td>191.7</td>
<td>330.5</td>
<td>268.5</td>
</tr>
<tr>
<td>190-200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>67.33</td>
<td>241.6</td>
<td>351.0</td>
<td>343.2</td>
</tr>
<tr>
<td>210-220</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>69.59</td>
<td>320.1</td>
<td>367.4</td>
<td>427.1</td>
</tr>
<tr>
<td>0-100</td>
<td>32.59</td>
<td>12.56</td>
<td>214.4</td>
<td>15.07</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

recomputation of $p_u$ in ALS for users with $n_u < K$ ratings: the original $O(K^2 \cdot n_u + K^3)$ is reduced with SMF to $O(K^2 \cdot n_u)$, and with KRR to $O(K \cdot n_u^2)$. I performed experiments to show how much computational performance gain can be brought by KRR.

I pointed out that the addition or deletion of ratings can be handled in $O(K^2)$ by SMF, and if $n_u < K$, then in $O(K \cdot n_u)$ with KRR + SMF, while the traditional way requires $O(K^3)$ operations. These proposed methods allow ALS to be used in practical recommender systems, where the system must respond instantly, when a new user rating arrives.

4.5 Summary

In this section I proposed many approaches to speed up matrix factorization and handle new ratings of users:

- I proposed a modification for BRISMF that can efficiently handle new ratings of users.
- I proposed to apply parameter optimization to get accurate BRISMF models in few epochs.
- I examined the applicability of the Sherman-Morrison formula for ALS based matrix factorizations.
- I examined the applicability of linear kernel ridge regression for ALS based matrix factorizations.
Chapter 5

Content-based filtering

In this chapter I introduce a method that can handle movie (item) descriptions. The focus is on the prediction of ratings on new movies, since in this case the only data we have about movies is the metadata, thus, based only on that we should be able to tell users whether they will like a movie or not. It is also an important question whether metadata is predictive enough or not. I will examine that in the followings. My proposed methods are usable for explicit feedback data.

5.1 Using NSVD1 for content-based filtering

In this section I propose a pure movie-metadata-based CBF framework. I described this approach in 37.

Paterek’s NSVD1 algorithm (described as a neural network on page 58) can be interpreted in the following way: first, users are represented by \( M \) dimensional binary vectors. Such a vector reflects which movies are rated by a user and which not, and is used to infer \( p_u \) by a linear transformation, denoted by the matrix \( W \). As opposed to a learning algorithm for MF that finds \( P \) and \( Q \), a learning algorithm for NSVD1 finds \( Q \) and \( W \). Users can also be represented by vectors derived not only from their ratings but also from their metadata (demographic data). From the viewpoint of a learning algorithm, there is no difference between a vector describing user metadata and a vector containing 1 at the movies rated by the user. In other words, we can consider the list of movie IDs rated by the user as a textual description about the user.

One can run the NSVD1 algorithm interchanging the role of users and movies. Then movies are represented by vectors indicating which users rated them. Movies can be represented also by their textual descriptions, with the vector space model commonly used in text-mining. For clarity, I refer to the original version as user-NSVD1, and to its dual as movie-NSVD1.

Algorithm 6 is the training algorithm for movie-NSVD1, based on 57, using the following notations:

- \( \mathbf{W} \in \mathbb{R}^{C \times K} \) is the transformation matrix that transforms movie metadata to feature vectors.
- \( \mathbf{w}_l \in \mathbb{R}^{K \times 1} \) is the transpose of the \( l \)-th row of \( \mathbf{W} \).
- \( \mathbf{X} \in \mathbb{R}^{N \times C} \) contains metadata about the movies; note that \( \mathbf{X} \) is fully specified as opposed to \( \mathbf{R} \), since we may assume that movie metadata are correct and complete,
while most elements of $\mathbf{R}$ are unknown.

- $\mathbf{x}_i \in \mathbb{R}^{C \times 1}$ is the transpose of the $i$-th row of $\mathbf{X}$.
- $x_{il}$ is the $(i, l)$-th element of $\mathbf{X}$.

With this notation:

$$q_i = \mathbf{W}^T \mathbf{x}_i = \sum_{l: x_{il} \neq 0} x_{il} \cdot w_l, \quad (5.1)$$

$$Q = \mathbf{XW}, \quad (5.2)$$

$$\hat{R} = \mathbf{PQ}^T = \mathbf{P}(\mathbf{XW})^T, \quad (5.3)$$

where $\hat{R}$ is the approximation (prediction) of $\mathbf{R}$.

---

1. Create random model.
2. repeat
3. one epoch:
4. for $i \leftarrow 1$ to $M$ do
5. $q_i \leftarrow \mathbf{W}^T \mathbf{x}_i$
6. $a \leftarrow q_i$
7. foreach $u : (u, i) \in \mathcal{R}$ do
8. $\hat{r}_{ui} \leftarrow b_u + c_i + p_u^T q_i$
9. $e_{ui} \leftarrow r_{ui} - \hat{r}_{ui}$
10. $b \leftarrow p_u$
11. $p_u \leftarrow p_u + \eta \cdot (e_{ui} \cdot q_i - \lambda \cdot p_u)$
12. $q_i \leftarrow q_i + \eta \cdot (e_{ui} \cdot b - \lambda \cdot q_i)$
13. $b_u \leftarrow b_u + \eta \cdot (e_{ui} - \lambda \cdot b_u)$
14. $c_i \leftarrow c_i + \eta \cdot (e_{ui} - \lambda \cdot c_i)$
15. end
16. $d \leftarrow 1/(\mathbf{x}_i^T \mathbf{x}_i)$
17. foreach $l : x_{il} \neq 0$ do
18. $w_l \leftarrow w_l + d \cdot x_{il} \cdot (q_i - a)$
19. end
20. end
21. until RMSE on the test set decreases ;

Algorithm 6: Training algorithm for movie-NSVD1

Line 5 of Algorithm 6 computes the movie feature vector. Lines 8–14 updates the movie feature vector (together with the user feature vector and the two biases), in the same way as in BRISMF. Line 16 is new in this algorithm compared to the NSVD1 algorithm in [57]; it allows $x_{il}$ to be arbitrary; in the original version $x_{il}$ is set to $1/\sqrt{n_i}$ for $(l, i) \in \mathcal{R}$, and 0 otherwise, thus, for any $i$, $\sum_l x_{il}^2 = 1$.

Lines 17–19 backpropagate the change in $q_i$ to $\mathbf{W}$. After this step, $q_i$ is equal to $\mathbf{W}^T \mathbf{x}_i$, i.e. as defined in line 5. To prove it, I rewrite that 3 lines in the following form:

$$\mathbf{W} \leftarrow \mathbf{W} + \frac{x_i}{\mathbf{x}_i^T \mathbf{x}_i} (q_i - a)^T, \quad (5.4)$$
and check what line 5 yields with the new $W$:

$$
\left( W + \frac{x_i}{x_i^T x_i} (q_i - a)^T \right)^T x_i = \\
= \left( W^T + (q_i - a) \frac{x_i^T}{x_i^T x_i} \right) x_i = W^T x_i + (q_i - a).
$$

Since $a$ is $W^T x_i$ by definition, they are equal.

Let us analyze the computational complexity of Algorithm 6:

- computing $p_u$: $O(K \cdot \sum_l: x_{il} \neq 0 \cdot 1)$, executed for each $i$.
- lines 8–14: $O(K)$, executed for $|R|$ times.
- line 16: $\sum_l: x_{il} \neq 0 \cdot 1$, executed for each $i$.
- lines 17–19: $K \cdot \sum_l: x_{il} \neq 0 \cdot 1$, executed for each $i$.

Let $\mathcal{X} = \{(i, l): x_{il} \neq 0\}$ denote the set of indices of non-zero elements of matrix $X$. With this notation, the time requirement of one epoch is:

$$
O(K \cdot |\mathcal{X}| + K \cdot |R|).
$$

The optimal number of epochs depends on the problem.

Note that the time complexity is very favorable: if $|\mathcal{X}|$ is proportional to the size of the movie descriptions measured in bytes (this holds usually when most of the information is textual, and we apply the vector-space model to represent texts as vectors), then this complexity is equal to reading the movie descriptions and the ratings $K$-times.

## 5.2 Related work

There are many works on CBF and hybrid CF-CBF systems. My methods are built on new methods that emerged in the Netflix Prize competition, namely on MF with ALS or incremental gradient descent, and NSVD1. Some of the CBF approaches were already surveyed in Section 2.4, page 33.

Zhang et al. proposed an interesting approach for CBF which is somewhat similar to mine: in their approach, $Q$ is filled with movie metadata (i.e. $Q = X$), and using the $R = PQ^T$ formula, $P$ is estimated by a least squares solver, however, the covariance matrix is modified with a matrix $\Sigma$, where $\Sigma$ is an estimation of the covariance matrix of the user feature vector of a user who has no ratings.

Their method alternates between recomputing $P$ and $\Sigma$. Both their method and mine aim to intensify (or suppress) in a collaborative way those patterns in the metadata that can be useful (or useless). For example: the term “Season” can have large impact on $\hat{r}_{ui}$ by either giving it high weight in the movie or in the user feature vectors. If it is a good feature (many users rely on that feature), it should get high weight on the movie side (allowing lower norm of $p_u$ in general). On the other hand, if it is a bad feature (few users rely on that feature), it should be given a low weight, allowing more important movie features to have higher weight. They concluded that the method was slow, thus they proposed a diagonal $\Sigma$ that could handle the NP dataset in 4 hours. They reported results only on a small subset of the NP dataset, which makes that incomparable to mine.
Collective matrix factorization  Singh et al. proposed a general approach to solve multiple matrix factorization tasks simultaneously \cite{49}, called collective matrix factorization (CMF). I briefly summarize their idea: suppose that we have two matrices (there may be missing values), one for the user-movie relation, describing how users rate movies \((R)\), the other for the movie-metadata relation \((X)\). We can factorize both matrices separately, and get movie feature vectors from factorizing \(R\), and get different movie feature vectors from factorizing \(X\). The idea is to share the movie feature vectors (no difference), and formulate a new optimization problem by linearly combining the target function of the two factorizations (cf. eq. \textcolor{red}{2.12}), with coefficients \(\alpha_1\) and \(\alpha_2\). The idea can be easily generalized to factorize any number of matrices simultaneously.

For content-based filtering, we are interested only in the movie-metadata and user-metadata relations. One can think of Singh’s optimization problem in the following, more intuitive way: we concatenate \(R \in \mathbb{R}^{N \times M}\) and \(X^T \in \mathbb{R}^{C \times M}\), and have a new \(R' \in \mathbb{R}^{(N+C) \times M}\) matrix of ratings and “pseudo-ratings". This matrix is then to be factorized. Suppose that one column of \(X\) is 1 iff the term “Season” occurs in the movie title, and 0 otherwise. In \(R'\), we have a corresponding pseudo-user, who rated every movie having “Season” in the title 1, and all other movies 0. When a new movie gets into the recommender system, no real ratings are available, but we have pseudo-ratings. As more ratings become available, the less the impact of pseudo-ratings is. Two issues have to be solved: handling \(\alpha_1\) and \(\alpha_2\), and the effective factorization of \(R'\). Since \(R'\) can contain many ratings (e.g. when \(C = 60000\) and \(M = 17770\), it is 10 times larger than the Netflix Prize dataset), incremental gradient descent may be slow. Hu et al. \cite{23} proposed an efficient weighted ALS algorithm to handle implicit feedback datasets, which I already mentioned in Section \textcolor{red}{4.3.2}. In that case the matrix is fully filled (mostly with zeros) and an importance weight is assigned to each element. To factorize \(R'\) with that method, we can set the weight of \((u, i) \notin \mathcal{R}\) examples to 0, the weight of \((u, i) \in \mathcal{R}\) to \(\alpha_1\), and the rest of the weights (which corresponds to the metadata part) to \(\alpha_2\).

5.3 Experiments

5.3.1 The Netflix Prize dataset

I evaluated my algorithm on the Netflix Prize dataset.

I collected movie metadata from \url{www.netflix.com} for all but 106 movies out of the 17770. I considered all metadata as text, and used the vector space model common in text-mining to represent texts as vectors. I distinguished between word occurrences in different zones by zone-prefixing. That is the word “comedy” occurring in the genre and in the title zones are represented in two different dimensions “genre:comedy” and “title:comedy” of the metadata vector space. I used the following zones: title, synopsis, actor, director, release year, genre, etc. I did not split the name of the actors and directors into first and last name.

Altogether, movies are described with \(C = 146,810\) dimensional vectors. These vectors have 81 nonzero values on average. Movie vectors were normalized.

Before experimentation, I subtracted the global mean, 3.6043, from all ratings. The prediction of the algorithms were linearly transformed to have the same mean and standard deviation as the Probe10 set.
All experiments were performed on an Intel Core2 Quad Q9300 CPU on 3.3GHz, using only 1 core.

5.3.2 Results

**Basic predictors** First, I experimented using no metadata, using the following method:
- IGD01: BRISMF, using the following parameters: $\eta = \lambda = 0.005$, $K = 20$. Best Probe10 RMSE in the 15th epoch: 0.9079. Running time: 304 sec (on Core 2 Quad 3.3GHz).

**NSVD1 with movie metadata** I now evaluate the movie-NSVD1 proposed in Section 5.1. I experimented with various subsets of movie metadata, which are denoted by:
- T1: keeping all features. $C = 146,810$; 81 nonzero features on average.
- T2: keeping only those features that occur in at least $T$ movies, where $T$ is set to 2. In this case, $C = 64,746$; 76 nonzero features on average.
- T4: like T2, but $T = 4$. $C = 33,838$; 72 nonzero features on average.
- T10: like T2, but $T = 10$. $C = 14,547$; 66 nonzero features on average.
- T20: like T2, but $T = 20$. $C = 7,340$; 60 nonzero features on average.

I used the same learning parameters as in IGD01, but now all movie features are computed by a linear transformation from their descriptions. I experimented with the following variant:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T4</th>
<th>T10</th>
<th>T20</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGD-I</td>
<td>0.9143</td>
<td>0.9155</td>
<td>0.9161</td>
<td>0.9200</td>
</tr>
</tbody>
</table>

Clearly, more metadata imply better prediction performance. However, this is obvious: consider a simple example, where each movie has only one unique metadata, that is, $C = M$, and $X$ is the identity matrix. This case falls back to the matrix factorization, since the movie features have no inter-dependency. As we add infrequent metadata (i.e. words that occur only in one or few movie descriptions), the movie features have more freedom (less inter-dependency between movies), as movie descriptions tend to be dissimilar, thus the results should get closer to the results of pure matrix factorization, i.e. to the RMSE of the IGD01 method (0.9079).

**Predicting ratings on new movies** CBF has a great advantage over CF: the ability to handle new movies. When a new movie is added to a recommendation system, it has no ratings, but metadata are usually available. In this case, CBF algorithms may help. I created the following setup to measure the capability of some CBF algorithms at handling new movies: first, I created 10 partitions for movies, and then applied 10-fold cross-validation: I trained a model using only the appropriate 9/10 of the original training data (i.e. all ratings except Probe10), and then evaluated the model on the appropriate 1/10 of the original test data (i.e. Probe10). Thus, one evaluation means
training on $\sim 0.9 \cdot (100\,480\,507 - 140\,840)$ ratings, and evaluating on $\sim 0.1 \cdot 140\,840$ ratings. Final Probe10 RMSE was calculated by adding the sum of squared errors of each of the 10 evaluations. I refer to this final Probe10 RMSE as $X_{10}$ RMSE.

If the descriptions of movies are as valuable as their ratings, then $X_{10}$ RMSE should be equal to the Probe10 RMSE of the previous experiments where movies has many ratings. On the other hand, if it is unhelpful in predicting user preferences, then $X_{10}$ RMSE should be not much better than that of a user-bias-based method (where the prediction is based on the average rating of the active user).

First, I define some basic predictors:

- user-average predictor: $\hat{r}_{ui} = b_u$, where 
  $$b_u = \frac{\sum_{i: (u,i) \in R} r_{ui}}{|\{i: (u,i) \in R\}| + \alpha_1}.$$ 
- bias predictor: $\hat{r}_{ui} = b_u + c_i$, where $b_u$ and $c_i$ are estimated as the global effects in the description. However, I run the estimation process more than once, thus, both $b_u$ and $c_i$ will be re-estimated. For new items, $c_i = 0$. Let $\alpha_1$ and $\alpha_2$ denote the $\alpha$ used to reestimate $b_u$ and $c_i$, resp.
- user-bias predictor with incremental gradient method: $\hat{r}_{ui} = b_u$ again, but I use incremental gradient method to give larger weights to newer examples.
- input-bias predictor: $\hat{r}_{ui} = b_u + w^T x_i$, where $b_u$ and $w$ are estimated using ALS. $b_u$ and $w$ are regularized with $\alpha_1$ and $\alpha_3$, resp. One can think at the input-bias predictor as the movie bias $c_i$ is estimated by a linear function of the descriptions of the movies.

I did many experiments to optimize $\alpha_1$, $\alpha_2$, and $\alpha_3$.

For user-average predictor, $X_{10}$ RMSE = 1.0616 ($\alpha_1 = 2$).

Interestingly, for the bias predictor, $\alpha_1 = 2$ and $\alpha_2 = 5 \cdot 10^5$ yielded the best results: $X_{10}$ RMSE = 1.0615. Larger values (e.g. $\alpha_2 = 2 \cdot 10^8$) increased RMSE only by $1/3$ of 0.0001, while smaller values (e.g. $\alpha_2 = 100$) increased it by 0.0038, even smaller $\alpha_2$ values were even worse. I interpret this as modeling movie biases does not decrease the $X_{10}$ RMSE.

For the user-bias predictor with incremental gradient method, $\eta = 0.05$ and $\lambda = 0.005$ gave $X_{10}$ RMSE = 1.0504.

For the input-bias predictor, I tried all the T1, ..., T20 datasets. T1 performed the best at $\alpha_1 = 5$ and $\alpha_3 = 5000$ and yielded $X_{10}$ RMSE = 1.0305. Changing the weighting scheme of the metadata vector (e.g. to tf-idf) did not improve the results.

Next, I examined the prediction performance of the proposed movie-NSVD1 for CBF without movie biases using T4. I fine-tuned the learning parameters for NSVD1 with $K = 30$ using the parameter optimizer of [60], optimizing only on the first tenth of the 10 folds. I stopped the parameter tuning process after 127 runs. The model with the best $X_{10}$ RMSE (1.0080) was built in 10 epochs, requiring 366s in total; the 10-fold cross validation thus required about 1 hour. I submitted this predictor to Netflix in order to test the possible overlearning of the parameter optimization. As Quiz RMSE was 1.0078, we could exclude this side-effect.

Then I tried the T1, ..., T20 datasets. T2 performed best, using tf-idf term weighting. Increasing $K$ yielded better results, with $K = 500$, I obtained $X_{10}$ RMSE = 0.9990. The running time was 2570 seconds.

Clearly, NSVD1 has better RMSE than the other methods. However, the Probe10 RMSE (not the cross-validated) of a simple bias predictor is far better than 1.0080: it is usually between 0.9700 and 0.9900. This means, that when we know the movie average,
5.3. EXPERIMENTS

c, it is a much more valuable piece of information than any metadata (the title, genre, etc. of movies). Next I investigate how many ratings a movie should have to obtain a reliable enough estimate of c.

**NSVD1: ratings are more valuable than metadata** I compared the following two methods: NSVD1 with the cross-validation procedure mentioned above (1/10 of the movies are skipped), and the bias predictor (using both user-bias and movie-bias) with incremental gradient method, where the first N ratings of the skipped movies were included in the training set. I varied N, and checked when the RMSE of such a simple predictor will pass over the RMSE of the movie-NSVD1. Results are summarized in Table 5.2. When all ratings are used, Probe10 RMSE is 0.9721.

One can observe in Table 5.2 that even N = 10 ratings are more valuable than the metadata. Recall that this comparison was performed between the very simple bias predictor evaluated on not-so-new movies and the optimized movie-NSVD1 evaluated on new movies. Most probably, having a more sophisticated rating-based method would yield an even larger gap between the results of the two approaches. On the other hand, much better methods may exist for predicting from metadata.

<table>
<thead>
<tr>
<th>N</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.0564</td>
</tr>
<tr>
<td>1</td>
<td>1.0447</td>
</tr>
<tr>
<td>2</td>
<td>1.0325</td>
</tr>
<tr>
<td>5</td>
<td>1.0131</td>
</tr>
<tr>
<td>10</td>
<td>0.9978</td>
</tr>
<tr>
<td>20</td>
<td>0.9880</td>
</tr>
<tr>
<td>30</td>
<td>0.9841</td>
</tr>
<tr>
<td>40</td>
<td>0.9819</td>
</tr>
<tr>
<td>50</td>
<td>0.9802</td>
</tr>
<tr>
<td>100</td>
<td>0.9772</td>
</tr>
<tr>
<td>200</td>
<td>0.9754</td>
</tr>
<tr>
<td>500</td>
<td>0.9740</td>
</tr>
</tbody>
</table>

**CMF with movie metadata, actor similarity** I implemented the CMF method of [49]. I was unable to get X10 RMSE results below 1.0800, however the algorithm was fast (400 seconds for training with K = 10).

Then I focused on the models. The factorization of the matrix of pseudo-ratings gives interesting results, compared to the W of movie-NSVD1. Each metadata element is associated with a K-dimensional feature vector in both approaches (wl in the movie-NSVD1 approach). Let W′ ∈ R^C×K denote the feature vectors of CMF for movie metadata.

In order to compare the two approaches, I investigated how effective are the models in finding similar actors. Given an actor, we can specify a ranked list of similar actors via the feature vectors of movie metadata using an appropriate similarity measure (I used S2 of [60]). When querying for Bruce Willis I obtained as top 3 similar actors:

- NSVD1: Mel Gibson, Julia Roberts, Tom Hanks
- CMF: Robert De Niro, Nicolas Cage, Keanu Reeves

The top 10 list of NSVD1 contains also one director and two synopsis words, while CMF returns only actors. For Jennifer Lopez I obtained:

- NSVD1: Ben Affleck, Sandra Bullock, Julia Roberts
- CMF: Sandra Bullock, Drew Barrymore, Julia Roberts

Although the results are similar, NSVD1 returned one male actor beside the two females. I examined a couple of other actors, and found that CMF yielded only actors for each
queried actor and interestingly, it did not mix male and female actors.

I also examined the movie release year similarity. Each release year was represented with a different binary feature. Both methods tended to give another release years as most similar metadata, however, CMF did slightly better. I observed similar results for CMF, when \( W' \) was computed after a regular MF-run.

Although it is hard to conclude in an unquestionable way which method is better in grouping similar metadata elements, in my experiments CMF performed consistently better in this concern.

To sum up, the \( K \)-dimensional feature vectors of CMF yields a more intuitive similarity than that of the movie-NSVD1. I explain this remarkable difference as follows. Assuming that only \( X \) or \( W' \) are to be optimized, only the following equations should be taken into consideration:

- **NSVD1**: \( Q = XW \)
- **CMF**: \( X = QW'^T \)

When we substitute one equation into the other, we get \( Q = QW'^T W \), or \( X = XWW'^T \). In other words: \( W'^T \) and \( W \) are a kind of pseudoinverses of each other.

Another interpretation: \( W \) is optimized to get movie feature vectors from movie descriptions, while \( W'^T \) is optimized to get movie descriptions from movie feature vectors, i.e. the direction of the mapping is reverse.

**Thesis 3.1** Based on Paterek's NSVD1, I suggested a method for content-based filtering with movie-metadata. I suggested evaluating content-based filtering methods on new movies. I pointed out that the proposed approach outperforms baseline methods: it can reach X10 RMSE = 0.9990, while with baseline approaches, I achieved only X10 RMSE = 1.0305.

**Thesis 3.2** I pointed out that when we give 10 ratings on new movies (thus they are not new anymore), the performance of even a simple bias predictor is better, than the above proposed method's performance on new movies (0.9990). From this, we can conjecture that from the viewpoint of recommender systems, even the average of 10 ratings of a new movie is more valuable (more predictive) than the textural description of a movie, containing the genre, actors, title and plot summary of the movie.

The importance of Thesis 3.2 is the following: many papers are published in the field of CF/CBF claiming that metadata can be useful in recommender systems. Many top contenders of the Netflix Prize competition claim that it is useless. In my opinion, metadata can improve only a badly parameterized method. This experiment tried to relate the usefulness of the two data sources, and concluded that movie metadata is practically useless (but only movie metadata).

## 5.4 Summary

In this Chapter, I presented an approach for content-based filtering (CBF) and particularly for CBF with movie-metadata. The approach is collaborative in the sense that metadata patterns can be suppressed or enhanced based on their predictive power.

- In Section 5.1, I proposed a method which is a simple modification of Paterek’s NSVD1 [32], termed user-NSVD1 here, where users are represented with sparse
binary vectors indicating which movies are rated by the user or not. In my proposed movie-NSVD1, the role of users and movies are interchanged, and movies are represented with the sparse vector representation of their metadata.

- I investigated how effective CBF methods are in predicting ratings on new movies. I showed that even 10 ratings of a new movie are more valuable than the best solely metadata-based representation. I think that this is due to the large gap between the movie descriptions and the movies themselves: people rate movies, not their descriptions. I reckon that CBF methods can be applied more successfully to news recommendation, where the texts themselves are to be recommended and the amount of new items is higher.

- I also investigated the capability of the algorithm to find similar movie actors. I concluded, that in this concern collective matrix factorizations, proposed by Singh et al. gives better results than movie-NSVD1. I discussed a possible reason for this.
CHAPTER 5. CONTENT-BASED FILTERING
Chapter 6
Explaining recommendations

In this chapter I propose a method for explaining recommendations of BRISMF. I described this method in [35].

6.1 Explaining recommendations of ALS

Users may wonder why certain items are recommended to them, as recommended items may seem unpreferred at the first sight. Therefore, a good recommender system should be able to explain its recommendations and gives hints on why a given product is likely to match the user's taste.

Hu et al. proposed a method to provide explanation for factorization based methods in [23] by explaining predictions. Although the method is for the implicit feedback case, it can be easily adapted to the explicit feedback as well:

Recall that in eqs. (2.25)–(2.26), ALS recalculates \( p'_u \) in the following way:

\[
\hat{r}_{ui} = q_i^T W_u d_u = \sum_{j: (u,j) \in R} q_i^T W_u q_j r_{uj},
\]

that is, for each movie that is rated (or watched) by the user, it is known how much its contribution is to the sum. In other words, the prediction \( \hat{r}_{ui} \) is decomposed into a sum.

\[
\text{where where} \quad p'_u = (\lambda n_u I + A_u)^{-1} d_u = (\lambda n_u I + Q[u]^T Q[u])^{-1} (Q[u]^T r_u)
\]

are the covariance matrix and vector used for ridge regression.

Let us introduce \( W_u = (\lambda n_u I + A_u)^{-1} \); here it is rewritten for explicit feedback. Then we have

\[
A_u = Q[u]^T Q[u] = \sum_{i: (u,i) \in R} q_i q_i^T,
\]

\[
d_u = Q[u]^T r_u = \sum_{i: (u,i) \in R} r_{ui} \cdot q_i.
\]

are the covariance matrix and vector used for ridge regression.
which is taken over all ratings made by \( u \). Authors explain \( \hat{r}_{ui} \) by those movies that have the highest contribution to the sum. Later, the word “contribution” will be used with different meanings: \( |q_i^T W_u q_j| \) and \( |q_i^T W_u r_{uj}| \). Note the \( W_u q_j \) vectors can be precomputed in \( O(K^2 \cdot n_u) \) (assuming \( W_u \) is available) to speed up the explanation of an arbitrary movie. Also note that when a user changes one of its ratings (i.e. some \( r_{uj} \)), neither \( W_u \) nor \( W_u q_j \) changes, which makes it efficient for such situations.

If \( \lambda > 0 \), then \( W_u \) is a symmetric positive definite matrix, thus it can be decomposed into \( W_u = V_u^T V_u \) where \( V_u \in \mathbb{R}^{K \times K} \). With this notation \( \hat{r}_{ui} \) can be rewritten as

\[
\hat{r}_{ui} = \sum_{j: (u,j) \in \mathcal{R}} (V_u q_i)^T (V_u q_j) r_{uj}.
\]

With the above notation, \( V_u \) describes how user \( u \) “thinks” about the items, and the user-independent similarity \( q_i^T q_j \) between two items is replaced with the user-dependent similarity \( (V_u q_i)^T (V_u q_j) \).

### 6.2 Explaining recommendations using dual formulation of RR

To calculate \( p_u \), we can use the equivalent dual formulation (kernel ridge regression with linear kernel) involving the Gram matrix \( Q[u]Q[u]^T \):

\[
p_u' = Q[u]^T (\lambda n_u I + Q[u]Q[u]^T)^{-1} r_u.
\]  

(6.3)

Let \( A_u^{\text{dual}} = Q[u]Q[u]^T \in \mathbb{R}^{n_u \times n_u} \). Let \( \alpha_u = (\lambda n_u I + A_u^{\text{dual}})^{-1} r_u \in \mathbb{R}^{n_u \times 1} \); then the calculation of \( p_u \) is rewritten as: \( p_u = Q[u]^T \alpha_u \). Let \( \alpha_{ui} \) denote that element of \( \alpha_u \) which refers to the \( i \)-th movie rated by \( u \). With this notation: \( p_u = \sum_{i: (u,i) \in \mathcal{R}} \alpha_{ui} \cdot q_i \), and

\[
\hat{r}_{ui} = \sum_{j: (u,j) \in \mathcal{R}} \alpha_{uj} \cdot q_i^T q_j
\]  

(6.4)

Again, we know the contribution of each movie watched by the user in the sum.

### 6.3 Explaining recommendations of BRISMF

**Dual formulation**

BRISMF approach was described in Section 2.3.1. In Section 4.1, I introduced a modification in the learning algorithm termed as “retraining user features”. The modified learning scheme does not change the prediction performance significantly.

After obtaining \( P \) and \( Q \) by BRISMF, the algorithm resets \( P \), and reruns the training algorithm, but \( Q \) is now fixed. Let \( n \) denote the number of epochs in the second run. Now I show that this modification also helps explaining recommendations: Since \( Q \) does not change in second run, we can recompute \( p_u \) by iterating over the ratings of \( u \) for \( n \) times. I denote this variant as BRISMF-U. We can rewrite equation (2.14) as
\[ p_u' = p_u \cdot (1 - \eta \cdot \lambda) + \eta \cdot e_u \cdot q_i. \] Thus \( p_u \) is the linear combination of its initial value \( p_u^0 \) and the \( q_i \) vectors rated by \( u \):

\[ p_u = \alpha_u p_u^0 + \sum_{i: (u,i) \in R} \alpha_{ui} q_i \]

(6.5)

In this way, we can explain recommendations in a similar way as proposed in Section 6.2.

**Deriving primal formulation**

ALS is not sensitive to the order of examples, as opposed to gradient methods, namely BRISMF-U. In [57] authors proposed to order examples by \( u \) and then by the date of the ratings, to get better RMSE scores. In [60] it has been pointed out that a BRISMF with simple manually set parameters can achieve test RMSE = 0.9104 when examples are ordered only by \( u \), and 0.9056 when they are ordered by \( u \) and then by the date of the ratings. In the NP competition, for each user, the examples in the test set are newer than in the training set, which is practical, since in a real recommender system the goal is to predict the present preference based on past ratings. Now I propose a method that can explain predictions of BRISMF-U in a way very similar to the explanation method for ALS with primal formulation (Section 6.1).

One can think at the ordering of the examples by date as giving higher weights (confidence) to newer examples. This leads to the idea that BRISMF-U can be related with weighted ridge regression (WRR), by finding the appropriate weight for each rating. In the followings, I assume user \( u \) in context, and further assume that \( \alpha_u p_u^0 = 0 \). I will show how to find weights for movies rated by \( u \), such that the result of weighted ridge regression will be (almost) equal to the result of BRISMF-U.

Let \( c_{ui} \) denote the weight (confidence) of \( u \) on item \( i \). Let \( C_u \in \mathbb{R}^{n_u \times n_u} \) be the diagonal matrix of the \( c_{ui} \) values. Let \( A_u = \sum_{i: (u,i) \in R} c_{ui} q_i q_i^T \) and \( d_u = \sum_{i: (u,i) \in R} c_{ui} r_{ui} = Q[u]^T C_u r_u \) be the weighted covariance matrix and weighted covariance vector resp.

The cost function of weighted ridge regression is the following:

\[ \lambda n_u p_u^T p_u + \sum_{i: (u,i) \in R} c_{ui} (p_u^T q_i - r_{ui})^2 \]

(6.6)

The \( p_u \) minimizing this cost function can be calculated by letting its derivative be equal to zero:

\[ 2(\lambda n_u I + A_u) - 2d_u = 0 \]

(6.7)

If \( \lambda n_u I + A_u \) is positive definite, then the unique solution of this equation is the global minimum: \( p_u = (\lambda n_u I + A_u)^{-1} d_u \). This is the case when all \( c_{ui} \geq 0 \) and \( \lambda > 0 \). When \( \lambda n_u I + A_u \) is not positive definite, the solution is not unique, or no solutions exist at all (the function is not bounded from below). In the followings, I assume \( \lambda > 0 \), but \( c_{ui} \) may be arbitrary.

We can compute \( p_u \) using the equivalent dual formulation as well:

\[ p_u = Q[u]^T \alpha_u, \] where \( \alpha_u = C_u (\lambda n_u I + Q[u]^T C_u)^{-1} r_u \).

(6.8)
Note that when $\lambda \neq 0$, then:

$$\lambda_{ui} \mathbf{I} + \mathbf{Q}[u]^T \mathbf{C}_u \mathbf{Q}[u]$$

is invertible \iff

$$\lambda_{ui} \mathbf{I} + \mathbf{Q}[u]\mathbf{Q}[u]^T \mathbf{C}_u$$

is invertible,

since the eigenvalues of $(\mathbf{Q}[u]^T \mathbf{C}_u)\mathbf{Q}[u]$ and $\mathbf{Q}[u](\mathbf{Q}[u]^T \mathbf{C}_u)$ are the same, apart from the eigenvalue 0. The addition of the term $\lambda_{ui} \mathbf{I}$ shifts these eigenvalues by $\lambda_{ui}$. Thus, after the addition either both the new matrices have the eigenvalue 0, or none, which means that either both are invertible, or none. Furthermore, if $\lambda > 0$, then either both are positive definite, or none.

Note that after running BRISMF-U, we are given with the $\alpha_u$ vector. A possible way to relate it with WRR is to find appropriate $c_{ui}$ values, such that $\alpha_u$ values computed from \ref{eq:6.8} equals to $\alpha_u$ from BRISMF-U. Consider the above equation for $\alpha_u$. Now I solve it for $\mathbf{C}_u$. First, multiplying it by $\mathbf{C}_u^{-1}$ from left (I will discuss later if $c_{ui} = 0$ for some $i$), it yields:

$$\mathbf{C}_u^{-1} \alpha_u = (\lambda_{ui} \mathbf{I} + \mathbf{Q}[u]\mathbf{Q}[u]^T \mathbf{C}_u)^{-1} \mathbf{r}_u$$

Now multiply with $(\cdot)^{-1}$ from left:

$$(\lambda_{ui} \mathbf{I} + \mathbf{Q}[u]\mathbf{Q}[u]^T \mathbf{C}_u)\mathbf{C}_u^{-1} \alpha_u = \mathbf{r}_u$$

Then it reduces to:

$$\lambda_{ui} \mathbf{C}_u^{-1} \alpha_u + \mathbf{Q}[u]\mathbf{Q}[u]^T \alpha_u = \mathbf{r}_u$$

Let $\hat{\mathbf{r}}_u = \mathbf{Q}[u]\mathbf{Q}[u]^T \alpha_u$ denote the predictions of BRISMF-U. Then it is rewritten as:

$$\lambda_{ui} \mathbf{C}_u^{-1} \alpha_u + \hat{\mathbf{r}}_u = \mathbf{r}_u$$

After reordering and multiplying with $\mathbf{C}_u$ it yields:

$$\lambda_{ui} \alpha_u = \mathbf{C}_u(\mathbf{r}_u - \hat{\mathbf{r}}_u)$$

thus:

$$c_{ui}(r_{ui} - \hat{r}_{ui}) = \lambda_{ui} \alpha_{ui}. \quad (6.9)$$

Now let us discuss some special cases of this solution:

- if for some $i$, $\alpha_{ui} = 0$, then we can set $c_{ui}$ to 0 (substituting it directly into \ref{eq:6.8} we can see that it is a good solution), and ignore in the above equations those rows and columns of $\mathbf{C}_u$, $\mathbf{Q}[u]$ and $\mathbf{r}_u$ that corresponds to these $c_{ui}$-s. Note that $c_{ui} = 0$ can occur only if $\alpha_{ui} = 0$ (recall that I assume $\lambda > 0$).

- Since $c_{ui}$ values computed by \ref{eq:6.8} can be negative, $\lambda_{ui} \mathbf{I} + \mathbf{A}_u$ might not be positive definite. When $\lambda_{ui} \mathbf{I} + \mathbf{A}_u$ is not positive definite, then we cannot relate WRR and BRISMF-U in this way: for any $c_{ui}$ values, which makes $\mathbf{A}_u$ positive definite, the $\alpha_u$ computed from \ref{eq:6.8} will be different from $\alpha_u$ of BRISMF-U.

- If for some $i$, $r_{ui} - \hat{r}_{ui} = 0$, i.e. that examples are learnt perfectly by BRISMF-U, but $\alpha_{ui} \neq 0$, then no $c_{ui}$ values can satisfy the equation.

If $\lambda_{ui} \mathbf{I} + \mathbf{A}_u$ is positive definite and $\forall i : r_{ui} - \hat{r}_{ui} = 0 \implies \alpha_{ui} = 0$, then WRR and BRISMF-U can be related, and then we can explain predictions as in \ref{eq:6.2} by computing

$$\mathbf{W}_u = (\lambda_{ui} \mathbf{I} + \mathbf{Q}[u]^T \mathbf{C}_u \mathbf{Q}[u])^{-1}.$$

Even if $\lambda_{ui} \mathbf{I} + \mathbf{A}_u$ is not positive definite, but it is invertible, and all $c_{ui}$-s are defined by \ref{eq:6.9}, we can still compute $\mathbf{W}_u$ by neglecting the fact that there does not exist an
optimal solution for \( (6.6) \) with these \( c_{ui} \)-s. When not all \( c_{ui} \)-s are defined by \( (6.9) \), we may set \( c_{ui} = 0 \) for the undefined ones, however, in this case the corresponding \( \alpha_{ui} \)-s will not be equal to those of BRISMF-U. I found by experimentation that setting negative and undefined \( c_{ui} \)-s to 0 does not affect prediction performance significantly: the largest difference of test RMSE of BRISMF-U and this modified WRR was 0.0002 on the Netflix Prize dataset.

Note that when we are given with \( \hat{r}_u \) and \( c_{ui} \), then \( \alpha_{ui} \) can be easily computed from \( (6.3) \), thus, there is no need to invert the regularized Gram matrix to get \( \alpha_{ui} \)-s. In case of the original formulation of ALS, all \( c_{ui} = 1 \), and \( \hat{r}_u \) is also known.

### 6.3.1 Comparing primal and dual formulation

Note that eqs. \((6.2)\) and \((6.4)\) explains predictions in two different ways. The primal formulation based explanation computes \( W_u \) that can compute a user-dependent similarity of any two items (as it has been noted by Hu et al.). The dual formulation computes \( \alpha_{uj} \)-values for each item \( j \) rated by \( u \), which tells how the user-independent similarity \( q^T_i q_j \) should be replaced with \( q^T_i \cdot (\alpha_{uj} q_j) \).

One can think of the primal formulation based explanation that a prediction is a weighed sum of the ratings made by the user, while the dual formulation explains predictions as a weighed sum of training errors. Let us introduce the similarities \( s^p_{uij} \) and \( s^d_{uij} \), that tells how similar are \( i \) and \( j \), according to \( u \), based on the primal and the dual formulation. Let \( s^p_{uij} = q^T_i W_u q_j \). Then the primal formulation based explanation is rewritten as:

\[
\hat{r}_{ui} = \sum_{j: (u,j) \in R} s^p_{uij} r_{uj}. \tag{6.10}
\]

Let \( s^d_{uij} = q^T_i q_j c_{ui} / (\lambda n_u) \). Then, based on \((6.4)\), the dual formulation based explanation is rewritten as:

\[
\hat{r}_{ui} = \sum_{j: (u,j) \in R} s^d_{uij} (r_{uj} - \hat{r}_{uj}), \tag{6.11}
\]

which is very unintuitive, since the prediction \( \hat{r}_{ui} \) is a linear combination of prediction errors.

### 6.4 Experiments

I perform experiments on the Netflix Prize dataset. For the experiments, I use BRISMF#250U mentioned in Section 4.1, but with \( K = 20 \), as a matrix factorization model, which I refer to as BRISMF#20U. Probe10 RMSE of this model is 0.9051. By clipping the predicted value in the range of 1 to 5, it goes down to 0.9045.

### 6.4.1 Examining primal formulation for gradient methods

The \( c_{ui} \)-s computed from eq. \((6.4)\) can be negative, or can be positive but too large, or can be even undefined (when \( r_{ui} - \hat{r}_{ui} = 0 \) and \( \alpha_{ui} \neq 0 \)). I propose a simple heuristic: set \( c_{ui} \) to 0 when \( c_{ui} < 0 \) or undefined. Intuitively a good matrix factorization algorithm with
date-based ordering of examples should not differ too much from ridge regression, which treats all examples equally (\(c_{ui} = 1\) for all \(i\)). Thus, I also propose to let \(c_{ui} := 100\) when \(c_{ui} > 100\). These heuristics increased Probe10 RMSE only to 0.9052. When I decreased the upper limit from 100 to 10, the Probe10 RMSE was again 0.9052. Thus, we are able to explain a 0.9051 prediction without involving too large numbers or indefinite optimization.

In practice, users are recommended with items having the highest predicted ratings. Matrix factorization is able to predict ratings for all of the unseen movies of \(u\): fast: the computational cost is only \(O((M - n_u) \cdot K)\). Since the above heuristics does not change prediction performance significantly (but allows to explain predictions of BRISMF-U), we can first compute the top ranked movies, and then deal with the explanation.

I did not observed \(r_{ui} - \hat{r}_{ui} = 0\) during testing. However, 2.3% of the \(c_{ui}\) values were below 0. The \(c_{ui}\) values ranged from \(-6 \cdot 10^6\) to \(5 \cdot 10^6\). The optimization problem in eq. (6.6) was not positive definite in 36% of the cases. Note that \(c_{ui}\) is computed for each training example, but eq. (6.6) is solved for each test example.

When all \(c_{ui}\)-s are nonnegative, then the eigenvalues of \(\lambda_nuIAu\) are at least \(\lambda_nu\). I observed, that without the \(c_{ui} \geq 0\) restriction that matrix had at least one eigenvalue being less than \(\lambda_nu\) in the 76.4% of the cases.

### Comparing different explanation algorithms

I evaluate different explanation algorithms by measuring how many times the most explanatory movies are the same. All methods are based on the result of BRISMF#20U, however, ALS-based methods use only the movie feature matrix \(Q\) of that model. I use the following abbreviations to refer to the methods:

- **AP**: primal formulation for ALS, according to eq. (6.2).
- **AD**: dual formulation for ALS, according to eq. (6.4).
- **GD**: dual formulation for gradient methods, like AD, but the \(\alpha_{ui}\)-s are provided by the gradient method.
- **GP**: primal formulation for gradient methods, like AP, but confidence-weighted.

Confidences are computed according to eq. (6.9).

I distinguish between two variants:

- **R**: using the \(s_{uij}^p \cdot r_{uj}\) and \(s_{uij}^d \cdot (r_{uj} - \hat{r}_{uj})\) with the highest absolute values, see eqs. (6.10)–(6.11).
- **N**: using only the \(s_{uij}^p\) and \(s_{uij}^d\) with the highest absolute values.

Results are summarized in Table. 6.1. The intent of this table is to give information about how the different methods are related to each other. I will examine their ability to provide reasonable explanations later.

Interestingly, GPR and APR are the most similar methods, however, Probe10 RMSE of ALS was only 0.9320. Although ALS could perform much better, I did not fine-tune the learning parameters, I used the same as BRISMF#20U. Both APR and APN were able to explain similarly to GPR, despite of the huge difference in prediction performance.

GDN is an outlier, as it has the least common most-explanatory movies with other methods. This may be due to the large numbers involved in the computation of \(s_{uij}^d\) caused by small training errors.

From the table, we can conclude that primal-based methods are similar to each other (similarity is always > 40%), while the dual based methods are quite different: similarity
6.4. EXPERIMENTS

Table 6.1: Comparing different explanation methods for the same model. Numbers indicate the percentage of how often two methods ranked the same movie as the most explanatory for a prediction, on the test set.

<table>
<thead>
<tr>
<th></th>
<th>APR</th>
<th>GPR</th>
<th>APN</th>
<th>GPN</th>
<th>GDR</th>
<th>GDN</th>
</tr>
</thead>
<tbody>
<tr>
<td>APR</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPR</td>
<td>71</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>APN</td>
<td>56</td>
<td>46</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPN</td>
<td>37</td>
<td>42</td>
<td>48</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GDR</td>
<td>14</td>
<td>17</td>
<td>16</td>
<td>20</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>GDN</td>
<td>9</td>
<td>8</td>
<td>10</td>
<td>14</td>
<td>6</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 6.2: This table indicates how many times a prediction for a movie from a set of similar movies is mostly explained by a rating on a movie from the same set, e.g. a prediction of a Matrix movie is top-explained by a rating on a Matrix movie. $n$ is the total number of such movies in the test set (Probe10).

<table>
<thead>
<tr>
<th>Movie collection</th>
<th>n</th>
<th>APR</th>
<th>GPR</th>
<th>APN</th>
<th>GPN</th>
<th>GDR</th>
<th>GDN</th>
<th>APR63</th>
<th>GPR63</th>
<th>GDR63</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Matrix</td>
<td>254</td>
<td>137</td>
<td>131</td>
<td>136</td>
<td>135</td>
<td>38</td>
<td>28</td>
<td>154</td>
<td>154</td>
<td>79</td>
</tr>
<tr>
<td>Friends</td>
<td>443</td>
<td>370</td>
<td>359</td>
<td>376</td>
<td>376</td>
<td>218</td>
<td>243</td>
<td>372</td>
<td>367</td>
<td>250</td>
</tr>
<tr>
<td>Die Hard</td>
<td>131</td>
<td>46</td>
<td>44</td>
<td>42</td>
<td>17</td>
<td>13</td>
<td>12</td>
<td>55</td>
<td>48</td>
<td>25</td>
</tr>
</tbody>
</table>

between primal and dual based methods are always $< 20\%$.

I also examined manually the explanations of the proposed methods. First, I looked how the movies of the Matrix trilogy (The Matrix, The Matrix: Reloaded and The Matrix: Revolutions) are explained. I also examined a variant of BRISMF#20U, where $K$ is 63. In this case, Probe10 RMSE is 0.8988 for the gradient method, and 0.9253 for the ALS method. I refer to the $K = 63$ variant by appending “63” to the method name. There were 254 cases out of the 140840, when one of the above 3 Matrix movies was to be predicted. Results are summarized in Table 6.2.

Both GP and AP methods had similar success in explaining a Matrix movie by a Matrix movie. The numbers for the dual methods are poor, and reflect that they are not good in explaining predictions.

Second, I examined a popular TV-series, the Friends, which aired for 10 seasons. I examined how often a Friends movie is explained by a Friends movie. In the Netflix Prize database there are 9 DVDs for the 9 out of 10 seasons, and there are also “best of” DVDs. Together, it is 17 DVDs with 443 ratings in the Probe10 set. Again, I found that the primal formulation based methods list these movies amongst the top explanatory movies much frequently, than the dual based methods. Third, I examined the Die Hard trilogy, with similar conclusions.

A main drawback of the above comparison is that a simple neighbor method will reach the highest numbers, explaining Matrix movies always by Matrix movies in the first place. However, such an explanation method is unpersonalized, i.e. the $s_{uij}$ similarities do not depend on $u$.

It may occur that a user rates The Matrix: Revolutions as a 3, and The Matrix: Reloaded as a 5. When the recommendation system is recommending her The Matrix as a 5, GPN and APN does not take into account that one is higher-rated than the
It may seem strange, that the system is recommending *The Matrix*, because the user rated *The Matrix: Revolutions* as a 3. GPR and APR take this information into account. However, when we change the rating system from “5 is the best” to “1 is the best”, this advantage turns into disadvantage, since the system is going to recommend low-predicted (thus good) movies by high-rated (thus bad) movies.

**Thesis 4** Based on the results of Bell and Koren [6], which provides a method for explaining recommendations of ALS-based MFs, I proposed a method which is able to explain recommendations of BRISMF. I performed some experiments that demonstrate the viability of the algorithm. The results are similar to those obtained by Hu et al’s method. I showed how the gradient descent algorithm can be related to the weighted ridge regression: if we are given with the dual model of a linear model, we can compute an importance weight for each example, such that running weighted ridge regression with those importance weights, the result will be (almost) equal to that model. In this way, Hu et al’s method – which can explain the predictions of weighted ridge regression – can be applied to explain the predictions of BRISMF algorithms.

### 6.5 Summary

Hu et al. [23] showed how the predictions of weighted ridge regression can be explained, where the explanation lists the most influential training examples regarding the predicted example.

In this work I introduced how to explain predictions of gradient descent based learning methods. Although the presented methods are applicable to any kind of gradient descent based learning (e.g. text document categorization), I put the work in the context of the Netflix Prize dataset and explanation of movie recommendations.

I showed how to relate gradient descent methods to weighted ridge regression (WRR), by assigning appropriate importance weights to the training examples. Given a model generated by gradient descent, we can almost always find importance weights such that a weighed ridge regression with these weights will yield exactly the same linear model. However, in some situations it is impossible to relate them, or the weights are too large. For these cases, I proposed a simple heuristics: set negative weights to zero, and clamp the too large values. I showed by experimentation that in this case the prediction of the weighted ridge regression only slightly differs from the prediction of the gradient method.

I also proposed two other explanation methods based on the dual formulation of gradient methods and WRR, but these methods did not proved to be efficient in providing reasonable explanations.
Summary

In this dissertation I presented my proposed methods for collaborative filtering problems. Achievements are organized into 4 thesis groups:

Thesis 1: I proposed many methods for improving RMSE of matrix factorizations

Thesis 1.1
In Section 3.1 I suggested using 8 parameters in BRISMF ($\eta^p$, $\eta^q$, $\lambda^p$, $\lambda^q$, $\eta^{pb}$, $\eta^{qb}$, $\lambda^{pb}$, $\lambda^{qb}$) instead of 4 ($\eta^p$, $\eta^q$, $\lambda^p$, $\lambda^q$) or 2 ($\eta$, $\lambda$). I applied automatic parameter optimization to set the parameters. I suggested user-subsampling and using small $K$ values to speed up the optimization process. With the above considerations I achieved 6.04% improvement over Netflix’s Cinematch algorithm.

I also evaluated the proposed parameter optimization procedure on the MovieLens and Jester datasets. I concluded that the introduction of 4 or 8 parameters is useless for these datasets. I also evaluated whether user-subsampling and the usage of small $K$ values are useful to improve the speed of parameter optimization. I concluded that using small $K$ values is useful. Subsampling the users is useful on the MovieLens dataset, but not on the Jester dataset.

Thesis 1.2
In Section 3.2 I suggested finding automatic BRISMF parameterizations that improve on a combination of previously existing methods. I showed on the Netflix Prize dataset, that the proposed approach works: on BRISMF#250, two models (MLMF#200 and MLMF#80) can improve by $0.0011 + 0.0004 = 0.0015$ RMSE points. This result allows to achieve the same performance as of BRISMF#1000, but within less than half of that time and memory requirement (1000 factors versus 250+200+80 factors).

I also evaluated the method on the Jester and MovieLens datasets. Again, the method can find models which blends well with a given model.

Thesis 1.3
In Section 3.3 I suggested that the regularization and learning rate parameters of BRISMF should depend on the number of ratings of the active user and item. I proposed to replace each of these parameters with a function with 7 parameters. Again, parameters are optimized with an automatic parameter optimization algorithm. This method can achieve 6.53% improvement over Netflix’s Cinematch algorithm. I also tested the method on the Jester dataset with success, and on the MovieLens dataset with moderate success.

Thesis 1.4
In Section 3.4 I suggested a simple modification of the learning rule of BRISMF
algorithm: in the best epoch, reset P and keep the learning process going on until RMSE on a validation set improves. This modification can improve the accuracy of BRISMF-based matrix factorizations. On the Netflix Prize dataset: The RMSE performance of BRISMF#250 and BRISMF#1000 on Quiz can be boosted by 0.0027 and 0.0021 resp. I also evaluated the method on the Jester dataset with success, and on the MovieLens without success. However, I pointed out, that this may be due to the small number of users in that dataset.

**Thesis 1.5**
In Section 3.5, I suggested a hybrid method that efficiently alloys BRISMF and NSVD1. The idea is to sum the output the two methods, and train them simultaneously. This can be seen as training two methods on the residual of each other. I put BRISMF and NSVD1 in the framework of multi-layer perceptrons. I experimented with the Netflix Prize data, where the method (Hybrid#1000-80) achieved 6.61% improvement over Cinematch. I also pointed out that the approach blends well with the method of Thesis 1.3: the blending of this 6.61% and that 6.39% approaches can achieve 7.04% improvement over Cinematch.

**Thesis 2**: I proposed many methods to speed up BRISMF and ALS based matrix factorizations, and also to efficiently handle new ratings of users, or new users.

**Thesis 2.1**
In Section 4.1, I proposed a modification for BRISMF that can efficiently handle new ratings of users. I pointed out by extensive experiments on the Netflix Prize dataset, that the method is efficient in terms of both speed and accuracy. As a result, only a small subset of the database is enough to get a reliable estimate of Q, and then fixing this matrix we can recompute P for each user separately, using all of her ratings. I also evaluated the method on the MovieLens dataset with success, and on the Jester dataset with moderate success.

**Thesis 2.2**
In Section 4.2, I proposed to optimize the parameters of BRISMF such that the best RMSE is achieved in the first few epochs. I pointed out by experiments on the Netflix Prize dataset, that only 200 seconds is enough to get a model in 2 epochs with Probe10 RMSE = 0.9071, on a Pentium M 2GHz (Dothan) CPU. I also tried the idea on the Jester and MovieLens datasets. The results are acceptable. On the MovieLens dataset, I concluded that if we had more users, the results would be more favorable.

**Thesis 2.3**
In Section 4.3, I examined the applicability of the Sherman-Morrison formula for alternating least squares based matrix factorizations. I showed that by using the Sherman-Morrison formula ALS based algorithms can be speed up both on explicit and implicit feedback datasets. For explicit feedback, I improved from $O(K^2 \cdot |R| + K^3 \cdot N)$ to $O(K^2 \cdot |R|)$ the time complexity of the recomputation of the user feature matrix (P). For implicit feedback $O(K^2 \cdot |R^+| + K^3 \cdot N)$ is reduced to $O(K^2 \cdot |R^+|)$. For the item feature matrix Q analog results hold. If a user provides a new rating, recomputing $p_u$ can be speed up from $O(K^3)$ to $O(K^2)$.

I also pointed out that SMF can also be applied to reduce the complexity of greedy feature selection algorithms on F predictors from $O(F^5)$ to $O(F^3)$. 

Thesis 2.4
In Section 4.4, I examined the applicability of linear kernel ridge regression for ALS based matrix factorizations. I showed, how linear kernel ridge regression can speed up the recomputation of $p_u$ in ALS for users with $n_u < K$ ratings: the original $O(K^2 \cdot n_u + K^3)$ is reduced with SMF to $O(K^2 \cdot n_u)$, and with KRR to $O(K \cdot n_u^2)$. I performed experiments to show how much computational performance gain can be brought by KRR.

I pointed out that the addition or deletion of ratings can be handled in $O(K^2)$ by SMF, and if $n_u < K$, then in $O(K \cdot n_u)$ with KRR + SMF, while the traditional way requires $O(K^3)$ operations. These proposed methods allows ALS to be used in practical recommender systems, where the system must respond instantly, when a new user rating arrives.

Thesis 3: Method for content-based filtering on new items.

Thesis 3.1
In Section 5.1, Based on Paterek’s NSVD1, I suggested a method for content-based filtering with movie-metadata. I suggested evaluating content-based filtering methods on new movies. I pointed out that the proposed approach outperforms baseline methods: it can reach X10 RMSE = 0.9990, while with baseline approaches, I achieved only X10 RMSE = 1.0305.

Thesis 3.2
In Section 5.1, I pointed out that when we give 10 ratings on new movies (thus they are not new anymore), the performance of even a simple bias predictor is better, than the above proposed method’s performance on new movies (0.9990). From this, we can conjecture that from the viewpoint of recommender systems, even the average of 10 ratings of a new movie is more valuable (more predictive) than the textual description of a movie, containing the genre, actors, title and plot summary of the movie.

The importance of Thesis 3.2 is the following: many papers are published in the field of CF/CBF claiming that metadata can be useful in recommender systems. Many top contenders of the Netflix Prize competition claim that it is useless. In my opinion, metadata can improve only a badly parameterized method. This experiment tried to relate the usefulness of the two data sources, and concluded that movie metadata is practically useless (but only movie metadata).

Thesis 4: Explaining recommendations:
Based on the results of Bell and Koren, which provides a method for explaining recommendations of ALS-based MFs, I proposed a method in Section which is able to explain recommendations of BRISMF. I performed some experiments that demonstrate the viability of the algorithm. The results are similar to those obtained by Hu et al’s method. I showed how the gradient descent algorithm can be related to the weighted ridge regression: if we are given with the dual model of a linear model, we can compute an importance weight for each example, such that running weighted ridge regression with those importance weights, the result will be (almost) equal to that model. In this way, Hu et al’s method – which can explain the predictions of weighted ridge regression – can be applied to explain the predictions of BRISMF algorithms.
Further research

In this dissertation, I have dealt with many different problems related to collaborative filtering and content-based filtering. Some results are very promising and encourage further research.

In section 3.4 I proposed to reset $P$, to obtain better RMSE. Things to be investigated: why this approach helps, and whether it can be applied for the training of neural networks (note that factorizing a matrix can also be seen as learning a neural network (page 57).

**Hybrid MF+NSVD1 method** Running MF on the residual of NSVD1 can be considered as follows: we train a hybrid MF+NSVD1 method for a given number of epochs, but the learning rates of the NSVD1 part is set to zero. Then we set NSVD1 learning rates to real values, and set the learning rates of MF to zero. If we extend the hybrid method by switching parts of them on-and-off (e.g the MF part or the NSVD1 part), then it is a generalization of training on residuals.

Different methods (like MF and NSVD1) may require different number of training epochs. By switching the training of MF or NSVD1 on-and-off for some epochs, we can alloy such methods that require quite a different number of epochs.

**Content based filtering** In section 5.1 I proposed to apply NSVD1 for content-based filtering. The approach was proven to be much better than many baseline approaches for the prediction of ratings on new movies. However, the results are still not good enough. Here the questions to be investigated are:

Is this approach good for a news recommender system? In my opinion, the approach does not work for movies, because people rate movies, not their descriptions. If an actor has played in many good movies, and also in many really bad movies (thus, he/she is a actually a good/famous actor), then this actor is not more predictive than an actor who has played only in average movies (he/she is an average/unknown actor). Maybe that is why metadata is useless for prediction. In case of a news recommender system, the content itself is what people rate, thus we can expect smaller difference between CF and CBF than in the movie-domain, and it makes more sense in the news-domain to apply CBF for the recommendation of new items.

It should also be investigated when ratings overweight the rating + movie metadata combination in prediction.

Note that the proposed approach for movie-CBF decomposes $R$ into $R = PQ^T = P(W^TX^T) = (PW^T)X^T$. I would like to investigate $PW^T$, since this matrix contains a $C$-dimensional feature vector for each user, which can describe users preferences on actors, genres, etc.
Further research

In [37], I suggested how the proposed approach can be applied to implicit feedback datasets, i.e. where the matrix of ratings is fully filled. The idea is to recompute the weights between the hidden layer and the output with the optimized algorithm which was proposed by Hu et al. [23], and has been further speed up in section 4.3.2. After recomputing the hidden layer, the weights between the input layer and the hidden layer can be optimized with gradient descent. It can also be easily extended to handle both user and item metadata, although similar approach has been proposed in [24]. This approach seems promising, and has to be evaluated extensively:

- it needs to be evaluated for implicit feedback datasets: is it able to capture relationships between metadata and users’ watching/viewing/buying habits?
- it needs to be evaluated for explicit feedback datasets, when the input of the network is implicit feedback data (like Paterek’s NSVD1, but on movies). Is this algorithm effective when new movies receives few ratings? Can this algorithm alloy movie metadata and implicit feedback data?
- Symeonidis et al. [54] suggested to explain recommendations not only by listing the most influential ratings of the user, but also by listing most influential actors, etc. of the movie. According to a user study, the approach seemed effective. Note that we are able to easily compute the effect of each actor in the movie with the movie-NSVD1 approach. An important question is: can it be applied to generate such explanations?
- the algorithm should be tested on the Movielens-1M dataset, where both user and item metadata are provided, to see whether user metadata is useful or not, and to see whether the algorithm is efficient in incorporating both information sources.
- the implicit feedback variant is generally applicable to multi-layer perceptrons with high dimensional sparse output. It should be investigated whether such neural network training problems exists, and whether the algorithm is applicable.

An interesting question regarding both the first and the second approach I proposed for content based filtering is: Can these approaches be used for other problems, where multiple regression / categorization tasks needs to be solved simultaneously? A potential application area is text categorization: documents need to be classified into predefined categories. When we have only one category (e.g spam / not spam), there is no point in applying this method. When we have many related category, which is the case of hierarchical text categorization, then it could be worth to emphasize some words and allow the related categorization problems to have effect on each other. I remark that Zhang and Koren [69] also applied their approach for text document categorization with success.

Another interesting question is: what is the limit of content-based filtering. Although I presented two approaches, it may be far from the best possible performance. For example, in the Netflix Prize contest, all of 7%, 8%, 9% and 10% improvements seemed impossible at a given point of time: when 7% was done, 8% seemed impossible, and so on. It is hard to predict what is the limit for content-based filtering.

Explaining recommendations  In Section 6 I proposed an explanation method for BRISMF-based matrix factorizations. In the Netflix Prize competition, blending multiple predictions is very common to get better RMSE [10]. When the blending is linear, and the prediction of the algorithms can be decomposed as a summation over the movies rated by the user, then the blended prediction can be decomposed as well, enabling the
explanation of the prediction. In the Netflix Prize competition many approach were proposed that can inherently explain predictions (neighbor methods, NSVD1 methods). ALS and gradient descent methods can also be explained. The blending of these approaches may result in a much better prediction. An important question to investigate: is the correct explanation of a better prediction a better explanation?

The explanation method of Hu et al. decomposes a prediction into the following form:

$$\hat{r}_{ui} = \sum_{j: (u,j) \in R} s_{uij} r_{uj}. \quad (6.12)$$

Note that this equation is a weighted nearest neighbor approach, where the values are to be defined. They can be computed via:

$$s_{uij} = (V_u q_i)^T (V_u q_j) \quad (6.13)$$

Where $q_i$ is the feature vector of the active example (active item, for which the rating is to be predicted), and $q_j$ is the feature vector of the $j$-th training example. The calculation of $V_u$ is defined in Section 6.1.

My proposed method of explaining predictions of gradient descent method relies only on the dual formulation of the method, i.e. the weight vector of the model is a linear combination of the training examples. Many linear model has a dual representation like Adaline (gradient descent in the Netflix Prize competition), perceptron, linear support vector machines (SVM), ridge regression, centroid classifier, etc. Thus, my proposed explanation algorithm is very general.

This implies, that, for example, in a text categorization task where the classifier is a linear SVM, we are able to explain the category prediction of an unseen document. A traditional approach to explain is to query documents that are similar according to Euclidean or cosine distance. By computing $V_u$ from the dual representation of the underlying linear model, we can transform the documents from the original vector space into a vector space, where the above weighted nearest neighbor method gives the same predictions, as the underlying linear model. An important question: are these explanations better than the traditional cosine distance based ones?
Bibliography


http://www.netflixprize.com/assets/ProgressPrize2007_KorBell.pdf


