Collaborative Filtering on the example of the Netflix Prize

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Informatiker (Fachhochschule)

eingereicht an der

Hochschule Anhalt (FH)
Hochschule für angewandte Wissenschaften

Fachbereich Informatik

von

Patrick Ott

geboren am 10.09.1984 in Lutherstadt Wittenberg

Betreuer (HS Anhalt (FH)):	Dr. Krause
Ausgabetermin:	3. Januar 2008
Abgabetermin:	18. Februar 2008

Declaration

I declare that this document and the accompanying code has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. All sources of information have been specifically indicated.

Zusammenfassung

In dieser Diplomarbeit werden verschiedene Ansätze dargestellt, wie man dem Problem beziehungsweise der Herausforderung des Netflix Prize begegnen kann. Nach einer kurzen Einführung zum Thema der Empfehlungssysteme (Recommendation Systems) werden der Netflix Prize selbst, sowie einige einfach Statistiken zum entsprechenden Datensatz vorgestellt. Außerdem behandelt die Diplomarbeit einen zweiten, vom Netflix Prize verschiedenen, Datensatz um die Integrität der Lösungen zu gewährleisten.

Die Diplomarbeit definiert ein mathematisches Framework für die Aufgaben des Collaborative Filtering (Filtern von gleich gesinnten Meinungen). Dieses Framework dient als Hilfestellung zur Definition des Problems des Netflix Prize sowie der potentiellen Lösungsansätze.

Im Hauptteil werden zwei verschiedene Möglichkeiten Vorhersagen in einem Empfehlungssystem zu erzeugen, erklärt: Zuerst wird die Nutzung verschiedener Faktorisierungsmethoden für die Rating-Matrix eines Empfehlungssystems basierend auf der Singulärwertzerlegung erklärt, dann wird die Nutzung einer Nächsten-Nachbarn-Suche erklärt und definiert. Es wird erläutert wie man Ähnlichkeiten zwischen den Filmen des Netflix Prizes definieren kann und wie man diese Ähnlichkeiten nutzen kann um Bewertungen zu schätzen.

Abschließend werden verschiedene Methoden präsentiert, um Vorhersagen verschiedener Schätzer zu kombinieren. Dabei werden drei verschiedene Arten solcher Kombinationen erklärt und definiert. Es handelt sich dabei um die Möglichkeit, die optimale lineare Kombination verschiedener Vorhersagen zu finden, sowie bestimmte Modelle zu verketten, indem man ein Modell die Residuen der Modelle vor diesem vorhersagen lässt. Als eine dritte Alternative wird die Idee der sogenannten verschachtelten Modelle erklärt und kurz untersucht.

Diese Diplomarbeit verfolgt einen experimentellen Ansatz. Das bedeutet, dass nicht jede Entscheidung (zum Beispiel in Bezug auf die Einstellung bestimmter Parameter) erklärt beziehungsweise tiefgreifend untersucht wird.

Abstract

This thesis investigates alternate approaches to the challenge of the Netflix prize. The first chapter contains a brief outline of the theory of recommendation systems.

The second chapter presents the task that must be accomplished to win the Netflix prize and provides some basic statistics describing the datasets involved. It also provides additional examples of recommendation systems.

Next, a basic mathematical framework for Collaborative Filtering tasks is defined. This framework is used to formulate the tasks of the Netflix prize and will also serve as a basis for further implementation of estimators other than those presented here.

The main body of the thesis describes two different algorithms for predicting ratings in a recommendation system of which the Netflix system is one example. The first algorithm is based on the well-known Matrix Factorization approach which has been used by many of the contest participants. The second algorithm is from the family of k Nearest Neighbor algorithms. This section introduces a similarity measure for movies and shows how it may be used to predict new ratings.

Finally, a procedure for combining the predictions of both algorithms to produce a final prediction that is more accurate than the prediction of either one alone, is introduced. Three different ways of combining models are presented. The first method finds the best linear combination of several predictions. The second method concatenates several estimators. Also the so-called nested models are presented and shortly evaluated.

This thesis follows an experimental approach, meaning that not each decission (for example for the setup of certain parameters) throughout the thesis is actually explained or deeply evaluated.

Keywords: Data Mining, Recommender Systems, Collaborative Filtering, Netflix Prize, Matrix Factorization

Grau, teurer Freund, ist alle Theorie und grün des Lebens goldner Baum.

Goethe, Faust I (Mephistopheles)

Für Susi.

Wo auch immer Du jetzt bist.

Acknowledgements

I would like to thank a number of people who gave me a lot of support during my studies of computer science at the Anhalt University or helped me in any possible way with my thesis. First of all I am very grateful to the Data Miners in Boston, especially Michael J.A. Berry who helped me a lot on my thesis and who introduced me to the Netflix Prize after all. Not to mention that it was him who offered me to do my internship at the Data Miners and who was very patient when it came to filling out all the forms for the internship program. I also would like to thank the other staff at the Data Miners office, especially Kathleen Wright for helping me out with all the small things during my time there and Gordon Linoff and Brij Masand for the interesting discussions we had.

I also wish to thank the supervisor of my thesis, Dr. Bernd Krause, for helping me out a lot with it and always giving me valueable input. I would also like to thank Prof. Dr. Schwenzfeger of the Anhalt University for making my exchange semester at the Hangzhou Dianzi University possible. I further thank the computer science department of the Hangzhou Dianzi University for the great time there during my fifth and seventh semester. I am especially grateful to the professors Yigang Wang and Xingqi Wang for everything they did.

I wish to thank my girlfriend, Yu Lingyun, for helping me out a lot with this thesis and also for all the support she gave me during this everlasting time. Schließlich möchte ich mich bei meinen Eltern für die herausragende Unterstützung während meines Studiums und natürlich auch für die Zeit davor bedanken.

0

Contents

Contents

Contents

4

1. Introduction

1.1. Recommendation Systems

Over the past decade and especially since the rapid development of the so called "Web 2.0", recommendation (or recommender) systems based on user opinions gained a high popularity. Basically there are two types of such systems. The content based approach creates and updates profiles of each user and object in the database [41]. Out of those profiles an algorithm is created that helps to give recommendations to users. User profiles could include a lot of information but mostly only those the general user is willing to give to the system, such as demographic information.

The alternative strategy is called Collaborative Filtering (CF), which relies only on past behavior of users and does not require any profiles for users or objects in the system. Such a system has the obvious advantage that it does not need external data, such as the mentioned demographic information. This means that there is firstly no need to collect such data (which is time consuming) and secondly there is also no need to bother the user with it. Also, the CF approach allows to uncover patterns in the user behavior that a content based approach might not be able to discover.

The idea of users giving recommendations to other users or, like most of the times, to a group of users is weakly connected to the topic of Human Computation [58] [60] [59]. In Human Computation the uniqueness of the human being and way of thinking is used to break tasks today's computers and algorithms can not solve efficiently on their own. In a CF system humans are used to form global opinions about objects in the system. Therefore it is possible to treat CF systems as a replacement for content based systems.

However, combinations of both systems can prove usefull [6] [38] [4] [44] [40].

Basically there are two different types of CF systems. Those two types are called explicit systems and implicit recommender systems. They differ in the form how they collect their data. An explicit recommender system is collecting data in the following ways

- Asking a user to rate an item/object on a scale. Such a scale is not necessarily ordinally scaled.

- Presenting two items to a user and asking him/her to choose the best one.

- Asking a user to create a list of items that he/she likes ("Wishlist"-feature).

- Asking a user to rank a collection of items from favorite to least favorite (not very common).

while an implicit system uses one of the following ways

- Observing the items that a user views in the system (also analyzation of viewing times is possible).

- Keeping a record of the items that a user purchases online.

- Obtaining a list of items that a user has ordered, listened to or watched.

- Analyzing the user's social network and discovering similar likes and dislikes (neighbor search).

So while with an explicit system the user is giving his opinion directly, with an implicit system the user is not asked for any opinion at all but his behavior while moving through the system and working with it is analyzed. This method has the advantage that it does not bother the user to give a rating. The disadvantage is, of course, that such an implicit system does not offer the preciseness of an explicit system.

A CF system is usually used to present users of the system items that they might like (or even not like). Besides those obvious functions such a system usually also fulfills more functions like providing the operator of the system with useful information which object might be a popular object in the future or providing information that could be used for marketing proposes. This makes especially sense for the very popular audio- and videoindustry. The KDD-Cup of the year 2007 (see section 1.2.2) hold one task for the competitors, that asked to predict the number of ratings the users in a recommendation system will give to certain objects in the next year. Such a question is especially interesting for the company or institution that runs the system. The answer to this question might reveal how many copies of such an object (in this case it had been movie DVDs) must be ordered to satisfy the requests of the users in the next year.

In this thesis we will concentrate only on explicit systems.

1.2. Motivation and Examples

1.2.1. The Netflix Prize

Throughout this thesis we will deal with a challenge that involves beating the famous recommendation system CineMatch, which was implemented by Netflix[1] to serve their customers with suggestions which movies they would like to order next from the rental system.

Netflix is the largest online DVD rental service in the United States of America. They claim their portfolio consists of over 80000 titles on 42 million DVDs. The service has 6.8 million subscribers. Those users gave more than 1.7 billion ratings for the movies in the portfolio. On average a member rates 200 movies [2].

The Netflix Prize[3] [10] is a competition that was launched by Netflix in the year 2005. The task is to learn from a training dataset of approximately

---

[1] http://www.netflix.com
[2] http://www.netflix.com/MediaCenter?id=5379&hnjr=8#facts
[3] http://www.netflixprize.com/

1. Introduction

100 million ratings that nearly half a million users (480189 to be exact) gave to 17770 movies and finally predict another 2.8 million ratings. Those ratings are given as a quadruplet consisting of the user (an integer ID), the movie (an integer ID), the date of the rating (day-precise) and the rating itself (an integer from 1 to 5). Netflix also provides the competitors with a probe dataset, that is included in the training data itself. However, for this probe set Netflix gives the score their recommendation system CineMatch is achieving. As an error measurement the Root Mean Square Error (RMSE), also known as the Root Mean Square Deviation (RMSD), is used. Netflixs own forecasting algorithm, which is called Cinematch, is able to give a RMSE of 0.9525. Any algorithm that is capable of improving this result by 10% (that is a RMSE of equal or lower than 0.8563) will be considered for the $1000000 grand prize. If, within one year, nobody was able to win the grand prize, a progress prize of $50000 will be awarded to the best solution by then. Over 19440 teams from 152 countries take part in the contest. By the time of June 2007 there were roughly 14000 submissions.

Our goal in this thesis will be to analyze the Netflix Prize dataset and implement some simple forecasting algorithms to outline what is possible. We will mostly restrict ourselves to the information that is given by the User (as an ID), movie (as an ID) and the rating. We will not use the information that is provided by the timestamp. For a time-sensetive view on the dataset of the Netflix Prize the reader is referenced to the proceedings of the KDD Cup and the KDD Cup itself [45] [28] [25] [15].

Besides the obvious mathematical motivation that the Netflix Prize provides, there is also another factor that should not be ignored: The computational factor. Today's 32-bit computer systems are able to handle a maximum of 4 GB local memory, on a Windows-based system even less[1]. Each of the newer Windows System with NT-core can provide a *single* program with a maximum of $2^{32}/2$ bytes $\approx$ 2 GB. The 17770 movies and half a million raters, that are given in the Netflix dataset, would form a 17770 $\times$ 480189 rating

---

[1]For further information on this issue the reader may consult the Microsoft Developer Network (MSDN) at `http://msdn2.microsoft.com/en-us/library/aa366525.aspx`

matrix with roughly 8.5×10^9 cells. Of this matrix we would only know approximately 100 million cells. The unknown cells stay empty. Considering that we save the ratings (from one to five) in the datatype byte that takes 8 bit, such a matrix would require at least 8.5 GB of RAM, which clearly overcomes the 32-bit restrictions. Besides the issues of space and memory another problem is the time computations do require. As can be easily seen, common Data Mining approaches such as neural networks or decision trees, will fail here because of two reasons: Firstly, the necessary data to feed such an algorithm is not provided, we would have to produce it; Secondly, such learning algorithms would just work too slow on such a huge dataset. Another motivation might be that Netflix is asking the users to work without software that would require any further licensing. This means programming has to be carried out in a free programming language. We will use C++ for speed-sensitive calculations (the main algorithms). For all other tasks, such as rearranging the datasets, we used the .NET-programming language C# due to its obvious advantages in usability compared to C++.

The algorithms in this thesis were developed and implemented to approach the Netflix Prize, hence the Prize will be our guideline through the thesis.

1.2.2. The KDD Cup 2007

Every year the KDD conference (Knowledge Discovery and Data Mining) comes with a cup that is referred to as the KDD Cup. The KDD Cup of the year 2007 [9] [4] dealt with the datasets of the Netflix prize. In detail two tasks were presented. The first task is to predict for a given User/Movie-combination (both given as IDs) whether the rating was done in 2006 or not. The second task asks the participants to estimate the number of ratings for a subset of the 17770 movies given by the original Netflix Prize. In this thesis we will not present any solution to those two tasks. However, at least the second task is a perfect example of how a recommendation system can be used in other ways than giving suggestions to the user. By estimating the number

---

[4]`http://www.cs.uic.edu/Netflix-KDD-Cup-2007`

of ratings for a movie, we can also give an outlook on how many movies will be rented overall. This means Netflix will be able to predict the number of DVD copies it would require to suit the needs of the customers. This information should not be underestimated because it will keep customers happy, since waiting times stay low and it will also optimize Netflixs financial efficiency, since the number of DVDs for a movie can be lowered (DVDs are sold on a regular basis) or increased (in case more users will want to watch the movie). The author participated in the further task ("how many ratings in 2007"), although it is not described in this thesis. On the naive RMSE-scale the team of the author placed 5'th. Later in the contest the organizers changed the scale to a RMSE that is based on the number of ratings flattened by the natural logarithm. On this scale the author's team achieved the 17'th place. For the winning solutions of task 1 see [31] [56] [36] [35] and for task 2 [45] [28] [25] [15].

1.2.3. Other examples

Well known recommender systems are those of webpages that emerged during the Web 2.0 boom and deal with multimedia content, such as selling music or sharing videos. Famous examples are iTunes[5] from Apple and YouTube[6] from Google. Also eBay offers a recommender system by telling the user what he or she might be interested in, just by looking at the previous viewed items.

Very similar to the Netflix recommender system is a problematic that the TiVo recommender system approaches [3]. TiVo is a popular brand of digital video recorder (DVR) in the United States. Their recommender system obviates the need to keep any persistent memory of each user's viewing preferences. At the same time they implemented a computational efficient system, which is based on the TiVo client-server system.

Another recommender system to mention is the system of the online merchant Amazon.com. It also gives a user suggestion of what he or she might

---

[5]`http://www.itunes.com`
[6]`http://www.youtube.com`

be interested in by keeping track of the user's movement through the system. It is documented in [34]. Amazon claims that 20% of their sales resulted from personal recommendations.

For a recent survey on recommender systems we refer to [2]. For more surveys and further reading we would like to refer the reader to [44] [24] and [50].

1.2.4. Organization of the thesis

The rest of the thesis is organized as follows: In chapter 2 we explain the Netflix Prize in detail and introduce the basic mathematical model to approach the problem of recommending items of the system. While in chapter 3 we explain the Matrix Factorization technique we are going to use to predict ratings, we explain in chapter 4 a Nearest Neighbor approach to the problem. Finally we explain how to combine different predictions from different algorithms to obtain a better solution than any single algorithm in chapter 5. Chapter 6 concludes the performance of the different prediction methods. We summarize the work and give an outview on what could be done in the future in Chapter 7. Part A of the appendix gives a short introduction in the programming techniques that were used to implement the solutions to the problem of predicting ratings in a recommender system. Part B of the appendix gives further experiments and examples. It is an addition to chapter 6. In Part C we explain the content of the DVDs that come with the thesis.

1. Introduction

2. The Netflix Prize in Detail

2.1. Basic Statistics

We already stated that the Netflix Prize consists of $p = 480189$ users, which are a subset of the total 6.8 million subscribers Netflix states to have. Those 480189 users assigned exactly 100480507 ratings to $17770 = q$ movies. This results in a rating matrix R, which saves each rating a user gave to a movie, with a total of $480189 \times 17770 = 8532958530$ elements, of which we only know roughly 1.2%.

Figure 2.1 shows the distribution of all ratings. The most often given rating is 4, which is also the median.

Table 2.1 shows the five most often rated movies. It is quite interesting that although "Independence Day" is much older than "Miss Congeniality" it still has fewer ratings.

Table 2.1 gives the five most popular movies (and series). In general there
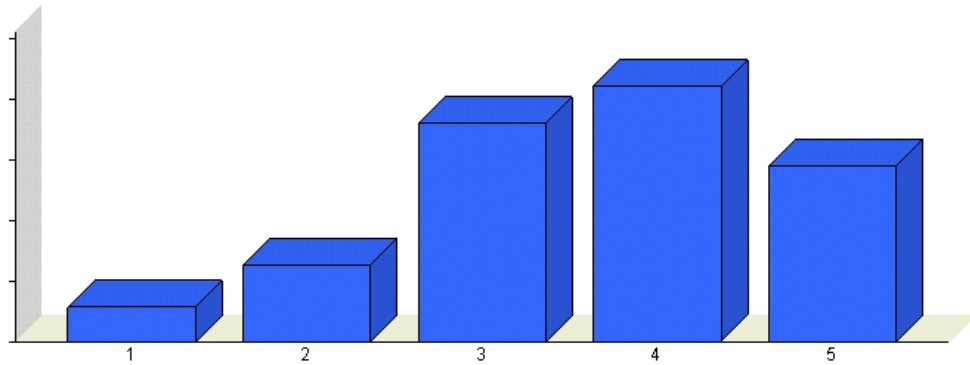
Figure 2.1.: Distribution of all rating instances

No.	Movie	Ratings
1	Miss Congeniality	232944
2	Independence Day	216596
3	The Patriot	200832
4	The Day after Tomorrow	196397
5	Pirates of the Caribbean I	193941

Table 2.1.: Top 5 of most often rated movies

No.	Movie	Mean Rating
1	Lord of the Rings III	4.7233
2	Lord of the Rings I	4.7166
3	Lord of the Rings II	4.7026
4	Lost, Season 1	4.6710
5	Battlestar Galactica, Season 1	4.6388

Table 2.2.: Top 5 best rated movies

are a lot of series in the top rated medias. This is because many people know whether they would like or would not like a series, since they already watched at least an episode of the series. Movies on the other hand are also ordered by people who might not really like them but do not know it yet.

Figure 2.2 shows the distributions of number of ratings per movie. It clearly shows that the majority of movies has a small number of ratings compared to a minority of movies with a lot of ratings[1]. The same effect can be observed with the users (see Figure 2.3). Only a small number of users actually rate a lot. In average each movie gets rated 5654 times but the median is only 561. "Mobsters and Mormons" is the least often rated movie with only 3 ratings. It is one of two movies with raters in the single digits. The other one is "Land Before Time IV".

The top 616 movies account for 50% of the ratings, the top 2,000 movies account for 80% of the ratings and the top 4,000 movies account for 90%. This is interesting because it shows that it is not so important to have an

---

[1]For a detailed view on this observation and a proposal how to isolate those "popular" movies see section 4.4.2.
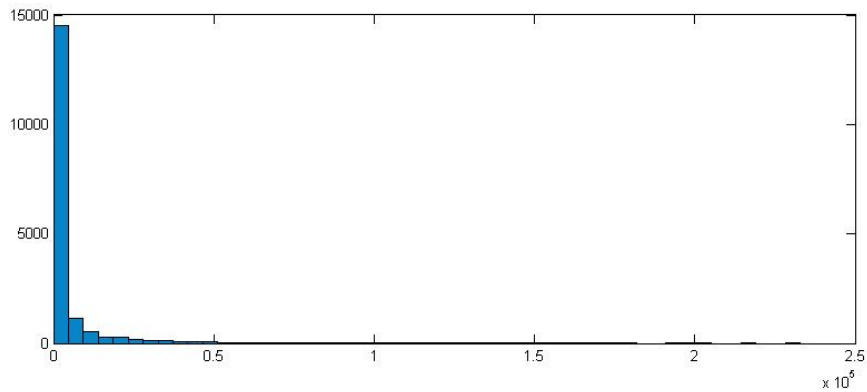
Figure 2.2.: Distribution of the number of ratings per movie
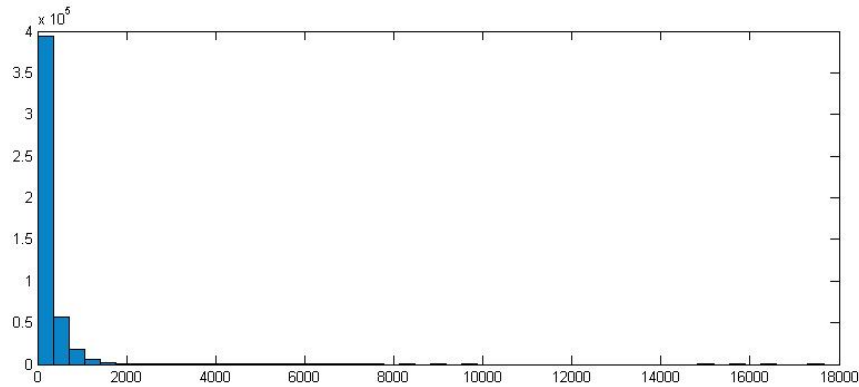
Figure 2.3.: Distribution of number of ratings per user

algorithm that works *very* well on all movies in the database but that it is also possible to have an algorithm that performs extremely well on a small number of movies and only good on the rest.

Table 2.1 shows the top five users in terms of the number of ratings they did. Those users are especially interesting, since all of them rated more than 80% of the movies. As can be seen from the table their ratings are extremely one-sided. They either vote many movies really bad (top three) or very good (spot four and five). Any prediction method that is based on user similarity can suffer from this, since those users are actually nearly any other user's neighbor. On the other hand, a method that is based on movie

15

No.	UserID	Ratings	1	2	3	4	5
1	305344	17653	9523	3037	2990	1397	706
2	387418	17436	7919	5916	2841	531	229
3	2439493	16565	15024	421	523	263	334
4	1664010	15813	5	320	3282	4089	8117
5	2118431	14831	6	210	3192	6572	4851

Table 2.3.: The users with the most ratings

similarity, like the one we are going to propose, can use this information very well since the method will find many neighbors that got rated low or high and therefore can figure out that the unrated example also gets a high or low rating. Basically these users do not deserve to be called "users", since it is highly unlikely a human being rated this many movies.

Figure 2.4 shows the distribution of the mean of all ratings of a user while Figure 2.5 shows the standard deviation of the ratings of each user. Both distributions are pretty much normal distributed. Interesting are the bumps on the very right of the user means and on the very left of the users standard deviation. It means that there are quite a lot of users that actually rated every movie five. Furthermore we can also see a bump at a rating of three and also the bar of four is relative high compared to the others. The bump on the left side of the standard deviation distribution also means that some raters rate all movies the same, since their standard deviation is zero. This information is important, we can actually predict the ratings of those users easily by just taking their mean (which is usually an integer). Interesting is also the little bump in the users standard deviations on the right of the histogram. Those are the users that rate a movie either 1 or 5.

2.2. Mathematical Model

We define the set U, describing the users

$$U = \{u_1, u_2, ..., u_p\}$$
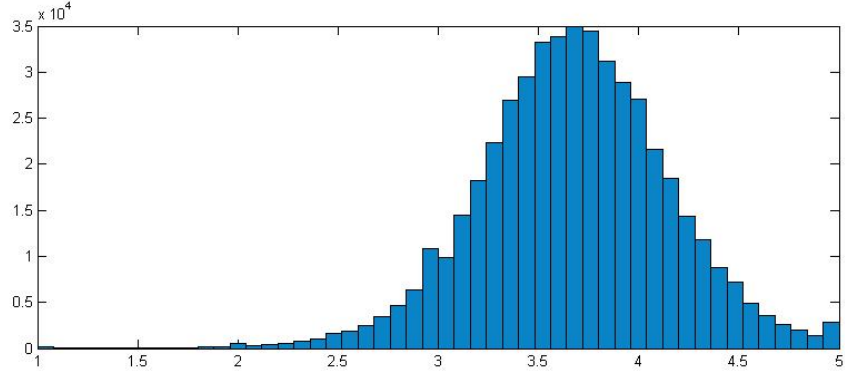
Figure 2.4.: Distribution of average user ratings
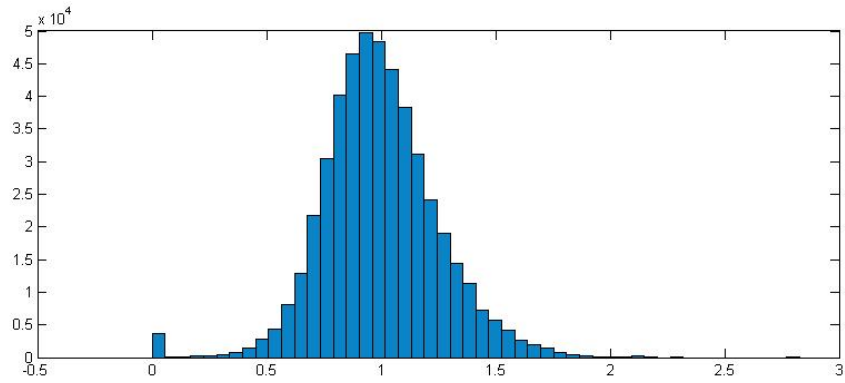
Figure 2.5.: Distribution of standard deviation of the ratings of each user

2. The Netflix Prize in Detail

For the Netflix Prize we have $p = 480189$ users. The set M is holding the movies

$$M = \{m_1, m_2, ..., m_q\}$$

with $q = 17770$. We define the set T as all the ratings we know

$$T \subset U \times M \times \{1, 2, 3, 4, 5\}$$

T defines all known ratings by a user and a movie and the rating the user gave to the movie. For the Netflix example the cardinality of this set is 100480507. T is the set we know and we will learn from. Also the dataset given by the Netflix Prize provides the participants with a date, which we do not make any use of.

The set Q is known as the qualification set. It contains of all the user/movie-pairs that the Netflix Prize asks us to qualify. The ratings for this set are of course withhold. For any other task than the Netflix Prize this set can be thought of as a final validation set, which is used to test the performance of the prediction models.

$$Q \subset U \times M$$

We also define the function e, it explains the mapping of a user $u_i \in U$ and a movie $m_j \in M$ to a rating r_{ij}.

$$e : U \times M \to O$$

$$e(u_i, m_j) = r_{ij}$$

O is the set of possible rating options. For the Netflix example it is $O = \{1, 2, 3, 4, 5\} \in \mathbb{R}$. e is not a model that predicts ratings for a given user / movie pair. The function e is just explaining every rating a user u gave or will (would) give to a movie m, so it also explains all rating instances in T

$$(u, m, r) \in T \Rightarrow e(u, m) = r$$

Now we are able to formally define the rating matrix R as

$$R = (r_{ij} = e\,(u_i, m_j))_{i=1,\ldots,p,\,j=1,\ldots,q}$$

and it is obviously true that in case $e\,(u_i, m_j)$ is defined for the user/movie-pair (u_i, m_j) it also has a place in R.

$$(u_i, m_j, r_{ij}) \in T \Rightarrow r_{ij} = e\,(u_i, m_j)$$

Remember here that R is the complete rating matrix. There are no empty cells in it. This matrix usually does not exist because it is unlikely that in a recommendation system each user rated every single movie. Furthermore we might only know a subset of this matrix, like we do in the Netflix Prize. The matrix that only carries the ratings we actually do know is called $\hat{R}$ and has the same dimensions as the matrix R.

$$\hat{R} = (\hat{r}_{ij})_{i=1,\ldots,p,\,j=1,\ldots,q}$$

Only those cells of $\hat{R}$ are reserved that actually also exist in T as a rating instance.

$$(u_i, m_j, r_{ij}) \in T \Leftrightarrow \hat{r}_{ij} \in \hat{R} \text{ is not empty}$$

As we already stated the goal of a recommendation system is to recommend new items a user might be interested in. To do so we need an estimator in form of an algorithm, that is able to learn from old ratings to predict ratings that have not been done yet. In this way we can predict a high rating a user might give to a movie and for this reason the system can recommend this movie to the user. We define the estimator $\hat{e}$ as

$$\hat{r}_{ij} = \hat{e}\,(u_i, m_j, \underline{\lambda}) \approx e\,(u_i, m_j) = r_{ij}$$

$\underline{\lambda}$ is a vector of parameters the estimator makes use of next to the already given learning examples in T. The system of the Netflix Prize allows real

2. The Netflix Prize in Detail

ratings in the interval of $[1; 5] \subset \mathbb{R}$, so it is

$$\hat{e}(u_i, m_j, \underline{\lambda}) \in [1; 5] \subset \mathbb{R}$$

Furthermore we define $\hat{e}$ by giving

$$e(u_i, m_j) = \hat{e}(u_i, m_j, \underline{\lambda}) + Z$$

and Z is the random variable which covers the error and holds the residuals. Z is often normally distributed with

$$Z \sim N(\mu, \sigma^2)$$

with mean μ and variance σ^2.

Since the Netflix Prize dictates the Root Mean Square Error (RMSE) as the tool for error measurement we can obviously define the goal of the prize to minimize this RMSE on the qualification set Q.

$$\left(\frac{\sum\limits_{(u_i, m_j) \in Q} (\hat{e}(u_i, m_j, \underline{\lambda}) - e(u_i, m_j))^2}{card(Q)} \right)^{\frac{1}{2}} \rightarrow \min$$

To train $\hat{e}$ it requires good parameters $\underline{\lambda}$. Minimizing the RMSE of the training set will produce these parameters. However, using the training set for validation as well will lead to overtraining. Therefore we remove a validation set V from T

$$V \subset T$$

and we use $T \backslash V$ for training. For a good performance on the validation set we need to learn on a big variety of training examples: $card(V) \ll card(T \setminus V)$. In our example we remove approximately 3 million ratings from T and use

them for validation. Now we can define the learning algorithm clearly by

$$\left(\frac{\displaystyle\sum_{(u_i, m_j, r_{ij}) \in V} \left(\hat{e}\left(u_i, m_j, \underline{\lambda}\right) - e\left(u_i, m_j\right) \right)^2}{card\left(V\right)} \right)^{\frac{1}{2}} \rightarrow \min_{\underline{\lambda}}$$

which is equivalent to only minimizing the numerator of the fraction under the squareroot

$$\Leftrightarrow \sum_{(u_i, m_j, r_{ij}) \in V} \left(\hat{e}\left(u_i, m_j, \underline{\lambda}\right) - e\left(u_i, m_j\right) \right)^2 \rightarrow \min_{\underline{\lambda}} \tag{2.1}$$

while $\hat{e}$ is training on the examples of $T \setminus V$. We can treat equation 2.1 as a second objective function next to the optimization problem of

$$\sum_{(u_i, m_j, r_{ij}) \in T \setminus V} \left(\hat{e}\left(u_i, m_j, \underline{\lambda}\right) - e\left(u_i, m_j\right) \right)^2 \rightarrow \min_{\underline{\lambda}} \tag{2.2}$$

In such a setup equation 2.1 has a higher priority than the objective function of equation 2.2 . Meaning that in case the maximum performance on V (the smallest possible RMSE) is reached training stops although it might still be possible to minimize the performance on $T \setminus V$.

The following chapters will approach this problem and give examples of how to minimize these objective functions. We will present two different algorithms $\hat{e}$. One that is based on the idea of Matrix Factorization (MF) and another one that is based on the principal of a Nearest Neighbor (NN) search.

2.3. Datasets of the Netflix Prize

To clear out misunderstandings we will shortly outline the available datasets of the Netflix Prize as well as the datasets we created out of them.

- **Netflix Training dataset:** 100480507 rating instances. This is the set T. This set seperates into

- **Training:** The training set we do actually use. This is $T \setminus V$.

- **Validation:** The validation set we use to validate the performence of our algorithms. This is the set V [2].

- **Netflix Qualification dataset:** This is the qualification set Q provided by Netflix. Once this set is predicted one can send it to Netflix using the provided webinterface.

- **Netflix Probe dataset:** A validation set provided by Netflix. We do not use this dataset in this thesis.

---

[2]Note here that for the calculation of Matrix Factorizations, which are explained later in the thesis, we use another smaller validation set which we remove from V. This set is called V_{MF}. So in the case of a Matrix Factorization the validation set V is replaced by $V \setminus V_{MF}$.

3. Matrix Factorization

The usage of Matrix Factorization (MF) for the Netflix Prize was firstly suggested by Brandyn Webb (a.k.a. Simon Funk) in his journal[1]. Since Singular Value Decomposition (SVD) [1] [22] [7] [18] [19] [21] is strongly connected to MF we are going the outline the basic idea behind SVD first.

3.1. Singular Value Decomposition

The basic idea behind the classical SVD-approach is to decompose a given rating matrix P of size $n \times m$ into three smaller matrices.

$$P = A\Sigma B^T$$

where A is of size $n \times n$, Σ of size $n \times m$ and B of size $m \times m$. Both matrices, A and B, are orthogonal matrices. The matrix Σ is a diagonal matrix with k non-zero entries. Therefore the effective dimensions of the three matrices above are $n \times k$, $k \times k$ and $k \times m$. These k diagonal entries ς_i of the matrix Σ are all positive with $\varsigma_1 \geq \varsigma_2 \geq ... \geq \varsigma_k > 0$. The columns of A are called the left singular vectors of P. Often they are also called the orthonormal eigenvectors of AA^T. The columns of the matrix B are called the right singular vectors of P (the orthonormal eigenvectors of $A^T A$).

If we now retain only the $r \ll k$ greatest singular elements, the product of those three matrices will be a matrix of rank r, namely $\hat{P}$. Furthermore this matrix is the closest rank-r approximation of P in terms of the Frobenius

---

[1]`http://sifter.org/~simon/journal/20061211.html`

3. Matrix Factorization

Norm.

$$\left\| \hat{P} - P \right\|_F^2 = \sum_{i=1}^{n} \sum_{j=1}^{m} |\hat{p}_{ij} - p_{ij}|^2 = trace\left(\left(\hat{P} - P \right) \left(\hat{P} - P \right)^T \right)$$

and A^T is the transpose of the matrix A.

SVD is just one of the many decomposition techniques for matrices of rectangular size [53] [17].

The Netflix Prize provides us with a matrix R that is very sparse, it has many unknown values. So the classical SVD-approach will only work with some tricks, like replacing the unknown elements with zero or using other more sophisticated methods [30] [12] [46]. By using incremental learning (which should be understood as learning data instance by data instance) we will avoid this problem. Incremental Matrix Factorization is just one way to implement incremental learning.

3.2. Incremental Matrix Factorization

3.2.1. The Base Algorithm

The technique presented here is very similar to the basics of SVD. Instead of decomposing R into three matrices we only focus on two matrices, called the feature-matrices.

$$R \approx SV = \hat{R}$$

where S now is a $p \times f$ matrix, and V is a $f \times q$ matrix. f is the number of used features. Those features desribe a user (matrix S) or a movie (matrix V). User u_i is described by row $\underline{s}_i^T$ of matrix S and each movie m_j is described by a column $\underline{v}_j$ of matrix V. For example we can think of a feature "action" that identifies how much action a movie flavors and how much a given user likes action-movies. The original approach, using three matrices, can be

24

transformed into the form of only using two matrices by

$$R = A\Sigma B = \left(A\sqrt{\Sigma}\right)\left(\sqrt{\Sigma}B\right)$$

A single prediction of a rating of a movie j by user i is then given by the dot product of the feature vectors of the user u_i and the movie m_j

$$e\left(u_i, m_j\right) = r_{ij} \approx \hat{r}_{ij} = \hat{e}_{MF}\left(u_i, m_j, \underline{\lambda}\right) = \underline{s}_i^T \underline{v}_j = \sum_{k=1}^{f} s_{ik} v_{kj}$$

where s_{ik} and v_{kj} are items of the vectors $\underline{s}_i$, $\underline{v}_j$. We define $\underline{\lambda}$ as all the elements of those two vectors[2].

$$\underline{\lambda} = \left(\begin{array}{cc} \underline{s}_i^T & \underline{v}_j^T \end{array}\right)^T$$

Now we can write out the quadratic error between the estimated rating and the real rating by

$$\varepsilon_{ij}^2 = (r_{ij} - \hat{r}_{ij})^2 = \left(r_{ij} - \sum_{k=1}^{f} s_{ik} v_{kj}\right)^2$$

To minimize this error, $\varepsilon_{ij}^2 \to \min_{\underline{\lambda}}$, we differentiate the equation and obtain

$$\frac{\partial \varepsilon_{ij}^2}{\partial \underline{\lambda}} = \frac{\partial}{\partial \underline{\lambda}}\left(r_{ij} - \sum_{k=1}^{f} s_{ik} v_{kj}\right)^2 = 2\left(r_{ij} - \sum_{k=1}^{f} s_{ik} v_{kj}\right)\left(-\frac{\partial}{\partial \underline{\lambda}}\sum_{k=1}^{f} s_{ik} v_{kj}\right)$$

which simplifies to

$$\frac{\partial \varepsilon_{ij}^2}{\partial \underline{\lambda}} = -2\left(r_{ij} - \sum_{k=1}^{f} s_{ik} v_{kj}\right)\left(\begin{array}{cc} \underline{v}_j^T & \underline{s}_i^T \end{array}\right)^T$$

---

[2] $\underline{\lambda}$ can be thought of as a container that stores all variables of which the estimators make use of. For the MF algorithms these variables are the ones we are going to optimize. $\partial \varepsilon_{ij}^2 / \partial \underline{\lambda}$ means the derivation of function ε_{ij}^2 with respect to all items of $\underline{\lambda}$ indicated by $\lambda_1, \lambda_2, ..., \lambda_n \nearrow \partial \varepsilon_{ij}^2 / \partial \underline{\lambda} = \left(\partial \varepsilon_{ij}^2 / \partial \lambda_1, \partial \varepsilon_{ij}^2 / \partial \lambda_2, ...\partial \varepsilon_{ij}^2 / \partial \lambda_n\right)^T$

3. Matrix Factorization

Now we can obtain an update equation[3] for each feature of S and V by

$$s'_{ik} = s_{ik} + 2\alpha \left(r_{ij} - \sum_{a=1}^{n} s_{ia}v_{aj} \right) v_{kj}$$

and

$$v'_{kj} = v_{kj} + 2\alpha \left(r_{ij} - \sum_{a=1}^{n} s_{ia}v_{aj} \right) s_{ik}$$

α is the learning rate. A number that explains how much of the errors slope is added to the new feature value. α should be rather small, around 0.001. n is the number of features that should be included. The algorithm we implemented is using $n = k$, meaning that each feature below the actual trained feature is included into the error equations and all other features are ignored. Another possibility is to include $n = f$, which however will add a few more calculations to the process. With the help of the update equations we can define the basic learning algorithm as follows

- **Step 1:** Initialize the features matrices S and V for the first time, either by a fixed number or randomly.

- **Step 2:** Loop until number of features f reached.
    - Loop[4] until a terminal condition is met
        * Iterate over each known element of R and update the feature vectors with the help of the update equations.
        * Recompute the RMSE on the training set or a validation set or both.

- **Step 3:** Compute the final RMSE and save feature matrices for later usage.

---

[3]These update equations follow directly out of the solution of $\partial\varepsilon_{ij}^2 / \partial\underline{\lambda}$. The idea to use update equations is based on the method of steepest descent. The difference here is that we minimize each summand of the RMSE equation separately. For a description on the method of steepest descent as well as its application to the Netflix Prize the reader might consult section 3.3.

[4]These loops within a feature training step are called "iterations" or "epochs" throughout the whole thesis.

3.2.2. Variations of the Base Algorithm

There are plenty of variations of this algorithm. One might consider training one feature at a time while another one might consider training all features simultaneously. It turns out that the later method converges faster in terms of required iterations. However, in our implementation we noticed that, though the need of more iterations over all, the first method works faster because there are several ways to tune the programming code by caching already calculated features. An advantage of the simultaneous training is that the variance of each feature does not decline as it does when each feature is trained independently. This means that the first feature already covers a huge part of the prediction while the later features only try to predict smaller residuals.

3.2.3. Linear additions to the method of iterative MF

Addition of regularizations parameters to MF

A few additions to the iterative MF have been proposed. One of them includes adding regularizations with a factor β to the update equations. This addition will partially suppress overtraining and therefore improve performance on unseen examples. The updates equations change to

$$s'_{ik} = s_{ik} + \alpha \left(2 \left(r_{ij} - \sum_{a=1}^{n} s_{ia} v_{aj} \right) v_{kj} - \beta s_{ik} \right)$$

and

$$v'_{kj} = v_{kj} + \alpha \left(2 \left(r_{ij} - \sum_{a=1}^{n} s_{ia} v_{aj} \right) s_{ik} - \beta v_{kj} \right)$$

The equivalent[5] error equation for these update equations is

$$\varepsilon_{ij}^2 = \left(r_{ij} - \sum_{a=1}^{n} s_{ia} v_{aj} \right)^2 + \beta \frac{1}{2} \sum_{k=1}^{n} \left(s_{ik}^2 + v_{kj}^2 \right)$$

---

[5]The error equation is equivalent in terms of that its derivation will result in a gradient, of which each component will be equivalent to the here presented update equations.

We call $\beta \frac{1}{2} \sum_{k=1}^{f} \left(s_{ik}^2 + v_{kj}^2 \right)$ the regularization term.

Although some sources[6] point out that these regularization parameters are somehow related to the technique used in Tikhonov regularization, we think the explanation why this addition suppresses overtraining, is rather simple. Since the variance of the features is declining during training, the regularization parameter βs_{ik} respectively βv_{kj} are also declining with each new feature trained. Meaning that those additions introduce an artificial error to the update equation, especially for the first features with a very high variance. This error declines over time. To put it simple it moves some weight from the earlier features to the later ones and therefore the algorithm can train longer and suppress overtraining a bit.

Linear Matrix Factorization

Paterek [39] suggested to add biases to the prediction model. He proposed to include two constants, one for each movie and one for each user, in the equation of $\hat{e}$. Those constants are trained simultaneously. We propose a slightly different modification of $\hat{e}$. For each feature we implement two constants, like proposed by Paterek, and train them together with the feature matrices. We call this method Linear Matrix Factorization (LMF)

$$\hat{e}_{LMF}\left(u_i, m_j, \underline{\lambda}\right) = \sum_{k=1}^{f} s_{ik} v_{kj} + c_i + d_j$$

where the weights c_i and d_j are trained simultaneously with the movie- and userfeatures. We can obtain the update equations by extending $\underline{\lambda}$ to

$$\underline{\lambda} = \left(\begin{array}{cccc} \underline{s}_i^{\,T} & \underline{v}_j^{\,T} & c_i & d_j \end{array} \right)^T$$

---

[6]Like Brandyn Webbs article about MF-techniques for the Netflix Prize: `http://sifter.org/~simon/journal/20061211.html`

and redefining ε_{ij} to

$$\varepsilon_{ij}^2 = \left(r_{ij} - \left(\sum_{k=1}^{f} s_{ik}v_{kj} + c_i + d_j \right) \right)^2$$

and therefore

$$\frac{\partial \varepsilon_{ij}^2}{\partial \underline{\lambda}} = 2 \left(r_{ij} - \left(\sum_{k=1}^{f} s_{ik}v_{kj} + c_i + d_j \right) \right) \left(-\frac{\partial}{\partial \underline{\lambda}} \left(\sum_{k=1}^{f} s_{ik}v_{kj} + c_i + d_j \right) \right)$$

The update equations for the elements of the feature matrices stay unchanged. The update equations for c_i and d_j are given by

$$c_i' = c_i + \chi \left(2 \left(r_{ij} - \left(\sum_{k=1}^{f} s_{ik}v_{kj} + c_i + d_j \right) \right) - \delta c_i \right)$$

and

$$d_j' = d_j + \chi \left(2 \left(r_{ij} - \left(\sum_{k=1}^{f} s_{ik}v_{kj} + c_i + d_j \right) \right) - \delta d_j \right)$$

χ is the learning rate for the constants, while δ is another regularization parameter. The final error for a given rating and its prediction by the estimator including all regularization parameters is

$$\varepsilon_{ij}^2 = \left(r_{ij} - \sum_{a=1}^{n} s_{ia}v_{aj} + c_i + d_j \right)^2 + \beta \frac{1}{2} \sum_{k=1}^{n} \left(s_{ik}^2 + v_{kj}^2 \right) + \chi \frac{1}{2} \sum_{k=1}^{n} \left(c_i^2 + d_j^2 \right)$$

3.2.4. Non-Linear additions to the method of iterative MF

Non-Linear Matrix Factorization

A further addition we want to add to the estimator is a non-linear inclusion of the features of each user and movie. The estimator changes to

$$\hat{e}_{NLMF}(u_i, m_j, \underline{\lambda}) = \sum_{k=1}^{f} \left(\sum_{a=1}^{r} \sum_{b=1}^{r} \theta_{abk} s_{ik}^{a} v_{kj}^{b} \right) + c_i + d_j$$

where $\sum_a \sum_b \theta_{abk} s_{ik}^a v_{kj}^b$ is called the internal polynomial kernel of the feature k of the estimator $\hat{e}_{NLMF}$. θ_{abk} is a factor multiplied with every non-linear and linear element. This factor allows the algorithm to decide which of the linear or non-linear parts of the equation are more important. There are r^2 different factors for each feature. We can arrange those coefficients in a vector for each feature

$$\underline{\Theta}_k = (\theta_{11k}, \theta_{12k}, ..., \theta_{1rk}, \theta_{21k}, ..., ..., \theta_{rrk})^T$$

The update equations for the elements of the feature vectors $\underline{s}$ and $\underline{v}$ change to

$$s'_{ik} = s_{ik} + \alpha \left(2 \left(r_{ij} - \sum_{g=1}^{n} \sum_{a=1}^{r} \sum_{b=1}^{r} \theta_{abg} s_{ig}^a v_{gj}^b + c_i + d_j \right) \right.$$
$$\left. \left(\sum_{a=1}^{r} \sum_{b=1}^{r} a\theta_{abg} s_{ig}^{a-1} v_{gj}^b \right) - \beta s_{ik} \right)$$

$$v'_{kj} = v_{kj} + \alpha \left(2 \left(r_{ij} - \sum_{g=1}^{n} \sum_{a=1}^{r} \sum_{b=1}^{r} \theta_{abg} s_{ig}^a v_{gj}^b + c_i + d_j \right) \right.$$
$$\left. \left(\sum_{a=1}^{r} \sum_{b=1}^{r} b\theta_{abg} s_{ig}^a v_{gj}^{b-1} \right) - \beta v_{kj} \right)$$

The update equations for the constant feature vectors $\underline{c}$ and $\underline{d}$ now include the complete error caused by the internal polynomial kernel and are straight forward.

For fixed feature vectors $\underline{s}$ and $\underline{v}$ as well as $\underline{c}$ and $\underline{d}$ it is possible to find the optimal parameters $\underline{\Theta}$ directly by solving a least square problem. This method is very similar to combining different predictors, in this case these are the r^2 summands of the internal polynomial kernel. For a detailed analyzation as well as the description of the least square problem and its solution the reader is refered to chapter 5.

The problem with the exact solution is that it is often too precise and takes any learning chance from the later epochs. This means that the algorithm is iterating too fast and overtrains quickly. Therefore we only suggest using the exact solution at the end of each feature training, to do a final optimization.

To learn the factors at the same time as the other features and constant we once again use the incremental approach using update equations. However, those update equations are different from those of the features and constants we introduced before. The point is that there is not one polynomial factor for each user and movie but only one for each polynomial component per feature. This means that we do not need to train them in combination with each user/movie pair but only at the end of each epoch. Another idea is to train them only every second or third epoch, which again will suppress overtraining. The goal is to minimize the error equation given by

$$\sum_{(u_i,m_j,r_{ij})\in T\setminus V} \varepsilon_{ij}^2 = \sum_{(u_i,m_j,r_{ij})\in T\setminus V} \left(\sum_{g=1}^{n}\sum_{a=1}^{r}\sum_{b=1}^{r} \theta_{abg} s_{ig}^a v_{gj}^b + c_i + d_j - r_{ij} \right)^2$$

We are training feature h, hence we are searching for a better vector $\underline{\Theta}_h$ to reduce the result of the error equation. After derivating the error equation with respect to $\underline{\Theta}_h$

$$\frac{\partial \sum_{ij}\varepsilon_{ij}^2}{\partial \underline{\Theta}_h} = \underline{0}$$

we achieve the update equations

$$\underline{\Theta}'_h = \underline{\Theta}_h + \varphi 2 \sum_{(u_i, m_j, r_{ij}) \in T \backslash V} \varepsilon_{ij} \underline{\vartheta}_{ijh}$$

and φ is the learning rate for the factors. $\underline{\vartheta}$ stays the same with

$$\underline{\vartheta}_{ijh} = \left(s_{ih}^1 v_{hj}^1, s_{ih}^1 v_{hj}^2, ..., s_{ih}^1 v_{hj}^r, s_{ih}^2 v_{hj}^1, ..., ..., s_{ih}^r v_{hj}^r \right)^T$$

To supress overtraining we once again add regularization parameters to the update equation

$$\underline{\Theta}'_h = \underline{\Theta}_h + \varphi \left(\left(2 \sum_{(u_i, m_j, r_{ij}) \in T \backslash V} \varepsilon_{ij} \underline{\vartheta}_{ijh} \right) - \varsigma \underline{\Theta}_h \right)$$

with ς being the parameter. It should be arranged higher than the other two regularization parameters, since training is only done once.

The starting value of each of the coefficients is a vital part of the algorithm and will ensure a reasonable performance. We suggest not to initialize the coefficients to any negative values because that might cause the effect that the different summands outweight each other. In our implementation we initialize the coefficients randomly in the range of $[1.5; 3.5]$.

Performance of this variation of the base algorithm is generally weak and not as good as the methods not using any internal polynomial kernel. However, this method might be used to explain more complex errors that a linear method (the basic approach) was not able to cover. Therefore we do only use this predictor as an addition to another better performing predictor, like a MF without an internal polynomial kernel.

3.2.5. Parameter Setup

There are several parameters to set up and fine-tune until the learning algorithm will work fine. Those parameters are the different learning rates α, χ, φ, the degree of the internal polynomial kernel r, the different regularization

factors β, δ, γ and also the number of features f.

We initialize all features in S and V to 0.1 and the constant vectors $\underline{c}$ and $\underline{d}$ to 0.9 for all elements.

As for the learning rates it depends on how fast one wants to train an estimator. Higher learning rates mean faster training but less final performance. The regularization parameters should be set in dependence to the number of features. For a good performance we suggest 96 to 164 features with regularization parameters between 1×10^{-2} and 4×10^{-2} and a learning rate between 1×10^{-4} and 3×10^{-4} for the non-constant feature matrices M and U and between 5×10^{-5} and 15×10^{-4} for the constant feature vectors $\underline{c}$ and $\underline{d}$.

Initialization parameters for polynomial kernels are hard to find and should be chosen differently in every situation, since polynomial estimators are usually only used in connection with another, not so complex, estimator.

3.2.6. The final algorithm

The dataset of the Netflix Prize has some idiosyncrasies we need to take care of in form of changing the algorithm to perform well. One improvement is that we define a minimum number of training epochs per feature as well as a maximum number of training epochs. Those two limits are dynamically adjusted depending on the number of the feature that is trained. This alternation of the main algorithm is necessary because the algorithm would otherwise stop training just after the first few epochs. Especially for the later features it requires a lot of epochs to actually come to the point where the RMSE on the validation set improves again. It is nearly as if the algorithm would have to walk a certain way before it can approach the next local (or global) minimum.

Validation is done on a smaller validation set with roughly 100000 instances. This is enough to ensure that the RMSE is precise enough and furthermore training is not slowed down significantly. In case the minimum number of training epochs for this feature is crossed and the RMSE on the

validation set is still declining, training on this feature stops and the algo-rithm moves on to the next feature. The improvement of RMSE is measured within five steps, meaning that for the RMSE on the validation set (in our case V_{MF}) of the i'th feature and the j'th epoch, $RMSE_i^j$, the given RMSE difference is

$$\Delta_i^j = RMSE_i^{j-5} - RMSE_i^j$$

A minimum of five training epochs for each feature is required.

We can now formally define the training algorithm

- **Step 1:** Initialize the features matrices S and V for the first time, S and V to 0.1 and $\underline{c}$ and $\underline{d}$ to 0.9.

- **Step 2:** Loop until number of features f reached.

  - Loop until maximum number of epochs is reached or Δ_i^j is neg-ative and the number of minimum training epochs was already crossed.

    * Iterate over each element of $T \setminus V$ and update the feature vectors with each iteration.

    * Recompute the RMSE on the validation set V.

  - Cache the predicted values and clip the predictions to $[1; 5]$.

- **Step 3:** Compute the final RMSE and save training and validation set.

We use a minimum number of training epochs of $75 + f$ and f is the actual feature number. The maximum threshold is $85 + 2f$.

3.3. Matrix Factorization by Batch Learning

A disadvantage of the summand by summand minimization is that it depends on the order of the training set. The algorithm will behave differently on two training sets that contain the same data but are ordered differently. To ap-proach this problem we suggest a variation of the base algorithm, which does

not minimize the overall error by minimizing each summand but minimizes the error each user and each movie causes. This can easily be accomplished by changing the update equations to

$$s'_{ik} = s_{ik} + \alpha \left(\left(2 \sum_{u_j \in U_i} \left(\left(r_{ij} - \sum_{a=1}^{n} s_{ia} v_{aj} \right) v_{kj} \right) \right) - \beta s_{ik} \right)$$

and

$$v'_{kj} = v_{kj} + \alpha \left(\left(2 \sum_{m_i \in M_j} \left(\left(r_{ij} - \sum_{a=1}^{n} s_{ia} v_{aj} \right) s_{ik} \right) \right) - \beta v_{kj} \right)$$

and U_i is the set that holds all users that rated the movie m_i in the training set

$$\forall \left(u, m_i, e\left(u, m_i\right) \right) \in T \Leftrightarrow u \in U_i$$

M_j is the set that holds all movies that were rated by a certain user u_j

$$\forall \left(u_j, m, e\left(u_j, m\right) \right) \in T \Leftrightarrow m \in M_j$$

This approach is actually the true solution to the minimization problem of section 2.2. While the proposed base algorithm in section 3.2.1 is minimizing each summand of the RMSE equation independently, the variation described here minimizes the whole RMSE. This is also known as the method of the steepest descent. For a fixed feature and a known epoch (iteration) of training k we can achieve the update equation for all features given by

$$\underline{\lambda}^{k+1} = \underline{\lambda}^k + \alpha \underline{\eta}^k$$

where $\underline{\eta}$ is the directional vector. Obviously the steepest directional vector is the antigradient of any $\sum_{ij} \hat{e}\left(u_i, m_j, \underline{\lambda}^k\right)$

$$\underline{\eta}^k = -\nabla \left(\sum_{ij} \hat{e}\left(u_i, m_j, \underline{\lambda}^k\right) \right)$$

A disadvantage of this implementation is that it does not perform as well as the incremental (summand by summand minimization) algorithm. This is because the incremental implementation might update the items of a feature vector at once and reuse it in the next situation the user or movie reappears. Also computing time doesnt improve (depending on the programming language) because we need to add new arrays to the program, which save the cached errors per user or per movie.

3.3.1. The Armijo rule

A valuable idea might be to speed up training by using dynamic learning rates α^k. The Armijo rule provides a restriction for the learning rate in each epoch of training a feature. In the case of our minimization problem a learning rate α^k satisfies the Armijo rule in case it satisfies the following inequation

$$\sum_{ij} \hat{e}\left(u_i, m_j, \underline{\lambda}^k + \alpha^k \underline{\eta}^k\right) \leq \sum_{ij} \hat{e}\left(u_i, m_j, \underline{\lambda}^k\right)$$

$$+ \kappa \alpha^k \left(\underline{\eta}^k\right)^T \nabla \left(\sum_{ij} \hat{e}\left(u_i, m_j, \underline{\lambda}^k\right)\right)$$

with $\kappa \in (0; 1)$. The choice for the directional vector $\underline{\eta}^k$ is left open. In case it is the antigradient, the product $\left(\underline{\eta}^k\right)^T \nabla \left(\sum_{ij} \hat{e}\left(u_i, m_j, \underline{\lambda}^k\right)\right)$ yields

$$\left(\underline{\eta}^k\right)^T \nabla \left(\sum_{ij} \hat{e}\left(u_i, m_j, \underline{\lambda}^k\right)\right) =$$

$$\nabla \left(\sum_{ij} \hat{e}\left(u_i, m_j, \underline{\lambda}^k\right)\right)^T \nabla \left(\sum_{ij} \hat{e}\left(u_i, m_j, \underline{\lambda}^k\right)\right)$$

Here also lies the problem of the adaptation of the Armijo rule for our minimization problem. The above product is usually very large, since it is nothing more than the sum of all elements of the gradient multiplied with itself. That

means that for the training of the first feature we will require a totally different κ than for the training of the second feature. Furthermore even a κ for the first feature is hard to find and requires a lot of fine-tuning. It also depends on the values the feature matrices are initialized to.

We were not able to implement a stable version of the algorithm that makes use of the Armijo rule. However, we state the approach here because we believe that a detailed study of the problem might offer a lot of possibilities. A well functioning Armijo rule will ensure a good selection of the learning rate and therefore offer a fast convergence of the algorithm itself. This would be especially helpful in a productive environment, where training speed is important to ensure that the prediction algorithm is always representing a great deal of the actual user base.

3.3.2. Parameter Setup

The adaptations of all other variations of incremental learning we presented earlier for batch learning is straight forward and therefore not discussed here.

Our testing and implementation performs relatively weak and not as good as the incremental approach (see chapter 6). However, the methods still offers opportunities. Next to the considerably weak performance the algorithm is also hard to stabilize, especially together with the usage of the Armijo rule. Furthermore the learning rate greatly depends on the way the feature matrices and vectors are initialized.

We suggest to use a dynamic learning rate that is adjusted every feature. A possibly way to do so is to define a base learning rate and multiply it with the square root of the number of feature that is trained. This will ensure that the learning rate is not too high at the beginning of the algorithm, so that it does not destabilize the whole process, and furthermore it will ensure that the learning rate is not too low with higher feature numbers, which would cause the algorithm to stop training. For an actual feature f we define the learning rate α as

$$\alpha = \alpha_B f^{\frac{1}{h}}$$

and α_B is the base learning rate. This value should be situated around 5×10^{-6}. For $h = 2$ we will achieve the square root that was mentioned above.

The algorithmic setup is actually nearly the same as with the incremental learning. It might be useful to set the maximum number of epochs per feature to a high number or define an unlimited upper threshold. This is especially useful in case one does not use any regularization parameters, such as $\beta = 0$. We use an unlimited upper limit of iterations and a lower limit of $650f$ and f is the actual feature.

4. Nearest Neighbor Search

The principle of a Nearest Neighbor (NN) search is often used to approach and solve Data Mining problems. The underlying idea of such a search is a similarity function, which defines similarity for the objects to classify. Since in our setup we do not classify but predict ratings, we will use the similarity function to identify similar objects in our system and from those objects we will try to predict the ratings. There are two possibilities on what to base the similarity function on, either on the users or on the movies. Since the number of users is considerably high and therefore the average number of ratings an user made is low, we do not consider users for the similarity function. Furthermore using the users as the base of the similarity function would lead to the problem of implementing a similarity matrix with a height and width of rougly half a million. This is just not practical with todays computer systems. The implementation presented here focuses on similarities among movies.

4.1. Movie Similarity

The question when two movies are considered similar and how similar those two movies actually are can be answered in many different ways. A lot of competitors of the Netflix Prize use the Pearson Correlation coefficient or the cosine similarity to measure similarity between movies. Those similarity measures include the ratings: If one user rates two movies similar or closely similar, the movies are considered equal.

In this implementation of a NN search we do not use the integer ratings for our similarity measure. We do not believe that two movies are equal only

because users rated them the same. A user might like and dislike two movies and still they might be the same. This is often true with sequells. While the first part is often very good and encourages people to view the second part, this one is actually not as good. People will be disappointed and rate the sequel worse than the first part. A rating-based similarity would fail in this situation.

For those obvious reasons we introduce a similarity between two movies that is only based on the fact how many people of all the people that saw the movies actually saw both movies. For two movies $m_1, m_2 \in M$ we define the sets $U_1, U_2 \subseteq U$ that hold the users that watched movie m_1 (set U_1) and movie m_2 (set U_2).

$$\forall\, (u, m_i, e\,(u, m_i)) \in T \Rightarrow u \in U_i$$

The similarity function κ is now defined as

$$\kappa : M \times M \rightarrow \mathbb{R}^+$$

$$\kappa\,(m_1, m_2) = \frac{card\,(U_1 \cap U_2)}{card\,(U_1 \cup U_2)}$$

Such a similarity function is often called a Jaccard similarity function. All movie similarities are saved in the similarity matrix S_κ.

$$S_\kappa = (s_{ij} = \kappa\,(m_i, m_j))_{i,j=1,\ldots,q}$$

This matrix S is symmetric, $S_\kappa = S_\kappa{}^T$, since κ is commutative

$$\kappa\,(m_i, m_j) = \kappa\,(m_j, m_i)$$

S_κ has $q^2 = 315772900$ cells. Considering a 32-bit float for implementation this matrix will require a total of $32q^2/(8 \times 10^9) \approx 1.26$ GB of memory. By only saving one triangle of the matrix we can slice this in half but accessing the elements might also be more complicated and require more time, depending on the programming language that is used for implementation.

Table 4.1.: Five Nearest Neighbors of selected movies

Lord of the Rings II	A Beautiful Mind	Pretty Woman
Lord of the Rings I	Good Will Hunting	Sweet Home Alabama
Lord of the Rings III	Silence of the Lambs	Dirty Dancing
Pirates of the Cari. I	Ocean's Eleven	Miss Congeniality
The Matrix	Catch Me If You Can	The Wedding Planner
Spider-Man	Shawshank Red.	Independence Day
Toy Story	**24: Season 1**	**WWII in Color**
Aladdin	24: Season 2	The Last Days of WWII
The Lion King	24: Season 3	Fighting Rats of Tobruk
Shrek	Alias: Season 1	Raiders of Leyte Gulf
Lord of the Rings I	The Sopranos: S2	City of Steel: Carrier
A Bug's Life	The Sopranos: S1	WWII: L.C. Archives
Justin Timberlake Live	**ST: Nemesis**	**ER: Season 3**
'N Sync Live I	ST: Insurrection	ER: Season 2
'N Sync Live II	ST: First Contact	ER: Season 1
Janet Jackson D.	ST II: ...	Will & Grace: Season 4
'N Sync Live III	ST V: ...	Law & Order: Season 3
Backstreet Boys Live	ST VI: ...	The Cosby Show: S1

Table 4.1 gives some example movies and their five nearest neighbors. As can be seen the similarity measure works well but is not free of error. Obviously there are some media that often occur as neighbors, such as the Lord of the Rings movies or Ocean's Eleven. These are most of the time the movies that are very popular and got rated very often in respect to the time how long they were available in the system. Those movies of course add a bias to the rating estimators we are going to explain in the next section.

In our implementation we only save the n most similar movies for each of the 17770 movies. n usually is $250 \leq n \leq 1000$. This has the advantage that the memory consumption stays low at $32qn/(8 \times 10^9)$ GB and therefore the implementation offers faster access times. Also, the first n movies often offer enough possible neighbors to feed a prediction algorithm. Therefore it is not required to save the others, which will result in smaller matrices (arrays) and faster access time. In our experiments we figured out that any n higher than 750 will not significantly improve the RMSE of the algorithm we are going

to explain in the next section.

4.2. Predicting single ratings

We define N_j^k as the set of the k nearest neighbors of movie m_j

$$N_j^k \subseteq M$$

and once again the set U_i as all the users that watched a movie m_i

$$U_i \subseteq U$$

Now the prediction for a rating an user u gave to a movie m_j is given by

$$\hat{e}\left(u_i, m_j, \underline{\lambda}\right) = \frac{\sum\limits_{m \in N_j^k \wedge u_i \in U_j} \omega\left(\kappa\left(m_j, m\right)\right) e\left(u_i, m\right)}{\sum\limits_{m \in N_j^k \wedge u_i \in U_j} \omega\left(\kappa\left(m_j, m\right)\right)}$$

For each movie in the set of the k most similar movies of movie m_j that were watched by user u_i we multiply the rating with the similarity of both movies. Finally devide the sum by the sum of all similarities. ω is a weighting function for the similarities and might for example be given by

$$\omega_r\left(s\right) = s^r$$

with $r \in \mathbb{R}^+$. For a high r higher similarities will be weighted more important than lower similarities. $\underline{\lambda}$ is again a vector of paramters. We can also imagine other weighting functions ω. One idea might be to involve the exponential function $\omega\left(s\right) = \exp\left(s\right)$ or to just use the unweighted ratings $\omega\left(s\right) = 1$. For the here described estimator $\underline{\lambda}$ yields

$$\underline{\lambda} = \left(\begin{array}{cc} n & k \end{array}\right)^T$$

For reasons of computational and prediction performance we limit the number of neighbors to a maximal number k. This means that once k neighbors were found and their similarities contributed to the estimator $\hat{e}$ learning is stopped. As already mentioned we also limit the number of similar movies per given movie to a lower number than the overall number of movies, by only using the n most similar movies to a given one. The reason for this is that in this way we do not have to save the complete similarity matrix, which would require a lot of storage space and walking all 17770 neighbors would also be very time consuming.

The advantage of any Nearest Neighbor method is that it does not require any learning. From a given similarity matrix and a rating matrix we can already compute predictions for unknown user/movie pairs. Furthermore the computation of such predictions is, if implemented in the right way, very fast. A disadvantage of this method is that in comparison to the MF methods it does not perform as well. However, in an environment where there is no need for peak performance but peak efficiency a Nearest Neighbor method can be the right choice. Furthermore it is possible to combine different predictions from different models, such as Nearest Neighbor searches and MF techniques.

4.3. Data Normalization

To improve the prediction performance it is wise to normalize the data before applying any Nearest Neighbor method. To do so it is necessary to find a way to bring the different ratings of the neighbors nearer to each other. This can, for example, be done by subtracting the mean rating of every movie m, μ_m, from the rating and add it to the final prediction again. The estimator changes to

$$\hat{e}\left(u_i, m_j, \underline{\lambda}\right) = \frac{\displaystyle\sum_{m \in N_j^k \wedge u_i \in U_j} \omega\left(\kappa\left(m_j, m\right)\right)\left(e\left(u_i, m\right) - \mu_m\right)}{\displaystyle\sum_{m \in N_j^k \wedge u_i \in U_j} \omega\left(\kappa\left(m_j, m\right)\right)} + \mu_{m_j}$$

Another idea is to use an estimator based on Matrix Factorization to roughly predict the ratings and then let the Nearest Neighbor method try to predict the residuals. For more information on the performance of the normalized and unnormalized methods the reader is refered to Chapter 6.

4.4. Hierarchical Clustering on the Similarity Matrix

Our goal in this step is to define a clustering based on the similarities of the movies. This will help us to create a so called *nested model*, which makes use of two or more estimator by nesting them. One could, for instance, cluster the movies and therefore the training set T and use these training sets to train different MF estimators.

From all the obvious clustering techniques [29] [5] [20] [11], here we are going to use hierarchical clustering. [48] [27]. We will only outline the methods used. For a detailed explanation the reader is directed to the cited publications. Agglomerative hierarchical clustering, which is used here, also known as the "bottom-up" approach forms clusters by starting with the single objects. At the beginning each object represents its own cluster. With the help of some distance function the most similar pair of clusters is found and merged. This algorithm is repeated until there is only one cluster left.

To compute the similarity of clusters hierachical clustering makes use of a distance function, which measures the dissimilarity of two objects (either single objects or objects in a cluster). To transform our similarity function $\kappa\left(m_1, m_2\right)$ we only need to subtract it from one, which will yield the Jaccard distance function $\pi\left(m_1, m_2\right)$

$$\pi\left(m_1, m_2\right) = 1 - \kappa\left(m_1, m_2\right) = \frac{card\left(U_1 \cup U_2\right) - card\left(U_1 \cap U_2\right)}{card\left(U_1 \cup U_2\right)}$$

and

$$S_\pi = \left(s_{ij} = \pi\left(m_i, m_j\right)\right)_{i,j=1,...,q}$$

There are several functions ζ available to measure similarity between clusters, these functions are also called linkage functions. Two of the most simple functions are the single linkage function, also known as the nearest-neighbor function, and the complete linkage function, which is also known as the furthest neighbor. Both define the distance of two clusters as the distance of two single items. The single linkage function defines the distance of two clusters as the distance of their two nearest objects, the complete linkage function as the distance between their two furthest objects. Let $M_1, M_2 \subset M$ be two clusters of movies with $M_1 \cap M_2 = \emptyset$.

$$\zeta_{single}\left(M_1, M_2\right) = \min\left(Q\right) \qquad \zeta_{complete}\left(M_1, M_2\right) = \max\left(Q\right)$$

and Q is a set consisting of all the movie distance of the movies in M_1 and M_2.

$$Q = \left\{\pi\left(m_i, m_j\right) \middle| \forall m_i \in M_1, \forall m_j \in M_2\right\}$$

Also other similarity measures are possible for those two clustering algorithms, such as the already mentioned Pearson Similarity.

Next to those two linkage functions there are also the average and centroid linkage, which make us of the entire collection of objects within a cluster. The average linkage function computes the average distance of all objects within two clusters.

$$\zeta_{average}\left(M_1, M_2\right) = \frac{\sum\limits_{q \in Q} q}{card\left(Q\right)}$$

The centroid linkage function returns the euclidian distance of the centroids of two clusters.

$$\zeta_{centroid}\left(M_1, M_2\right) = \zeta_{cen}\left(M_1, M_2\right) = \left\|\overline{M_1} - \overline{M_2}\right\|$$

It is not mandatory to define what the centroid of a movie cluster is. [29] presents an update equation that can easily be used to update the distance between two clusters. Consider two clusters R and U. At an earlier step R was formed by the merger of the clusters A and B. Now the centroid

distance between R and U is given by

$$\zeta_{cen}\left(R,U\right) = \left(\frac{|A|}{|R|}\zeta_{cen}^2\left(A,U\right) + \frac{|B|}{|R|}\zeta_{cen}^2\left(B,U\right) - \frac{|A|\,|B|}{|R|^2}\zeta_{cen}^2\left(A,B\right)\right)^{\frac{1}{2}}$$

and $|A| = card\left(A\right)$. We start of with clusters of size one (the single movies) and for those items the initial distances are given by the above defined distance matrix S_π. The proof for the update equation can be found in [29]. [29] also states that it is possible to apply the update equation to any kind of dissimilarity measures, with or without squares. But as [55] and [5] state this may lead to strange results and is therefore not recommended. The cluster trees so produced may also be non-monotonic. This occurs when the distance from the union of two clusters, M_1 and M_2, to a third cluster is less than the distance from either M_1 or M_2 to that third cluster.

Hierarchical clustering stops when a predetermined number of clusters was formed.

4.4.1. Clusterings in Praxis

For the dissimilarity matrix proposed above only the complete linkage results in a good clustering. Neither the single linkage nor the clustering involing the average linkage function result in roughly equally sized clusters.

There is an obvious pattern evolving when using hierarchical clustering on the above described dissimilarity matrix. Each setup usually results in a very big cluster with the majority of movies and at least one small cluster. We were not able to find a setup that allowed us to create several roughly equally sized clusters.

When a hierarchical clustering with complete linkage is applied to the dissimilarity matrix S_π and two clusters are created, then cluster one carries 133 movies and the other cluster carries the other 17637 movies. The first cluster however, is a very nice cluster, since it carries mainly "Wrestling" - DVDs or some media regarding the TV series "Doctor Who". While it is surprising that actually these two issues get mixed together it is certainly

also surprising that the hierarchical clustering was able to sort these media out.

4.4.2. A Proposal: Iterative Hierarchical Clustering

At the end of the writing of the thesis the author discovered an interesting feature regarding the hierarchical clustering on S_π with complete linkage. As already mentioned such a clustering using two clusters would result in one cluster $A_1 \subset M$ of the size of 133 movies and another cluster $A_2 \subset M$ of the size of 17637 movies. It is true that $A_1 \cup A_2 = M$ and $A_1 \cap A_2 = \emptyset$.

Now an interesting discovery is that when hierarchical clustering is applied to the set A_2 it can fall into two other clusters again. This of course does require the removal of all elements of A_1 from the similarity matrix. The two resulting clusters are called $B1$ and $B2$ with $B_1, B_2 \subset A_2$ and $A_1 \cup B_1 \cup B_2 = M$ and $A_1 \cap B_1 \cap B_2 = \emptyset$. Cluster B_1 is of size 4625 and cluster B_2 of size 13012. The interesting thing here is that cluster B_1 covers 87.209% of the dataset, while cluster B_2 covers only 12.7265%. This is interesting, since cluster B_1 carries a lot less movies than B_2 but covers a lot more training instances of $T \setminus V$.

Such an iterative hierarchical clustering, as we call it, can be carried on as long as required or there are no valueable clusters left. Figure 4.1 gives an example of how such a clustering may look like in tree form. Each node in this tree is equipped with some information: The name of the cluster (A_1, A_2, B_1, ...), the number of movies in this cluster and the percental amount of how many training instances of the training set $T \setminus V$ each cluster covers.

An interesting observation is for example that the cluster B_1 only consists of 4625 of all the 17770 movies but covers more than 87% of the training set. These are the movies that get rated very often. Those movies are treated similar, since in our implementation two movies are similar if they both were watched by many users. Of course a disadvantage of this implementation is that the very popular movies always seem similar to each other. Iterative hierarchical clustering is obviously able to localize these movies.
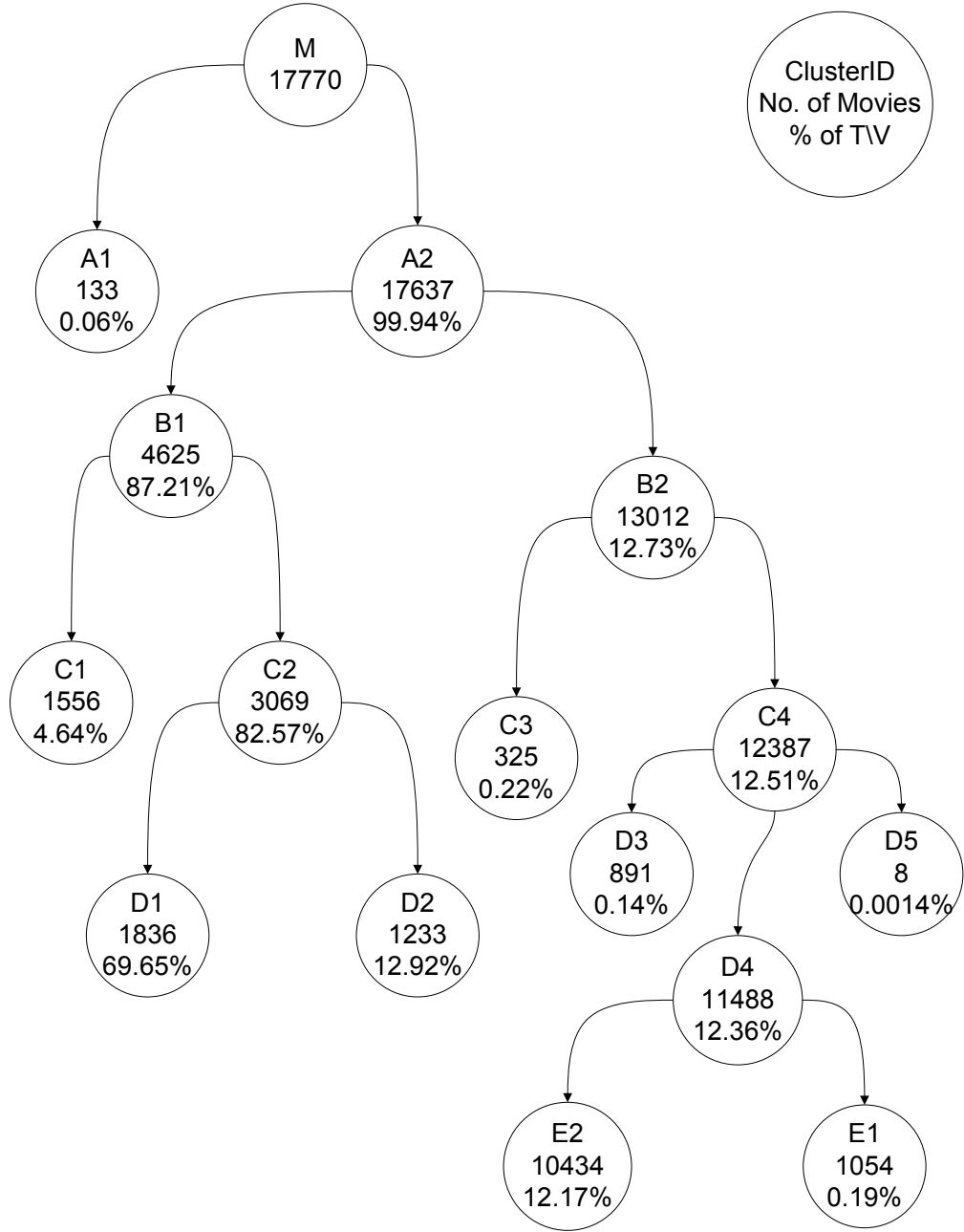
Figure 4.1.: An example tree for iterative hierarchical clustering using mostly a number of two clusters per split as well as the complete linkage function on the similarity matrix S_π.

5. Combining Predictions

The reason why to combine different predictions to a final one is simple:
To reach a lower RMSE on the validation set. There are several ways to
combine different predictions. Here we want to present three methods. The
first method is based on the idea of "blending" different predictions to a
new one by solving a least square optimization problem. The second way
involves a consecutive way of predicting different ratings. Several methods
are combined to one by connecting them in series and each method tries
to predict the residuals the predictors before it could not cover. As another
option it is possible to combine those two combinational methods by blending
several concatenated and blended models into a new one. Furthermore we
shortly outline the idea behind a new type of models, called nested models.
These estimators combine two or more estimators by somehow incorporating
them.

All in all the ways of improving the RMSE presented here are computa-
tional very expensive. The amount of time required to train and implement
the different estimators might outweight the improvement of the RMSE. The
here presented options do specifically target the task of the Netflix Prize but
not the usage in a productive system.

5.1. Blending Models

For a given set of predictors $E = \{\hat{e}_1, \hat{e}_2, ..., \hat{e}_w\}$ we achieve the following final
estimator $\hat{e}$.

$$\hat{e}\left(u_i, m_j, \underline{\lambda}\right) = \sum_{k=1}^{w} \psi_k \hat{e}_k\left(u_i, m_j, \underline{\lambda}_k\right)$$

with

$$\underline{\lambda} = \left(\begin{array}{cccc} \psi_1 & \psi_2 & \cdots & \psi_w \end{array} \right)^T$$

and λ_k as the parameters of each estimator of E. We can calculate the optimal $\underline{\lambda}$ by minimizing the quadratic error $\hat{e}$ produces on the validation set.

$$\sum_{(u_i, m_j, r_{ij}) \in V} \left(\hat{e}\left(u_i, m_j, \underline{\lambda}\right) - e\left(u_i, m_j\right) \right)^2 \to \min_{\underline{\lambda}}$$

Following this approach by carrying out the derivation with respect to the different weights we achieve a system of linear equations given by

$$\Xi \underline{\lambda} = \underline{\tau}$$

We furthermore define

$$\underline{\xi} = \left(\begin{array}{cccc} \hat{e}_1 & \hat{e}_2 & \cdots & \hat{e}_w \end{array} \right)$$

and achieve the coefficient matrix of this linear system of equations

$$\Xi = \sum_{(u_i, m_j, r_{ij}) \in V} \underline{\xi}^T \underline{\xi}$$

while the vector of constants is given by

$$\underline{\tau} = \sum_{(u_i, m_j, r_{ij}) \in V} \underline{\xi}^T e\left(u_i, m_j\right)$$

We can solve this system with the help of any solving algorithm for linear systems of equations or by carrying out the direct matrix equation

$$\underline{\psi} = \left(\Xi^T \Xi \right)^{-1} \Xi^T \underline{\tau}$$

The resulting weights $\psi_1, \psi_2, ..., \psi_w$ minimize the above objective equation and solve the least square problem. Other sources [8] [39] proposed blending models using a linear regression fit. A similar approach is to use one estimator

that just predicts every rating the same. For example such an estimator might give back the global mean rating, $\hat{e}\left(u_i, m_j, \underline{0}\right) = 3.6066$.

In our implementation we compute the optimal weights on one part of the validation set V and check the performance improvement on the other part. In this way we can ensure that the least square solution is not just overtraining on the validation set.

5.2. Concatenation of models

Two models are concatenated by having the second model predict the residuals of the first one. This idea is also applicable to an unlimited number of models w

$$\hat{e}\left(u_i, m_j, \underline{\lambda}\right) = \sum_{k=1}^{w} \hat{e}_k\left(u_i, m_j, \underline{\lambda}_k\right)$$

where each model covers the residuals the models before it produced. Therefore the rating matrix $\hat{R}$ for the ℓ'th model $(2 \le \ell \le w)$ is

$$\hat{R}_\ell = \left(\hat{r}_{ij} - \sum_{k=1}^{\ell-1} \hat{e}_k\left(u_i, m_j, \underline{\lambda}_k\right)\right)_{i=1,\ldots,p,\, j=1,\ldots,q}$$

Experiments show that some estimators are not able to cover to residuals of other estimators. That results in a RMSE that is worse than the RMSE of the model before. An example might be a Nearest Neighbor method that is not able to improve the RMSE on a well trained MF technique. On the other hand more complex models can improve the prediction performance slightly, even if alone they are not able to give a good result.

5.3. Nested models

Nested models are models that in some way include other models and use this inclusion to enhance the prediction performance or improve productivity. In this thesis we will experiment only on one kind of nested models. These

are the models that split the dataset into clusters based on the dissimilarity matrix described in Chapter 4.

5.3.1. Using subsets of the training set

Any clustering algorithm will enable us to create subsets of the training set. We can imagine a hieracical clustering on the similarity matrix of the movies creating three clusters of movies. We can apply this clustering to the training set and train different MF estimators or even KNN estimators on each subset of the training set. This might be especially useful for MF techniques, since they could profit from a smaller training base as long as the training base is self-contained. Smaller training sets mean a shorter training time. Of course this goes in most of the times not without a increasement of the overall error but a shorter training time is often more important than the lowest error, especially in a productive system where each minute of training time might equivalent to a lost order from the system.

5.3.2. Other nested models

Of course there are other nested models one could think of. Here we are going to present two interesting ideas, which are worth investigating.

We can think of a MF estimator that includes the features of the k most similar movies of the movie to predict a rating for.

$$\hat{e}\left(u_i, m_j, \underline{\lambda}\right) = \left(\sum_{a=1}^{n} s_{ia} v_{aj} + c_i + d_j\right) + \left(\sum_{m_b \in N_k} \sum_{a=w}^{n} s_{ia} v_{ab}\right)$$

and $N_k \subset M$ are the k nearest neighbors of movie m_j. n is either the actual trained feature or the number of maximum features. w decides which features should be included in training. If n is the actual feature that is being trained and $w = n$ only the actual features of the nearest neighbors are included in the estimator. We might also think of a weighted sum of those two single predictors.

Another approach is to use the feature matrices created by a MF to create clusters of movies or users. The feature matrices of the MF techniques, namely S and V describe the movies and the users. They can be thought of as a profile for those users and movies. By defining any similarity measure or dissimilarity measure [52] (such as the euclidian distance) we are able to compute a similarity matrix for those profiles. Those similarity matrices can again be used to feed a Nearest Neighbor algorithm.

Furthermore it is possible to use unsupervised clustering algorithms [37] [13] [26] [16], such as k-means, to cluster the movies and users into different groups. This has one advantage in comparison to the hierachical clustering based on distance matrices: Using for example the k-means algorithm will also enable us to cluster the users and not only the movies, while with any hierachical agorithm based on the similarity matrix we would easily run into storage problems when computing the similarity matrix for half a million users.

5. Combining Predictions

6. Experiments

In this chapter we are going to present the performance of the different prediction methods. To ensure a valid and precise measurement we remove a set of roughly 3 million ratings from the given training set of the netflix prize. Of this set of 3 million rating instances we shaped off another 90463 ratings. Those were used for validation of the MF techniques and to determine when to stop training a feature (set V_{MF}). The total size of the validation set V therefore reduces to 2924999. The precise size of the training set T is 97465045.

To determine the performance of the blending and concatanation methods we split the validation set again into two roughly equally sized parts. To measure the performance of combined models we require the first part for learning and the second for validation once again. Otherwise we would train on the same set as we would learn on. That of course leads to overtraining. The two validation sets are called V_1 and V_2 with $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$. For all the combined models we are going to train on set V_2 and validate on set V_1. For all other prediction models we use the complete set V for validation.

6.1. Naive Methods

We are going to give the RMSE of some naive prediction methods, including setting all predictions to the global mean 3.6066. Table 6.1 shows the performance of those naive methods on the validation set. We use the following notation: μ_{m_j} indicates the mean rating of a movie m_j and μ_{u_i} indicates the mean rating of a user u_i.

It is obvious that the naive approaches do not perform very well, especially

$\hat{e}\,(u_i, m_j, \underline{\lambda}) =$	RMSE on V
$\mu = 3.6033$	1.0843
μ_{m_j}	1.0105
μ_{u_i}	1.1750
$\left(\mu_{m_j} + \mu_{u_i}\right)\big/2$	1.0540

Table 6.1.: RMSEs for incremental MF, maximum iterations $85 + 2f$, minimum iterations $75 + f$

MF preferences	$f = 32$	$f = 64$	$f = 96$	$f = 128$	$f = 164$
$\alpha = 2.5 \times 10^{-4}$ $\chi = 10^{-4}, \beta, \delta = 0.015$	0.8168	0.8105	0.8081	0.8070	0.8072
$\alpha = 2.5 \times 10^{-4}$ $\chi = 10^{-4}, \beta\delta = 0.02$	0.8170	0.8095	0.8068	0.8053	0.8043
$\alpha = 2.5 \times 10^{-4}$ $\chi = 10^{-4}, \beta\delta = 0.0225$	0.8181	0.8104	0.8075	0.8059	0.8049
$\alpha = 2.5 \times 10^{-4}$ $\chi = 10^{-4}, \beta, \delta = 0.025$	0.8186	0.8113	0.80910	0.8069	0.8054
$\alpha = 2.5 \times 10^{-4}$ $\chi = 10^{-4}, \beta, \delta = 0.03$	0.8215	0.8152	0.8137	0.8116	0.8091

compared to our proposed methods.

6.2. Matrix Factorization

Table 6.1 gives the RMSE of some examples of implementations of the MF algorithms. The setup used for all instances of incremental learning consists of a fixed learning rate of $\alpha = 2.5 \times 10^{-4}$ for the feature matrices S and V and $\chi = 10^{-4}$ for the constant vectors $\underline{c}$ and $\underline{d}$. The regularization parameters for the matrices and vectors β, δ are given in the range of 0.015 and 0.03. A maximum of 164 features were trained. The maximum number of iterations was set to $85 + 2f$ while the minimum number was set to $75 + f$ and f is the number of the actual trained feature.

Figure 6.1 shows the declining error of the implementations on the validation set that is used to determine when training should stop. As can be seen
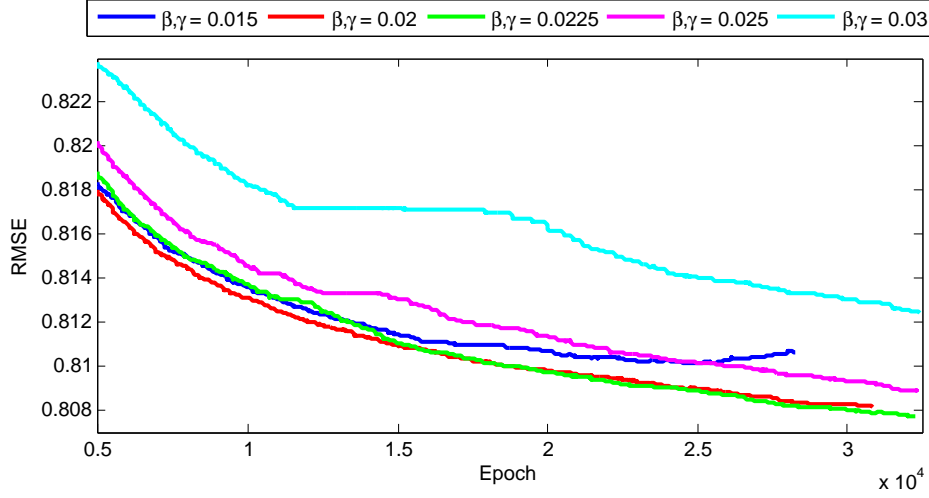
Figure 6.1.: Incremental MF performance on validation. Epochs from 5000 to 30000. Maximum of 164 features. Learning rates fixed at $\alpha = 0.00025$ and $\gamma = 0.0001$.

regularization parameters of $\beta, \delta = 0.0225$ perform the best. However, the graph also shows that with a higher number of features other regularization parameters, such as $\beta, \delta = 0.025$, could pull ahead. Furthermore it is obvious that the number of minimum training epochs per feature was set too low for the implementation using regularizations parameters of $\beta, \delta = 0.025$ and $\beta, \delta = 0.03$ since there is a horizontal line indicating that training did not improve at all during that time. This is due to the fact that the algorithm did not reach the point, in terms of training epochs per feature, where the RMSE on the validation set actually begins improving. With a higher number of minimum training iterations per feature the implementation using $\beta, \delta = 0.025$ might perform best.

Table 6.2 explains the RMSE for different ragularization parameters with batch learning. There is no upper threshold for the iterations, the lower one is $500 \times f$, where f is the actual trained feature. The learning rates were adjusted dynamically by using the formula $\alpha = \alpha_B f^{\frac{1}{h}}$ with $h = 4$. For 164 features the setup using the regularization parameters of $\beta, \delta = 5 \times 10^{-5}$

Table 6.2.: RMSEs for batch learning MF, maximum iterations ∞, minimum iterations $500 \times f$

MF preferences	$f = 1$	$f = 3$	$f = 5$	$f = 7$	$f = 9$
$\alpha_B = 10^{-5}, \chi_B = 5 \times 10^{-6}$ $\beta = 10^{-5}, \delta = 10^{-5}$	0.8837	0.8656	0.8541	0.8463	0.8413
$\alpha_B = 10^{-5}, \chi_B = 5 \times 10^{-6}$ $\beta = 3 \times 10^{-5}, \delta = 3 \times 10^{-5}$	0.8840	0.8655	0.8545	0.8461	0.8410
$\alpha_B = 10^{-5}, \chi_B = 5 \times 10^{-6}$ $\beta = 5 \times 10^{-5}, \delta = 5 \times 10^{-5}$	0.8839	0.8654	0.8543	0.8457	0.8408
$\alpha_B = 10^{-5}, \chi_B = 5 \times 10^{-6}$ $\beta = 7 \times 10^{-5}, \delta = 7 \times 10^{-5}$	0.8845	0.8658	0.8545	0.8469	0.8414
$\alpha_B = 10^{-5}, \chi_B = 5 \times 10^{-6}$ $\beta = 9 \times 10^{-5}, \delta = 9 \times 10^{-5}$	0.8851	0.8665	0.8548	0.8468	0.8415

yields the best results. Figure 6.2 shows the number training epochs plotted against the reached RMSE on the validation set of the MF.

Compared to incremental MF, MF via batch learning does not offer a nearly as good performance. It also does not offer any computational advantage in our implementation. Therefore we have to recommend the usage of incremental MF for the Netflix Prize.

6.3. Nearest Neighbor Methods

The NN methods do not perform as well as the MFs. However, due to their design they do not require any training and are therefore extremely convenient in a productive system. Table 6.3 gives the RMSE of a Nearest Neighbor method where each rating is normalized by the movie mean. Different weighting functions as well as a different number of neighbors were evaluated. A weighting function of $\omega(s) = s^{4.5}$ and either 20 or 25 neighbors offers the best performance. Figure 6.3 visualizes the context for weighting functions of the form s^e while Figure 6.4 visualizes the different RMSEs for weighting functions of the form b^s. We also tested the variation of the algorithms without any normalization of the ratings. The results can be
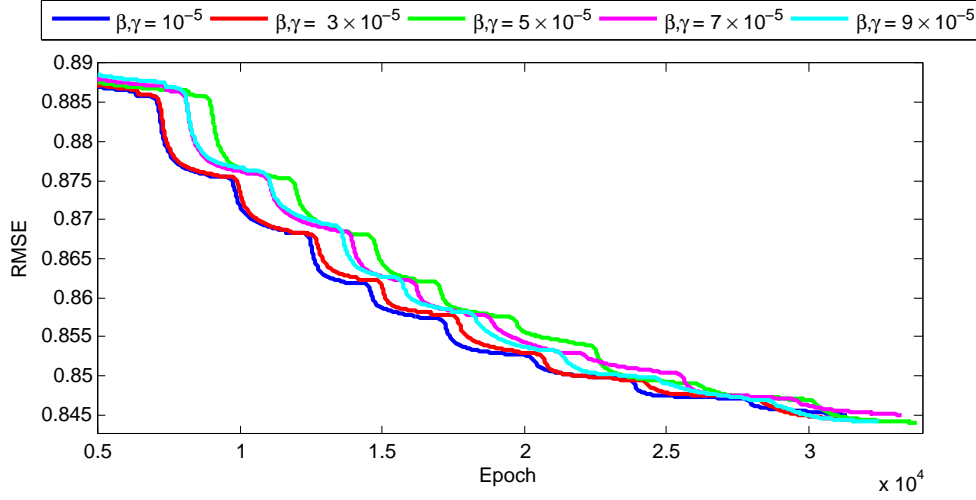
Figure 6.2.: Batch learning MF performance on validation. Epochs from 5000 to 30000. Maximum of 8 features. Learning rates dynamically with $\alpha_B = 10^{-5}$ and $\chi_B = 5 \times 10^{-6}$.

found in the appendix (B.2). The variations not using any normalization by movie mean are not performing as well as the NN implementations using the normalization by movie mean.

Nearest Neighbor methods, no matter if they use a normalization by movie mean or not, do not perform as well as incremental Matrix Factorizations. However, they do not require any training and if implemented in an efficient way and in a nicely performing programming language NN methods are also able to predict ratings in real-time.
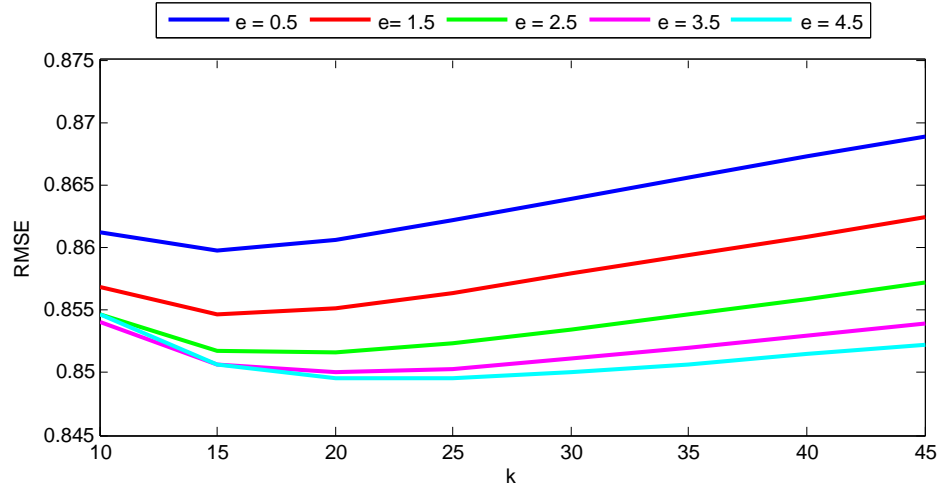
Figure 6.3.: RMSEs of Nearest Neighbor algorithm with different number of neighbors. Ratings normalized by movie means. Weighting functions of form $\omega_e(s) = s^e$.
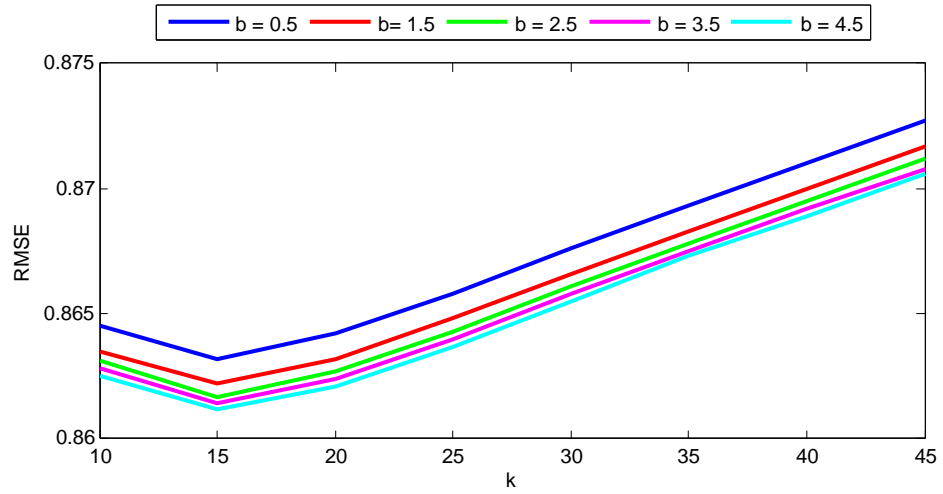
Figure 6.4.: RMSEs of Nearest Neighbor algorithm with different number of neighbors. Ratings normalized by movie means. Weighting functions of form $\omega_b(s) = b^s$.

Table 6.3.: RMSEs for different NN methods. Rating matrix substracted by movie means.

$\omega(s)$	$k=10$	$k=15$	$k=20$	$k=25$	$k=30$	$k=35$	$k=40$	$k=45$
$s^{0.5}$	0.8612	0.8598	0.8606	0.8622	0.8639	0.8656	0.8673	0.8689
$s^{1.0}$	0.8588	0.8570	0.8578	0.8592	0.8608	0.8624	0.8640	0.8656
$s^{1.5}$	0.8568	0.8547	0.8552	0.8564	0.8579	0.8594	0.8609	0.8624
$s^{2.0}$	0.8555	0.8530	0.8531	0.8542	0.8555	0.8568	0.8582	0.8595
$s^{2.5}$	0.8547	0.8517	0.8516	0.8524	0.8535	0.8547	0.8559	0.8572
$s^{3.0}$	0.8542	0.8500	0.8506	0.8511	0.8521	0.8531	0.8542	0.8553
$s^{3.5}$	0.8541	0.8506	0.8500	0.8503	0.8511	0.8520	0.8530	0.8539
$s^{4.0}$	0.8543	0.8505	0.8497	0.8499	0.8504	0.8512	0.8521	0.8529
$s^{4.5}$	0.8547	0.8506	0.8496	0.8496	0.8501	0.8507	0.8515	0.8522
$s^{5.0}$	0.8552	0.8510	0.8498	0.8497	0.8500	0.8505	0.8511	0.8519
0.5^{s}	0.8645	0.8632	0.8642	0.8658	0.8676	0.8693	0.8710	0.8727
1.0^{s}	0.8639	0.8626	0.8635	0.8652	0.8670	0.8687	0.8704	0.8720
1.5^{s}	0.8635	0.8622	0.8632	0.8648	0.8666	0.8683	0.8700	0.8717
2.0^{s}	0.8633	0.8619	0.8630	0.8645	0.8663	0.8680	0.8697	0.8714
2.5^{s}	0.8631	0.8617	0.8627	0.8643	0.8661	0.8678	0.8695	0.8712
3.0^{s}	0.8629	0.8615	0.8625	0.8641	0.8659	0.8677	0.8693	0.8710
3.5^{s}	0.8628	0.8614	0.8624	0.8640	0.8658	0.8675	0.8692	0.8708
4.0^{s}	0.8626	0.8613	0.8622	0.8639	0.8656	0.8674	0.8691	0.8707
4.5^{s}	0.8625	0.8612	0.8621	0.8637	0.8655	0.8673	0.8689	0.8706
5.0^{s}	0.8624	0.8611	0.8620	0.8636	0.8654	0.8672	0.8688	0.8705

6.4. Concatenation of Models

In this section we would like to present different experiments on the concatenation of models. In example 1 of Table 6.4 we concatenated three models. The first model ($\hat{e}_1$) is a MF via batch learning and yields a RMSE of 0.8408. The second model ($\hat{e}_2$) is a NN method without normalization by mean and gives an concatenated RMSE of 0.8209. The third and final model ($\hat{e}_3$) is a incremental MF with $f = 128$ features. We expected that this model might reduce the overall RMSE below the best one of a single incremental MF. However, it did not.

In Example 2 $\hat{e}_1$ and $\hat{e}_2$ are similar to the ones in Example 1. $\hat{e}_3$ is this time a MF with an internal polynomial kernel of $r = 2$ and $f = 48$ features. The polynomial factors $\underline{\Theta}$ were initialized randomly in the interval $[1; 4]$ and the MF used an unlimeted number of training iterations per feature as the maximum threshold and a minimum number of $75 + f$ and f is the actual trained feature.

Example 3 is quite similar to Example 2, only that the MF using an internal polynomial kernel of $r = 2$ was replaced with one using a degree of $r = 3$. Also the learning rates changes slighty. Both implementations do not improve the performance by an amount we would have expected. The problem with the non-linear MFs is that they are extremely hard to control, meaning it is difficult to find a good lower and upper threshold for the training iterations per feature. We experienced that they can very well improve performance significantly but only within one feature. As soon as the next feature starts training they either have too less minimum iterations, so they dont improve at all, or the number of minimum iterations is too high, so they improve for a while but then turn around and seriously increase the RMSE. For example 2 and 3 we found a setup that was not overtraining and improved the RMSE a little.

Example 4 concatenates a well-performing incremental MF using $f = 164$ features with an incremental MF ($\hat{e}_2$) of degree $r = 2$ as well as with an incremental MF ($\hat{e}_3$) of degree $r = 3$. While $\hat{e}_2$ is able to improve the RMSE

of $(\hat{e}_1)$ slightly, $(\hat{e}_3)$ is not able to improve it after all.

In Example 5 we let a NN method with normalization try to predict the residuals a NN method without normalization created. It is interesting that such a setup is not able to reach the RMSE of a single NN method with normalization. Also we tried to improve the performance further by concatenating the models with an incremental MF using $f = 128$ features. However, compared to a single MF or other setups (Example 1) the final RMSE was very bad. We believe that the preprocessing by the NN methods actually motivates the MF to iterate to a quite bad local minimum.

Table 6.4.: Different Concatanations

Example 1	$\hat{e}_1$: Batch MF with $f = 9$ and $\beta, \delta = 5 \times 10^{-5}$ $\hat{e}_2$: NN without norm. and $k = 35, \omega(s) = s^{4.5}$ $\hat{e}_3$: Incr. MF with $f = 128$ and $\beta, \delta = 0.0185$ and $\alpha = 1.75 \times 10^{-4}, \chi = 10^{-4}$			Overall
E	$\hat{e}_1$	$\hat{e}_2$	$\hat{e}_3$	
RMSEs	0.8408	0.8209	0.8062	0.8062
Example 2	$\hat{e}_1$: Batch MF with $f = 9$ and $\beta, \delta = 5 \times 10^{-5}$ $\hat{e}_2$: NN without norm. and $k = 35, \omega(s) = s^{4.5}$ $\hat{e}_3$: Incr. MF with $f = 48$, r=2, $\beta, \delta = 0.01$ and $\alpha = 2.5 \times 10^{-5}, \chi = 10^{-5}$			Overall
E	$\hat{e}_1$	$\hat{e}_2$	$\hat{e}_3$	
RMSEs	0.8408	0.8209	0.8163	0.8163
Example 3	$\hat{e}_1$: Batch MF with $f = 9$ and $\beta, \delta = 5 \times 10^{-5}$ $\hat{e}_2$: NN without norm. and $k = 35, \omega(s) = s^{4.5}$ $\hat{e}_3$: Incr. MF with $f = 48$, r=3, $\beta, \delta = 0.01$ and $\alpha = 10^{-5}, \chi = 5 \times 10^{-6}$			Overall
E	$\hat{e}_1$	$\hat{e}_2$	$\hat{e}_3$	
RMSEs	0.8408	0.8209	0.8188	0.8188
Example 4	$\hat{e}_1$: Incr. MF with $f = 164, r = 1$ and $\beta, \delta = 0.0225$ and $\alpha = 2.5 \times 10^{-4}, \chi = 10^{-4}$ $\hat{e}_2$: Incr. MF with $f = 16, r = 2$ and $\beta, \delta = 0.015$ and $\alpha = 1.75 \times 10^{-4}, \chi = 10^{-4}$ $\hat{e}_3$: Incr. MF with $f = 4, r = 3$ and $\beta, \delta = 0.01$ and $\alpha = 3.5 \times 10^{-5}, \chi = 1.5 \times 10^{-5}$			Overall
E	$\hat{e}_1$	$\hat{e}_2$	$\hat{e}_3$	
RMSEs	0.8049	0.8046	0.8046	0.8046
Example 5	$\hat{e}_1$: NN without norm. and $k = 30, \omega(s) = s^4$ $\hat{e}_2$: NN with norm. and $k = 30, \omega(s) = s^4$ $\hat{e}_3$: Incr. MF with $f = 128, r = 1$ and $\beta, \delta = 0.0225$ and $\alpha = 2.5 \times 10^{-4}, \chi = 10^{-4}$			Overall
E	$\hat{e}_1$	$\hat{e}_2$	$\hat{e}_3$	
RMSEs	0.8870	0.8734	0.8376	0.8376

6.5. Blended Models

Table 6.5 gives different examples of blended models. In Example 1 we blend five predictions, which were all taken from one and the same MF, only the number of features differs. As can be seen blending the five models improved the best RMSE of 0.8045 to 0.8042. This is a small improvement. However, considering that it basically comes for free (the computational effort is minimal) it is worth to be mentioned. We also noticed a slight improvement of the overall RMSE in Example 2, where we combine several predictions from one and the same batch learning MF. In Example 3 and 4 we combine different Nearest Neighbor methods, once changing the number of neighbors and another time changing the type of the weighting function among the predictions. Example 5 combines three MF and two NN models. The example shows that the considerably weak KNN estimator is actually able to improve the overall performance significantly. Examples 6 and 7 combine the predictions of several estimators. Example 7, which is combining the predictions of all estimators presented in this thesis, yields the best result with a RMSE 0.7933. That is a major improvement compared to the lowest RMSE of 0.8043 that a single algorithm (incremental MF) can produce.

6.6. Nested Models

In section 4.4.2 we proposed a technique called iterative hierarchical clustering. For each cluster of the clustering tree presented in Figure 4.1 we compute a different MF. We use MFs with an internal polynomial kernel of $r = 1$ as well as regularization parameters of $\beta, \delta = 0.025$. The methods for each cluster differ in the number of features they make use of as well as the minimum and maximum training iterations per feature. Table 6.6 gives the setup of each MF for each cluster. "Min. ep." and "max ep." give the minimum and maximum base iterations per feature. Each feature f is now trained a minimum of f times "min. ep." iterations and a maximum of f times "max. ep." epochs. Such a setup is similar to the one we used for the

Table 6.5.: Different Blendings

Example 1	5 incr. MFs with $f = 32\,(\hat{e}_1)\,, 64\,(\hat{e}_2)\,,$ $96\,(\hat{e}_3)\,, 128\,(\hat{e}_4)\,, 164\,(\hat{e}_5)$ and $\beta, \gamma = 0.0225$					Overall
E	$\hat{e}_1$	$\hat{e}_2$	$\hat{e}_3$	$\hat{e}_4$	$\hat{e}_5$	
RMSEs	0.8176	0.8100	0.8071	0.8055	0.8045	0.8042
Weights	-0.0799	0.2087	-0.0075	0.0446	0.8368	
Example 2	5 batch MFs with $f = 1\,(\hat{e}_1)\,, 3\,(\hat{e}_2)\,,$ $5\,(\hat{e}_3)\,, 7\,(\hat{e}_4)\,, 9\,(\hat{e}_5)$ and $\beta, \gamma = 5 \times 10^{-5}$					Overall
E	$\hat{e}_1$	$\hat{e}_2$	$\hat{e}_3$	$\hat{e}_4$	$\hat{e}_5$	
RMSEs	0.8834	0.8651	0.8543	0.8456	0.8406	0.8395
Weights	0.0653	0.0461	0.0291	0.1148	0.7454	
Example 3	5 norm. NN methods with $k = 25$ and weighting $\omega\,(s) = s^{0.5}\,(\hat{e}_1)\,, s^{1.5}\,(\hat{e}_2)\,, s^{2.5}\,(\hat{e}_3)\,, s^{3.5}\,(\hat{e}_4)\,, s^{4.5}\,(\hat{e}_5)$					Overall
E	$\hat{e}_1$	$\hat{e}_2$	$\hat{e}_3$	$\hat{e}_4$	$\hat{e}_5$	
RMSEs	0.8626	0.8567	0.8526	0.8506	0.8500	0.8469
Weights	0.8048	-3.0794	7.9876	-10.1205	5.3931	
Example 4	5 norm. NN methods with weightings $\omega\,(s) = s^{4.5}$ and $k = 25\,(\hat{e}_1)\,, 30\,(\hat{e}_2)\,, 35\,(\hat{e}_3)\,, 40\,(\hat{e}_4)\,, 45\,(\hat{e}_5)$					Overall
E	$\hat{e}_1$	$\hat{e}_2$	$\hat{e}_3$	$\hat{e}_4$	$\hat{e}_5$	
RMSEs	0.8500	0.8504	0.8511	0.8518	0.8525	0.8478
Weights	0.7712	0.1180	0.0830	0.0819	-0.0688	
Example 5	3 MFs with $f = 164$ and $\beta, \gamma = 0.02\,(\hat{e}_1)\,, 0.0225\,(\hat{e}_2)\,,$ $0.025\,(\hat{e}_3)$. Also 2 NN models with $\omega\,(s) = s^{4.5}$ and $k = 40\,(\hat{e}_4)\,, 50\,(\hat{e}_5)$					Overall
E	$\hat{e}_1$	$\hat{e}_2$	$\hat{e}_3$	$\hat{e}_4$	$\hat{e}_5$	
RMSEs	0.8043	0.8045	0.8518	0.8518	0.8525	0.7998
Weights	0.4529	0.2775	0.0906	0.8779	-0.6982	
Example 6	All estimators from examples 1 to 4.					Overall
E	$\hat{e}_1$	$\hat{e}_2$	$\hat{e}_3$	$\hat{e}_4$	$\hat{e}_5$	
RMSEs	0.8176	0.8100	0.8071	0.8055	...	0.7965
Weights	-0.0816	0.1902	-0.0104	0.0509	...	
Example 7	All estimators of this thesis. More than 150 single predictions					Overall
E	$\hat{e}_1$	$\hat{e}_2$	$\hat{e}_3$	$\hat{e}_4$	$\hat{e}_5$	
RMSEs	0.8837	0.8719	0.8656	0.8592	...	0.7933
Weights	0.5126	-0.6454	0.3282	-0.2631	...	

Table 6.6.: Clusters with different MFs and the resulting RMSEs of the specific clusters.

Cluster	% of V	f	min ep.	max ep.	Clust. RMSE	RMSE in V
A_1	100	128	75	85	0.8047	0.8047
A_1	0.065	16	95	200	0.8272	0.8863
A_2	99.94	128	75	85	0.8046	0.8046
B_1	87.21	128	75	85	0.7958	0.7950
B_2	12.73	7	95	200	0.9051	0.8678
C_1	4.67	6	95	150	0.8673	0.8022
C_2	82.54	128	75	85	0.7958	0.7945
C_3	0.22	8	95	150	0.9611	0.8873
C_4	12.51	7	95	200	0.9051	0.8674
D_1	69.66	128	75	85	0.8003	0.7953
D_2	12.88	16	95	150	0.8349	0.7907
D_3	0.14	7	95	200	0.9958	0.9328
D_4	12.37	7	95	200	0.9048	0.8667
D_5	0.002	7	95	200	1.0726	0.9622
E_1	0.19	7	95	200	1.0134	0.9511
E_2	12.17	7	95	200	0.9040	0.8652

base algorithm presented in section 3.2.6. Table 6.6 also gives the RMSE of each MF on a cluster (Clust. RMSE). The column "RMSE in V" gives the RMSE of the subset of the validation set that was determined when training on the *whole* training set and not just on a specific cluster. The setup for this specific MF made use of 128 features, $\beta, \delta = 0.025$, $75 \times f$ minimum epochs, $85 \times f$ maximum epochs and learning rates of $\alpha = 3 \times 10^{-4}$ and $\chi = 10^{-4}$ as indicated in the first row of the table. Aside from cluster A_1 all indepedentend MFs perform worse than a MF, which was trained on the entire training set. However, the performance of some clusters is considerably good. MFs on small clusters usually require lot less time to train because of the smaller amount data behind it but also because of the reduced number of features used.

In Table 6.7 we combine several clusters to form the complete validation set V once again. The given RMSEs summarize the performance if the

Table 6.7.: Different cluster combinations

Clusters	Total RMSE
M	0.8047
$A_1 \cup A_2$	0.8046
$A_1 \cup B_1 \cup B_2$	0.8106
$A_1 \cup C_1 \cup C_2 \cup C_3 \cup C_4$	0.8141
$A_1 \cup C_1 \cup D_1 \cup D_2 \cup C_3 \cup D_3 \cup D_4 \cup D_5$	0.8223
$A_1 \cup C_1 \cup D_1 \cup D_2 \cup C_3 \cup D_3 \cup E_1 \cup E_2 \cup D_5$	0.8224
$A_1 \cup C_1 \cup C_2 \cup B_2$	0.8139
$A_1 \cup B_1 \cup C_3 \cup C_4$	0.8107

validation set would be put together again by the different clusters. In a productive system such an approach would be efficient because the smaller clusters require less training time. We can see from Table 6.7 that even excessive clustering does not impair the final RMSEs too much. Therefore we can only recommend such an approach for the usage in a productive environment, probably even with a higher number of clusters.

6.7. Movielens Dataset

As an addition we also tested the performance of the single algorithms on an alternative dataset, the Movielens dataset[1] [43]. It consists of one million ratings for 3900 movies by 6040 users.

6.7.1. Matrix Factorization

Table 6.8 and Figure 6.5 summarize and visualize the performance of incremental MF on the Movielens dataset. The same setup as for the Netflix Prize was used. Figure 6.5 shows that for all implementations there are a number of training epochs where training does not improve at all or only improves slightly. During those epochs the number of minimum training iterations

---

[1]`http://www.grouplens.org/`

Table 6.8.: RMSEs for incremental MF on Movielens dataset, maximum iterations $85 + 2f$, minimum iterations $75 + f$

MF preferences	$f = 32$	$f = 64$	$f = 96$	$f = 128$	$f = 164$
$\alpha = 2.5 \times 10^{-4}, \chi = 10^{-4}$ $\beta = 0.0, \delta = 0.0$	0.9006	0.8828	0.8828	0.8676	0.8632
$\alpha = 2.5 \times 10^{-4}, \chi = 10^{-4}$ $\beta = 0.015, \delta = 0.015$	0.8980	0.8905	0.8764	0.8757	0.8627
$\alpha = 2.5 \times 10^{-4}, \chi = 10^{-4}$ $\beta = 0.02, \delta = 0.02$	0.8980	0.8831	0.8740	0.8651	0.8600
$\alpha = 2.5 \times 10^{-4}, \chi = 10^{-4}$ $\beta = 0.0225, \delta = 0.0225$	0.8980	0.8803	0.8763	0.8642	0.8636
$\alpha = 2.5 \times 10^{-4}, \chi = 10^{-4}$ $\beta = 0.025, \delta = 0.025$	0.8980	0.8824	0.8791	0.8737	0.8620
$\alpha = 2.5 \times 10^{-4}, \chi = 10^{-4}$ $\beta = 0.03, \delta = 0.03$	0.8980	0.8814	0.8807	0.8659	0.8653

was set too low resulting in the situation that a feature stops training before it actually could start improving the overall RMSE.

Also Table 6.9 and Figure 6.6 give the performance of a MF via batch learning on the Movielens dataset. Here we used a different setup as with the Netflix Prize dataset, mainly because the original setup did not work for this dataset, since the number of minimum training epochs was too low. For the Movielens dataset we used a minimum number of training epochs per feature of $750 \times f$ and once again f is the actual trained feature. This of course results in a lot more training epochs overall for $f = 9$ features.

We found that MF via batch learning can perform as well as incremental MF. However, to reach the RMSE of the incremental approach MF via batch learning requires a lot more training epochs. Therefore we also have to recommend using the incremental approach for the Movielens dataset. It reaches a low RMSE in a shorter time than a MF via batch learning could do. It is therefore more productive.
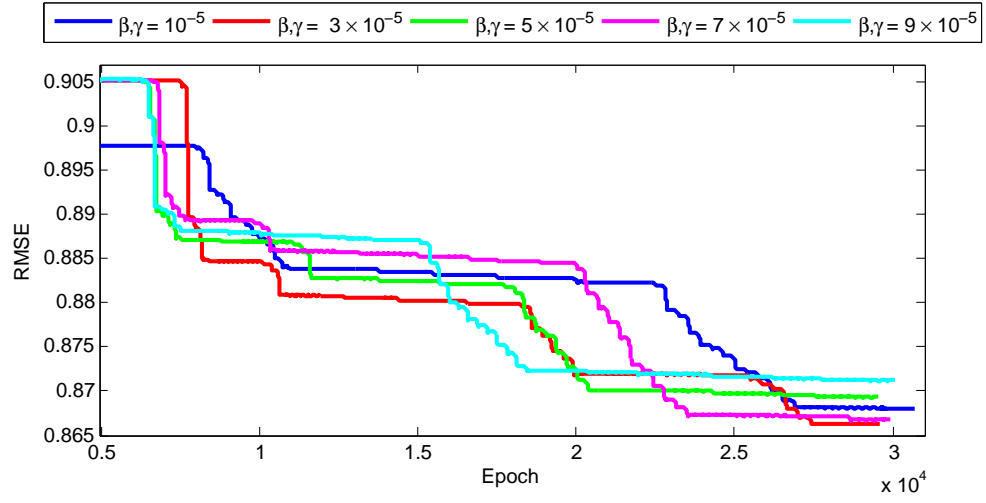
Figure 6.5.: Incremental MF performance on validation of Movielens dataset. Epochs from 5000 to 30000. Maximum of 164 features. Learning rates fixed at $\alpha = 0.00025$ and $\gamma = 0.0001$.

Table 6.9.: RMSEs for batch learning MF on Movielens dataset, maximum iterations ∞, minimum iterations $1750 \times f$

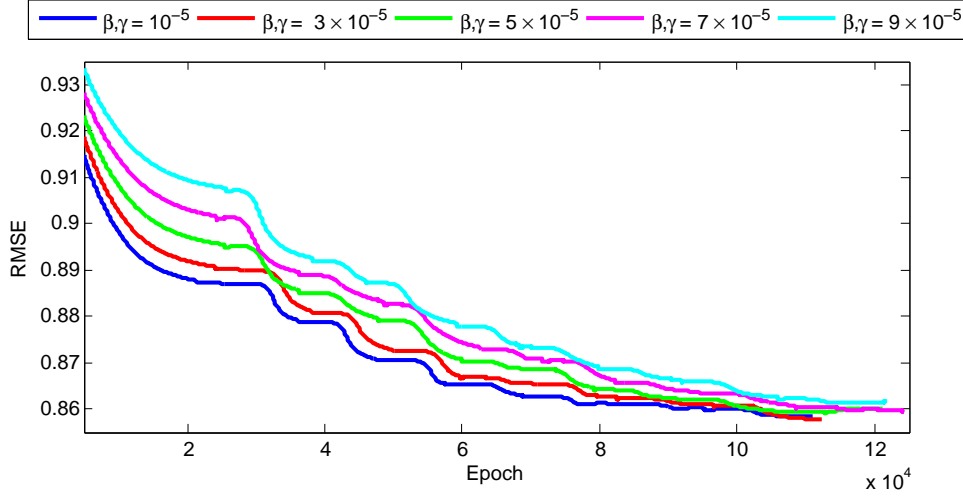| MF preferences	$f = 1$	$f = 3$	$f = 5$	$f = 7$	$f = 9$
$\alpha_B = 10^{-5}, \chi_B = 5 \times 10^{-6}$ $\beta = 10^{-5}, \delta = 10^{-5}$	0.8837	0.8656	0.8541	0.8463	0.8413
$\alpha_B = 10^{-5}, \chi_B = 5 \times 10^{-6}$ $\beta = 3 \times 10^{-5}, \delta = 3 \times 10^{-5}$	0.8854	0.8767	0.8643	0.8578	0.8566
$\alpha_B = 10^{-5}, \chi_B = 5 \times 10^{-6}$ $\beta = 5 \times 10^{-5}, \delta = 5 \times 10^{-5}$	0.8897	0.8745	0.8650	0.8591	0.8578
$\alpha_B = 10^{-5}, \chi_B = 5 \times 10^{-6}$ $\beta = 7 \times 10^{-5}, \delta = 7 \times 10^{-5}$	0.8954	0.8773	0.8668	0.8602	0.8586
$\alpha_B = 10^{-5}, \chi_B = 5 \times 10^{-6}$ $\beta = 9 \times 10^{-5}, \delta = 9 \times 10^{-5}$	0.9011	0.8826	0.8694	0.8618	0.8589

Figure 6.6.: Batch learning MF performance on validation of Movielens dataset. Epochs from 5000 to 120000. Maximum of 8 features. Learning rates fixed at $\alpha = 0.00025$ and $\gamma = 0.0001$.

6.7.2. Nearest Neighbor Methods

We also used Nearest Neighbor methods to test the performance of the estimators on the Movielens dataset. Table 6.10 gives the RMSEs of various combinations of weighting functions and number of neighbors. NN methods involving a weighting function of the form $\omega_r(s) = s^r$ also perform better on this dataset than the estimators involving the exponential function as the weighting function. Peak performance is reached for $r = 4.5$ with $k = 35$ neighbors. Figures 6.7 and 6.8 visualize the context. Both figures show a strong resemblance to the respective figures of the Netflix Prize dataset (Figures 6.3 and 6.4). This is not so surprising if one keeps in mind that both collaborative filtering systems recommend the same kind of items (movies) and also are based on a five-star integer rating.

Table 6.10.: RMSEs for different NN methods on the Movielens dataset. Rating matrix substracted by movie means.

| $\omega(s)$	$k=10$	$k=15$	$k=20$	$k=25$	$k=30$	$k=35$	$k=40$	$k=45$
$s^{0.5}$	0.8748	0.8701	0.8695	0.8693	0.8700	0.8706	0.8716	0.8724
$s^{1.0}$	0.8736	0.8687	0.8679	0.8676	0.8681	0.8687	0.8695	0.8703
$s^{1.5}$	0.8726	0.8675	0.8664	0.8660	0.8664	0.8668	0.8676	0.8683
$s^{2.0}$	0.8719	0.8664	0.8651	0.8645	0.8648	0.8651	0.8658	0.8664
$s^{2.5}$	0.8714	0.8656	0.8641	0.8633	0.8634	0.8636	0.8642	0.8647
$s^{3.0}$	0.8712	0.8652	0.8633	0.8624	0.8624	0.8625	0.8629	0.8633
$s^{3.5}$	0.8714	0.8650	0.8629	0.8618	0.8616	0.8616	0.8619	0.8622
$s^{4.0}$	0.8718	0.8652	0.8629	0.8616	0.8612	0.8611	0.8613	0.8615
$s^{4.5}$	0.8726	0.8657	0.8632	0.8617	0.8612	0.8609	0.8610	0.8611
$s^{5.0}$	0.8737	0.8665	0.8638	0.8622	0.8615	0.8611	0.8611	0.8611
0.5^s	0.8765	0.8720	0.8716	0.8715	0.8723	0.8730	0.8741	0.8750
1.0^s	0.8761	0.8716	0.8712	0.8711	0.8719	0.8726	0.8737	0.8746
1.5^s	0.8759	0.8714	0.8710	0.8709	0.8716	0.8723	0.8734	0.8743
2.0^s	0.8758	0.8712	0.8709	0.8707	0.8714	0.8721	0.8733	0.8741
2.5^s	0.8757	0.8711	0.8707	0.8706	0.8713	0.8720	0.8731	0.8740
3.0^s	0.8756	0.8710	0.8706	0.8705	0.8712	0.8719	0.8730	0.8739
3.5^s	0.8755	0.8709	0.8705	0.8704	0.8711	0.8718	0.8729	0.8738
4.0^s	0.8754	0.8709	0.8705	0.8703	0.8710	0.8717	0.8728	0.8737
4.5^s	0.8754	0.8708	0.8704	0.8703	0.8710	0.8716	0.8727	0.8736
5.0^s	0.8753	0.8707	0.8703	0.8702	0.8709	0.8716	0.8727	0.8735
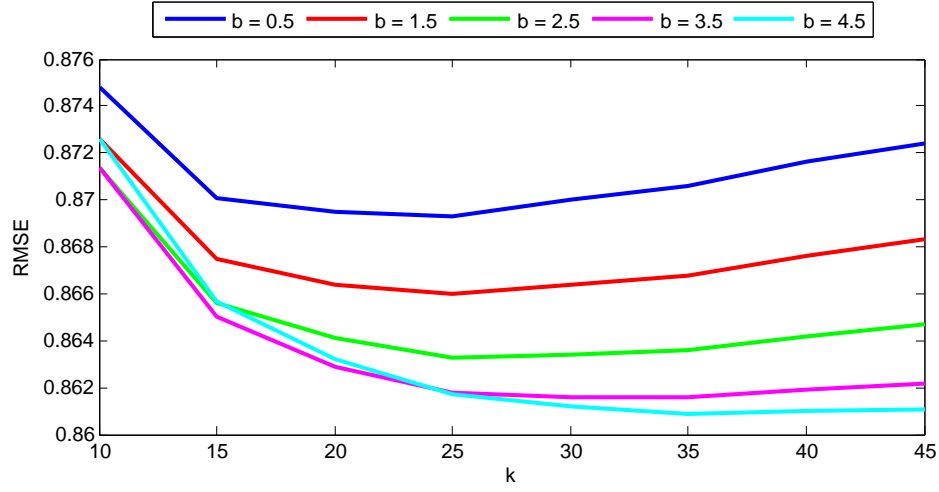
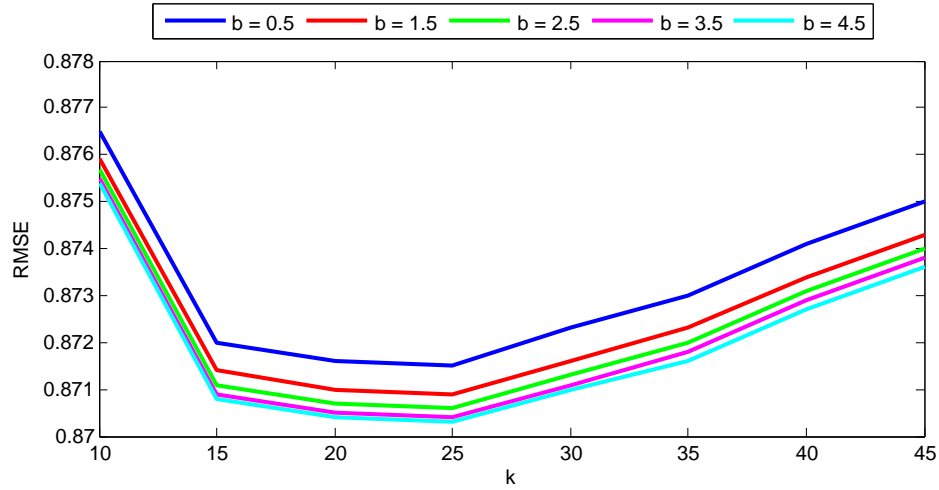Figure 6.7.: Nearest Neighbor algorithms on the Movielens dataset. Weighting functions of form s^e.

Figure 6.8.: Nearest Neighbor algorithms on the Movielens datasets. Weighting functions of form b^s.

6. Experiments

7. Conclusion

In this thesis we approached the problem of the Netflix Prize and how to realize collaborative filtering (CF) in general. We proposed two main algorithms for predicting ratings users will asign to items in a recommender system. In case of the Netflix Prize those items were movies. The algorithms were based on the method of Matrix Factorization as well as on a Nearest Neighbor search. Different variations of both algorithms were implemented. As for the Matrix Factorization we implemented several variations, which for example added constant factors to the prediction equation or introduced a non-linear weighting of the feature matrices to the formula. For the Nearest Neighbor algorithm we mentioned different weighting functions for the similarity and also discussed the topic of how performance can be improved by normalizing the data within the actual prediction step.

We also introduced three methods of combining prediction models. One method was based on the concatenation of different models, in a way that one model predicts the residuals the models before it could not cover. Next to this we described and suggested a least square blending, which is capable of finding the best linear combination of a given number of models. Furthermore we introduced the idea of nested models, which in some way incorperate one model into another. The concatenation of different models did prove useful for a certain order of certain models. We found that a concatenation of a well performing NN method with a well performing MF does not perform as well as the single MF. On the other hand the combination of different MFs with different learning rates and regularization parameters slightly improved the performance of the first model. However, most of the times it is hard to even outperform the RMSE of a well set MF. Furthermore we found that

7. Conclusion

MFs with internal polynomial kernels of $r > 1$ are not or only slightly able to improve the performance of incremental MF using a kernel of $r = 1$. We think this observation is not a result of the approach or technique itself but rather of the fact that it is extremely complicated to set the parameters of a MF with a kernel greater than $r = 1$. We had our difficulties to find working minimum and maximum thresholds for training interations per feature. It often occured that the minimum number of training epochs was set too low, so the RMSE on the validation set did not improve at all during training or the minimum number was set too high, so that the RMSE actually improved but got a lot worse with the later iterations. While for MFs using an internal kernel of $r = 1$ we were able to simple define the lower and upper threshold of training iterations per feature by some simple rules, like $75 + f$ for the lower one and $85 + 2f$ for the upper one (and f is the actual trained feature), we were not able to use those rules succesfully for MFs using an internal polynomial kernel of either $r = 2$ or $r = 3$.

Also we evaluated the performance of all models on the Netflix Prize dataset as well as on the Movielens dataset in chapter 6. We explained how the models react on different preferences as well as on different training instances, such as clusters of the complete training dataset. We noted that clustering the training set will lead to a worse performance than using the complete training set. However, when using for instance different Matrix Factorizations for each cluster we can easily parallelize these algorithms and therefore use todays computer systems more efficiently. This fact could be useful in a productive system, which does not aim on the lowest error but on the highest efficiency. Matrix Factorizations on smaller datasets train faster and are easier to parallelize.

We found that Matrix Factorization techniques perform better than techniques based on the NN theory. However, NN methods do not require any training and the time to compute one prediction is, if implemented in an efficient way, still fast enough for a real-time system. In the end the choice is left to the person who implements the system into a productive environment and how high the requirements for performance, in both means of computational

76

speed or a lower error, are. As already mentioned we found that Nearest Neighbor methods perform faster but not as good in terms of the error as the MF techniques. Using subsets or only samples of the training set might improve the computational speed of a MF and not lower the final error too much. From this point of view the nested models based on the clustering of the training set, as presented in this thesis (sections 4.4 to 4.4.2), can be thought of a compromise of speed and preciseness.

7.1. Future Work

There is a lot to say about where further work can be done on the algorithms as well as on the implementation itself.

7.1.1. Prediction Models

It is of course a good idea to implement more types of models, not only models that are based on the NN theory and MF techniques. Restricted Bolzman Machines [47] have proven to perform well on the dataset of the Netflix Prize, so have variational bayesian approaches [33]. Furthermore there are a lot of straightforward and easy to implement methods, such as Slope One [49] [32], which do not perform well but might be a valueable addition to a combinational approach.

Also further research on the methods presented in this thesis might be worthwhile. Mainly all of the top ranked teams of the Netflix Prize use either one of those two approaches or they use at least one of them in a combination with other models. The at the moment leading team of the AT&T labs uses a prediction model, which is also based on the blending of several predictions [8]. In [61] the author describes different ensembles of Matrix Factorization and applies them to the dataset of the Netflix Prize.

A very promising approach is also the usage of Ridge Regression on the user feature matrix of a MF. In [39] Paterek created a Ridge Regression model for each user u_i by using the movie features of each movie that was

rated by user u_i. The method performs quite well and is furthermore easy to implement. Also training is very quick, so it is a valueable addition to the basic Matrix Factorization as presented in this thesis. We can think of it as a nested model.

Other nested models, such as the inclusion of the features of the k nearest neighbors into a MF do require further research and a performance evaluation. The clustering of the training set has proven useful to combine training speed and performance in terms of a considerably low RMSE. Further research has to be done into the direction on how to keep the error low but improve computing time at the same time.

7.1.2. Implementation

Our implementation is based on C++ code and is already very fast. Further work can be done on the inclusion of the instruction sets of the newer processors [14], such as SSE. If implemented right these would save a significant amount of training time. Also further research on the way how the data is stored can improve execution time for all algorithms.

As for this moment the framework does not support any multi-threaded programming style, meaning that each instance of the framework can only run on one core of the processor (logical or physical). The usage of nested models that train several MFs at once can overcome this problem. However, they are not more than a workaround. A more efficient usage of the processing units can save a lot of time, especially when it comes to the computational performance of the Nearest Neighbor algorithms.

For our algorithms we used the Microsoft compiler that comes with Visual Studio. Other compilers, such as the Intel C++ Compiler [14] or the gcc [54] [23] or any other compiler [51], might offer a superior performance and are therefore worth investigating.

Also the inclusion of a GPU (Graphic Processing Unit) in the programming process might be a useful idea. A GPU is in general a lot more powerful than a CPU but a lot harder to program. A first step might be the study

of the CUDA (Compute Unified Device Architecture) technology [42], which enables a user to run C programming code on a GPU.

7.1.3. Filtering and Enriching the Datasets

To validate the performance of the algorithms we removed a validation set of 3 million ratings from the complete training set of 100 million ratings. We did expect a significant cut in prediction performance on the qualification set of the Netflix Prize when training on the reduced set in comparison to the complete training set. However, the decreasement of the error from the training set with 97 million training instances to the one with 100 instances was rather small. It is therefore a logical conclusion that the removal of certain data instances might improve the performance of different prediction algorithms overall. Matrix Factorization techniques, for example, suffer from users that rated a lot of movies all the same. This is especially true for some users of the Netflix Prize and therefore worth investigating.

On the other side it seems to be a valueable idea to test whether it is possible to enrich the dataset in any way. The here discussed algorithms are capable of giving a very low RMSE on a certain group of users and movies. Adding these predictions to the training set might improve the overall RMSE on the qualificaton set of the Netflix Prize further.

Also the way how to combine different models requires more research. Some users have a standard deviation of zero, which means that they rated all the movies the same. The predictions for those users are simple to make and the usage of these rating instances in a least square blending or concatanation will only bias the final parameters. So outcluding them before training a model but also before combining different models might be a good idea to improve performance furthermore.
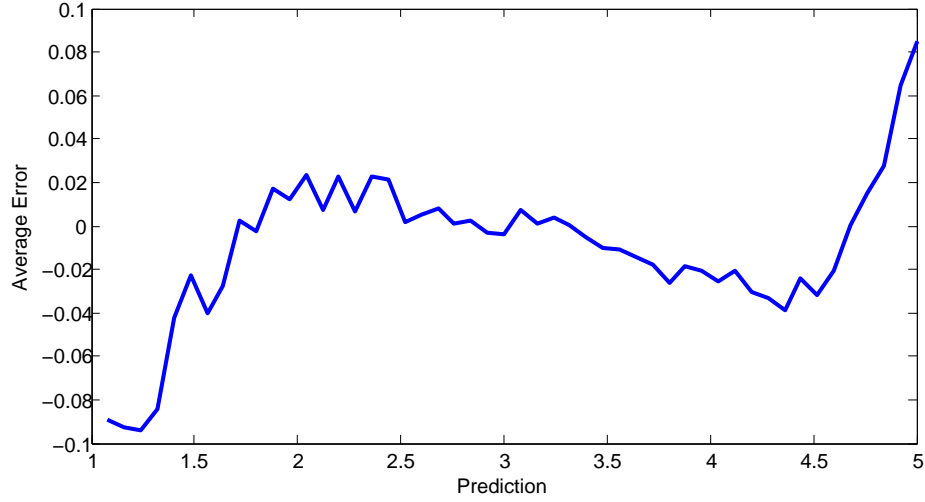
7. Conclusion

Figure 7.1.: Actual Prediction plotted against Average Prediction Error

7.1.4. Histogram Shifting and Average Error Normalization

While the predictions of our estimators are given within the interval $[1; 5]$ as real numbers the actual ratings of the Netflix Prize were natural numbers of $\{1, 2, 3, 4, 5\}$ stars. Therefore it might be a valueable idea to shift the histograms of the predictions to make them fit the distributions of the ratings of the training set. However, the question stays how to round the predictions. The publication on the Gravity Recommendation System [57] presented some rounding approaches. None of them turned out to be useful for improving the RMSE on their validation set. However, they might still be useful to approach histogram shifting.

One more way to improve performance of the algorithms is a technique we call Average Error Normalization. Figure 7.1 shows the mean error of each prediction of MF model. Of course such a plot requires some form of binning of the prediction values. For the plot we used a setup of 50 bins. The predictions of the validation set were arranged in those bins and for each bin the average error was computed. Then each bins upper threshold

80

was plotted against the average error. As can be seen from the plot it is obviously true that the smaller predictions are actually a little too small while the higher predictions are a little too high. By normalizing this plot to the x-axis, meaning to subtract the offset from each item in the bin, we are able to improve the resulting RMSE by a small amount. Furtheremore this normalization could be used in every feature training step of a MF. This means that once a feature finished training the predictions of that feature (and the features before it) are not only clipped to the $[1; 5]$ interval but also normalized in terms of the average error per bin.

7.1.5. Inclusion of the time component

Both of the evaluated datasets, the Netflix Prize set and the MovieLens dataset, come with a timestamp indicating when a rating was given. As mentioned at the beginning of this thesis we did remove the date from any of our prediction models. The dates of the ratings in the qualification set of the Netflix Prize, however, are arranged at the end of the observation period. This is only logical, since the systems job will be to predict ratings that have not been done yet. From this point of view the here presented validation set, which is just based on a random selection within the training set, is actually not a good choice for validation purposes. Therefore it is a good idea to create a time-dependend validation set with instances from the end of the observation period. The validation set used in this thesis only works under the assumption that the ratings are not time-depended. After a detailed study of the rating behaviour of the different users we made this assumption and therefore see the ratings not on a timeline.

7.2. Performance on qualification set

We want to point out that the thesis was rather aimed directly on the Netflix Prize itself but on evaluating methods how the Prize *could* be approached. The removal of a validation set with a size of 3 million rating instances of

7. Conclusion

course influenced the performance of the algorithms on the qualification set of the Netflix Prize. Also the approach of the thesis was to realize efficient and productive prediction algorithms for recommender systems.

The single NN methods are giving RMSEs in the range of 0.94 to 0.98 while the incremental MFs are able to reach a RMSE around 0.92. The blendings of all of our methods resulted in a RMSE slightly below 0.9, which is nothing very special. However, with a smaller validation set the algorithms might perform a lot better. Also none of these numbers include the fine-tuning ideas we mentioned in section 7.1. From the publications on the Netflix Prize [8] [57] [61] [] so far we can see that with a smaller validation set and some fine tuning the here proposed methods are able to achieve a RMSE lower than 0.89.

Bibliography

[1] Herve Abdi. Singular value decomposition (svd) and generalized singular value decomposition (gsvd). In Neil Salkind, editor, *Encyclopidia of Measurement and Statistics*. Thousands Oaks, CA, 2007.

[2] Gediminas Adomavicius and Alexander Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):734–749, 2005.

[3] Kamal Ali and Wijnand van Stam. Tivo: making show recommendations using a distributed collaborative filtering architecture. In *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 394–401, New York, NY, USA, 2004. ACM Press.

[4] Joshua Alspector, Aleksander Kolcz, and Nachimuthu Karunanithi. Comparing feature-based and clique-based user models for movie selection. In *DL '98: Proceedings of the third ACM conference on Digital libraries*, pages 11–18, New York, NY, USA, 1998. ACM.

[5] Michael R. Anderberg. *Cluster Analysis for Applications*. Monographs and Textbooks on Probability and Mathematical Statistics. Academic Press, Inc., New York, 1973.

[6] Marko Balabanovi and Yoav Shoham. Fab: content-based, collaborative recommendation. *Commun. ACM*, 40(3):66–72, 1997.

Bibliography

[7] David Bau III and Lloyd N. Trefethen. *Numerical linear algebra.* Society for Industrial and Applied Mathematics, 1997.

[8] Robert M. Bell and Yehuda Koren. Improved neighborhood-based collaborative filtering. In *Proceedings of the KDD Cup 2007*, pages 7–14. ACM Press, 2007.

[9] James Bennett, Charles Elkan, Bing Liu, Pad. Smyth, and Domonkos Tikk. Introduction on the kdd cup. In *Proceedings of the KDD Cup 2007*, pages 1–2. ACM Press, 2007.

[10] James Bennett and Stan Lanning. The netflix prize. In *Proceedings of the KDD Cup 2007*, pages 3–6. ACM Press, 2007.

[11] Pavel Berkhin. Survey of clustering data mining techniques. Accrue Software, Inc.

[12] Michael W. Berry. Large scale sparse singular value computations. Technical report, Department of Computer Science, University of Tennessee, 107 Ayres Hall Knoxville TN, 37996-1301, 2002.

[13] James C. Bezdek. *Pattern Recognition with Fuzzy Objective Function Algorithms.* Kluwer Academic Publishers, Norwell, MA, USA, 1981.

[14] Aart J. C. Bik. *The Software Vectorization Handbook: Applying Intel Multimedia Extensions for Maximum Performance.* Intel Press.,US, 2004.

[15] Hariprasad Bommaganti and Anand Nagarajan. k-split based approach to predict movie rating frequency. In *Proceedings of the KDD Cup 2007*, pages 84–87. ACM Press, 2007.

[16] A. J. Cole and D. Wishart. An improved algorithm for the jardine-sibson method of generating overlapping clusters. *The Computer Journal*, 113(2):156–163, 1970.

[17] Antonio J. Conejo, Enrique Castillo, Roberto Minguez, and Raquel Garcia-Bertrand. *Decomposition Techniques in Mathematical Programming: Engineering and Science Applications.* Springer, 2006.

[18] J. Demmel and W. Kahan. Accurate singular values of bidiagonal matrices. *SIAM J. Sci. Statist. Comput.*, 11(5):873–912, 1990.

[19] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[20] Brian Everitt, Sabine Landau, and Morven Leese. *Cluster Analysis.* A Hodder Arnold Publication; 4th edition, 2001.

[21] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, 1996.

[22] Gene H. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14(5), 1970.

[23] Arthur Griffith. *GCC: The Complete Reference.* McGraw-Hill Osborne Media, 2002.

[24] Jonathan L. Herlocker, Joseph A. Konstan, Loren G. Terveen, and John T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Trans. Inf. Syst.*, 22(1):5–53, 2004.

[25] Nikhil Rastogi James Malaugh, Sachin Gangaputra. Kdd cup 2007, how often will that movie be rated? In *Proceedings of the KDD Cup 2007*, pages 80–83. ACM Press, 2007.

[26] N. Jardine and R. Sibson. The construction of hierarchic and non-hierarchic classifications. *The Computer Journal*, 11(2):177–184, 1968.

[27] Stephen C. Johnson. Hierarchical clustering schemes. 23(3):241–254, 1966.

Bibliography

[28] Joe Luis Florez Jorge Sueiras, Daniel Velez. A combination of approaches to solve task "how many ratings?" of the kdd cup 2007. In *Proceedings of the KDD Cup 2007*, pages 75–79. ACM Press, 2007.

[29] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Probability and Mathematical Statistics. John Wiley and Sons, Inc., New York, 1989.

[30] Miklo Kurucz, Andras A. Benczur, and Károly Csalogány. Methods for large scale svd with missing values. In *Proceedings of the KDD Cup 2007*, pages 54–58. ACM Press, 2007.

[31] Miklo Kurucz, Andras A. Benczur, Tamas Kiss, Istvan Nagy, Adrienn Szabo, and Balazs Torma. Who rated what: a combination of svd, correlation and frequent sequence mining. In *Proceedings of the KDD Cup 2007*, pages 48–53. ACM Press, 2007.

[32] Daniel Lemire and Anna Maclachlan. Slope one predictors for online rating-based collaborative filtering, 2007.

[33] Yew Jin Lim and Yee Whye Teh. Variational bayesian approach to movie rating prediction. In *Proceedings of the KDD Cup 2007*, pages 15–21. ACM Press, 2007.

[34] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, 2003.

[35] Ting Liu, Yonghong Tian, and Wen Gao. A two-phase spectral bigraph co-clustering approach for the who rated what task in kdd cup 2007. In *Proceedings of the KDD Cup 2007*, pages 63–68. ACM Press, 2007.

[36] Yan Liu and Zhenzhen Kou. Predicting who rated what in large-scale datasets. In *Proceedings of the KDD Cup 2007*, pages 59–62. ACM Press, 2007.

[37] James B. MacQueen. Some methods for classification and analysis of multivariate observations. *Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1966.

[38] Prem Melville, Raymond J. Mooney, and Ramadass Nagarajan. Content-boosted collaborative filtering for improved recommendations. Technical report, Department of Computer Sciences, University of Texas, Austin, TX 78712, 2002.

[39] Arkadiush Paterek. Improving regularized singular value decomposition for collaborative filtering. In *Proceedings of the KDD Cup 2007*, pages 39–42. ACM Press, 2007.

[40] Michael J. Pazzani. A framework for collaborative, content-based and demographic filtering. *Artificial Intelligence Review*, 13(5-6):393–408, 2004.

[41] Michael J. Pazzani and Daniel Billsus. Content-based recommendation systems. *Lecture Notes in Computer Science*, 4321:325–341, 2007.

[42] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[43] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *CSCW '94: Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186, New York, NY, USA, 1994. ACM Press.

[44] Paul Resnick and Hal R. Varian. Recommender systems. *Commun. ACM*, 40(3):56–58, 1997.

[45] Yan Liu Saharon Rosset, Claudia Perlich. Making the most of your data: Kdd cup 2007 "how many ratings" winners report. In *Proceedings of the KDD Cup 2007*, pages 69–74. ACM Press, 2007.

Bibliography

[46] Ruslan Salakhutdinov and Andriy Mnih. Probabilistic matrix factorization. Technical report, Department of Computer Science, University of Toronto, Kings College Rd, M5S 3G4, Canada, 2007.

[47] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted boltzmann machines for collaborative filtering. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 791–798, New York, NY, USA, 2007. ACM Press.

[48] Gerard Salton and Christopher Buckley. Clustering. Institute for Perception, Action and Behaviour, Division of Informatics, University of Edinburgh.

[49] Badrul Sarwar, George Karypis, Joseph Konstan, and John Reidl. Item-based collaborative filtering recommendation algorithms. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 285–295, New York, NY, USA, 2001. ACM Press.

[50] J. Ben Schafer, Joseph Konstan, and John Riedi. Recommender systems in e-commerce. In *EC '99: Proceedings of the 1st ACM conference on Electronic commerce*, pages 158–166, New York, NY, USA, 1999. ACM Press.

[51] Herbert Schildt and Gregory L. Guntle. *Borland C++ Builder: The Complete Reference*. McGraw-Hill Companies, 2001.

[52] Ingo Schmitt. *Multimedia-Datenbanken: Retrieval, Suchalgorithmen und Anfragebearbeitungen*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, 2004.

[53] David Skillicorn. *Understanding Complex Datasets: Data Mining with Matrix Decompositions*. Chapman and Hall/CRC, 2007.

[54] Richard Stallman. *Gnu Reference: Using and Porting the Gnu Complier Collection Gcc*. Iuniverse Inc, 2000.

[55] Detlef Steinhausen and Klaus Langer. *Clusteranalyse: Einführung in Methoden und Verfahren der automatischen Klassifikation.* De Gruyter, Berlin, 1977.

[56] Jorge Sueiras, Alfonso Salafranca, and Jose Luis Florez. A classical predictive modeling approach for task "who rated what?" of the kdd cup 2007. In *Proceedings of the KDD Cup 2007*, pages 34–38. ACM Press, 2007.

[57] Gabor Takacs, Istvan Pilaszy, Bottyan Nemeth, and Domonkos Tikk. On the gravity recommendation system. In *Proceedings of the KDD Cup 2007*, pages 22–30. ACM Press, 2007.

[58] Luis von Ahn, Manuel Blum, and John Langford. Telling humans and computers apart automatically. *Commun. ACM*, 47(2):56–60, 2004.

[59] Luis von Ahn and Laura Dabbish. Labeling images with a computer game. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 319–326, New York, NY, USA, 2004. ACM Press.

[60] Luis von Ahn, Ruoran Liu, and Manuel Blum. Peekaboom: a game for locating objects in images. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 55–64, New York, NY, USA, 2006. ACM Press.

[61] Mingrui Wu. Collaborative filtering via ensembles of matrix factorizations. In *Proceedings of the KDD Cup 2007*, pages 43–46. ACM Press, 2007.

Bibliography

90

A. Programming

A.1. Overview

For programming we did use C++ in form of Visual C++. The used IDE was Visual Studio Express for C++ in the version 2005. The guideline always was to maximize speed. To do so we fine-tuned several parameters of the programs. The programming style is object oriented and for each model (either MF or KNN) we are able to create one object.

For some programs we did not use C++ because it is not very comfortable to program it. Instead we used either C#, for example for reordering the datasets, or MatLab, for example for computing the optimal linear combination of some estimators.

The main program, however, was programmed in C++. We implemented several classes and data structures. It is possible to create several estimators, feed them with data, train them and link them to other estimators to combine them as a estimator concatenation.

All methods and attributes as well as all classes are still public within the programming environment. This is due to the fact that development still is not finished. The implementation should be thought of as an testing environment. It still requires a lot of work to transform it into a stable program.

In the following sections we will describe the main program as well as the MatLab algorithms. We will not explain the C# programs, since they are straight forward and do not require any explanation. Usually the main part of those programs only consinsts of a few lines of code. The C# programs are also not vital for the algorithms to work. C# was only used to preprocess

data or to move it from one storing location (for example a database) to another, like a textfile.

A.2. Main Program

As already mentioned the main part of the thesis was realized in C++. We will not give an explanation for all methods and attributes of each class. However, we will show how to create and train objects. For a detailed documentation on the classes the reader is refered to the source code and the given commentations in it.

As for the main program the following objects are available

- **Helper:** A Helper class. Offering different methods, such as saving datasets.

- **Model:** The highest class. All other estimators inherit from this class. It offers basic functionality.

    - **MF:** The base class for all MF algorithms.

        * **MFuno:** Incremental Matrix factorization (MF) with an internal polynomial kernel of $r = 1$.

        * **MFduo:** Incremental MF with an internal polynomial kernel of $r = 2$.

        * **MFtrio:** Incremental MF with an internal polynomial kernel of $r = 3$.

        * **MFunoBatch:** MF with an internal polynomial kernel of $r = 1$ using batch learning for training.

    - **KNN:** The base class for all Nearest Neighbor algorithms.

        * **KNNpower:** Nearest neighbor method using a weighting function of $\omega_r(s) = s^r$

        * **KNNexp:** Nearest neighbor method using a weighting function of $\omega_b(s) = b^s$

RIUser
+movie : unsigned short
+rating : unsigned char
+alreadyPredicted : float

Sim
+similarity : float
-index : short

Model
+*DataField : std::vector<RIUser>
+*DataFieldVali : std::vector<RIUser>
+*DataFieldValiSVD : std::vector<RIUser>
+*DataFieldQuali : std::vector<RIUser>
+maxUsers : int
+maxMovies : short
+TrainSize : int
+ValiSize : int
+ValiSizeSVD : int
+QualiSize : int
+type : char
+Model()
+~Model()
+virtual CompletePredictRating() : float
+FillTrainingField(in Feat : int, in cachit : bool) : float
+FillValiField(in Feat : int, in cacheit : bool) : float
+FillValiSVDField(in Feat : bool, in cacheit : bool) : float
+FillQualiField(in Feat : int, in cacheit : bool) : float

KNN
+*TempArray[] : float
+*TempCache[] : std::vector<float>
+*SI[] std::vector<sim>
+*UserAvg[] : float
+*MovieAvg[] : float
+*UserNos[] : short
+*MovieNos[] : int
+k : int
+kn : int
+Avg : bool
+KNN()
+~KNN()
+DuplicateCache()
+ClearTempCache()
+virtual SetData(in &InDF:vector<RIUser>, in ...)

MF
-**UserFeature[][] : float
-**MovieFeature[][] : float
-*cUser[] : float
-*dMovie[] : float
-Features : int
-max_IterationsBase : int
-min_IterationsBase : int
-ActFeature : int
-ActEpoch : int
-ActGlobalEpoch : int
+flrate : float
+clrate : float
+alpha : float
+delta : float
+ActRMSE : float
+ActVRMSE : float
+ActTime : float
+ActFiveDiff : float
+FiveCache[5] : float
-*UserNos[] : short
-*MovieNos[] : short
+MF()
+~MF()
+SaveTrainField(in ToFile : wchar_t)
+SaveValiField(in ToFile : wchar_t)
+SaveValiSVDField(in ToFile : wchar_t)
+SaveQualiField(in ToFile : wchar_t)
+virtual InitializeFeatures()
+virtual DoTrainingCycle()
+virtual DoTrainingFeature()
+virtual DoTraining()
+virtual DoValidationSVD()
+virtual CacheTraining(in final : bool)
+virtual CacheValiSVD(in final : bool)
+virtual CacheVali(in final : bool)
+virtual CacheQuali(in final : bool)
+virtual ShowStatus()
+virtual WriteStatus()
+virtual SetData(in &InDF:vector<RIUser>, in ...)

KNNpower
+exponent : float
+KNNpower(in Users : int, in Mov : int, in TS : int, in VS : int, in QS : int)
+~KNNpower()
+predictRating(in movie : short, in user : int) : float

KNNexp
+base : float
+KNNexp(in Users : int, in Mov : int, in TS : int, in VS : int, in QS : int)
+~KNNexp()
+predictRating(in movie : short, in user : int) : float

MFunoBatch
-*GradUserCache[] : float
-*GradMovieCache[] : float
-*GradUserCacheConst[] : float
-*GradMovieCacheConst[] : float
+MFuno(in Feat : int, in Mov : int, in Users : int, in TS : int, in VS : int, in VS : int, in QS : int)
+~MFuno()
+SetPreferences(in FIR : float, in CIR : float, in beta : float, in delta : float, in maxIt : int, in minIt : int)
+InitializeFeatures()
+predictRating(in uv : float, in mv : float, in c : float, in d : float, in cache : float, out &out : float) : float
+predictRatingWoConst(in uv : float, in mv : float, in cache : float, out &out : float) : float
+FeatureUpdate()
+MovieConstFeatureUpdate(in &d : float, in &twoerr : float, out &out : float)
+UserConstFeatureUpdate(in &c : float, in &twoerr : float, out &out : float)

MFuno
+MFuno(in Feat : int, in Mov : int, in Users : int, in TS : int, in VS : int, in VS : int, in QS : int)
+~MFuno()
+SetPreferences(in FIR : float, in CIR : float, in beta : float, in delta : float, in maxIt : int, in minIt : int)
+InitializeFeatures()
+MovieFeatureUpdate(in &mv : float, in &uv : float, in &twoerr : float, out &out : float)
+UserFeatureUpdate(in &mv : float, in &uv : float, in &twoerr : float, out &out : float)
+MovieConstFeatureUpdate(in &d : float, in &twoerr : float, out &out : float)
+UserConstFeatureUpdate(in c : float, in twoerr : float, out out : float)
+predictRating(in uv : float, in mv : float, in c : float, in d : float, in cache : float, out &out : float) : float
+predictRatingWoConst(in uv : float, in mv : float, in cache : float, out &out : float) : float

MFduo
-***CoeffMatrix[][][] : float
+MFduo(in Feat : int, in Mov : int, in Users : int, in TS : int, in VS : int, in VS : int, in QS : int)
+~MFduo()
+SetPreferences(in FIR : float, in CIR : float, in beta : float, in delta : float, in maxIt : int, in minIt : int)
+InitializeFeatures(in randi : int, in slope : float, in constant : float)
+MovieFeatureUpdate(in &mv : float, in &uv : float, in &twoerr : float, out &out : float)
+UserFeatureUpdate(in &mv : float, in &uv : float, in &twoerr : float, out &out : float)
+MovieConstFeatureUpdate(in &d : float, in &twoerr : float, out &out : float)
+UserConstFeatureUpdate(in c : float, in twoerr : float, out out : float)
+predictRating(in uv : float, in mv : float, in c : float, in d : float, in cache : float, out &out : float) : float
+predictRatingWoConst(in uv : float, in mv : float, in cache : float, out &out : float) : float
+CMUpdate00(in prod : float, in uv : float, in mv : float, out &out : float)
+CMUpdate01(in prod : float, in uv : float, in mv : float, out &out : float)
+CMUpdate10(in prod : float, in uv : float, in mv : float, out &out : float)
+CMUpdate11(in prod : float, in uv : float, in mv : float, out &out : float)
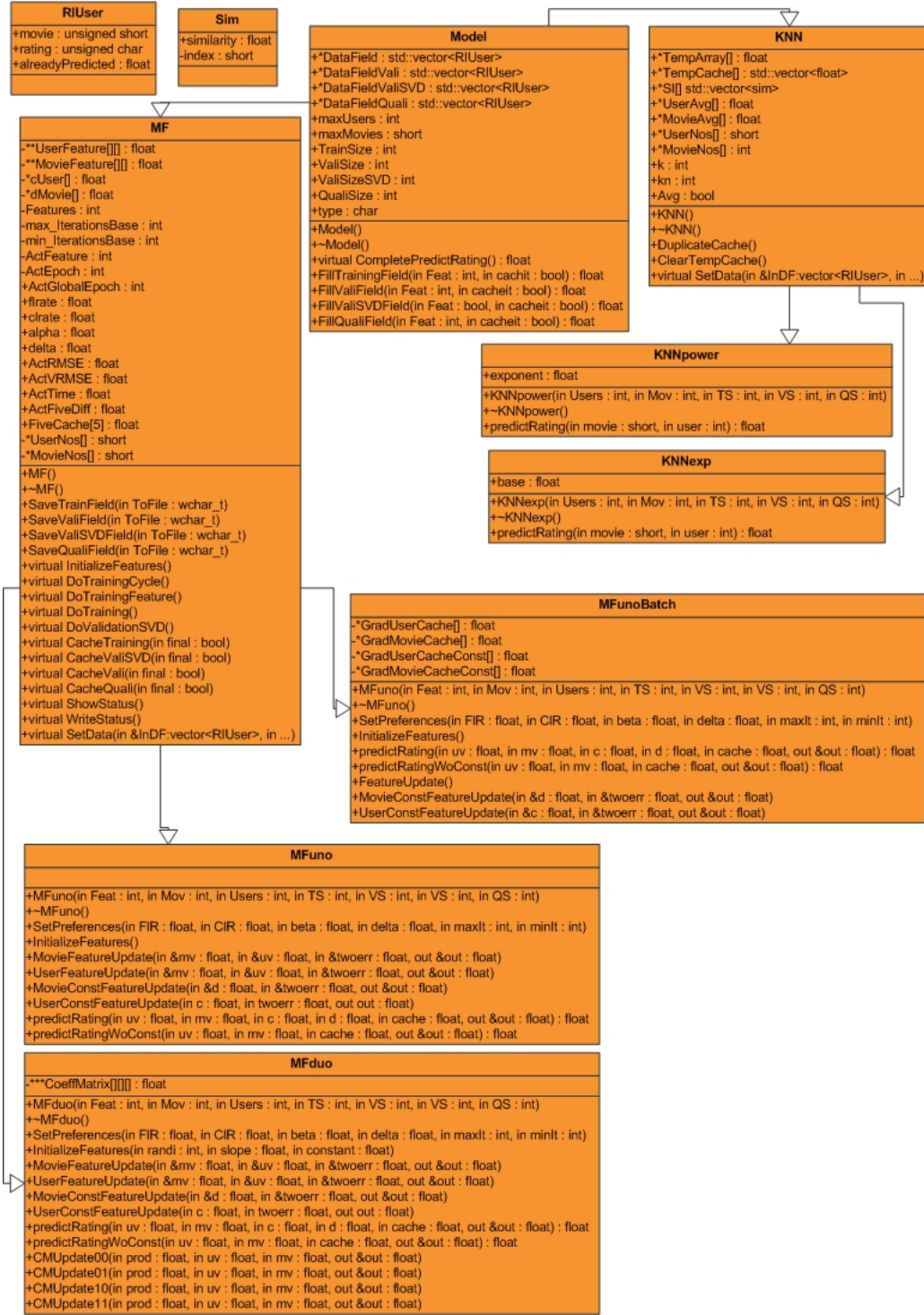
Figure A.1.: Basic class diagram of the implementation

Furthermore the following data structures (structs) are available.

- **RIUser:** Saving one data item from the point of a user. It consists of.
  - **movie:** An unsigned short indicating the MovieID the user watched.
  - **rating:** A char indicating the rating the user gave to the movie.
  - **alreadyPredicted:** A float saving every prediction any algorithm already did. This is necessary for concatenating models.

- **Sim:** A struct indicating the similarity of two movies.
  - **similarity:** A float, indicating the strength of similarity.
  - **index:** Indicating the second MovieID.

As can be seen the actual way of storing the similarity matrix might be interesting. The datastructure is an array of vectors. The size of the array is equivalent to the movies available (in case of the Netflix Prize those are 17770 movies). The vectors carry the above explained structs called Sim. They are ordered by similarity and must be stored in this order into the datafield. The index field tells the MovieID of the movie that is similar to the first one that is indicated by the dimension of the array. On the DVDs we supply each row of the similarity (for each movie) ordered by similarity.

The general datafields for Training, Validation and Qualification are stored as an array of vectors with a size that is equivalent to the number of users. The vectors carry RIUser-structs. Each struct explains one rating instance. It would also be possible to save it in the form of an array of a depth that is equivalent to the number of movies. However, it turned out that the approach of a long array and short vectors works faster than the opposite one.

Since we do not require specific access to the items of the datasets and datefields, neither for the similarity matrix nor for the rating matrix, we can save it like this. However, in case we would want to use an algorithm that specifically needs to access single fields of one of the matrices we would not be able to do this with these datafields. A workaround would be to implement

a binary search for specific instances within the fields. However, that would be extremely slow compared to a native approach.

A.2.1. Creating and filling datafields

As already mentioned datafields are nothing more than an array of vectors of RIUser structs. To create such an array in C++ one will require the header file vector.h. The code to create such an array is given below.

```
const int maxUsers = 480189;
// DF for Training
vector<RIUser> DataField[maxUsers];
// DF for Validation of MF method
vector<RIUser> DataFieldValiSVD[maxUsers];
// DF for big validation set
vector<RIUser> DataFieldVali[maxUsers];
// DF for Qualification
vector<RIUser> DataFieldQuali[maxUsers];
```

In the above example we created four datasets. We will need all those datasets for a MF model. To fill these datasets we require a pointer to them. These pointers will also be used to link the model objects to the datafields. In this way we can ensure that each model uses the same datafield. This has two reasons: Firstly, duplications of the memory fields would require further memory and they are already very large; Secondly, in this way we can easily concatenate models since we can save the already predicted values in the RIUser structs field "alreadyPredicted". To fill a datafield with data we do require a helper-object. The method for filling a training data field is different from the ones for the validation and qualification data fields. This is due to the fact that when filling the training datafield we also create some other arrays. Those arrays are the arrays UserNos and MovieNos, saving the number of ratings per user and per movie as well as the array UserAvg and MovieAvg, saving the average ratings per user and per movie. Let us first define those arrays and also the pointers to them.

A. Programming

```
const int maxMovies = 17770;

// define the arrays
short UserNos[maxUser];
int MovieNos[maxMovies];
float UserAvg[maxUsers];
float MovieAvg[maxMovies];

// define the pointers
short* PointerUsers = UserNos;
int* PointerMovies = MovieNos;
float* PointerUserAvg = UserAvg;
float* PointerMovieAvg = MovieAvg;
```

Now let us definet the pointers that point to the datafields.

```
vector<RIUser>* PointerTrain = DataField;
vector<RIUser>* PointerValiSVD = DataFieldValiSVD;
vector<RIUser>* PointerVali = DataFieldVali;
vector<RIUser>* PointerQuali = DataFieldQuali;
```

To fill the datafields with data we will require a string variable to point to the csv-file that saves the data field. The csv-file is required to have four columns with the first column being the MovieID, the second column being the UserID, the third column being the already predicted value (the cache) and the fourth column being the actual rating. In case of the qualifications set the fourth column will be all zero. We define the path variables by

```
string PathTrain =
(string)"G:\\netflix\\DataFields\\Train-empty.csv";
string PathValiSVD =
(string)"G:\\netflix\\DataFields\\ValiSVD-empty.csv";
string PathVali =
(string)"G:\\netflix\\DataFields\\Vali-empty.csv";
```

```
string PathQuali =
(string)"G:\\netflix\\DataFields\\Quali-empty.csv";
```

Train-empty.csv is a csv-file where the cached values are all zero. Those files
are given on the DVDs and can be used anytime. As already mentioned we
will require a helper-object to fill the datafields. The following code segment
shows how to create such an object and how to fill the training dataset.

```
Helper HelperObject;
int TrainSize = HelperObject.LoadDataFieldTrain(*PointerTrain,
PathTrain.c_str(), maxUsers, maxMovies, *PointerUsers,
 *PointerMovies, *PointerUserAvg, *PointerMovieAvg);
```

The method LoadDataFieldTrain of the Helper HelperObject returns an in-
teger, which is indicating the size of the training set. This is important for
later prediction algorithms (for example for computing the RMSE). Now we
can fill the other datafields by executing the following code.

```
int ValSizeSVD = HelperObject.LoadDataField(*PointerValiSVD,
PathValiSVD.c_str(), maxUsers);
int ValSize = HelperObject.LoadDataField(*PointerVali,
PathVali.c_str(), maxUsers);
int QualiSize = HelperObject.LoadDataField(*PointerQuali,
PathQuali.c_str(), maxUsers);
```

In case that we later want to train a KNN method we have to read in
and store the similarity matrix as well. We can do this by defining yet
again an array of vectors and filling it up with the help of the helper-object.
The way a similarity is saved on the harddisk is special however. As we
already mentioned we store the similarity vector of each movie seperatly.
Those vectors are ordered by similarity. We save the numerator and the
denumerator in seperate files. In this way we can leave the decission whether
to use single or double precision to the programmer. A third file gives the
MovieIDs that are connected to the similarities. To read the data we have to
provide the helper-object with the paths to the folders where those files lie in.

The following code defines those paths. Each folder consists of 17770 files, one for each movie. The filenames just consist of the MovieIDs as integers.

```
string DirNum =
(string)"G:\\netflix\\SimilarityMatrix\\Numerator";
string DirDen =
(string)"G:\\netflix\\SimilarityMatrix\\Denominator";
string DirInd =
(string)"G:\\netflix\\SimilarityMatrix\\Indexes;
```

Now we can define the datafield and read in the similarities. We have to define the maximum number of similar movies to read in, since we do not want to store the complete similarity matrix. This is done with the help of the integer number maxN.

```
vector<Sim> SI[maxMovies];
vector<Sim> *PointerSI = SI;
int maxN = 1500;
HelperObject.ReadSimilarityMatrix(DirNum, DirDen, DirInd, maxN,
*PointerSI, maxUsers, maxMovies);
```

Now all datafields are filled with data and we can actually move on to defining some prediction models. One should make sure that the helper-object stays in memory since we will need it later to save the datafields to a file.

A.2.2. Defining and training models

At this point we want to show how to define a prediction model and how to train it. We will define two methods, a Nearest Neighbor method and a Matrix Factorization, starting with the later. In case those models are linked to the same datafields they are already being concataneted when training the second one.

Matrix Factorization Models

```
int Features = 164;
MFuno aMF(Features, maxMovies, maxUsers, TrainSize, ValSizeSVD,
ValSize, QualiSize);
```

The MFs used here will use 164 features and an internal polynomial kernel of $r = 1$ (no kernel), thus it is a MFuno (uno for $r = 1$). We call the SetData method of the model to link datafields to it.

```
aMF.SetData(*PointerTrain, *PointerValiSVD, *PointerVali,
 *PointerQuali, *PointerUsers, *PointerMovies);
```

Now we will initialize the feature matrices. This method does not require any parameters, since a MF with internal polynomial kernel of $r = 1$ is always initizalied to 0.1 for the feature matrices and 0.9 for the constant feature vectors. However, a MF with a higher internal polynomial kernel will require three parameters: The initialization value for the random number generator, a slope and a constant to compute the random numbers. The following code will initialize the matrices.

```
aMF.InitializeFeatures();
```

Now as a last step we need to set the preferences of the model. These include the learning rates α and χ as well as the regularization parameters β and δ. Also we need to tell the algorithm the base values for the maximum and minimum iterations. We can do all this by the following code.

```
float alpha = 0.00025;
float chi = 0.0001;
float beta = 0.0225;
float delta = 0.0225;
int max_IterationBase = 85;
int min_IterationBase = 75;

aMF.SetPreferences(alpha, chi, beta, delta,
max_IterationBase, min_IterationBase);
```

A. Programming

After all of these steps we can start training.

```
aMF.DoTraining();
```

This will cause the model to go to training mode. Depending on the number of features and maximum iterations as well as the speed of the computer the algorithm runs on, this can take a few minutes to several days. After training finishes we will write all predictions into the cache of the datafields. We can do this by running the following code.

```
aMF.FillQualiField(Features, true);
aMF.FillValiField(Features, true);
aMF.FillValiSVDField(Features, true);
aMF.FillTrainingField(Features, true);
```

These methods will make the algorithm run through the entire datafields and predict them. The second boolean parameter decides whether the method should write the prediction to the cache. This flag needs of course to be true in most cases. However, sometimes it can be helpful to set it to false. For example when trying out multiple predictions with a KNN model.

Nearest Neighbor Models

Defining NN methods is quite similar to defining MF methods. First we create a KNN object. We use the class KNNpower and we want the ratings to be normalized by the movie average. This can be done by the following code.

```
KNNpower aKNN(maxMovies, maxUsers, TrainSize,
ValSize, QualiSize);
aKNN.Avg = true;
```

Also we set the number of neighbors ($k = 35$) and the maximum number of neighbors available (maxN).

```
aKNN.k = 35;
aKNN.kn = maxN;
```

100

Now we link the datafields to the object. We will link the same datafields to the KNN model as we linked to the MF model before. In this way we will automatically concatenate these two models because the KNN algorithm will reuse the cached values of the MF algorithm and therefore only try to predict the residuals the MF model created.

```
aKNN.SetData(*PointerTrain, *PointerValiSVD, *PointerVali,
*PointerQuali, *PointerSI, *PointerUsers, *PointerMovies,
*PointerUserAvg, *PointerMovieAvg);
```

We want to use a weighting function $\omega\left(s\right) = s^{2.5}$, so we set

```
aKNN.exponent = 2.5;
```

KNN algorithms do not require any training, so they do not offer any DoTraining() method. At this point we can already fill the datafields by executing the following code. It is important to duplicate the cache before filling the datafields, otherwise the algorithm would use already cached values for the prediction as well.

```
aKNN.DuplicateCache();
aKNN.FillQualiField(Features, true);
aKNN.FillValiField(Features, true);
aKNN.FillValiSVDField(Features, true);
aKNN.FillTrainingField(Features, true);
```

It does not matter to which integer number "Features" is set to because a KNN algorithm does not make use of any features.

A.2.3. Saving datafields

At this point we are finished training both predictors and we can now write the datafields to file, so we can read them in later again.

```
HelperObject.SaveDataField(*ZeigerTrain,
"G:\\netflix\\DataFields\\Train-MF_KNN.csv", maxUsers);
```

```
HelperObject.SaveDataField(*ZeigerValiSVD,
"G:\\netflix\\DataFields\\ValiSVD-MF_KNN.csv", maxUsers);
HelperObject.SaveDataField(*ZeigerVali,
"G:\\netflix\\DataFields\\Vali-MF_KNN.csv", maxUsers);
HelperObject.SaveDataField(*ZeigerQuali,
"G:\\netflix\\DataFields\\Quali-MF_KNN.csv", maxUsers);
```

A.3. Compiler Options

For our implementation we used Visual Studio and as a result the Microsoft Compiler for C++. For this compiler (and for all the others) there are a lot of flags to set, which will significantly improve the performance of the program. In our implementation we used nearly the default setup of Visual Studio. However, we changed a few compiler flags, which gave a dramatic boost in prediction performance. The compiler flags changed are listed below.

- **O2:** The O2 flag will optimize the program for speed. Using this flag might result in a larger execution file (exe).

- **Ob1:** Will only treat functions as inline when they are actually marked with _ _inline. The program makes use of the _ _inline extensions whenever it is possible to improve performance with the usage of these macro extensions.

- **Oi:** Activates internal functions, which speeds up execution time but creates larger code.

- **Ot:** Preferes faster code rather than small code.

- **arch:SSE2:** Enables the ussge of the Streaming SIMD Extensions. Although not used in the program itself, this may improve perfomance once specific functions are programmed. You will need a CPU that supports the necessary instruction sets.

- **fp:fast:** Changes the floating point model from precise to fast.

A.3.1. Defining non-linear Matrix Factorizations

There are several estimators one can define. Some of them differ a little in the way how they are created and initialized. Matrix Factorizations with internal kernels greater than $r = 1$ are created the same way as MF with a kernel of $r = 1$. For example, to create a MF with an internal kernel of $r = 2$ we use the following command:

```
int Features = 164;
MFduo aMFduo(Features, maxMovies, maxUsers,
TrainSize, ValSizeSVD, ValSize, QualiSize);
```

To connect the data we can use the SetData method, which is accessed exactly the same way as with the MF using an internal kernel of only $r = 1$. However, the InitializeFeatures-method is a little different. To intialize the feature vectors as well as the weights of the polynomial components $\underline{\Theta}$ we can use the following command:

```
int randomizer = 12345;
float slope = 3.0;
float constant = 1.0;
aMFduo.InitializeFeatures(randomizer, slope, constant);
```

The items of $\underline{\Theta}$, which are given as $\vartheta_{1,2,\dots,r}$, are then created randomly following the rule

$$\vartheta_{1,2,\dots,r} = slope \times rand + constant$$

and *rand* is a random number taken from a uniform distribution between 0 and 1.

A.3.2. Blending different predictions

To blend different predictions the user will need a predicted validation set and a predicted qualification set as well as MatLab. We assume that those validaton and qualificaton sets are given as a csv-file like the ones the SaveDataField-methods of the framework will return. MatLab can easily read in these csv-files, they will be given in matrix form. The third column of each matrix

A. Programming

will carry the predicted values. For this example we will use three different predictions. The names of these matrices are Vali1, Vali2, Vali3, Quali1, Quali2 and Quali3. Also there are two vectors z1 and z2 given. Those vectors describe the indices of the validation sets to split them into V_1 and V_2. We will use V_2 for validation and V_1 for training. To compute the best linear combination of all predictions we first have to define the matrices P, PVal and Q. All matrices carry the predictions of one model per column. Also we create R and RVal which are the real ratings of the different validation set splits. We can find those ratings in the fourth column of the validation matrices.

```
P(:,1) = Vali1(z1,3);
P(:,2) = Vali2(z1,3);
P(:,3) = Vali3(z1,3);
PVal(:,1) = Vali1(z2,3);
PVal(:,2) = Vali2(z2,3);
PVal(:,3) = Vali3(z2,3);
Q(:,1) = Quali1(:,3);
Q(:,2) = Quali2(:,3);
Q(:,3) = Quali3(:,3);
R = Vali1(z1,4);
RVal = Vali1(z2,4);
```

From this point it is very easy to blend the predictions. One just has to call the OptimalSolution-function.

```
[ newPred frmse lams QPred] =
OptimalSolution( P, PVal, R, RVal, Q );
```

The algorithm will output some information about the different validation sets and about the final RMSE. The returned values are:

- **newPred:** A vector giving the final combination of P.

- **frmse:** The final RMSE of newPred.

104

- **lams:** The weights for the different predictions.

- **QPred:** The final prediction for the qualification file of Netflix.

A. Programming

B. Further Experiments and Plots

B.1. Matrix Factorization

Figure B.1 shows an interesting picture. While for the validation set (Figure 6.1) the regularization parameters actually show some effect, they do not do at all on the training set. The higher the parameters β, δ are set the worse the RMSE on the training set is. The observation that the RMSE on the validation set improves when using well set parameters β, δ while it does not improve the performance on the training set obviously points out that those parameters can supress overtraining substantially.

Figure B.2 plots the RMSE on validation against the RMSE on the training set. The last few epochs were captured because only those final training steps actually differ substantially. It is clearly seen that the regularization parameters have an effect on overtraining and cause an impact on the error of the validation set.

Figure B.1.: Incremental MF performance on training set. Epochs from 5000 to 30000. Maximum of 164 features. Learning rates fixed at $\alpha = 0.00025$ and $\gamma = 0.0001$. The higher the regularization parameters, the lower the performance.
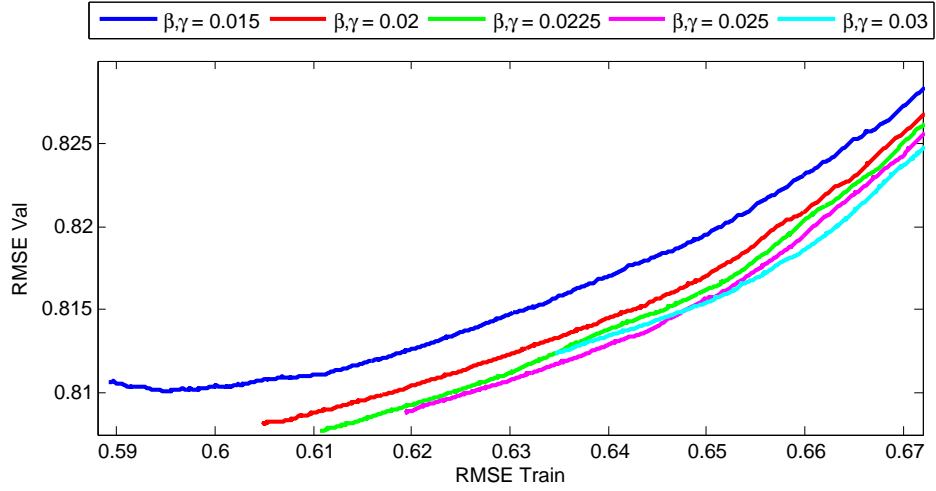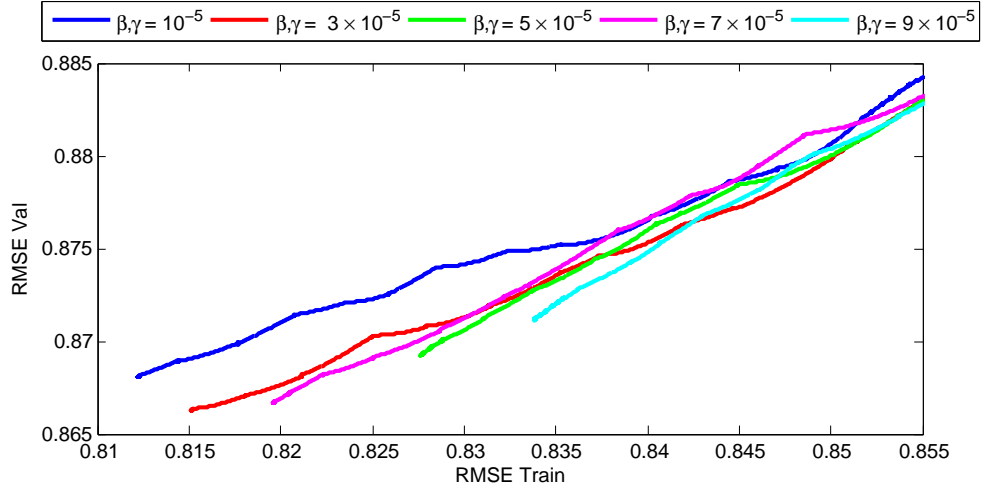

Figure B.2.: Approximately the final 5000 training epochs of different incremental MFs. RMSE on training set plotted against the RMSE on the validation set.

B.2. Nearest Neighbor Methods

Table B.2 gives the RMSEs of different NN methods without normalization by movie mean. In comparison to Table 6.3 it becomes obvious that the method using the normalization by movie mean performs a lot better. We therefore can only recommend this method. For the NN approach without normalization by movie mean the combination of a weighting function of $\omega(s) = s^5$ and $k = 20$ neighbors offers the best performance. The exponential weighting functions perform worse than the weighting functions of form $\omega_r(s) = s^r$.

Table B.1.: RMSEs for different NN methods without normalization by movie mean.

$\omega(s)$	$k = 10$	$k = 15$	$k = 20$	$k = 25$	$k = 30$	$k = 35$	$k = 40$	$k = 45$
$s^{0.5}$	0.8956	0.8963	0.8987	0.9022	0.9055	0.9086	0.9115	0.9141
$s^{1.0}$	0.8927	0.8929	0.8952	0.8983	0.9015	0.9044	0.9071	0.9097
$s^{1.5}$	0.8904	0.8900	0.8920	0.8947	0.8976	0.9003	0.9029	0.9053
$s^{2.0}$	0.8887	0.8878	0.8893	0.8917	0.8943	0.8968	0.8992	0.9014
$s^{2.5}$	0.8877	0.8862	0.8873	0.8894	0.8917	0.8939	0.8961	0.8981
$s^{3.0}$	0.8871	0.8851	0.8859	0.8876	0.8896	0.8916	0.8936	0.8954
$s^{3.5}$	0.8868	0.8844	0.8849	0.8864	0.8881	0.8899	0.8917	0.8933
$s^{4.0}$	0.8868	0.8841	0.8843	0.8855	0.8870	0.8886	0.8902	0.8917
$s^{4.5}$	0.8871	0.8840	0.8840	0.8849	0.8863	0.8877	0.8891	0.8905
$s^{5.0}$	0.8880	0.8842	0.8839	0.8846	0.8858	0.8870	0.8883	0.8896
0.5^s	0.8996	0.9010	0.9033	0.9071	0.9102	0.9136	0.9165	0.9191
1.0^s	0.8985	0.9002	0.9026	0.9063	0.9095	0.9126	0.9155	0.9183
1.5^s	0.8982	0.8998	0.9019	0.9058	0.9090	0.9121	0.9150	0.9177
2.0^s	0.8980	0.8993	0.9016	0.9054	0.9086	0.9118	0.9147	0.9174
2.5^s	0.8978	0.8990	0.9013	0.9051	0.9083	0.9116	0.9144	0.9171
3.0^s	0.8976	0.8988	0.9011	0.9048	0.9081	0.9113	0.9142	0.9169
3.5^s	0.8975	0.8985	0.9009	0.9046	0.9079	0.9111	0.9140	0.9167
4.0^s	0.8973	0.8984	0.9008	0.9044	0.9077	0.9110	0.9138	0.9165
4.5^s	0.8972	0.8982	0.9006	0.9043	0.9076	0.9108	0.9137	0.9164
5.0^s	0.8971	0.8981	0.9005	0.9041	0.9074	0.9107	0.9135	0.9162

Figures B.3 and B.4 visualize the performance of the different NN methods without normalization. The figures show a non-surprising resemblance to the
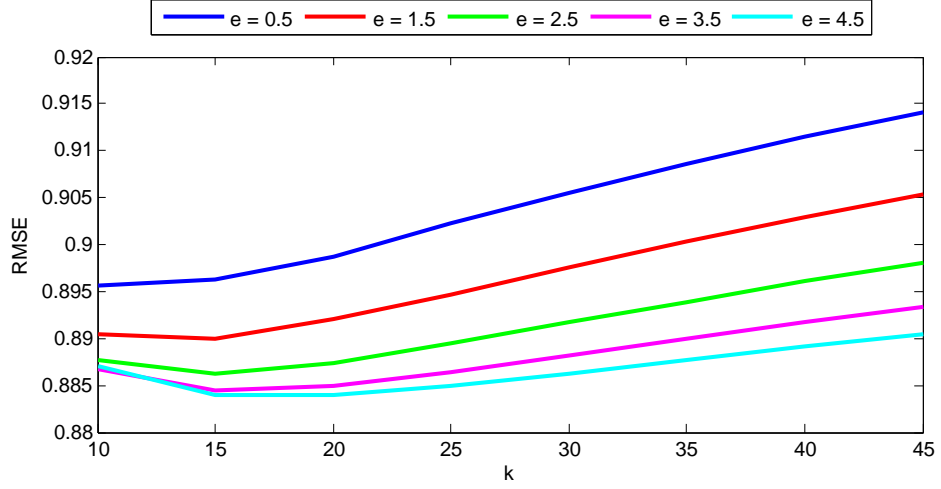
Figure B.3.: RMSEs of Nearest Neighbor algorithm with different number of neighbors. Weighting functions of form $\omega_e(s) = s^e$.

Figures 6.3 and 6.4.

B.3. Movielens Dataset

B.3.1. Matrix Factorization

Figure B.5 plots the RMSE of the training set of the Movielens dataset against the training epochs. We can see the same picture as on the validation set: For all different regularization parameters the improvement on the training set stops for some time and then begins again. This is due to the fact that during this period of training there were just too few minimal iterations per feature. This resulted in the situation that training on these feature stopped before it could actually start improving the overall RMSE. Different from the Netflix Prize dataset, however, is that in this plot different colored lines actually do cross. That means that the different regularization parameters have an effect on the performance of the estimators on the training set, at least on first sight. A second look reveals that those crossings actu-
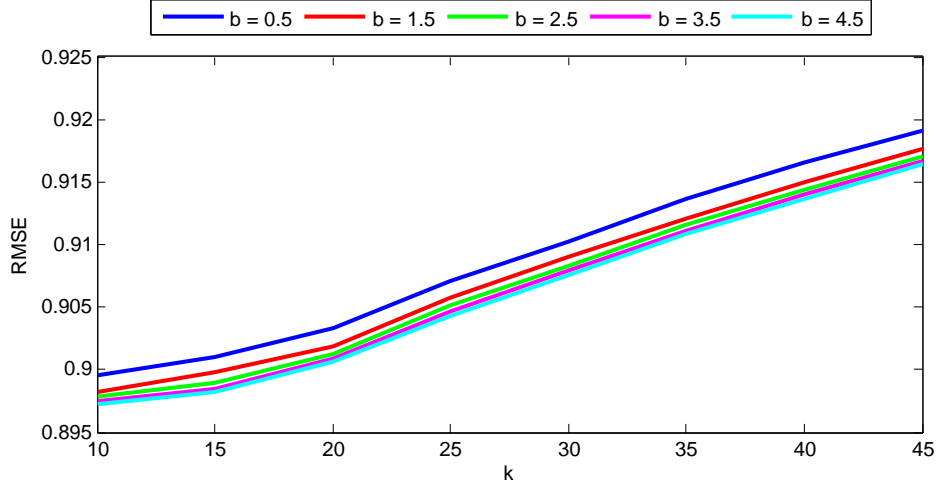
Figure B.4.: RMSEs of Nearest Neighbor algorithm with different number of neighbors. Weighting functions of form $\omega_b(s) = b^s$.

ally might only be related to the above described issue of too few training iterations per feature.

Figure B.6 plots the last few training epochs of the MFs on the Movielens. We can find the RMSE on the training set on the x-axis and the RMSE of the validation set on the y-axis. One can clearly see how regularization parameters are able to improve the results.
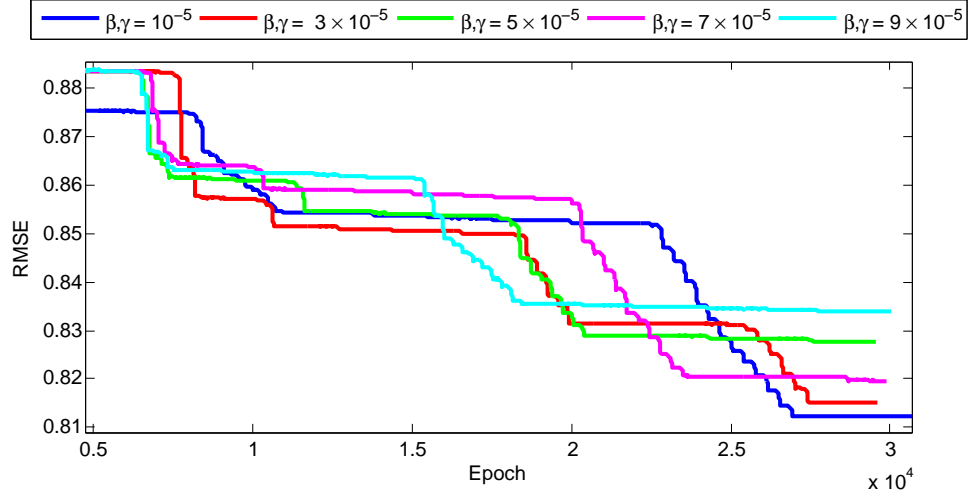
Figure B.5.: Batch learning MF performance on training set of the Movielens dataset. Epochs from 5000 to 30000. Maximum of 164 features. Learning rates fixed at $\alpha = 0.00025$ and $\gamma = 0.0001$.

Figure B.6.: Approximately the final 5000 training epochs of different batch learning MFs on the Movielens dataset. RMSE on training set plotted against the RMSE on the validation set.

B.3.2. Nearest Neighbor Methods

Table B.3.2 gives the RMSEs of the different NN methods, which did not use
the normalization by movie mean. The used methods differ in the weighting
function used and the number of neighbors. A combination of a weighting
function of $\omega(s) = s^{4.5}$ and $k = 20$ neighbors offers the best performance.
Similar to the dataset of the Netflix Prize, the NN algorithms using an ex-
ponential weighting function perform worse than the ones using a power
weighting function. In comparison to the NN methods using the normaliza-
tion by movie mean a clear picture evolves: The usage of normalization is
improving the RMSE of the NN methods for both datasets, the Netflix Prize
dataset and the Movielens dataset. We therefore recommend to use such a
normalization, also because it is easy to implement and the time for comput-
ing the predictions does only improve slightly. We forgo on visualizing the
performance of the NN methods without normalization. Such plots can be
easily extracted from Table B.3.2.

Table B.2.: RMSEs for different NN methods on the Movielens dataset without normalization by movie mean.

$\omega(s)$	$k=10$	$k=15$	$k=20$	$k=25$	$k=30$	$k=35$	$k=40$	$k=45$
$s^{0.5}$	0.9112	0.9088	0.9103	0.9123	0.9149	0.9176	0.9204	0.9229
$s^{1.0}$	0.9095	0.9067	0.9077	0.9094	0.9118	0.9142	0.9168	0.9191
$s^{1.5}$	0.9080	0.9047	0.9053	0.9067	0.9090	0.9109	0.9132	0.9153
$s^{2.0}$	0.9067	0.9029	0.9032	0.9042	0.9060	0.9079	0.9099	0.9118
$s^{2.5}$	0.9058	0.9015	0.9013	0.9020	0.9035	0.9051	0.9069	0.9085
$s^{3.0}$	0.9053	0.9005	0.8999	0.9002	0.9014	0.9028	0.9043	0.9057
$s^{3.5}$	0.9051	0.8999	0.8988	0.8989	0.8997	0.9008	0.9021	0.9033
$s^{4.0}$	0.9053	0.8996	0.8982	0.8979	0.8985	0.8993	0.9004	0.9014
$s^{4.5}$	0.9058	0.8998	0.8980	0.8974	0.8978	0.8983	0.8992	0.9000
$s^{5.0}$	0.9067	0.9003	0.8982	0.8974	0.8975	0.8979	0.8985	0.8991
0.5^s	0.9136	0.9116	0.9135	0.9158	0.9187	0.9218	0.9249	0.9276
1.0^s	0.9131	0.9109	0.9129	0.9152	0.9181	0.9210	0.9241	0.9269
1.5^s	0.9128	0.9108	0.9126	0.9149	0.9177	0.9207	0.9237	0.9264
2.0^s	0.9127	0.9105	0.9123	0.9146	0.9174	0.9204	0.9234	0.9261
2.5^s	0.9125	0.9104	0.9122	0.9144	0.9172	0.9201	0.9232	0.9258
3.0^s	0.9124	0.9102	0.9120	0.9143	0.9171	0.9200	0.9230	0.9256
3.5^s	0.9123	0.9101	0.9119	0.9141	0.9169	0.9198	0.9228	0.9254
4.0^s	0.9122	0.9100	0.9118	0.9140	0.9168	0.9197	0.9227	0.9253
4.5^s	0.9121	0.9099	0.9117	0.9139	0.9167	0.9195	0.9225	0.9252
5.0^s	0.9121	0.9099	0.9116	0.9138	0.9166	0.9194	0.9224	0.9250

C. On the DVDs

There are two DVDs that come with this thesis. Both DVDs carry the datasets of the Netflix Prize as well as the datasets of the Movielens project. Furthermore they also provide the different C++ and MatLab codes. The directory structure is as follows:

- **DVD1, Training and Validation:** This DVD carries the training dataset $T \setminus V$ (97465045 rating instances) as well as the big validaton set $V \setminus V_{MF}$ (2924999 rating instances) and the small validation set V_{MF} (90463 rating instances), which we use to determine when a MF should stop training.

  - **Folder SourceDataSets:** Holds all the necessary datasets. These datasets can be loaded using the LoadDataFieldTrain or Load-DataField methods of a HelperObject.

  - **Folder ValidationSplit:** Gives the vectors indicating the indices that will seperate the validation set V into V_1 and V_2. The vectors are given as .mat-files and .csv-files.

  - **Archive MovieSimilarities.zip:** Contains the similarity matrix S_κ. The similarities are given by enumerator and denominator of the fraction indicating the similarity. They are already ordered for each movie, highest similarity first.

    * **Folder coocc_ordered:** The enumerators of the fractions describing the rows of the ordered similarity matrix.

    * **Folder total_ordered:** The denominators of the fractions describing the rows of the ordered similarity matrix.

* **Folder indexes_ordered:** The MovieIDs which are connected to the respective similarities.

- **DVD2, Training only and Movielens dataset:** This DVD carries the training dataset $T \setminus V_{MF}$ without the big validation set but with the validation set for the MF methods removed.

  - **Folder SourceDataSets:** Holds the training set (Train-empty.csv).

  - **Folder MovieLens:** The Movielens dataset

    * **Folder SourceDataSets:** Movielens Datasets.

    * **Folder MovieSimilarities:** Similarity Matrix of the Movielens dataset (folder structure is not further evaluated, see DVD1).

  - **Archive MovieSimilarities.zip:** Similarity Matrix (folder structure is not further evaluated, see DVD1).

  - **Folder CPPcode:** Contains all the C++ code.

  - **Folder MATLABcode:** Contains all MatLab code.

List of Tables

List of Figures

List of Figures