Exploiting user preference similarity transitivity in nearest neighbour recommender algorithms

Johan Malmberg Magnus Öinert

August 3, 2015.

LUND UNIVERSITY

Master's thesis work carried out at
the Department of Mathematical Statistics.

Supervisors: Prof. Søren Vang Andersen, Dr. Stefán I. Aðalbjörnsson

Examiner: Prof. Andreas Jakobsson

Abstract

This thesis explores the Nearest Neighbour recommender algorithm and a proposed way of recomputing the elements of a set, containing values from a similarity metric. In essence, this extends the neighbourhood used in the algorithm by making use of close neighbours' neighbours. This proposal is motivated by the hypothesis that high user preference similarity is transitive. A programme was implemented in Matlab, and one of the MovieLens data sets was used. Three experiments were run. These indicated that further exploration of this property and the proposed methods could be of interest. The final experiment was run on the simplest form of a Nearest Neighbour recommender, with the added re-computational step, and suggests that there is a small, but statistically significant, improvement due to this new step (for one of the proposed methods) confirming the hypothesis. The improvement of the errors of estimated rating predictions were in the order of magnitude 0.1 %.

Keywords: Recommender systems, Recommender algorithms, *kNN*, Nearest Neighbour, MovieLens, Collaborative Filtering.

Acknowledgements

We would like to thank our research group at Lund University; Dr. Stefán Aðalbjörnsson, Magnus Örn Berg, Prof. Søren Vang Andersen, Prof. Andreas Jakobsson, Johan Svärd, Ted Kronvall and Simon Burgess, for interesting discussions and valuable feedback. We would further like to thank QuanoX S.à.r.l. for a valuable business cooperation. Additionally, we thank GroupLens at the University of Minnesota for conveniently providing much needed data. Finally, we say thank you to our beloved friends and family, for having helped us get to where we are.

Contents

Chapter 1
Introduction

Two roads diverged in a wood, and I-
I took the one less traveled by,
And that has made all the difference.

- Robert Frost, *The Road Not Taken*

"Recommender Systems are software tools and techniques providing suggestions for items to be of use to a user" [2].

Recommender systems are found on numerous websites on the Internet. Companies like Amazon, CDON, Youtube, Google, and Netflix use recommender systems. Whenever you see a "people who bought this also bought: " or "you might also like this: " on a web page those are recommendations based on your previous actions; may it be a view on Youtube or a deal on Amazon. By incorporating what a user already has seen, bought or rated (referred to as a user having consumed an item) the system is designed so that it can predict what a person will watch next, wants to buy or will rate.

Recommender Systems are mostly implemented to:

- increase user satisfaction,
- better understanding what a user wants,
- increase the number of items consumed,
- exploit the full inventory [2].

This chapter introduces recommender systems as a concept. What makes a recommendation good or bad is discussed, as well as an overview of neighbourhood based methods.

1.1 Introduction to thesis

Chapter 1 introduces what a recommender system is, with an explanation of notations, some history and older techniques. The Nearest Neighbour algorithm is also described in this chapter.

In the next chapter, chapter 2, a new algorithm is proposed that explores the possibilities of making use of the property of preference similarity transitivity in the Nearest Neighbour algorithm. The chapter describes different alterations of the new proposals that will be explored. Chapter 3 focuses on experiments conducted on the new proposals explained in chapter 2, also covering the data used in the process. In chapter 4 conclusions are drawn from the outcome of the experiments. In chapter 5 future work is discussed as to what could be further explored down this path.

1.2 Recommender systems

1.2.1 General definition of recommender systems

A recommender system's purpose is to make inference about the level of appreciation by a user of an item, that has not had any previous interaction, typically by generating useful personalised suggestions of some kind of items to users, based on the users' preferences [6]. The items in question could be songs, news articles, restaurants, movies, or as they often are: consumable products. A user is a human individual. The input to the recommender system is information of previous interactions between users and items, consisting of implicit consumption behaviour or explicit ratings. The outputs are lists, one for each user, of items that by a heuristical justification is believed to be appreciated by the users. Alternatively, or equivalently, the output can be a prediction of the consumption behaviour or ratings made by the users on the items. The focus in this thesis is rating estimation.

1.2.2 Notation

Items is the general term which denotes what the system recommends to users, e.g. books, songs, wines etc, and *Users* simply correspond to the individuals that receive recommendations, e.g. a personal account on a web based music streaming service. Ratings refers to what some of the users thinks of some of the items, in terms of appreciation, e.g. number of times a song has been played by a user or number of "stars" given as a rating to a movie by a user (implicit and explicit ratings respectively).

Let $\mathcal{U}$ be a set of users and $\mathcal{I}$ a set of items. We are given a set of ratings $\mathcal{R} \in \{\mathcal{S}\}^{|\mathcal{U}|\times|\mathcal{I}|}$ where $\mathcal{S}$ is the set of possible values a rating can take (scores). As the ratings $r \in \mathcal{R}$ exists for pairs of user and item, a rating by a user $u \in \mathcal{U}$ on an item $i \in \mathcal{I}$ is denoted r_{ui}.

To identify the subset of users that have rated an item i, we use the notation $\mathcal{U}_i$. Likewise, $\mathcal{I}_u$ represents the subset of items that have been rated by a user u. The similarity weight between a user u and user v, calculated by applying a similarity metric $s(u, v)$, is denoted w_{uv} or alternatively $w(u, v)$.

When designing a recommender algorithm, we essentially seek to create a function $f : \mathcal{U} \times \mathcal{I} \rightarrow \mathcal{S}$ that is trained on the set $\mathcal{R}$ and will be able to create a set of predicted

ratings, $\hat{\mathcal{R}}$, for those pairs (u, i) for which r_{ui} does not exist.

The ratings are often most represented as a matrix, where each column corresponds to an item, and each row a user. Typically this matrix is sparse, as it only has entries in those places that correspond to an already made rating,

$$\mathcal{R} = \overbrace{\begin{matrix} r_{u_1 i_1} & r_{u_1 i_2} & r_{u_1 i_3} \\ r_{u_2 i_1} & r_{u_2 i_2} & r_{u_2 i_3} \\ r_{u_3 i_1} & r_{u_3 i_2} & r_{u_3 i_3} \end{matrix}}^{Items} \Bigg\} Users \tag{1.1}$$

1.2.3 History of recommender systems

The earliest recommenders that were made, back in the early 90s, utilized information retrieval techniques [6]. Information retrieval was considered as a large database, with text based items, which was queried. By investing a lot of time indexing the database for meta information and content occurrence, the most adequate items could be returned. This demanded static content, or information that did not change often, which today could be considered more or less a search engine. Term frequency - inverse document frequency; the frequency by which a term appears in a text given the number of times it appears in the corpus, was often used to establish how relevant an item was [2].

Recommenders evolved from there into handling non static content, i.e., handling content changing from day to day such as newspaper articles [6]. To adapt to this, recommenders built preference profiles for the users that were static instead of the content. Thereby the cost changed from indexing the data to creating the profiles. These profiles could be matched with the new information, filtering the content. At first these profiles were done by hand, but as time passed it turned into a machine learning process where the algorithm created the profiles. This is a way of making recommendations using meta data and are thus content based recommenders, which is discussed below in section 1.2.4.

In turn this development of recommenders lead to what today is known as collaborative filtering; where a user's preferences are more elaborate than just a few key words. As it easier to know if content belongs to a topic but harder if it belongs to a certain taste, profiles thus became better at screening the data. Today collaborative filtering is the dominating method in the recommender system industry [2].

1.2.4 Techniques

Content-based recommenders

Content based filtering is the most intuitive approach to giving recommendations to a user [2]. This works by finding items with similar characteristics to that of those already rated by a user. This approach can predict fairly accurate recommendations to a user by simple mathematics. One way of doing this is by constructing a vector x_i for an item an informative schema is set up for that item describing its characteristics and the nature of that item. Similarly a vector x_u is set up and is assumed to contain information about each user, seen as a preference profile. The profile for a user can be updated whenever a new rating becomes available to the system.

Essentially content based methods need some human made classification and profiling of what an item is and creates a personality-containing vector of how these profiling elements are perceived to a user. Alternatively, this classification could be made by specific machine learning algorithms, for instance using audio analysis when classifying music.

Collaborative filtering

Where as content based algorithms are useful, these only pick up on the characteristics of a user and thus will recommend items similar to what has already been consumed [7]. A user will not ever be recommended non-related items. Another approach is preferred in order to increase the diversity, and the serendipity [1], of the recommender system.

Collaborative filtering algorithms, in their simplest form, presume nothing about the characteristics of the items or the users in the system [2]. The only input is a set of ratings and the output is personalised recommendations. This is the core purpose of a recommender system, that it can tailor suggestions depending on the receiver, aiming to do better than an unpersonalised top list.

Collaborative filtering essentially takes into account how items are rated by different kinds of users, and fills in the gaps by predicting the scores of the missing ratings so that these fit into the model. For collaborative filtering there are two main methods that are used, the neighbourhood- and the model-based methods. This thesis will focus on the neighbourhood based method and how to improve the existing algorithms.

The nearest neighbour is one of the earliest collaborative filtering algorithms [2, 7]. It is still popular due to its simplicity and sufficiently good performance. It is also cost efficient in terms of computations [2]. Furthermore, it has some additional benefits such as being able to justify its recommendations: "others also looked at". The nearest neighbour algorithm is discussed in detail in the next section.

1.3 The Nearest Neighbour-algorithm

One collaborative filtering algorithm, central to this thesis, is the Nearest Neighbour algorithm (*kNN*). Neighbourhood based algorithms work much in the same way as peoples own intuitive way of creating recommendations [2]. To create predictions for a user's appreciation of an item, one finds a neighbourhood of users that are similar in preference to the user in question, based on mutually rated items. The ratings made by the neighbours on the item in question can then be used to make a prediction. These algorithms filter through all the rating data and create predictions without knowing anything about the properties of the items, or anything about the user's preferences. The aim is to create recommendations that are better than just suggesting overall popular items - i.e., recommending items that would be rated highly, specific to a user.

[1] Serendipity refers to a user finding an interesting item, through a recommendation, that she or he would not have discovered otherwise.[2]

1.3.1 The standard algorithm

There are two main steps to creating recommendations in the *kNN*-algorithm. The first step is to calculate similarity values between users (or items, if it is item based) using the existing ratings. The next step is to create rating predictions, commonly using those similarity values together with the existing ratings. These steps are described in detail below. A user-based system is described, which is what is used in this thesis work (see section 1.3.2).

Similarity measure

To be able to choose a neighbourhood when estimating ratings, one needs a similarity measure. The similarity measure is used to compute a value that is an estimation of how similar two users are in preference personality (or items are in similarity, if the recommender is item-based).

There are several commonly used measures: cosine vector, Pearson correlation and inverse root mean squared difference (IRMSD) [2]. The cosine vector similarity (1.2), considers users' ratings as vectors in a high dimensional space, where the items are the dimensions, and calculates an (almost) analogous angle between these users in a subspace of mutually rated items. The similarity calculation becomes:

$$s(u,v) = \frac{r_u \cdot r_v}{\|r_u\|_2 \, \|r_v\|_2} = \frac{\sum\limits_{i \in \mathcal{I}_{uv}} r_{ui} r_{vi}}{\sqrt{\sum\limits_{i \in \mathcal{I}_u} r_{ui}^2 \sum\limits_{j \in \mathcal{I}_v} r_{vj}^2}}, \tag{1.2}$$

where $\|\cdot\|$ is the L_2-norm.

The Pearson correlation is the linear correlation between two users taking the effects of mean and variation into consideration (1.3), the Pearson correlation will give a similarity range between -1 and 1. This is different from the other metrics mentioned here, which could potentially have an improving effect on the prediction results.

$$s(u,v) = \frac{\sum\limits_{i \in \mathcal{I}_{uv}} (r_{ui} - \bar{r}_u)(r_{vi} - \bar{r}_v)}{\sqrt{\sum\limits_{i \in \mathcal{I}_u} (r_{ui} - \bar{r}_u)^2 \sum\limits_{j \in \mathcal{I}_v} (r_{vj} - \bar{r}_v)^2}}, \tag{1.3}$$

where $\bar{r}$ is the mean value of all ratings by user r.

The final metric, IRMSD (1.5), is the inverse of the root mean squared difference, with the alteration that only the common items are considered.

The root mean squared difference, is defined in (1.4). The similarity metric used in most of the work in this thesis is the inverted root mean squared difference with a slight modification (1.5),

$$d(u,v) = \sqrt{\frac{\sum_{i \in I_u \cap I_v} (r_{ui} - r_{vi})^2}{|I_u \cap I_v|}}, \tag{1.4}$$

$$s(u,v) = \begin{cases} 0 & if\ |I_u \cap I_v| = 0 \\ \frac{1}{d(u,v)+\epsilon} & else \end{cases}. \tag{1.5}$$

A term, ϵ, is added in the denominator, to avoid computational problems should the distance be zero. The value of ϵ was set to 1. This introduces a bias in the estimation of "true" similarity, but avoids the singularity. The magnitude of it does however not change the internal ordering of any user's neighbours. Should there be no mutually rated items, i.e $|I_u \cap I_v| = 0$, the similarity $s(u,v)$ was set to zero.

Many different similarity measures exists in the literature and the cosine vector is commonly used. However, the measures seems to be quite correlated and have similar performance effects [7, 10]. As mentioned above, the IRMSD has been the similarity measure mostly used in this thesis, partially because it is easy to work with non negative numbers. These similarities are used as weights in the algorithm, and a similarity $s(u,v)$ is denoted w_{uv}. These weights can be represented in a user-user matrix. An example is shown below (1.6).

$$\mathbf{W} = \overbrace{\begin{matrix} w_{u_1u_1} & w_{u_1u_2} & w_{u_1u_3} \\ w_{u_2u_1} & w_{u_2u_2} & w_{u_2u_3} \\ w_{u_3u_1} & w_{u_3u_2} & w_{u_3u_3} \end{matrix}}^{Users} \Bigg\} Users \tag{1.6}$$

.

Note that the term *metric* is used as the colloquial meaning of a measurement, and not in the sense of the mathematical definition (distance function).

Rating estimation

The input needed to create estimates/predictions of ratings is the similarity weights as well as the original ratings. To create a rating estimate for a user u on an item i, the weights of u are first sorted by size in descending order. The weights of u is meant as the weights w_{uv} for every user $v \in \mathcal{U} \setminus \{u\}$, i.e., the weights that signify the similarity between u and the other users. This corresponds to a row, or equivalently a column, in the weight matrix $\mathbf{W}$. This weight matrix is simply the set of weights organised in a matrix, where the weight w_{uv} is the element on the uth row and vth column.

The next step is to look at which users have rated the item in question, i, and use the ratings made by the k nearest users (in terms of highest similarity) that have also rated i to create an estimate. This essentially means that you look for other people who historically have had similar preferences to a certain user, and use their ratings on an item to estimate the value of a hypothetical future rating done by the user in question on that item. There are a few ways of creating the rating estimates. One is by simply averaging the k nearest ratings, i.e.,

$$\hat{r}_{ui} = \sum_{v \in \mathcal{N}_i(u)} \frac{1}{|\mathcal{N}_i(u)|} r_{vi} \tag{1.7}$$

where $\mathcal{N}_i(u)$ is the neighbourhood, a set of users that have rated item i and are the closest in similarity to u. The cardinality is k, if possible, and smaller if less than k users have rated the item.

One can also create the rating estimate as a weighted average of the neighbours' ratings, using the similarity weights,

$$\hat{r}_{ui} = \frac{\sum_{v \in \mathcal{N}_i(u)} w_{vi} r_{vi}}{\sum_{v \in \mathcal{N}_i(u)} |w_{vi}|}. \tag{1.8}$$

The value of k is normally chosen by cross-validation, tuning it by testing different values on the data already available. Values above 50 are often used, but this may vary between different recommender tasks [7].

Example

	The Godfather	Star Wars IV	Schindlers List	Casablanca	LotR III
Johan	2	1	5	5	4
Søren	4	?	1	?	1
Magnus	2	?	3	4	1
Andreas	5	1	5	3	2
Stefan	1	4	3	4	1

Table 1.1: Table showing ratings for five users and five movies

An example is shown in Table 1.1. A rating of 5 means the user loved the movie, 1 means a user disliked the movie, and a '?' means the movie has not yet been consumed or rated by the user. If Magnus want to known whether or not to consume Star Wars Episode 4, he can do that by using collaborative filtering on the data available. The order of the neighbours to Magnus in preference similarity are Stefan, Johan, Andreas and Søren, given by (1.4). For a value of $k = 2$, the estimated predicted rating for Magnus and Star Wars IV would be the average of Stefan's and Johan's ratings, which is $\frac{4+1}{2} = 2.5$.

Standard improvements

Normalisation of ratings is a further development that is not specific to neighbourhood based algorithms but one that is commonly used when looking to improve one. It is a way of pre-processing the rating data and compensating for differences in ways users use the rating scale [2, 8]. A common way of doing this is mean centering the ratings for the users, that is, subtracting the users' rating means off the raw ratings. This simply means that the recommender takes in the pre-processed ratings, and gives an output which is estimations of the transformed ratings, to which the mean will be added back on. A raw rating r_{ui} is transformed to a mean centered rating $h(r_{ui})$,

$$h(r_{ui}) = r_{ui} - \overline{r}_u, \tag{1.9}$$

where $\overline{r}_u$ is the mean of all of user u's ratings. One can also mean centre around the item averages, in which case r_u is just substituted for r_i above. The recommender runs the transformed ratings, and the predictions are then transformed by the inverse, h^{-1}, to create the final ratings: $\hat{r}_{ui} = h^{-1}(\widehat{h(r_{ui})})$.

A way of developing this further, that in a similar way to mean centering accounts for variability is called z-score normalisation. This normalises the mean centered ratings in the aspect of their variance. The transformation is,

$$h(r_{ui}) = \frac{r_{ui} - \overline{r}_u}{\sigma_u}, \tag{1.10}$$

where σ_u is in practise replaced with the estimated variance of the ratings made by user u. This can be developed further, such as the pre-processing transformations used in the Netflix algorithm [9].

Sophisticated improvements

The *kNN* algorithm was one of the first algorithms proposed [7] and is still used because of its simplicity and adequately good accuracy. As this has become an extensive field of research, the state of the art has (in most cases) become other techniques. There is however still a case for *kNN*-like algorithms' possibility of outperforming other algorithms. A 2014 paper published at the ACM conference [2] made the case for a neighbourhood based algorithm called *Unified Nearest Neighbour* that outperformed the current matrix factorisation techniques on some commonly used data sets [11].

The cold start problem

One specific problem is to create estimations when a new user, enters the system [2]. This is know as the cold start problem. As this user has made no (or very few) ratings, one does not have much to go by when calculating similarities. The cold start problem is not specific to neighbourhood based algorithms. The proposed methods in chapter 2 may be able to handle this situation well. Another, simple, way of dealing with the problem is to use unpersonalised top lists in place of personalised recommendations. This is a convenient and fairly effective way of recommending items to new users, but it may not give an impressive first impression of the recommender system in question.

1.3.2 User- and item-based recommendations

User- and item-based kNN-recommenders are two methods with the same goal, to predict how much a user will like an item. One is based on the relation between users and the other is based on the relation between items [2]. A conceptual comparison would be if a user asked its friends, who are known to have similar preferences, for recommendations for a

[2] Association for Computer Machinery

book to read (User-based) versus if the basis for the evaluation of the book is how much you like books that are considered (rated) the same as the book in question (Item-based).

The posed question here is usually when one should use one or the other, and the answer depends on the ratio between the number of items and users, among other factors. If there are more users than items, item-based is usually preferred as less memory is required to store the W-matrix while the complexity is none the less the same [2]. Another aspect to weigh in is the rate at which new items or users arrive, if content is changing on an everyday basis a user-based recommender might be better and vice versa.

The advantage of using a user based recommender is that it builds up a neighbourhood of users for you and have a higher probability of recommending serendipitous items to a user as that kind of information can be caught in the behaviour of a users neighbourhood. The diversity of the items recommended can be much higher, as an item based recommender focuses on the similarities between items. However, a user based recommender tends to not create as accurate recommendations leaving a larger error margin for the predictions. An item based recommender on the other hand does produce better and closer predictions of items, but is not very good at serendipity. Item based recommenders tend towards producing a homogeneous result set; that is, if a user likes action movies - action movies will be suggested. A difference between how the methods are used is in how the systems justifies the recommendations. An item can be recommended because it is similar to an item a user has consumed (item based). An item can be recommended because other users who are similar to the users in question have liked it (user based).

Chapter 2
Nearest Neighbour's Neighbours

There are many methods one can apply to improve the standard *kNN*-algorithm for specific uses, such as rating normalisation and wisely choosing the rating estimator. This chapter presents a novel idea on how the accuracy of the *kNN*-algorithm can be improved by using an intuitively reasonable idea. The idea is based on the hypothesis that high user-user preference similarity is transitive.

2.1 The hypothesis

One important step in the *kNN* recommender system is to estimate similarities between users by appropriately assigning pairs of users with similarity weights, which are used to create the neighbourhoods. The resulting similarity weights, w_{uv}, are usually represented in the matrix $\mathbf{W}$, where user u and user v correspond to a row and a column respectively. The matrix has no values on the diagonal (users' similarities with themselves) and are also potentially "missing" values elsewhere, for pairs of users that have no common rated/consumed items. The (estimated) similarity is then said to be zero.

One can think of this as there being some kind of true similarity between users, and the computation of the weights is an attempt to estimate these similarities. Because of the sparseness of the rating matrix, these estimations might be very poor, for some users. The algorithm step proposed in this chapter, which is really a reweighing of the neighbours, attempts to improve those similarity estimates.

In fact, this matrix is a representation of a graph. More specifically, an undirected and weighted graph. It is a general example of an adjacency matrix. An intuitive way of thinking about the graph is that people to different degrees are similar to each other (the graph is complete; all similarity edges are present, though some are incorrectly estimated to be zero). However, there are several possible paths to take between two users if one allows the path length to be longer than one edge long, and when estimating user similarities on real data this leads to certain discrepancies. If user A gets a high similarity value to user B,

and B has a height similarity value with C, then it is still very possible that the estimated similarity between user A and C is low, due to estimation uncertainty, when they in fact likely have similar preferences. This "noise" could potentially be reduced by making use of other paths other than the direct edge between two users. The way to do this that is presented in this thesis, is from here on referred to as the Nearest Neighbours' Neighbours (*NNN*).

2.2 Primary similarity weights

The proposed algorithm will rely on the standard similarity weights, which we call primary weights. These primary weights were created in accordance with what was described in section 1.3.1, using the inverted root mean squared difference (1.5). This matrix, or set of weights, is the fundamental building block for the rest of the algorithm. The idea is to still primarily use these weights, but weigh in a "correction term" that might improve the performance of the algorithm. We still denote these primary weights, as w. The set of all weights, organised in a user-user matrix, is denoted $\mathbf{W}$.

2.3 Creating alternative similarity weights

A few options when creating alternative similarity weights were tested. The equations include three users, u, x and v; where u is the user in question (the users whose similarity weights we wish to update), x is the closest neighbour to u and v is the user to whom the distance from u is estimated.

The proposed justification for this method is that the closest neighbour's neighbourhood would include information that user u did not possess. If a u and x have an overlap of a few items while x and v have another set of overlap, then v's ratings could be used to predict the ratings for items u have not consumed.

In all of the below proposals, x denotes the closest neighbour to u, i.e.,

$$x = \arg\max_{y \in U} w(u, y). \tag{2.1}$$

2.3.1 Proposal A

The first proposal is to construct a weight for a user u by multiplying the weight to their closest neighbour with the weight between the nearest neighbour and the user to whom one is estimating the similarity to. The (square) root of this is used as the alternative weight, as it should in a sense recreate the appropriate variability (and magnitude) in the weights,

$$w'(u, v) = \begin{cases} \sqrt{w(u, x) \times w(x, v)}, & v \neq x \\ w(u, v), & v = x \end{cases}. \tag{2.2}$$

The latter factor $w(x, v)$ simply acts as an indicator of similarity between u and v, and the factor $w(u, x)$ makes the alternative weights higher when that similarity is high, that is when the neighbours u and x are very close to each other in estimated preference.

2.3.2 Proposal B

This proposal is similar to the first one, but possibly more intuitive and simple. One simply substitutes a user's weights for that user's closest neighbour's weights. The idea is that most users have at least one close neighbour, whose weights would be almost as indicative of similarity to other users as ones own weights:

$$w'(u, v) = \begin{cases} w(x, v), & v \neq x \\ w(u, v), & v = x \end{cases}. \tag{2.3}$$

2.3.3 Proposal C

This proposal is to make a weighted average of a user's l closest neighbour's corresponding weights,

$$w'(u, v) = \frac{\sum_{i=1}^{l} (l + 1 - i) \times w_{x_i v}}{\sum_{i=1}^{l} (l + 1 - i)} \tag{2.4}$$

where x_i is the i^{th} closest neighbour to u, i.e., x_1 is the same as x above, and

$$x_i = \underset{y \in U \setminus \{x_{i-1}, \ldots x_1\}}{\arg\max} \; w(u, y) \tag{2.5}$$

for $i > 1$.

2.3.4 Proposal D

The last proposal just sets the alternative weight to a reweighing of the standard weight:

$$w'(u, v) = \begin{cases} w(u, v) \times w(u, x) \times w(x, v), & v \neq x \\ w(u, v)^3 & v = x \end{cases} \tag{2.6}$$

A property of the weights in this set is that a weight $w'(u, v)$ is zero if $w(u, v)$ is zero. It thus does nothing to order the user pairs' weights that have zero similarity according to the primary weights. The higher power results in a sharper discrimination between original weights that are low and high.

2.3.5 Properties of the alternative weights

Note that these weights are not invariant to direction, i.e., w'_{uv} is not necessarily the same as w'_{vu}. The alternative weight matrix is not symmetric. Hopefully, these alternative weights

u1	1 2 1 1 5 2
u2	1 5 1 3 1 4 2
u3	2 1 4 2 5 4 2

Figure 2.1: Example of a situation where two users are most likely very similar (u_1 and u_3) but do not share any mutually rated items

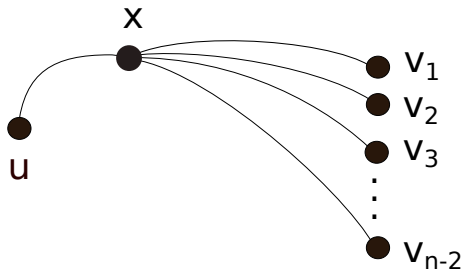

Figure 2.2: Representation of the similarities being calculated using a simple path.

is a good indicator of the true preference similarity - as the primary weights are. Realistically, no two users/people are the same, and thus it would introduce bias when weighing in another user's weights. However, calculating a similarity weight is merely an estimate of true preference similarity, so if the offset that the bias introduces is low, compared to the noise of the estimated similarity, one can still expect to gain in accuracy. The time complexity of this step is $O(n^2)$ where n is the number of users.

The weights that are used for the experiments are non-symmetric. One could argue that similarity is undirected, but the similarity weights are however used when a neighbourhood is chosen for a specific user. Your closest neighbour can maybe say more about how similar the rest of the users are to you, than the closest the neighbours to the rest of the users can say. It is however possible that it would be a good idea to average the weights, i.e., averaging the matrix with its own transpose.

2.4 Combining the weights

Using the alternative weights alone when predicting ratings would probably result in a slightly worse performance than using the original weights alone, as they are less accurate estimates of preference similarity. The most natural way to use them would be to combine them with the original primary weights, i.e., somehow adding them to make final weights that are used for the rating estimation. The final weight will hereafter be denoted with a star: $\mathbf{W}^*$.

2.4.1 A linear combination

A simple and reasonable way of combining the two sets of weights is to just make a linear combination, as displayed (2.7) below;

$$w^* = w + \beta w' \tag{2.7}$$

where the most suitable value of β should reasonably be between 0 and 1, as they (the weights) are postulated to correlate positively with true similarity but not carry as much useful information as the original weights. This is something that is investigated in the experiments, in particular Experiment 3.

It is worth to note that this way of constructing the weights make them non-symmetric, i.e., the similarity between user u and user v depends on which direction one is looking. This is of course something that could be averaged out, or one can argue that you can trust your close neighbours weights more than users that are close neighbours to the users you want to know you closeness to.

2.4.2 Combining conditionally

It is possible that the *NNN* offers improvement for some users, but not at all for others. It is then reasonable to change similarity weights based on some conditions on the users,

$$w^*_{uv} = \begin{cases} w_{uv} & \text{if } condition \\ w'_{uv} & \text{else} \end{cases} \tag{2.8}$$

The condition would be something concerning user u, for instance that $|\mathcal{I}_u|$ is smaller than a certain threshold and that $|\mathcal{I}_u \cap \mathcal{I}_x|$ is big enough.
This is really just a problem of appropriately partitioning the set of users to choose between $\beta = 0$ and $\beta = \phi$ for some $\phi > 0$. One could also try to create a function that takes some user information in and gives a dynamic value of β, which would potentially result in lower errors than having an optimised fix β if that function is chosen well enough. Possibly other machine learning algorithms could be used to design a function like that. The first proposal, Proposal A, is a way to give more weight, when combining the similarity weights, to users who are very similar to their nearest neighbour (because of the factor w_{ux}). It may however, in retrospect, be more wise to give more weight for the users that both have the high similarity weight to their closest neighbour as well as a high certainty in this estimate of the similarity.

2.5 Estimating the ratings

The proposed Nearest Neighbour's Neighbour algorithm is different from the standard *kNN* in the way the similarity weights are calculated, which is the first part of a neighbourhood based algorithm. The second and final part is creating rating estimates/predictions. This algorithm does not, as of now, offer any new way of doing this. A choice of what rating estimator to use is nevertheless necessary. For most of the experiments, in the next chapter, the simple averaging estimator is used, which is discussed in Sec. 2.5.1.

2.5.1 Standard *kNN* rating estimation method

A simple way of creating predictions by averaging the ratings in the neighbourhood (2.9), was mainly used in the experiments for a few reasons. In most cases it seemingly performed as well as a weighted average estimator. Also, a difference in the rating predictions will have to be the result in a different neighbourhood having been taken out, that is that the similarity weights' values has changed enough to result in a different order of them. If one is able to choose better neighbourhood (giving better results using the chosen estimator), then there are several possibilities in how to proceed from there, other than that it is good in itself,

$$\hat{r}_{ui} = \frac{1}{|\mathcal{N}_i(u)|} \sum_{v \in \mathcal{N}_i(u)} r_{vi}. \tag{2.9}$$

2.5.2 Further alternatives

A common way of predicting the ratings is by computing a weighted average of the ratings for the item in question, by the users in $\mathcal{N}_i(u)$. These weights are commonly set to be the same as the similarity weights, as in (1.8). It would however be very coincidental if these are the optimal weights for this purpose. It is natural to set it up as a standard regression problem where you model a rating as,

$$r_{ui} = a_1 r_{v_1 i} + a_2 r_{v_2 i} + a_3 r_{v_3 i} + ... + a_j r_{v_j i} + ... + a_k r_{v_k i}, \tag{2.10}$$

where $r_{v_j i}$ is the j^{th} neighbour in closeness to u that has also rated item i. One could train the parameters a_j on the existing data and use the resulting set to create the estimated ratings, for instance with a least squares fitting, see appendix A. Another possibility is to further develop the weighted average (1.8) by changing the weights proportions,

$$w_{uv} = b_1 w_{uv} + b_2 w_{uv}^2 + b_1 w_{uv}^3, \tag{2.11}$$

where the optimal vector containing the b-coefficients would be evaluated and cross validated.

Chapter 3
Experiments and results

This chapter contains experiments conducted on the *NNN*, evaluation and a discussion of the outcome.

3.1 Implementation

3.1.1 Data

Two data sets were to be used, The MovieLens data set of 100k ratings and the MovieLens data set of 1 million ratings. The data sets are data collected by the University of Minnesota [1], it is considered a standard within the machine learning (and thus recommenders) community and have the advantage of being used as a basis for many published research results. [1]. It is therefore a highly useful data set where the results can be compared to other research results found online.

3.1.2 Parallelisation

For all experiments conducted in the thesis Matlab was utilized. Due to the size of the matrices used, often above 3000 users and 1500 items, calculations tended to be slow. This was primarily true for calculating distance metrics and estimating individual ratings where the complexity always was $O(n^2)$, where n is the number of users. Calculation time was improved by introducing Matlabs parallel toolbox.

To speed up the computation time even further, the computer cluster Alarik at Lunarc was used. Lunarc is a High Performance Computing (HPC) Center at Lund University, where Alarik is the latest cluster using a SLURM queueing system containing 208 nodes, each with 2 8 core 64-bit processors (AMD6220), i.e., 3328 CPUs in total [5]. What was

[1] Data can be found at http://grouplens.org/

new in Alarik was that Matlab could directly interface to the cluster without writing batch scripts. With this interface, simulations with different parameters were run simultaneously.

3.2 Experiment 1

Experiment 1 is a simulation designed to compare a set of compound weights, w^*, and standard weights, w, to the users' theoretical true preference similarities. The aim is to investigate whether a matrix $\mathbf{W}^*$, as described in chapter 2, can be closer to the users' true preference values than that of the standard $\mathbf{W}$. As these preferences are naturally unavailable, an attempt was made to generate a matrix of weights thought to be as accurate as possible in actual similarity. The idea is to generate this matrix of weights, $\mathbf{W_{GT}}$, that one can take as a ground truth for similarity, and then conduct an evaluation of how close the estimated similarities ($\mathbf{W}$ and $\mathbf{W}^*$) are to it. The $\mathbf{W_{GT}}$ was constructed by computing similarity weights of a very dense user-item matrix. Elements from this user-item matrix were removed, to make it sparse, and the weight matrices $\mathbf{W}$, $\mathbf{W}'$ and $\mathbf{W}^*$ were computed from this sparse matrix. The two matrices ($\mathbf{W}$ and $\mathbf{W}^*$) were then compared to the "truth", the $\mathbf{W_{GT}}$, to see how well they coincided.

The MovieLens 100k data set was used for this experiment. It is a set of 943 users and 1682 items, and 100 000 ratings. The rating matrix thus consists to 93.7 % of non entries. A sub-matrix of this matrix was created by eliminating rows and columns, namely the ones containing the fewest entries, resulting in a 604 by 396 matrix with a denseness of 75 %. This matrix was split into two sets, one for parameter training and one for evaluation, see Fig. 3.3. The similarity metric of choice (1.5) was applied on each of these rating matrices to create justifiably accurate matrices of similarities; $\mathbf{W_{GT}}$.

Next, the two rating matrices were sparsed to a high sparsity by eliminating circa 90 % the last timestamped ratings, resulting in two matrices that were as sparse as the original data set.

The same similarity metric (1.5) as just mentioned were applied to these rating sets, creating sets of weights (see Fig. 3.3 boxes $\mathbf{W_{train}}$ and $\mathbf{W_{eval}}$). From these weights, matrices of alternative weights, $\mathbf{W'_{train}}$ and $\mathbf{W'_{eval}}$, as described by *Proposal A*, were computed. A graphical interpretation can be overviewed in Fig. 3.3 of the entire process.

A function describing the difference, or distance, between the ground truth and a compound weight matrix is stated (3.1). This function's minimum on $(\alpha,\beta) \in \mathbb{R}^2$ as well as $\beta = 0, \alpha \in \mathbb{R}$ was evaluated (see Fig. 3.2),

$$f(\alpha,\beta) = \sum_{u\in\mathcal{U}}\sum_{v\in\mathcal{U}}(w_{GT}(u,v) - (\alpha w(u,v) + \beta w'(u,v)))^2. \tag{3.1}$$

This essentially compares the weights in a standard weight matrix $\mathbf{W}$ with that of the $\mathbf{W_{GT}}$, simultaneously comparing the compound weight matrix $\mathbf{W}^*$ with the W_{GT}. The reason a factor of α is present in both cases, is because there might be a difference in for instance mean value in the matrices. These differences are not interesting, as what is relevant to a *kNN* is the proportions of the weights in a row or column. Adding a factor of α gives the script full freedom to fit the matrices together, without changing the internal order or proportions of the weights.

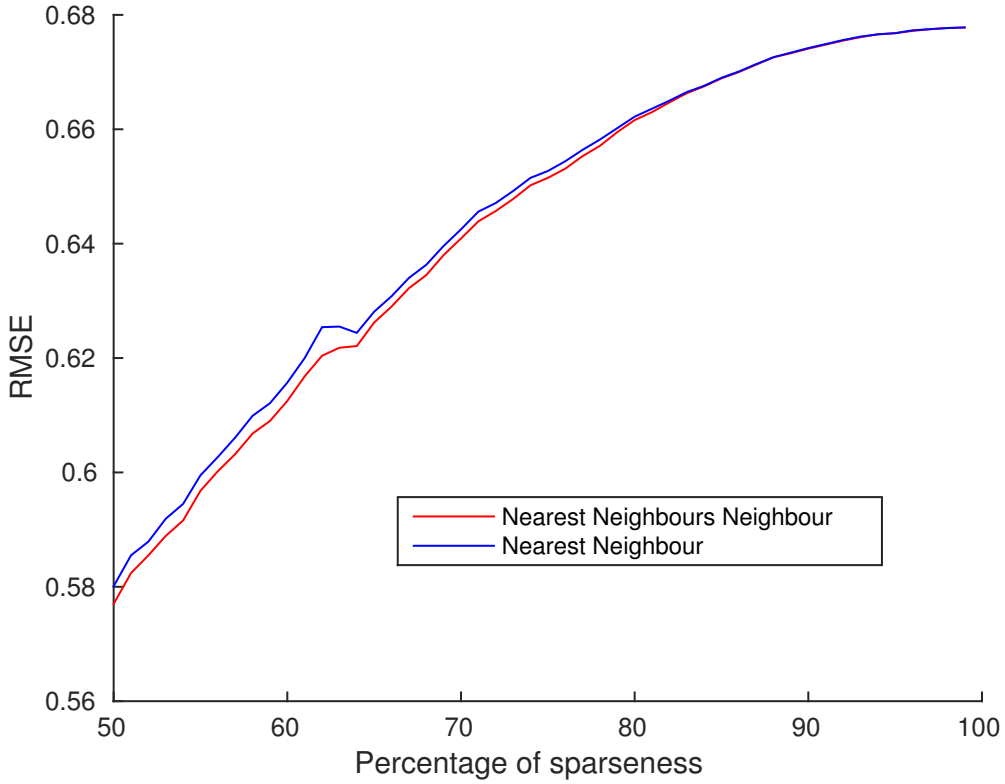

Figure 3.1: Comparison of the fit of $\mathbf{W}$ and $\mathbf{W}^*$ to $\mathbf{W}_{\mathbf{GT}}$, respectively, over different levels of sparseness. Note the truncated axis. The sparseness is simply a ratio and the unit of the root mean square error is the same as the similarity.

The training data set supplied the appropriate values of α and β,while the other set was used to evaluate the fit, the minimum of (3.1), for $\beta > 0$ and $\beta = 0$ respectively.

This setup was carried out over a range of sparsity levels. The rating matrices were sparsed to different levels and the rest of the experiment was carried on. The results are shown in Fig. 3.1.

One could question how well the $\mathbf{W}_{\mathbf{GT}}$ describes true similarity. Similarity is difficult to quantise, or even define in an ordinal way. It can however be thought of as whatever metric results in the overall best neighbourhood selection. As the neighbourhood is used to create rating predictions, whose corresponding errors should be low, it is reasonable that choosing them based on the lowest average squared error will result in low average squared prediction errors. As more data is available if the matrix is dense, this should result in a similarity matrix that is as accurate, in the sense described above, as possible. The values of α and β used in the calculations in Fig. 3.1 are shown in Fig. 3.2.

The optimal values of β seem to lie above zero, except in extremely sparse situations. This is reasonably interpreted as it being the result of having very little data present and overfitting (or even rank deficiency). The previous positive values of beta indicates that there is useful information in the matrix $\mathbf{W}'$, and that this is not a case of overfitting data, because of the additional parameter. These values of the coefficients were chosen because

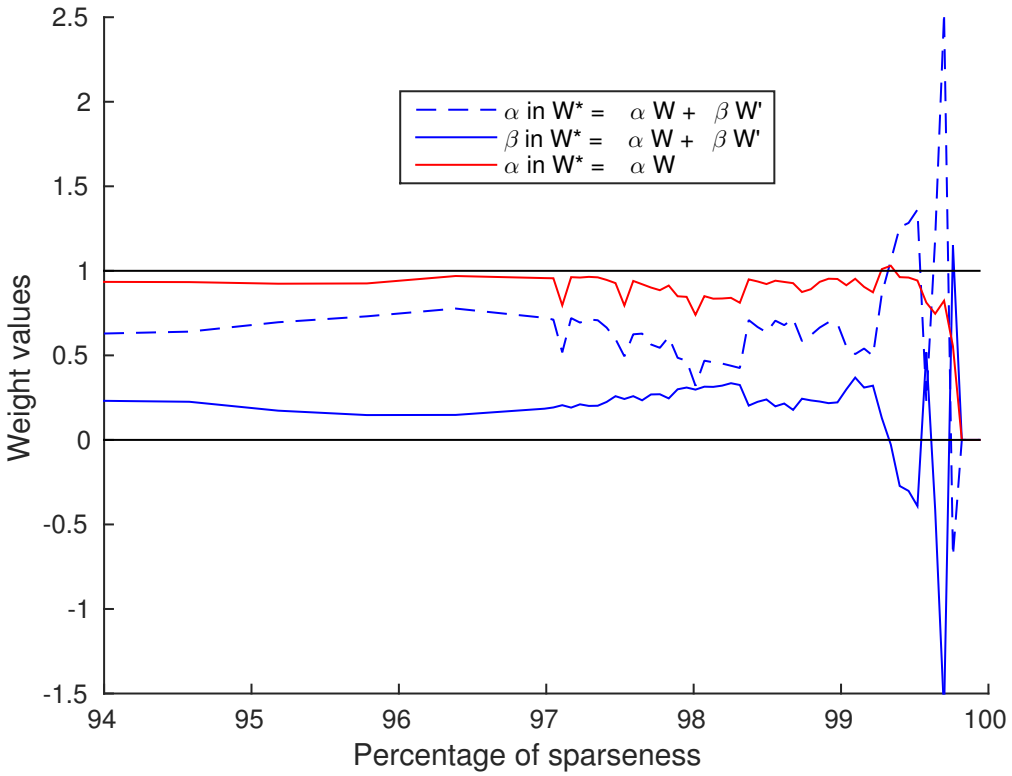

Figure 3.2: The chosen (optimised) values of α and β.

they result in the best fit in training, and they do indeed result in a better fit to $\mathbf{W_{GT}}$ than if β had been fixed to 0, when evaluated on an validation set.

As one can see, the estimated optimal values of β look somewhat noisy over the span of sparseness, so the fit could have been better with an averaged, smoothed, series of values in the evaluation.

There does not really appear to be a trend (increasing β) as maybe one would hope and expect.

The *NNN*-made weight matrix seems to be more coherent with the ground truth matrix than how much the standard *kNN* weight matrix is, which is promising for the *NNN* algorithm. They do however appear to converge as the sparseness of the data gets higher. There is a small gap between the error values at low sparseness, that then gets smaller and smaller as more data is removed. Ideally the case would be the other way around, as high sparsity is a problem in recommender systems. Nevertheless, the values corresponding to the *NNN* is consistently lower, justifying a further look into the potential of Neighbours' Neighbours.

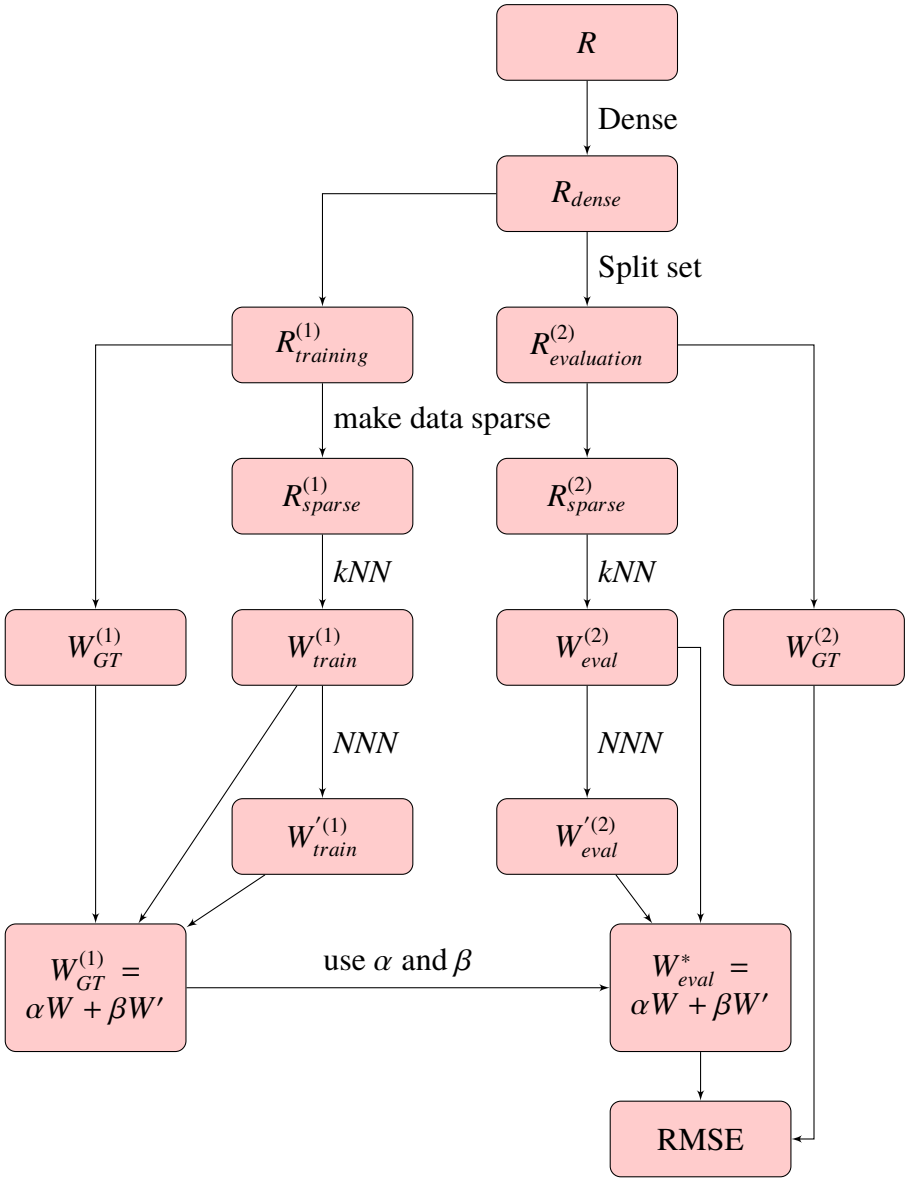

Figure 3.3: Flow chart describing process in Exp. 1. First a dense submatrix is created from the original data (R_{dense}). The data set is split into two sets ($R_{training}$ and $R_{evaluation}$. Entries are removed from the training data to make it more sparse (R_{sparse}) before the similarity metric is applied, the result is $W^{(1)}_{train}$. The $W^{(1)}_{train}$ is run through the *NNN* algorithm and $W^{'(1)}_{train}$ is created. From the block named $R_{training}$ the similarity metric is applied again to produce the $W^{(1)}_{GT}$. A least squares fit is used to compute α and β values in the bottom left block. Next the evaluation data is used ($R_{evaluation}$). Entries are once again removed for a higher sparsity and the similarity metric is run ($W^{(2)}_{eval}$). The same *NNN* algorithm, as for the training data, is then run to get the $W^{'(2)}_{eval}$. The α and β from the training set is then used to calculate the W$*$. From the $R_{evaluation}$ a $W^{(2)}_{GT}$ is created. The difference (RMSE) is then calculated between the W^* and $W^{'(2)}_{GT}$, as well as for the $W^{(2)}_{eval}$ and the $W^{(2)}_{GT}$, to see how well the new weights versus the standard ones fit with the ground truth matrix.

3.3 Experiment 2

This experiment intended to investigate possible improvements, made by the *NNN*, of the performance in predicting actual ratings. The four alternative weight proposals, as described in the last chapter, were used in the *NNN* algorithm. The benchmark was the standard *kNN* algorithm.

The same dense data set as used in *Experiment 1*, in Sec. 3.3, was used here. Half of it was used to tune parameters (α and β) and the other half was used for the evaluation (the exact same values as shown in Fig. 3.2). That data was made sparse by removing the last made ratings, to different sparseness levels, and these ratings were later used in the evaluation, to compute the error/difference between these actual values and the predictions of them. The standard averaging rating estimator, as described in (2.9), was used. The value of k was set to 5 by considering the size of the data set.

The result is shown in Fig. 3.4. The capabilities of the *NNN* looks very promising. They do not seem to improve the performance when the rating matrix is on the denser side, but appears to be better in high sparsity. Some of the methods seemed to be performing well, others not much differently to the standard *kNN*.

Nevertheless, this justifies a further look into the performance of the *NNN*. This was however done on a very small rating matrix, unrealistically small for most applications. As the method of sparsing the rating matrix, as well as partitioning the ratings for the validation set, were based on removing the fraction of the ratings made after a certain point in time, it was not possible to cross validate the performance. In hindsight, removing ratings like this is probably not more realistic than uniformly randomly removing them. Furthermore, one could have switched the parameter calibration set and the set used for the run. It does however seem appropriate to move on to a bigger data set, and that is exactly what was done in *Experiment 3*.

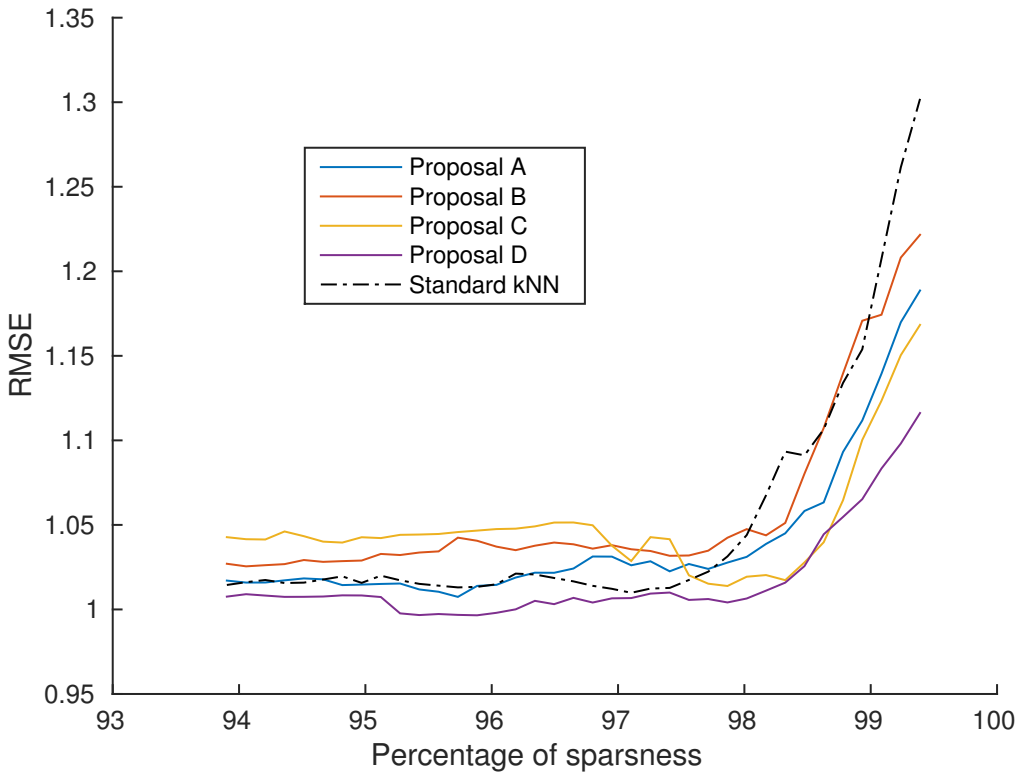

Figure 3.4: Performance on the rating accuracy, calculated over an interval of rating data sparseness.

3.4 Experiment 3

This experiment investigates the effects of weighing in the $\mathbf{W}'$ when making predictions on reasonably big data sets. The MovieLens "1M" (one million ratings) set was used. This set was split into four sets (to have some to work with and one for validation). The rating matrix was simply divided into quarters and any empty rows or columns were removed. This resulted in four rating matrices that were about 3000 by 2000 in size. One of these matrices was used to run this experiment. Some of the ratings were removed randomly to give the resulting matrix a sparseness of circa 99% (and the removed ratings were later used for evaluation). The standard weights $\mathbf{W}$ were calculated. An option to create $\mathbf{W}'$ was chosen and $\mathbf{W}^*$ was calculated according to,

$$\mathbf{W}^* = \mathbf{W} + \beta\mathbf{W}' \tag{3.2}$$

for values of β in the range $[-0.05, 0.50]$, with an increments of 1/100. These different $\mathbf{W}^*$-matrices were then used in the rating estimation, and resulted in a prediction matrix and a corresponding value of total error for each value of β tested. The pairs of values for β and the total prediction errors were then analysed, to see if the optimum (the beta value resulting in the minimum error) was significantly higher than zero, and to get an idea of the magnitude of the improvement (error reduction).

Again, the IRMSD was used as a similarity measure and the averaging rating estimator (2.9) was used. The value of k was set to 40 based on an evaluation of the distributions of the number of ratings the items had. Furthermore, a run was made with the standard *kNN* algorithm, and the optimum value of k appeared to be somewhere in that range. The total error was quite constant in the range between 20 and 50. However, only one simulation was made of this. The value of l in Propsal C was set to 3, as this seemed reasonable and did not require time for optimisation.

The results has some variability to them so a cross validation was made, by running it ten times in a Monte Carlo fashion, to get different random divisions and different outputs that was averaged to get a smooth and more accurate curve. This is shown for the different proposals in Fig. 3.5, Fig. 3.6, Fig. 3.7, and Fig. 3.8. The interval includes some negative values of β, which was chosen with the intention of investigating the consequences of removing what was considered useful information. Further more, it is necessary if one wants to identify an optimum at $\beta = 0$, that is the original *kNN*.

Note the truncated vertical axis. Somewhat disappointingly, a value of $\beta > 0$ does very little to improve the total error, though the optimum is clearly an improvement, for proposal C, see Fig. 3.7. For the simulation involving proposal A, there is a very small dip in error, for $\beta > 0$.

The statistical significance of the indications of these simulations is discussed in Sec. 3.4.1.

3.4.1 Testing of hypothesis

To evaluate the significance, the individual Monte Carlo simulations were looked upon. The results are plotted in Fig. 3.9, Fig. 3.10, Fig. 3.11 and Fig. 3.12, intended to illustrate the variability in the simulations.

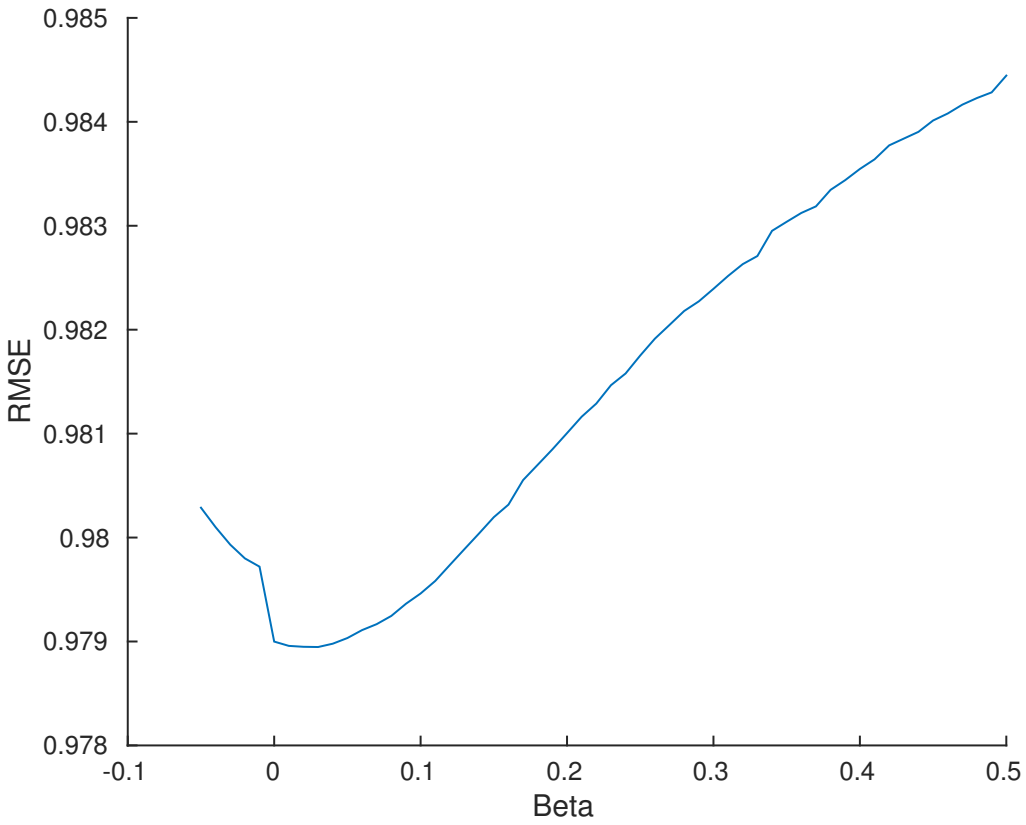

Figure 3.5: Rating prediction errors, using proposal A. The vertical axis shows the resulting root mean squared error. The values of β are the fractions of the added alternative similarity weights.

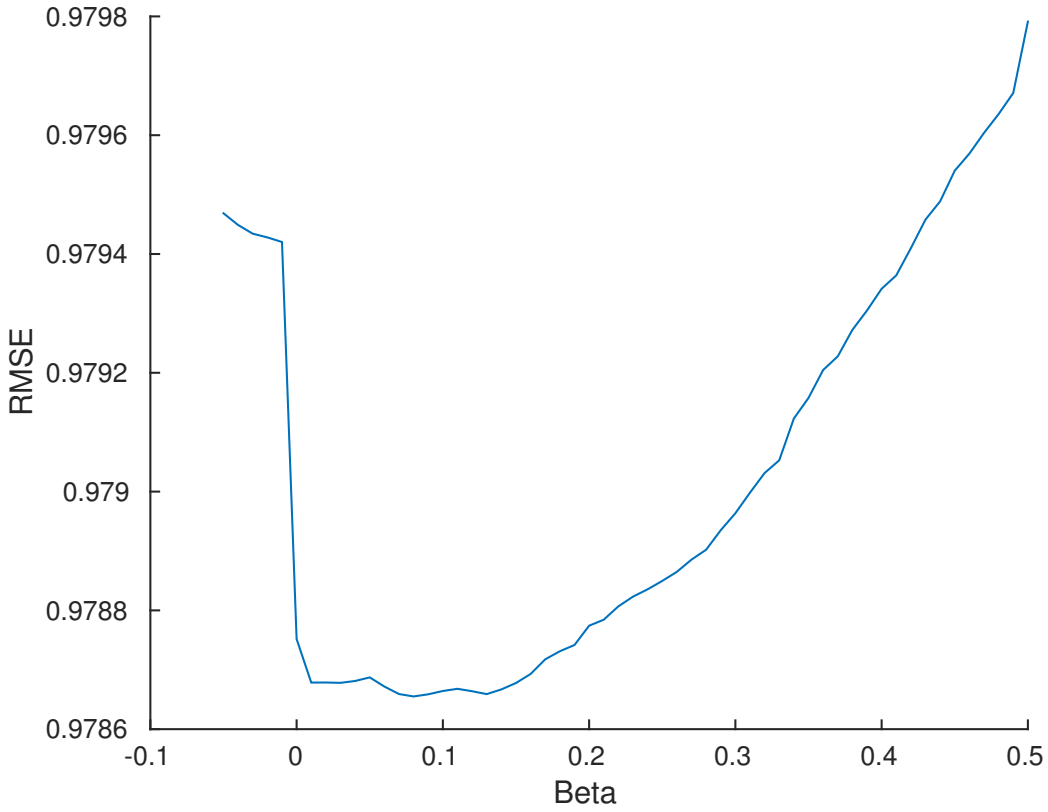

Figure 3.6: Rating prediction errors, using proposal B. The vertical axis shows the resulting root mean squared error. The values of β are the fractions of the added alternative similarity weights.

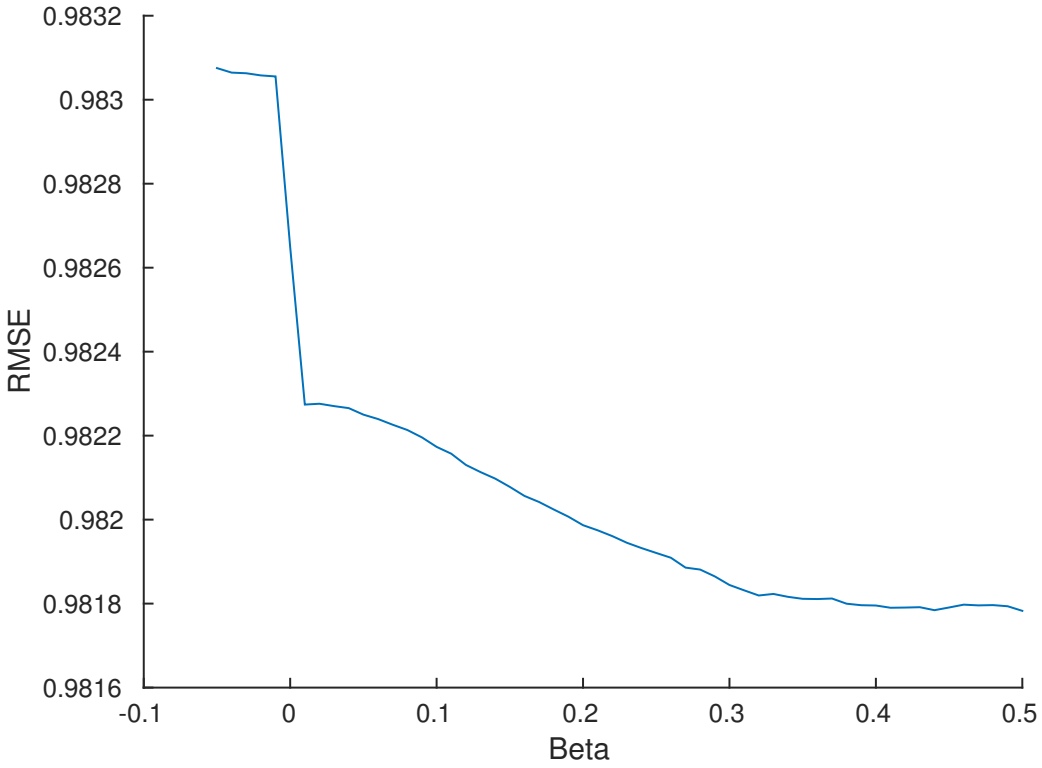

Figure 3.7: Rating prediction errors, using proposal C. The vertical axis shows the resulting root mean squared error. The values of β are the fractions of the added alternative similarity weights.

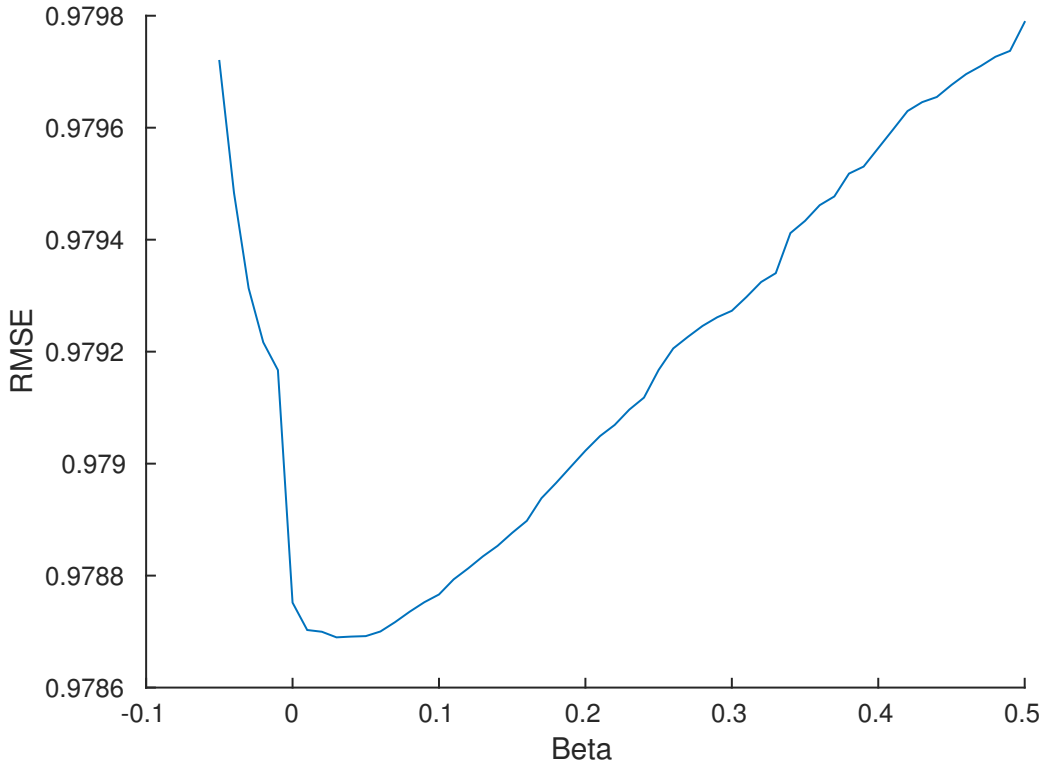

Figure 3.8: Rating prediction errors, using proposal D. The vertical axis shows the resulting root mean squared error. The values of β are the fractions of the added alternative similarity weights.

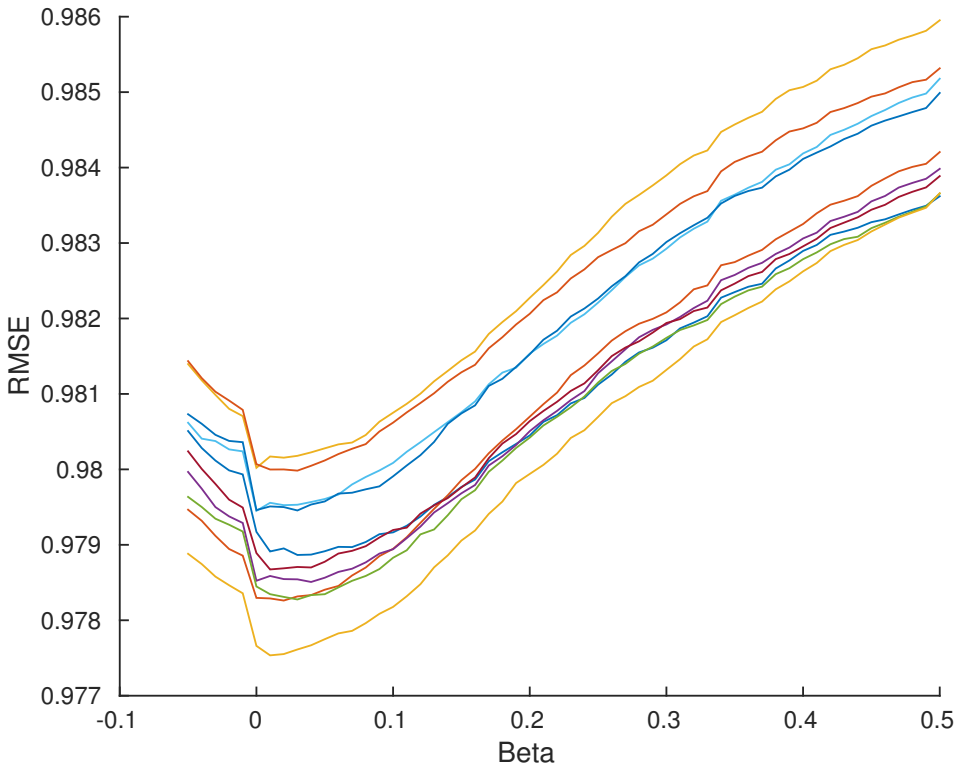

Figure 3.9: Rating prediction errors, using proposal A.

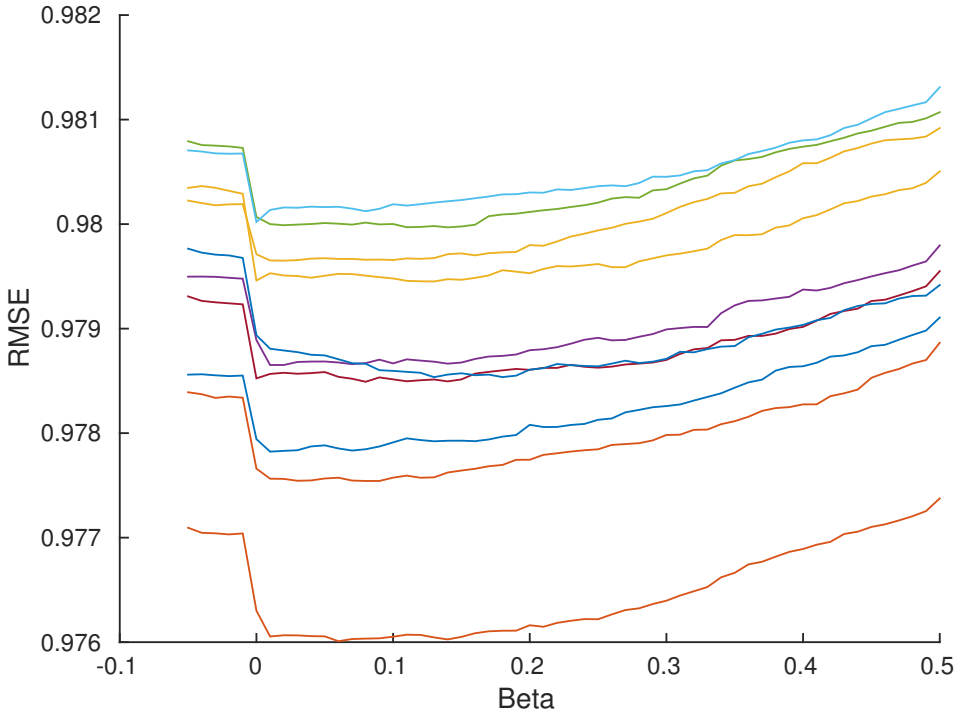

Figure 3.10: Rating prediction errors, using proposal B.

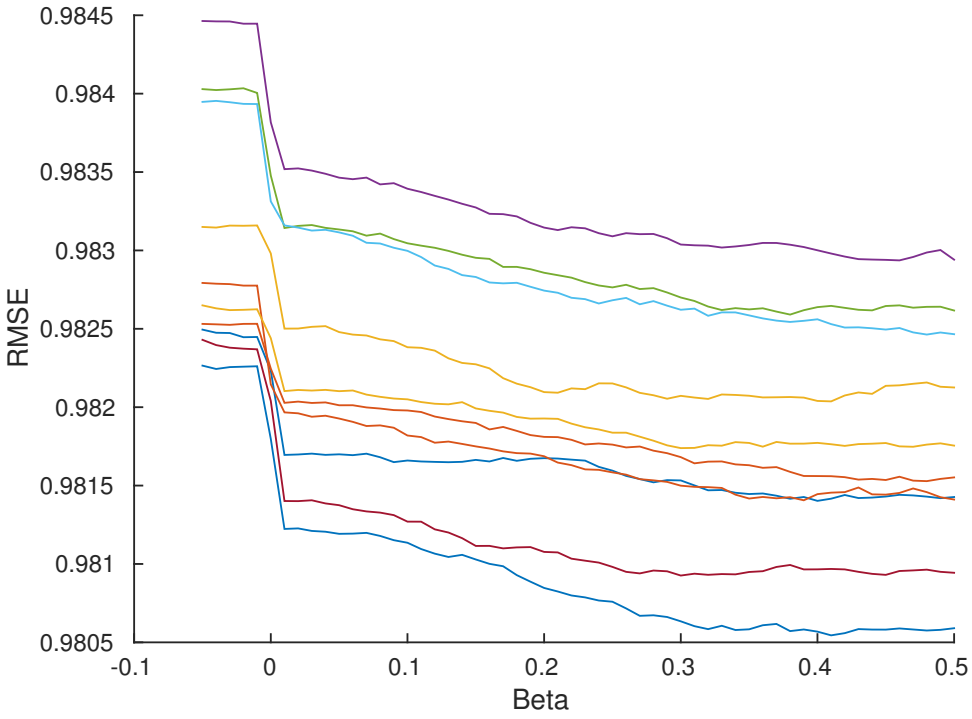

Figure 3.11: Rating prediction errors, using proposal C.

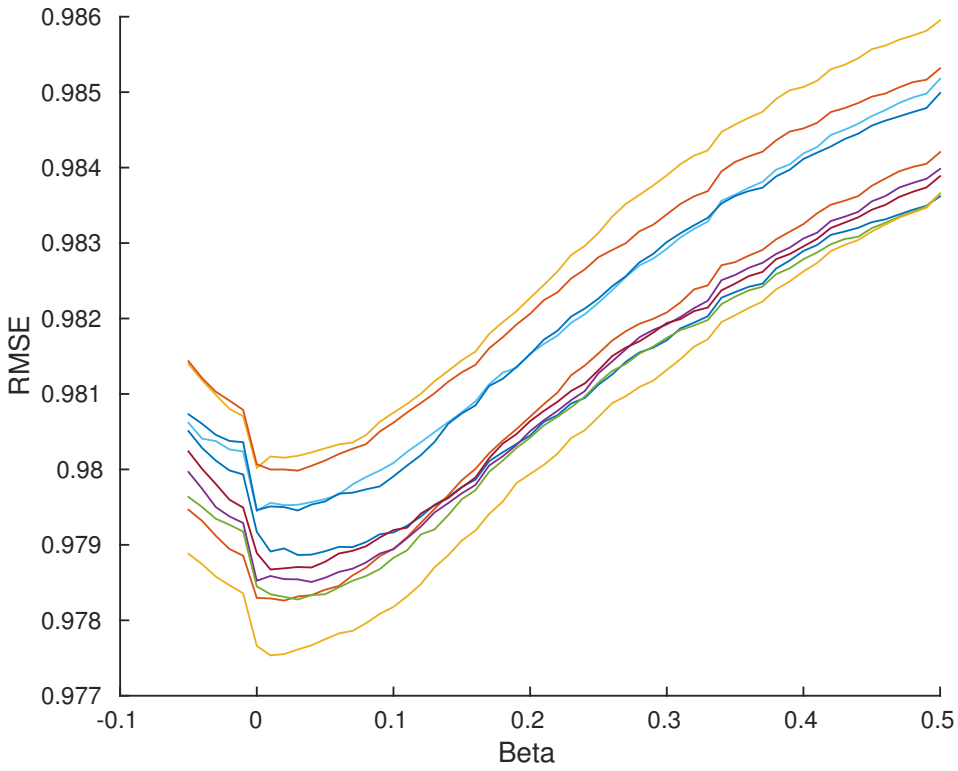

Figure 3.12: Rating prediction errors, using proposal D.

	A	B	C	D
β_{comp}	0.1	0.1	0.1	0.1
MC1	0.0071	0.0309	0.5730	-0.0042
MC2	-0.6502	0.0865	0.3262	-0.0619
MC3	-0.7303	-0.0199	0.5983	-0.1294
MC4	-0.4177	0.2236	0.4236	0.0620
MC5	-0.3809	0.0678	0.4316	-0.0431
MC6	-0.6351	-0.1695	0.3153	-0.2148
MC7	-0.3057	0.0093	0.7696	0.0111
MC8	-0.4475	0.3415	0.6623	0.1379
MC9	-0.5508	0.2491	0.2671	0.0322
MC10	-0.5167	0.0550	0.3900	0.0641

Table 3.1: Showing the differences in each Monte Carlo simulation for *Proposal A - D* for $\beta = 0.1$ compared to $\beta = 0$. Values shown were scaled by a factor 10^3.

One would want to test whether the results that indicated an improvement (lower RMSE) by using $\beta > 0$ were due to chance. A way to do this is to compare the difference in RMSE, between the values corresponding to $\beta = 0$ and $\beta = \beta_{comp}$ respectively. How β_{comp} is chosen is discussed in details below. For a simulations, for one proposal, there are 10 individual runs. We test the hypothesis:

H_0: The difference $RMSE_{\beta=0} - RMSE_{\beta_{comp}}$ is equally probable to be positive as negative.

Against the alternative hypothesis:

H_1: The difference $RMSE_{\beta=0} - RMSE_{\beta_{comp}}$ is more probable to be positive than negative.

The distribution of the outcome (number of runs where the difference is positive) is then binomially distributed. For the simulation involving Proposal C, the probability that the outcome would be what it is, or more extreme (which does not exist), is one in 2^{10}, which equals 0.000977. This p-value certainly seems low, but as this is dealing with a multiple testing problem, the significance level needs to be adjusted. One way of doing this is by using the Bonferroini correction [4]. The experiment-wide significance level $\overline{\alpha}$ can be adjusted by setting the equality in (3.3).

$$\overline{\alpha} \leq k \cdot \alpha_{\{one\ comparison\}} \tag{3.3}$$

For a level of significance, $\overline{\alpha} = 0.05$, a the new threshold for a single test would be 0.0125 as there are four tests (k). The test of Proposal C certainly achieves that, as its p-value is lower. Sadly this is not true for the other proposals. Thus the null hypothesis can be rejected while maintaining a familywise error rate of 0.05.

It is however mentionworty that H_0 is not equivalent to saying that the expected value

of a RMSE-result is the same for evaluations at $\beta = 0$ as $\beta = 0.1$. Assuming that the distribution of these realisations of the RMSE is non-skewed however, it is implied by H_0.

β_{comp} was chosen to be 0.1. This may seem unfair, looking at the plots of some of the other proposals. There is however the problem that the information that was available about the effects of β was the same information that the tests were supposed to run on. What is left then, is to base it on a hunch. It would be interesting to choose the here observed minima as values for β_{comp}, and run new simulations (preferably on a new data set, e.g. another quadrant) where the hypothesis tests would be made again. This has however not been done, due to time constraints. But there is at this point at least one conclusion one can draw about one of the proposals, and some soft information in the results.

Another noteworthy assumption that is made for the hypothesis test is that the success of the *NNN* was independent for each round. This is almost true but not quite. The ratings were not removed randomly, but the percentage to be removed were removed from each row (which preserves the proportions of the distribution of the number of ratings made by users). Any columns that became empty were removed.

3.4.2 Further results

The main results are presented above. But some additional runs were made on variations of the algorithm. An item based recommender was tested (which only involved transposing the original rating matrix before using it as the input). A recommender using the cosine similarity metric (1.2) was used, in place of the IRMSD (1.5). Also, four runs were made, similar to the main runs in the previous section, using the weighted rating estimator (1.8) in place of the averaging one (2.9). The results are shown in Fig. 3.13, Fig. 3.14, Fig. 3.15 and Fig. 3.16.

These last two figures 3.17 and 3.18 were just added as bonus simulations, just to see how they turned out, without having any indication that these specific combinations would be good or bad.

All of these appear to have promising results. The weights may be improved in the sense that they aid the regression problem, when using them in the rating estimator, more than the usefulness they have in choosing a neighbourhood.

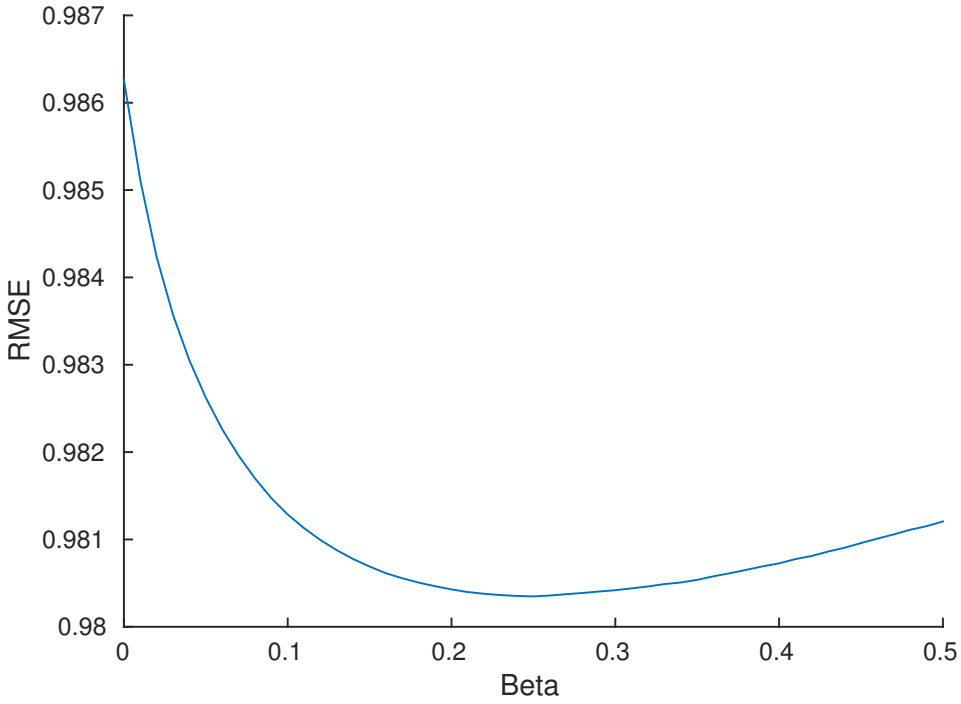

Figure 3.13: Rating prediction errors with weighted rating estimator, using proposal A.

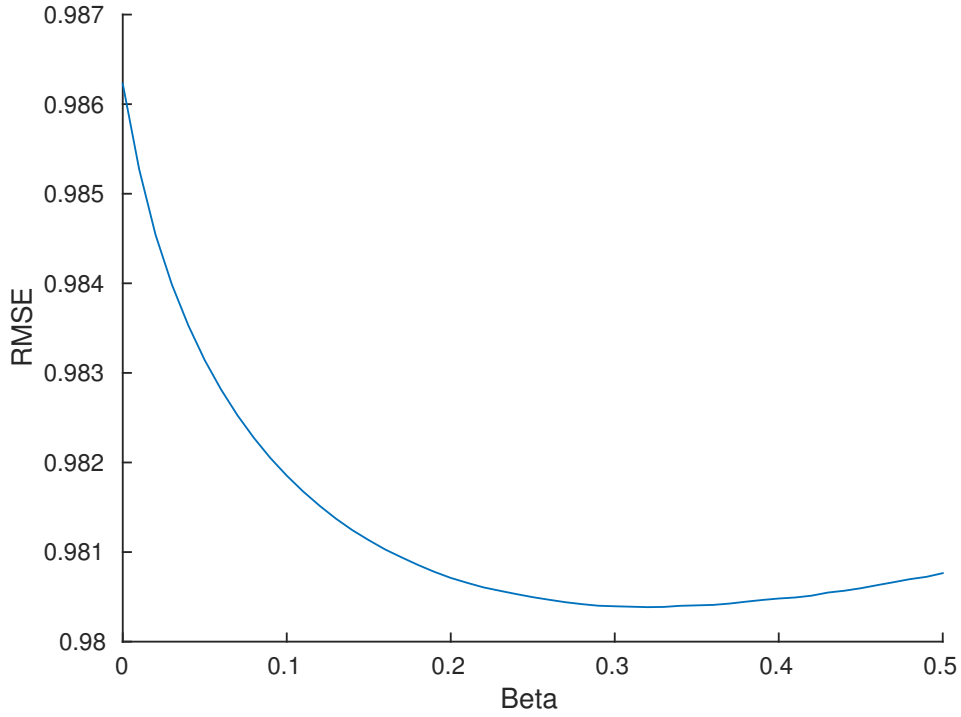

Figure 3.14: Rating prediction errors with weighted rating estimator, using proposal B.

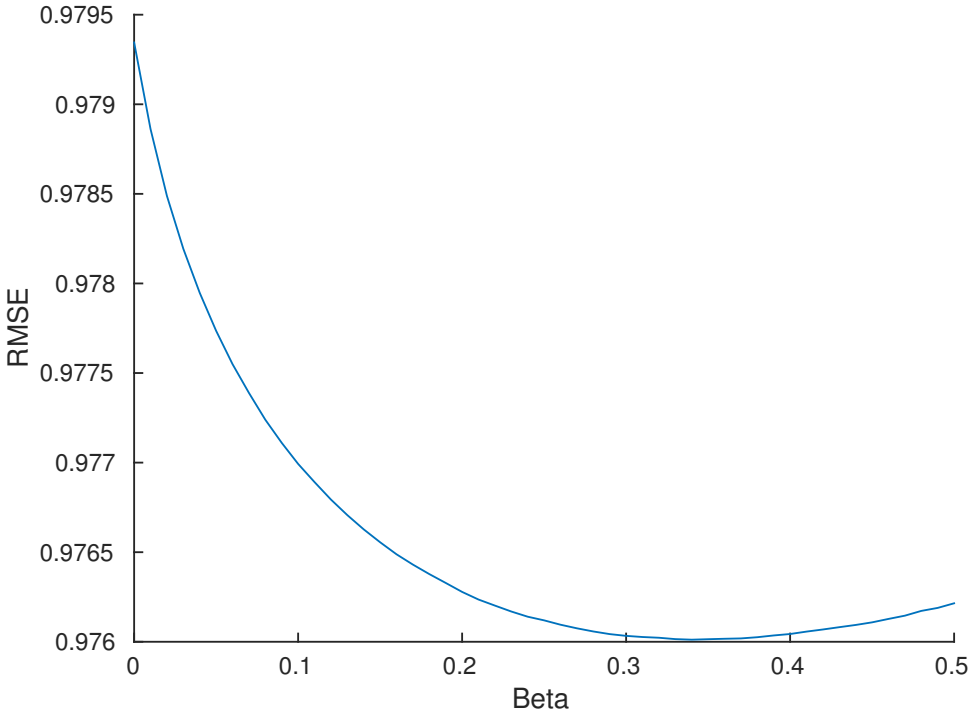

Figure 3.15: Rating prediction errors with weighted rating estimator, using proposal C.

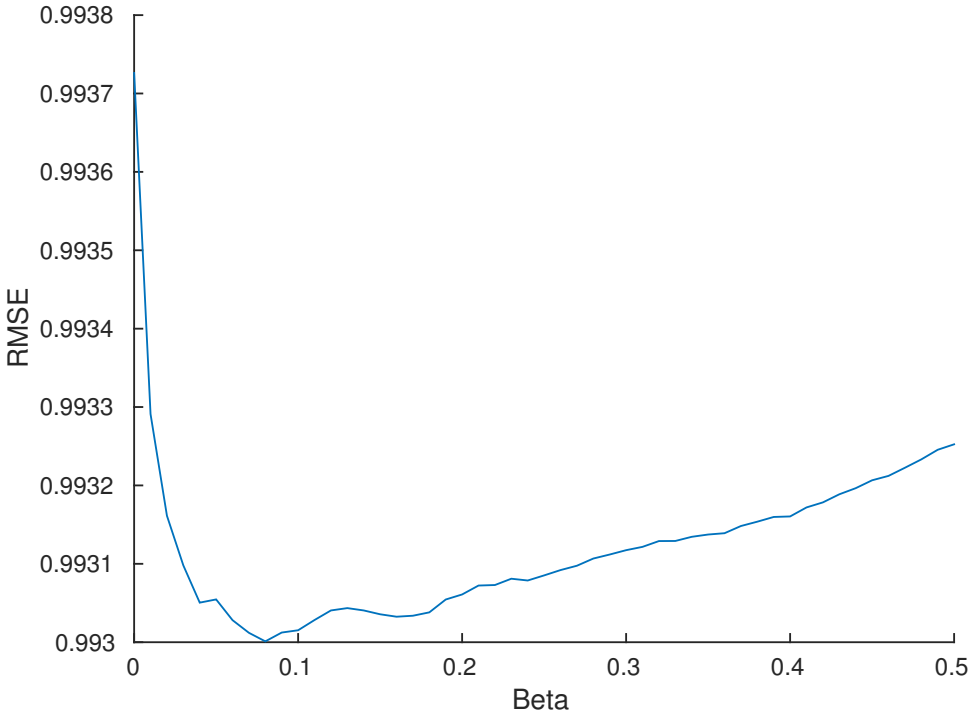

Figure 3.16: Rating prediction errors with weighted rating estimator, using proposal D.

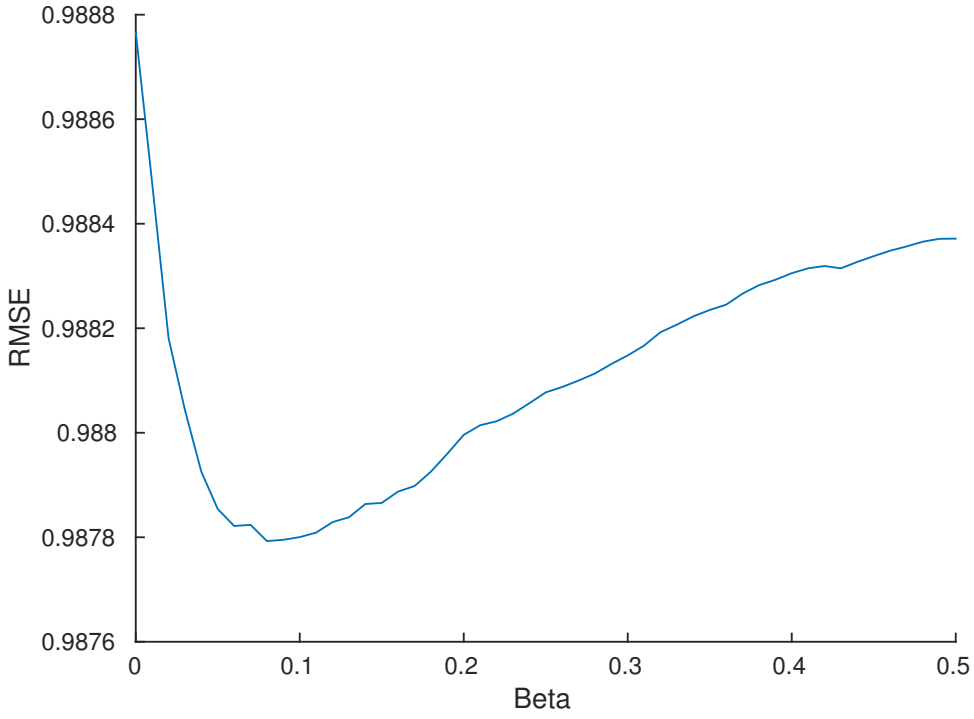

Figure 3.17: Rating prediction errors with averaging rating estimator, using the cosine similarity and proposal C.

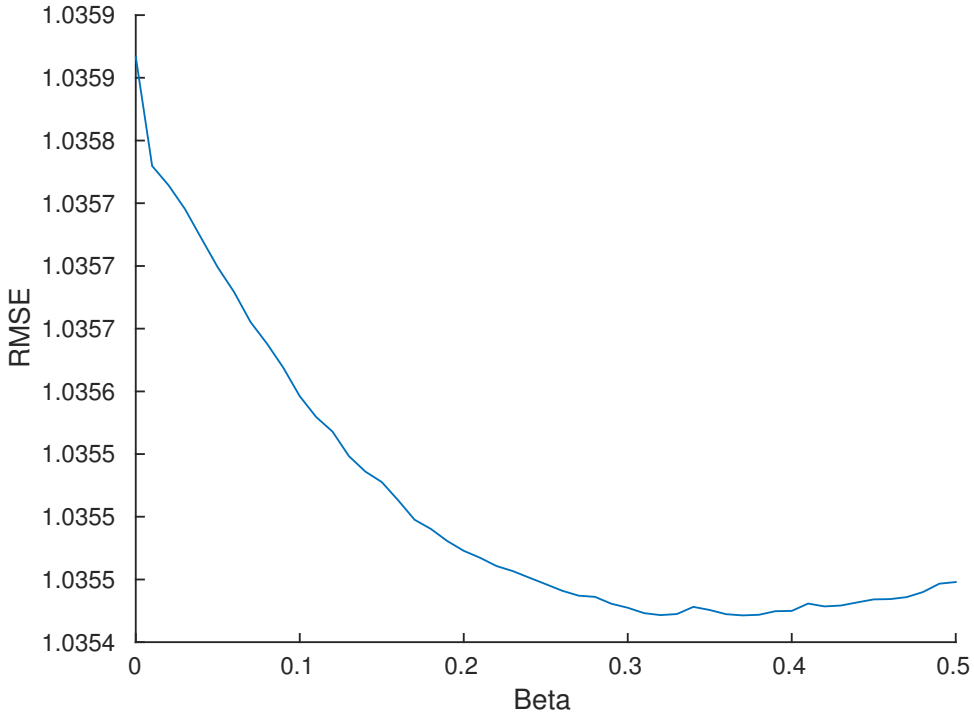

Figure 3.18: Rating prediction errors with averaging rating estimator, item-based, using proposal D.

Chapter 4
Conclusions

The results in Experiment 3 shows that one of the proposals, Proposal C in Sec. 2.3, has a positive effect on the prediction accuracy. The total error is lower when the algorithm uses a value of $\beta > 0$ compared to $\beta = 0$. The improvement is small, but is statistically significant, according to the reasoning in Sec. 3.4.1. The measured improvement here is about 1 ‰.

It is important to bear in mind that this is not an improvement made on a nearest neighbour algorithm that has first been tuned to push the limit of the RMSE down. In reality, pre-processing using rating normalisation, just to name one, would certainly be explored to optimise the recommender. However, if the step that is the *NNN* can improve a simple algorithm like this, it is reasonable to think that it can improve, more or less, an algorithm that has been well adapted to a specific problem.

As there is an improvement when the averaging rating estimator (2.9) is used, this indicates that the neighbourhood that is chosen (i.e., the neighbours that have positive weights) has changed, and that this new neighbourhood is better, if only slightly. Furthermore, the fact that there is an improvement when the weighted rating estimator (2.9) is used could depend on that the neighbourhood is better. But as the improvement is seemingly larger using this estimator, it suggests that the actual values of the weights are more appropriate when making the weighted average. However, the improvement is very small. This is relative to, for instance, how much of an effect a change in the level of sparsity can have on the errors. There are essentially two steps to the *kNN*, and there is the problem of knowing where the bottleneck is. This thesis has focused on improving the first step, the one that orders the users in the neighbourhood, which has yielded a very small improvement. The second step, the ratings estimator, could be further investigated, which has only lightly been touched upon in App. A.

It is possible that the weight matrix is most often almost as good as it can be. In fact, a quick test run was eventually made (based on Experiment 2) where the ground truth matrix, $\mathbf{W}_{\mathbf{GT}}$, replaced the weight matrix calculated from the sparse rating matrix - and did not yield a noticeably better result.

This was of course, again, under certain circumstances, but it is still reasonable to think that an available perfect theoretical similarity matrix will not be the bottleneck, when there are few ratings available for an item, or the neighbours are not very similar, even if you know exactly how similar they are.

It would be preferable to find the subset of users that actually make best use of the *NNN* part; however, due to the absence of patterns, as far as this thesis work has identified, no further investigation was made other than looking at the number of neighbours users had and their improvements. It turned out that the users on both sides, both those that improved and worsened their scores the most, had the number of neighbours in the same range, rendering the results inconclusive as to whether patterns were present or not concerning the number of items the users had rated.

A special case where the *NNN*-algorithm is hypothetically better to the standard *kNN* is when a user has rated only a few unusual items (items with few rating) but has one neighbour that has rated these items, and rated them similar to the user in question (which would imply a high similarity score, that is also trustworthy). When the user in question takes out a neighbourhood to create an estimate of one of her unrated items, she only has one "good" neighbour and only a few other ones with positive similarity weights - weights that might be inaccurate. In this case, the "good" neighbour that is accurately close possesses weights to other users, where these weights could be good proxy variables for what the similarity should be for the user in question. These similarity weights would then be used to take into account ratings made by users who the user originally (inaccurately) had an estimated zero-similarity with. However no proof has been found or provided in the thesis for this and the hypothesis is so far only speculative.

The method for sparsing the rating matrices that was used in Experiment 1 and 2 rendered a problem. The method was simply to remove the most recently made ratings, based on their timestamps. One problem was was that the MovieLens database is a fairly old one (from the 90's) which meant people did not watch a movie and then rate it immediately afterwards (as today for HBO or Netflix) but rented physical copies of movies and rated them after possibly having watched many of them. The data was furthermore collected over several years. When ratings were removed by time, entire user profiles were removed at once leaving only a few users with intact profiles while most were completely or almost completely removed. This in turn created the scenario described in the previous paragraph, but where all users, except for a few, had no ratings and thus would get the exact same estimation. In Experiment 3, however, the ratings were removed at random causing a more normal behaviour of a sparsed data set than was present in Experiment 1 and 2. This may be part of an explanation of the results of Experiment 2.

Experiment 3 showed that exploiting user preference similarity transitivity is useful in that it can yield better predictions. Experiment 1 and 2 are inconclusive as to if this effect is greater for problems involving higher sparsity. However, because of the thought experiment above, the transitivity usefulness could shine through more at higher sparsity levels as information at hand becomes more scantly available.

Chapter 5
Further work

This chapter contains some ideas for what potential future work this thesis could inspire. General ideas regarding recommender systems are discussed and one concrete suggestion is given.

5.1 Discussion

What has been presented in this thesis is a novel way of making use of preference similarity transitivity in a neighbourhood based recommender system. It has been demonstrated that some of the methods have some, albeit small, positive effect on prediction accuracy. There are however most likely more sophisticated and potentially much better methods of doing this. A natural step to take from this would be to calculate the similarity weights using several steps through a neighbourhood. Thinking of the users as a big graph with similarity edges opens up possibilities to reweigh edges using any paths that could be relevant. One could also include the items, to think of it as a bipartite graph. Essentially, that is what the data set is, a bipartite graph with two vertex sets, users and items. What the standard *kNN* algorithm does, is to create weighted edges within one of these sets by using paths that goes through the other, and applying a function to the values of the edges of the paths. This creates the similarity weights. The step of estimating ratings, is in this representation, a way of creating weighted edges between the two sets in this bipartite graph. The non-existent edges between a pair of elements in the two sets are created by stepping to the similar elements in ones own set, and weighing together the values of their edges across to the element in the other set.

But this has potential to be improved, and this thesis is of course an attempt to do so in the first part of the algorithm, by re-weighing the edges within one of these sets (users). However, as stated before, there are some indication that this is not the most crucial step. Similar ways of thinking could be applied to the creation of the rating estimation - maybe one could somehow average, weightedly, (almost) all paths one could take between a user

and an item. That is, the rating by a user of an item could be used when a rating prediction is made, for a very close user on an item if that item is also very similar to the first mentioned item.

A first step in an approach based on graph theory would be to consider the $\mathbf{W}$-matrix as an adjacency matrix. A novel way to create the alternative weights could be to multiply this matrix by itself, giving rise to a matrix containing representations of the strength of the paths of length 2 between two users in the graph (the graph that is the users and the similarity values). This is somewhat similar to proposal C. There are, in retrospect, many simple alternatives like this that could easily have been tried, and potentially yielded useful results. The resulting weights could element wise be raised to the power of a number determined by cross-validation, to become better as weights for the rating estimator. These suggestions require very few extra parameters in the model, and can potentially have a significant improving effect on the errors (as the optimal value of the power in this case is unlikely to be 1) without being prone to over modelling. Tricks like this can potentially make the choice of a similarity metric arbitrary, which could save computational time.

A piece of potential further work that is close at hand, which was touched upon in Experiment 3, is to combine the similarity weights conditionally. To have a dynamic β (and k for that matter) probably results in better predictions, if the functions are chosen well enough. It is very possible that the alternative weights could be very useful in situations where a user has few ratings, maybe mostly ratings done on "rare" items, but also at least one accurately similar neighbour. To evaluate if a high similarity weight has confidence to it may be important, as one can have a high estimated similarity to another user that is mainly due to chance. Separating user-item pairs where the *NNN* is advantageous and disadvantageous respectively, is a classification problem in itself.

5.1.1 A concrete piece of work

One concrete suggestion for a future project would be to implement for instance the Netflix grand prize winning algorithm (from 2009) and add the *NNN* similarity weight reevaluation step, to the neighbourhood based part of the algorithm. Running it on the same data set, for a range of values of β (just like Experiment 3) would potentially answer whether exploiting user preference similarity transitivity is useful, if these methods would be a way to do that. Obviously, running these algorithms using the simplest form of a *kNN* does not provide evidence for the usefulness of the proposed method, as they would not be used anyway. That is why it would be important to evaluate the proposed methods, the proposals of *NNN*, on a specific algorithm that is fairly up to date and has the potential to be used. One important thing to consider is the sparseness of the data and the size of the data space (number of items and users), as the Netflix data set was more dense than 1% entries and the number of users and items were 480000 and 18000 respectively [3]. There may be other current uses of a *kNN* (that are used on less data) which could be used to answer whether the *NNN* proposed in this thesis has a practical role.

Appendix A

Exploring the alternative regression basis for the rating estimator

This appendix contains a short discussion about the regression made, using the ratings made by the neighbours in the neighbourhood, to predict ratings.

Making rating predictions is often made by using a weighted average of the sorresponding ratings in ones neighbourhood,

$$\hat{r}_{ui} = \sum_{v \in N_i(u)} \frac{w_{uv}}{\sum_{v \in N_i(u)} |w_{u,v}|} r_{vi}, \tag{A.1}$$

where w_{uv} commonly is the Cosine Vector between the ratings made by user u and user v respectively. This is however a completely heuristic option of choosing weights for the regression. It makes sense that the ratings made by users that are deemed similar on the rest of the ratings give more weight in the rating estimator. But what if one were to try to optimise these weights? A suggestion of an attempt was made in (2.10), which is a constrained way of doing it. One could also try to set it up as a standard sum of least squares problem. As the *kNN* has two main steps to it (computing neighbourhoods and estimating the ratings) and the main part of this thesis explores the first of these steps, some work was put into investigating the latter step as well. This did however not yield results that were organised enough to be presented here, but the thought presented just above here can hopefully inspire a discussion, and maybe an experiment, in the future.

Appendix B

Matlab Code

Listing B.1: Function that computes the W' from the proposals in section 2.3.

```
function Wprime = NNNprime(W, option)
% Wprime = NNNprime(W, option)
%   Options:
%       'A' - Proposal A
%       'B' - Proposal B
%       'C' - Proposal C
%       'D' - Proposal D

    n = size(W,1);
    Wprime = zeros(size(W));

    if(option == 'A')
        for u = 1:n
            [~, sorted] = sort(W(u,:), 'descend')
                ;
            x = sorted(1);
            for v = 1:n
                if(v == x)
                    Wprime(u,v) = W(u,v);
                else
                    Wprime(u,v) = sqrt(W(u,x)*W(x
                        ,v));
                end
            end
        end
```

```
24      end
25
26      if(option=='B')
27          for u = 1:n
28              [~, sorted] = sort(W(u,:), 'descend')
                    ;
29              x = sorted(1);
30              for v = 1:n
31                  if(v == x)
32                      Wprime(u,v) = W(u,v);
33                  else
34                      Wprime(u,v) = W(x,v);
35                  end
36              end
37          end
38      end
39
40      if(option == 'C')
41          l=3;
42          sumweights = 1/sum([l:-1:1])*[l:-1:1];
43          for u=1:n
44              [sortedweights, index] = sort(W(u,:),
                    'descend');
45              for v=1:n
46                  Wprime(u,v) = sumweights*W(index
                        (1:l),v);
47              end
48          end
49      end
50
51      if(option=='D')
52          for u = 1:n
53              [~, sorted] = sort(W(u,:), 'descend')
                    ;
54              x = sorted(1);
55              for v = 1:n
56                  if(v == x)
57                      Wprime(u,v) = W(u,v)^3;
58                  else
59                      Wprime(u,v) = W(u,v)*W(u,x)*W
                            (x,v);
60                  end
61              end
62          end
63      end
64
```

```
65
66
67
68      for i = 1:n
69          Wprime(i,i)=0;
70      end
71
72  end
```

Listing B.2: Function that creates the final weight matrix W^*.

```
1  function Wstar = Wcombine(W, Wprim, alpha, beta)
2
3          Wstar = alpha*W+beta*sqrt(Wprim);
4
5  end
```

Bibliography

[1] Algorithms.io. *Provide Automated Movie Recommendations*. 2013. URL: http://http://catalog.algorithms.io/pages/page/id/31 (visited on 05/14/2015).

[2] Kantor P. B. et al. *Recommender Systems Handbook*. Springer, 2011.

[3] James Bennett and Stan Lanning. "The netflix prize". In: *Proceedings of KDD cup and workshop*. Vol. 2007. 2007, p. 35.

[4] George EP Box, J Stuart Hunter, and William G Hunter. "Statistics for experimenters: design, innovation, and discovery". In: *AMC* 10 (2005), p. 12.

[5] Hein J. *Playing Alarik: Architecture and Performance*. University Lecture. 2012.

[6] Ekstrand M.D. Konstan J.A. *Introduction to Recommender Systems*. MOOC, Coursera. 2015.

[7] Ekstrand M.D. Konstan J.A. *Introduction to User-User Collaborative Filtering*. MOOC, Coursera. 2015.

[8] Ekstrand M.D. Konstan J.A. *Variations and Enhancements*. MOOC, Coursera. 2015.

[9] Yehuda Koren. "The bellkor solution to the netflix grand prize". In: *Netflix prize documentation* 81 (2009).

[10] Ellen Spertus, Mehran Sahami, and Orkut Buyukkokten. "Evaluating similarity measures: a large-scale study in the orkut social network". In: *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM. 2005, pp. 678–684.

[11] Koen Verstrepen and Bart Goethals. "Unifying nearest neighbors collaborative filtering". In: *Proceedings of the 8th ACM Conference on Recommender systems*. ACM. 2014, pp. 177–184.