Universiteit Utrecht

Department of Information and Computing Sciences

Context-aware recommender systems

Master Thesis, ICA-3373452

August 27, 2015

Author
Floor van Steeg

Supervisor
drs. H. Philippi

Abstract

Recommender systems try to predict users' preferences for certain items, given a set of historical data. Multiple different techniques are available that make these systems accurate and one of them that delivers promising results is matrix factorization. This thesis explores how these systems work and presents a method to incorporate contextual data into a factorization technique to get predictions based on context. Specifically, a music recommender based on Candecomp/Parafac tensor factorization is proposed that uses implicit feedback collected from music listeners. The results are empirically tested and compared with other non-contextual recommender techniques. The prediction quality of the matrix factorization technique is unfortunately not improved by our proposed tensor factorization recommender on the used Last.fm dataset. However, an adjusted dataset with artificially made contextual data does get better results, but this may not reflect a real-world situation.

Contents

1 Introduction

Recommender systems try to predict what items a user will like. It estimates the user's preferences based on some information it collected. These systems are more and more present in our daily lives, whether we realize it or not.

In its most simple form recommender systems use some kind of global information to recommend items for every user at once. For example the top 10 most downloaded songs are presented to customers at online music stores, or the highest rated articles are listed on a blog's front page. These systems provide an easy way of recommending the most popular items – all that's needed is aggregating item ratings and selecting the top-k items with the highest rating.

Another way of recommending items is by serving hand curated lists. These lists are created by *domain experts*, who have knowledge of the items in their field. In this case the recommender system is a human, who predicts what users will like based on his own preferences and experience.

Both *popularity* and *expert* based recommendations treat all users as the same and exploit the global preferences or the preference/knowledge of a single person. Online stores, however, require a more *personal* approach to recommendations, due to an abundance of items available. Most brick-and-mortar stores cannot tailor the shop to the needs of each individual customer, because there is a limit to the amount of items they can store and have on display. While the popular top 10 lists can be used in physical stores, personal recommendations are simply not worthwhile.

Online stores can have a lot more items in their catalog: physical music stores may have several thousands of CDs, whereas the iTunes store and streaming service Spotify both contain multiple millions of songs [3, 56]. A large range of those items is not very popular, but together they might deliver a bigger business than the most popular items. Where previously the Pareto Principle was in effect – also known as the 80%/20% rule: the small top fragment (20%) of products delivers a large portion (80%) of the sales – for online shops the less popular niche merchandise can produce the bigger part of the business. It is therefore of interest to get the right items from these

niche markets to the right users. [68]

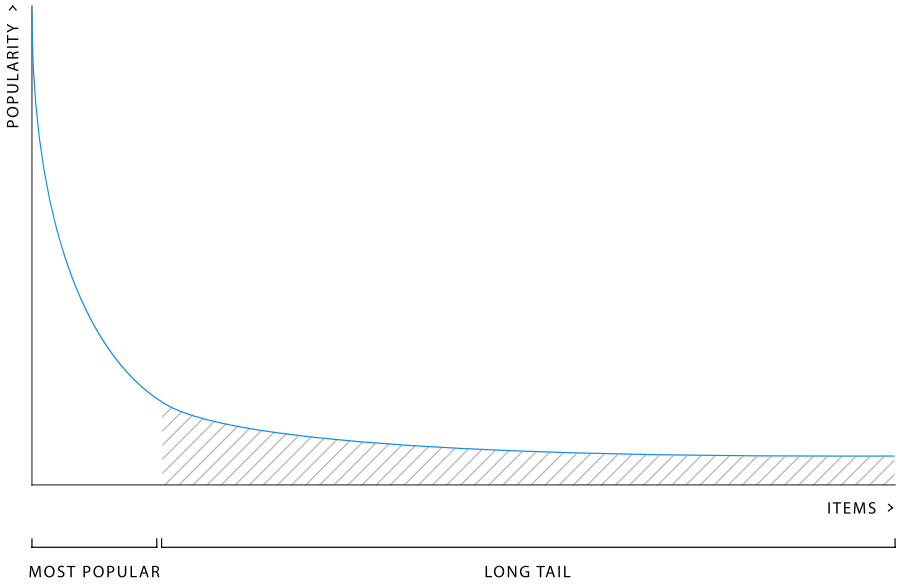

Figure 1.1: The long tail problem: getting niche items to the right users. Physical stores only display the most popular items, while online stores can provide a lot more.

This problem is called the long tail, see Figure 1.1. It shows the (sorted) popularity of different items. Where physical stores mostly just focus on the small portion of most popular items, online stores can provide the full range of less popular products in the long tail as well. Due to the benefits of serving items from the long tail and the practical impossibility of showing all these items to the user, online stores reach for personal recommender systems. [48]

1.1 Goals

There has been plenty of research towards recommender systems. In Chapter 2 we'll give a concise overview of multiple commonly used techniques and some implementations. Traditionally recommendation systems only use the historical preferences of users to recommend music. Our goal is to create a context-aware recommendation system, which uses (historical) *context* information (time of day, weather, mood etc.) as well, to recommend music for a certain moment. This way we expect to improve the prediction results and

make a system that can give better recommendations. We specifically focus on time as context, since it is easily obtainable.

Another goal is to make use of implicit feedback data, instead of the higher quality and more often used explicit feedback. We will need to take this into account when designing a recommender system, since they don't work in the same way.

Finally we would like to implement a system that is usable in real-world applications, therefore the running time and memory usage should be reasonable. We aim for a system that can make live predictions, but training can be done offline.

1.2 Overview

First we will give a synopsis of recommender systems in general (Chapter 2). Chapter 3 continues on matrix factorization, a dimensionality reduction technique with promising results. We'll talk about the underlying idea of this approach and show how to implement such a system for implicit feedback datasets with alternating least squares, based on previous work by Hu et al. [28]. Then we'll analyze context in general (Chapter 4), explore tensor decomposition techniques (Chapter 5) to add context information into the model and show how we implement this in our application. Finally we will discuss our evaluation method and the results collected from our implementation in Chapter 6.

1.3 Notation

There are a lot of different notations used in this research field, so to avoid confusion we'll list the notations we have adopted here. They are largely similar to Kolda in [32].

- Higher order tensors uppercase bold cursive: $\boldsymbol{\mathcal{A}}$
- Matrices uppercase bold: $\mathbf{A}$
- Vectors lowercase bold: $\mathbf{a}$
- Scalars lowercase regular: a
- Subscripts denote rows, columns and elements and use the same notation as above, thus: column i of matrix $\mathbf{A}$ is shown as $\mathbf{a}_i$ and element j in vector $\mathbf{a}_i$ is shown as a_{ij}

- Uppercase characters denote the upper bound of indices: $n \in 1 \dots N$
- Predicted rating/preference for item i by user u in context t is shown with a hat: $\hat{r}_{uit}$
- Regularization terms: λ

2 Recommender systems

Personal recommender systems try to make educated guesses about what items a user likes by looking at the users' historical preferences. This means that by using a (large) dataset of (historical) preferences between users and items, recommender systems try to estimate the preference values for which currently no data is available. The system needs to find relations in the dataset such that it gets knowledge about how much the user likes certain items. There are multiple ways of finding these relations and we can categorize recommender systems by the two main methods they use: *content based* or *collaborative based*. We'll cover both systems in Sections 2.2 and 2.3 respectively, but first we'll discuss what forms the basis of each recommender system: the *utility matrix*.

2.1 Utility matrix

As mentioned before, recommender systems use the preferences of users for certain items to recommend other items the user might like. (Here "items" is a generic term and most systems are focused on and optimized for just a specific type of items, e.g. music, movies, news articles, etcetera.) These known preferences are stored in a so-called utility matrix, where each user-item pair is comprised of a value depicting the degree of preference of that user for that item. An example is given in Table 2.1, where the ratings for different movies are given. A rating of 5 stars represents that a user likes a movie, whereas a low rated movie is not liked by the corresponding user.

We can acquire the data to fill the utility matrix in two ways: with explicit ratings from users, or with implicit ratings inferred from user behavior. *Explicit feedback* requires extra user action. After watching a movie a user might go to a movie database website such as IMDB and express his appreciation (or criticism) by giving the movie a rating in the form of 1 to 5 stars. These ratings are valuable since the user himself explicitly provides them, however, they might be biased since only the data from users taking

	LOTR	Harry Potter	Pulp Fiction	Titanic	Forrest Gump
Alice		1		5	4
Bob		5		1	2
Cherryl	5	4		1	
Dave	1		4		5
Eddy	5	5		2	
Frank		2		4	3

Table 2.1: Utility matrix, with user ratings for different movies. Empty cells denote that the user has not rated the corresponding movie.

the effort to rate is represented.

The other method, *implicit feedback*, has the advantage of not requiring any extra interaction from the user: the recommender system can work behind the curtains and the user doesn't need to worry about it, it just works. We can for example keep track of which TV shows a user watches and how long he is watching them as a measure of preference. A user that watches a specific show frequently indicates that he likes that show. This type of feedback is generally easier available than the higher quality explicit ratings. A typical inconvenience however, as we will discover later in Section 3.3, is that the blank positions in the utility matrix are actually zero. In the explicit feedback case we now nothing about those user-item pairs, but with implicit feedback it might mean the user dislikes the item

Whether we choose implicit or explicit feedback, utility matrices are often very sparse (much sparser than in Table 2.1), which means that there are a lot of unknown values. This is due to the often vast amount of items and users and the relatively low number of ratings per item. The goal of a recommender system is to fill in these unknown values of the utility matrix. For example: will Alice like The Lord of the Rings? As we will see in the next sections, a content based recommender might for instance notice that LOTR is quite similar to the Harry Potter movie, but Alice doesn't like that one, so she probably won't like The Lord of the Rings either.

Formally the recommender system is modeled by a function F that predicts the unknown rating $r_{ui} \in R$ for item $i \in I$ by a user $u \in U$:

$$F : U \times I \to R \tag{2.1}$$

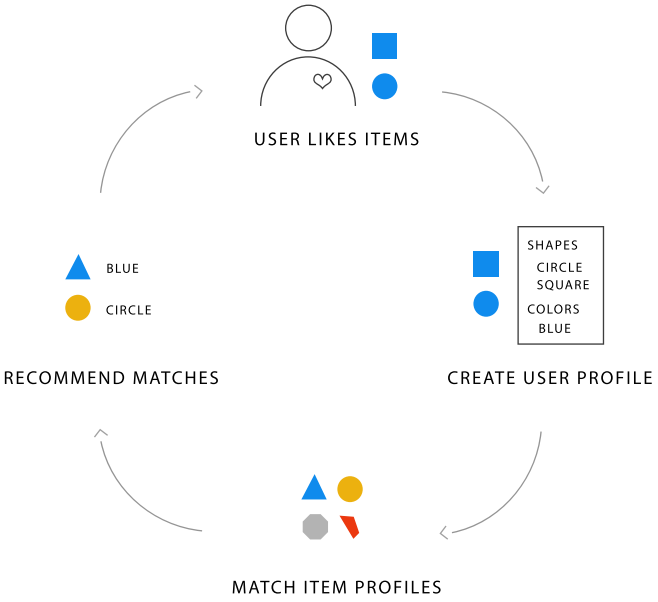

Figure 2.1: Content based recommender system

2.2 Content based

Content based recommender systems look at the properties of items or users in the dataset. We can for instance categorize books in different genres: thriller, romance, adventure etc. If a user has a high preference for books in the thriller genre, he probably also likes other books in that genre. We can do the same for users: if users aged < 10 generally like animation movies, another user aged < 10 would probably also like animation movies. Recommender systems that use a content based approach create item and/or user profiles based on their properties and use these profiles to find other similar items/users, see Figure 2.1.

Content based recommender systems have a number of advantages compared to other systems [40]:

1. The recommender system can be used right away: it does not depend on ratings from other users.

2. It's able to recommend new items that no user has heard of before, simply because those items have similar profiles to another known item.

3. The recommended items are easy to explain, it is possible to list the features of the item profiles and the similar items themselves.

	adventure	adventure	crime	romance	romance
	fantasy	fantasy	drama	drama	drama
	LOTR	Harry Potter	Pulp Fiction	Titanic	Forrest Gump
Bob		5		1	2
Dave	1		4		5

Table 2.2: Utility matrix with item profiles added.

However it is not always trivial to create good profiles for all items. For example it can be hard to extract features from images or songs since their properties are often vague and subject to interpretation (is it rock? pop-rock? electronic-pop-rock-with-al-little-bit-of-punk?). Furthermore, users won't discover items outside of their profile. The classic exploration versus exploitation problem relates to many recommender systems and prevents users to step out of a bubble of similar items. Items a user might like, but have a completely different profile than currently liked items won't be recommended. [48]

Example

In Table 2.2 the item profiles for each movie are added to the example utility matrix. They consist of two genre classifiers and can be used to compare the movies. What will be Bob's value for the The Lord of the Rings movie? Since he liked Harry Potter and its item profile matches that of The Lord of the Rings, Bob will probably also like the The Lord of the Rings movie. On the other hand, Dave will most likely not appreciate Harry Potter, but will probably like Titanic better, whose item profile matches Forrest Gump's. Note that we use genre here for illustrative purposes, there can be many different properties added to the item-profile, such as director, actors, writers, etcetera.

One approach to getting the predicted preferences is by comparing item and user profiles as vectors. Each element of the item profile vector represents a property and is either 0 or 1. A 1 if the item matches that property, 0 otherwise. For users we can do the same while using the rating weights for the individual properties. In Table 2.3 we highlight the profile vectors for the movies Titanic and Harry Potter and the user Dave.

As we can see from the results, Dave's vector is more closely related to Titanic's than to Harry Potter's. Measuring the similarity between two vectors can be done with for example the cosine distance, which we will

Titanic		Harry Potter		Dave		
adventure	0	adventure	1	adventure	1/5	= 0.1
fantasy	0	fantasy	1	fantasy	1/5	= 0.1
crime	0	crime	0	crime	4/5	= 0.80
drama	1	drama	0	drama	9/10	= 0.90
romance	1	romance	0	romance	5/5	= 1

Table 2.3: Profile vectors for Titanic, Harry Potter and user Dave.

further expand upon in the next section. When the similarities between all users and their not-rated movies are calculated the recommender system can recommend which movies the user should go see, by simply sorting the resulting values.

2.3 Collaborative filtering

Where content based recommender systems find similar properties, collaborative filtering (CF) systems find similar ratings. These systems compare users and items only by past user behavior (i.e. the rows and columns of the utility matrix), without looking at their properties. The term *collaborative filtering* was first used by Goldberg et al., authors of one of the earliest recommender systems: Tapestry [22]. The system was created to filter e-mails from mailing lists and newsgroups, where users collaborated by writing annotations about the messages. Later, more automated systems were developed like GroupLens [33], which gave personalized recommendations on Usenet posts (using k-Nearest Neighbors, see Section 2.3.1). [53, 18]

Of course, for those types of items for which it is difficult to get the right properties and features, a collaborative approach is more suited than a content based one. Another conceptual advantage over content based methods is that collaborative filtering recommendations are based on the quality of items, not on just the properties some items might have in common. If two items share the same properties, that does not necessarily have to mean that a user who likes one will like the other as well, since it might be a low quality item. It isn't likely that a collaborative filtering recommender on the other hand, would suggest items that are rated low by other users, despite that item having features the user likes. High ratings are typically an indication of high quality, or high preference. [19]

Furthermore, collaborative methods don't have the problem of not being

able to recommend items outside of the user's profile. When other (similar) users simply like those items outside of another user's profile they can be considered for recommendation.

A disadvantage however is the need for ratings from other users. This is called the cold start problem: the system can only recommend items if there are enough ratings to calculate similarities. New items and new users cannot be included immediately, since there is no preference information available right away.

There are two main approaches to CF: memory-based (more commonly known as neighborhood-based) and model-based methods. Neighborhood methods are better at finding similarities in local relations, whereas model-based approaches learn more general structures that cover a larger part (if not all) of the data. [34]

2.3.1 Neighborhood

The neighborhood method is based on the idea that if certain users rate some items more or less the same, they will rate other items the same as well. It looks in the neighborhood of similar users and recommends items that the k-Nearest Neighbors (KNN) have preferred. Since the users have preferred similar items in the past, it is likely that they rate items similarly in the future as well, see Figure 2.2. This form of recommending closely resembles an automated kind of word of mouth, where users endorse items to each other.

Of course we can do the same thing for items: if a user likes an item, the recommender system will look in the neighborhood of similarly rated items to recommend the best options. The advantage of this item-oriented neighborhood method is that the predictions are easier explainable. Users know the item they have preferred in the past where an item-oriented recommendation is based on, whereas they are not familiar with the similar users a user-oriented recommendation is based on [34] – they can be complete strangers.

The nice thing about neighborhood-based methods compared to model-based systems is that they don't require costly recomputations when new items or users are added. In real world applications items and users are constantly being added to the database. A neighborhood approach only needs to find the similar neighbors for the new items and users, it can keep the already computed neighbors for other items/users. The same is true for the addition of new ratings: only those items/users with new ratings need to recompute their similarities [19]. For model-based systems this often requires retraining, which is computationally expensive.

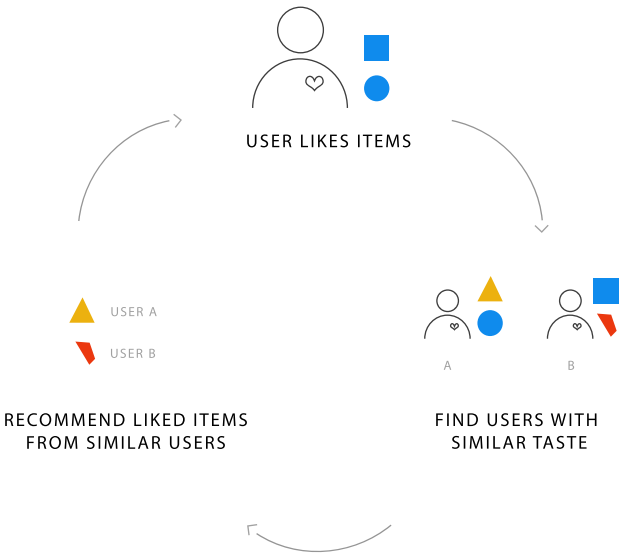

Figure 2.2: Collaborative based recommender system

Furthermore, neighborhood-based recommender systems are relatively easy to implement and there are few parameters to estimate compared to model-based systems. Where model-based recommenders need extensive training with multiple free parameters, the simplest form of neighborhood-based systems just has a single parameter k for the number of neighbors [7]. The value for k must be carefully chosen though: if it is too small, the system will be affected by noise. If on the other hand k is too large, neighbors get included that should not have influence at all.

Implementation

The performance of KNN is largely influenced by the way similarities are calculated. To measure similarity, we can compare the rows or columns from the utility matrix and calculate the distance between these vectors. This can be done in multiple ways.

The simplest method is the Jaccard Distance, which bases similarity on the number of rated items that users have in common. Users with many rated items in common are considered similar, whereas users that rated different sets of items are not similar. This does not always make sense intuitively, since it discards all information regarding the degree of preference for items. Therefore, users that rated the same items completely opposite are still considered similar. If we for instance look at Table 2.1 again, Titanic and Harry

	LOTR	Harry Potter	Pulp Fiction	Titanic	Forrest Gump
Alice		$-7/3$		$5/3$	$2/3$
Bob		$7/3$		$-5/3$	$-2/3$
Cherryl	$5/3$	$2/3$		$-7/3$	
Dave	$-7/3$		$2/3$		$5/3$
Eddy	$3/3$	$3/3$		$-6/3$	
Frank		$-3/3$		$3/3$	$0/3$

Table 2.4: Normalized utility matrix. Blank values can be treated as 0, which is considered neutral.

Potter would be considered similar movies, while they actually get opposite ratings. It really depends on what type of data is collected in the utility matrix, if it contains information about recent purchases (where each purchase means the user likes the item) the Jaccard Distance might be a good measure of similarity.

A different measurement is Cosine similarity, which can be calculated by dividing the dot product of two vectors (for user u and v) by the product of their lengths.

$$cos(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|} = \frac{\sum_{i=1}^{N} r_{ui} \times r_{vi}}{\sqrt{\sum_{i=1}^{N} r_{ui}^2} \times \sqrt{\sum_{i=1}^{N} r_{vi}^2}} \tag{2.2}$$

It conveniently measures the angle between two vectors: if $\mathbf{u}$ and $\mathbf{v}$ are exactly the same (not taking their magnitude into account) it will output $cos(\mathbf{u}, \mathbf{v}) = 1$, while if they are exactly opposite it will give $cos(\mathbf{u}, \mathbf{v}) = -1$. Note however that there are a lot of missing values in the utility matrix. We could treat those as zeroes, but that would bias them towards disliking an item. Another problem is that users often have differently scaled ratings, for example in a range of 1 to 5, some might consider 3 a positive preference, while others might think it's a negative rating. A way to solve both problems is to normalize all values by subtracting the user's average rating value from its ratings. This has the effect that unknown values are considered more neutral and the vectors are better comparable overall. The ratings in Table 2.4 are normalized. Now, if we for example compare Alice and Bob according to the Cosine similarity, we get:

$$\frac{(-7/3)\times(7/3)+(5/3)\times(-5/3)+(2/3)\times(-2/3)}{\sqrt{(7/3)^2+(-5/3)^2+(-2/3)^2}\times\sqrt{(-7/3)^2+(5/3)^2+(2/3)^2}}=-1$$

Whereas the similarity between Bob and Cherryl will be:

$$\frac{(2/3)\times(7/3)+(-7/3)\times(-5/3)}{\sqrt{(7/3)^2+(-5/3)^2+(-2/3)^2}\times\sqrt{(5/3)^2+(2/3)^2+(-7/3)^2}}\approx 0.6282$$

These are expected results if we take a look at the utility matrix: Bob and Cherryl rate almost the same, while Alice and Bob rate completely opposite. Of course we can again do the same with the columns of the matrix, when comparing items. This way we'll discover for example that Harry Potter and Titanic are quite dissimilar with regard to their ratings.

The commonly used Pearson correlation coefficient directly combines the Cosine similarity function with this normalization and is defined as:

$$\rho_{uv}=\frac{\sum_{i=1}^{N}(r_{ui}-\bar{r}_u)\times(r_{vi}-\bar{r}_v)}{\sqrt{\sum_{i=1}^{N}(r_{ui}-\bar{r}_u)^2}\times\sqrt{\sum_{i=1}^{N}(r_{vi}-\bar{r}_v)^2}} \tag{2.3}$$

After computing the similarities between all users, we can select the nearest neighbors of each user u: the k users with highest similarity to u. The set $\mathcal{N}(u)$ contains these nearest neighbors of u. In order to get the prediction $\hat{r}_{ui}$ we need the subset of neighbors that have rated item i, which we'll call $\mathcal{N}_i(u)$. Then $\hat{r}_{ui}$ is given by a weighted average of the preferences for item i by all $v\in\mathcal{N}_i(u)$.

$$\hat{r}_{ui}=\frac{\sum_{v\in\mathcal{N}_i(u)}\rho_{uv}r_{vi}}{\sum_{v\in\mathcal{N}_i(u)}\rho_{uv}} \tag{2.4}$$

Again this can be normalized, since in Equation 2.4 the r_{vi} values might be scaled differently for different users as well.

$$\hat{r}_{ui}=\bar{r}_i+\frac{\sum_{v\in\mathcal{N}_i(u)}\rho_{uv}(r_{vi}-\bar{r}_v)}{\sum_{v\in\mathcal{N}_i(u)}\rho_{uv}} \tag{2.5}$$

In [34] instead of just accounting for the mean user ratings, a baseline estimate is added to the prediction, which includes adjustments for global

effects in the rating dataset. This avoids predicting a too high preference value for globally low-rated items that happen to have many neighbors with high average ratings.

$$b_{ui} = \mu + b_u + b_i \tag{2.6}$$

Where μ equals the average rating over all items and b_u and b_i denote the deviation from μ for the ratings from user u and for item i respectively. Both b_u and b_i are estimated by solving the least squares problem:

$$\min_{b*} \sum_{u,i} (r_{ui} - \mu - b_u - b_i)^2 + \lambda(\sum_u b_u^2 + \sum_i b_i^2) \tag{2.7}$$

Plugging 2.6 into 2.5 gives us the following estimation function:

$$\hat{r}_{ui} = b_{ui} + \frac{\sum\limits_{v \in \mathcal{N}_i(u)} \rho_{uv}(r_{vi} - b_{vi})}{\sum\limits_{v \in \mathcal{N}_i(u)} \rho_{uv}} \tag{2.8}$$

Since the utility matrix is sparse and some items are rated by just a few users, we need to check if the predictions are reliable. If lots of users prefer an item, that item might be a better option than an item that gets a high rating from just a single user. That's why a support factor w_{uv} is introduced by many nearest neighbor systems and is included in the similarity weight s_{uv}.

$$s_{uv} = w_{uv}\rho_{uv} \tag{2.9}$$

Several implementations are considered for w_{uv}, typically in the user-based case based on how many rated items two users have in common and for the item-based case based on how many users rated an item. [27, 19, 57] For example:

$$w_{uv} = \frac{n_{uv}}{n_{uv} + \lambda} \tag{2.10}$$

Where n_{uv} equals the number of items that two users u and v rated in common and λ is a scaling factor: the higher λ the more important is the amount of commonly rated items.

Item and user-based

As mentioned before, estimating the preference of user u for item i can be done by either finding similar users and getting the average rating for i from the neighbors, or by finding item neighbors for i and average the ratings for

those items given by u. In the item-based case basically the same steps as in the user-based neighborhood method are used. An adaptation of Equation 2.8 for the item-based case therefore gives:

$$\hat{r}_{ui} = b_{ui} + \frac{\sum_{j \in \mathcal{N}_u(i)} s_{ij}(r_{uj} - b_{uj})}{\sum_{v \in \mathcal{N}_u(i)} s_{ij}} \tag{2.11}$$

In order to find item recommendations for some user u, we need to predict the ratings for all unrated items by u. In other words: we need to predict the blanks for row u in the utility matrix. In the user based case once we have found the similar users, we can compute the predicted preferences of u for all items. Whereas in the item-based case, if we find the similar items for the first unrated item, we can only predict the rating for that item. We need to find similar items for all other items, before we can recommend items to u.

On the other hand, items are better comparable than users. For instance, where an item mostly belongs to a single genre, a user can have a preference for multiple genres. Note that these underlying properties are inferred from the rating behaviors from users and not collected from their meta data itself. Since it is harder to find users with the same preference for multiple genres, than it is to find items with the same single genre, finding similar items can get better results [48]. However, this may as well be a drawback, since only items similar to already liked items are recommended. This gives us the same problem as with content-based approaches: the user is not likely to explore items outside of their personal bubble [19]. With user-based neighbors such items outside of the comfort zone *can* be recommended, exactly because a similar user may like multiple genres. The choice basically boils down to an exploration versus exploitation problem.

Clearly in favor of the item-based approach, as said in the beginning of this section, is that they can better explain their predictions. A user is not familiar with their neighbors, these are mostly just strangers, but he does know what items he has used or rated before. An item based system can therefore explain a prediction by listing the items it is based on. Compare for example: "the movie LOTR is recommended to Bob, because he liked Harry Potter" and "the movie LOTR is recommended to Bob, because Eddy liked it as well". The second case does not mean anything to Bob if Eddy is a stranger. Note that word of mouth recommending only works if the recommending party is a close friend, family or otherwise an authority. In [62], the importance of explanations for recommendations is discussed further.

The choice between item and user-based is often due to their performance.

Both have the same computational complexity, but if a system has more items than users, than a user-based approach is preferred, since it is less expensive to compute the similarities between all users. This is of course the other way around when there are more users than items. [19]

Further reading

We've only covered the basics of neighborhood-based collaborative filtering here – numerous variations and improvements to the steps in calculating the predictions exist. For example Stefanidis et al. [57] consider domain experts and close friends as well, instead of just similar users. Or Töscher et al. [63] calculate similarities by solving a regression problem and the neighborhood method is combined with matrix factorization (Chapter 3) to increase prediction accuracy.

See the comprehensive survey by Desrosiers and Karypis in [19] for more variations in similarity measures and normalization techniques amongst others.

2.3.2 Latent factors

Where memory-based recommenders use all rating information directly for predicting, model-based systems first create a predictive model, which is then used to estimate preferences. This model is created by learning from the training dataset (the utility matrix), where it finds hidden patterns and features. These are the latent factors, describing the characteristics of the users and items (Figure 2.3). In comparison with content-based and neighborhood-based recommenders, models find the different features of users and items (like content-based) by only looking at the rating data (like neighborhood-based). It creates the user-profiles and item-profiles such that in a new latent factor space both entities can be compared. Note that, again, no meta data from items or users is used; only the rating behavior.

Model-based systems are often more robust against the sparsity of the utility matrix and can give superior predictions with little available data than neighborhood methods. On the other hand, training can often take a long time and the models are susceptible to overfitting, which should be accounted for [60, 19].

There are multiple different approaches that have been used to learn the models from a dataset, dividable into two main classes: classification and dimensionality reduction.

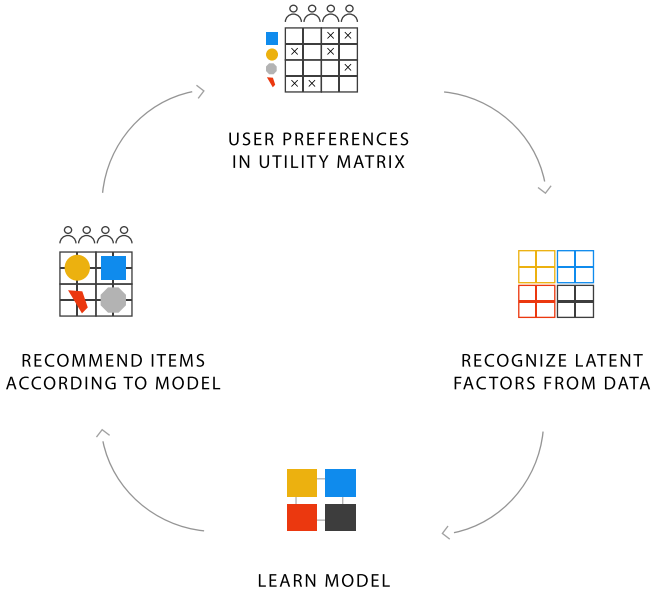

Figure 2.3: Model based recommender system

Classification

Classification methods, such as *Bayesian Networks* [45, 10], *Support Vector Machines* [24, 66, 44], *Neural Networks* [31] and *Restricted Boltzmann Machines* [52], classify an item into a certain category based on the item's features. We can use these classifiers as content-based recommenders (using the properties of the content), but it is also possible to apply them in a collaborative filtering approach. This works by viewing users as attributes and their ratings as the values for these attributes [11].

Classification algorithms can be used if preference data in the utility matrix is categorical [60]. We therefore need to convert the collected implicit or explicit feedback into separate classes. Miyahara and Pazzani in [41] for instance turn the preference data into binary categories: $c \in \{like,\ dislike\}$. Since this removes the degree of preference, it may lead to inferior predictions. It is also possible to just have each available rating represent a separate class (i.e. $c \in \{1, 2, 3, 4, 5\}$). Since the ratings are ordered, this is an ordinal classification. These multiclass systems might require some alterations to the classification algorithms if they're binary by default.

Neural network In a neural network recommender system, just like with k-Nearest Neighbors, there is an item and user-based approach. We can

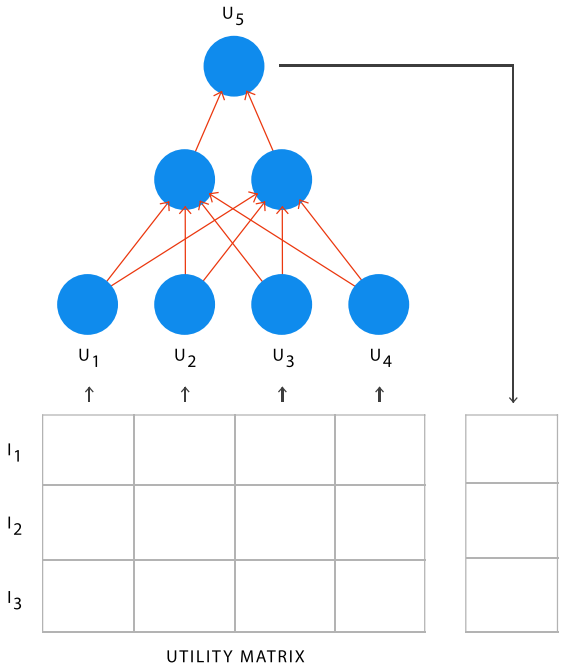

Figure 2.4: Neural network example. The ratings for items $(I_1, I_2, I_3, \ldots)$ from the other users $(U_1, U_2, U_3, U_4, \ldots)$ are used as input, which the neural network transforms using weights on the edges and hidden nodes into the predicted rating for U_5.

create a neural network for each user (with as input nodes all other users – see Figure 2.4), or a neural network for each item (with as input nodes all other items).

In the user-based case, training and recommending for user u goes as follows: when feeding the training data into the input nodes, the network learns the weights for its hidden nodes and their connections, such that the output (approximately) corresponds with the ratings u gave. The resulting model describes the behavior of the target users preference, determined with the other users' ratings. If we want to know the predicted rating for an item i that u has not rated yet, we use the ratings for i from the other users as input and get the result $\hat{r}_{ui}$ from the output node. The same works the other way around for the item-based case.

Since using all users or items as input may have an impact on performance, Kim et al. [31] choose similar users (or similar items) as input for the networks. This gives better results than using all users as input (or a random sample), as irrelevant items and users are not used for the prediction.

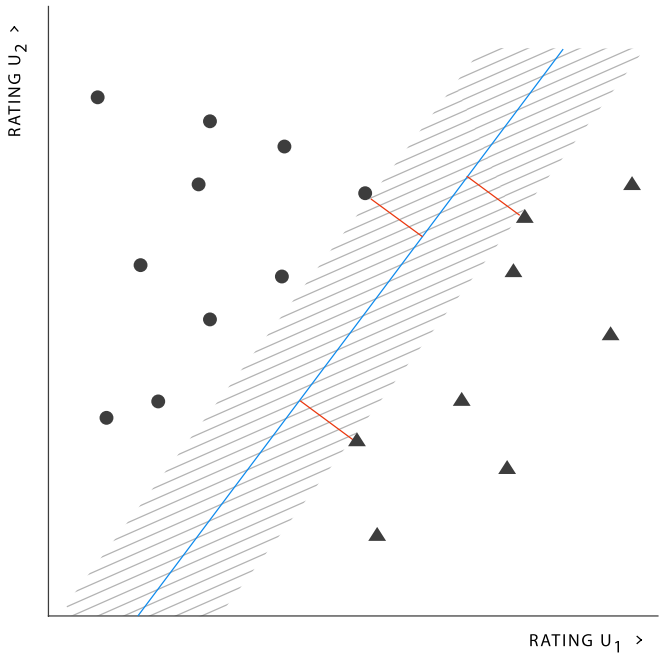

Figure 2.5: Support vector machine example. The ratings from two users U_1 and U_2 are plotted in the graph, while each rating is displayed as a • (U_3 liked the item) or a ▲ (U_3 disliked the item). It finds the hyperplane that separates both classes with the largest margin. Unknown ratings are classified according to which side of the hyperplane they lie on (again using the ratings from U_1 and U_2)

Support vector machine Likewise, we can create support vector machines (SVM) for each user/item. The SVM tries to separate data into two clusters by constructing hyperplanes that have a maximum distance to the data from both categories. Since the SVM classifies into two spaces, we need to convert the ratings into binary data (*like*, *dislike*), or use multiple machines per user. For instance in [24], they choose to have 4 SVMs: one for training if items lie in class 1 (positive) or class 2-5 (negative), one for class 1-2 (positive) or 3-5 (negative), another classifies 1-3 (positive) and 4-5 (negative) and the last distinguishes 1-4 (positive) and 5 (negative). With these 4 binary SVMs it is possible to classify an item into one of the five classes, representing a rating from 1 to 5.

To learn the SVM for a user u, the model is fed the other users' ratings for all items that u rated. Then a hyperplane is found that splits the multi-dimensional space into two parts. One part consists of all positives (user u

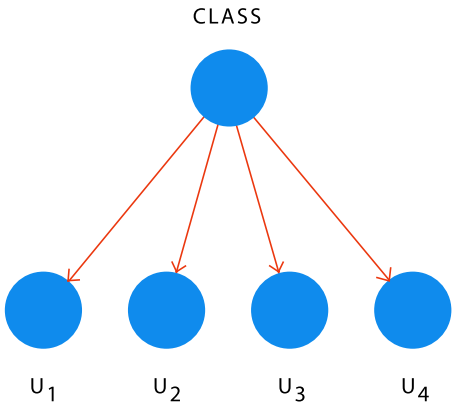

Figure 2.6: Naive Bayesian Network example.

liked the item) and the other consists of all negatives (user u disliked the item). These like and dislike results are for a binary system, for multiple classes the negative and positive spaces work as described above.

When predicting a rating for item i, we simply use the ratings from the other users and let the hyperplane decide to which class i belongs. See Figure 2.5 for an example.

Bayesian networks Bayesian networks use probabilistic graphical models to predict ratings. Several strategies can be used to create the model, they differ on how the parameters or the structure is learned and how the features are selected. The network structure is a directed acyclic graph (DAG) that encodes the dependencies between nodes. Each node represents a variable defined by a set of mutually exclusive states: its state space. In a user-based approach these variables are the different users, with the separate rating classes as their state space. A node in the graph is conditionally independent from its non-descendants given its parents. The root node represents the class we are looking for (i.e. the rating from the current user).

A simple Naive Bayes classifier assumes all features are independent, given the class. This means that the corresponding DAG contains a root node with arcs to all users, but no arcs between them (see Figure 2.6). We can calculate the probability of a certain user-item pair (u, i) being of a class c, given the ratings that other users gave i. If we do that for all of the available classes, we can simply choose the class with the highest probability (maximum a posteriori hypothesis) as our prediction. Formally (using 5 star rating classes):

$$\hat{r}_{ui} = \arg \max_{c \in \{1,2,3,4,5\}} P(\, c \mid r_{v_1 i},\ r_{v_2 i},\ \ldots, r_{v_N i}\ )$$

Where $r_{v_1 i},\ r_{v_2 i},\ \ldots,\ r_{v_N i}$ are the ratings for i by other users. Given the independence assumption from the Naive Bayes classifier, this gives:

$$\hat{r}_{ui} = \arg \max_{c \in \{1,2,3,4,5\}} P(c) \times P(\, r_{v_1 i} \mid c\,) \times P(\, r_{v_2 i} \mid c\,) \times \cdots \times P(\, r_{v_N i} \mid c\,)$$

Both $P(c)$ and $P(\, r_{v_j i} \mid c\,)$ can be learned from the observed training data [41, 60, 59, 2], such that the estimated preference for unknown items can be predicted.

In [29, 17] they use mixture models (a special case of Bayesian Networks), which cluster the users based on the way they rate. The Expectation Maximization algorithm learns the conditional probabilities ($P(\, r \mid c\,)$) and the number of classes (the state space of the root node, i.e. the number of classes the users are divided in) is set such that the likelihood of the model given the data is maximized (using the Bayesian Information Criterion to penalize the size).

However, the assumption that features are independent is not always true, there can be correlated attributes. That's why often more complex Bayesian Networks are used. For example, Breese et al. [10] create a model where each node corresponds with an item and the conditional probability tables for each node are represented by a decision tree. The network structure is learned by the algorithm to find dependencies between items. It results in a network where the parent nodes of each node are the best predictors for its preference value.

Finally we would like to note that k-Nearest Neighbors is of course a classification method as well, but it doesn't generate a model. Most of the calculation and decision making is done when predicting, which makes the prediction step more expensive compared to model-based classification algorithms.

Dimensionality reduction

Since they typically have to deal with a lot of features (i.e the rating from each user is a feature of an item), recommender systems suffer from the *curse of dimensionality*. That means that the performance of learning algorithms (such as the before mentioned neighborhood and classification systems) decreases when the number of features increases. For example, the similarity between items becomes less meaningful with a high dimensionality, while

the computation time for these systems is often increased with the number of features [15]. Above all that, the actual useful rating information in the utility matrix is very sparse as well.

To deal with these problems, dimensionality reduction techniques are used. These techniques, such as *Principal Component Analysis* (PCA), *Linear Discriminant Analysis* (LDA) and *Singular Value Decomposition* (SVD), simplify the data model and transform the higher dimensional space into a low-dimensional representation [2]. This way we can get a more compact representation of the data, where noisy and irrelevant features are discarded, while the important data is kept.

While dimensionality reduction can be used as a preprocess step – another memory or model-based technique then uses the low dimensional result as its input – the results are found to be very useful for predicting preferences directly as well [2]. They have recently gained much popularity, thanks to their accuracy and scalability: it is possible to quickly generate recommendations of high quality. The trade-off, however, is that these systems require expensive training steps to create the reduced dimensional space.

The most popular technique is based on the SVD algorithm, which decomposes a matrix $\mathbf{A}$ into two orthogonal matrices $\mathbf{U}$ and $\mathbf{V}$ and a diagonal matrix $\mathbf{\Sigma}$:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

SVD originates from the late 19th century, when it was discovered by multiple authors independently [58]. Note that this decomposition is exact: we get the original matrix $\mathbf{A}$ back from the multiplication. Unfortunately, SVD is undefined for matrices with unknown values, like the utility matrix. Earlier algorithms tried to fill the missing values before applying SVD, but this can make the matrix inaccurate. Furthermore, since a lot of new data is added this way, computing the SVD of these large dense matrices is very expensive.

More recently, matrix factorization techniques based on SVD have been used, which use only the observed ratings from the utility matrix [37]. This technique was first detailed by Funk [21] and has become very popular since [61]. It factorizes the utility matrix into two smaller matrices. Multiplying these smaller feature matrices $\mathbf{X}$ and $\mathbf{Y}$ results in an approximated utility matrix, where the previously unknown values are now predicted (Figure 2.7).

Intuitively the $\mathbf{X}$ matrix contains the user features, while the $\mathbf{Y}$ matrix contains the item features. A movie feature could be an obvious feature like *action* vs. *drama*, but might also be something that is not easily interpretable. The user features represent how much the user likes the corre-

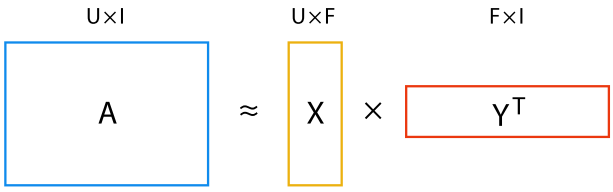

Figure 2.7: Matrix factorization example. The utility matrix $\mathbf{A}$ is decomposed into a $users \times features$ matrix $\mathbf{X}$ and an $items \times features$ matrix $\mathbf{Y}$.

sponding item features [36].

Since both feature matrices are constrained to a specific size, only the most influencing features are kept while learning. In order to prevent overfitting of the sparse training data, matrix factorization techniques need to regularize the model. Chapter 3 will further explore the details of PCA, SVD and SVD-like factorization techniques and their relation, in order to give a better understanding of how these methods work.

Factorization methods offer a more accurate and more scalable alternative to neighborhood models. Yet, a lot of commercial recommenders are based on the latter. The reason for this is not only that neighborhood models are simpler to implement and more intuitive, but also that they are easier explainable. It is hard to extract meaning from a learned factorization model, it is like a *black box*: we don't exactly know what the model has learned. As explained before, for neighborhood models this is much simpler. Furthermore, it is straightforward to add new users or items to the neighborhood system. Matrix factorization techniques, however, require an expensive recomputation of the feature matrices or use a more advanced folding-in technique to allow new users or items. [36, 60]

2.3.3 Netflix Prize

In October 2006, on-demand media streaming service Netflix launched a competition to beat the accuracy of their recommender system by 10%. After three years, in September 2009, Netflix awarded the prize of $1,000,000 to the BellKor's Pragmatic Chaos team, who was the first to beat the challenge. During the competition Netflix awarded progress prizes to the best algorithm thus far, provided it improved upon the previous progress prize by at least

	Advantages	Disadvantages
Content-based	- Does not require ratings from other users - Easy to explain recommendations - Able to recommend new/unpopular items	- Requires domain knowledge - Not based on item quality - Hard to recommend items outside user profile
Memory-based	- Easy to explain recommendations - Intuitive and simple - Easy to add new data	- Problems with sparsity - Hard to recommend for new items and users
Model-based	- Able to discover latent features - Better accuracy - Highly scalable	- Model building is expensive - 'Black box': not immediately clear what is learned

Table 2.5: Overview of recommender system techniques.

1%.

The challenge gave a big boost to research in the field of collaborative filtering, since it gave access to a dataset containing 100 million movie ratings, which was the first of its size that was publicly available. The large-scale database attracted thousands of scientists and students. [34, 37]

While model-based methods outperformed memory-based ones [34], the algorithms with best results used a blend of multiple techniques. This way, the different methods can complement each other and provide a good overall prediction. Bell et al. [8] show the results of different techniques and their combinations on the Netflix Price dataset.

2.4 Summary

In this chapter we've seen multiple recommender techniques that solve the prediction problem in different ways. See Table 2.5 for an overview of the pros and cons of content, memory and model-based systems.

As mentioned in 2.3.3, the different techniques can also be used together in a *hybrid* recommender system. This way the disadvantages of each approach could be avoided by the advantages of the other. For example: when combined with a neighborhood recommender, a content-based recommender

could recommend new items, while the neighborhood recommender could recommend items for which enough rating data is available. Or a dimensionality reduction technique can preprocess the data to avoid the scalability problems of a neighborhood system.

While hybrid systems provide better predictions, for our goal of improving the prediction accuracy by adding contextual information, we chose to develop on the matrix factorization technique. First of all, this method gave good results on its own in previous research and is very scalable. It does not require explicitly modeling the data either, so we don't need to have in-depth domain knowledge and it allows for generalizing the 2-dimensional utility matrix into a higher order tensor to make it context aware.

3 Matrix factorization

In this chapter we'll explore the underlying techniques that form the foundation of a lot of matrix factorization recommender systems: *principal component analysis* and *singular value decomposition*. After getting a solid understanding of these systems, we'll then discuss the implementation for matrix factorization using *explicit* and *implicit* data. The alternating least squares algorithm that is used for the implicit feedback system, serves as a basis for the work in Chapter 5, where we'll incorporate the contextual data.

3.1 Foundation

3.1.1 Principal component analysis

The data retrieved from experiments or other sources is often so much, that it becomes hard to see relations and discover what is important directly. It can be noisy and unclear, while the hidden relations can actually be very simple. To make sense of the data we need to convert it to a more structured and easier to understand format. A way to tackle this problem is to find the components that describe the data best, i.e. finding the variables in the data with the highest variance and removing those that are redundant. This is exactly what principal component analysis (PCA) tries to do.

Say we have collected the ratings of n users for two items α and β, then we can create two sets a and b that contain those n measurements for each item. The **variance** for either individual set is defined as:

$$\sigma_a^2 = \frac{1}{n}\sum_i (a_i - \mu_a)^2, \qquad \sigma_b^2 = \frac{1}{n}\sum_i (b_i - \mu_b)^2$$

Where μ_a and μ_b are the means of both sets. So if we assume a and b are sets with zero mean ($\mu_a = \mu_b = 0$) we get:

$$\sigma_a^2 = \frac{1}{n}\sum_i a_i^2, \qquad \sigma_b^2 = \frac{1}{n}\sum_i b_i^2$$

A high variance indicates that the users do not agree on the rating for the item, while a low variance means that the users rate the item largely the same. This means we can reduce the information for low variance items (a lot of rating information is redundant, since it is the same), while the data for high variance items is important data which we should keep.

The ***co*variance**, the degree of linear relationship between two variables (in this case between two items), is defined as follows:

$$cov(a, b) = \frac{1}{n} \sum_i a_i b_i$$

A high positive or negative value indicates that a and b are highly correlated and therefore contain redundant information (a lot of ratings are the same), while a value close to zero means that a and b show an independent relation (a lot of ratings differ).

So far we've used two items α and β. In most cases however, there are more than two variables, we can have *thousands* or even *millions* of items which are rated. To generalize to more variables let's change the measurement sets into vectors:

$$\begin{aligned} \mathbf{x}_1 &= [a_1, a_2, \ldots, a_n] \\ \mathbf{x}_2 &= [b_1, b_2, \ldots, b_n] \\ &\vdots \\ \mathbf{x}_m &= [\ldots] \end{aligned}$$

and merge those into a $m \times n$ matrix $\mathbf{X}$:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_m \end{bmatrix}$$

Intuitively this means that each row in $\mathbf{X}$ contains the ratings for a single item (by n different users) and each column in $\mathbf{X}$ contains the ratings from a single user (for m different items). Note that this can be seen as a utility matrix.

Now the variance and covariance for different items can be computed as follows in vector format:

$$\sigma^2_{\mathbf{x}_i} = \frac{1}{n}\mathbf{x}_i\mathbf{x}_i^T$$
$$cov(\mathbf{x}_i, \mathbf{x}_j) = \frac{1}{n}\mathbf{x}_i\mathbf{x}_j^T$$

It is easy to combine both into a so-called *covariance matrix* for $\mathbf{X}$:

$$cov(\mathbf{X}) = \frac{1}{n}\mathbf{X}\mathbf{X}^T$$

Here the diagonal terms of $cov(\mathbf{X})$ contain the *variance* of items, while the off-diagonal terms contain the *covariance* between two items (note that on the diagonal $cov(a, a) = \sigma_a^2$). This means that the highest values on the diagonal are most important (high variance), while the highest off-diagonal magnitudes depict high redundancy between item pairs and are therefore least important.

Ideally we would like the covariance matrix to only have (positive) values on the diagonal, while the rest is zero, i.e. we want $cov(\mathbf{X})$ to be a diagonal matrix. The question now becomes: how to modify $\mathbf{X}$, such that $cov(\mathbf{X})$ is diagonal?

To get there we first need to see that any symmetric matrix $\mathbf{S}$ can be diagonalized by an orthogonal matrix of its eigenvectors (see Theorem 4 in [55]).

$$\mathbf{S} = \mathbf{E}\mathbf{D}\mathbf{E}^T$$

Where each *column* of orthogonal matrix $\mathbf{E}$ is an eigenvector of $\mathbf{S}$ and $\mathbf{D}$ is a diagonal matrix (containing the eigenvalues of $\mathbf{S}$). Since the covariance matrix of $\mathbf{X}$ is symmetric, we can also diagonalize that by an orthogonal matrix of its eigenvectors:

$$cov(\mathbf{X}) = \mathbf{E}\mathbf{D}\mathbf{E}^T$$

Now, if we construct a matrix $\mathbf{P} = \mathbf{E}^T$ (thus: $\mathbf{P}$ contains the eigenvectors of $cov(\mathbf{X})$ as *rows*) and multiply $\mathbf{X}$ by $\mathbf{P}$:

$$\mathbf{Y} = \mathbf{P}\mathbf{X}$$

we can see that the covariance matrix of the result, $cov(\mathbf{Y})$, is diagonal. [55]

$$
\begin{aligned}
cov(\mathbf{Y}) &= \frac{1}{n}\mathbf{Y}\mathbf{Y}^T \\
cov(\mathbf{Y}) &= \frac{1}{n}\mathbf{P}\mathbf{X}(\mathbf{P}\mathbf{X})^T \\
cov(\mathbf{Y}) &= \frac{1}{n}\mathbf{P}\mathbf{X}\mathbf{X}^T\mathbf{P}^T \\
cov(\mathbf{Y}) &= \mathbf{P}(\frac{1}{n}\mathbf{X}\mathbf{X}^T)\mathbf{P}^T \\
cov(\mathbf{Y}) &= \mathbf{P}(cov(\mathbf{X}))\mathbf{P}^T \\
cov(\mathbf{Y}) &= \mathbf{P}(\mathbf{E}\mathbf{D}\mathbf{E}^T)\mathbf{P}^T \\
cov(\mathbf{Y}) &= \mathbf{P}(\mathbf{P}^T\mathbf{D}\mathbf{P})\mathbf{P}^T \\
cov(\mathbf{Y}) &= (\mathbf{P}\mathbf{P}^T)\mathbf{D}(\mathbf{P}\mathbf{P}^T) \\
cov(\mathbf{Y}) &= (\mathbf{P}\mathbf{P}^{-1})\mathbf{D}(\mathbf{P}\mathbf{P}^{-1}) \\
cov(\mathbf{Y}) &= \mathbf{D}
\end{aligned}
$$

(Since $\mathbf{P}$ is orthogonal, by definition $\mathbf{P}^T = \mathbf{P}^{-1}$)

Recall that the goal was to find the components which describe the data best, i.e. the components that have the highest variance and remove redundancy. Since multiplying $\mathbf{X}$ by $\mathbf{P}$, the eigenvectors of $cov(\mathbf{X})$, gives us a diagonal covariance matrix, we can conclude that the principal components of $\mathbf{X}$ therefore are the eigenvectors of $cov(\mathbf{X})$ (i.e. the eigenvectors of $\mathbf{X}\mathbf{X}^T$).

In Figure 3.1 the principal components for the ratings of two items i_1 and i_2 are shown. As we can see, the first feature F_1 presents the largest variance and is therefore the most significant component. The second feature describes the next most variance, and so on for the other eigenvectors (note that Figure 3.1 only exemplifies two dimensions; for more items there are consequentially more eigenvectors).

To reduce the dimensionality of the utility matrix, we can simply calculate the eigenvectors and eigenvalues of its covariance matrix and throw away the eigenvectors corresponding to the smallest eigenvalues [48]. These smallest components contribute the least to the data. We could for example remove F_2 from the data in Figure 3.1, since there is much more variance in the F_1 direction. All rating points will subsequently be lying on F_1, which results in a loss of information, but the loss is aimed to be minimized. It is often the case that the first k principle components correspond to a large portion of the variances, while the remaining principle components contribute much less [55].

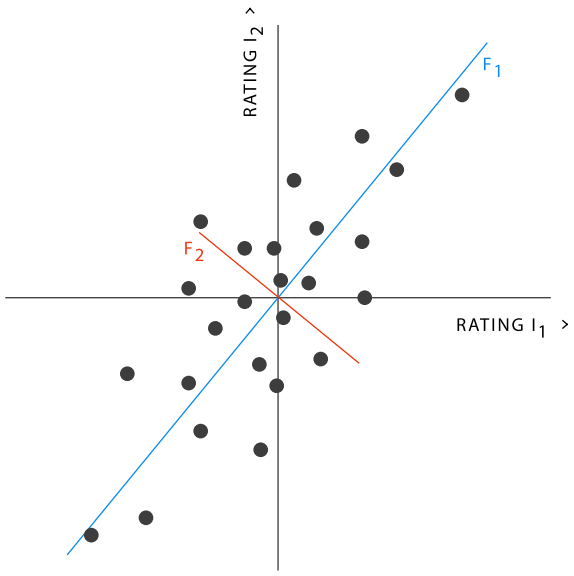

Figure 3.1: Principal components analysis finds a new orthonormal basis that maximizes the variance of the measurements along its axes. Here the mean ratings for two items i_1 and i_2 and two features that explain their variance are depicted.

One of the assumptions PCA makes is that the data is linearly related to the variables. Furthermore, it expects the principal components to be orthogonal. Of course this is often not the case and in those situations PCA may give poor results [2].

3.1.2 Singular value decomposition

As said before, singular value decomposition (SVD) is closely related to the above described PCA. Recall that we previously mentioned that any symmetric matrix $\mathbf{S}$ can be decomposed into $\mathbf{EDE}^T$. SVD loses the symmetric constraint and says that any real matrix $\mathbf{X}$ can be decomposed into:

$$\mathbf{X} = \mathbf{U\Sigma V}^T$$

Where $\mathbf{U}$ is a orthogonal matrix containing the *left-singular values*, $\mathbf{V}$ is an orthogonal matrix containing the *right-singular values* and $\mathbf{\Sigma}$ is a diagonal matrix with the so-called *singular values*. Note that this is an exact decomposition: the original matrix $\mathbf{X}$ can be fully reconstructed given $\mathbf{U}$, $\mathbf{\Sigma}$ and $\mathbf{V}$.

To demonstrate the relation with PCA, we'll express the covariance matrix of $\mathbf{X}$ in terms of the SVD:

$$\begin{aligned}
\mathbf{X} &= \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \\
\mathbf{X}\mathbf{X}^T &= (\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T)(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T)^T \\
\mathbf{X}\mathbf{X}^T &= (\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T)(\mathbf{V}\mathbf{\Sigma}\mathbf{U}^T) \\
\mathbf{X}\mathbf{X}^T &= \mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^T
\end{aligned}$$

Recall that $cov(\mathbf{X}) = \mathbf{X}\mathbf{X}^T = \mathbf{E}\mathbf{D}\mathbf{E}^T$ (normalization term $1/n$ is omitted for readability), thus we can see that $\mathbf{U}$ actually contains the eigenvectors of $\mathbf{X}\mathbf{X}^T$ and $\mathbf{\Sigma}$ is a diagonal matrix consisting of the square roots of the eigenvalues of $\mathbf{X}\mathbf{X}^T$. If we do the same for $\mathbf{X}^T\mathbf{X}$ (which calculates the variance and covariance between *columns* of $\mathbf{X}$), we'll see that $\mathbf{V}$ contains the eigenvectors of $\mathbf{X}^T\mathbf{X}$ and $\mathbf{\Sigma}$ contains the eigenvalues of $\mathbf{X}^T\mathbf{X}$ as well (these are the same as the eigenvalues of $\mathbf{X}\mathbf{X}^T$). Compare $\mathbf{U}$ and $\mathbf{V}$ to the direction of the principal components and the values in $\mathbf{\Sigma}$ to their length, i.e. the variance of the principal components. [64]

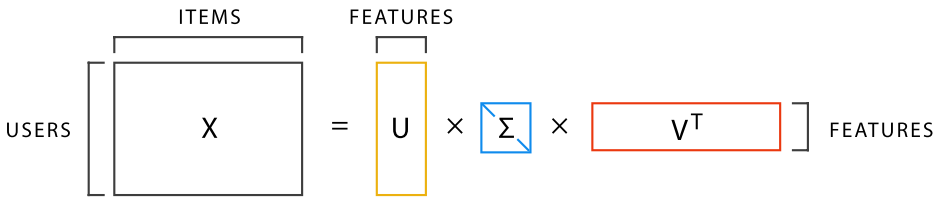

Figure 3.2: A singular value decomposition for utility matrix $\mathbf{X}$. Diagonal matrix $\mathbf{\Sigma}$ contains the eigenvalues of the covariance matrix, these values depict the strength of their corresponding features.

Intuitively, the left and right-singular values represent a mapping to features. If we let $\mathbf{X}$ be a *users* $\times$ *items* utility matrix, then $\mathbf{U}$ maps users to features, while $\mathbf{V}$ maps items to features. For example: when an item has a strong (positive) connection to a certain feature, it will get a high value for that feature. Likewise, the user features show how much a user prefers a corresponding item feature. Finally, the values in the central $\mathbf{\Sigma}$ matrix represent the strength of the features. If there is a lot of evidence in the observed data for a certain feature (i.e. there are many ratings for movies in the romance genre) than the value for this feature will be high. [48]

The values in $\mathbf{\Sigma}$ are always positive and are sorted from highest to lowest. Just like in PCA, the highest values depict the most important features to $\mathbf{X}$ and we can discard the less important components to reduce the dimensionality.

The Eckart-Young theorem [20] says that if we take the largest k singular values in $\mathbf{\Sigma}$ and compute the approximated resulting matrix $\hat{\mathbf{X}}$, then $\hat{\mathbf{X}}$ is the best rank-k (defined in next sentences) approximation for $\mathbf{X}$. Here, best approximation means that the Frobenius norm of the difference between $\hat{\mathbf{X}}$ and the original matrix $\mathbf{X}$ is minimized (i.e. $\min_{\hat{x}_{ij}} \|\mathbf{X} - \hat{\mathbf{X}}\| = \min_{\hat{x}_{ij}} \sqrt{\sum_{ij}(x_{ij} - \hat{x}_{ij})^2}$). The *rank* of a matrix denotes the number of independent rows (or equivalently: independent columns) it contains, i.e. the maximum number of rows which cannot be constructed by a linear combination of other rows [48].

Let's take for example Figure 3.3, which shows the singular value decomposition of our movie ratings utility matrix from Table 2.1. Since the utility matrix is rank-5, there are 5 singular values in the decomposition. We can see, however, that the smallest singular value (0.41) is a lot smaller than the others. That means that the corresponding feature is not that important for the overall utility matrix. If we remove that value (and the corresponding columns of $\mathbf{U}$ and $\mathbf{V}$), the SVD will give us a rank-4 approximation of $\mathbf{X}$, which will be the best rank-4 approximation possible (See Figure 3.4).

We can see that the result is very close to $\mathbf{X}$ (note that the result would be even closer if we used a higher precision). Although the reduction of dimensionality is not very large here - in fact the storage size of the SVD in the example is larger than the storage size of the utility matrix itself - for bigger real-world databases the reduction will be more obvious. While the number of users and items will be huge, the number of features can more or less stay the same (intuitively: the amount of movie genres does not grow beyond a certain point if more movies are added). For example if we have a database with 1000 items and 1000 users (10^6 values) and take 20 features, the SVD will store just $20(1+1000+1000) = 4 \cdot 10^4$ values – a 96% reduction!

Let us discuss how we can recommend items from the SVD. Both items and users can be represented in the feature space by multiplying their rating vector with respectively $\mathbf{U}$ and $\mathbf{V}$. Consider for example Dave, with rating vector $\mathbf{d} = \begin{bmatrix}1 & 0 & 4 & 0 & 5\end{bmatrix}$. To represent Dave in the feature space we'll multiply $\mathbf{d}$ by $\mathbf{V}$ (mapping item ratings to feature space) and $\mathbf{\Sigma}$ (adjusting the weights in the feature space) from Figure 3.4.

$$\boldsymbol{\delta} = \mathbf{dV\Sigma} = \begin{bmatrix}-31.80 & -31.68 & 22.37 & -0.14\end{bmatrix}$$

We can see Dave dislikes movies with the first two features, while he

$$
\underset{\mathbf{X}}{\begin{bmatrix} 0 & 1 & 0 & 5 & 4 \\ 0 & 5 & 0 & 1 & 2 \\ 5 & 4 & 0 & 1 & 0 \\ 1 & 0 & 4 & 0 & 5 \\ 5 & 5 & 0 & 2 & 0 \\ 0 & 2 & 0 & 4 & 3 \end{bmatrix}} = \underset{\mathbf{U}}{\begin{bmatrix} -0.39 & -0.51 & -0.38 & 0.34 & 0.33 \\ -0.39 & 0.00 & -0.11 & -0.89 & 0.16 \\ -0.46 & 0.43 & 0.19 & 0.23 & 0.62 \\ -0.23 & -0.50 & 0.83 & -0.02 & -0.07 \\ -0.55 & 0.44 & 0.06 & 0.17 & -0.59 \\ -0.37 & -0.34 & -0.34 & 0.04 & -0.36 \end{bmatrix}}
$$

$$
\underset{\mathbf{\Sigma}}{\begin{bmatrix} 11.69 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 7.98 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 5.19 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 3.42 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.41 \end{bmatrix}}
$$

$$
\underset{\mathbf{V}^T}{\begin{bmatrix} -0.45 & -0.65 & -0.08 & -0.46 & -0.39 \\ 0.48 & 0.34 & -0.25 & -0.32 & -0.69 \\ 0.40 & -0.11 & 0.64 & -0.59 & 0.27 \\ 0.59 & -0.66 & -0.02 & 0.45 & -0.11 \\ 0.22 & -0.12 & -0.72 & -0.37 & 0.53 \end{bmatrix}}
$$

Figure 3.3: Singular value decomposition for the utility matrix. Note that for readability the values are rounded, with infinite precision this rank-5 decomposition will be exact.

$$
\underset{\mathbf{U}}{\begin{bmatrix} -0.39 & -0.51 & -0.38 & 0.34 \\ -0.39 & 0.00 & -0.11 & -0.89 \\ -0.46 & 0.43 & 0.19 & 0.23 \\ -0.23 & -0.50 & 0.83 & -0.02 \\ -0.55 & 0.44 & 0.06 & 0.17 \\ -0.37 & -0.34 & -0.34 & 0.04 \end{bmatrix}} \underset{\mathbf{\Sigma}}{\begin{bmatrix} 11.69 & 0.00 & 0.00 & 0.00 \\ 0.00 & 7.98 & 0.00 & 0.00 \\ 0.00 & 0.00 & 5.19 & 0.00 \\ 0.00 & 0.00 & 0.00 & 3.42 \end{bmatrix}}
$$

$$
\underset{\mathbf{V}^T}{\begin{bmatrix} -0.45 & -0.65 & -0.08 & -0.46 & -0.39 \\ 0.48 & 0.34 & -0.25 & -0.32 & -0.69 \\ 0.40 & -0.11 & 0.64 & -0.59 & 0.27 \\ 0.59 & -0.66 & -0.02 & 0.45 & -0.11 \end{bmatrix}} =
$$

$$
\underset{\hat{\mathbf{X}}}{\begin{bmatrix} 0.00 & 1.03 & 0.10 & 5.09 & 3.93 \\ 0.03 & 5.04 & 0.06 & 1.06 & 1.96 \\ 4.93 & 4.03 & 0.19 & 1.15 & -0.09 \\ 0.98 & -0.04 & 3.97 & -0.06 & 4.97 \\ 5.05 & 4.96 & -0.18 & 1.91 & 0.10 \\ 0.02 & 1.99 & -0.11 & 3.96 & 3.07 \end{bmatrix}}
$$

Figure 3.4: The lowest singular value is removed and the approximated utility matrix $\hat{\mathbf{X}}$ is calculated, which is close to $\mathbf{X}$ in Figure 3.3.

prefers movies connected to the third feature. Note again that we don't know exactly what the features represent, although we can guess the first two features corresponds to fantasy and adventure, since Dave rated The Lord of the Rings poorly.

There are a couple of things we can do with this result. We can find similar users, by comparing the feature space vectors of others with the one from Dave. That is, we can use the k-Nearest Neighbors algorithm on the feature space instead of on the raw ratings. Another thing we can do is recommend movies with a similar feature space vector. For example the movie Titanic has feature space vector $\begin{bmatrix}-62.89 & -20.75 & -15.88 & 5.27\end{bmatrix}$, which has a cosine distance of 0.328231 with Dave's, while Harry Potter ($\begin{bmatrix}-89.66 & 21.79 & -2.85 & -7.73\end{bmatrix}$) has cosine distance 0.548383. That means the features of Titanic are closer to Dave's feature preferences than those of Harry Potter are. This is expected if we consider that LOTR is related to Harry Potter and Titanic to Forrest Gump (Dave rated the latter high and LOTR low). Notice how we can directly compare movies and users this way, using just ratings!

The last method of predicting ratings is transforming the feature space back to rating space, i.e. multiplying $\boldsymbol{\delta}$ by $\mathbf{V}^T$.

$$\boldsymbol{\delta}\mathbf{V}^T = \begin{bmatrix}7.97 & 7.53 & 24.78 & 11.50 & 40.32\end{bmatrix}$$

These are the predicted ratings for Dave according to the SVD. It results in recommending Titanic (rating score 11.50) as well, since it is the highest not-rated movie.

So far we've treated unknown values as zero, but that does give a negative bias to unrated movie-item pairs. Unfortunately the SVD is not well defined for sparse matrices like the utility matrix. A solution is to use the average rating for items, normalized by substracting the user average in place of the unknown values [2]. However, computing the full SVD is rather computationally expensive (especially for the very large datasets recommender systems are dealing with) and also not very efficient: we only need the largest singular values. Methods exist to directly get a truncated SVD and only compute k singular values, but for an $n \times m$ matrix that still takes $O(mn\log(k))$. [25]

3.2 Explicit feedback

For collaborative filtering it turns out we don't actually need to find an exact SVD. Again, the SVD isn't well defined for sparse matrices and since we want only the top singular values it is overkill to compute a full singular value decomposition. This, together with the realization that the Frobenius

norm of the difference between the original and the approximated matrix is minimized, led Funk [21] to a different technique of factorizing.

This method decomposes $\mathbf{R}$ into just the two feature matrices $\mathbf{X} \in \mathbb{R}^{|U| \times f}$ and $\mathbf{Y} \in \mathbb{R}^{|I| \times f}$. The orthogonality constraint as seen in the SVD is dropped as it is not necessary for CF either.

$$\hat{\mathbf{R}} = \mathbf{X}\mathbf{Y}^T$$

The singular values in $\mathbf{\Sigma}$ (which are essentially scaling factors) are in this case merged into $\mathbf{X}$ and $\mathbf{Y}^T$.

Conceptually it works the same as the SVD, with the two feature matrices mapping items and users into a feature space. Choosing the amount of features (the number of columns in $\mathbf{X}$ and $\mathbf{Y}$) is analogous to selecting the top k singular values. Using many features will describe the data well, but will also model the noise, while less features only take into account the most influencing factors of the data.

However, finding the feature matrices is a little different. There is no need for calculating eigenvectors and eigenvalues: we can simply create them by minimizing the Frobenius norm of the difference between $\mathbf{R}$ and $\mathbf{X}\mathbf{Y}^T$, i.e. minimizing the root mean squared error (RMSE) of all predictions. This can be computed by for example applying a stochastic gradient descent or alternating least squares algorithm. Roweis shows (as cited in [6]) that the iterative process results in a global minimum and therefore $\mathbf{X}\mathbf{Y}^T$ is the closest rank-f approximation of the utility matrix.

Predicting a rating is done by multiplying a user feature vector $\mathbf{x}_u$ by a item feature vector $\mathbf{y}_i$, i.e. $\hat{r}_{ui} = \mathbf{x}_u\mathbf{y}_i$. Then we can calculate the error of each prediction with $(r_{ui} - \hat{r}_{ui})^2$. Note that minimizing the RMSE is in this case equal to minimizing the sum of squared errors. A benefit of this model is that it only considers known ratings while learning (a lot less values to consider than in the full matrix!). It is defined as follows:

$$\arg\min_{x_*, y_*} \sum_{r_{u,i}\ known} (r_{ui} - \mathbf{x}_u^T\mathbf{y}_i)^2 \tag{3.1}$$

To find the solution the gradient descent algorithm computes the error and tries to decrease it in the next step by adjusting the prediction value in the direction of the error gradient. That is, the derivative of the error (e_{ui}) for a single prediction value is calculated with respect to the corresponding values in the user (x_{uk}) and item (y_{ik}) feature matrices. Then in the next iteration both (x_{uk}) and (y_{ik}) are updated according to the derivative. Formally:

$$\frac{1}{2}\frac{\partial e_{ui}}{\partial x_{uk}} = (r_{ui} - \mathbf{x}_u^T\mathbf{y}_i) \cdot -y_{ik}$$

Note that $\mathbf{x}_u \mathbf{y}_i^T = \sum_{k=1}^{K} x_{uk} y_{ik}$. Similarly we'll find:

$$\frac{1}{2}\frac{\partial e_{ui}}{\partial y_{ik}} = (r_{ui} - \mathbf{x}_u^T \mathbf{y}_i) \cdot -x_{uk}$$

Now x_{uk} and y_{ik} are updated as follows (in opposite direction of the gradient):

$$x'_{uk} = x_{uk} + \gamma \cdot (r_{ui} - \mathbf{x}_u^T \mathbf{y}_i) \cdot y_{ik}$$
$$y'_{ik} = y_{ik} + \gamma \cdot (r_{ui} - \mathbf{x}_u^T \mathbf{y}_i) \cdot x_{uk}$$

Here, γ is the learning rate. The algorithm keeps iterating and lowering the error for all predictions (and thus the RMSE) until conversion.

By learning the model with Equation 3.1 we can get a result that perfectly resembles the utility matrix, but that means there is a high risk of overfitting the training data. In that case, the model learns the noise of the data instead of the underlying structure. Therefore a regularization term is added, which penalizes the size of the feature matrices.

$$\arg\min_{x_*, y_*} \sum_{r_{u,i}\ known} (r_{ui} - \mathbf{x}_u^T \mathbf{y}_i)^2 + \lambda(\|\mathbf{x}_u\|^2 + \|\mathbf{y}_i\|^2)$$

With parameter λ we can set the weight of the regularization.

Again we can compute the derivatives and update the feature values according to the gradient in each iteration [61, 21]:

$$\frac{1}{2}\frac{\partial e_{ui}}{\partial x_{uk}} = (r_{ui} - \mathbf{x}_u^T \mathbf{y}_i) \cdot -y_{ik} + \lambda \cdot x_{uk}$$
$$\frac{1}{2}\frac{\partial e_{ui}}{\partial y_{ik}} = (r_{ui} - \mathbf{x}_u^T \mathbf{y}_i) \cdot -x_{uk} + \lambda \cdot y_{ik}$$
$$x'_{uk} = x_{uk} + \gamma \cdot (r_{ui} - \mathbf{x}_u^T \mathbf{y}_i) \cdot y_{ik} - \lambda \cdot x_{uk}$$
$$y'_{ik} = y_{ik} + \gamma \cdot (r_{ui} - \mathbf{x}_u^T \mathbf{y}_i) \cdot x_{uk} - \lambda \cdot y_{ik}$$

In addition to the feature matrices, we can add the baseline estimates introduced in Equation 2.6 here as well. The parameters b_u and b_i can be learned together with the feature matrices [37, 46]:

$$\arg\min_{b_*, x_*, y_*} \sum_{r_{u,i}\ known} (r_{ui} - \mu - b_u - b_i - \mathbf{x}_u^T \mathbf{y}_i)^2 + \lambda(b_u^2 + b_i^2 + \|\mathbf{x}_u\|^2 + \|\mathbf{y}_i\|^2) \quad (3.2)$$

Finally, after learning the model we can recommend items. The predictions for a user u are obtained by multiplying the user features ($\mathbf{x}_u$) by the feature vector of each item i ($\mathbf{y}_i$) and adding the baseline estimates.

$$\hat{r}_{ui} = \mu + b_u + b_i + \mathbf{x}_u^T \mathbf{y}_i$$

The predictions with the highest scores are recommended to the user.

Several adaptations of this *SVD-like* method have been introduced, such as NSVD1 and NSVD2 [46], which lower the number of parameters by representing users as the items they prefer and SVD++ [34] which also considers implicit feedback. Maximum margin matrix factorization makes sure the predictions are discreet values within $\{1, \ldots, r\}$ and Non-negative matrix factorization makes sure all values in $\mathbf{X}$ and $\mathbf{Y}$ are positive [65]. These methods predict the *explicit* ratings from users, the next section will focus on recommending with *implicit* data.

3.3 Implicit feedback

In [28] an adaptation is proposed on the explicit feedback model to make it possible to use implicit feedback. Recall that implicit feedback means that we collect rating data from user behavior. It doesn't require user interaction and is generally more available than the higher quality explicit ratings [42].

Unfortunately the drawback of this type of feedback is that we need to take into account the unknown ratings. If a song has not been listened to by a specific user, it can mean two things:

1. The user does *not like* the song and does not listen to it
2. The user does *not know* the song and has not listened to it yet

In the explicit ratings model an unknown rating means just that the item is not known to the user. If the user does not like the item, he would rate it explicitly negative. Where we could skip learning unknown ratings in the explicit model (Equation 3.2), now we cannot dismiss them; the 'unknown' values in the utility matrix are now actually zero. This means that the stochastic gradient descent algorithm is not efficient anymore and we need to find another way to compute feature matrices $\mathbf{X}$ and $\mathbf{Y}$. The following section describes how the implicit model works.

Let r_{ui} be the implicit rating for item i by user u:

$$r_{ui} \in \mathbb{N}$$

These implicit ratings are split into two components: a preference and a confidence. The binary preference of user u to item i (the user likes the item or not) is denoted by p_{ui}.

$$p_{ui} = \begin{cases} 1, & r_{ui} > 0 \\ 0, & r_{ui} = 0 \end{cases}$$

Then c_{ui} indicates the confidence of observing p_{ui}, i.e. a high number of implicit feedback should mean that the system is confident the user likes the item ($p_{ui} = 1$). On the other hand, for an item that has not been used at all there is a low confidence that the user dislikes it ($p_{ui} = 0$) (it might be an unknown item to the user).

$$c_{ui} = 1 + \alpha r_{ui}$$

Here constant α regulates how much larger observed preferences should contribute to the confidence relative to lower observed preferences. This gives a minimal confidence of 1, while more evidence of preference increases the confidence in $p_{ui} = 1$ linearly.

The predicted preference for user u to item i is calculated by the dot product of the user features $\mathbf{x}_u$ and item features $\mathbf{y}_i$.

$$\hat{r}_{ui} = \mathbf{x}_u^T \mathbf{y}_i$$

We want $\mathbf{x}_u^T \mathbf{y}_i$ to be as close to p_{ui} as possible for each u, i, while those relations with a higher confidence are more important and have a higher weight. This brings us to the cost function, which is defined as follows [28]:

$$S = \sum_{u,i} c_{ui}(p_{ui} - \mathbf{x}_u^T \mathbf{y}_i)^2 + \lambda(\sum_u \|\mathbf{x}_u\|^2 + \sum_i \|\mathbf{y}_i\|^2) \tag{3.3}$$

Included is again the regularization term, which makes sure we don't overfit the data. It penalizes the length of the feature vectors and is adjustable by parameter λ.

Instead of gradient descent, the alternating least squares (ALS) algorithm is used to minimize S. ALS exploits the fact that if we fix either $\mathbf{x}_u$ or $\mathbf{y}_i$ in 3.3, the minimization of the cost function becomes a least squares problem. In other words: S becomes quadratic, for which we can easily find a global minimum. Therefore, if we assume $\mathbf{y}_i$ is fixed, to find $\mathbf{x}_u$ such that S is minimized we need to set the derivative of S with respect to $\mathbf{x}_u$ equal to zero. We'll first differentiate S with respect to x_{ku} and work from there to get an expression for $\mathbf{x}_u$ (the derivation is ours, resulting in Equation 3.4 from Hu et al. [28]):

$$\frac{1}{2}\frac{\partial S}{\partial x_{ku}} = 0 \qquad \forall u, k$$

$$\frac{1}{2} \cdot 2 \sum_i c_{ui}(p_{ui} - \mathbf{x}_u^T \mathbf{y}_i) \cdot -y_{ki} + 2\lambda \cdot x_{ku} = 0 \qquad \forall u, k$$

$$\sum_i c_{ui}(p_{ui} - \mathbf{x}_u^T \mathbf{y}_i) \cdot -y_{ki} + \lambda \cdot x_{ku} = 0 \qquad \forall u, k$$

$$\sum_i y_{ki} \cdot c_{ui}(\mathbf{x}_u^T \mathbf{y}_i - p_{ui}) + \lambda \cdot x_{ku} = 0 \qquad \forall u, k$$

$$\sum_i y_{ki} \cdot c_{ui} \cdot \mathbf{y}_i^T \mathbf{x}_u + \lambda \cdot x_{ku} = \sum_i y_{ki} \cdot c_{ui} \cdot p_{ui} \qquad \forall u, k$$

To write this in matrix form and rearrange it to solve for $\mathbf{x}_u$, we'll first have to construct $n \times n$ matrix $\mathbf{C}^u$, which contains the confidence values c_{ui} on the diagonal, i.e. $C^u_{ii} = c_{ui}$. We also need the vector $\mathbf{p}_u$ containing all n preference values for user u.

$$\begin{aligned}
\mathbf{Y}^T\mathbf{C}^u\mathbf{Y}\mathbf{x}_u + \lambda\mathbf{x}_u &= \mathbf{Y}^T\mathbf{C}^u\mathbf{p}_u && \forall u \\
(\mathbf{Y}^T\mathbf{C}^u\mathbf{Y} + \lambda\mathbf{I})\mathbf{x}_u &= \mathbf{Y}^T\mathbf{C}^u\mathbf{p}_u && \forall u \\
\mathbf{x}_u &= (\mathbf{Y}^T\mathbf{C}^u\mathbf{Y} + \lambda\mathbf{I})^{-1}\mathbf{Y}^T\mathbf{C}^u\mathbf{p}_u && \forall u \qquad (3.4)
\end{aligned}$$

The features for each user can now be computed. A naive calculation of $\mathbf{x}_u$ will take:

- $O(f^2 n)$ for the calculation of $(\mathbf{Y}^T\mathbf{C}^u\mathbf{Y} + \lambda\mathbf{I})$
- $O(f^3)$ for the matrix inversion $(\mathbf{Y}^T\mathbf{C}^u\mathbf{Y} + \lambda\mathbf{I})^{-1}$
- $O(f n_u)$ for the computation of $\mathbf{Y}^T\mathbf{C}^u\mathbf{p}_u$

Where n_u equals the number of observations for user u (i.e. the number of items for which $r_{ui} > 0$) and f equals the number of features. This totals to $O(f^2 n + f^3 + f n_u)$ for a single user, or $O(f^2 nm + f^3 m + fN)$ for m users (where N equals the total amount of observations).

Fortunately we can bring this down by noticing that $\mathbf{C}^u - \mathbf{I}$ contains just n_u non-zero entries (recall that $c_{ui} = 1 + \alpha r_{ui}$). Now we can rewrite Equation 3.4 to:

$$\mathbf{x}_u = (\mathbf{Y}^T\mathbf{Y} + \mathbf{Y}^T(\mathbf{C}^u - \mathbf{I})\mathbf{Y} + \lambda\mathbf{I})^{-1}\mathbf{Y}^T\mathbf{C}^u\mathbf{p}_u \qquad \forall u \qquad (3.5)$$

Enabling us to precompute $\mathbf{Y}^T\mathbf{Y}$ (since it is not dependent of u) which takes $O(f^2n)$ time. The computation of $\mathbf{Y}^T(\mathbf{C}^u - \mathbf{I})\mathbf{Y}$ takes just $O(f^2n_u)$. Computing the full feature matrix $\mathbf{X}$ for all users then takes $O(f^2N + f^3m + f^2n)$.

After all user features are updated, they are in turn fixed in order to get the item features. We can use the same derivation for $\frac{1}{2}\frac{\delta S}{\delta y_{ki}}$ which gives:

$$\mathbf{y}_i = (\mathbf{X}^T\mathbf{C}^i\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{C}^i\mathbf{p}_i$$

This, by using the same technique as above for $\mathbf{X}$, is computable in $O(f^2N + f^3n + f^2m)$ for all n items.

Alternating the computation of $\mathbf{X}$ and $\mathbf{Y}$ until conversion (typically only a few iterations are needed until no improvement is noticeable) gives us a good approximation of the preference matrix $\mathbf{P}$ in reasonably fast running time.

4 Context-aware recommendations

So far we've only talked about users, items and their relation represented by ratings. We've assumed that the predictions are valid for each situation, given the preferences and properties of those users and items. There are many factors however, that could influence the preference of a user. A user might for instance prefer books about his hobbies in the weekend, while he reads work related topics on workdays.

These situational properties (time, weather, mood, location etc.) can change predictions significantly, e.g. someone looking for a vacation destination in the summer does probably not want to go to a skiing resort. Consumer decision making is influenced by context, according to behavioral research [1]. Therefore it is important to also consider the context when predicting. Furthermore, using context in recommender systems provides more trust in recommendations as concluded by Gorgoglione et al. [23]. This increased trusts leads in turn to customers willing to pay higher prices for products, which improves sales.

The notion of context has been studied in multiple disciplines and many gave a different definition [1]. For recommender systems we can think of context as the additional information that may be relevant for making a recommendation [4].

This chapter briefly describes different kinds of contextual information (4.1) and possible implementations for recommender systems (4.2). Then we propose a way to incorporate context into the implicit feedback matrix factorization method from Section 3.2.

4.1 Contextual information

Just like rating information, we can obtain context information *explicitly* or *implicitly*. For explicit information the user has to intentionally specify the

context. The user can for example specify with whom he watched a movie or the action of setting a phone on silent or audible can be explicit context information. This might not always be reliable information however, since a user might forget information or actions, especially when asked about a lot of contextual variables and the consumption of the item was some time ago [9].

On the other hand, implicit context information is gathered without extra user interaction. This of course perfectly fits to the implicit way of collecting rating information. We can for example use different kinds of sensors to measure weather (hot, rainy), user activity (running, sitting), location (work, home), company (family, friends) etc. This information can be recorded directly when the user consumes an item.

After the contextual information is collected, we have to decide how to represent the data. Some information can be continuous (e.g. temperature, time), while others are categorical (e.g. company, activities). The recommender system should be able to handle these different types of information, or we can convert to a single type. We can for instance categorize the temperature by specifying intervals which play an important role in influencing the rating data (hot: $t \geq 23°$, moderate: $11° < t < 23°$, cold: $t \leq 11°$). Of course the challenge is to find good partitions for the collected data.

It is easier however, to convert continuous data to categorical than the other way around. A recommender using categorical context data is therefore generally more suitable for different kinds context information.

4.1.1 Temporal information

One of the most useful and often easily accessible pieces of contextual information is time [12]. When user behavior is implicitly collected, the temporal data of the behavior can be recorded as well. For example when browsing websites the access times for the different webpages are stored along the website data with little extra effort.

With explicit rating data time is used a little differently. Here the temporal data of the action of rating an item can be stored, but that is different from the temporal data of the consumption of the item; a user might rate a movie weeks after he has seen it. It is of course possible to exploit this kind of data as well (e.g. the time between rating a movie and its release [47]), but it is important that it's a different kind. Another possibility is of course explicitly asking for time data when a user rates an item.

There is a couple of ways we can use the temporal data. We can use it in a more absolute way, where we assume that older ratings are less important then newer ratings (preferences drift over time), or we can use it in a more

relative way, where we find patterns in the ratings matching the periodic nature of time (i.e. ratings differ by season).

For the absolute (fresh-based) approach we can use a damped or sliding window model, to give newer ratings more weight than older ratings [57]. The damped window gradually decays the weight of ratings the older they are, whereas the window model is more abrupt and does not consider ratings once they fall outside of a chosen timeframe.

Since temporal information is continuous we can directly use it for the fresh-based approach, but for the periodic method we need to partition it into separate categories. It depends on the domain which partitions make sense, for example a vacation recommender may choose seasonal partitions (winter, spring, summer, autumn), while day of the week or time of the day makes more sense for a music recommender [12, 5].

We focus on time as contextual information, because it is easily obtainable and can be used continuous as well as categorical. Furthermore using time in recommender systems has previously improved the prediction quality of recommender systems [35, 5], although some results may be contradictory [12]. The dataset we use for evaluating contains temporal data for the implicit feedback (Section 6.1).

4.2 Using context

For context-aware recommenders the problem F of predicting ratings (Equation 2.1) is extended with a set of contextual dimensions C:

$$F : U \times I \times C \to R \tag{4.1}$$

The dimensions in C can be of a different type and can have different values.

Incorporating this context into recommender systems can be done in three ways [1]: contextual pre-filtering, contextual post-filtering and contextual modeling (see Figure 4.1). Each of these techniques can be combined with the memory or model-based recommenders as described earlier.

Contextual pre-filtering filters the rating data on a certain context before the recommender system calculates the predictions. This means the ratings are predicted using a constructed subset of the training data, which contains just the ratings relevant to the given context. For example if a user wants to find a vacation in the summer, the recommender system only considers ratings from other users for vacations in the summer (and possible spring), but not winter, in order to calculate predictions. Then the highest ranked predictions can be proposed to the user. Baltrunas and Amatriain [5] propose a pre-filtering technique which creates micro profiles for each user. Each

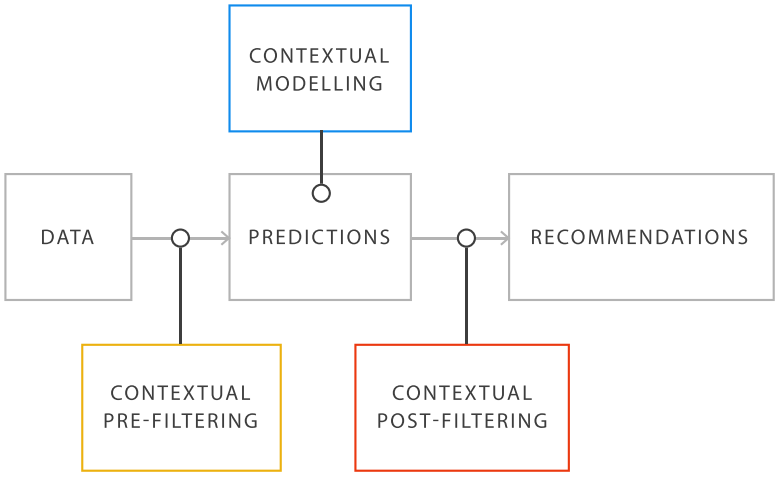

Figure 4.1: Three possible ways to incorporate contextual information in a recommender system.

micro profile represents the user for a different time context. Another implementation is by splitting items according to contexts, if they show significant differences between those context conditions [4]. That way a single item can be virtually split into multiple items for different contexts.

Post-filtering on the other hand uses the full rating data for predicting, but only recommends results that are relevant for the given context. In the previous vacation example, the system now predicts ratings for all vacations, using all rating data, but once the predictions are done only those vacations relevant to the summer (and possibly spring) are proposed. This can be done by either filtering out irrelevant recommendations, or by adjusting the ranks of the predictions based on their context relevance. Hariri et al. [26] use post-filtering to rerank neighborhood based predictions by computing a contextual score for each item, that represents the suitability of the item for the user's context.

The advantage of both pre-filtering and post-filtering is that they can be implemented by using traditional context-unaware recommenders. The additional filtering of the data can simply be added respectively before or after the prediction step of the system.

With multiple context dimensions it is important to not always filter the data based on the context exactly. The given contextual situation might after all be too specific to get enough results. For instance recommending a vacation destination in The Netherlands where the weather type is sunny. Furthermore, some context variables might not be significant and should be

replaced by a broader variable (e.g. recommend music at 5:53 PM).

Finally, contextual modeling uses context directly while predicting. This requires a different kind of recommender in which multidimensional data can be used to make predictions. For example the neighborhood approach can be extended to a multidimensional recommender by using a multidimensional similarity method. The simplest way to achieve this is by setting the distance to infinity if the context does not match, which basically results in an exact pre-filtering technique.

Another method is the temporal dynamics model (timeSVD++) [35], which is built upon SVD++ and incorporates two temporal effects: (1) an item's popularity changes over time, (2) users change their baseline ratings over time. The baseline estimate (Equation 2.6) is therefore updated with time-dependent biases:

$$b_{ui}(t) = \mu + b_u(t) + b_i(t) \tag{4.2}$$

While the user features are also made time-aware: $\mathbf{x}_u(t)$. This model improved the predictions on the Netflix Prize dataset compared to the context-unaware SVD++ model.

Since pre or post-filtering the data based on the context can lead to loss of information [30] and the model-based recommender approaches outperformed the memory-based ones, we'll focus on extending the matrix factorization technique outlined in Section 3.3 with context awareness using contextual modeling. In the next section we'll propose a linear blend of multiple 2D matrix factorization systems. Chapter 5 uses tensor factorization to learn the time context together with the user and item preferences.

4.2.1 Context-aware matrix factorization

A simple and naive adaptation of the 2 dimensional matrix factorization to the 3 dimensions – users, items, time – is to create two feature matrices per dimension, see Figure 4.2. We can compute the preferences between items-users, between users-time and between items-time individually the same way as in the 2D case. When we need to find the preference prediction for some user, item, time combination, we can blend the three results.

The user-time features however, don't hold extra useful information. Since recommendations will always be for a certain user in a certain context, the preferences for a context is intuitively not something we need. For example, discovering that a user likes to listen in the evenings, does not influence the predictions for that user. On the other hand, if a certain item is more popular on Fridays that does give us useful information, which we can

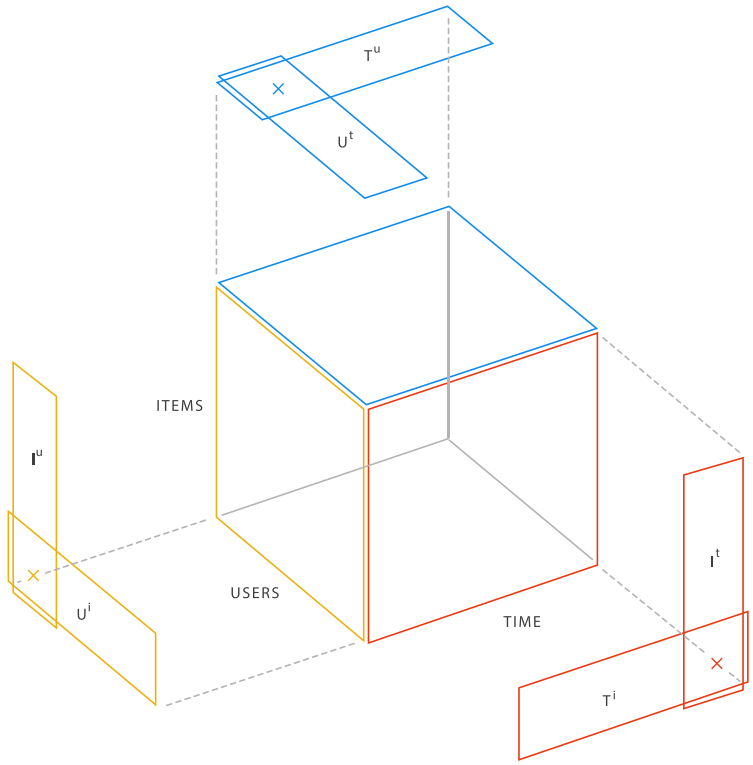

Figure 4.2: Two feature matrices per dimension

use for the predictions. That's why we don't learn the user-time features, but only blend the results from the user-item and item-time dimensions.

The predicted preference function becomes:

$$\hat{r}_{uit} = \alpha \cdot \hat{r}_{ui} + \beta \cdot \hat{r}_{it}$$
$$\hat{r}_{uit} = \alpha \cdot \mathbf{u}_u^T \mathbf{i}_i^u + \beta \cdot \mathbf{i}_u^{t\,T} \mathbf{t}_t$$

Where $\mathbf{U} \in \mathbb{R}^{|U| \times f}$ and $\mathbf{I}^u \in \mathbb{R}^{|I| \times f}$ are the feature matrices for the user-item interaction and $\mathbf{I}^t \in \mathbb{R}^{|I| \times f}$ and $\mathbf{T} \in \mathbb{R}^{|T| \times f}$ are the feature matrices for the item-time interaction. Parameters α and β are estimated after the user-item and item-time interactions both are learned separately according to Equation 3.4. When $\alpha = 1$ and $\beta = 0$ this model will be equal to the non-contextual matrix factorization.

A somewhat similar model, the Pairwise Interaction Tensor Factorization (PITF), has been proposed by Rendle and Schmidt-Thieme [50], which models user-tags and item-tags interactions. The PITF model is learned at once

using a Bayesion Personalized Ranking algorithm, which applies stochastic gradient descent on random samples.

Our naive approach calculates each dimensional preference individually, but ideally we would like to train all three dimensions at once, such that we can discover latent factors that span all dimensions. Chapter 5 will explore tensor decomposition techniques, which try to do just that.

5 Tensor decomposition

A tensor is a generalized name for a multi-dimensional array. The *order* of a tensor represents the number of dimensions (also known as *modes*). Thus, an Nth-order tensor consists of the product of N vector spaces, e.g. a simple array is a first-order tensor and a matrix is a second-order tensor. Tensors with higher dimensions (three or more) are called higher-order tensors.

Tensor decomposition is the generalization of matrix factorization to higher-order tensors. It allows for decomposition of N-dimensional tensors, such that any number of context variables can be added to the utility matrix. Similar to 2D matrix factorization, the decomposed tensor consists of the features of the different dimensions in the same latent factor space. That means that the contextual dimensions can be directly compared with the item and user dimensions and it is possible to discover latent features that span all dimensions. Since we focus on just the time variable as context, we'll use third-order tensors as example.

Related work such as Multiverse recommendations [30] and Bayesian probabilistic tensor factorization [67] show promising results compared to other context-aware systems and recommenders without context. With Multiverse recommendations Karatzoglou et al. report a 5%-30% improvement over non-contextual recommenders.

Multiple ways to decompose a tensor exist and many are closely related, see the comparison by Ceulemans and Kiers [14]. In this chapter we'll discuss two main methods of tensor decomposition: *Tucker Decomposition* (Section 5.1) and *Candecomp/Parafac* (Section 5.2). In 5.2.1 we'll show our implementation of Candecomp/Parafac for implicit feedback datasets.

5.1 Tucker Decomposition

The Tucker Decomposition is also known as Higher-Order SVD (HOSVD) and is – as that name indicates – a generalization of SVD to higher order tensors. It decomposes an Nth-order tensor into a single core tensor mul-

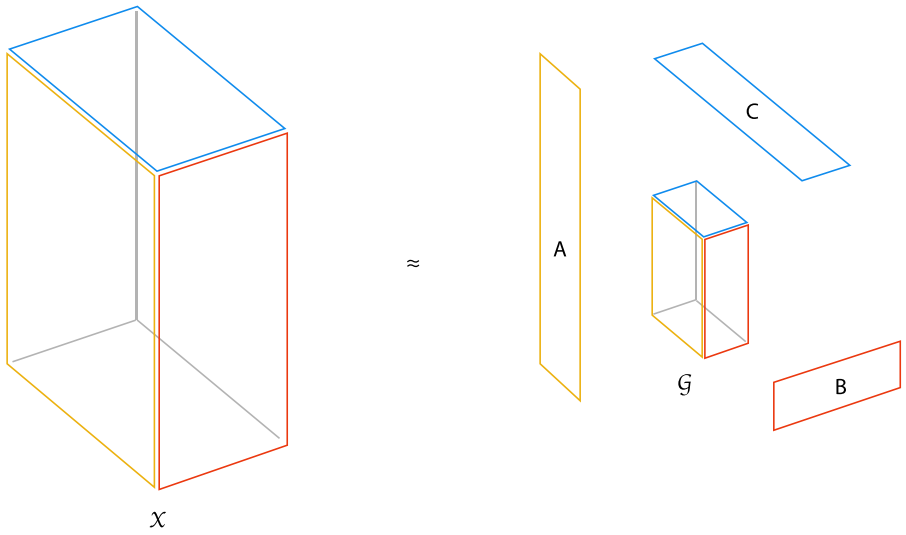

Figure 5.1: Tucker Decomposition of a third-order tensor $\mathcal{X}$

tiplied by N component matrices (i.e. a matrix for each dimension of the tensor). The core tensor acts as a scaling factor, which depicts the relationships between the component matrices, just like the singular values in SVD. The component matrices can be compared with the principal components (singular vectors) for each mode (dimension) of the matrix, see Figure 5.1.

For a utility tensor $\mathcal{R} \in \mathbb{R}^{I \times J \times K}$ the prediction of a rating $\hat{r}_{ijk}$ is calculated as follows:

$$\hat{r}_{ijk} = \sum_{p=1}^{P} \sum_{q=1}^{Q} \sum_{r=1}^{R} g_{pqr} a_{ip} b_{jq} c_{kr} \tag{5.1}$$

Here, $\mathcal{G} \in \mathbb{R}^{P \times Q \times R}$ is the core tensor and $\mathbf{A} \in \mathbb{R}^{I \times P}$, $\mathbf{B} \in \mathbb{R}^{J \times Q}$ and $\mathbf{C} \in \mathbb{R}^{K \times R}$ are the component matrices [32]. Although the component matrices are often orthogonal, for recommender systems this is not necessary [30].

Parameters P, Q and R represent the number of features for each dimension. These can be individually adjusted and give the Tucker decomposition full control over the number of features used for the users, items and context dimension. This is especially useful for large-scale databases, where the vast number of users or items can give storage problems.

An exact Tucker decomposition can be computed by setting P, Q and R to the corresponding n-rank of the tensor (denoted by $rank_n(\mathcal{R})$). The n-rank of a tensor is defined as the rank of its n-mode unfolding. Here, the n-mode unfolding is a flattened representation of the tensor in the form of a

matrix. For example, a third-order tensor $\mathcal{X} \in \mathbb{R}^{x \times y \times z}$ can be flattened to the matrices $\mathbf{X}_{(1)} \in \mathbb{R}^{x \times yz}$, $\mathbf{X}_{(2)} \in \mathbb{R}^{y \times xz}$ and $\mathbf{X}_{(3)} \in \mathbb{R}^{z \times xy}$, each containing all the elements of $\mathcal{X}$. Then $rank_n(\mathcal{X}) = rank(\mathbf{X}_{(n)})$ [16]. The n-rank is in other words, the rank of the tensor in one of its dimensions. For all dimensions Combined, a tensor is defined to be rank-$(R_1, R_2, ..., R_N)$, where $R_n = rank_n(\mathcal{X})$.

The component matrices are then computed one-by-one by taking the $rank_n(\mathcal{R})$ leading left singular vectors of $\mathbf{R}_{(n)}$. Once the component matrices are calculated, the core tensor $\mathcal{G}$ can be constructed as follows:

$$\mathcal{G} = \mathcal{R} \times_1 A \times_2 B \times_3 C \tag{5.2}$$

Where $\times_n$ is the n-mode product, element-wise defined as [32]:

$$(\mathcal{X} \times_n U)_{i_1 \ldots i_{n-1} j i_{n+1} \ldots i_N} = \sum_{i_n 1}^{I_n} x_{i_1 i_2 \ldots i_N} u_{j i_n} \tag{5.3}$$

In the same way as with a SVD, we then can compute a Tucker decomposition with a lower rank (i.e. a rank-$(R_1, R_2, ..., R_N)$ decomposition where one or more $R_n < rank_n(\mathcal{R})$), by removing less important features. This truncated Tucker decomposition is also not exact and approximates the decomposed tensor, but requires less storage [39]. Unfortunately, while truncating a HOSVD does give a good approximation of the given tensor, it does not guarantee to lead to the *best* approximation for the given rank. Thus, unlike what SVD does for matrices, a truncated HOSVD does not result in a minimized Frobenius norm of the differences between the original tensor and the approximation for the given rank.

De Lathauwer et al. propose Higher-Order Orthogonal Iteration (HOOI) [16] to compute a low-rank decomposition with a better fit. The component matrices are initialized in the same way as with HOSVD, but an ALS-based algorithm is used to iterate to a better approximation. HOOI however, does not always lead to a global optimum either.

Similarly to the conversion of SVD to matrix factorization, Karatzoglou et al. [30] adapt HOSVD for recommender systems and use a stochastic gradient descent algorithm to learn their tensor model. Since they use explicit feedback the algorithm considers just the N known ratings and the component matrices and core tensor are regularized based on their Frobenius norm just like 2D matrix factorization. The computation time to create the decomposition is $O(NPQR)$. As noted before, this Multiverse recommendations method gives promising results and delivers a 2.5% to 12% increase in performance compared to other context-aware recommender techniques such as item-splitting.

The disadvantage of Tucker decomposition is the computation time of a prediction. Since Equation 5.1 has three nested loops, the prediction time for a single rating is cubic in the number of features $O(PQR)$. Learning the model can be done offline, but predicting should be done on-demand. Precomputing all predictions is infeasible due to the large storage requirements.

5.2 Candecomp/Parafac

Recall that we mentioned that the rank of a matrix denotes the number of independent rows it contains. Formally the rank of an Nth-order tensor $\mathcal{X}$ is defined as the smallest number of Nth-order *rank-one* tensors, whose sum equals $\mathcal{X}$ (not to be confused with the n-rank of a tensor as introduced previously). Here, an Nth-order tensor is rank-one if it can be decomposed into an outer product of N vectors. For example, a matrix $\mathbf{X}$ (i.e. a second-order tensor) has rank R if it can be decomposed into a sum of R rank-one matrices (i.e. outer products of two vectors):

$$\mathbf{X} = (\mathbf{a}_1 \circ \mathbf{b}_1) + (\mathbf{a}_2 \circ \mathbf{b}_2) + \cdots + (\mathbf{a}_r \circ \mathbf{b}_r) \tag{5.4}$$

Notice how this is by definition of the matrix product another way of describing matrix factorization:

$$\mathbf{X} = \mathbf{A}\mathbf{B}^T = \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \tag{5.5}$$

Where R equals the number of features in $\mathbf{A}$ and $\mathbf{B}$. Generalizing to a third-order tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ this gives:

$$\mathcal{X} = \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r \tag{5.6}$$

Where $\mathbf{a}_r \in \mathbb{R}^I, \mathbf{b}_r \in \mathbb{R}^J, \mathbf{c}_r \in \mathbb{R}^K$. This is called the Candecomp/Parafac decomposition, see Figure 5.2. It results in a factorization of an Nth-order tensor into a sum of R rank-one Nth-order component tensors. The concept was first proposed by Hitchcock in 1927 and later popularized independently in 1970 as Candecomp (canonical decomposition) by Carrol and Chang and Parafac (parallel factors) by Harshman. We therefore refer to it as Candecomp/Parafac, or in short CP.

CP decomposition can be seen as a special case of Tucker decomposition, where the core tensor $\mathcal{G}$ is superdiagonal (1 on the diagonal, 0 otherwise) and the component matrices all have an equal number of features (i.e.

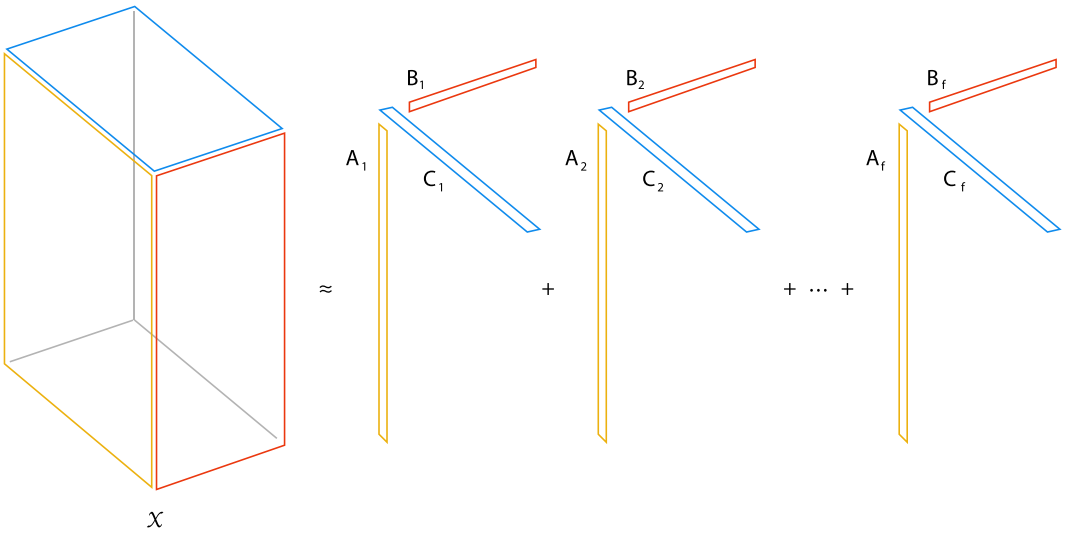

Figure 5.2: CP decomposition of a third-order tensor $\mathcal{X}$

$P = Q = R$) [32]. Tucker decomposition is therefore more general, but as discussed before the computation time is $O(R^3)$, while CP requires $O(R)$ for a third-order tensor. Also note that CP for second-order tensors is equal to matrix factorization (see relation between Equations 5.4, 5.5 and 5.6), whereas Tucker decomposition is not.

To find an exact CP decomposition we need to know the before mentioned rank of a tensor. Where computing the rank of a matrix is relatively easy by for instance calculating the SVD, for tensors this is unfortunately NP-hard. Most solutions just try for $R = 1, 2, 3 \ldots$ until they find a 100% matching rank, but this is first of all not really efficient, and second of all there can be approximations of lower rank that fit arbitrarily close [32].

Once we know the rank and therefore the number of components needed, it is relatively straightforward to compute the CP decomposition: we can simply use an alternating least squares method as we do in the matrix case, solving for a single factor matrix while fixing the others until convergence [32].

For recommender systems we are once again not looking for an exact CP decomposition. A close approximation is what we need. Unfortunately the Eckart and Young theorem that shows that the best rank-k approximation of a matrix $\mathbf{A}$ is given by the highest k factors of the SVD [20], does not apply to tensors as well. Setting R smaller than the rank of a tensor and selecting the best factors of the CP decomposition does therefore not guarantee the best approximation.

While the Tucker decomposition is the most used tensor factorization technique for recommender systems [51], there are some implementations of CP. Xiong et al. propose a Bayesian probabilistic tensor factorization [67], which in their evaluations consistently outperforms the corresponding Bayesian probabilistic matrix factorization. They use amongst others the Netflix dataset with explicit ratings and the associated time variables as context. On this dataset however, the authors don't manage to improve the performance of timeSVD++.

Shi et al. propose TFMAP [54], a tensor factorization for mean average precision (MAP) maximization. Compared to the matrix factorization method from Section 3.3 and Bayesian Personalised Ranking [49] (both non-contextual) TFMAP gives respectively a 14% and 8% improvement in MAP for the dataset they used.

Both approaches outperforming non-contextual recommender systems shows that CP decomposition is a good candidate for adding context to recommenders, despite not always giving an optimal solution in terms of fit with the original tensor. This, together with the faster computation times compared to the Tucker decomposition, made us choose CP for adding contextual data. The following section will propose an adaptation of the non-contextual matrix factorization method for implicit feedback, using CP decomposition.

5.2.1 CP for implicit feedback

We use the alternating least squares algorithm to compute the CP decomposition and follow the same technique as the implicit feedback model without context from Section 3.3.

Let $\mathcal{P}$ be a binary third-order tensor, containing the preferences for items:

$$p_{ijk} = \begin{cases} 1, & r_{ijk} > 0 \\ 0, & r_{ijk} = 0 \end{cases} \tag{5.7}$$

For user i, item j and time k. Note that the time context is split in multiple time bins to make it categorical. The third-order tensor $\mathcal{C}$ contains the confidence of observing p_{ijk}:

$$c_{ijk} = 1 + \alpha r_{ijk} \tag{5.8}$$

Observing $p_{ijk} = 0$ gives a low confidence of the user i actually disliking item j for time k. The confidence in user i liking item j for time k when observing $p_{ijk} = 1$ is linear proportionate to the number of times the user consumed j in time k.

We use the preference values, confidence values and a regularization term, which penalizes large component vectors to prevent overfitting, to define the cost function:

$$S = \sum_{ijk} c_{ijk}(p_{ijk} - \sum_{r} x_{ir}y_{jr}z_{kr})^2 + \lambda \sum_{r}(\|\mathbf{x}_r\|^2 + \|\mathbf{y}_r\|^2 + \|\mathbf{z}_r\|^2) \tag{5.9}$$

Then we can fix $\mathbf{Y}$ and $\mathbf{Z}$, such that we get a quadratic function in the terms of $\mathbf{X}$, which can be minimized by setting the derivative of S with respect to x_{ijk} to zero:

$$\begin{aligned}
\frac{1}{2}\frac{\partial S}{\partial x_{ir}} &= 0 && \forall i, r \\
\sum_{jk} c_{ijk}(p_{ijk} - \sum_{\rho=1}^{R} x_{i\rho}y_{j\rho}z_{k\rho}) \cdot -y_{jr}z_{kr} + \lambda x_{ir} &= 0 && \forall i, r \\
\sum_{jk} y_{jr}z_{kr}c_{ijk}(\sum_{\rho=1}^{R} x_{i\rho}y_{j\rho}z_{k\rho} - p_{ijk}) + \lambda x_{ir} &= 0 && \forall i, r \\
\sum_{jk} y_{jr}z_{kr}c_{ijk}(\sum_{\rho=1}^{R} x_{i\rho}y_{j\rho}z_{k\rho}) + \lambda x_{ir} &= \sum_{jk} y_{jr}z_{kr}c_{ijk}p_{ijk} && \forall i, r \\
\sum_{\rho=1}^{R}(\sum_{jk} y_{jr}z_{kr}c_{ijk} \cdot y_{j\rho}z_{k\rho})x_{i\rho} + \lambda x_{ir} &= \sum_{jk} y_{jr}z_{kr}c_{ijk}p_{ijk} && \forall i, r
\end{aligned} \tag{5.10}$$

To solve this for $\mathbf{x}_i$ we'll construct matrix $\mathbf{\Omega}^{(i)} \in \mathbb{R}^{R \times R}$ and vector $\boldsymbol{\psi}^{(i)} \in \mathbb{R}^R$, where

$$\begin{aligned}
\omega^{(i)}_{\alpha\beta} &= \sum_{jk} y_{j\alpha}z_{k\alpha}c_{ijk} \cdot y_{j\beta}z_{k\beta} \\
\psi^{(i)}_{\alpha} &= \sum_{jk} y_{j\alpha}z_{k\alpha}c_{ijk}p_{ijk}
\end{aligned}$$

This allows us to rewrite Equation 5.10 to:

$$\begin{aligned}
(\mathbf{\Omega}^{(i)} + \lambda\mathbf{I})\mathbf{x}_i &= \boldsymbol{\psi}^{(i)} && \forall i \\
\mathbf{x}_i &= (\mathbf{\Omega}^{(i)} + \lambda\mathbf{I})^{-1}\boldsymbol{\psi}^{(i)} && \forall i
\end{aligned} \tag{5.11}$$

Naive implementation gives us a running time of $O(R^2IJK)$, which is very high. Luckily we can use the same technique as before to lower the time needed to compute $\mathbf{x}_i$. Since $\mathbf{P}_i$ has only n_i non-zero terms (n_i equals the number of ratings given by i), we can calculate $\psi_\alpha^{(i)}$ in $O(n_i)$. Constructing $\boldsymbol{\psi}^{(i)}$ therefore takes $O(Rn_i)$.

If we rewrite $\boldsymbol{\Omega}^{(i)}$, such that:

$$\omega_{\alpha\beta}^{(i)} = \sum_{jk} y_{j\alpha} z_{k\alpha} (c_{ijk} - 1) \cdot y_{j\beta} z_{k\beta} + \sum_{jk} y_{j\alpha} z_{k\alpha} \cdot y_{j\beta} z_{k\beta}$$
$$\omega_{\alpha\beta}^{(i)} = \sum_{jk} y_{j\alpha} z_{k\alpha} (c_{ijk} - 1) \cdot y_{j\beta} z_{k\beta} + \gamma_{\alpha\beta}$$

Where the matrix $\boldsymbol{\Gamma} \in \mathbb{R}^{R \times R}$ is precomputed in $O(R^2JK)$ time and is independent of i, thus:

$$\gamma_{\alpha\beta} = \sum_{jk} y_{j\alpha} z_{k\alpha} \cdot y_{j\beta} z_{k\beta}$$

Since $c_{ijk} - 1 = 0$ for all ijk without implicit feedback (i.e. where $r_{ijk} = 0$), we can construct $\boldsymbol{\Omega}^{(i)}$ much faster. The running time for computing $\mathbf{x}_i$ according to Equation 5.11 is therefore:

- $O(R^2 n_i)$ for constructing and calculating $\boldsymbol{\Omega}^{(i)} + \lambda \mathbf{I}$
- $O(Rn_i)$ for constructing $\boldsymbol{\psi}_i$
- $O(R^3)$ for the matrix inversion $(\boldsymbol{\Omega}^{(i)} + \lambda \mathbf{I})^{-1}$ and matrix multiplication $(\boldsymbol{\Omega}^{(i)} + \lambda \mathbf{I})^{-1} \boldsymbol{\psi}^{(i)}$
- $O(R^2JK)$ for computing $\boldsymbol{\Gamma} \in \mathbb{R}^{R \times R}$

To update the features for each user i and compute the full matrix $\mathbf{X}$ we get a running time of $O(R^2N + R^3I + R^2JK)$, where N equals the total number of implicit feedback. In the same way, we can compute $\mathbf{Y}$ by fixing $\mathbf{X}$, $\mathbf{Z}$ and compute $\mathbf{Z}$ by fixing $\mathbf{X}$ and $\mathbf{Y}$ in similar time. Note that the running time is linear in the input size. We finally alternate the updating of the features for each dimension until convergence.

Initialization

Since the algorithm does not guarantee that a global optimum is reached, the initialization of the feature matrices is important. We can randomly initialize them, but we might end op in a suboptimal local minimum. Running the decomposition multiple times and picking the best result with random initialization is not an option, because that would simple take too much time.

A better and also more intuitive option is to use the data from the utility tensor for initialization, since that's what the feature matrices should predict anyway. We therefore use a method similar to random Acol initialization [38]. The feature columns are filled according to averages from the corresponding values in the utility tensor. These averages are used as variance and mean for a gaussian distribution, which makes sure not every value is the same.

We compared this initialization method to multiple random and mean methods with different variances and the Acol variant consistently resulted in the best predictions. This makes us believe that we can consistently find the best predictions for the CP decomposition.

6 Evaluation

There are various options available to evaluate recommender systems and since each research can take a different approach, it is important to specify exactly how the evaluation is done to be able to compare results. The next sections will give a short description of the diverse protocols, while we explain the choices we made.

6.1 Conditions

The evaluation of recommender systems can be done either *online* (with user interaction), or *offline* (without user interaction). Online evaluations compare the opinions of the users for the item recommendations, which gives arguably the most valuable feedback since it directly gets user satisfaction levels. It is however hard to set up such a system and it requires a large number of users willing to give their feedback. An offline evaluation doesn't bring such a high cost and just uses a part of the historical data to analyze the prediction quality of the recommender system. The tests are easily reproducible and comparing different recommender systems is straightforward [12]. That's why many other research publications use the offline type of evaluation and why we'll use it here as well.

To test the prediction capabilities of our CP method we used a dataset containing the music listening history for Last.fm users. Last.fm is a music recommendation service that collects scrobbles from their signed-up users. Each scrobble means that a user has listened to a specific song, which is our implicit feedback. The dataset is obtained by using the Last.fm api and is freely available to download [13]. It contains a little over 19 million individual scrobbles for 992 users and around 15 million songs.

Because we want an offline evaluation, the dataset should be split in a training, validation and test set. This partitioning of the data is done to prevent overfitting. When estimating the free parameters of the model, the training set is used to train the factorization model, while the validation

part is used to measure the performance. After the parameters that give the best results are found, we use the combined training and validation set for another round of training and finally measure the quality of the predictions on the test set. Again this is done to prevent overfitting of the validation set, since that is used to select the best parameter settings. It is important that the parameters won't be changed after the validation phase, since that could overfit the test set. Therefore, the outcome of the evaluation of the test set is the final value we'll use as a performance measurement and to compare other recommender systems. In other words: the model tries to fit the *training* data as well as possible, while the parameters are tweaked to fit the *validation* set as well as possible. The test set will give the final performance on a never-seen-before part of the data.

While cross-validation (the process of splitting into multiple subsets and using each subset as test data and the remaining subsets as training data) prevents overfitting even more, we believe there is no real need for it, since the dataset is large enough. The test and validation set contain enough implicit feedback to give statistically significant results. We choose a 20% split for the test data and another 20% split of the remainder for the validation set. This leaves 64% of the full dataset as training data, see Figure 6.1 for a small overview.

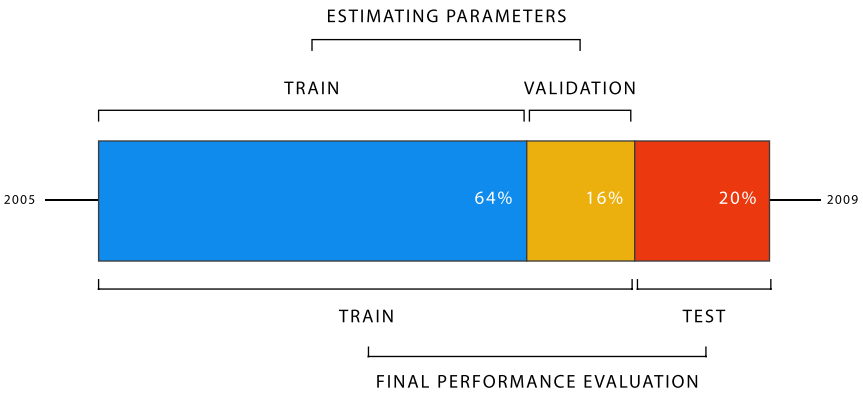

Figure 6.1: Training, validation and test set

In order to separate the training and test sets we can choose to do a *community-centered* split, or a *user-centered* split. The community-centered split will simply take the full dataset of all users and separate the ratings. This could mean however, that some users (or items) are not included in either the training or test set, due to for example large differences in the amount of ratings between users. The recommender system can't evaluate

users not in the training set, since it has no knowledge about their preference. Likewise, it can't evaluate users not in the test set, since there is nothing to compare our predictions with. A user-centered split tries to prevent that and splits the ratings for each user separately. That makes sure that each user will be represented in both the training and test set.

Next is the choice between a *time-independent* and *time-dependent* split. A time-independent split will randomly pick ratings, either from the full dataset or per user according to respectively a community-centered or user-centered split. This immediately brings the problem of having ratings in the training set which are more recent than some ratings in the test set. That means the recommender system has future knowledge about user preferences and might give if the system an unfair advantage. Time-dependent splitting of the data ensures that the rating data is ordered by their timestamps. It prevents knowledge of future preferences, since all ratings in the test set are more recent than those in the training set.

Note that a strict time-dependent split (where all preferences in the test set are more recent than those in the training set) is only possible with a community-centered split, because in a user-centered split some preferences in the training set for one user may be more recent than the preferences in the test set for another user. This combination of time-dependent and community-centered is considered the most realistic split, since it would occur in real-world situations as well: up until a certain time the system will have collected implicit feedback and the recommender predicts the future preferences for the users. The problem of having users or items not present in the training or test set are a common real-world situation as well – consider a new album that will be out in the near future, a collaborative system does not have any information about which users will like it. Users that have no recorded history yet are of course not considered either.

Because of these similarities to real-world applications, we'll use the time-dependent community-centered split for our evaluations.

6.2 Metrics

After the data is processed we need a way to measure the performance of the recommender system. An often used metric for explicit feedback is the Root Mean Squared Error. This measures the distance between the predicted ratings and the ratings in the test set. Unfortunately we can't do this for implicit feedback, since the implicit ratings are not as easily comparable – there is no information about dislikes and no upper bound on the number of times an item can be consumed.

Another method is to represent the predictions as a ranked list of items and compare that with the test set. An often used metric in information retrieval is Mean Average Precision (MAP), also used in [54]. For each item in the test set the precision is calculated. At rank k in the predictions list the precision is defined as:

$$p_k = \frac{1}{k} \sum_{l=1}^{m} x_k$$

Where x_k equals 1 if item k is relevant (it is in the test set), or 0 if it is not relevant (not included in test set). These precision values are averaged for each query and finally the mean is calculated for all averages. Because it assigns a lot more weight to items in the top of the ranking, then to those in the bottom, it prefers systems that return relevant items fast. However, this may not be the best metric, since it assumes that a user dislikes the items that are not in the test set. But that is no reliable information about false positives, recall that blanks (zeroes) in implicit data are not always dislikes.

Another metric we used is the mean weighted rank (WRank), also used by Hu et al. [28]. Since it is recall based, it only considers the relevant items and their predictions. For each user (and each timebin) we'll predict the preferences for all items and sort them to get an ordered list. Then, $rank(i, j, k)$ denotes the (relative) position of item j in that ordered list, for user i and context k. Finally the ranks are weighted according to how many times a user listened to a song. That means that the rank for a song that the user listened to often in the test set is more important than a song the user listened to only few times.

$$WRank = \frac{\sum\limits_{i,j,k \ \in \ test \ \ set} r_{ijk} \times rank(i, j, k)}{\sum\limits_{i,j,k \ \in \ test \ \ set} r_{ijk}}$$

A WRank of 0 is considered the best score, since that means that the right items (i.e. the songs in the test set) are the ones with the highest prediction scores. A totally random prediction will give an expected average rank of 0.5.

6.3 Results

To incorporate the contextual time dimension we have to split the timestamps associated with the item consumption into timebins. We tried several different splits, including 24 timebins – one for each hour of the day –

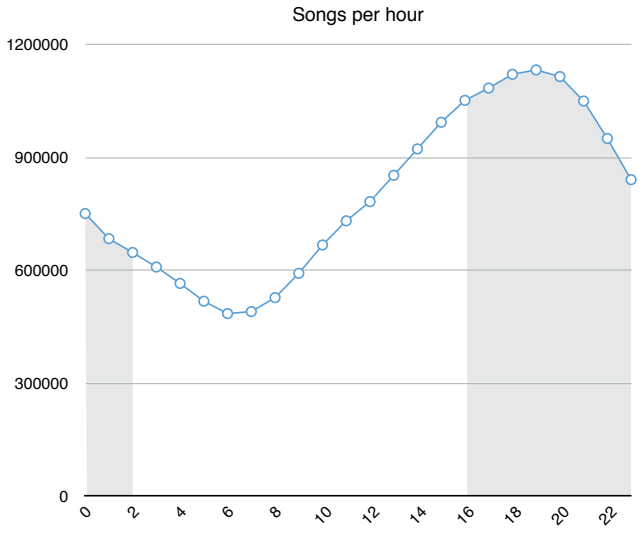

Figure 6.2: Amount of songs per hour in the dataset. We split the data between 02:00 and 16:00.

a workday/weekend binary split and a morning/evening split. We observed that the best way to split the data was to create equal sized bins, i.e. where each timebin held an almost equal amount of implicit feedback. This is because when there is an uneven split, the recommender system might predict the items in the smaller timebin better, but the ones in the larger bins worse.

The best performing split turned out to be the one where we separated songs listened to from 16:00 until 02:00. This split showed the most variance in listening behavior and gave the best prediction results. Figure 6.2 shows the amount of songs listened to per hour and the split we've chosen.

While estimating parameters α and λ (the confidence and regularization factors, as introduced in Equation 5.8 and 5.9 respectively) we observed the following:

- If λ is high, the predicted preference values will be closer to 0, since there is a preference for low values in the feature matrices (i.e. a relatively low penalty for false negatives)
- If α is high, the predicted preference values will be closer to 1, since there is a large reward for a high confidence (i.e. a relatively low penalty for false positives)

This applied to all items, whether they had high or low implicit feedback. For explicit feedback this balance is something to take into account, but since we

use ranking based metrics this does not really matter – as long as the order is correct. The evaluation of multiple values on the validation set showed that $\lambda = 1$ and $\alpha = 30$ gives good results.

For both matrix factorization and tensor factorization the number of features that gives best results lies around 10. We observed that a higher number makes the models capture too much irrelevant features and the performance therefore degrades, while a lower number of features captures not enough features.

The final results for both models are plotted in Figure 6.3 and 6.4. For comparison the popularity predictor is included as well. This predictor ranks items based on its global popularity and is one of the simplest recommender systems. It is therefore of interest to at least outperform this system with the other recommenders. Results for the naive context-aware matrix factorization are omitted, since they are equal to (or just slightly worse than) the 2D matrix factorization results. This method therefore shows no further improvements.

The values associated with random predictions are 0.00106 for MAP and 0.5 for WRank. As we can see, all three recommender systems outperform a random algorithm by far. The MAP scores for both factorization techniques flatten around 18 iterations. Tensor factorization improves upon the popularity predictor by more than 50%. However, compared to non-contextual matrix factorization our context-aware approach lags behind. That system consistently keeps outperforming tensor factorization by around 17%.

The same is true for the WRank scores, albeit a little less significant. Matrix factorization gives around 3% better predictions, compared to tensor factorization. Although it is close, we did not expect our approach to be a worse predictor. Increasing or decreasing the number of features does not give a better performance either.

Our context-aware solution does not improve the results of the non-contextual recommender, which can mean two things: the algorithm is not not able to pick up latent-factors in higher-order dimensions, or the contextual data in this particular Last.fm dataset does not contain enough structure to be able to use it for recommending.

The latter means that the added dimension adds another layer of noise, which does not help improving the performance. To test that the algorithm works we artificially constructed a dataset where items are divided in timebins according to their artist. This means that in this set a song is always put into the same timebin, such that we can reject the possibility that this set contains not enough latent information in the time dimension. The tensor based algorithm does give in this case a much better WRank (0.0428137 - a 30% improvement) compared to the matrix based one. The MAP stays al-

most the same however, which means that the distribution of relevant items on the ranked prediction list stays largely equal, while some of the most preferred items (i.e. the songs with most listens in the test set) were ranked higher.

This tells us the tensor based approach is in fact able to learn structures in higher dimensions, albeit structures with highly dependent relations. To test if it is at all possible to improve recommendations using the contextual information in the Last.fm dataset, we can try other contextual pre-filtering, post-filtering or model-based methods. This is something that we can research and test further in the future.

Finally we compare the running times for the 2D and 3D methods. As we can see in Figure 6.5 the measured running times per iteration are both close to linear in terms of the input size, which confirms our earlier deduced running times. Furthermore, the tensor factorization method takes a little less than twice as much time compared to the matrix factorization (measured with two timebins). This, together with worse prediction quality, makes our approach to exploiting context with a CP system for the evaluated dataset in its current form unfortunately no viable alternative.

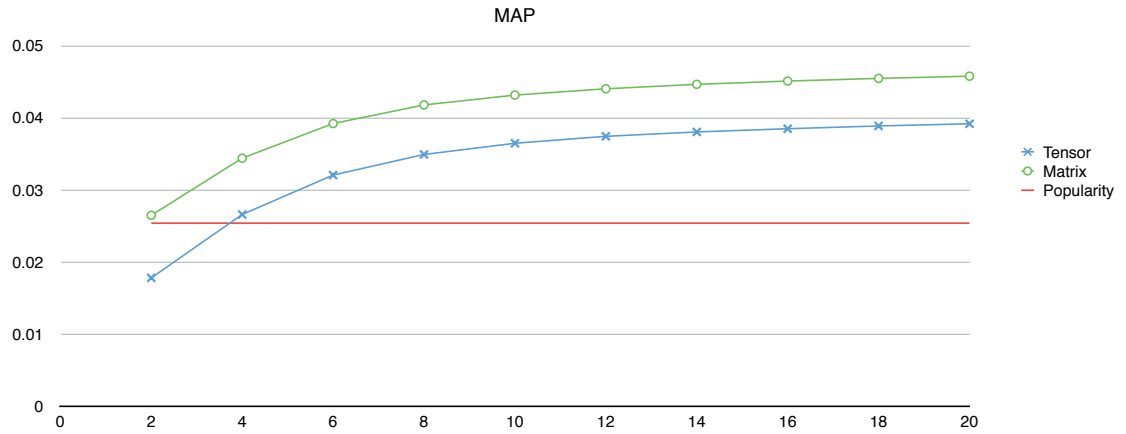

Figure 6.3: Mean average precision for tensor factorization, matrix factorization and popularity predictor after 20 iterations. Higher MAP equals a better performance.

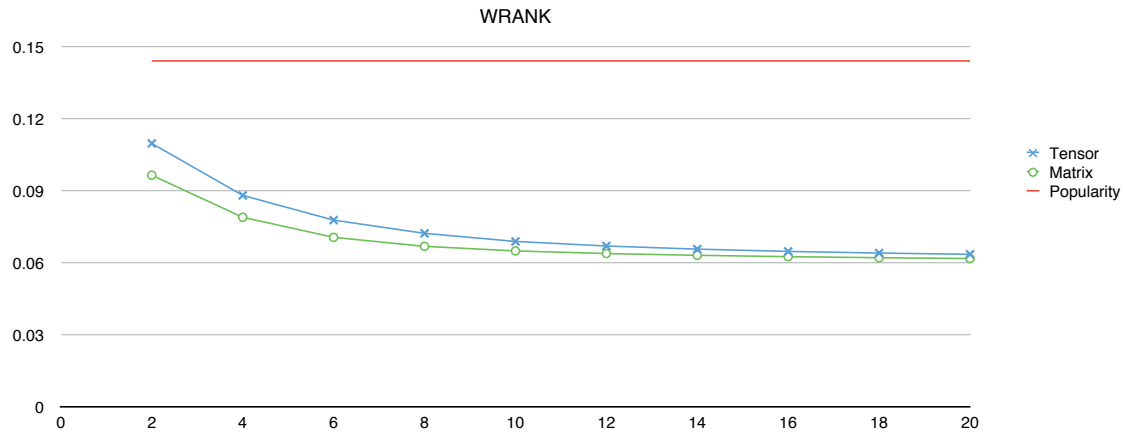

Figure 6.4: Mean weighted rank scores for 20 iterations. Lower rank equals a better performance.

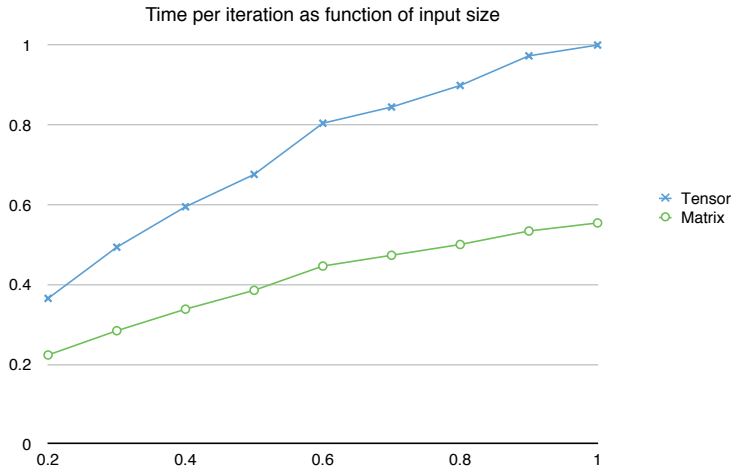

Figure 6.5: Normalized running time per iteration.

7 Conclusion

In this thesis we have extended the 2D matrix factorization method for implicit feedback datasets to 3D tensor factorization based on Candecomp/Parafac. This way we can incorporate context in a recommender system to improve prediction results. We explored the systems which matrix factorization is based upon, specifically PCA and SVD, and gained a deeper understanding of why this method works. Furthermore, we compared different recommender techniques like neighborhood recommenders and neural networks and discussed their advantages and disadvantages.

To incorporate context-awareness we chose to focus on adding a time dimension to the utility matrix. This makes the utility matrix even more sparse and more difficult to factorize. Since pre-filtering and post-filtering techniques don't use all available information and model-based recommenders have previously provided good prediction results for the Netflix Prize amongst others, we adopted contextual modeling to integrate context into matrix factorization.

Two main tensor factorization techniques were presented: Tucker decomposition and Candecomp/Parafac. While some recommender systems are based on the Tucker decomposition, we favored CP, because of the lower computation time of predictions and the similarities with matrix factorization. The computation time is especially important when dealing with an online system serving predictions for many users at once. Precomputing predictions is infeasible due to the large storage requirements.

While most recommenders use explicit feedback, we chose to focus on implicit feedback. This kind of data is more easily available and requires no extra user interaction. The drawback is that implicit feedback is of a lower quality and does not include negative feedback. This needs to be taken into account when designing a recommender system.

Finally we listed important evaluation methodologies and explained the metrics we used to measure our recommender's performance. We implemented the context-aware tensor factorization model for implicit feedback and compared the results with the non-contextual matrix factorization and

popularity predictor. Unfortunately the results show no improvements compared to the non-contextual factorization method, which may be due to a lack of latent factors in the contextual data.

To test if the tensor based algorithm actually works and can be used to improve prediction results, we artificially created a dataset with adjusted timebins. Results showed that in that case the solution does indeed work and gives up to a 30% improvement over non-contextual matrix factorization. However this artificial highly-related information may not be available in real world datasets, which means that we cannot currently recommend our proposed method as a viable solution in practice, but this requires testing other datasets to be conclusive.

Investigating if the contextual data in the Last.fm dataset can be exploited in other ways to improve recommendations is something we need to research in the future. The structure of the contextual information may be more local, instead of the global latent features that factorization methods try to discover.

The running time measurements showed a linear relation with the input size, which shows that alternating least squares is a scalable solution for large scale databases. Compared to matrix factorization it is around 50% slower with two timebins. While this may be acceptable if it showed significant performance improvements, this is unfortunately not the case for our measurements.

7.1 Further research

There are a couple of things we can do to improve our results. First of all, our work only captures global latent features, spanning all data. Solutions like SVD++ track local user drifts – assuming preferences for a single user changes over time. This may better model the time dimension than searching for global influences.

It would be interesting to compare our results with a solution based on Tucker decomposition. The extra free parameters in that model may capture the latent features more easily. The running time increase may be worth it depending on the increase in performance.

Since the tensor decomposition is a type of dimensionality reduction, we would like to test what the prediction quality will be for a neighborhood recommender based on the reduced dimensionality. Users, items and context are after all directly comparable using the corresponding feature vectors. While performance of neighborhood recommenders may be less than model-based ones, applying it to the reduced latent space may take away the noise

and the difficulty of comparing users and items. This may result in improved prediction quality.

7.1.1 Groups

While before mentioned future additions should improve performance, the next feature is something that makes the recommender more generic and usable for more applications. Since we can generalize the CP algorithm to more dimensions, we can add an arbitrary amount of contexts. We would like to research if it is possible to make one of those contextual dimension the group of people a user is in.

In a party with a group of friends, or during the day at work with colleagues it is often difficult to find music that everyone (or at least the majority) would like to hear. For these scenarios we want to know what multiple people together - as a group - prefer, such that we can recommend music to them all at once.

Most literature about recommender systems for groups only focuses on creating individual predictions and combining them after the prediction phase (like contextual post-filtering). Based on this combination of preferences music is recommended: for instance least-misery will recommend music that maximizes the lowest predicted preference in the group, while an average combination recommends music with highest average prediction [43].

But what would be a solution to truly incorporate groups into the model? We can see individual users as a different context dimension, but this would create a very high-dimensional utility matrix, which would currently require too much computing power to decompose with tensor factorization. Food for thought for a future recommender implementation.

Bibliography

[1] Gediminas Adomavicius and Alexander Tuzhilin. Context-aware recommender systems. In Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors, *Recommender Systems Handbook*, pages 217–253. Springer US, 2011.

[2] Xavier Amatriain, Alejandro Jaimes, Nuria Oliver, and Josep M. Pujol. Data mining methods for recommender systems. In Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors, *Recommender Systems Handbook*, pages 39–71. Springer US, 2011.

[3] Apple. Apple unveils new itunes. `http://www.apple.com/pr/library/2012/09/12Apple-Unveils-New-iTunes.html`, 2012. [Online; accessed 23-June-2015].

[4] Linas Baltrunas. *Context-aware collaborative filtering recommender systems*. PhD thesis, Free University of Bozen-Bolzano, 2011.

[5] Linas Baltrunas and Xavier Amatriain. Towards time-dependant recommendation based on implicit feedback. In *Workshop on context-aware recommender systems (CARS09)*, 2009.

[6] Robert Bell, Yehuda Koren, and Chris Volinsky. Modeling relationships at multiple scales to improve accuracy of large recommender systems. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 95–104, New York, NY, USA, 2007. ACM.

[7] Robert M. Bell and Yehuda Koren. Scalable collaborative filtering with jointly derived neighborhood interpolation weights. In *IEEE International Conference on Data Mining (ICDM)*, 2007.

[8] Robert M. Bell, Yehuda Koren, and Chris Volinsky. The BellKor solution to the Netflix Prize.

[9] Fredrik Boström. Andromedia-towards a context-aware mobile music recommender. 2008.

[10] John S. Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, UAI'98, pages 43–52, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[11] Maritza L. Calderón-Benavides, Cristina N. González-Caro, José de J. Pérez-Alcázar, Juan C. García-Díaz, and Joaquin Delgado. A comparison of several predictive algorithms for collaborative filtering on multi-valued ratings. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, pages 1033–1039, New York, NY, USA, 2004. ACM.

[12] Pedro G. Campos, Fernando Díez, and Iván Cantador. Time-aware recommender systems: a comprehensive survey and analysis of existing evaluation protocols. In *User Modeling and User-Adapted Interaction*, volume 24, pages 67–119. Springer Netherlands, 2014.

[13] Òscar Celma. Music recommendation datasets for research. `http://www.dtic.upf.edu/~ocelma/MusicRecommendationDataset/lastfm-1K.html`, 2010. [Online; accessed 9-April-2015].

[14] Eva Ceulemans and Henk A. L. Kiers. Selecting among three-mode principal component models of different types and complexities: A numerical convex hull based method. *Br J Math Stat Psychol*, 59:133–150, 2006.

[15] Pádraig Cunningham. Dimension reduction. In Matthieu Cord and Pádraig Cunningham, editors, *Machine Learning Techniques for Multimedia*, Cognitive Technologies, pages 91–112. Springer Berlin Heidelberg, 2008.

[16] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. On the Best Rank-1 and Rank-(R1,R2,...,RN) Approximation of Higher-Order Tensors. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1324–1342, January 2000.

[17] Bertrand Dechoux. Recommender systems, naive bayes networks and the netflix prize. 2009.

[18] Mukund Deshpande and George Karypis. Item-based top-n recommendation algorithms. *ACM Trans. Inf. Syst.*, 22(1):143–177, January 2004.

[19] Christian Desrosiers and George Karypis. A comprehensive survey of neighborhood-based recommendation methods. In Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors, *Recommender Systems Handbook*, pages 107–144. Springer US, 2011.

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

[21] Simon Funk. Netflix update: Try this at home. `http://sifter.org/~simon/journal/20061211.html`, 2006. [Online; accessed 9-April-2015].

[22] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Commun. ACM*, 35(12):61–70, December 1992.

[23] Michele Gorgoglione, Umberto Panniello, and Alexander Tuzhilin. The effect of context-aware recommendations on customer purchasing behavior and trust. In *Proceedings of the Fifth ACM Conference on Recommender Systems*, RecSys '11, pages 85–92, New York, NY, USA, 2011. ACM.

[24] Miha Grčar, Blaž Fortuna, Dunja Mladenič, and Marko Grobelnik. Knn versus svm in the collaborative filtering framework. In Vladimir Batagelj, Hans-Hermann Bock, Anuška Ferligoj, and Aleš Žiberna, editors, *Data Science and Classification*, Studies in Classification, Data Analysis, and Knowledge Organization, pages 251–260. Springer Berlin Heidelberg, 2006.

[25] N. Halko, P. G. Martinsson, and J. A. Tropp. Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions, 2010.

[26] Negar Hariri, Bamshad Mobasher, and Robin Burke. Context-aware music recommendation based on latent topic sequential patterns. In *Proceedings of the Sixth ACM Conference on Recommender Systems*, RecSys '12, pages 131–138, New York, NY, USA, 2012. ACM.

[27] Jonathan L. Herlocker, Joseph A. Konstan, Al Borchers, and John Riedl. An Algorithmic Framework for Performing Collaborative Filtering. In *Proceedings of the 22Nd Annual International ACM SIGIR*

Conference on Research and Development in Information Retrieval, SIGIR '99, pages 230–237, New York, NY, USA, 1999. ACM.

[28] Yifan Hu, Yehuda Koren, and Chris Volinsky. Collaborative filtering for implicit feedback datasets. In *In IEEE International Conference on Data Mining (ICDM 2008)*, pages 263–272, 2008.

[29] Rong Jin, Luo Si, and Chengxiang Zhai. A study of mixture models for collaborative filtering. *Information Retrieval*, 9(3):357–382, 2006.

[30] Alexandros Karatzoglou, Xavier Amatriain, Nuria Oliver, and Linas Baltrunas. Multiverse recommendation: n-dimensional tensor factorization for context-aware collaborative filtering. In *In Proceedings of the fourth ACM conference on Recommender systems*, 2010.

[31] Myung Won Kim, Eun Ju Kim, and Joung Woo Ryu. Collaborative filtering for recommendation using neural networks. In Osvaldo Gervasi, Marina L. Gavrilova, Vipin Kumar, Antonio Laganà, Heow Pueh Lee, Youngsong Mun, David Taniar, and Chih Jeng Kenneth Tan, editors, *Computational Science and Its Applications ICCSA 2005*, volume 3480 of *Lecture Notes in Computer Science*, pages 127–136. Springer Berlin Heidelberg, 2005.

[32] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM REVIEW*, 51(3):455–500, 2009.

[33] Joseph A. Konstan, Bradley N. Miller, David Maltz, Jonathan L. Herlocker, Lee R. Gordon, and John Riedl. Grouplens: Applying collaborative filtering to usenet news. *Commun. ACM*, 40(3):77–87, March 1997.

[34] Yehuda Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *In Proc. of the 14th ACM SIGKDD conference*, pages 426–434, 2008.

[35] Yehuda Koren. Collaborative filtering with temporal dynamics. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 447–456, New York, NY, USA, 2009. ACM.

[36] Yehuda Koren. Factor in the neighbors: Scalable and accurate collaborative filtering. *ACM Trans. Knowl. Discov. Data*, 4(1):1:1–1:24, January 2010.

[37] Yehuda Koren and Robert Bell. Advances in collaborative filtering. In Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors, *Recommender Systems Handbook*, pages 145–186. Springer US, 2011.

[38] Amy N. Langville, Carl D. Meyer, and Russell Albright. Initializations for the Nonnegative Matrix Factorization. In *KDD 2006*, 2006.

[39] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. A multilinear singular value decomposition. *SIAM J. Matrix Anal. Appl*, 21:1253–1278, 2000.

[40] Pasquale Lops, Marco de Gemmis, and Giovanni Semeraro. Content-based recommender systems: State of the art and trends. In Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors, *Recommender Systems Handbook*, pages 73–105. Springer US, 2011.

[41] Koji Miyahara and Michael J. Pazzani. Collaborative filtering with the simple bayesian classifier. In Riichiro Mizoguchi and John Slaney, editors, *PRICAI 2000 Topics in Artificial Intelligence*, volume 1886 of *Lecture Notes in Computer Science*, pages 679–689. Springer Berlin Heidelberg, 2000.

[42] Douglas W. Oard and Jinmook Kim. Implicit feedback for recommender system. Technical report, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer, 1998.

[43] Mark O'Connor, Dan Cosley, Joseph A. Konstan, and John Riedl. Polylens: A recommender system for groups of users. In *In Proceedings of the European Conference on Computer-Supported Cooperative Work*, pages 199–218. Kluwer Academic, 2001.

[44] Kenta Oku, Shinsuke Nakajima, Jun Miyazaki, and Shunsuke Uemura. Context-aware svm for context-dependent information recommendation. In *Proceedings of the 7th International Conference on Mobile Data Management*, MDM '06, pages 109–, Washington, DC, USA, 2006. IEEE Computer Society.

[45] Han-Saem Park, Ji-Oh Yoo, and Sung-Bae Cho. A context-aware music recommendation system using fuzzy bayesian networks with utility theory. In Lipo Wang, Licheng Jiao, Guanming Shi, Xue Li, and Jing Liu, editors, *Fuzzy Systems and Knowledge Discovery*, volume 4223 of *Lecture Notes in Computer Science*, pages 970–979. Springer Berlin Heidelberg, 2006.

[46] Arkadiusz Paterek. Improving regularized singular value decomposition for collaborative filtering. In *Proceedings of KDD cup and workshop*, volume 2007, pages 5–8, 2007.

[47] Martin Piotte and Martin Chabbert. The pragmatic theory solution to the netflix grand prize. *Netflix prize documentation*, 2009.

[48] Anand Rajaraman, Jeffrey D. Ullman, and Jure Leskovec. *Mining of Massive Datasets*. Cambridge University Press, December 2011.

[49] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. Bpr: Bayesian personalized ranking from implicit feedback. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, UAI '09, pages 452–461, Arlington, Virginia, United States, 2009. AUAI Press.

[50] Steffen Rendle and Lars S. Thieme. Pairwise interaction tensor factorization for personalized tag recommendation. In *Proceedings of the third ACM international conference on Web search and data mining*, WSDM '10, pages 81–90, New York, NY, USA, 2010. ACM.

[51] Giuseppe Ricci, Marco de Gemmis, and Giovanni Semeraro. Mathematical methods of tensor factorization applied to recommender systems. In Barbara Catania, Tania Cerquitelli, Silvia Chiusano, Giovanna Guerrini, Mirko Kmpf, Alfons Kemper, Boris Novikov, Themis Palpanas, Jaroslav Pokorn, and Athena Vakali, editors, *New Trends in Databases and Information Systems*, volume 241 of *Advances in Intelligent Systems and Computing*, pages 383–388. Springer International Publishing, 2014.

[52] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted boltzmann machines for collaborative filtering. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 791–798, New York, NY, USA, 2007. ACM.

[53] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proc. 10th International Conference on the World Wide Web*, pages 285–295, 2001.

[54] Yue Shi, Alexandros Karatzoglou, Linas Baltrunas, Martha Larson, Alan Hanjalic, and Nuria Oliver. Tfmap: Optimizing map for top-n context-aware recommendation. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, pages 155–164, New York, NY, USA, 2012. ACM.

[55] Jonathon Shlens. A tutorial on principal component analysis. In *Systems Neurobiology Laboratory, Salk Institute for Biological Studies*, 2005.

[56] Spotify. Information — spotify press. `https://press.spotify.com/nl/information/`, 2015. [Online; accessed 23-June-2015].

[57] Kostas Stefanidis, Irene Ntoutsi, Kjetil Nørvåg, and Hans-Peter Kriegel. A framework for time-aware recommendations. In StephenW. Liddle, Klaus-Dieter Schewe, AMin Tjoa, and Xiaofang Zhou, editors, *Database and Expert Systems Applications*, volume 7447 of *Lecture Notes in Computer Science*, pages 329–344. Springer Berlin Heidelberg, 2012.

[58] G. W. Stewart. On the early history of the singular value decomposition. *SIAM Review*, 35(4):551–566, 1993.

[59] Xiaoyuan Su and Taghi M. Khoshgoftaar. Collaborative Filtering for Multi-class Data Using Belief Nets Algorithms. In *ICTAI '06: Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence*, pages 497–504, Washington, DC, USA, 2006. IEEE Computer Society.

[60] Xiaoyuan Su and Taghi M. Khoshgoftaar. A survey of collaborative filtering techniques. *Adv. in Artif. Intell.*, 2009:4:2–4:2, January 2009.

[61] Gábor Takács, István Pilászy, Bottyán Németh, and Domonkos Tikk. Scalable collaborative filtering approaches for large recommender systems. *J. Mach. Learn. Res.*, 10:623–656, June 2009.

[62] Nava Tintarev and Judith Masthoff. Designing and evaluating explanations for recommender systems. In Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors, *Recommender Systems Handbook*, pages 479–510. Springer US, 2011.

[63] Andreas Töscher, Michael Jahrer, and Robert Legenstein. Improved neighborhood-based algorithms for large-scale recommender systems. In *Proceedings of the 2nd KDD Workshop on Large-Scale Recommender Systems and the Netflix Prize Competition*, NETFLIX '08, New York, NY, USA, 2008. ACM.

[64] Michael E. Wall, Andreas Rechtsteiner, and Luis M. Rocha. Singular value decomposition and principal component analysis. In DanielP. Berrar, Werner Dubitzky, and Martin Granzow, editors, *A Practical Approach to Microarray Data Analysis*, pages 91–109. Springer US, 2003.

[65] Mingrui Wu. Collaborative filtering via ensembles of matrix factorizations. In *Proceedings of KDD Cup and Workshop*, volume 2007, 2007.

[66] Zhonghang Xia, Yulin Dong, and Guangming Xing. Support vector machines for collaborative filtering. In *Proceedings of the 44th Annual Southeast Regional Conference*, ACM-SE 44, pages 169–174, New York, NY, USA, 2006. ACM.

[67] Liang Xiong, Xi Chen, Tzu-Kuo Huang, Jeff G Schneider, and Jaime G Carbonell. Temporal collaborative filtering with bayesian probabilistic tensor factorization. In *SDM*, volume 10, pages 211–222. SIAM, 2010.

[68] Hongzhi Yin, Bin Cui, Jing Li, Junjie Yao, and Chen Chen. Challenging the long tail recommendation. *Proc. VLDB Endow.*, 5(9):896–907, May 2012.