Master thesis
Data Science

Radboud University

Building a fair recommendation system

Author:
Aucke Bos
s4591496

First supervisor/assessor:
Prof. dr. ir. Arjen de Vries
`arjen.devries@cs.ru.nl`

External supervisor:
Tea Stojanovic
`tea.stojanovic@infosupport.com`

Second assessor:
Prof. Martha Larson
`m.larson@cs.ru.nl`

January 20, 2022

Acknowledgments

I would like to thank a few people, without whom this thesis would not have become what it is now. Firstly, thanks to both Arjen de Vries and Tea Stojanovic for the daily supervision. The (bi)weekly meetings helped me greatly whenever I was stuck on this project; the ease of contacting you both was a relief. Secondly, the discussions, stand-ups, and feedback of the group of interns at Info Support definitely helped me get new insights whenever I encountered problems. I am especially grateful to Sjoerd van Dijk, who really took the time to reason with me about certain problems I had, which surely gave me new insights and helped me solve some major problems. Thanks also to Nikki Thissen for the personal support in our weekly meetings. You helped me when I was struggling with both personal and professional problems and made sure I kept going and was disciplined until the completion of the project. Lastly, thanks to my roommates, partner, and family, who kept encouraging me and had trust in the completion of my thesis. I learned some valuable new skills during this project, and am truly proud of the result.

Abstract

With the number of websites growing every day, the number of choices to be made online can be overwhelming to users. Therefore, the need for personalized recommendation systems is increasing. Many platforms provide recommendations to their users, but they are often misaligned with the user's interest. The question of whether recommendations are based on fair decision-making often remains unanswered.

This research consists of three parts. Firstly, we provide an overview of the many approaches one can use for building a recommendation system. Secondly, we take a look at fairness. What are the considerations to be made? What is the definition of fairness and why is it important? How can fairness be measured, and how can systems make fair decisions? Answers to these questions are discussed; the insights are finally used to build a fair recommendation system that tackles one of the biggest challenges in recommendation systems. The evaluations show that although the system does not improve upon the state-of-the-art in terms of effectiveness, it is able to make fairer recommendations without a significant loss in effectiveness.

Contents

Chapter 1

Introduction

1.1 What are recommendation systems?

People navigating the web are often overwhelmed with the choices they have to make [25]. In many cases, it is impossible to manually make a decision between many options. Recommendation systems help users with these decisions. Based on information extracted from users, items, and interactions, they decide which items might be interesting for which users. To get an idea of what such a system looks like, we discuss some specific examples of recommendation systems. These examples will stress the widespread use of recommendation systems in daily life, and bring to light some of the challenges they entail.

- Video streaming services such as Netflix use a recommendation system to recommend movies to users. The number of videos that are available is way too large for users to see all of them. Therefore they are in great need of a recommendation system. In this system, the items are all the movies that are present, the users are people who want to watch a movie.

  In this example, items can be described by attributes like genre, producer, and creation date. In many movie recommendation systems, users are only described by the movies that they have seen so far. The history of the user is then used to deduce his preferences for other movies. If we consider the act of watching a movie by a user as an indication that the user likes the movie, we will encounter problems. If a user only watches the movie for a few minutes before turning it off, the system should probably register negative feedback instead of positive feedback. Hence we might need a more sophisticated way to translate watch time into 'rating'. In this example, we are using watch time as an *implicit feedback* report. Netflix also uses *explicit feedback* in their recommendation system, by asking users whether they liked a specific movie. We elaborate more upon feedback types and feedback retrieval in Section 2.1.3.

A certain type of problematic recommendation behavior we often encounter in recommendation systems is called *popularity-based ranking*. This occurs when a recommendation system (RS) tends to recommend popular (highly rated) items to every user. This type of recommendation is problematic for many reasons, as discussed in Section 2.1.2. Recommendations based on popularity are often seen in real-life recommendation systems, and one should definitely take this pitfall into account when designing one. Even a company as Netflix cannot fully solve this problem yet, as we often see movies that are recommended to almost all users.

- Probably one of the most used recommendation systems: Google search. While you might not directly think of the Google search engine as a recommendation system, it is quite easy to understand how we can see it as one.

  The number of available websites and web pages is again way too large to manually investigate all of them when looking for some information. Therefore you use Google to enter a search query, and you expect the system to return the most relevant web page(s) for your task. In some cases, you know specifically which website you are looking for, and you expect this to be the first result of your query. In some cases, you are researching a broad subject, and you want, for example, 10 relevant websites that contain information about the subject. In that case, you want Google to recommend the 10 best-matching items to you, given your query. In literature, we call a search engine like this a *ranking system*. As we will see later on, ranking systems and recommendation systems are closely related (Section 3.1.4.1).

  In this setting, the users are primarily described by the query they are providing. Additional attributes might also be used to describe the user, for example, location, browser type, and search history. When the system describes users by their historical search history, it can be said that they are described by the pages they have viewed [52]. This way of representing users is the basis for Collaborative Filtering Systems (Section 2.3). On the other hand, we have items (web pages); they are described by the text they contain, outgoing links to other pages, pages referring to the page, the age of the page, etcetera.

- The more classical recommendation system: The "You might also like these items.." popups you encounter during online shopping. These are recommendation systems in the most literal sense.

  You as a user can be described by many different attributes. Classical attributes are your gender, age, ethnicity, etcetera. Other relevant attributes might be your past purchase history, your interaction with other users, and the time you spend on web pages. The items are the products that are being sold. Such items contain classic attributes in many cases, like color, size, and attribute set. One can however also describe items in terms of the users that have bought them. Consider a webshop that sells a jacket. The employee that added the product to the system, did not select the gender for which the jacket is suited. If we find that the jacket is sold several times to men, and never to women, we can deduce the gender for which the jacket is suited. When we know this, we should not recommend this product to female users. This way we have used (the behavior of) the users to describe attributes of the items. A Knowledge-Based Recommendation System (Section 2.4) would deduce specifically that the jacket is suited for men. A Collaborative Filtering approach would not recognize it as being a gender but rather sees that the jacket is suited for similar (e.g. male) rather than dissimilar (e.g. female) users.

A problem arises for new items with few attributes. If the attributes are not descriptive enough, using them is not enough to accurately recommend them to users for which they might be interesting. We cannot deduce attributes for these items by their selling history, since the item is new in the system. This problem is known as the *cold-start* problem, and it is present in many recommendation systems. This problem arises for both new users and new items, and we discuss it extensively in Section 2.1.1.

By providing these examples of real-life recommendation systems, we hope to give the reader a general idea of the broad scope of recommendation systems. We have also mentioned several of the many challenges that will come across when building such a system. One of the goals of this thesis is to tackle one of the most stubborn problems in recommendation systems. To select one, we first provide an overview of the different approaches to building a recommendation system. Then we select one problem and choose a specific approach that performs well but indeed suffers from that problem. We alter the approach to better handle the challenge and implement and evaluate it.

1.2 What is fairness?

The focus of this project is on the fairness of recommendation systems. Many unfair decisions can be made in the recommendation process. In Chapter 3, we provide an overview of different types of fairness definitions, and how they integrate with recommendation systems. Based on this survey, we choose one fairness metric and apply it to the specific RS we implemented. The fairness of the system is measured, and the balance between fairness and effectiveness is investigated.

Let us briefly demarcate what we mean by fairness. In literature, people usually mean the *lack of discrimination*, when talking about fairness [15, 39, 48, 40]. Let us define what this means in three different machine learning systems:

- The goal of classification algorithms is to classify an item into one or more clusters. The decision of which category an item belongs to is based on the attributes of the item. Some attributes might be *protected* or *sensitive* attributes. If we want the algorithm to be fair, we want to be certain that the value an item has for this attribute does not influence the category the item is categorized into.

  As a more specific example, consider a single-label clustering system with two categories: positive or negative. The items are people, and their attributes are for example gender, ethnicity, age, study, place of birth, etcetera. The system wants to determine which people are suited for a certain job, hence it will classify them into positive or negative. If we want this system to be fair in terms of ethnicity, we could define this as follows: Suppose we have item i with ethnicity $r \in R$, where R is the set of all possible ethnicities. i is clustered into cluster 0 (negative). Now for all $r' \in R - \{r\}$, if we assign r' as the ethnicity of i, i should still be clustered into class 0. This approach is an *individual counterfactual* way of looking at fairness, which we will elaborate upon in Section 3.2.1.2.

  Another possibility would be to look at the distribution of ethnicities in both classes. If they are not evenly distributed along with the classes, this might also indicate unfairness. This type of fairness is a *group-based base-rate* approach based on statistical parity, as discussed in Section 3.2.1.1.

- A ranking algorithm has the goal of ranking (ordering) a set of items I based on some quality measure [39]. Similar to clustering algorithms, sensitive attributes must not influence the rank of an item.

  Suppose we have a ranking system for academic papers. Users can enter a search query, and the system provides a list (ranking) of relevant papers. In the ranking system, the authors of the papers might be taken into account when matching the query with papers. But this system should provide items based on relevance to the query, and not based on the popularity of the author. Thus we would want to make sure that the number of links to papers of a certain author does not influence the rank of a paper of that author in the list, apart from the indication that these links give about the relevancy of the content.

- A recommendation system tries to find interesting items for a user. The 'score' of an item for a user is usually defined in terms of a predicted rating: To what extent the system thinks the user would like the item. In a fair recommendation system, the value of protected attributes would not influence the predicted rating, and thus whether or not the item would be recommended [39]. A recommendation system could recommend one item sequentially each time, or it could produce a ranked list of recommendations. In the latter case, fairness metrics are directly applicable. In the first case, we can group several recommendations into a list, such that fairness can be measured over the ranking, as with ranking systems. Alternatively, metrics exist for sequential recommendation systems, as discussed in Section 3.2.3.

These examples of fairness in algorithms stress the importance of the concept. The designer of a system should always pay extra attention to sensitive attributes, and make sure they are handled with care. Thus when we define 'ethnicity' as a sensitive attribute, a clustering algorithm should not differentiate on it. Adding fairness constraints to an existing system introduces tension: On the one hand, we have the need for personalized and accurate recommendations, and on the other hand, we have regulatory constraints to ensure fairness over the outcomes [12]. In Section 3 we provide an overview of approaches to defining and measuring fairness.

1.3 Research questions

Our research consists of three parts. Each part has its dedicated research question and chapter in this thesis.

1.3.1 Challenges in recommendation systems

In this project, we investigate different approaches for building a recommendation system. We are specifically interested in the challenges that are found in each approach, and how they relate to each other. We want to select one challenge that is found in many approaches, and will try to tackle it by combining the strengths of different methods; we call this *hybridization*, which is discussed in Section 2.7. Therefore we first do an extensive literature review into papers about recommendation systems. The goal is to investigate their pros and cons and to relate them to each other. This literature review is used to select one problem that we will try to tackle.

Hence our first research question reads:

RQ 1 *What approaches exist for building a recommendation system, and what are their strengths and weaknesses?*

1.3.2 Approaches to defining fairness

Taking fairness into account in a recommendation system is not straightforward. In literature, many different approaches are used. Many of them focus on *distributive fairness*, which is fairness measured in terms of distributions of different attributes within different groups over the resulting rankings [7, 37]. An alternative viewpoint is looking at the fairness of the features that are used. Is it fair to take someone's ethnicity into account when making job recommendations? This type of fairness is called *procedural fairness* and is for example discussed by Grgić-Hlača et al. [22].

Different fairness metrics have different relevancy in different contexts. We substantiate this by two examples that focus on different sides of fairness. In the case of a vacancy recommendation system, the vacancies are the items, the users are people looking for a job. We would want to ensure that this system does not hide a vacancy for someone because she is female, but rather makes its choices based on the qualifications of the user. Thus this system must be fair for the users. Fairness for the items is less relevant in this context. In the setting of movie recommendations, we want to give each movie producer a fair chance of being recommended. The producer is an attribute of the items, hence the system should focus on being fair for the items (producers). Being unfair to the users has fewer consequences in this case. This distinction between the side of fairness, namely user versus item side, is discussed extensively in Section 3.2.2.

This thesis explores the area of fairness. Hereby we do not only focus on recommendation systems, but also on ranking and clustering algorithms. We discuss how these different problems are related to recommendation systems, and how fairness can be defined in its context. Our goal is to provide an overview of different approaches, such that we can choose one metric to use in our implementation. This metric, which must be relevant in the context of our implementation and dataset, is used to measure the fairness of our recommendation system. We try to improve fairness through *reranking*, and investigate the balance between fairness and effectiveness.

Hence our second research question states:

RQ 2 *What approaches exist for fairness in recommendation systems, and which are relevant in which context?*

1.3.3 Implementing a fair recommendation system

Using the overviews we provide for answering questions 1 and 2, we implement our own recommendation system. The goals of this system are:

1. Combine recommendation approaches to tackle a frequently encountered problem.

2. Provide fair recommendations while maintaining effectiveness.

The system we implement is called BanditMF, it is discussed extensively in Chapter 4; it combines matrix factorization with multi-armed bandits. The effectiveness of the system is measured and discussed. We measure the fairness of the original system and try to improve it through reranking. The fair and unfair versions are compared, focusing on the balance between effectiveness and fairness.

Hereby our third and final research question is:

RQ 3 *How do fairness constraints in recommendation system influence the balance between fairness and effectiveness?*

Chapter 2

Approaches to recommendation systems

This chapter provides an overview of the most common approaches to recommendation systems. We start by describing challenges that are encountered in many RS approaches. Then we mention six approaches to building an RS. In these descriptions, we define the idea, pros and cons, and provide a discussion. The goal of this overview is to investigate which problems occur in which systems, to be able to select one problem that will be focused on for our own implementation. The choice of the approach and problem to tackle is discussed in the conclusion of this chapter.

2.1 General challenges

2.1.1 Cold start problem

The cold start problem derives its name from car engines failing to start when the weather is cold. This problem occurs in almost all recommendation systems one way or another, although some approaches are more sensitive to it than others. Cold in the context of recommendation systems means that few interactions are known for an entity. The entity can either be a user or an item.

2.1.1.1 Item cold start

Cold items are new items in the system. For such items, we have few or no interactions. As we will see later on in this chapter, many recommendation systems base their decisions on interactions between users and items. If few interactions are known, the system is unable to build a profile for the item based on those interactions. In practice, this often results in the system being reluctant to recommend the cold item at all, to any user. This behavior arises because most systems recommend items with the highest predicted rating. The predicted rating for an item of which we know almost nothing usually does not belong to one of the highest predicted ratings. Not all approaches suffer from this approach: Content-Based Filtering Systems (Section 2.2) can compute similarity based on other attributes than interactions, hence cold items can still be recommended. Knowledge-Based systems use the properties of items to recommend them, and thus do not bother about the historical interactions.

2.1.1.2 User cold start

A cold user is a user that has just entered the system. The recommendation engine has no (or few) prior interactions with the user, and thus cannot build a user profile. Many recommendation systems fall back to popularity-based filtering in such cases, this is an unfair approach discussed in the next section. While the system is gathering feedback of a cold user for the (often poor) recommendations it makes, it builds up the profile. The system gets better and better the more feedback (reward) it receives. Recommendation systems that try to tackle this problem are usually hybrid approaches: they combine several approaches, trying to counter the disadvantages of each individual approach.

2.1.2 Popularity bias

Popularity bias is found often in recommendation systems. With this term, we mean the tendency to recommend popular items to a user; it occurs most often with cold users. The reason this happens is obvious: If the system cannot find a good recommendation based on the profile of the user, it can safely choose a popular item since it has a high chance of receiving a high rating. Evaluation metrics often do not capture this behavior: the reward that is received is generally high. However, we should try to avoid this behavior, as is it is not desired in many individual cases. We list some of the problems with popularity-based ranking:

- The rich get richer, the poor get poorer. Many systems over-recommend popular items, therefor popular items will become more popular, and less popular items do not get a fair chance.
- Not every user likes popular items. For example, if we look at music recommendations: the music genre *house* is liked by many people. But if some users mainly listen to *hard rock*, we should not recommend house music to them just because the genre is popular. This seems obvious, but many recommendation systems struggle with this behavior.
- Users are driven towards the popular opinion. If we look at news article recommendations, it is not desired to push the users to a specific

opinion. Research into article recommendation shows that this can and does happen [58]; the opinion of people might be unconsciously altered by the popularity bias of recommendation systems.

- Recommending popular items does not give the system a better understanding of the user's profile. Recommending the more niche items might give much more insight into the types of users we have in the system [1].
- We can see the discovery of unpopular items as a social good. If systems only recommend popular items, the few producers of those items might achieve a monopoly [13].

In the implementation in Chapter 4, we measure fairness in terms of popularity bias. Evaluations show that without any constraints, the system has a strong tendency towards recommending popular items. The reranking procedure tries to eliminate this tendency while maintaining a high level of effectiveness.

2.1.3 Gathering feedback

For a recommendation system to improve effectiveness, it needs to receive feedback on its recommendations. Such feedback is often represented in terms of a *rating*, which indicates to what extent a user likes an item. It would be easiest to ask for *explicit feedback*: asking the user to explicitly give a rating for the item. As a practical example: Netflix sometimes asks its users to 'thumbs up' or 'thumbs down' a movie that they have seen. Webshops often ask their users to rate the items they have bought, using a 0-5 scale. Explicit ratings can take different formats [42]. In the case of the Netflix example, they are binary: users either like an item, or they do not. They can also be numerical: the rating of a bought item can have 0-5 stars. Ordinal ratings take a form like 'strongly dislike', 'neutral', or 'like'. Irrespective of the format, these explicit ratings are the most informative and unambiguous type that a system can consume.

The downside of explicit ratings is clear: they require an effort of the user, hence they are expensive and scarce. Therefore a system often relies on *implicit feedback* gathering. Implicit feedback is gathered by analyzing user behavior and deducing the user's opinion of items based on his behavior. This feedback thus requires no extra effort of the user; the user does not even need to be aware. These ratings can be gathered by tracking the browsing behavior of a user. Alternatively, purchase history can be converted into explicit ratings, for example by defining the purchase of an item as a positive rating. Gathering implicit feedback is often challenging, and the best way to do so is domain-specific; we give some examples. When deducing whether users like a movie, we could track the number of times the movie is watched.

If a user watches a movie until the end, we can conclude a positive rating for that movie. If the movie is stopped early, this is a sign of dislike. Moreover, consider another example, in the context of webshops. Suppose a webshop has a list of items that is shown to the user on an outlet webpage. If the user clicks on an item, this can be defined as somewhat positive feedback. If the user actually buys it, this would be the most positive feedback we can get. Moreover, in some domains, the item should be excluded from the recommendation system when it has been sold to the user. This might be simply because it is out of stock, but it might also be because we assume the user does not buy the item again in the (near) future. If the user simply scrolls past the item, this would either be no feedback, or negative feedback.

Most systems use a hybrid approach [25]: Combining explicit and implicit feedback. Explicit feedback is most valuable, but gathering much of it is too expensive. Hence one would also gather (different types of) implicit feedback, and convert all types to the same scale, for example a 0-5 rating. The ratings are converted into a *rating matrix*: a matrix with one row for each user, and one column for each item; the values are the deduced ratings. We elaborate upon this representation in the description of matrix factorization Section 4.1.1.

The challenges we discussed in this section are encountered in many (approaches for) recommendation systems. They will reoccur in the rest of this chapter; in the conclusion of the chapter we select one of them; the implementation described in Chapter 4 focuses on tackling this challenge.

2.2 Content-Based Filtering Systems

The first of six approaches for recommending items to users is called Content-Based Filtering (Figure 2.1). Content-Based Filtering Systems (CBFSs) select items based on the correlation between content and users [54]. Thus the filtering is based on the content of items and users. The content of an item can be described in several ways. In the example of webshops, content can be the attributes of a product, for example, retail price or color. In the case where research papers are recommended based on a search query, documents could be represented by their bag of words[1]. Regardless of the representation, CBFSs compute the similarity between items. When it needs to predict a recommendation for a user, it searches for items that are similar to the items that the user likes. Which items the user likes can be deduced either directly by provided ratings, or by comparing the user model with the item model. In the latter case, a CBFS thus also uses user-item similarity. We discuss several specific methods of CBFSs.

[1] `https://en.wikipedia.org/wiki/Bag-of-words_model`

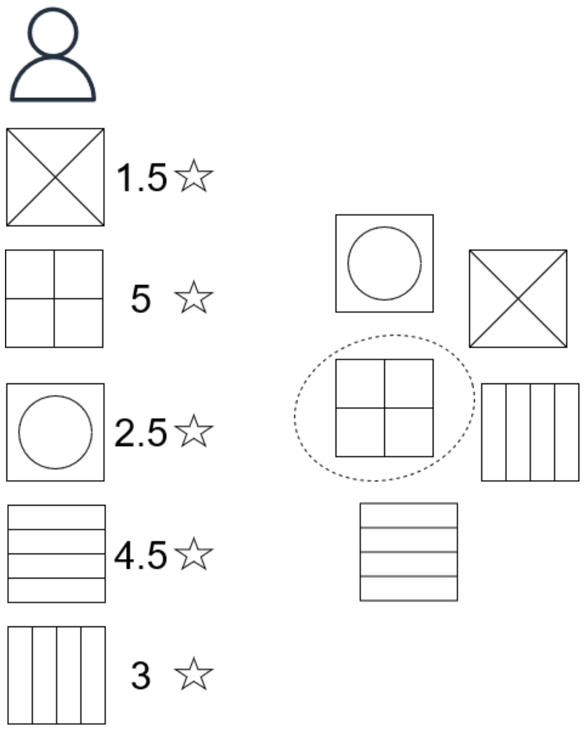

Figure 2.1: Content-Based Filtering. Items are recommended based on their similarity: the system recommends items that are similar to previously liked items.

2.2.1 Vector representation: TF-IDF

We can represent items and users using vectors: the vector-space model. One of the main advantages is an easy comparison of different entities: we can simply use a similarity measure like cosine similarity to compare vectors. The higher the similarity (or lower the distance), the more alike the two entities are. One can then recommend items that are similar to the user or items that the user likes. The users are oftentimes represented by a weighted combination of the items that they have rated, where the rating is used as a weight. This way of *user profiling* is used more often; we discuss it in Section 2.2.2. In this representation, the items can be compared to the user, while hereby thus comparing indirectly to all the items that the user has rated.

TF-IDF is one of the most used vector-representation in systems where the items are textual documents [10]. With this method, we represent each document by a vector, where each dimension represents a unique term in the whole collection. The value of that dimension for a document is its TF-IDF value. TF-IDF stands for *Term Frequency - Inverse Document Frequency.*

The term-frequency of a term t and a document d is given by Equation 2.1.

$$TF(t,d) = \frac{f_{t,d}}{\sum\limits_{t' \in d} f_{t',d}} \tag{2.1}$$

Here, $f_{t,d}$ is the frequency of term t in document d. Thus the term frequency is the frequency of a term relative to the total number of terms in the document. The inverse document frequency of a term is given by Equation 2.2.

$$IDF(t) = \log\left(\frac{|D|}{|\{d \in D | t \in d\}|}\right) \tag{2.2}$$

Here, D is the set of all documents. Thus the inverse document frequency is the log of the total number of documents divided by the number of documents that contain the term. The TF-IDF value is now simply the multiplication of these two values. The idea is that its value is higher for more important words. A term is important in a document if it occurs often in that document, but not in many other documents. We can now represent each document (item) in the corpus (collection) by a vector of TF-IDF values, one value for each unique term in the collection. The user profiles are represented by vectors as well, these are the weighted vectors of the items the user has rated. The cosine similarity indicates the predicted rating: the higher the similarity between the vectors of a user and an item, the higher the predicted rating.

2.2.2 CBFS using optimization techniques

Elngar et al. discuss several approaches to CBFSs [16], one of which is optimization using for example stochastic gradient descent. In this approach, users and items are represented in the same attribute space. For example, a movie matches certain categories up to a certain level, e.g. action or drama. Users like these same categories up to some extent, and are thus represented in the same dimensions. If we know to what extent a movie is an action movie, and to which extent a user likes action movies, we can approximate the *rating* a user would give to a movie. Movies with a high expected rating would then be recommended to users.

We can represent a user j as a vector θ^j, where the values are j's preferences for movie categories. I is the set of items; vector X^i represents movie i in the same dimensions: to what extent the movie is about a certain category. We assume we have a sparse *rating matrix*, which contains known ratings of users for movies. $Y^{i,j}$ is the value of that matrix for user j and item i. $r(i,j)$ is a function that returns 1 if user j has rated item i, and 0 otherwise; b^j is the number of items rated by user j. We can now learn the representation of θ^j by optimizing θ as in Equation 2.3.

$$argmin(\theta)\frac{1}{b^j}\sum_{i:r(i,j)=1}^{I}(\theta^j X^i - Y^{i,j}) + \lambda R(\theta^j) \tag{2.3}$$

Here, λ is a regularization parameter, and R is a regularization function. We can minimize this function for each user by using gradient descent, resulting in a representation for each user in the given dimensions. The sparse *rating matrix* can now be filled with predicted ratings, by multiplying θ with X.

2.2.3 CBFS Using decision trees

Elngar et al. [16] mention another way of approaching content-based filtering, based on decision trees. Indeed, decision trees have been used in the context of recommendation systems [33, 21].

A decision tree starts at a root node, which has branches for each answer to a specific question. The challenge in training a decision tree is to find the most ‘separative’ questions on each node, such that one reaches a leaf as soon as possible when answering the questions for a certain input. After the tree is trained, its leaves contain values for the target attribute. When we want to know the predicted value of this attribute for a new item, we traverse the tree from root to leaf. At each node, we answer the question for the item, until we reach a node. The node tells us the predicted value for the target attribute.

Li et al. [33] describe a process to use a decision tree as a content-based recommendation system:

1. Input of training data. The training data consists of two parts. Firstly, we need ratings of users for items. These ratings are the target attribute, which need to be known for the training set. Secondly, we need attributes to base our decisions on at each node. Thus we need attributes and corresponding values for each item in the data set.

2. Construct a decision tree. Using the training data, we select an attribute at each node. Traversing the tree is done by taking the edge corresponding to the value of the item for the attribute in question. We train a separate tree for each user. Here we will encounter the user cold-start problem, as discussed in Section 2.1.1.2.

3. Classification of unvalued items. For items that have not yet been rated by a user, we can traverse the tree of the user. The leaf node tells us the predicted rating of the user for the item.

4. Building a candidate list. We sort the unvalued items decreasing by the predicted rating.

5. Result presentation. The top n items of the list are presented as recommendations to the user.

6. Training data addition. We use implicit and/or explicit feedback to update the training set. After the set has been updated, we recompute the decision tree. Creating such a tree is lightweight enough to be done at each step [33].

The main shortcoming of this approach is that one needs to train a large number of trees: one tree for each user (or for each item, in case user attributes are used). Another problem is that, when generating a list of recommendations for a user, the tree has to be traversed once for each item in the list. Approaches have been developed to overcome this latter issue, for example by Gershman et al. [21]: they use lists of items as leaves instead of single items.

2.2.4 Association rule mining

Association rule mining is a method that can extend CBFSs. A CBFS relies on finding correlations between items; association rules can help to find relations. We explain the concept on the basis of an example. The domain is a webshop in tech products. If a user adds a tablet to his or her shopping cart, it is a good idea to recommend a case for the tablet. Finding the relation between tablets and cases might be hard using pure content-based filtering. The reason for this is that they are not the same product type (namely tablets versus cases), not the same price range (expensive versus cheap), and are used for different purposes (browsing the internet / protecting a device). Association rules can be mined by investigating the training data. In the data we can find that some customers bought a case with a tablet, so we might extract the rule "If someone buys a product of type *tablet* of brand X, then he/she might be interested in buying a product of type *case* of brand X". Association rules usually take the format of if/then [47], and are thus used to find relations between products that might not be found using classical approaches.

As discussed in Section 2.3, these types of relations might also be found using Collaborative Filtering. However, in most cases, those relations are more general. In our example, this might result in recognizing that "If someone buys a product of type *tablet*, then he/she might be interested in buying a product of type *case*". This rule is more general and might result in the system recommending a case that is not suited for the specific tablet. Moreover, CFSs do not explicitly provide these rules, so they are less interpretable.

We note that an association rule might still be too general. In the example of the first rule: we know that some brands have different models, and cases are often suited for only one or a few of the models. *Constrained-based* approaches are therefore a good extension of this approach, which we describe in Section 2.4: Knowledge-Based Systems. We also note that association rule mining can be seen as a kind of knowledge-based approach, as the rules are often a form of domain knowledge. If rules are used to find relations between users, they can also be used in the context of Collaborative Filtering. Hence this approach does not fall strictly under the scope of content-based systems, but can rather be extended to other approaches.

2.2.5 Discussion

Having discussed different approaches to content-based filtering, we now state the pros and cons; they are summarized in Table 2.1. One of the advantages of the CBFS approach is that users do not need each other [25]. This means that when we have a sufficiently detailed description of a user, and we have representations of the items in our system, we can accurately recommend items to that user. This is not the case in every approach, since other approaches use the similarity between users to come up with recommendations. The second advantage of CBFSs over other systems is that they do not suffer from the item cold-start problem. Since items must be described by their attributes, we can recommend them directly to any known user, without the need for prior interactions. Because of this attribute dependence, CBFSs are also quite explainable. The decisions are based mainly on the attributes of the items, so we can understand why an item would be recommended to a user. Another pro for this method is that we are able to incorporate domain-specific knowledge by design. For different domains, different types of item attributes are relevant. This approach is completely built around those attributes, thus a domain expert can easily select the most relevant attributes for the recommendation system, or can extend the system with attributes that are deemed relevant.

Interestingly, the last advantage can also be a disadvantage: we always need attributes of items for a CBFS to work. Other approaches are able to find relations between entities without the need for attributes, for example by user behavior, context, interactions between users and items. A system like this can only be used if relevant attributes are known about the items. Secondly, the risk of overspecialization is eminent [25]. When the profile of a user is becoming more and more detailed (because many items are rated or seen), it becomes difficult to recommend novel items to him. Recommendations are selected based on what was liked in the past, thus the user will find similar items in the future. An example of this pitfall in practice is given by Ledwich and Zaitsev [31]; they discuss *"Youtube's rabbit hole of radicalization"*. Lastly, the user cold-start problem is very prominent here, as a CBFS does not use context or prior behavior. As long as the number of ratings of a user is too low it cannot make (accurate) predictions.

Pros	Cons
Users do not need each other	Domain-specific knowledge required
No item cold-start	Overspecialization (low level of novelty)
Domain-specific knowledge incorporated by design	User cold-start
Explainable	

Table 2.1: Pros and cons of a CBFS.

2.3 Collaborative Filtering Systems

While CBFSs recommend items based on similarities to previously liked items, Collaborative Filtering Systems (CFSs) focus on similarity between users (Figure 2.2). Users collaboratively filter the items they like, because the system computes similarities between users. The basic idea was first introduced by Maes et al. [46], it is as follows: Suppose a recommendation engine must provide a recommendation for user u, of which we know some historical ratings h. Many other users exist in the system, all having their own history. The system computes the similarity of u with the other users and selects the most similar user u', having history h'. It now searches in h', and selects those items that are possible candidates for recommendation to u. It can now simply select the item i with the highest rating of u' to predict to u. The reason why this should work: u' has a similar taste to u, and u' likes i, thus u will probably also like i. Instead of comparing with the most similar user, CFSs in practice select the top n similar users, predict averaged ratings weighted by similarity, and then select the item with the highest predicted rating.

At the core of a CFS is a similarity measure for users. Most systems represent users in terms of vectors of length equal to the total number of items, where the values are the ratings. A vector distance measurement can then be used to measure similarity. More extensive approaches are also available, like comparing user browsing behavior [47], or by building personas using known attributes like age and gender.

A distinction is often made between *memory-based* and *model-based* collaborative filtering systems.

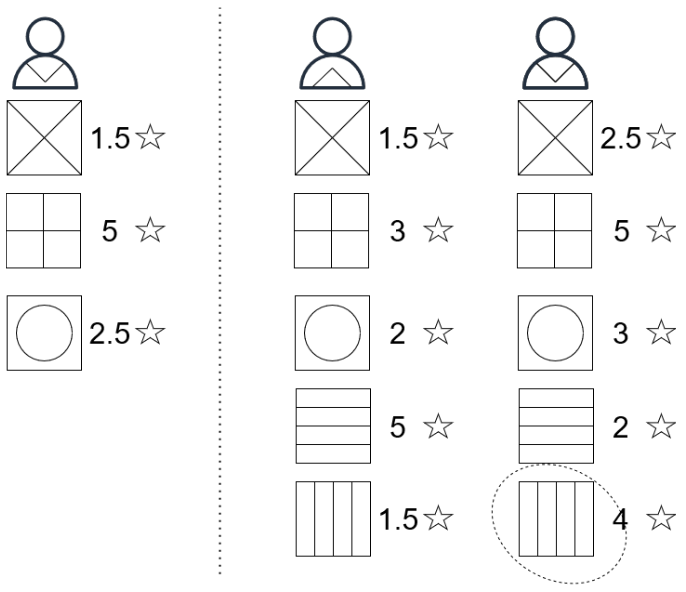

Figure 2.2: Collaborative Filtering. Items are recommended based on similarity between users: the system recommends items that are liked by similar users.

2.3.1 Memory-based Collaborative Filtering

Memory-based systems are the least complex; their main components are similarity computation and prediction [49]. These systems do not train a machine-learning model to learn the representation of any users or items, but simply define them in terms of their historical ratings. User-based systems are those we described before: they find similar users and use their preferences for a recommendation. A major drawback of these is that the amount of users is often significant, which might result in performance and memory issues. Item-based systems start by looking at items: They find items similar to the items that we know the user likes, and recommend those items. The major drawback of this approach is the lack of diversity, as the system only recommends items that are similar to previously liked items.

In both item- and user-based memory-based approaches, the system needs to compute similarity to many other entities. If we would also like to learn the system online, e.g. learn from the feedback it receives, the system must recompute similarity often. This is a computationally expensive task since the domain could contain millions of users and items. Another challenge is that the rating matrix that is used is very sparse, which could result in memory problems.

Several solutions to these problems have been proposed [44]. Before computing similarity, the system could apply sub-sampling, to pre-select only a subset of users with a high probability of being similar. Another option is to cluster users and/or items, to reduce the dimensionality of the rating matrix: an approach we also use in the implementation in Chapter 4. Dimensionality reduction could also be realized via *Principle Component Analysis (PCA).* The entities are then represented in some *latent space*, which is of a much smaller dimension. The features in that dimension could intuitively be seen as the preferences of users for certain topics [44], or some multi-label categories of items. The similarities can then be computed over these latent vectors, which is computationally cheaper because it has much smaller dimensions. A final approach we mention is *community-based recommendations.* This approach is especially effective in the domain of social networks [43]. In such systems, users are not compared by vector similarity, but by creating communities. Communities could be defined by friends in a social network, but also by other attributes of users that we think might be useful for clustering. Examples of those attributes are the posts that people like or click, the subjects that people write about, or the geographical region of someone. As we will see later, in Section 2.4, this idea has similarities to the DRS approach, but they differ in the attributes on which the clusters are based.

2.3.2 Model-based Collaborative Filtering

Model-based CFSs build a model of users and/or items in some latent space. These models are trained using some historical data and updated while receiving more feedback from the system. Where memory-based systems use these representations to directly compute similarity, model-based approaches use a model to find interesting items. The model tries to predict which items are interesting for a user; the feedback of the user is used to refine the user representation. One of the advantages over memory-based approaches is that these do not need to store the complete rating matrix into memory. Once the model is trained, the original rating data is often not needed anymore [44], and the model can be incrementally updated while receiving feedback.

Many models exist for Collaborative Filtering, one of which is a Naive Bayes classification approach, which tries to predict the probability of each class given a user: $P(c|u)$. Here, the classes are defined by the list of all possible ratings [44]. Shani et al. [45] propose to view the recommendation problem as a Markov Decision Process. They state that after each recommendation, a new list of recommendations (predictions) must be computed. This process is a sequential one; and can be described as an MDP.

One of the most well-known model-based CFS approaches is Matrix Factorization. This approach consumes the rating matrix and uses mathematical properties of matrices to reduce its dimensions. We use this approach in our own implementation and discuss it in Section 4.1.1.

Pros	**Cons**
Domain-independent	Cold-start problem
High level of novelty	Data sparsity
	Low scalability

Table 2.2: Pros and cons of a CFS.

2.3.3 Discussion

An overview of the advantages and disadvantages of the collaborative filtering approach is provided in Table 2.2. The main advantage of collaborative filtering is that it only needs a list of ratings as input. Hence a CFS can be applied to many different domains, without alteration of the model. Another advantage is that this approach is capable to recommend new types of items to users [25]. If an item is very different from items that a user has previously seen, the system can still deduce that the user might like this type of item, if other users have shown to like it. Thus the user might be ‘surprised’ by certain recommendations that the system makes. These might be recommendations that the user would not have thought of, but can still be rather interesting. This increases the *novelty* of the system.

CFSs also have some disadvantages, the most prominent one being the cold-start problem. The system utilizes no other information about users or items except the rating history. If that history is absent for either a user or an item, e.g. that entity is cold, the system has no information to base its decision on. Decisions for cold users often appear random or are popularity-based. Approaches that tackle this problem usually combine the CFS and CBFS approaches. Another challenge in CFSs: data sparsity. The rating matrix is in practice very sparse: each user sees and rates only a very small fraction of all items. The sparsity in such large matrices often results in memory and computational issues. Besides, the similarity computation of sparse vectors is often not very precise; the distances between items in a high-dimensional space are always large. As the last challenge, we mention scalability [25]. While a CFS might work well when deployed, its complexity grows with the number of users and items. Thus for large-scale domains, one might run into problems when using the model in production, when the number of items and/or users increases.

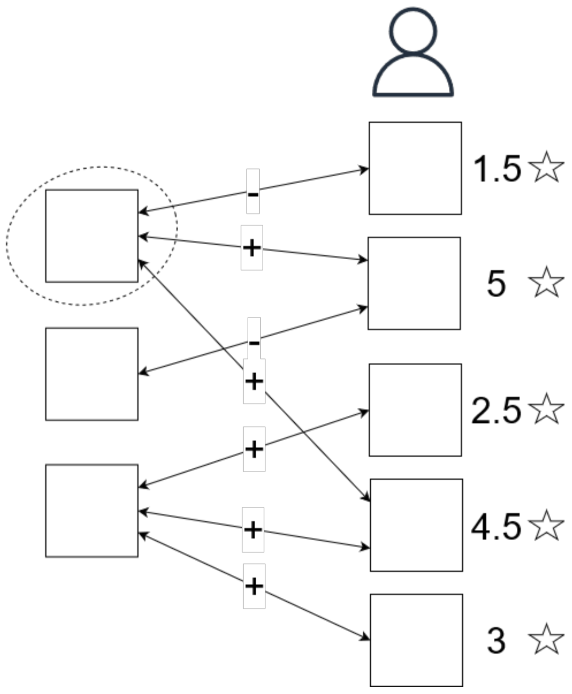

Figure 2.3: Knowledge-Based Systems. Items are recommended based on the relationships between attributes: the system recommends items that are (through domain-knowledge) positively related to items that it knows the users likes.

2.4 Knowledge-Based Systems

As the name suggests, Knowledge-Based Systems (KBSs) use domain knowledge to base their recommendations on (Figure 2.3). Knowledge in this context means that the system knows the attributes and properties of users and items, and knows their meaning and relations. As a specific example, consider a webshop that sells winter sports products like skis, snowboards, clothing, and gear. A knowledge-based system would use properties of products to connect them. For example, it knows that a product of the type *snowboard boot* would be interesting for people who are looking for a snowboard or have already bought one. This knowledge could be used in a 'You might also like' popup whenever a user adds a snowboard to the shopping cart. KBSs often retrieve the users' needs via some kind of dialogue [17], e.g. it asks the user-specific questions to filter out items that might be relevant.

Burke [11] defines three types of knowledge that can be used in these types of systems.

- *Catalog knowledge* is the knowledge about the catalog of items. In the example of the winter sports webshop, this would for example be the category of a product (snowboard), the gender it is suited for, or its dimensions.

- *Functional knowledge* is about the relation between a user's need and an item's attribute. In our example, our system would use the functional knowledge that someone who buys a snowboard might also be interested in a snowboard boot.
- *User knowledge* is used to build a model of the users. This knowledge can be the properties of people, for example, the neighborhoods they live in, or their gender. Another type of user knowledge is the search query a user enters. For example, the webshop could ask the user to tick one of two boxes: "I am a skier" or "I am a snowboarder". With this knowledge, the system would show either products related to skiing, or products related to snowboarding.

A major distinction in KBSs is case-based versus constraint-based systems. Both approaches are similar in the sense that they need to deduce the users' needs based on some input and find items that match the requirements [17]. Case-based reasoning looks at recommendation as a similarity-based problem. Such systems find items that are similar to what a user enters into the system, the data being the user's own attributes or for example a search query. How the similarity of input and items is computed, is domain-specific. Constrained-based approaches use explicit constraints that are entered by the user. This information is often entered in terms of filters that the user provides, for example, "Snowboards of length $>$ 150 cm and $<$ 165 cm". With these constraints, the system filters the items that (mostly) fit the user's needs.

2.4.1 Utility-based approach

Utility-based recommendation systems build a user model by computing a utility function for each user and use the model to compute predicted ratings via the utility function [61]. A utility function computes the utility of items, given the values for some attributes. As an example, consider Airbnb. Someone who is looking for an apartment for a holiday will have certain preferences. Some preferences are precise, like the city in which the apartment must reside. Some preferences are less restrictive, like a price range or the number of rooms. A utility function takes these attributes and can compute the preference of the user for a combination of those attributes. A utility-based approach tries to compute such a function for each user and then orders all items based on the utility. These approaches fall under the category of Knowledge-Based Systems, as the utility function is based on domain-specific attributes. Besides these attributes, historical rating data is needed, as these show the preference for users for different combinations of attributes.

Pros	Cons
No cold-start problem	Feature engineering required
Highly adaptive	User effort required
	Low level of novelty

Table 2.3: Pros and cons of a KBS.

2.4.2 Discussion

The pros and cons of this approach are shown in Table 2.3. Compared to the other approaches, the major advantage of the KBS approach is the absence of the cold start problem. The system does not need a history of user-item interactions to compute recommendations. If a new item enters the system, the recommendation engine can directly recommend it to users that are looking for products with such attributes. New users can also simply enter a query or provide information about their needs, and the system should be able to recommend interesting items. This is often the main reason that this approach would be chosen over other options. Rating history is often sparse in domains where products are only bought once, and where users only interact with a few items. Examples are technology like TVs and computers, furniture, and winter sports products. In domains where users 'consume' and interact with many products, for example in the case of music streaming services, an RS often has access to an extensive set of interactions. Such data would not be used in a knowledge-based approach, thus these systems are less interesting in those cases.

Another advantage of KBSs is that they can adapt to changing needs [11]. To explain this, we again look at the example of a winter sports shop. A user might at some point change from being a skier to being a snowboarder. If the user would have yet interacted often with a CBFS or CFS, the system would have a hard time adapting to the changing needs. It would take a lot more time and interactions before the system would recognize that the user is now looking for snowboard gear. In a KBS, the user simply provides new answers to the questions the system asks, and the system immediately knows to recommend snowboards instead of skis.

The reason why this approach would not work in many systems is that it requires knowledge of attributes and their relations for items and users. If this knowledge is not known by design, it requires feature engineering to retrieve it, which might take a lot of effort. Thus KBSs are a good choice in systems where these attributes and their relations are readily available. In the example of the winter sports webshop, the system might already know that the attribute sets 'snowboard' and 'snowboard-boot' are related, hence this knowledge can be used directly. In other domains, these features

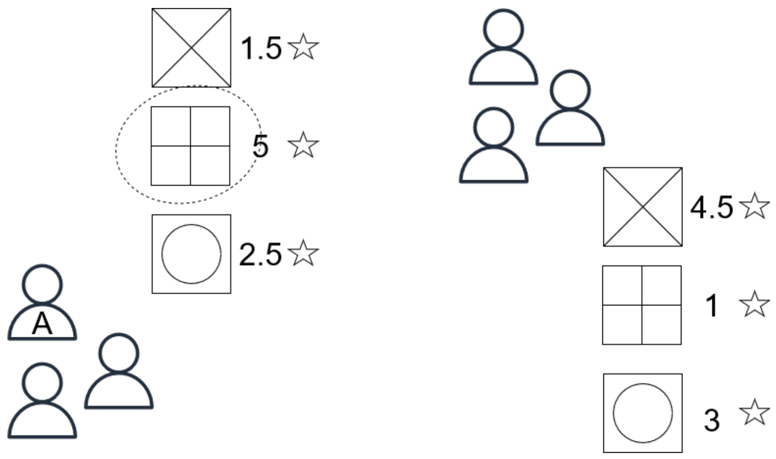

Figure 2.4: Demographic Systems. Items are recommended based on demographic properties of a user: the system recommends items that are liked by other users in the same demographic cluster.

need to be explicitly engineered; an example is a search engine for blog posts. The only attributes of the items are the TF-IDF values in the BOW setting. Hence relations between blog posts need to be manually engineered and added as attributes, for example by correlating the subjects they are discussing or the authors that wrote them. Substantive relations between items are not readily available; this is often the case when the items consist of natural text. Defining those connections is often cumbersome, hence one must consider whether a KBS approach is the right choice in such domains.

The need for features might also be tiresome for users, as the system requires the user to provide information about his needs. A CFS would deduce the user's profile by his behavior, hereby not bothering the user with questions about his needs.

As with CBFS, recommendations of a KBS might be too static. The system will recommend items based on the needs of a user, and it will thus not provide novel or new items to the user.

2.5 Demographic Systems

A Demographic Recommendation System (DRS) uses the demographic profiles of the users to base its decisions on (Figure 2.4). The assumption in these systems is that different demographic clusters should be provided with different recommendations, while users in the same cluster should receive similar recommendations [42]. These clusters are found with attributes like locality, gender, sex, education, or age. In some domains, gathering such information is easier than in others. In many cases, it is hard to define the influence of different information types on the demographic cluster of a user

Pros	Cons
Simplicity of data	Very general recommendations
Explainable	Rating data required
No domain knowledge required	Item cold-start
	Risk of unfairness

Table 2.4: Pros and cons of a DRS.

[38]; oftentimes domain knowledge is required for this decision. A DRS uses clustering based on these demographic attributes to create clusters of users that should have similar interests. Predicted ratings are then deduced as in CFSs: by looking at the known ratings for the item of similar users; and items with high predicted ratings are recommended.

2.5.1 Discussion

The advantages and disadvantages of this approach are shown in Table 2.4. The main advantage of this approach is that it is simple. The amount of data that it requires is limited, and in many cases easy to gather. Its outcomes are also very explainable: Items are recommended because other users like those items, and those users are alike the users through some demographic properties. As a last advantage: the DRS is very general and requires no domain knowledge.

At the same time this is a downside of this approach: it is often too general. A DRS clusters users based on some demographic attributes, but these attributes might not be sufficient to cluster users with similar preferences. It may very well be that different users in one cluster have very different interests; a DRS cannot deal with this. This may happen if too few attributes are taken into account, but also if the attributes that are used are not of importance when defining interests.

Although the input of a DRS is simple, it still needs the history of users to define which items they like. After similarity is computed, predictions are based on the ratings of similar users. Thus we encounter the same problem as with Collaborative Filtering systems: a lot of data is needed. This also means that it is hard to recommend cold items. Cold items do not occur in the history of any user. Hence the users that are similar to u have not recommended that item, hence that item will not be recommended to u.

We will discuss fairness more extensively in Chapter 3, but approaches like these are sensitive to being unfair. Demographic attributes can be quite personal, and one must carefully consider whether such attributes must be used to base recommendations on.

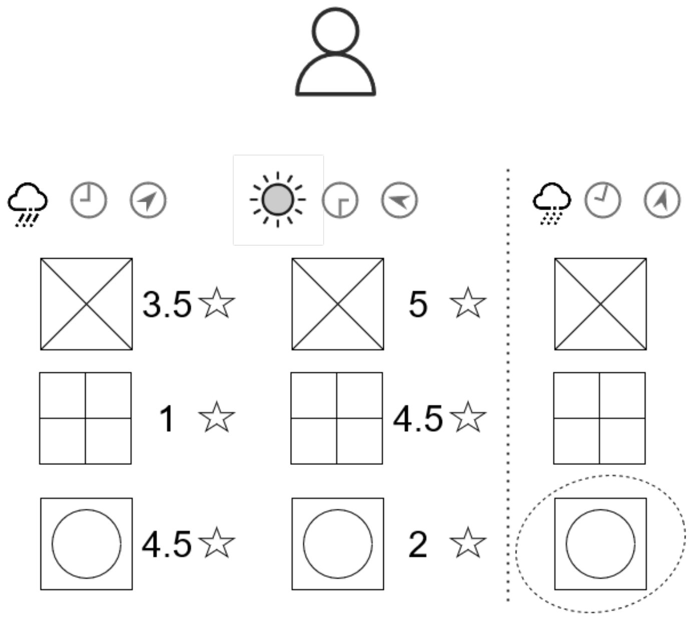

Figure 2.5: Context-Aware Recommendation Systems. Items are recommended based on contextual (dynamic) attributes: the system recommends items that are liked in similar contexts.

2.6 Context-Aware Systems

A Context-Aware Recommendation System (CARS) uses the context of a user to recommend items (Figure 2.5). The difference between user context and a demographic property is that the latter is static, while context often changes over time [49]. Examples of contextual attributes are location, time, and weather; they are all highly changing attributes. Especially the location is often used for mobile devices, which makes a context-aware approach very useful in this setting. Examples of the added value of contextual attributes are recommendation systems that do not recommend sunscreen when it's snowing, and ones that know that scarves would not be sold during summer.

Contextual information can generally be added to a system in one of three ways [50]:

- *Context post-filtering* is the most straightforward method. Here, context is used to filter out items that can never be relevant given the current context. An example is someone who is looking for rental bikes. Such a person is often on foot, and will only be interested in shops that rent bikes within a small range. Any shops that fall outside this perimeter can be filtered out using contextual post-filtering, using location as a contextual attribute. In these cases, we do not filter out any input data but exclude irrelevant items from the final ranking.

Pros	**Cons**
Increase of effectiveness	Retrieval of contextual attributes needed
Easily added to existing systems (pre- and post-filtering)	Increase of complexity (contextual modeling)
Irrelevant data is dropped (pre-filtering)	Domain-specific knowledge required

Table 2.5: Pros and cons of a CARS.

- *Context pre-filtering* filters out input data that is not relevant for the current recommendation. Reconsider our earlier example of someone who has changed from being a skier to being a snowboarder. After the change, this person will be interested in snowboard-related items. The context we use is time, and we know that earlier in time the user was a skier. If we find out that after some point the user prefers snowboarding, we can filter out all ratings before that moment in time. This gives us a more accurate profile of the *current preferences* of the user, and hence improves the predictions. Pre-filtering thus filters out input data instead of output data.

- *Contextual modeling* is a more intrusive approach as opposed to the other two. The first two approaches use an existing recommendation system and alter input or output. Contextual modeling adds an extra dimension to the input data, hence the input matrix now has dimensions #users $\times$ #items $\times$ #context [5, 4]. This means that the recommendation engine must be altered to support the new input type. Much research has been put into this, including contextual matrix (tensor) factorization techniques [28] and contextual multi-armed bandits [32].

2.6.1 Discussion

A context-aware approach is an approach that is added on top of an existing recommendation system; the pros and cons of such an extension are described in Table 2.5. The main advantage is obvious: relevant contextual attributes can increase effectiveness. Especially when users show changing preferences, a CARS can adapt immediately. Another advantage, which holds only in the case of pre- and post-filtering, is that context can easily be added to an existing system. Hence we can increase the effectiveness of a system with relatively low effort, given we have access to relevant contextual attributes. Pre-filtering also drops irrelevant input data, which can speed

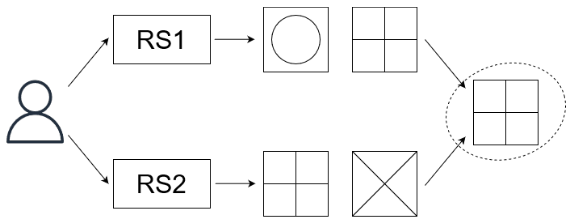

Figure 2.6: Hybrid Recommendation Systems. Items are recommended based on the outcomes of more than 1 system: the system recommends items based on the joint recommendations of more than 1 system.

up decision-making.

This brings us to the challenge in a CARS: retrieving contextual attributes. In some domains, these are easier to retrieve than others, but in any case, they need to be available. Another challenge is the increase in complexity. Especially when using contextual modeling, the dimensions of the data can increase drastically. It might also be difficult to determine which context is relevant for deciding which item to recommend. Using irrelevant attributes needlessly adds complexity. In many cases, the context is domain-specific, thus using it requires specific knowledge.

2.7 Hybrid Systems

Hybrid Recommendation Systems (HRSs) combine two or more approaches into one system (Figure 2.6), aiming to counter the disadvantages of the individual approaches. Combining more than one system can be done in many different ways, Burke describes seven [11] approaches. The figures 2.7 - 2.13 are used as a visual explanation, and show the flow in the case of two recommendation systems; note that these systems can be generalized to more than two.

- *Weighted* hybrid approaches (Figure 2.7) combine the results of several systems into a single rating. The combination is done through some weighting scheme, such that some systems can have more influence on the final result than others. Things become more complex if we want to adjust the weights for several users or types. But in general, this is a straightforward way to combine systems.

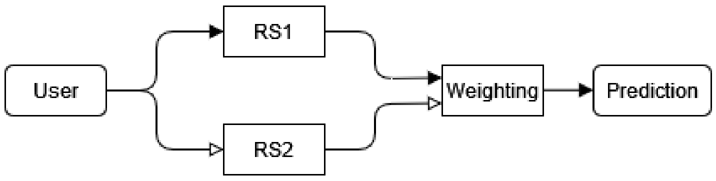

Figure 2.7: Weighted recommendation: both RS1 and RS2 compute recommendations, both outputs are merged through a weighting algorithm

- *Switched* hybridization (Figure 2.8) chooses one system out of several options upon each recommendation request. The choice can be based on any information, for example, user properties or context. The advantage is that a suitable system is used each time; the challenge is to find a good decision function.

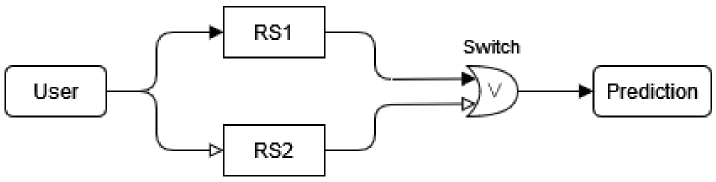

Figure 2.8: Switched recommendation: a switching algorithm decides in each individual case whether to use RS1 or RS2.

- When several recommendations have to be made at once, one can use *mixed* hybridization (Figure 2.9). Such systems compute recommendations via several approaches and recommend items from all systems. Hence the final recommendation list is based on several approaches, which might give a more widespread view. The downside is that the outputs of several systems might seem somewhat contradictory in some cases.

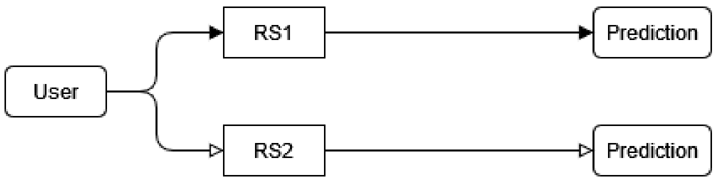

Figure 2.9: Mixed recommendation: recommendations of both RS1 and RS2 are shown.

- *Feature-combined* hybridization (Figure 2.10) uses the features of one approach as input for another approach. Both the original input and the output features of the first system are used as input to the second system. For example, we could use the information of a collaborative filtering system as a content attribute in a content-based filtering system. This approach thus still uses one system as the main RS but uses other RSs as 'feature extractors' to add extra features.

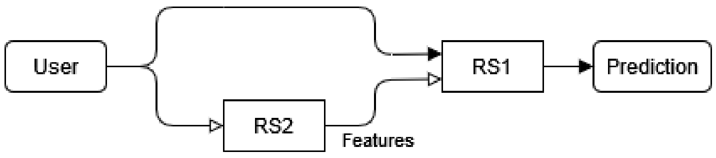

Figure 2.10: Feature-combined recommendation: the output features of RS2 as well as the original input features are used as input to RS1.

- *Feature-augmented* hybridization (Figure 2.11) is, like feature combination, also a sequential process. In this case, a first system 'augments' the data, and the updated data is served as input to the base system. The base system does not need to be altered: it receives the same input format. An example is that a content-based filtering system could produce predicted ratings for some user-item combinations. These ratings could be merged with the original rating data, and the combination is used as input to a CFS. The advantage of this approach is that it can easily be added to an existing system: it improves (augments) the input data, but does not alter the original RS.

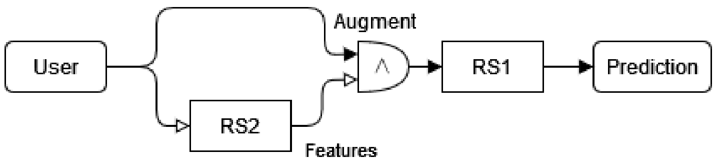

Figure 2.11: Feature-augmented recommendation: the features of RS2 are used to augment the original data; the result is used as input to RS1.

- A *cascaded* hybrid system (Figure 2.12) uses a secondary system to 'break ties'. The main recommendation system creates a ranking, and a secondary system is used to choose between items that retrieve a similar score by the original system. This way the second approach is only used when needed and effective, and not in cases where it performs badly. An example is that a CFS should not be used with few known ratings and many known attributes, but for users with few known attributes but a long history, a CFS as a secondary system might prove to be effective more effective than the primary CBFS. As we will see in Chapter 4, we use this approach in our implementation.

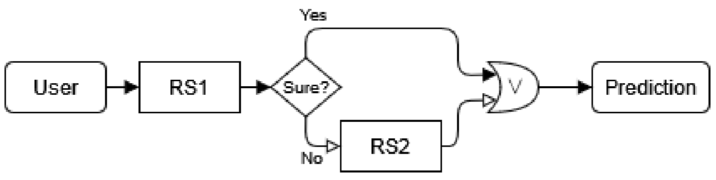

Figure 2.12: Cascaded recommendation: RS1 is used by default; RS2 jumps in when uncertainties arise.

- In *meta* hybridization (Figure 2.13), a complete system is used as input for another system. In contrast to feature combination, where the output of one system is used as input for another, we now use a whole model as input to the main system. The main advantage is that a model usually represents the data in a more compact and dense way, hence the original model does not have to work with sparse input data. The drawback is complexity and the fact that the RS must be altered heavily, such that it can consume a model instead of raw input.

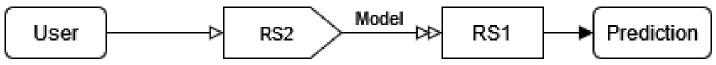

Figure 2.13: Meta recommendation: RS2 itself is used as input to RS1.

Sharma and Singh [47] call the combination of more than 1 recommendation system into a single system *monolithic hybridization.* This way of combining systems is one of three possibilities:

- We speak of *monolithic hybridization* when one system makes use of several strategies. For example, this could be a recommendation system that is based on a user-item matrix but also takes context into account by using *tensor factorization* [28].
- A *parallelized hybrid system* uses several approaches via different systems and combines their output for a final recommendation. For example rankings of two recommendation systems could be combined into one ranking via a voting system like Borda Count[2].
- A *pipelined hybrid system* uses consecutive systems one after another. The output of one system serves as the input to the following, eventually resulting in a recommendation list based on several approaches.

2.7.1 Discussion

We have seen that many different ways exist to combine recommendation systems; the pros and cons of using hybridization are listed in Table 2.6. In general, the reason to combine systems is to increase effectiveness. If enough data is available to use more than one approach, actually utilizing that data usually increases the effectiveness of the system. However, one tradeoff has to be made: does the increase in effectiveness outweigh the increase in complexity? Using more approaches and more data makes the system more complex, thus it must have added value. If a system should be generalizable to more domains, one might also decide not to use several approaches: A purely collaborative filtering approach generalizes easily to other domains, while adding content-based decisions decreases this generalization performance.

[2]`https://en.wikipedia.org/wiki/Borda_count`

Pros	Cons
Increase of effectiveness	Increase of complexity
Utilize all available data	Decrease of generalizability
Counter disadvantages of individual approaches	

Table 2.6: Pros and cons of a HRS.

2.8 Conclusion & Discussion

In this chapter, we have discussed six of the most well-known approaches to recommendation systems. The first two are most used: Content-Based Filtering and Collaborative Filtering. The first one recommends items that look like the items which it knows are liked by the user, using a similarity measure on the attributes of the items. This approach is suitable in situations where attributes of items are known. Such an approach is explainable and does not suffer from item cold-start. In the latter approach, the users collaboratively filter recommendations: the system recommends items that are liked by users that have a similar interest. If an extensive history of interactions is known, one would choose this approach. Studies have shown that it works well in practice, although it struggles with new users and items. One advantage over most other systems: the novelty of items is quite high, as it does not solely recommend items that are similar to previously liked items.

Knowledge-Based Systems are, like CBFSs, systems that use attributes of items. The difference is that a KBS also uses user attributes, and is aware of the *meaning* of the attributes. Hence such a system is domain-specific. If extensive knowledge of attributes and their relations is known, this approach should work very well. The main reason to not use such an approach would be that it requires user effort, as in most cases the user must explicitly define his requirements for the items, which would not be needed in for example a CFS.

Demographic Systems group users based on demographic properties. The idea is that users in the same group have similar interests, and should thus be recommended the same items. This approach is simple to implement and is often highly explainable. However relevant demographic attributes must be gathered. DRSs are often used in combination with other approaches, as demographic attributes in most cases do not provide enough information to separate different types of users.

Context-awareness is an extension on top of an existing system. It adds context to the data, where context is highly variable information, like lo-

Table 2.7: Overview of advantages and disadvantages of the discussed RS approaches

Content-Based Filtering		**Collaborative Filtering**	
+	-	+	-
Users don't need each other	Domain-specific knowledge required	Domain-independent	Cold-start problem
No item cold-start problem	Overspecialization	High level of novelty	Data sparsity
Domain-specific knowledge incorporated	User cold-start		Low scalability
Explainable			

Knowledge-Based		**Demographic**	
+	-	+	-
No cold-start problem	Feature engineering required	Simplicity of data	Very general recommendations
Highly adaptive	User effort required	Explainable	Rating data required
	Low level of novelty	No domain knowledge needed	Item cold-start problem
			Risk of unfairness

Context-Aware		**Hybrid**	
+	-	+	-
Increase of effectiveness	Retrieval of contextual attributes needed	Increase of effectiveness	Increase of complexity
Easily added to existing systems	Complexity	Utilize all available data	Decrease of generalizability
Irrelevant data is dropped	Domain-specific knowledge required	Counters individual disadvantages	

cation or time. Research has shown that this can increase effectiveness, however, it can also drastically increase complexity.

A Hybrid approach combines more than one system. Seven different ways to combine approaches have been discussed. Hybridization adds complexity but also improves effectiveness. If enough data is available to use more than one approach, one could consider the hybridization of two or more approaches.

For each of these approaches, many models are found in the literature. Which approach is suited for which case is very domain-specific, and mainly dependent on the type of available data. Whichever approach is used, it will have advantages and disadvantages, as summarized in Table 2.7; some of these can be removed by using hybridization.

2.8.1 Discussion & Implementation

In this survey, we have seen that all approaches have weaknesses. Some weaknesses and problems occur in most of the approaches; one of these is selected to be tackled by our implementation. We also saw that hybridization is a good way to counter the disadvantages of different approaches, therefore we apply this in our implementation as well.

The third part of this research consists of the implementation and evaluation of a recommendation system. We choose to use a CFS using matrix factorization. The main reason for this choice is that this method has shown to be very effective in literature. Secondly, because this research aims at RSs in general, we want the system to be widely applicable; therefore, it must be domain-independent. Since matrix factorization consumes a rating matrix only, it can easily be applied in different domains.

The goal of the system is to tackle one of the problems we have encountered in this chapter. Based on this survey, we choose the cold-start problem. Although some approaches are affected more by it than others, this problem occurs in almost any system. We focus specifically on the user cold-start problem: recommending items to new users. Matrix factorization suffers heavily from this problem, the reason for this will become clear in Section 4.1.1.

To conclude, we have chosen a CFS approach as a basis of our implementation and tried to tackle the user cold-start problem. How we approached this challenge and how we altered matrix factorization to handle cold users, are discussed in Chapter 4.

Chapter 3

Approaches to fairness

In this chapter, we take a look at ways to define fairness. We discuss several distinctions that can be made for fairness models, and elaborate upon their differences. The goal of this survey is to get an overview of the complexity of the problem. The main approaches that are used in literature will come about, and specific implementations can be classified in the approaches we discuss here. In the first section, we discuss some preliminary terms which are often used in the context of fairness. Thereafter we describe several distinctions in which specific models can be classified. For each distinction, we shortly discuss the choice we make in our own implementation, along with the reason for that choice. The distinctions are finally used in a discussion about the different fairness objectives. The goal is to select one approach that will be used to measure the fairness level of our implementation in Chapter 4. The metric that is selected is discussed in the last section of this chapter.

To stress the importance of the subject, we note that there is *no free lunch* in terms of designing fair algorithms. If one would build a system without taking fairness into account, unfair behavior can, and most probably will occur in one or several of many ways. Examples are:

- *Algorithmic bias.* An algorithm might base its decision heavily on one or several protected attributes. For example: when selecting an application for a job, the ethnicity of the applicant might be a strong indicator for the algorithm to decide whether a user is a good candidate or not. Such behavior is unintended and even illegal by law.

- *Blindness does not solve algorithmic bias.* With blindness, we mean specifically excluding attributes on which we do not want to base our decisions. This approach fails due to the probable presence of *proxy attributes.* Intrinsic to the data, these attributes can still give away the value of a protected attribute for which the algorithm has been blinded. For example: If a system is blind to the ethnicity of an applicant, it might base its decisions on zipcode. For many geographical areas, it is the case that mainly people of a specific ethnicity live there. Thus the algorithm has used zipcode as a proxy to ethnicity, and ultimately still bases its decisions on the sensitive attribute.

- *Biased data.* The data one feeds to an algorithm is often biased; this is also seen in Section 4.5. In those cases, almost any algorithm will learn to be biased as well; it is extremely hard to overcome this issue. Consider, for example, a police system that tries to detect people that might commit a crime with high probability. The data we feed the system consists of previously committed crimes. If historically relatively many people in a specific neighborhood commit crimes, the algorithm will be biased to select people in that area as possible future crime committers. Moreover, actual crime committers in other regions might not be recognized, because the system focuses solely on historical patterns.

- *Incomplete data.* Incomplete data can also cause algorithms to be unfair. In classification problems, the labels might be biased because data points are missing. In those cases, a clustering algorithm oftentimes also suffers from this bias.

Biased algorithms are a major problem in literature and practice, therefore it is important to take fairness into account when designing a recommendation system. The fact that there is an extensive amount of literature about fairness in ranking, classification, or recommendation, shows the importance of the issue. Many real-life examples exist of unfair algorithms with a high impact on society. One example is risk-assessment software used by the police for risk-assessment regarding possible criminality, which happened to be biased against a certain ethnicity [26]. Another is a holiday recommendation system that recommends more expensive holidays to Apple users [36]. Our own implementation in Chapter 4 will show to have a high level of unfairness as well. This chapter helps the reader get insight into the topic by giving an overview of ways to look at fairness; it gives an idea about the challenges that exist in this domain and emphasizes that the designer of an algorithm must always check that conclusions are based on fair decisions.

3.1 Preliminaries

We discuss some preliminary terminology that will be used during the rest of this chapter. These include some terms that are used throughout the chapter, some challenges that are often encountered, and some methods to achieve fairness.

3.1.1 Disparity

The term disparity is often used in literature when talking about fair algorithms. The reason for this lies in American law, more specifically the Title VII of the United States Civil Rights Act; it states that both *disparate treatment* and *disparate impact* are forbidden. Disparate treatment is the behavior of mistreating someone or some group by certain impermissible criteria. In terms of algorithmic fairness, these criteria are often called sensitive or protected attributes; these are those attributes on which we wish not to discriminate and not to base our decisions. *Disparate impact* is often a result of disparate treatment; with it we mean the unfair impact some decision has for certain individuals or groups. Treatment is often seen as intentional, while impact can be unintentional.

Most algorithmic fairness models focus on disparate impact, as the impact is easier measurable. However, much literature can also be found on disparate treatment [6, 60, 22], sometimes referred to as *procedural fairness.* The distinction between treatment and impact is sometimes vague, even in law, the definition of 'impact' is not always clear.

3.1.2 Protected attributes

As mentioned, algorithmic fairness is often based on *protected* or *sensitive* attributes. Sensitive attributes of users might be ethnicity or gender, but could also be attributes that might not directly 'feel' sensitive. *User activity* is such an attribute: Li et al. [34] use user activity to group users. They found that recommendation systems tend to favor active users. For example, in matrix factorization, the predictions are determined by known ratings, hence if a certain user has often interacted with the system, that user has more influence on the predicted ratings of all users. As another example, the producer of a movie might not directly be seen as a sensitive attribute, but favoring some producers over others is undesired in many cases. This could result in the 'rich get richer and poor get poorer' behavior, as discussed in the introduction of this chapter.

The attribute(s) that should be protected, or used to group users/items, are specific to the domain. Therefore one must carefully think about which attributes to take into account when implementing a fair recommendation, ranking, or clustering algorithm.

3.1.3 Achieving fairness

Pitoura et al. make a distinction between three possible ways to add fairness to a recommendation system [39]. Existing fairness models can be classified in either of these three, we discuss them shortly:

- *Pre-processing* methods are focused on altering the data that is used as input for the recommendation system. The main goal is to remove bias in the data, to prevent the system from adopting this bias. It is based on the idea of *bias in, bias out* [41]. One of the advantages of this method is that it is independent of the model, hence it can be applied to any RS. If fairness pre-processing is successfully performed, this should prevent disparate *treatment*.
- *Post-processing* methods focus on preventing disparate *impact*. They alter the output of a recommender system, such that certain constraints are met. Many of such models are ranking constraints, which assume the recommendation system is a black box. Usually, these constraints come at a cost of accuracy [39], and a balance between accuracy and fairness has to be found. This method is also used in our implementation, as discussed in Section 4.6.
- *In-processing* alters the recommendation system by adding constraints. They are incorporated in the training loop and do not alter the input or output data. Examples are the four fairness metrics introduced by Yao and Huang [59], which are easily incorporated in the loss function of classical matrix factorization techniques. The advantage of this method is that regardless of the (bias in) the input data, the model should learn to give unbiased results.

3.1.4 Recommendations as a ranking or classification problem

Fairness models have usually been proposed for clustering algorithms and rankers, rather than for recommendation systems. However, a recommendation system can easily be modeled as any of these systems. This way the fairness models proposed for clusterers or rankers are applicable in the domain of recommendation systems as well. We discuss how the recommendation problem relates to these problems.

3.1.4.1 Recommendations as a ranking problem

The ranking problem has its roots in information retrieval. The goal is to order a list of documents on relevancy, based on some search queries. The classical example is a search engine like Google, which computes the relevance for the documents to a query, and ranks the documents on descending relevancy. Recommendation systems are closely related to learning-to-rank problems. The main differences are that an RS often does not require a query, but only a user identifier as input. It also has a more extensive goal of not only returning relevant documents but also diverse and distinct items, such that more than simply similar items are recommended. Another major difference is that a recommendation system is focused on personalization: the items that are recommended should be based on the model, history, or preference of a user, instead of only the similarity to a search query.

Most RSs actually rank their items as well: they score each item for a given user and order them on decreasing score. Hence, fairness metrics based on reranking can often be used in recommendation engines too, by reranking their prediction list. An exception is sequential recommendation since these systems update their decision function after every single recommendation; they do not produce a ranked list, but a single item. Underwater most of these systems still use some kind of ranking, hence one can still use the original metric using *in-processing* techniques, as we will also see in Section 4.6.

3.1.4.2 Recommendations as a classification problem

The goal of a classifier is to classify the input into two or more classes. A recommendation system can easily be cast to such a problem, by using a threshold. The idea is that an item is either recommended, or it is not; this view turns recommendation into binary classification. Comparing this to the ranking problem, this is a simplification, as it does not take decreasing user attention into account. But in this way a minimum threshold for a predicted score can be used: if the score is above the threshold the item is recommended, else it is not. A classification fairness model can now be used for the recommendation system!

3.2 Fairness distinctions

The distinctions we describe here are based on the taxonomy defined by Pitoura et al. [39]. There are three aspects, on each aspect, there are two choices, we elaborate upon them in this section.

3.2.1 Fairness level: group versus individual

The first distinction is fairness for individuals versus fairness for groups. For both of these approaches, we need to be able to define how similar items are, but also how similar several outputs of the algorithm are.

In most ML algorithms, the input is represented by vectors. In these cases, we can easily compute the similarity of two items by vector similarity metrics. Examples of these are Cosine similarity, Pearson's correlation, and Kendall's tau. We could also use straightforward distance metrics like Euclidean or Manhattan distance. Different metrics are suitable in different cases. In an optimal world, the similarity of fully dissimilar items is 0, and equal items have a similarity of 1.

The measure of output similarity tells us to what extent two outputs, being either items or groups of items, are similar in terms of fairness. A similarity measure for both rankings and recommendation systems depends heavily on the specific context. One thing one must definitely consider is *position bias*. This behavior is present in all ranking and recommendation engines and has to do with the fact that users tend to pay more attention to top-ranked items / early recommendations. In classic research, this bias is taken into account while measuring effectiveness, for example, in *normalized Discounted Cumulative Gain (nDCG)*, which is the most used metric in ranking systems. However, this knowledge is important for fairness as well. Assume a ranking that is fair when looking at the top 100 items, but unfair when looking at the top 10. Irrespective of the definition of fair, this ranking should be considered unfair due to position bias: the top 10 items receive almost all the attention of the user. This definition of *position bias* is also incorporated in the reranking procedure of Section 4.6. Another important distinction to make is the one between *equality* and *equity* [39]. With equality, we mean the similarity of entities, while equity refers to the value of an entity, for example, the computed score in a ranking system.

3.2.1.1 Group fairness

Group fairness measures how fair algorithms are across groups. Such measurements are often based on the notion of *statistical parity*. A fair algorithm, according to statistical parity, ensures portions of different groups exist in rankings, clusters, or recommendation lists [9]. Groups are created based on one or more protected attributes. Hence group fairness is measured using statistical properties, Friedler et al. [20] define three types. They are all based on the assumption that there is a sensitive attribute S, and that group fairness requires that the values of this attribute to be represented fairly in an output ranking. Assume we are performing binary classification, the preferred class is $Y = 1$, and S is a sensitive attribute, $S = 1$ means the item is in the privileged group. We could easily phrase a recommendation engine in this setting, for example by saying that $\hat{Y} = 1$ if the item is in the top 10, and the group S of an item is defined based on some sensitive attribute. The three types are as follows.

- *Base-rate* approaches do not take the actual distribution of the data into account but only look at the prediction $\hat{Y}$. A base-rate fairness metric looks at the statistical probability that an item has prediction 1 for both $S = 1$ and $S \neq 1$: Equation 3.1.

$$\frac{P(\hat{Y} = 1 | S \neq 1)}{P(\hat{Y} = 1 | S = 1)} \tag{3.1}$$

  If this ratio is close to one, the classification is fair according to the base rate approach. So for both privileged and unprivileged items, the chance of having a preferable output is equal. We do not look at the ground truth values Y, because the distribution in the data may be skewed. Irrespective of this distribution, the output of a classifier must be fair.

- *Accuracy-based* metrics take a look at the error rate of the output. A fair system should have approximately an equal error rate for both the privileged and unprivileged classes. Several measures could be used for accuracy-based fairness, an example is *equal opportunity* [23], given by Equation 3.2

  $$P(\hat{Y} = 1 | Y = 1, S = 1) = P(\hat{Y} = 1 | Y = 1, S = 0) \tag{3.2}$$

  This equation asks that the true positive rate is equal for both privileged and unprivileged groups. Thus as with the base-rate approach, this metric asks for equal chances for both groups. The ratio of the two values is called the error rate balance [14].

- *Calibration-based* metrics are based on the calibration of a classifier. A classifier is well calibrated if $P(Y = 1 | \hat{Y} = p) = p$ for the predicted probability p for class 1 [23]. Calibration-based fairness metrics want the classifier to be calibrated equally well for privileged and unprivileged groups. So for any predicted score $\hat{Y} = y$, we want Equation 3.3 to hold.

  $$P(Y = 1 | \hat{Y} = y, S = 1) = P(Y = 1 | \hat{Y} = y, S \neq 1) \tag{3.3}$$

  Since these probabilities are less easily translated to a ranking setting, we give an example of calibration-based fairness in ranking, following Abdollahpouri et al. [3]. The measurement they use compares two probability distributions for each user; the more equal they are, the fairer the ranking. We have a list of categories C (based on the sensitive attribute), and a user u. $\mathbf{R}_u$ is the set of items rated by u, $\mathbf{L}_u$ is the list of items recommended to u (e.g. the ranking). The distributions are $p(c|u) = \frac{|\{i \in \mathbf{R}_u | i \in c\}|}{|\{\mathbf{R}_u\}|}$ and $q(c|u) = \frac{|\{i \in \mathbf{L}_u | i \in c\}|}{|\{\mathbf{L}_u\}|}$. Hence one compares whether the category distribution in the recommendations is equal to the category distribution in the dataset. Steck introduced this metric [51], and he also uses weights for the items in the equations. These weights could for example be used to model the decrease of attention (like in *nDCG*), by giving lower weights for items later in the list $\mathbf{L}_u$. Decreasing weights are used in our implementation, by means of a *discounting vector*, as discussed in Section 4.6.

3.2.1.2 Individual fairness

Measuring fairness over a group does not by definition result in fairness for an individual [15]. If a fairness-aware recommendation system wants to meet statistical parity, it must make sure that each group is equally represented, where the definition of equality differs per metric. In an attempt to adhere to this requirement, it could randomly select some items from the group, and add them to the recommendation list [9]. This is fair in terms of group fairness, but it can be very unfair to the individuals in the group: suitable items might not be recommended, while unsuitable items might. Therefore one might wish to measure fairness for individuals instead of for groups. Individual fairness is based on the *similar treatment principle* [19]: similar items should result in similar outcomes. Pitoura et al. [39] describe two types of individual fairness

- *Distance-based* fairness uses a distance measure. According to the *similar treatment principle*, we want similar items to be treated similarly. The similarity of items should be directly related to the similarity of outcomes: a higher similarity should result in a lower distance in the outcome. In terms of ranking, this would mean that similar items occur close to each other in a ranking, while dissimilar items are distant from each other.

  As a concrete example, we elaborate upon an approach by Dwork et al. [15] using the Lipschitz condition. Consider two items x, y. A classifier maps each item to a probability distribution over outcomes, say $A(x), A(y)$ for x, y. Now the Lipschitz condition states that the distance between the output of the algorithm of two items must be at most the distance of the items themselves, or, for all x, y, $D(A(x), A(y)) \leq d(x, y)$, where D is a distance metric over distributions, and d is a distance metric over inputs. If now the algorithm satisfies this property for all x, y in the set of items, the algorithm is fair. One could model the output distribution of a ranker as the probability of the item occurring at position i in the final ranking.

- *Counterfactual fairness* was introduced by Kusner et al. [30], and measures fairness with respect to a counterfactual world. Suppose we have item i with group g (based on one or more sensitive attributes), receiving output $\hat{y}$. Now a fair algorithm would give the same output $\hat{y}$ if the group of i was $g' \neq g$, e.g. in a counterfactual world. This is fair since this would mean that the decision of the algorithm does not depend on the sensitive attribute(s).

3.2.1.3 Implementation: choice for fairness level

In our own fairness metric, we focus on group fairness. The main reason is that two groups are naturally present in the dataset: popular versus unpopular groups. The first group is obviously seen as the advantaged group, while the other group has a disadvantage. It is important that unpopular items (producers) still get a fair chance in the recommendation system, hence we want to measure fairness in terms of the probability of unpopular items ending up in the recommendation list. Moreover, using this metric means no other attributes of items are needed to compute similarity, hence we do not have to gather this information. If our implementation is fair with respect to the balance between popular and unpopular items, we would classify our algorithm as fair. As a first step, the metric we use is a *base-rate approach*: It does not take the actual distribution of popular and unpopular items into account. A more sophisticated metric would also look at the size of the popular and unpopular sets in the training data, but we leave this to future work.

3.2.2 Fairness side: user versus producer

In the domain of recommendation systems, we talk about two types of entities: users and items. In terms of fairness, we can either measure it on the side of users (consumers), or on the side items (or their producers) [39].

Item-side fairness is the most commonly used and intuitive definition; it is also the one we usually talk about when discussing fairness. In this setting, we want similar items to be ranked (recommended) similarly, or in the case of group fairness: similar groups should be ranked similarly.

On the other side, we have the users of the system. One of the main differences with item-side fairness is that items are passive, while users are active [12]. Fairness on the producer side is fully determined by the recommender, while the user has an influence on the result. So, when a system has a tendency to provide somewhat unfair recommendations, the user can still influence the results by for example entering another query, while the producer obviously cannot. When talking about user similarity, like with individual item-similarity, we want similar users to receive similar recommendations. To get a feeling of why this is important, we give an example in the domain of recommending vacancies to people. Consider two users who have similar qualifications, and there is a job that is well-paid. If this job is recommended to the first user but not to the second, the second user could feel disadvantaged, as that user might also like to apply for the job. We provide two examples of user-side fairness models. As we will see, item-side fairness approaches are mostly applicable on the side of the user as well.

Li et al. [34] discuss user-oriented fairness by grouping users into advantaged and disadvantaged groups. They define less active users, e.g. users with few ratings in the dataset, as disadvantaged. The idea is that, in CFSs, the predictions are heavily based on the predictions of those active users. Hence recommendations will be tailored to their needs, and less active users will receive less relevant recommendations. Moreover, the majority of the users will be less active, hence the recommendations will be tailored to only a few active users. They propose to solve the unfairness by a reranking algorithm, where they use an *accuracy-based* approach for fairness across user groups.

Yao and Huang [59] discuss four metrics for user-side fairness in rankings. The metrics are all accuracy-based; we discuss only one, as their ideas are very similar. The metrics are based on two user groups: advantaged group g and disadvantaged group $\neg g$. *Value unfairness* is a fairness metric based on the estimation error, it is computed via Equation 3.4.

$$U_{val} = \frac{1}{n}\sum_{j=1}^{n} |(E_g[y]_j - E_g[r]_j) - (E_{\neg g}[y]_j - E_{\neg g}[r]_j)| \tag{3.4}$$

n is the number of items, $E_g[y]_j$ is the average predicted score for item j in the advantaged group, $E_g[r]_j$ the average rating for that group. $E_{\neg g}[y]_j$ and $E_{\neg g}[r]_j$ are the same values for the disadvantaged group. Hence the unfairness increases if the errors for the advantaged and disadvantaged groups diverge. Hereby inaccurate recommendations by themselves are not unfair, but they are if they are not equally inaccurate across user groups. This definition can easily be converted to the item side, by averaging over all users instead of over all items, e.g. over all columns in the *rating matrix* instead of over all rows. Yao and Huang built these metrics for matrix factorization; they are straightforwardly differentiated, and can easily be added as a regularization term to the default loss function.

3.2.2.1 Implementation: choice for fairness side

In the setting of movie recommendations, as with our own implementation, fairness for producers is more important than fairness for users. If users get recommended 'unfair' items (whatever the definition of unfair might be), the impact would be low: in the worst-case users watch movies they are not fond of. The impact for consistent unfair recommendations with respect to items is higher: if certain producers of movies never get recommended by a system, the sales of the movies might be much lower than if they would have been recommended. Therefore our fairness metric should measure fairness on the item side.

3.2.3 Output multiplicity: single output versus multi-output

Up until now, we have discussed fairness for single outputs. In some cases, outputs are ranked lists, but they were still *single lists.* In practice, we sometimes care about fairness over several runs. For example, we might require that the results to a number of queries are fair when we are looking at their combination, and not to each ranking separately [39]. This requirement falls under multiple output fairness constraints.

As an example for multi-output fairness, we discuss the idea of *amortized individual item-side fairness* for rankings, as proposed by Biega et al. [8]. The idea is that rankings are fair for the items if for each item the ratio of equity and attention is similar. Equity is the relevance, and attention decreases as the items occur lower in the ranking. A ranking now adheres to equity of attention if Equation 3.5 holds.

$$\frac{a_1}{r_1} = \frac{a_2}{r_2}, \forall i_1, i_2 \tag{3.5}$$

Here, i_1, i_2 are ranked items, a_1, a_2 are their attention (based on the position in the ranking), and r_1, r_2 are their utility, e.g. their expected rating. As Biega et al. note, equity of attention can almost never be held for individual rankings, as many items have equal equity but attention is never equal. Hence they define the term *amortized equity of attention* as the sum of the equities of attention being equal over all list of rankings. Unfairness is then defined as the absolute difference between the sum of attention and the sum of equity. They also discuss how to rerank a list of items to find the balance between optimal equity and optimal fairness using Integer Linear Programming.

3.2.3.1 Implementation: choice for output multiplicity

As we will see in Chapter 4, our implementation is one in *sequential context*: it uses reinforcement learning. During evaluation, we want to measure the change in fairness over time. While the system is being evaluated, it is improving by means of the rewards it receives. We want to capture whether the fairness level of the algorithm increases, decreases, or maintains a balance over time. A fairness metric over multiple outputs would compute fairness over all rankings (or over a batch of rankings), hence it would be unable to capture this change over time. Our fairness metric should measure fairness over single outputs, to be able to measure change over time.

3.3 Conclusion & Discussion

We have given an overview of ways to approach fairness. In general, the goal of fairness models is to prevent disparate impact and/or treatment of algorithms. Fairness can be achieved by removing the bias in the input data (pre-processing), reranking the output using fairness constraints (post-processing), or changing the learning objective of an algorithm to penalize unfair decisions (in-processing). Constraints can focus on different sides of fairness, the best choice for a model is domain-specific. Individual fairness wants the algorithm to be fair among individuals: similar subjects or objects should have similar output. Group fairness is often based upon statistical parity and requires that different (protected) groups have similar outputs. User-side fairness wants to be fair for the users, while item-side fairness tries to give each item (producer) a fair chance of being recommended. In some domains, fairness in single outputs is not desired or not feasible, in which case fairness can be measured over multiple outputs.

For each of these approaches, many models are found in the literature. A selection has been described here, mainly for the sake of explanation. Different fairness models optimize for different aspects of fairness, and in most cases ensuring one type of fairness means being unfair on other aspects. Therefore the choice for a fairness model must be made carefully, and one must be aware of the consequences.

3.3.1 Discussion & Implementation

The survey has shown that fairness is a complex concept. Many different considerations have to be taken into account, resulting in many different definitions. Different metrics focus on different fairness concepts, and no metric is able to capture all aspects. Choosing a metric inevitably means focusing on some aspects and giving less attention to others. Which metric is suitable depends on context, data, and impact of the system. Often, different metrics could be used and justified in a certain context, and the designer of a system must choose which aspects to focus on.

For each of the three distinctions discussed in this chapter, we have discussed which side weighs more heavily in the context of our own implementation. Concluding these remarks, our fairness metric should measure the *group-based item-side single-output fairness* level. We use a *base-rate approach* to measure fairness between the popular and unpopular groups. As we will see later, we try to adhere to the metric by using a *post-processing* technique.

Chapter 4

Implementing a fair recommendation system

The final part of this research covers the implementation of a fair recommendation system. Before we discuss the approach extensively, we first mention two preliminary concepts that are used in the implementation.

4.1 Preliminaries

The approach is based on a combination of matrix factorization and multi-armed bandits. This section introduces both of these concepts, as they serve as preliminary knowledge to understand how the recommendation system works.

4.1.1 Matrix Factorization

The first concept is matrix factorization, one of the most used approaches in CFSs [29]. One of its advantages is that it is domain-independent: the data it consumes is only a *rating matrix*, consisting of interactions between users and items. This matrix has one row for each known user and one column for each item in the system. The values of the cells are the ratings of that user for that item if that rating is known. Ratings can be provided explicitly or generated out of implicit ratings. In the latter case, they must be converted to explicit ratings before applying matrix factorization.

Matrix Factorization (MF) is based on the idea that both users and items can be represented in some shared latent space. As input, we retrieve rating matrix $\mathbf{R}_{|\mathbf{U}|\times|\mathbf{I}|}$, where $\mathbf{U}$ is our set of users, and $\mathbf{I}$ is our set of items. The matrix is very sparse: almost all cells are empty. This is because, in real life, a user only sees (and thus rates) only a small fraction of the items in the system. Only for those items that the user has interacted with, we know a rating. The goal of MF in the context of CFSs is to fill the cells of **R**, such that we get a fully defined matrix $\hat{\mathbf{R}}$, where the values are predictions for the ratings of the users and items. If we would then request a recommendation for a certain user, we would recommend the item(s) with the highest predicted rating(s) for that user.

To find matrix $\hat{\mathbf{R}}$, MF assumes it is generated via Equation 4.1.

$$\hat{\mathbf{R}} = \mathbf{P}_{|\mathbf{U}|\times f} \cdot \mathbf{Q}^T_{|\mathbf{I}|\times f} \tag{4.1}$$

Where f is the dimension of the latent space. Hence **P** defines the users in the latent space, and **Q** defines the items in the latent space. Note that f is much smaller than both $|\mathbf{U}|$ and $|\mathbf{I}|$. MF only works if the original matrix **R** has a low rank, e.g. a lot of dependencies exist between rows (users) and columns (items). It is widely accepted that this holds for a rating matrix [57]: many users have similar preferences, and many items share similarities as well. Now matrix factorization will capture these dependencies, by factorizing **R** into two smaller matrices **P** and **Q**. In the context of CFSs for movies, one could see the latent space as a list of movie types. Users are then described by defining to what extend they like a certain movie type, and movies are described by to what extend they fall within a certain category (action movie, comedy, etc). The predicted rating for a user u and item i is than computed by multiplying their latent vectors. Matrix factorization now tries to find matrices **P** and **Q** through gradient descent. The loss is computed over the cells that are known in **R**, and the matrices are updated accordingly. Many variants exist for Matrix Factorization [29], but all of them are based on this same principle. Which variant we use and how we apply it in our approach are explained in Section 4.2.1.

4.1.2 Multi-armed bandit

The next concept is the multi-armed bandits. This is a way of tackling the exploration-exploitation trade-off; it is used more often in recommendation systems [32]. A 'nickname' for slot machines in a casino is a *single-armed bandit.* A slot machine could be seen as a bandit, as it 'steals' money from the people who play it. Originally, slot machines had a large handle (arm), which must be pulled to roll the machine.

The Multi-Armed Bandit (MAB) framework derives its name from these machines. The goal of this framework is to maximize the cumulative reward in a reinforcement learning setting, by continuously making the *exploration-exploitation tradeoff.* We explain this tradeoff by considering a casino as an example. Suppose you are the only one in a casino, and the casino contains many slot machines, e.g. the casino is a multi-armed bandit. Your goal is to maximize your reward: make the most money. Each slot machine has a different payout; these are unknown to you when you start this experiment. You will learn more and more about these payouts as you keep track of them while running the experiment. One approach could be to start by pulling each arm once since you know nothing about any of the machines. Now you can take different strategies. You could keep pulling the slot machine that gave the highest initial reward, hence *exploiting* the high expected gain. However, since you are in a casino, you know the slot machines contain randomness. It could very well be that another machine has a higher average reward, but that its initial pull was simple ‘bad luck’. Hence you could also decide to *explore* the other machines a little more, before dedicating yourself to the machine with the highest reward. But if we choose this approach, where do we set the threshold of switching to exploitation instead of exploration? Questions like this have several answers, resulting in different MAB algorithms.

The application of this problem setting in the real world is eminent. It is for example a very suitable alternative for A/B testing. A/B testing is often used in webshops, where the owner wants to compare different versions of a website, with the goal to find the one that yields the highest reward. Reward could be, for example, conversion, the number of sales, turnover, etcetera. An A/B test shows one of two versions to each customer and keeps track of rewards. After an extensive amount of time, one of the two will turn out to be the better option. The advantage of an MAB approach is that more than two options can be evaluated at the same time. The algorithm will balance between exploration and exploitation, and will eliminate sub-optimal options over time; eventually, it will choose the best version.

Another example is an advertisement service that needs to decide which ads to show to users. The ads to choose are the arms, and the reward might be the *click-through rate* (CTR). Yet another example could be the task of finding the best medication for a certain disease. Clinical trials with different drugs could use an MAB algorithm, ultimately leading to the best-suited drug for the disease.

How the MAB problem setting is applied in our system, and which algorithm we choose for the exploration-exploitation trade-off, is discussed in Section 4.2.3.

4.2 BanditMF

The goal of the implementation is to build a recommendation system based on the collaborative filtering approach that focuses on tackling the user cold-start problem. The system we use is based on BanditMF, which is introduced by Xu [57]. Whenever this system needs to make a recommendation for a user, it chooses one of two subsystems. As long as a user is cold, it uses a Multi-Armed Bandit approach as described by Felício et al. [18]. As soon the user is not cold anymore (the exact definition of this will be discussed later), we switch to using Matrix Factorization instead. Multi-armed bandits are used initially because matrix factorization does not work well for cold users.

The reason for this is that for a cold user u similarity to other users cannot be accurately computed. The matrix that matrix factorization produces will contain predicted ratings for u, but these are based on little to no data.

The combination of matrix factorization and multi-armed bandits is a novel one, and can best be classified under the category of *cascaded hybridization*. Intuitively, we prefer to use matrix factorization over the multi-armed bandit, because its predictions are based on the many relationships between items and users, whereas the decisions of multi-armed bandits are quite primitive. However, the predictions for matrix factorization only make sense for non-cold users, since no relations can be found between a cold user and other users or items.

Referring to Figure 2.12, matrix factorization is recommendation system RS1. The uncertainty is defined by the question of whether user u is cold. If that is the case, matrix factorization is unsure, and BanditMF will use RS2: the multi-armed bandits. It is important to note that the multi-armed bandits are not a real recommendation system by themselves, as they ultimately correspond to merely a popularity-based ranking. The idea of BanditMF however, is that these bandits can help in kick-starting the matrix factorization unit for cold users.

Xu does not provide the source code for his system, so we implemented it ourselves. The specifics about the implementation are described in this chapter. We differ from the original system in several ways and refer the reader to Xu's work [57] for the original description.

The general flow of the system is shown in Figure 4.1. We discuss each component in detail in this chapter. The figure shows three types of components:

- The rectangles with solid borders are components of BanditMF.
- The rectangles with dashed borders are components that live in a so-called 'Environment'. The existence of an Environment is needed when running BanditMF. The reason for this will become clear later, in Section 4.3.
- The diamond indicates the coldness threshold the system incorporates. We elaborate upon this threshold in Section 4.2.3.

The rest of this chapter is structured as follows. In the remainder of this section, we describe the components and data flow of our implementation of BanditMF. Next, in Section 4.3, we discuss the need for an environment and its components. In the next section, we discuss the evaluation of BanditMF, including the experimental setup and results. Since we want not only to evaluate the system on effectiveness but also on fairness, we implement a fairness metric. The metric and the fairness of BanditMF in terms of that metric are discussed in Section 4.5. We alter BanditMF, aiming to increase its fairness level; the method we use for that is discussed in Section 4.6, its performance is discussed as well. A conclusion is drawn in the final chapter.

We now continue to discuss each component of BanditMF as shown in Figure 4.1. Note that BanditMF runs in a reinforcement learning setting. The system is continuously updated while evaluating, which is why we call the first fit the initial fit. Later fits are based on the rewards received through the testset; these are referred to as *refits*.

4.2.1 MFU

The Matrix Factorization Unit factorizes the original input matrix $\mathbf{R}$ into the fully defined predictions matrix $\hat{\mathbf{R}}$. The optimization is done through Alternating Least Squares [62]; it uses item and user biases in the optimization function, the latent space has size 50.

After fitting, it is used to gather predictions for non-cold users. Retrieving these predictions is very fast: It simply retrieves the row for the queried user. The row contains predictions for each item. The MFU simply forwards the list of predictions to the Ranker.

While the system is running, e.g. while used in production or being evaluated on a testset, the matrix is refit periodically. This is desired, because we are retrieving rewards for each prediction, and we want to use this new information to improve future recommendations. Ideally, we would rerun the factorization after each reward, but that is computationally not feasible. Hence $\hat{\mathbf{R}}$ is recomputed when some threshold `refit_at` is achieved, based on $\mathbf{R}'$. Here, $\mathbf{R}'$ is the original matrix $\mathbf{R}$, but it includes the ratings derived from the new rewards. The value of `refit_at` is based on the cumulative regret; we measure its influence in our hyperparameter search, as discussed in Section 4.4. This batch-sequential update setting makes BanditMF a *sequential recommendation system.* Adding fairness constraints to such systems is less straightforward than to normal RSs, as mentioned in Section 3.2.3. We discuss the fairness metric in Section 4.5.

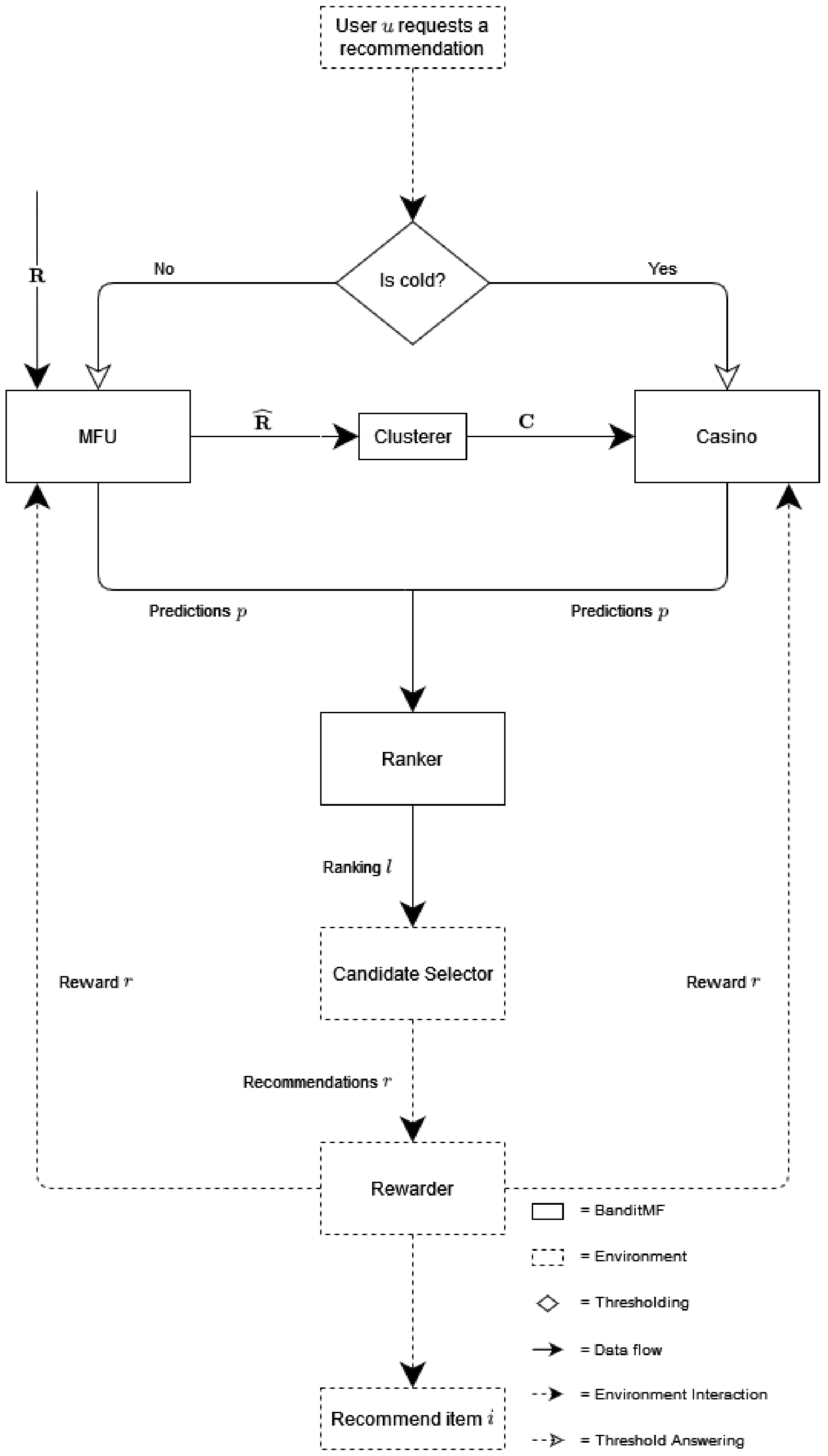

Figure 4.1: The flow of recommendations by BanditMF.

4.2.2 Clusterer

The Clusterer is only used while fitting. It receives predictions matrix $\hat{\mathbf{R}}$ as input and produces a clustered matrix $\mathbf{C}_{\mathbf{k}\times|\mathbf{I}|}$ as output. Hence the Clusterer creates $k =$ `num_clusters` clusters of users. For each cluster c, the prediction for item i is the average of the predictions of the users in the cluster for that item. We use KMeans clustering with `num_clusters` clusters and 20 initializations. The choice for 20 initialization is to balance between accuracy and performance: more initialization means more accuracy (up to a certain level), but lower performance. `num_clusters` is a hyperparameter, we discuss the tested values in Section 4.4. Using other clustering algorithms is left for future work, as discussed in Section 5.1.

4.2.3 Casino

The Casino is a set of multi-armed bandits, namely one MAB for each cold user. During the initial fit, we must decide for each user in the training set whether to directly use the MFU for predictions or to use the Casino first. This decision is based on a threshold, `initial_cu_threshold`, and its value means the following: for every user that has more than `initial_cu_threshold` ratings in the initial training set (e.g. the non-cold users), we will directly use the MFU; for all other users (e.g. cold users), we use the Casino first. While running BanditMF in production, the value for this parameter is important, because it decides which users will initially use the Casino instead of the MFU. This parameter is not of importance in our hyperparameter search due to the way we split our data, as is explained in Section 4.4.2.

The goal of the Casino is to assign cold users to a cluster of non-cold users. Using the MFU for cold users would not be effective, hence we use the Casino initially. It tries to find the cluster to which the cold user belongs; when the certainty for a specific cluster is high enough, it switches to using the MFU for the user.

Upon initial fit, the Casino receives the clustered rating matrix $\mathbf{C}$. Each MAB has `num_clusters` arms: one for each cluster. When the casino encounters a new user, e.g. one that did not occur at all in the training set, it creates an MAB for that user on the fly.

When creating MABs, we set the mean reward for all arms equally, because we do not know to which cluster the user belongs (as that is exactly what we are trying to find out). Hence we pull each arm once and use the global average prediction in $\mathbf{C}$ as the reward for each pull.

Fitting the Casino consists of generating an MAB for each cold user, and setting the initial rewards. During evaluation (testing), the Casino selects the MAB belonging to the user. It pulls an arm based on the UCB1 algorithm [32]. The arm indicates which cluster is selected. As with the

MFU, the result is a predicted rating for each item, which is forwarded to the Ranker. The Ranker eventually selects the item with the highest predicted rating as the one to be recommended.

When the Casino receives a reward for the (u, i) tuple, the reward is saved on the arm (cluster) that was used to generate the prediction. Hence whenever the reward is high, we increase the chance of the user belonging to that cluster. The intuition behind this approach is based on the fact that the Ranker selects the item with the highest predicted rating. Thus if item i is selected, the general opinion of the cluster is apparently that the item is interesting. If the reward of the user is positive, this thus means the user u agrees with the cluster, hence the chance of u belonging to c increases.

For the users that were marked as cold during initial fit by the `initial_cu_threshold`, we want to switch to using MFU at some point: the point where the user switches from cold to non-cold. This moment in time is defined by another threshold: the `certainty_window`. This window is based on the history of the MAB of the user. Whenever the system sees that the MAB prefers a certain arm (cluster) for a certain amount of time, it decides to assign the user to that cluster. More specifically, we define `certainty_window` proportional to `num_clusters` and check the total number of pulls for each arm at each timestep. If this value is the highest for one arm for `certainty_window` timesteps in a row, we mark the user as non-cold. `certainty_window` is a hyperparameter we wish to tune for our data, we discuss its values in Section 4.4.

Finally, we mention that alternative approaches could be used for switching from the Casino to the MFU. Experimenting with different approaches is outside of the scope of this project, but we mention some possibilities for future work in Section 5.1.

4.2.4 Ranker

In the original BanditMF algorithm, the Ranker is not very interesting. It ranks the items decreasing on predicted rating. To recommend one item, one can select the first item that has not been seen in both a) the training set and b) the test set.

While the system is running in production, one might want to loosen constraint a): The fact that a user has yet seen an item in the training set should not mean we cannot recommend it to him/her anymore. The reason we do want to conform to this constraint in evaluation mode is that we want to keep the training and testing set separated. If we would not do this, the recommender could learn to simply recommend the highest rated items in the training set, thereby achieving a relatively high mean reward. This behavior is called overfitting and should be avoided. Hence we forbid the system to recommend items in the training set.

Forbidding constraint b), namely that the system recommends items that have yet been recommended in this run (items in the testset), is done for the obvious reason that we want the evaluation to terminate at some point. If we enforce this constraint, we know the system terminates as soon as all items in the testset have been recommended by the system. This holds because of how we define the Candidate Selector, as discussed later. This property is useful while comparing different versions of the system, and also prevents overfitting on the test/validation set. While in production, this property should be dropped or loosened, as discussed in Section 4.3.1.

4.3 The environment

As mentioned, BanditMF is a *reinforcement-learning* approach [53]. Therefore, it can only run in an environment; the environment should be capable of rewarding the choices that BanditMF makes. The rewards influence future decisions, for example, whether to use the Casino or the MFU whenever a recommendation is requested. The consequence of this setting is that evaluation becomes a little uncomfortable: the system is still learning (and hence improving its effectiveness) while it is being evaluated. Directly after the initial fit, the system would not be very effective; while evaluating and hence receiving rewards, the system is getting better and better in recommending interesting items to users. This also means that to evaluate BanditMF, a *Simulation* needs to be created that mimics the environment that would exist when the system runs in production.

The way the environment is mimicked using simulations is discussed extensively in Section 4.4.3. In this section, we discuss the two components that an environment must at least contain. These components behave differently in a production environment than in a simulation (testing/evaluation), we mention the differences between the two.

4.3.1 Candidate Selector

The Candidate Selector drops the items in the ranking that should not be considered valid candidates for the user. Out of the remaining items, the environment selects the first one, as this is the valid item with the highest predicted rating.

In simulation mode, the Candidate Selector drops all items that BanditMF has seen as a recommendation for this user before. E.g. any item that was in the training set for this user or that was yet recommended to this user during the simulation, will be dropped. It drops items seen in the training phase to prevent overfitting, as discussed in the previous section. It also prevents items to be recommended more than once, to enforce termination. This way we know how many items can be recommended for each user in the test set before the set of ratings is exhausted, and thus the evaluation will finish.

In production mode, this component should be configured depending on the domain. For example: In some domains, having seen the highest possible rating for an item might mean that the user has bought that item. In some contexts, it is safe to assume that a user will not buy the same item twice, thus we do not want to recommend this item ever again. In other cases, one might want to set a cool down timer whenever an item has been recommended, such that it forbids the system to recommend the same items over and over again. In any case, one should be less strict in production mode than in simulation mode, as in the latter case, the system forbids items to be recommended more than once.

4.3.2 Rewarder

The Rewarder receives a tuple (u, i), and computes a reward. Similar to the Candidate Selector, it shows different behavior in different modes.

While in production mode, the Rewarder observes the response to a recommendation. The observation can be done in many different ways, examples are eye tracking, computing CTR, and scrolling behavior. The Rewarder should deduce a reward, which is provided as feedback to the MFU and the Casino. In some domains we might measure several rewards; in those cases, one must think about how to combine these ratings.

In simulation mode, there is no actual user to retrieve feedback from, hence it is challenging to define a reward for a recommendation. There is not one single best solution to this problem, other methods than the one described here could be used. We describe some other possibilities in Section 5.1, implementing those is outside the scope of this project. In its current implementation, the Candidate Selector simply forbids the recommender to recommend items for which no rating is known. It does so by dropping all items, except those that are in the testing set. When rewarding, one is sure that for any recommendation, the Rewarder can read the actual rating that was given in the test rating matrix. This rating is converted to a reward by dividing it by the maximum possible reward, to get a reward between 0 and 1. After each reward, the rewarded item is removed from the candidates for u, such that the system will not recommend the same item more than once (See the discussion about the Ranker in Section 4.2.4). Although this is a solution to the problem of unknown ratings, it is quite a primitive one; it drops much information and adds heavy constraints. Implementing a more sophisticated Rewarder, for example by deducing ratings using data imputation, is left for future work.

4.4 Evaluation

This section discusses the evaluation of BanditMF. First, we describe the (choice for) the dataset that is used. Then we elaborate upon how the data is used for training and testing, as this choice is not straightforward. In the next subsection, two evaluation methods are described. the methods aim not only at measuring the effectiveness of the system but also at searching the optimal values for some hyperparameters on the dataset. Next to these evaluations, the effectiveness of BanditMF is compared to three baselines. Finally, the results of the evaluations are shown and discussed.

4.4.1 The dataset

To evaluate the effectiveness of the system, the 'Movielens latest' dataset[1] is used [24]. It contains 100836 ratings of 610 users on 9723 items. The reason this dataset is used is twofold: Firstly because it is used in much related literature, this makes it much easier to compare effectiveness to other works. Secondly, Movielens contains exactly the data that is needed for the original implementation: a list of *(user, item, rating)* tuples.

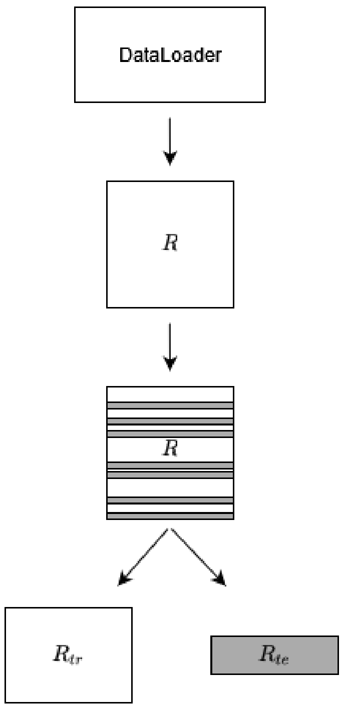

Figure 4.2: Splitting R into R_{tr} and R_{te}: select n users and exclude all ratings of those users from R_{tr}.

4.4.2 Training and testing

The way the data is split is visualized in Figure 4.2. The DataLoader provides the functionality to output the rating matrix. This matrix can contain the data of other datasets than Movielens, but in the evaluation discussed here the Movielens Latest dataset is used. Rather than simply extracting a number of ratings from R to compose R_{te} (A ShuffleSplit), the splitting is done through grouping (A GroupShuffleSplit). `n_users` users are selected to form R_{te}, all known ratings of those users in R are extracted as R_{te}, the rest forms R_{tr}. This choice is made because BanditMF focuses on tackling the cold-start problem. When evaluating its effectiveness, it is thus important that the users in the testset are known to be cold at the start of the testing phase. During the testing phase, most users switch from cold to non-cold, and the evaluator can measure whether effectiveness increases when this happens.

[1]`https://grouplens.org/datasets/movielens/latest/`

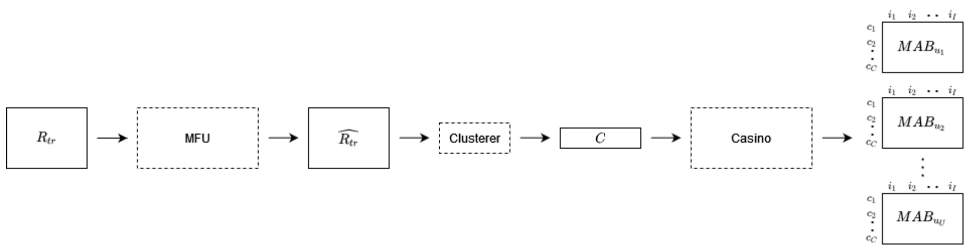

Figure 4.3: Training BanditMF: matrix factorization, clustering, MAB creation.

4.4.2.1 Training phase

The steps of the training phase are shown in Figure 4.3. BanditMF consumes the training rating matrix R_{tr}; training consists of three steps:

1. Using the Matrix Factorization Unit, R_{tr} is converted to a fully defined matrix $\hat{R}$. When the training is done, and recommendations are requested, $\hat{R}$ will be used to extract rows of predictions for users.

2. $\hat{R}$ is clustered using the Clusterer. The result is a matrix C, with the same number of columns, but only `num_clusters` rows: one for each cluster. The values are the average of the predicted values in each cluster.

3. The Casino consumes C and a list of cold users. For each cold user, it creates an MAB, each MAB has one arm for each cluster, each arm initially has the same reward. Since none of the users in R_{te} occur in R_{tr}, no MABs will be created for these users yet. Whenever the first recommendation is requested during the testing phase, an MAB will be created on the fly.

After the training phase, BanditMF is ready to recommend items to users, using $\hat{R}$ for non-cold users, and the MABs in the Casino for cold users. The testing phase is simulated using a Simulator, to mimic using the RS in production. The steps of the testing phase are shown in Figure 4.4.

4.4.2.2 Testing phase

During this phase, the simulation iterates over the (u, i, r) tuples, and provides the user of each tuple separately to BanditMF. For each u, BanditMF computes a ranking l. l is computed using either the MFU or the Casino. If u is cold, the Casino is used. To get the ranking, it selects the MAB of the user, pulls an arm (e.g. selects a cluster), and ranks the row of that arm on decreasing predicted rating. While iterating R_{te}, BanditMF is rewarded by the Rewarder (as discussed below), and hence the user is getting ‘warmer’.

At some point, the tuple (u, i, r) will thus belong to a non-cold user, and the MFU is used instead. In these cases, the row of u is extracted from $\hat{R}$, and it is again ordered on descending predicted rating. Thus in both cases, BanditMF produces a ranking l.

This ranking, along with user u, item i, and rating r, is provided to the Rewarder. Using the Candidate Selector (which is part of the Environment), the Rewarder drops all invalid items from l, resulting in a candidate list c. A reward is computed based on (u, i, r, c), and provided back to BanditMF, which updates its MFU and Casino accordingly.

The Rewarder (also part of the Environment) is, next to providing rewards to BanditMF, responsible for evaluating the effectiveness of BanditMF at each timestep. Hence it exposes a `compute_score` function, which computes the effectiveness of the RS on R_{te} after the simulation has finished. To do so, it saves the needed information on each iteration and uses it afterward to measure effectiveness. Two different evaluation methods, corresponding to two implementations of a Simulation (and their corresponding `compute_score`), have been implemented; they are discussed in the next section.

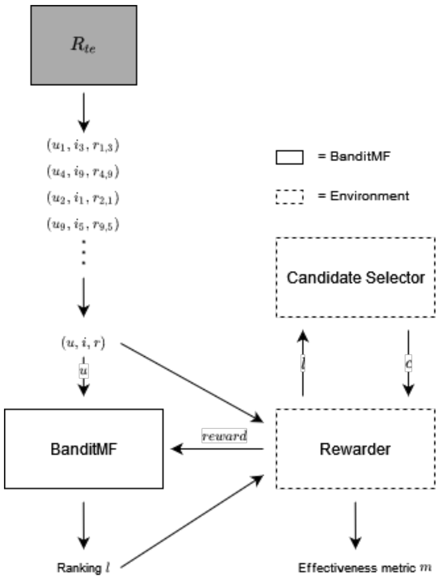

Figure 4.4: Testing BanditMF: iterating tuples, recommending items, rewarding and scoring results.

4.4.3 Evaluation methods

BanditMF is optimized through hyperparameter optimization; the effectiveness of the system is also evaluated and discussed in this section.

As mentioned in Section 4.2, the RS contains four hyperparameters. Since R is split into a training and testing set using GroupShuffleSplit, the parameter `initial_cu_threshold` has no influence on the effectiveness in the current setting: any user in R_{te} is initially cold, irrespective of the value of `initial_cu_threshold`. Three hyperparameters remain to be optimized:

- `num_clusters` defines how many clusters are created in the Clusterer, e.g. the value of k in K-means. The intuition behind this value is that we categorize all users in the system into k categories. If $k = 2$ this distinction is binary, for example, users that like popular movies and users that do not. If k is large, we might argue that we are clustering users based on the primary movie type they like, for example, action movies or comedies. This parameter is tuned to find out what number of clusters provides the best effectiveness on Movielens.

- `certainty_window` defines the size of the window used to measure the certainty of the MAB for a user. The larger the window, the longer it takes for a user to become non-cold. If the MAB for a user has selected one arm more often than each other arm for `certainty_window` times in a row, the user is switched from cold to non-cold.

- `refit_at` is proportional to the maximum rating in the dataset, which is 5 in the case of Moveielens. On each (re)fit, the cumulative regret in the MFU is set to 0. On each reward, the regret is increased by the difference between the prediction and the actual rating. For example, if the RS predicted a rating of 3.85 for an item that was actually rated a 2 by the user, the cumulative regret is increased by 1.85. As soon as the cumulative regret exceeds the `refit_at` threshold, the MFU is refit. Refitting means recomputing the matrix factorization based on all data seen up until now and resetting cumulative regret. Ideally, the Casino is updated as well, but this is left for future work, as discussed in Chapter 5.

  The difference between this parameter and the other two is that intuition tells us lower is better: The more often one refits the MFU, the more data the factorization is based on, the more accurate its predictions should be. This hypothesis is evaluated by incorporating some values for the parameter in the search.

We search the hyperplane of parameters using Scikit Optimize[2]. We use the GaussianProcessRegressor with random initialization points and parallelize batches using Joblib[3]. For a balance between accuracy and efficiency, the values of the parameters are limited. The boundaries were set by manually investigating possible optimal values. The search space is defined as follows.

- `num_clusters` $\in [2, 6]$
- `certainty_window` $\in [1, 5] \cdot$ `num_clusters`
- `refit_at` $\in \{5, 10, 15\} \cdot$ `max_rating`

While searching the hyperplane, one needs to define a loss function to minimize. The loss function is based on the *normalized Discounted Cumulative Gain nDCG*. This is

[2]`https://scikit-optimize.github.io/`
[3]`https://joblib.readthedocs.io/`

the most used metric in recommendation systems; it incorporates decreasing user attention through log discounting. Due to this property, the relevance of items lower in a ranking have a lower weight. Intuitively this is because the chance that the user (actively) sees such item is smaller, hence the equity of items higher in the ranking are of more importance. The $nDCG$ can be measured over a complete ranking, or over the top k items: $nDCG@k$. The loss to minimize is defined as $1 - nDCG@k$. The two different evaluations compute this value in different ways, we discuss both approaches.

4.4.3.1 Batch evaluation

The batch evaluation is depicted in Figure 4.5, its score is based on a ranking that is created after the simulation has finished. The simulation shuffles the list of users based on ratings in R_{te}, and iterates each user u through the RS. For each u the RS provides a prediction list l, out of which the simulation selects candidates c. The first candidate is selected as the item that is being recommended: i. The reward for the RS is the difference between the predicted rating, p, and the actual rating for i: r. The reward is provided to the RS, and the tuple (u, i, p) is saved in a list, s, for scoring the RS after the simulation has finished.

At the end of the simulation, s contains one tuple for each rating in R_{te}. s is split per user, creating one list s_u for each user. s_u is now regarded as a ranking, and scored using $nDCG@k$, by comparing it to the optimal ranking of the items for u. The result is an $nDCG@k$ score for each $k \in K$ for each user $u \in R_{te}$. The $nDCG@k$ is averaged over all users, giving us avg $nDCG@k$; one $k_i \in K$ is selected as the cutoff point. The loss is defined as $1-$ avg $nDCG@k_i$

The idea behind this scoring mechanism is that although in between recommendations of one user recommendations to other users are performed, within each user it is desired that earlier recommendations weigh heavier in the scoring than others. Recommendations to other users mean that system is receiving new rewards in between, but for a specific user u the effectiveness of the RS for other users is not of any importance.

The loss function based on the avg $nDCG@k_i$ is used as the optimization function in the optimizer of `skopt`[4].

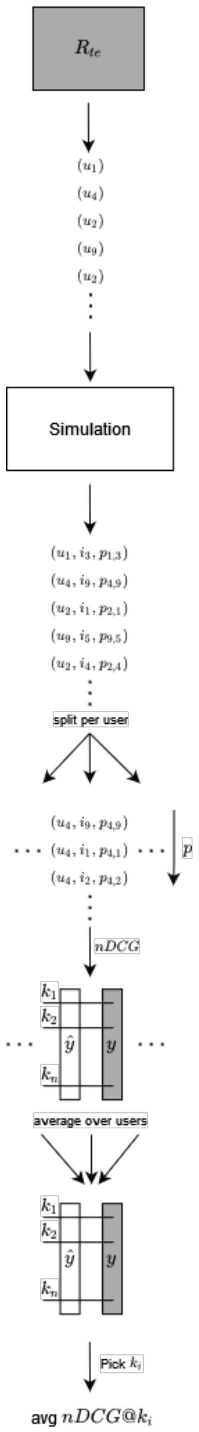

Figure 4.5: Batch evaluation

[4]`https://scikit-optimize.github.io`

4.4.3.2 Prequential evaluation

The prequential evaluation method evaluates BanditMF not by scoring the ranking created over the simulation results, but by measuring effectiveness before each item in R_{te} is seen. This idea is based on *Prequential learning*, as introduced by Vinagre et al. [55]. The method is based on the *test-then-learn principle*: For each (u, i, r) tuple in the stream of the simulation, one first tests RS in its current state, then rewards the RS, before continuing to the next item in the stream.

In the case of batch evaluation, one randomly splits the data in a training and testing set and trains on the first and tests on the latter. Vinagre et al. mention several problems with this approach in the 'streaming environment' setting, which is one where the system is continuously learning from user feedback, as is the case for BanditMF. Two of the mentioned problems are very applicable to the problem discussed in this paper. The first is *incremental updates*: The model is incrementally updating as new data points arrive. Batch evaluation methods approach the system as a static one: one that does not change over time. However, this assumption is not true in our setting, since BanditMF is learning while being trained and evaluated. Therefore, a static evaluation measure is not intuitive, as it is unable to capture change in effectiveness over time. The second is *recommendation bias*: user behavior is influenced by the output of the system. One would not see this problem while evaluating on a static test set, as only static ratings are available. But when such a system would be deployed in production, the act of solely recommending an item increases the chance that the user likes it, simply because it is shown. Such a bias can be captured in prequential evaluation since it measures the system's effectiveness over time.

Vinagre et al. evaluate in four steps, we state them here, along with the applicability to this project

1. *If u is a known user, use the current model to recommend N items to u, otherwise, go to step 3*

   Whenever a user is not known to the system, BanditMF is still able to recommend items by using the Casino. Hence step 2 will never be skipped.

2. *Score the recommendation list given the truly observed item i*

   A recommendation list needs to be created for each item in the test stream. This is ensured by requesting n recommendations for the user u in each (u, i, r) tuple, this ranking of n items will be scored. Since this ranking is not directly related to i itself (it might not contain i), the part 'given observed item i' is not applicable here.

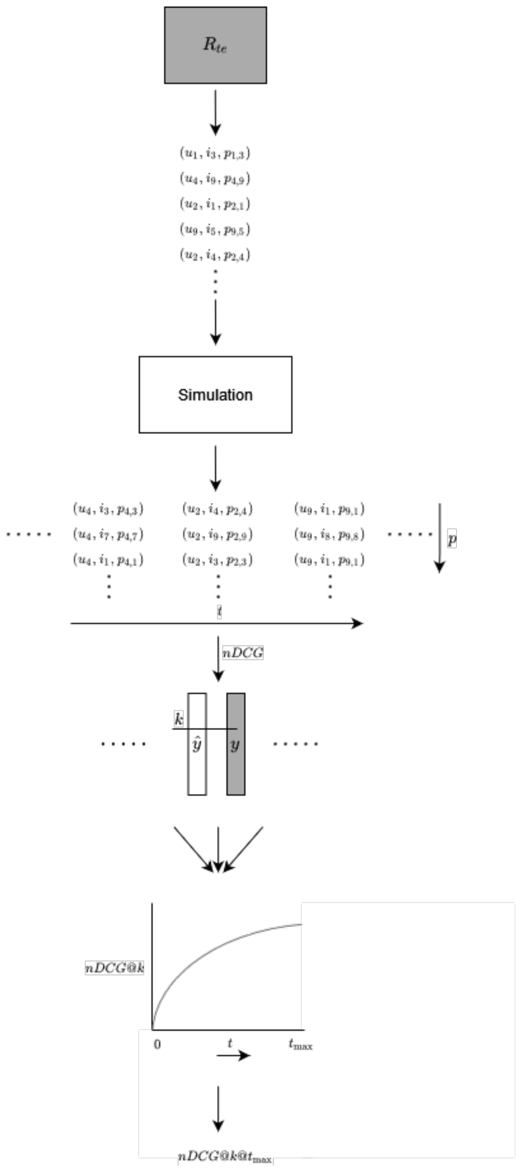

Figure 4.6: Prequential evaluation

3. *Update the model with the observed event*

   BanditMF will be rewarded with (u, i, r), irrespective of the ranking it produced at this time step.

4. *Proceed to the next event in the dataset*

   Continue to the next (u, i, r) tuple.

The evaluation method is depicted in Figure 4.6. Each tuple (u, i, r) is passed through the simulation. For a tuple, the top n items of the candidates c are selected as 'ranking'. For this ranking, the simulator computes $nDCG@k$, taking the candidates (with their corresponding ratings in R_{te}) as optimal ranking. Taking c as optimal instead of all items for u is important, as the list of candidates shrinks after each recommendation for u. This is the case because the system is not allowed to recommend items that it has yet been rewarded for, as described in Section 4.3.1.

The shrinkage of c means that as time proceeds, c is shorter for each u, resulting in higher $nDCG@k$ values. k is kept constant (10 in the experiments); $nDCG@k$ can be computed as long as $len(c) > k$. For large c, for example, 200, BanditMF thus has a large pool of items to choose from. The chance of it selecting low-rated items is present. Because for a list of 200 items, it is very likely that k of them will have high ratings, and selecting low-rated items results in a low $nDCG@k$ score. While this is desired, capturing this error is harder for short c: picking the highest-rated items by chance is likely, since there are simply fewer items to pick from! This problem is not yet solved, and its implications are seen and discussed in the next section.

Prequential evaluation results in a value of $nDCG@k$ for each timestep, providing a graph of the learning curve. This curve is very noisy, as the value differs for each user in R_{te}, and the users are shuffled. Therefore, to conclude in a final score, the evaluation takes the average $nDCG@k$ over the last t time steps. The loss used to optimize using `skopt` is 1 - $nDCG@k@t_{max}$. The results are discussed in the next section.

4.4.4 Results

For both evaluation methods, the hyperplane of hyperparameters is searched. Since the values of the parameters are limited, only 60 combinations exist. Therefore `skopt` will search all possibilities, instead of pruning some combinations. The evaluation algorithm is shown in Algorithm 1. Movielens (R) is split into a training and testing set. The training set is used in the evaluation, the RS will later be tested on R_{te} with the optimal parameter combination. R_{tr} is split into 5 splits of train & validation sets. Each split has a testset that contains all ratings of 25 users. All combinations of hyperparameters are computed and iterated. For each combination, all splits are simulated. The results of each split are aggregated (averaged) and logged to mlflow[5]. After the simulation has finished, every combination has been tested using 5-fold cross-validation, the scores are averaged.
The results of batch and the prequential evaluation are shown in Table 4.1 and Table 4.3 respectively. Each row corresponds to a combination of the

[5]`https://www.mlflow.org/`

Algorithm 1: Running an evaluation

```
   Data: $R$, hyper_parameters
   Result: Evaluation results logged to mlflow
1  $R_{tr}, R_{te}$ = split($R$, test_size=10%);
2  splits = GroupShuffleSplit($R_{tr}$,test_size=25, n_splits=5);
3  forall combination ∈ hyper_parameters.combinations() do
4      scores = list();
5      forall train,val ∈ splits do
6          simulation = simulate(train, val, combination);
7          score = simulation.score();
8          logs = mlflow.log(score);
9          scores.append(score);
10     end
11     aggregated_score = aggregate(scores);
12     mlflow.log(aggregated_score);
13 end
```

three hyperparameters; their values are shown in the first three columns. The columns 'Avg switch' and '% Switch' respectively indicate the average number of recommendations needed for a user to switch from cold to non-cold, and the fraction of users for which this occurred before the testset was exhausted. The rows are ordered on descending loss, and only the top 10 and bottom 10 rows are shown.

4.4.4.1 Batch evaluation

In Table 4.1, the loss is based on the *nDCG*@100, which is averaged over users and splits. Taking a value for k lower than 100 does not make sense, since most users switch from cold to non-cold at 25-50 recommendations. To capture both the effectiveness of the Casino and the MFU in one loss, k should be at least twice the average switch moment.

Low variance in loss We see that all losses are very close to each other: the highest value is less than 9% higher than the lowest value. The reason for this minor difference might be explained as follows. The hyperparameters influence the average switch moment, which is positively correlated to the percentage of switched users. As expected we see that if the certainty window increases, so does the average switch moment. Likewise if the number of clusters increases, the switch moment increases as well. However, the lowest value for the switch moment is 24: on average all users in all iterations switch from cold to non-cold after at least 24 recommendations. *nDCG* uses logarithmic discounting to model decreasing user attention. This means that

Table 4.1: Evaluation results for batch evaluation. The rows are ordered on descending loss; only the top 10 and bottom 10 rows are shown.

#	**window len**	**# clusters**	**refit at**	**Avg switch**	**% Switch**	*nDCG*@100	**Loss**
1	10	2	50	30	83.2%	0.91389	0.08611
2	8	2	25	27	90.4%	0.91258	0.08742
3	10	2	75	31	82.4%	0.91257	0.08743
4	8	4	50	55	70.4%	0.91217	0.08783
5	4	4	25	46	77.6%	0.91172	0.08828
6	6	2	25	24	92.0%	0.91171	0.08829
7	12	4	50	64	64.0%	0.91166	0.08834
8	4	4	25	43	70.4%	0.91159	0.08841
9	6	2	50	24	92.0%	0.91147	0.08853
10	12	4	25	65	63.2%	0.91147	0.08853
51	30	6	50	139	32.8%	0.90833	0.09167
52	5	5	75	58	63.2%	0.90825	0.09175
53	30	6	75	137	36.0%	0.90786	0.09214
54	10	5	75	71	57.6%	0.90775	0.09225
55	12	6	25	91	46.4%	0.90765	0.09235
56	30	6	25	126	37.6%	0.90759	0.09241
57	24	6	75	119	36.8%	0.90748	0.09252
58	25	5	75	117	37.6%	0.90734	0.09266
59	15	5	50	86	48.0%	0.90688	0.09312
60	18	6	50	109	38.4%	0.90633	0.09367

items lower in a ranking have less influence on the value of $nDCG@k$ for large k. At position 25 in the ranking, items already have a very minor 'saying' in the outcome of the $nDCG@k$ (and thus the loss). Hence whatever the values for the hyperparameters are, they will only have a minor effect on the final value for the loss.

Optimal hyperparameter values Although the differences in losses are minor, they are still present. Looking at Table 4.1, it seems that fewer clusters result in a lower loss: the value for `num_clusters` is lower in the top 10 than in the bottom 10 rows. At first sight, it seems that the same holds for the `certainty_window`. However, this can be explained by the direct correlation with the number of clusters. If we take the relative window size, namely the window size divided by the number of clusters, the correlation to the loss value disappears completely. There seems to be no real pattern for the value of `refit_at`: both high and low values appear in the top and bottom half. This is counterintuitive, as one would think that more refitting would increase effectiveness, but it does not in this case. To substantiate these findings, we compute the Pearson correlation matrix[6].

[6] `https://en.wikipedia.org/wiki/Pearson_correlation_coefficient`

Table 4.2: Pearson correlation matrix for batch evaluation.

	window len	rel window len	# clusters	refit at	Avg switch	% Switch	nDCG@100	Loss
window len	1.00	0.79	0.47	-0.00	0.87	-0.82	-0.47	0.47
rel window len	0.79	1.00	-0.09	-0.03	0.43	-0.39	-0.06	0.06
# clusters	0.47	-0.09	1.00	0.03	0.83	-0.86	-0.65	0.65
refit at	-0.00	-0.03	0.03	1.00	0.01	-0.02	-0.17	0.17
Avg switch	0.87	0.43	0.83	0.01	1.00	-0.97	-0.66	0.66
% Switch	-0.82	-0.39	-0.86	-0.02	-0.97	1.00	0.64	-0.64
nDCG@100	-0.47	-0.06	-0.65	-0.17	-0.66	0.64	1.00	-1.00
Loss	0.47	0.06	0.65	0.17	0.66	-0.64	-1.00	1.00

The matrix is shown in Table 4.2. We note the following:

- The relative window length is not correlated to the loss, hence neither is the parameter `certainty_window`.
- The number of clusters is related to the loss: using more clusters results in a higher loss.
- The value for `refit_at` is not related to the loss.
- If fewer users switch from cold to non-cold (e.g. high avg switch or low % switch), the loss decreases. This is expected: The effectiveness for the MFU is higher than that of the Casino, and the MFU is used more often if more users are switched earlier.

The results do not strongly indicate a correlation between the parameters and the loss. From this evaluation, we decide to take the following values for the parameters:

- `num_clusters` = 2, using fewer clusters means more switched users, which results in a lower loss.
- `certainty_window` = 3 (· `num_clusters`), there is no strong correlation, hence we pick the middle value.
- `refit_at` = 50, there is no strong correlation, hence we pick the middle value.

Although there is no strong correlation, an $nDCG$@100 of 0.914 indicates the system is definitely able to provide decent recommendations.

Table 4.3: Evaluation results for prequential evaluation.The rows are ordered on descending loss; only the top 10 and bottom 10 rows are shown.

#	**window len**	**# clusters**	**refit at**	**Avg switch**	**% Switch**	$nDCG@10@t_{max}$	**Loss**
1	4	2	75	18	90.0%	0.84932	0.15068
2	30	6	25	141	35.6%	0.84833	0.15167
3	20	4	75	71	56.7%	0.84787	0.15213
4	12	3	75	47	66.7%	0.84686	0.15314
5	6	6	25	67	62.2%	0.84678	0.15322
6	5	5	75	51	64.4%	0.84671	0.15329
7	10	5	50	59	55.6%	0.84650	0.15350
8	15	5	50	71	51.1%	0.84650	0.15350
9	30	6	75	145	35.6%	0.84646	0.15354
10	15	5	50	74	51.1%	0.84624	0.15376
51	12	4	25	55	65.6%	0.84361	0.15639
52	24	6	75	127	36.7%	0.84352	0.15648
53	15	5	50	70	51.1%	0.84339	0.15661
54	12	3	25	47	66.7%	0.84338	0.15662
55	4	4	25	39	77.8%	0.84337	0.15663
56	3	3	75	25	90.0%	0.84336	0.15664
57	20	5	75	87	43.3%	0.84329	0.15671
58	2	2	75	13	94.4%	0.84316	0.15684
59	5	5	50	51	64.4%	0.84303	0.15697
60	6	2	75	23	88.9%	0.84274	0.15726

4.4.4.2 Prequential evaluation

The results for the prequential evaluation are shown in Table 4.3. The loss for this evaluation method is based on the $nDCG@10@t_{max}$[7]. The choice for $k = 10$ instead of $k = 100$ is made because of the following reasoning. In this evaluation method, the effectiveness of the RS is measured at each time step. Therefore it is not needed to choose k at least twice the value of the average switch moment, since the score will be computed at any moment, thus also at all moments before and after the switch. Taking k too large would not make sense, since an RS in practice does not show one hundred recommendations to a user at once. Therefore, it is desired that the value of the loss decreases if the effectiveness of the RS increases when it recommends a few items: $k = 10$. It is important to note that the scores of this evaluation cannot be compared to those of the previous evaluation, due to the difference in k. As stated by Wang et al. [56], the value for $nDCG@k$ converges to 1 when k approaches infinity. $nDCG@k_2 > nDCG@k_1$ whenever $k_2 >> k_1$; this is true for these evaluations, hence the scores cannot be properly compared.

[7]This value is very noisy since the $nDCG$ values differ drastically between each user in each timestep. Therefore the final value is the average $nDCG@10$ for the last 100 time steps.

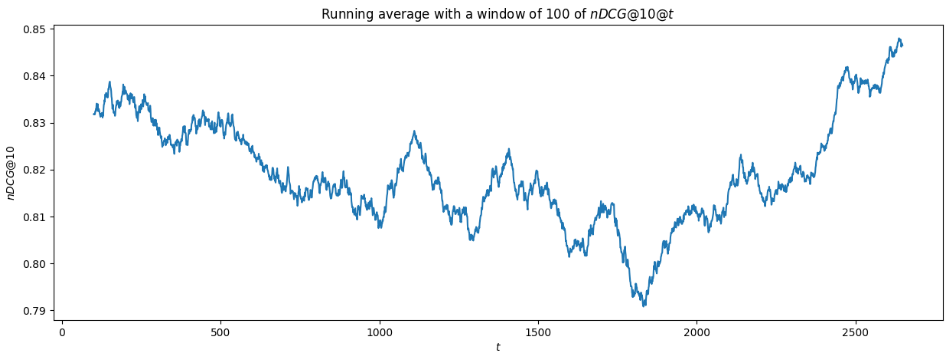

Figure 4.7: Effectiveness over time for prequential evaluation with 6 clusters, a window size of 30, and refitting at a regret of 75. The running average of $nDCG@10@t$ with a window size of 100 is shown on the y-axis. The time is shown on the x-axis and is expressed in the number of tuples that has been iterated.

Low variance loss In this case, the difference between the highest loss and the lowest loss is even smaller: less than 5%. This behavior cannot be explained as in the previous section: the loss indicates the loss at the final time step (and thus the effectiveness of the RS after all ratings in R_{te} have been seen), thus the values of the hyperparameters should influence this score greatly. The reason that this is not reflected in the results might be that the loss is computed through effectiveness after the simulation has been completed. At this point, all data has always been seen, irrespective of the parameter combination. Thus at this point, each version will always be using the MFU, which is based on the same data, hence the effectiveness is similar.

To see differences between configurations, one would need to measure the *change in effectiveness* and compare those. High performance would mean that the effectiveness of the RS approximates the final effectiveness earlier in time than in the case of bad performance. While this behavior has not been extensively investigated, we discuss it briefly by looking at Figures 4.7 and 4.8. These figures show the change of the running average of $nDCG@10@t$ for two different configurations. The first has many clusters, a large certainty window, and does not refit often. The second configuration has few clusters, a small certainty window, and refits often. One would expect that the second configuration approximates the final score earlier (e.g. performs better) than the first configuration. However, this behavior is not reflected in the figures. The change of the score looks very similar; manually investigating other configurations shows that the plots look very similar for *all* combinations. The increase of the $nDCG$ in the last part of the evaluation is probably because of the way candidates are selected, as

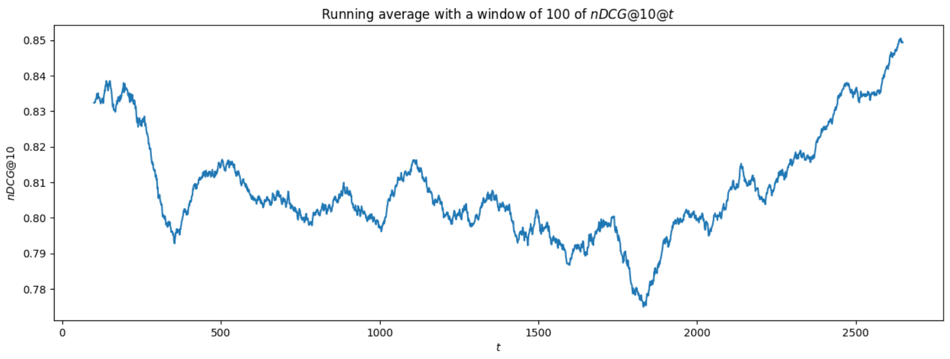

Figure 4.8: Effectiveness over time for prequential evaluation with 2 clusters, a window size of 4, and refitting at a regret of 25. The running average of $nDCG@10@t$ with a window size of 100 is shown on the y-axis. The time is shown on the x-axis and is expressed in the number of tuples that has been iterated.

discussed in Section 4.4.3.2: the optimal DCG is taken over a shorter list of candidates later in time, therefore the $nDCG$ will always increase over time. It is left for future work to investigate why all configurations show similar patterns. A conclusion for the best configuration cannot be drawn from this evaluation, hence we stick to the choices made in the previous section.

4.4.5 Comparison to baseline recommendation systems

To further measure the effectiveness of BanditMF, we compare its effectiveness to three 'baseline' recommendation systems.

4.4.5.1 Matrix Factorization

The idea behind BanditMF is that a multi-armed bandit approach might improve performance with respect to pure matrix factorization when used for cold users. In this section, we compare the $nDCG@10$ for two versions of BanditMF:

1. Default BanditMF. This configuration is the one as described up until now. The hyperparameters are set as discussed in the previous sections.

2. BanditMF with `certainty_window` = 0. This RS always uses the MFU instead of the Casino, since the certainty window used to check whether a user is non-cold is 0, hence all users are always non-cold.

Apart from the certainty window, the configurations are exactly identical. Therefore, any differences in effectiveness would be the result of using the

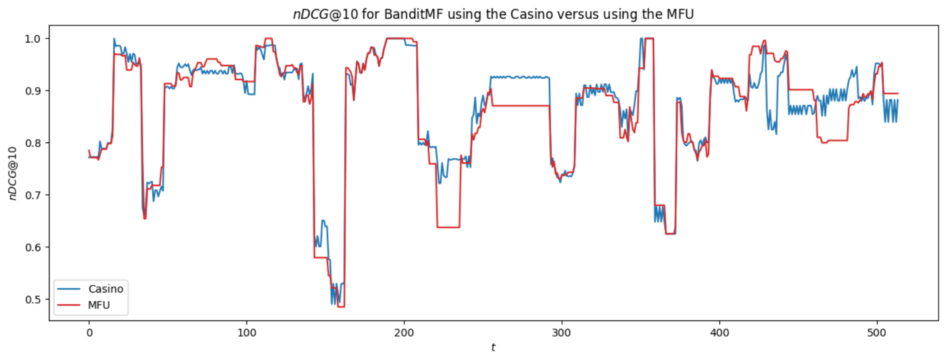

Figure 4.9: $nDCG@10$ for one fold for BanditMF using either only the Casino or only the MFU.

Casino instead of using solely the MFU. 10-fold cross-validation is performed, with validation sets containing all ratings of 25 users. Since we are only interested in the performance on cold users here, all ratings of a user in the testset are dropped from the simulation as soon as the user switches from cold to non-cold. This means that whenever a user switches from cold to non-cold, the evaluation continues to the next user.

The results of the evaluation are odd: The effectiveness of both systems is almost equal for all folds. Neither performs consistently significantly better than the other; the $nDCG@10$ averaged over all folds is similar: 0.859 versus 0.855. The change of the $nDCG@10$ over time for both versions for one fold is shown in Figure 4.9. The figure shows that the performance of both versions is similar for every recommendation: if the MFU performs badly for a certain user, so does the Casino, likewise when the MFU performs well. Manually investigating other runs showed that this behavior occurs in all runs.

This result indicates that using BanditMF does not have a positive impact on effectiveness when compared to matrix factorization. However, it might very well be that the recommendations made to cold users are more aligned with the user's interest when coming from the Casino versus the MFU. The user's interest is not reflected in the effectiveness score determined on the data we have; measuring alignment with the user's interest in more detail is left to future work.

4.4.5.2 Popularity-based ranking

The second baseline is popularity-based ranking. This simple recommendation system does not provide personalized recommendations: it recommends the same items to every user. For each item in the dataset, it computes the average rating; it ranks items on decreasing average rating. The idea is that popular items are on average rated highly, hence this is called popularity-based ranking.

The performance of this RS is computed through 10-fold cross-validation, where each testset contains all ratings of 25 users. Batch evaluation is performed (see Section 4.1); the $nDCG@k$ is computed for each user in the testset for several k. Within folds the scores are averaged over users; the final scores are averaged over all 10 folds. The result: $nDCG@10 = 0.836$, $nDCG@100 = 0.919$. Similar to what we saw in Section 4.4.4.2, $nDCG@k$ increases with k ($nDCG@200 = 0.940$), which is as expected. We see that for both $k = 10$ and $k = 100$ BanditMF and the popularity-based RS achieve similar effectiveness.

It is quite as expected that popularity-based ranking achieves high effectiveness, since items that are rated highly in the training set have a higher a priori probability of being rated highly in the testset. Therefore achieving a score similar to popularity-based ranking is not bad. As discussed in the next section, BanditMF is fairer than popularity-based ranking, and hence preferred over it.

4.4.5.3 Random recommendations

The easiest recommendation method one can think of is random recommendation: recommend random items to users. The performance of such a system should be low. As with popularity-based ranking, recommendations are not personalized in this setting.

We implemented and tested this baseline using the same approach as above: 10-fold cross-validation with 25 users in each testset. The average $nDCG@10 = 0.754$. Even for such a simple recommendation method, the scores are still quite high. This can be explained by the distribution of values in the dataset: there are much more high ratings than there are low ratings; the average rating is 3.25. Thus when an $nDCG@k$ is computed, many ratings of the movies are similar (4, 4.5, or 5), thus the $nDCG$ will be high for most of the possible combinations of rankings.

Although higher than intuitively expected, the effectiveness of random recommendation is significantly lower than that of popularity-based ranking, matrix-factorization, and BanditMF.

4.5 Fairness of BanditMF

The evaluation of BanditMF showed that the system is capable of providing decent recommendations to both cold and non-cold users. However, Section 4.4.5.2 showed that it is quite easy to achieve a high $nDCG$ score by simply basing recommendations on popularity. Such a popularity-based system is not desired in practice, because only the most popular items will get

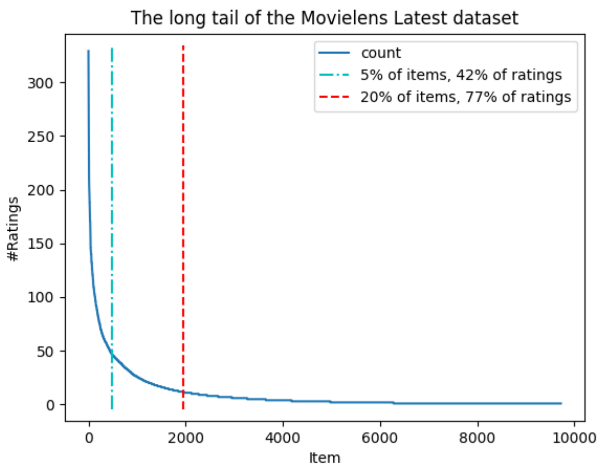

Figure 4.10: The long tail of the Movielens Latest dataset.

recommended. We know intuitively that this is not desired, but that is not captured in traditional performance metrics.

This is exactly where fairness shows to be very important. We do not only want a recommendation system to score high on performance metrics but also want it to be fair. Fairness in the context of movie recommendations is defined on the *item side*: we want all items to have a 'fair chance' of being recommended. Obviously, fairness for items themselves is not the issue here, but the fairness for the directors of the movies is. We are not worried that movies of popular directors like Steven Spielberg or Alfred Hitchcock will get their movies recommended to the public. However, new producers should get a fair chance of being recommended as well, given that their movies are aligned with the preferences of the users. Recommendation systems have an important role in ensuring the 'minority' gets a fair chance, as discussed in Section 1.1.

Next to being fair for producers, this is beneficial to the users too. If a recommendation system is able to provide novel recommendations, this widens the scope of the movies the user will encounter. Moreover, highly talented producers will get a fair chance to reach the public, improving the quality of movie offerings in general.

For these reasons, we measure the fairness of BanditMF using a metric based on the *Average Percentage of Tail items (APT)*, as introduced by Abdollahpouri et al. [2]. This is a *group-based* metric that measures fairness for producers. The fairness is measured across two groups, which we call the advantaged group G^+ and the disadvantaged group G^-. The division is made based on the widely known *long-tail* structure of Movielens. This structure has been studied extensively, and it is found in many datasets. A figure of the long tail in the Movielens Latest dataset is shown in Figure 4.10.The items are ranked on descending number of ratings in the dataset. The dashed line separates the items into two sets: the top 20% versus the bottom 80%. The number of ratings is an indication of popularity: in general, popular items get rated more often. The curve shows that few of the most popular items get most of the ratings: The top 20% receives almost 80% of the items. The dashed-dotted line shows the split for the top 5%, which receives almost half of the ratings. This tells us that the dataset by itself is very skewed. By the *bias in bias out* principle, this means BanditMF probably provides unfair recommendations: it tends to recommend popular items over unpopular ones.

APT stands for *Average Percentage tail Items*, and is able to measure this tendency. The set G^+ consists of all items in the top 20% of the dataset, named the *short-head*. The disadvantaged group G^- consists of the disadvantaged bottom 80% items: the *long-tail*. *APT* now computes a score over a list of one ranking per user, through Equation 4.2 [8].

$$APT = \frac{1}{|U|} \sum_{u \in U} \frac{|i, i \in (l_u \cap G^-)|}{|l_u|} \tag{4.2}$$

Here, U is the set of users; l_u is the ranking for user u. The metric thus averages over all users, and for each user computes the fraction of disadvantaged items in the rankings for that user. The idea behind the metric is that, due to the long tail structure, a recommendation system by default tends to recommend short-head items; it is fairer if it recommends items in G^- as well. A value of $APT = 0.5$ means an equal number of items in G^+ and G^-. Since $|G^-| > |G^+|$, one might even desire a higher value of APT, but in our evaluation, we state that $APT = 0.5$ is the optimal value.

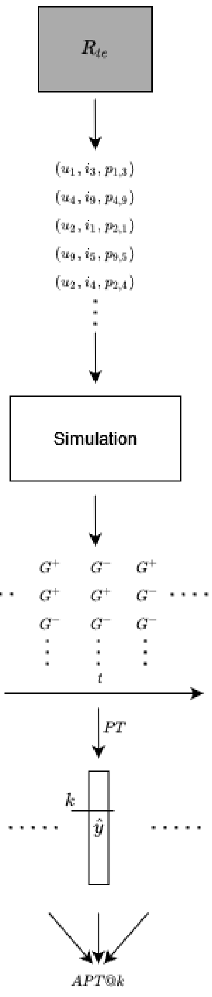

Figure 4.11: Evaluating fairness using $APT@k$.

4.5.1 Results

In line with the Prequential evaluation method (Section 4.4.3.2), we wish to measure the fairness of BanditMF at each timestep. The evaluation method is shown in Figure 4.11. As in the prequential evaluation method, BanditMF recommends k items at each timestep.

[8]The equation is slightly altered here, we refer the reader to the work of Abdollahpouri et al. for the original equation.

For each ranking l_u it computes the *Percentage Tail items (PT)*:

$$PT = \frac{|i, i \in (l_u \cap G^-)|}{|l_u|}$$

Since we are specifically interested in the difference in the fairness of the Casino and the MFU, we compute the APT, APT_C and APT_M, which are the average of the PT, average PT for cold users (e.g. those computed through the Casino), and average PT of non-cold users (e.g. those computed through the MFU) respectively.

We run 25-fold cross-validation. Therefore we create 25 splits of train,test data. In each split, 25 users are randomly selected to form the testset. BanditMF is configured based on the results of the previous evaluations: `num_clusters` = 2, `refit_at` = 50, and `certainty_window` = 6. For each run, the APT's are computed by averaging over all time steps. The final APT's are the averages over the folds. The results are as follows.

$$APT@10 = 0.117 \tag{4.3}$$

$$APT_C@10 = 0.038 \tag{4.4}$$

$$APT_M@10 = 0.130 \tag{4.5}$$

We manually investigate the plots of the PT values. Each fold shows similar behavior; the plot for the first 2500 time steps of one fold is shown in Figure 4.12. This figure shows the PT@10 and the $nDCG$@10 at each timestep. Note that in this evaluation the users are sorted, e.g. the known ratings of one user are all iterated before the next user's ratings are iterated. This explains the gaps in the $nDCG$@10: For each user, the $nDCG$ cannot be computed for the last 10 items, as in those cases the size of the rankings is smaller than 10. The figure also indicates the points in time where a new user starts and those where the user switches from cold to non-cold.

Based on the APT values and Figure 4.12, we note the following:

- In general, the APT is very low: On average, about 1 out of 10 items in the rankings is one of G^-.
- The MFU is fairer than the Casino. The latter has an APT of about 0.01: It almost never selects items from G^-. If we look at how the Casino is defined, this is as expected. When the Casino pulls an arm (selects a cluster), it needs to recommend k items of that cluster. It does so by ranking the items on decreasing predicted ratings and selecting the top k. The predicted rating for each item is the average of the known ratings for that item for all users in the cluster. Therefore the top k items will be those items that are 'popular', hence they are probably one of G^+.

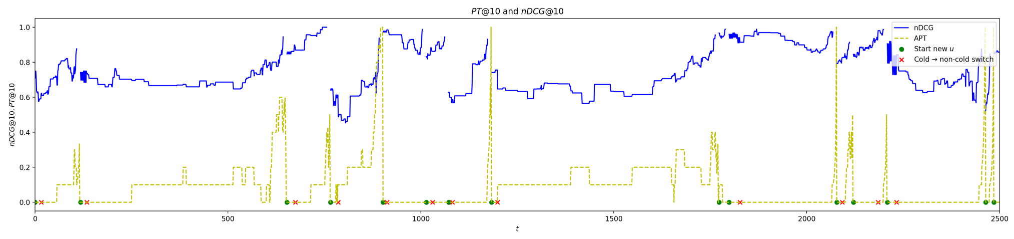

Figure 4.12: $nDCG$@10 and PT@10 for the first 2500 time steps of one fold. The MFU is fairer than the Casino. The $nDCG$ cannot be computed for the last 10 time steps of each user because the ranking is shorter than 10 items in those cases

- APT_M has more weight than APT_C. This is because the Casino is only used for cold users. The users are only cold for the first few items that are recommended to them, the rest of all items in the testset for that user are recommended using the MFU.
- The value of PT increases during the last few items of each user; this can be explained as follows. Since the system does not recommend many disadvantaged items, these items 'remain' in the pipeline. It must recommend these items at some point because it iterates the complete testset, and each item can only be recommended once. Therefore, the ratio $\frac{|G^-|}{|G^+|}$ grows over time, and in the final few time steps the system must recommend the disadvantaged items, thus the PT increases.

The results show that BanditMF is very unfair in terms of this metric. In the next section we discuss how BanditMF is made fair, and to what extent fair decision-making influences performance.

4.6 FairBanditMF

In this final section, we alter BanditMF to make it fairer: FairBanditMF.

The fairness metric, APT, is based on the work of Abdollahpouri et al. [2]; they also discuss a way to incorporate the metric into the loss function of matrix factorization. This method is based on a binary square matrix, which contains a 1 in position i, j if item i and j are in the same group (G^+ or G^-), else it is 0. The metric is called *Intra-List Binary Unfairness (ILBU)*; they evaluated the optimization based on this metric on two datasets: Movielens 1M and Epinions [35], it increases fairness significantly and decreases $nDCG$ insignificantly.

For BanditMF, a method is needed that increases the fairness of both the Casino and the MFU. Since both subsystems output a ranking of items, a *post-processing* method through *reranking* is suitable. As discussed in

Algorithm 2: Ranking items fairly using the FairRanker

```
Data: $k, c, p$
Result: Ranking $r$ of length $k$
1  $c$ = sorted($c$, decreasing=True);
2  $r$ = list();
3  exposures = list(0.0, 0.0);
4  while len(r) < k do
5      if bernoulli($p$) then
6          category = exposures.index(min(exposures));
7          item = $c$.first_of_category(category);
8      else
9          item = $c$.first();
10         category = item.category();
11     $c$.remove(item);
12     $r$.append(item);
13     exposures[category] += $\frac{1}{\log_2(\mathrm{len}(r)+1)}$;
14 end
```

Section 3.1.3, such methods need to find a balance between fairness and effectiveness. This section is dedicated to investigating that balance.

The reranking method we use is based on a *generative process* described by Pitoura et al. [39, §7.1]. The process is generative in the sense that, to select k items of a ranking of length $n > k$, it initializes an empty ranking r and incrementally adds items to that ranking based on fairness constraints. As with our performance metric, we stay in the setting of the *nDCG* framework [27]: we base fairness on *the fairness of exposure* of both groups of items, as introduced by Singh and Joachims [48]. They compute exposure through a discounting vector v, where each index i corresponds to the position in the ranking; v_i indicates the 'importance' of that position. As Pitoura et al. mention, setting $v_i = \frac{1}{\log_2(i+1)}$ results in a logarithmic discounted position vector; we use this vector in our implementation. The reranking procedure is implemented in the Ranker.

4.6.1 FairRanker

Ensuring fair outputs of BanditMF is done through a new implementation of the Ranker that was described in Section 4.2.4. Instead of simply ranking items based on decreasing predicted ratings, the FairRanker is a *generative* process that uses *fairness of exposure* based on the *logarithmic discounting vector v*. The reranking procedure is shown in Algorithm 2. The Ranker consumes three variables. k is the number of items to return in the output. c is the ranking of all candidates, it maps each item to the predicted rating.

Initially, the algorithm sorts c on decreasing predicted rating, such that the first item of c is always the item with the highest rating. p is a Bernoulli threshold, its value must be between 0 and 1. The higher p, the fairer the output of the Ranker. The output of the algorithm is a fairly ranked list r of k items.

The Ranker keeps track of the exposures of both categories G^{-} (0) and G^{+} (1) for the created ranking r; they are set to 0 initially. It adds one item to r each time, as long as $\text{len}(r) < k$. For each item, a Bernoulli trial with a probability of success p is performed. If the trial fails, the item is simply the next item in the list, hence no fairness is taken into account. If the trial succeeds, the Ranker looks at the exposures of both groups. The first item of the group with the least exposure is selected as the next item. The selected item is removed from c (as it cannot be selected anymore) and added to r. The exposure of the category of the item is increased by $\frac{1}{\log_2(i+1)}$, where i is the position of the item (e.g. the current length of r). The result is a ranking r of length k, which is fair to some extent. If $p = 0$, the Bernoulli trial always fails, i.e. the Ranker simply selects the top k items: it behaves equally to the original Ranker. If $p = 1$, the Bernoulli trial always succeeds, and the next item is always the first item of the least exposed category: the Ranker balances the exposure of both groups as fair as possible. In this case, the *APT* of each ranking approaches 0.5. FairBanditMF is now defined as BanditMF that uses the FairRanker instead of the Ranker, we evaluate its fairness.

4.6.2 Results

Before investigating the balance between *APT* and *nDCG*, we first look at the influence of the new ranking procedure.

4.6.2.1 Fairness level of the FairRanker

To find out how well the reranking procedure works, we run the same evaluation as in Section 4.5.1, but now using the FairRanker with $p = 0.8$. The results are as follows:

$$\begin{aligned} APT@10 &= 0.408 \\ APT_C@10 &= 0.328 \\ APT_M@10 &= 0.421 \end{aligned}$$

Again, we manually investigate the *PT* plots. The plot for one of the folds is shown in Figure 4.13. We note the following:

- FairBanditMF is much fairer than the original system: The *APT* increases by 250%, with 4.08 being close to fair (fair means having an *APT* of 0.5).

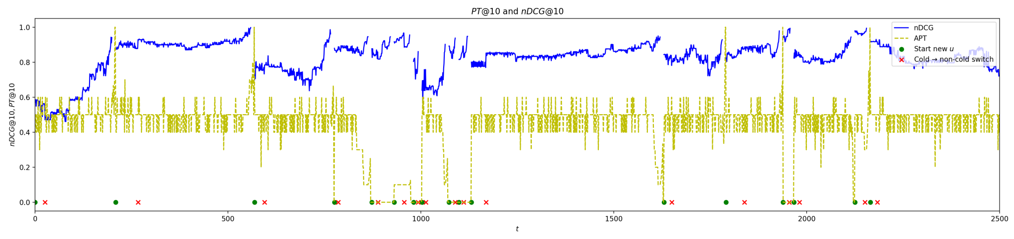

Figure 4.13: $nDCG$@10 and PT@10 for the first 2500 time steps of one fold using the FairRanker with Bernoulli threshold 0.8. FairBanditMF is much fairer than BanditMF. The $nDCG$ cannot be computed for the last 10 time steps of each user because the ranking is shorter than 10 items in those cases

- The recommendations of the MFU are still fairer than those of the Casino. This probably comes from the fact that the PT still increases for the last few items of each user (as reflected in Figure 4.13). The reason for this increase is not clear.
- The $nDCG$ curve does not change drastically when compared to the original BanditMF. This is unexpected, as we would expect a decrease of the $nDCG$. The evaluation in the next section will clarify what is happening here.
- There are still users for which the system is not able to recommend any items of G^-. This occurs for users with only a few items in the testset, hence it probably means that in these cases no items of G^- occur in the testset at all.

We conclude that the FairRanker is very capable of ensuring that the recommendations of BanditMF are fair for the producers. We investigate what value of p one should pick for a good balance between effectiveness and fairness.

4.6.2.2 Fairness versus Effectiveness

To find the balance between effectiveness and fairness, we take a look at the influence of different values of p on the APT and $nDCG$. We use the same hyperparameters as before. Using 5-fold cross-validation with 25 users in each testset, we simulate FairBanditMF for $p \in \{0.1, 0.2, \ldots, 1.0\}$. For each p, the metrics are averaged over all 5 folds; within each fold, they are averaged over all time steps. The results are shown in Figure 4.14. If we look only at Figure 4.14 (a), it seems that the $nDCG$ is not decreasing a lot. However, if we re-label the axis, as done in Figure 4.14 (b), we clearly see a trend.

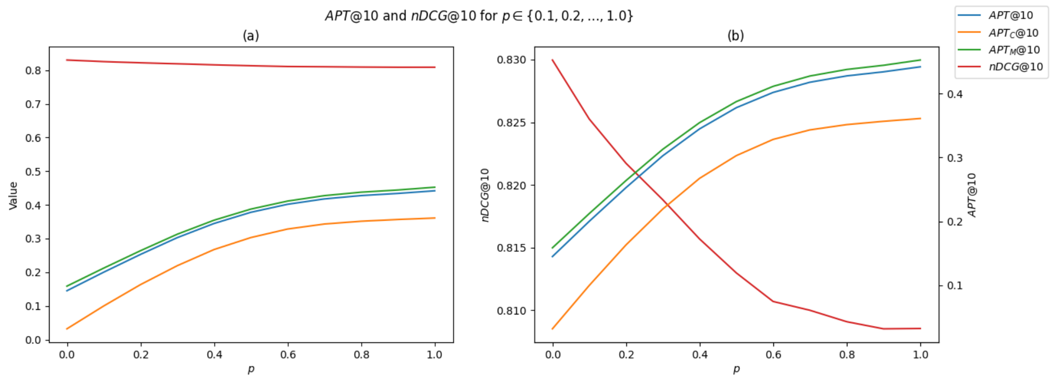

Figure 4.14: $nDCG$@10 and APT@10 for $p \in \{0.1, 0.2, \ldots, 1.0\}$. Both plots show the same data, (b) is a dual-scale figure, while (a) has a single y-axis.

The differences of the values for $p = 0$ and $p = 1$ are as follows:

$$\begin{aligned} nDCG@10 \text{ decreases from } 0.830 \text{ to } 0.809, \text{ a decrease of } &\approx 2.5\% \\ APT@10 \text{ increases from } 0.145 \text{ to } 0.442, \text{ an increase of } &\approx 205\% \\ APT_C@10 \text{ increases from } 0.032 \text{ to } 0.361, \text{ an increase of } &\approx 1000\% \\ APT_M@10 \text{ increases from } 0.159 \text{ to } 0.452, \text{ an increase of } &\approx 185\% \end{aligned}$$

This behavior is similar to what we saw in the evaluation of BanditMF (Section 4.4.4): The effectiveness does decrease, but only by very small values. It is unclear why the value does not change more drastically; one would expect it to be very low when $p = 1$. The reason might be that many items have an equal true rating, as the labels are 'binned' in the sense that people can rate with an accuracy of 0.5. Therefore, reranking some items might not result in a change of the $nDCG$, because the items have the same true score. If item i and item j both have a rating of 4.5, changing their position in the ranking does not change the $nDCG$. In reality, a user might like item j a bit more than item i, for example, the true ratings could be 4.3 and 4.7. Because this difference is not reflected in the labels, it is neither in the score.

Nevertheless, we can still clearly see the influence of p on both the effectiveness and the fairness. As expected, higher values for p result in a higher APT and a slightly lower $nDCG$. Moreover, the RS is almost able to achieve full fairness with $p = 1$: 0.442 is very close to 0.5. Picking the right value for p is a consideration of the designer of a system, and the best value differs per situation. If we look at Figure 4.14 (b), the curves get less steep after $p = 0.5$, so this might be a good default value. At this point the $nDCG$ has an average value of 0.813, the APT is 0.378.

Chapter 5

Conclusions and Future Work

This project researched the yet unsolved problem of making fair recommendations. To structure the research, three questions have been stated in Chapter 1, this paper is structured in line with these questions.

Firstly, the concept of recommendation systems has been studied. The different approaches found in the literature are discussed; their advantages and disadvantages are investigated. By providing an overview of all these approaches in Chapter 2, weighing their pros and cons, and discussing implementations found in literature, we answered our first research question: "What approaches exist for building a recommendation system, and what are their strengths and weaknesses?". The conclusion of this chapter is that many approaches exist, and none of them is perfect. Each of them has pros and cons, and often it is best to combine several methods. But even the most complex and thoughtful implementations will have weaknesses, either in fairness or effectiveness, and the problem of personalized recommendations is definitely still open.

The second part focused on fairness in recommendation systems. We have investigated what approaches exist for measuring fairness, and how they are applicable in the context of recommendation systems. The choices that have to be made when choosing a fairness metric have been discussed; we have also looked at how a recommendation system could adhere to such metrics. Fairness is not an easy concept, and no single best fairness metric can be determined. By providing, in Chapter 3, an overview of the considerations to be made, we have answered the second research question: "What approaches exist for fairness in recommendation systems, and which are relevant in which context?". In conclusion, it is shown that choices have to be made, and a perfect fairness metric does not exist. The designer must choose to focus more on some aspects and less on others, and select its metric accordingly.

These two chapters provided the basis for Chapter 4. In this chapter, a recommendation system called BanditMF is discussed. The goal of the system is twofold. Firstly it tries to tackle one of the most common challenges in recommendation systems: the user cold-start problem. Secondly, it aims to make fair recommendations while maintaining effectiveness. Both its effectiveness and its fairness have been studied extensively. The basic idea behind it is to extend the commonly used approach of matrix factorization, by using multi-armed bandits for cold users. As the evaluations have shown, this approach, unfortunately, does not improve effectiveness on the dataset we have tested it on; although it is able to provide fair recommendations without a significant loss of effectiveness. Many possibilities exist for future research, some of these could improve the performance of BanditMF drastically.

5.1 Future work

In this section, we provide a list of options for future work. These possibilities consist of but are not limited to tuning more parameters, testing on other datasets, and doing more research on specific topics.

Some technical improvements can be made on BanditMF, to improve performance. One of these is refitting the Casino. As discussed in Section 4.2.1, the MFU is refit many times during evaluation. The refit moment is based on a threshold; if the cumulative regret exceeds a limit, $\hat{\mathbf{R}}$ is recomputed. At these moments, it would be best if the MABs in the Casino were refit as well. New data is available, and recomputing the clusters of users should result in clusters that are a better fit to the currently known information. This is not straightforward, since the history of the multi-armed bandits is connected to the current clusters. If clusters are recomputed, the predicted ratings for the clusters differ, and the history of the pulls would not be in line with the new clusters (and thus with the users in those clusters).

Another improvement could be made to the Casino: the way certainty is measured. In the current implementation (see Section 4.2.3), the system looks at the history of pulls; if one arm is preferred for a certain amount of time, the user is assigned to that cluster. Other possibilities for cluster assignment exist. One possibility is to look at the distance of the user to all centroids. This distance could be a simple vector distance, computed over all of the currently known ratings of the user and the average ratings in the cluster. The Casino could then assign the user to a cluster (eg switching it from cold to non-cold) if the distance to a centroid is below a threshold. This method does not only look at the most likely cluster but also ensures that the Casino is used until the user is close enough (defined by the threshold) to that cluster.

The clustering technique could also be investigated. In this research, we used K-means, in line with the work of Xu [57]. However, research has shown that k-means is not very accurate in high dimensions, and users are represented using high-dimensional vectors in BanditMF. Hence the clusters that are used in the Casino might not be very accurate. Using other clustering techniques more suited for a high-dimensional space might improve the performance of the Casino.

The size of the latent space that is used in the MFU is 50 in this research. Other values for this hyperparameter could be investigated.

Another topic to investigate further is other datasets. In this research, we have evaluated solely on Movielens. The effectiveness and fairness might be different on other (types of) data. What if the dataset is much denser, and clusters would be much more accurate? What if almost all users are cold, does the Casino have a bigger benefit in that case?

It would also be interesting to think about how BanditMF would be used in practice. The usage of an Environment (Section 4.3) makes that the system is rather easily incorporated in for example a webshop. The main functionality that would still need to be implemented is rewarding the recommender. Users would need to be recognized across sessions, and their behavior would need to be tracked. This is a very interesting topic, also from the viewpoint of fairness. In a production environment, the effectiveness of the system could also be measured through user feedback, which might be used as rating data (see Section 2.1.3). Such feedback is in practice often more useful than the $nDCG$ or APT scores.

Lastly, cold items are an issue in the current implementation. New items do not occur in $\hat{R}$, hence not in C, hence not in the Casino. Therefore they would never be recommended, and always stay cold. Currently, if BanditMF would be rewarded for a cold item, the MFU would need to be refit directly, since no column exists for that item in $\hat{R}$.

We conclude the research by stating the frequent closing *"More research is needed"*. In its current state, BanditMF has no preference over matrix factorization, as they perform equally well. However, research into the aforementioned topics could show that its performance might be better in other contexts, or with other configurations. The reranking procedure showed to work well; it provides the designer of a system a threshold value to set based on the desired level of fairness of a ranker.

Bibliography

[1] Himan Abdollahpouri. Popularity Bias in Ranking and Recommendation. In *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society*, Honolulu HI USA, January 2019. ACM.

[2] Himan Abdollahpouri, Robin Burke, and Bamshad Mobasher. Controlling Popularity Bias in Learning-to-Rank Recommendation. In *Proceedings of the Eleventh ACM Conference on Recommender Systems*, Como Italy, August 2017. ACM.

[3] Himan Abdollahpouri, Masoud Mansoury, Robin Burke, and Bamshad Mobasher. The Connection Between Popularity Bias, Calibration, and Fairness in Recommendation. *arXiv:2008.09273 [cs]*, August 2020.

[4] Gediminas Adomavicius, Bamshad Mobasher, Francesco Ricci, and Alexander Tuzhilin. Context-Aware Recommender Systems. *AI Magazine*, 2011.

[5] Ebunoluwa Ashley-Dejo, Seleman Ngwira, and Tranos Zuva. A survey of Context-aware Recommender System and services. In *2015 International Conference on Computing, Communication and Security (ICCCS)*. ICCCS, December 2015.

[6] Solon Barocas and Andrew D. Selbst. Big Data's Disparate Impact Essay. *California Law Review*, 2016.

[7] Asia J. Biega, Fernando Diaz, Michael D. Ekstrand, and Sebastian Kohlmeier. Overview of the TREC 2019 Fair Ranking Track. *arXiv:2003.11650 [cs]*, March 2020.

[8] Asia J. Biega, Krishna P. Gummadi, and Gerhard Weikum. Equity of Attention: Amortizing Individual Fairness in Rankings. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, Ann Arbor MI USA, June 2018. ACM.

[9] Reuben Binns. On the Apparent Conflict Between Individual and Group Fairness. *arXiv:1912.06883*, December 2019.

[10] Poonam B.Thorat, R. M. Goudar, and Sunita Barve. Survey on Collaborative Filtering, Content-based Filtering and Hybrid Recommendation System. *International Journal of Computer Applications*, January 2015.

[11] Robin Burke. Hybrid Recommender Systems: Survey and Experiments. *User Modeling and User-Adapted Interaction*, November 2002.

[12] Robin Burke. Multisided Fairness for Recommendation. *arXiv:1707.00093 [cs]*, July 2017.

[13] Òscar Celma and Pedro Cano. From hits to niches?: or how popular artists can bias music recommendation and discovery. In *Proceedings of the 2nd KDD Workshop on Large-Scale Recommender Systems and the Netflix Prize Competition - NETFLIX '08*, Las Vegas, Nevada, 2008. ACM.

[14] Alexandra Chouldechova. Fair Prediction with Disparate Impact: A Study of Bias in Recidivism Prediction Instruments. *Big Data*, June 2017.

[15] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Rich Zemel. Fairness Through Awareness. *arXiv:1104.3913 [cs]*, November 2011.

[16] Ahmed Elngar, Jyotir Chatterjee, Sarika Jain, and Priya Gupta. *Recommender System with Machine Learning and Artificial Intelligence - Practical Tools and Applications in Medical, Agricultural and Other Industries.* John Wiley & Sons, March 2020.

[17] A. Felfernig and R. Burke. Constraint-based recommender systems: technologies and research issues. In *Proceedings of the 10th international conference on Electronic commerce*, ICEC '08, New York, NY, USA, August 2008. ACM.

[18] Crícia Z. Felício, Klérisson V.R. Paixão, Celia A.Z. Barcelos, and Philippe Preux. A Multi-Armed Bandit Model Selection for Cold-Start User Recommendation. In *Proceedings of the 25th Conference on User Modeling, Adaptation and Personalization*, Bratislava Slovakia, July 2017. ACM.

[19] Will Fleisher. What's Fair about Individual Fairness? *AIES '21: AAAI/ACM Conference on AI, Ethics, and Society, Virtual Event, USA, May 19-21, 2021*, 2021.

[20] Sorelle A. Friedler, Carlos Scheidegger, Suresh Venkatasubramanian, Sonam Choudhary, Evan P. Hamilton, and Derek Roth. A comparative study of fairness-enhancing interventions in machine learning. In

Proceedings of the Conference on Fairness, Accountability, and Transparency, FAT 2019, Atlanta, GA, USA, January 29-31, 2019*, Atlanta GA USA, January 2019. ACM.

[21] Amir Gershman, Amnon Meisels, Karl-Heinz Luke, Lior Rokach, Alon Schclar, and Arnon Sturm. A Decision Tree Based Recommender System. *10th International Conference on Innovative Internet Community Services CS, Jubilee Edition 2010, June 3-5, 2010, Bangkok, Thailand*, 2010.

[22] Nina Grgic-Hlaca, Muhammad Bilal Zafar, Krishna P Gummadi, and Adrian Weller. Beyond Distributive Fairness in Algorithmic Decision Making: Feature Selection for Procedurally Fair Learning. *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, 2018.

[23] Moritz Hardt, Eric Price, Eric Price, and Nati Srebro. Equality of Opportunity in Supervised Learning. *CoRR*, 2016.

[24] F. Maxwell Harper and Joseph A. Konstan. The MovieLens Datasets: History and Context. *ACM Transactions on Interactive Intelligent Systems*, December 2015.

[25] F. O. Isinkaye, Y. O. Folajimi, and B. A. Ojokoh. Recommendation systems: Principles, methods and evaluation. *Egyptian Informatics Journal*, November 2015.

[26] Angwin Julia, Larson Jeff, Mattu Surya, and Kirchner Lauren. Machine Bias, May 2016.

[27] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems*, October 2002.

[28] Alexandros Karatzoglou, Xavier Amatriain, Linas Baltrunas, and Nuria Oliver. Multiverse Recommendation: N-dimensional Tensor Factorization for context-aware Collaborative Filtering. In *Proceedings of the 2010 ACM Conference on Recommender Systems, RecSys 2010, Barcelona, Spain, September 26-30, 2010*. ACM, January 2010.

[29] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix Factorization Techniques for Recommender Systems. *Computer*, August 2009.

[30] Matt J. Kusner, Joshua R. Loftus, Chris Russell, and Ricardo Silva. Counterfactual Fairness. *arXiv:1703.06856 [cs, stat]*, March 2018.

[31] Mark Ledwich and Anna Zaitsev. Algorithmic Extremism: Examining YouTube's Rabbit Hole of Radicalization. *arXiv:1912.11211 [cs]*, December 2019.

[32] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. A Contextual-Bandit Approach to Personalized News Article Recommendation. *Proceedings of the 19th international conference on World wide web - WWW '10*, 2010.

[33] P. Li and S. Yamada. A movie recommender system based on inductive learning. In *IEEE Conference on Cybernetics and Intelligent Systems, 2004.* IEEE, December 2004.

[34] Yunqi Li, Hanxiong Chen, Zuohui Fu, Yingqiang Ge, and Yongfeng Zhang. User-oriented Fairness in Recommendation. *Proceedings of the Web Conference 2021*, April 2021.

[35] Paolo Massa and Paolo Avesani. Trust-aware Recommender Systems. In *Proceedings of the 2007 ACM Conference on Recommender Systems, RecSys 2007, Minneapolis, MN, USA, October 19-20, 200.* ACM, 2007.

[36] Dana Mattioli. On Orbitz, Mac Users Steered to Pricier Hotels. *Wall Street Journal*, August 2012.

[37] Rishabh Mehrotra, James McInerney, Hugues Bouchard, Mounia Lalmas, and Fernando Diaz. Towards a Fair Marketplace: Counterfactual Evaluation of the trade-off between Relevance, Fairness &amp; Satisfaction in Recommendation Systems. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, CIKM '18, New York, NY, USA, October 2018. ACM.

[38] Michael J. Pazzani. A Framework for Collaborative, Content-Based and Demographic Filtering. *Artificial Intelligence Review*, December 1999.

[39] Evaggelia Pitoura, Kostas Stefanidis, and Georgia Koutrika. Fairness in Rankings and Recommendations: An Overview. *arXiv:2104.05994 [cs]*, August 2021.

[40] Evaggelia Pitoura, Kostas Stefanidis, and Georgia Koutrika. Fairness in Rankings and Recommenders: Models, Methods and Research Directions. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, Chania, Greece, April 2021. IEEE.

[41] Ashesh Rambachan and Jonathan Roth. Bias In, Bias Out? Evaluating the Folk Wisdom. *arXiv:1909.08518 [cs, stat]*, December 2020.

[42] Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors. *Recommender Systems Handbook*. Springer, 2011.

[43] Shaghayegh Sahebi and William Cohen. Community-Based Recommendations: a Solution to the Cold Start Problem. In *Workshop on recommender systems and the social web, RSWEB*. RSWEB, October 2011.

[44] Ben Schafer, Ben J, Dan Frankowski, Dan, Herlocker, Jon, Shilad, and Shilad Sen. *Collaborative Filtering Recommender Systems*. January 2007.

[45] Guy Shani, Ronen I. Brafman, and David Heckerman. An MDP-based Recommender System. *arXiv:1301.0600 [cs]*, May 2015.

[46] Upendra Shardanand and Pattie Maes. Social information filtering: algorithms for automating "word of mouth". In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, USA, May 1995. ACM.

[47] Richa Sharma and R. Singh. Evolution of Recommender Systems from Ancient Times to Modern Era: A Survey. *Indian Journal of Science and Technology*, 2016.

[48] Ashudeep Singh and Thorsten Joachims. Fairness of Exposure in Rankings. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, July 2018.

[49] Pradeep Singh, Pijush Dutta Pramanik, Avick Dey, and Prasenjit Choudhury. Recommender Systems: An Overview, Research Trends, and Future Directions. *International Journal of Business and Systems Research*, January 2021.

[50] Alexander Smirnov, Alexey Kashevnik, Andrew Ponomarev, Nikolay Shilov, Maksim Schekotov, and Nikolay Teslya. Recommendation system for tourist attraction information service. In *14th Conference of Open Innovation Association FRUCT*. IEEE, November 2013.

[51] Harald Steck. Calibrated recommendations. In *Proceedings of the 12th ACM Conference on Recommender Systems*, Vancouver British Columbia Canada, September 2018. ACM.

[52] Jan Suchal and Pavol Navrat. Full Text Search Engine as Scalable k-Nearest Neighbor Recommendation System. In *Artificial Intelligence in Theory and Practice III - Third IFIP TC 12 International Conference on Artificial Intelligence, IFIP AI 2010, Held as Part of WCC 2010, Brisbane, Australia, September 20-23, 2010. Proceedings*. Springer, August 2010.

[53] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* Adaptive Computation and Machine Learning series. MIT press, Cambridge, MA, USA, 2 edition, November 2018.

[54] Robin van Meteren and Maarten van Someren. Using Content-Based Filtering for Recommendation. In *Proceedings of the machine learning in the new information age: MLnet/ECML2000 workshop*. MLnet, 2000.

[55] João Vinagre, Alípio Jorge, and João Gama. Evaluation of recommender systems in streaming environments. *arXiv preprint arXiv:1504.08175*, October 2014.

[56] Yining Wang, Liwei Wang, and Yuanzhi Li. A Theoretical Analysis of NDCG Ranking Measures. *CoRR*, 2013.

[57] Shenghao Xu. BanditMF: Multi-Armed Bandit Based Matrix Factorization Recommender System. *arXiv:2106.10898 [cs]*, June 2021.

[58] JungAe Yang. Effects of Popularity-Based News Recommendations ("Most-Viewed") on Users' Exposure to Online News. *Media Psychology*, May 2015.

[59] Sirui Yao and Bert Huang. Beyond Parity: Fairness Objectives for Collaborative Filtering. *CoRR*, 2017.

[60] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez Rodriguez, and Krishna P. Gummadi. Fairness Beyond Disparate Treatment & Disparate Impact: Learning Classification without Disparate Mistreatment. *Proceedings of the 26th International Conference on World Wide Web*, April 2017.

[61] Yong Zheng. Utility-Based Multi-Criteria Recommender Systems. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC 2019, Limassol, Cyprus, April 8-12, 2019*. ACM, April 2019.

[62] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *Algorithmic Aspects in Information and Management, 4th International Conference, AAIM 2008, Shanghai, China, June 23-25, 2008. Proceedings*. Springer, June 2008.

Appendix A

Technical details

The technical details, alongside the implementation in Python, can be found on Github.[1]

[1]`https://github.com/wmeints/Building-a-fair-recommendation-system`