Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

The Recommendation Problem

Master's Thesis

Adrian van Schie

vadrian@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Darya Melnyk, Thomas Ulrich
Prof. Dr. Roger Wattenhofer

May 2, 2017

Abstract

In the Recommendation Problem items are recommended to users with the intention to make good recommendations. Given an unknown binary preference matrix, the goal is to discover a certain number of 1-entries with as few queries as possible. We present different offline and online algorithms and compare them on various types of matrices in order to study the differences between the algorithms. We also evaluate the algorithms on two real-world datasets.

Contents

Chapter 1

Introduction

Imagine you are the owner of a bookstore. You want to help your customers in finding books they like. Naturally, better recommendations means better business. A big factor that played a role in the rise of services like Amazon and Netflix is their ability to make good product recommendations. Users are able to rate products and receive recommendations based on their preferences. While Amazon uses 1 to 5 star ratings, Netflix moved from stars to a rating system where users rate products with thumbs up or thumbs down.

In contrast to Amazon and Netflix who have large amounts of data available, we study settings where we start with zero knowledge, i.e., nothing is known about the user-item preferences. At each timestep a user is chosen uniformly at random and a query is made in the form of an item recommendation. We immediately learn the value of the queried entry. The termination criterion of the process is defined by the setting. We study two settings, which can be described as follows with the bookstore example. In the first setting, the goal is to make all customers happy. A customer becomes happy after a number of good recommendations have been made. For example, one user may become happy with three good recommendations and another user may already be happy after being recommended one book she likes. In the second setting, the goal is to make a certain total number of good recommendations. Imagine a greedy bookseller with the goal of selling as many books as possible. In this case, not everybody needs to be satisfied. In both settings we are interested in the total number of recommendations until the process terminates.

We consider various offline and online algorithms. Since a pure offline algorithm would be too strong with complete knowledge about the user-item preferences, we provide less information and call such algorithms *quasi-offline* algorithms. One quasi-offline algorithm is given for each item the number of users who like the item. Another quasi-offline algorithm is given a probability distribution on the users' preferences. The weakest online algorithm simply recommends items uniformly at random. A more sophisticated online algorithm maintains weights on the items and randomly samples items proportional to these weights. The third online algorithm is tailored to perform well on clustered data.

A theoretical analysis of the algorithms proved to be difficult. We therefore decided to do a comparison in practical simulations. A preference matrix is a two-dimensional binary matrix where an entry tells us whether a user likes a certain item or not. We evaluate the algorithms on different types of preference matrices. The evaluation is done by comparing the total number of recommendations until termination of the process. The preference matrices differ in their structure; in one matrix, for example, there is a small subset of very popular items, while the majority of items is unpopular. In another matrix all items are relatively equal in their number of likes, but the matrix has a highly clustered structure. We also compare the algorithms on two real-world datasets. The first dataset comprises movie ratings, and in the second dataset users rated jokes.

Chapter 2

Related Work

The recommendation problem has been studied in many different settings. Alon et al. [1] studied binary online settings where the goal was to approximate the preference matrix as good as possible after a polylogarithmic number of rounds. Real-world preference matrices can be highly clustered; for example, movies and books can be categorized and users have an affinity for certain categories. Kumar et al. [10] studied a model where user preferences correspond to clusters. They proposed an algorithm which is designed to do well on preference matrices with only two clusters and they start with two known ratings of each user. We use their idea to derive an algorithm that performs well on clustered matrices. Uitto and Wattenhofer [14] generalized the binary setting by making no assumptions about the preference matrix. In their setting, one good item needs to be recommended to each user. They perform a competitive analysis and weaken the offline algorithm. They reduce the problem to the *Min-Sum Set Cover (MSSC)* problem and work with a greedy 4-approximation algorithm proposed by Bar-Noy et al. [5] and also studied by Feige et al. [8]. They show that their proposed online algorithm achieves a competitive ratio of $O(\sqrt{n}\log^2 n)$. Further, they prove that any online algorithm is at least a factor of $O(\sqrt{n})$ worse than the weakened offline algorithm, hence their online algorithm is within an $O(\log^2 n)$ factor from the lower bound.

Our first setting is a generalization of the setting in [14]. Instead of recommending one good item to each user, we want to make a number of recommendations to each user, which is dependent on the individual number of items a user likes. Similarly to [14], we weaken the offline algorithm. One such offline algorithm is an approximation to the *Generalized Min-Sum Set Cover (GMSSC)* problem, which was introduced by Azar et al. [3]. GMSSC is a generalization of MSSC and both problems are NP-hard. The input for the GMSSC problem consists of a collection S of sets, each set containing a number of items. In addition, the input consists of a covering requirement for each $s \in S$. The output is an ordering of the different items. The ordering minimizes the average number of items from the ordering needed to meet the individual covering requirements. The authors proposed an $O(\log r)$ approximation algorithm for GMSSC, where r is a parameter that is dependent on the maximum covering requirement of any

set. An improved constant factor approximation algorithm has been proposed by Bansal et al. [4]. They formulate a linear program and make use of randomized rounding. Skutella and Williamson [13] improved the algorithm of Bansal et al. to achieve a better constant-factor approximation.

We present a *Multiplicative Weights* online algorithm in section 4.2.2 whose core idea comes from the *Multiplicative Weights Update method*. The Multiplicative Weights Update method is used in fields like Machine Learning, Optimization, and Game Theory. The Multiplicative Weights Update method has to repeatedly make a decision based on n experts' opinions. Each decision yields a payoff which may change from round to round. A round's payoffs are learned after the decision has been made. The Multiplicative Weights Update method maintains weights on the experts' opinions and makes a decision based on these weights. Initially, all experts' opinions have the same weight. After each round, the weights are updated, depending on how each expert performed. The Multiplicative Weights Update method achieves a payoff which is comparable to the highest payoff any expert achieved. Arora et al. [2] give a good overview over the Multiplicative Weights Update method in their survey paper.

Chapter 3

Model

Our model consists of a set of n users U and a set of m items I. We assume that $m \in \Theta(n)$. Otherwise, for example, if there are many items that nobody likes, a quasi-offline algorithm has a big advantage since it can ignore these items. A binary $n \times m$ matrix M is given, where the entry $M_{i,j}$ describes whether user i likes item j or not (either a 1 or a 0). We refer to an item that a user u likes as a *good* item for u and a recommendation of a good item is called a *good* recommendation. An item's *popularity* denotes the number of users who like the item. At each timestep a user is selected uniformly at random to whom we have to recommend an item. Immediate feedback is given, i.e., the entry $M_{i,j}$ is learned when item j has been recommended to user i. We introduce two settings which are characterized by their termination criterion for the process. The performance of a recommendation algorithm is measured by the total number of recommendations until the process terminates; fewer recommendations are better.

Definition 3.1. (Quasi-Competitiveness) An online algorithm A is *α-quasi-competitive* if for all inputs I

$$c(A(I)) \leq \alpha \cdot c(OPT_q(I)) + O(1),$$

where OPT_q is the optimal quasi-offline algorithm and $c(\cdot)$ is the cost function of A and OPT_q, respectively.

3.1 Setting 1: Satisfying each User

A user is defined to be *satisfied* once a certain number of good items have been recommended. At each timestep a user is chosen uniformly at random from the set of unsatisfied users. The process terminates once all users are satisfied. In our more general setting we want to recommend to each user a fraction f ($0 < f \leq 1$) of her good items. In other words, a user u who likes l_u items becomes satisfied once $\lceil f \cdot l_u \rceil$ good recommendations have been made. Setting $f = 1/m$ will result in a 1-item setting, which was studied in [14].

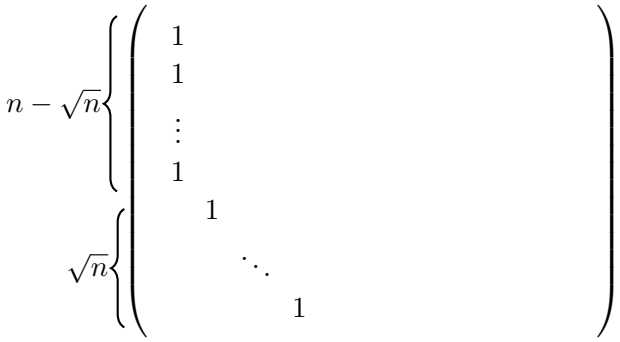


Figure 3.1: A user-item matrix where each user likes exactly one item. One popular item is liked by $n - \sqrt{n}$ users. The remaining $\sqrt{n}$ users like an item with popularity 1.

It was proved in [14] that in the 1-item setting, the quasi-competitive ratio is $\Omega(\sqrt{n})$. The same example can be applied to Setting 1. Figure 3.1 illustrates a preference matrix where all users like exactly one item. There is one item with popularity $n - \sqrt{n}$ and $\sqrt{n}$ items with popularity 1. A quasi-offline algorithm which knows the popular item has a runtime of $O(n)$, since the algorithm knows the items with popularity 1 and therefore satisfying the $\sqrt{n}$ users liking an item of popularity 1 takes $O(n)$ time. An online algorithm also cannot do better than sampling items for these $\sqrt{n}$ users. The difference to an offline algorithm, though, is that an online algorithm does not know which items have popularity 1; it has to sample from all m items. Since $m \in \Theta(n)$, satisfying these $\sqrt{n}$ users will require $\Omega(n^{3/2})$ recommendations. In other words, a quasi-offline algorithm can find the 1-entries of the diagonal efficiently in contrast to an online algorithm. The quasi-competitive ratio in Setting 1 therefore is $\Omega(\sqrt{n})$.

3.2 Setting 2: Recommending a Number of Items

In Setting 1 there may be entries that are hard to find. For example, a user may like only 1 item, which is liked by no other user. The best any online algorithm can do to find this item is sampling. For this reason we want to weaken the requirement to satisfy each user. In this setting the process terminates once a number of k good recommendations in total have been made. In Setting 1 a user that becomes satisfied may still have items that she would like, but have not been recommended. In Setting 2, such a user would still remain in U until the process terminates. A user will only leave, if every item has been recommended to her before the process terminates.

We could show that in Setting 1 the quasi-competitive ratio is at least $\sqrt{n}$. Unfortunately, using the same example with the matrix in figure 3.1 will not yield a reasonable lower bound on the quasi-competitive ratio for Setting 2.

Chapter 4

Algorithms

In this chapter, we present quasi-offline and online algorithms which are evaluated on different input matrices. All algorithms are presented in a generic way. This way the algorithms apply to both Setting 1 and Setting 2. The *done()* and *update()* functions are used in all algorithms. In Setting 1, the *done()* function evaluates to True when all users are satisfied, i.e., when U is empty. The *update()* function removes users who become satisfied from U. In Setting 2, the algorithm terminates once k good recommendations have been made in total. The algorithm keeps track of the number of good recommendations and updates it in the *update()* function.

4.1 Quasi-Offline Algorithms

An offline algorithm which is given complete knowledge about the preference matrix is too strong; this algorithm would make only good recommendations. Therefore we work with quasi-offline algorithms, i.e., algorithms which are given varying degrees of information about the preference matrix.

4.1.1 Harmonic Weights Algorithm

In the 1-item setting from [14] the preference matrix M was created by selecting n vectors according to a probability distribution defined over the users' preference vectors. Their quasi-offline algorithm receives this probability distribution. An algorithm that minimizes the expected number of total recommendations is optimal in this case. As already mentioned, the problem for the 1-item setting is known as the *Min-Sum Set Cover* problem. For our more general Setting 1 the problem is known as the *Generalized Min-Sum Set Cover* problem.

The *Harmonic Weights (HW)* quasi-offline algorithm is an algorithm which was proposed in [3]. It is an $O(\log r)$ approximation, where r is the maximum number of items we have to recommend to any user in Setting 1. The input to the algorithm consists of the users' preference vectors and the number of

items that need to be recommended to each user. The output is an ordering of the different items, which minimizes the average number of recommendations for a user if recommendations are made in this order. The Harmonic Weights algorithm works in rounds and appends in each round one item to the ordering. The algorithm defines weights for users and items. A user's weight depends on the number of items that need to be recommended to the user minus the good items already present in the ordering. Users who need fewer items until they become satisfied are assigned a higher weight. An item's weight is the sum of weights of users who like the item. In each round the item with highest weight is selected and appended to the ordering. The ordering does not necessarily need to contain every item, since all users may become satisfied with a subset of items. The ordering is computed before any recommendation is made. Recommendations are made from the ordering. Apart from removing satisfied users from U, the $update()$ function also increases the ind_u variable, which keeps track of the next item to recommend to user u.

Algorithm 1 Harmonic Weights Quasi-Offline Algorithm - Precompute

$L \leftarrow \emptyset$ ▷ ordered list of items
while U not empty **do**
  for $u \in \{u \in U| \quad |\{i \in L | u \text{ likes } i\}| < l_u\}$ **do**
    $w_u \leftarrow 1/(l_u - |\{i \in L | u \text{ likes } i\}|)$
  end for
  for $i \in I$ **do**
    $w_i \leftarrow \sum_{\substack{u \in U \\ u \text{ likes } i}} w_u$
  end for
  $i \leftarrow \arg\max_{i \in I} w_i$
  $I \leftarrow I \setminus \{i\}$
  remove all users from U who are satisfied with items from L
end while
return L

Algorithm 2 Harmonic Weights Quasi-Offline Algorithm

$L \leftarrow$ precompute with Algorithm 1
$ind_u \leftarrow 0$ for each $u \in U$
while not done() **do**
  choose u uniformly at random from U
  choose item $L[ind_u]$
  update(u, i)
end while

The Harmonic Weights algorithm does not have any connection to Setting 2; since not every user needs to become satisfied, it cannot be described by the GMSSC problem. An ordering that is optimal in Setting 1 is not necessarily

optimal in Setting 2.

4.1.2 Best Remaining Item Algorithm

This quasi-offline algorithm is given the column sums of the preference matrix, i.e., the algorithm learns each item's popularity. Additionally, in Setting 1 the algorithm learns a user's whole preference vector once the user becomes satisfied. We present an algorithm which recommends to each user the item with the highest fraction of remaining 1-entries, which has not been recommended yet to this user. The item with the highest fraction of remaining likes has the highest probability for success, since the users are selected uniformly at random. We name this algorithm the *Best Remaining Item (BRI)* algorithm. The BRI algorithm can be used in both settings. Apart from its basic functionality, the *update*() function manages the bookkeeping needed to compute the fraction of remaining 1-entries r_i for each item i. When a user becomes satisfied in Setting 1, additional updates are made since the whole preference vector of that user is learned.

Algorithm 3 Best Remaining Item Quasi-Offline Algorithm

```
while not done() do
    choose u uniformly at random from U
    choose i not yet recommended to u with highest fraction r_i
    update(u, i)
end while
```

4.2 Online Algorithms

While quasi-offline algorithms are given information about the users' preferences, online algorithms start with zero knowledge about the preference matrix.

4.2.1 Uniformly at Random Algorithm

We present a simple online algorithm which recommends to each user items uniformly at random, without replacement. We call this algorithm the *Uniformly At Random (UAR)* online algorithm.

Lemma 4.1. *In setting 1, the runtime of the UAR algorithm is in* $O(fn^2)$.

Proof. By linearity of expectation, the expected value of total recommendations can be written as the sum of the expected values of the users. The expected value for a single user depends only on the user; it is the expected number of

recommendations until having discovered $\lceil f \cdot l_u \rceil$ 1-entries from a set of l_u 1-entries and $m - l_u$ 0-entries. This expected number of recommendations of a user u is $(m+1) \cdot \lceil f \cdot l_u \rceil / (l_u + 1)$.

$$\begin{aligned}\mathbb{E}[\text{Total recommendations}] &= \sum_{u \in U} \mathbb{E}[\text{Recommendations for } u] \\ &= \sum_{u \in U} (m+1) \cdot \frac{\lceil f \cdot l_u \rceil}{l_u + 1}\end{aligned}$$

Since $|U| = n$ and $m \in \Theta(n)$, the UAR algorithm's expected number of recommendations is in $O(fn^2)$. □

Algorithm 4 UAR Online Algorithm

```
while not done() do
    choose u uniformly at random from U
    choose i uniformly at random from I, without replacement
    update(u, i)
end while
```

4.2.2 Multiplicative Weights Algorithm

The online algorithm in [14] works in two phases; the algorithm discovers popular items in a sampling phase and recommends these popular items in a greedy phase. The algorithm switches between these two phases until all users are satisfied. The main idea of the *Multiplicative Weights (MW)* algorithm is to make the sampling adaptive; instead of alternating between a sampling phase and a greedy phase the algorithm maintains a weight for each item and samples according to these weights. The higher an item's weight is compared to the other weights, the higher is its probability to be sampled. After each recommendation, the item's weight is updated; a good recommendation increases the weight and a bad recommendation decreases it. Popular items are expected to have a higher weight than unpopular items, i.e., eventually these popular items will be recommended with higher probabilities. This core idea of sampling is known as the Multiplicative Weights Update method and it is applied in multiple areas like Machine Learning, Optimization, and Game Theory.

The Multiplicative Weights online algorithm works as follows. Each item is assigned a weight, initially all set to 1. Whenever a user arrives, an item is chosen proportional to the weights of the items which have not yet been recommended to the user. The reason for sampling without replacement is that we do not want to recommend an item twice to the same user. Depending on whether the user liked the item or not, the item's weight is increased or decreased by multiplying

with some constant factor. Eventually the weights of relatively popular items become exponentially bigger than the weights of less popular items.

Comparing with the description of the Multiplicative Weights Update method given in chapter 2, we do not learn the payoff of *all* decisions in hindsight. This would be equivalent to learning a user's whole preference vector, but we only learn whether the user liked the recommended item or not. The strength of the Multiplicative Weights Update method is that it achieves a payoff that is comparable with the best payoff achieved by any fixed decision. Such a statement does not make any sense in our settings, since a fixed decision corresponds to recommending always the same item.

Algorithm 5 Multiplicative Weights Online Algorithm

```
α ← increase-parameter
β ← decrease-parameter
w_i ← 1 for each item i
while not done() do
    choose u uniformly at random from U
    choose i randomly proportional to the weights w_i, without replacement
    if u likes i then
        w_i ← w_i · (1 + α)
    else
        w_i ← w_i · (1 − β)
    end if
    update(u, i)
end while
```

As we will see in chapter 5, the BRI and MW algorithms perform well on preference matrices where some items are more popular than others. The algorithms will detect popular items and recommend them to the users. Since these items are liked by relatively many users, the success rate is higher than recommending items uniformly at random.

4.2.3 Block Algorithm

There exist preference matrices where all items are equally popular even though the matrix contains structure. Such preference matrices are often encountered in the real-world, for example in movie rating systems. Movies can be categorized into genres and users may have an affinity for certain genres and dislike movies of other genres.

We propose a simple algorithm that is tailored for such structured data. The algorithm takes a user-based approach, i.e., it makes recommendations based on the preferences of users that like similar items. The *Block* algorithm finds for a given user u_1 a user u_2 that is closest to u_1 in terms of the number of items

that both users liked in previous recommendations. Then it recommends an item which u_2 liked, but has not been recommended to u_1. If no such item exists, the algorithm finds the next closest user until a recommendation is made or no more users remain who have liked at least one item which u_1 also liked. In the case where the algorithm could not make a recommendation that way, it recommends an item chosen uniformly at random from the set of items which have not yet been recommended to u_1.

Algorithm 6 Block Online Algorithm

```
while not done() do
    choose u uniformly at random
    i ← null
    S ← {u}
    while S ≠ U do
        u' ← argmax_{u''∈U\S} shared_likes(u, u'')
        if shared_likes(u, u') = 0 then
            break
        end if
        if u' likes an item that has not been recommended to u then
            i ← choose item u' liked, but has not been recommended to u
            break
        end if
        S ← S ∪ u'
    end while
    if i = null then
        choose i uniformly at random from I, without replacement
    end if
    update(u, i)
end while
```

Chapter 5

Evaluation

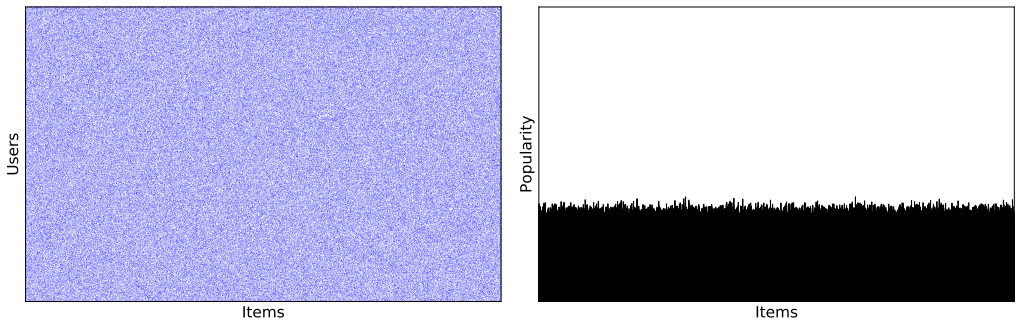
We implemented the algorithms from chapter 4 and compared their performances on different types of preference matrices. In all cases the preference matrix consists of $n = 800$ users and $m = 3000$ items. The performances are averaged over 20 runs. The Harmonic Weights quasi-offline algorithm is an approximation algorithm for Setting 1. Since the algorithm has no relation to Setting 2, we excluded it in the evaluation for this setting. In all simulation runs we worked with $f = 0.5$ for Setting 1 and $k = (\sum_{u \in U} l_u)/2$ for Setting 2. With these parameters the same number of good recommendations needs to be made in both settings, namely half the number of all 1-entries in the preference matrix.

We searched for good parameters for the Multiplicative Weights algorithm. There was no big difference between different α parameters, although α values like 0.3 performed a bit better than $\alpha = 0$. For the β parameter a value between 0.3 and 0.6 performed best. Smaller values like 0.1 led to a noticeably worse performance. We chose $\alpha = 0.3$ and $\beta = 0.3$ for the Multiplicative Weights algorithm in all subsequent comparisons.

5.1 Theoretical Preference Matrices

Theoretical preference matrices are constructed with a generative model. In the simplest case, each user likes an item with a certain probability. Other matrices are created by modifying the probability distribution on the items' popularities. We also present a highly clustered matrix where the preferences are modeled with clusters.

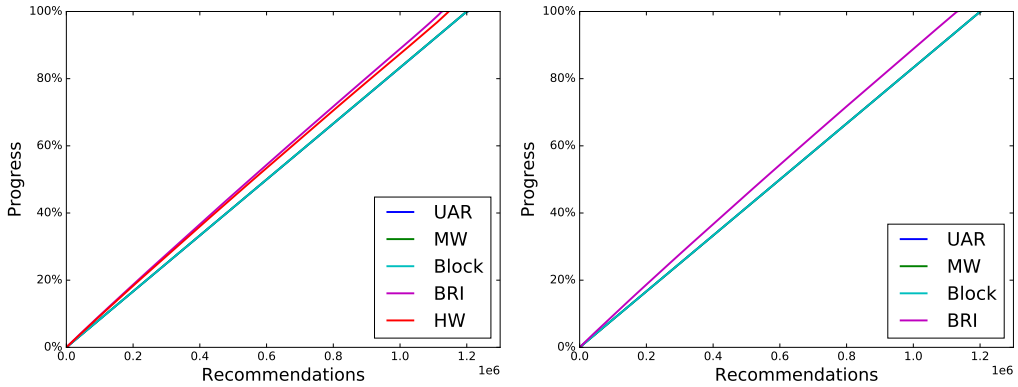
5.1.1 Random Matrix



(a) Preference matrix (b) Likes distribution

Figure 5.1: *Random* matrix

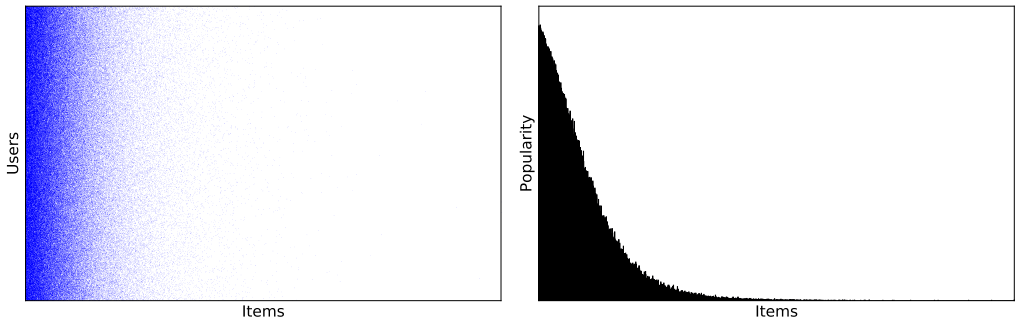
On the *random* preference matrix (Figure 5.1) every item has the same popularity in expectation. A probability p defines the probability that a user likes a single item. An entry of the preference matrix is 1 with probability p and 0 with probability $1 - p$, respectively. Thus, an item has an expected number of $p \cdot n$ likes. Figure 5.2 shows the performances on a random matrix with $p = 0.3$. All algorithms' performances are comparable with the UAR algorithm's performance. The two quasi-offline algorithms have slightly better performances since they can easily detect items with a popularity above average.



(a) Setting 1 (b) Setting 2

Figure 5.2: Algorithm performances on the *random matrix*.

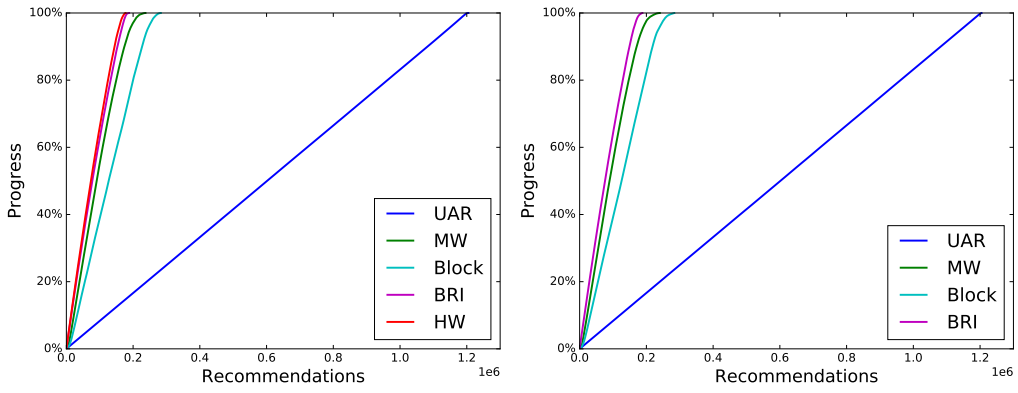
5.1.2 Exponential Matrix



(a) Preference matrix (b) Likes distribution

Figure 5.3: *Exponential* matrix

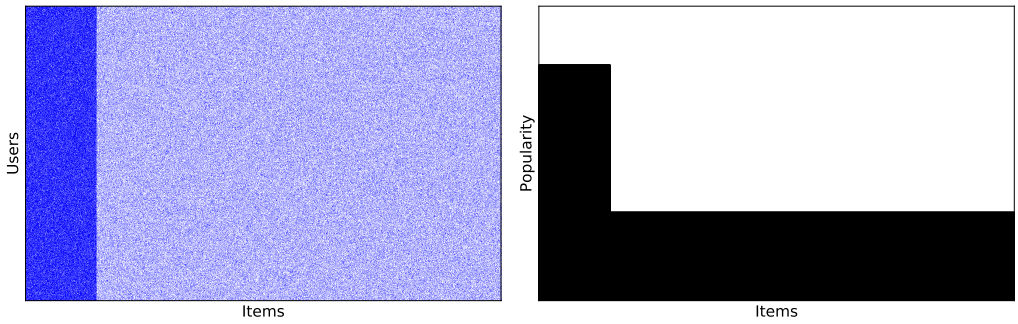
We have seen in the previous section that on random matrices all algorithms are comparable with the UAR algorithm since there is not much to exploit. On *exponential matrices* (Figure 5.3), the distribution on the items' number of likes follows an exponential distribution; a small set of popular items is liked by a majority of users, while most items are liked by relatively few users. The UAR algorithm performs badly, as expected. In general, an algorithm that succeeds in detecting popular items is expected to perform well. We can see in figure 5.4 that all algorithms have a much better performance than the UAR algorithm. The MW algorithm's performance is close to that of the two quasi-offline algorithms, even though it starts with zero knowledge.



(a) Setting 1 (b) Setting 2

Figure 5.4: Algorithm performances on the *exponential matrix*.

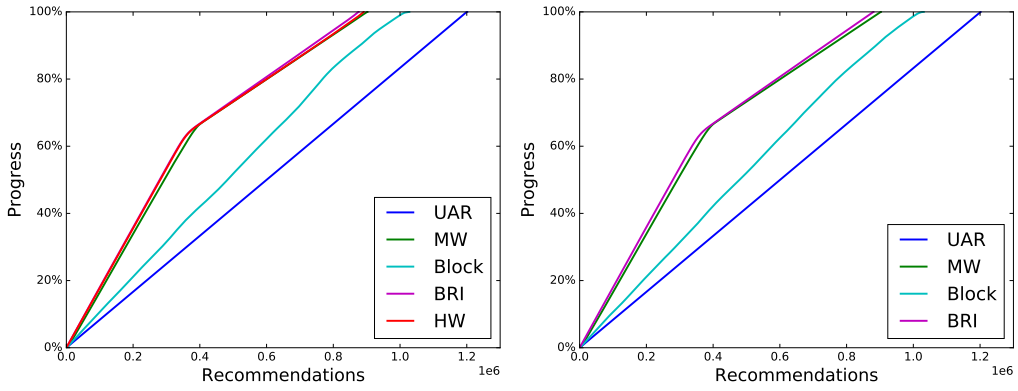
5.1.3 High-low Matrix



(a) Preference matrix (b) Likes distribution

Figure 5.5: *High-low* matrix

The *high-low* matrix (Figure 5.5) consists of two types of items. Items of different types differ by their popularity. Items of the first type are liked by $\lceil p_0 \cdot n \rceil$ users and items of the second type are liked by $\lceil p_1 \cdot n \rceil$ users, p_0 and p_1 in $[0, 1]$. The column-vector corresponding to an item with popularity l is chosen randomly from the set of binary length-n permutations with exactly l 1-entries. The interesting case is where items of one type are fairly popular, while items of the other type are not. The faster an algorithm detects items of the more popular type and recommends these items, the better its performance. Figure 5.6 shows the performances on a high-low matrix where 15% of items belong to the first type with $p_0 = 0.8$ and $p_1 = 0.3$. We observe that the performance graph of better performing algorithms consists of two lines. In the first, steeper line many recommendations were items of the better type, until there were no

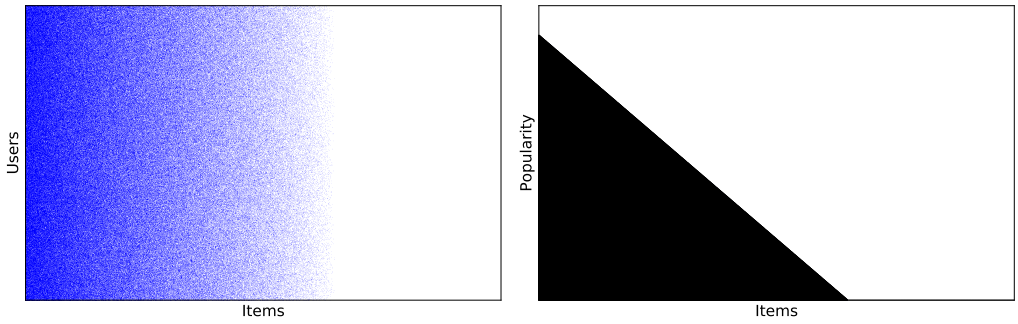


(a) Setting 1 (b) Setting 2

Figure 5.6: Algorithm performances on the *high-low matrix*.

more items left of this type. The performance then dropped, since the remaining items belong to the inferior type. The performance depends on the number of 1-entries belonging to items of the superior type compared to the total number of recommendations the algorithm has to make. The performance gap is highest at the point where no more good-type items are remaining for the stronger algorithms.

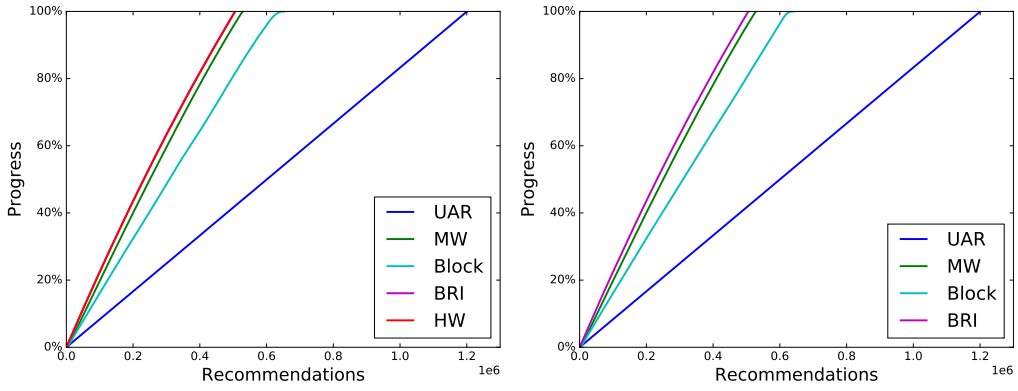
5.1.4 Linear descending Matrix



(a) Preference matrix (b) Likes distribution

Figure 5.7: *Linear descending* matrix

In a *linear descending* matrix (Figure 5.7), the items' popularities drop linearly from item to item. Given p_0 and p_1, $p_0 \geq p_1$, and parameter q, all in $[0, 1]$. The first $\lceil q \cdot m \rceil$ items have a linear number of likes, from $p_0 \cdot n$ linearly descending to $p_1 \cdot n$. The remaining items are liked by no user. The column vectors are chosen from the set of permutations, like in the previous matrix. The performance

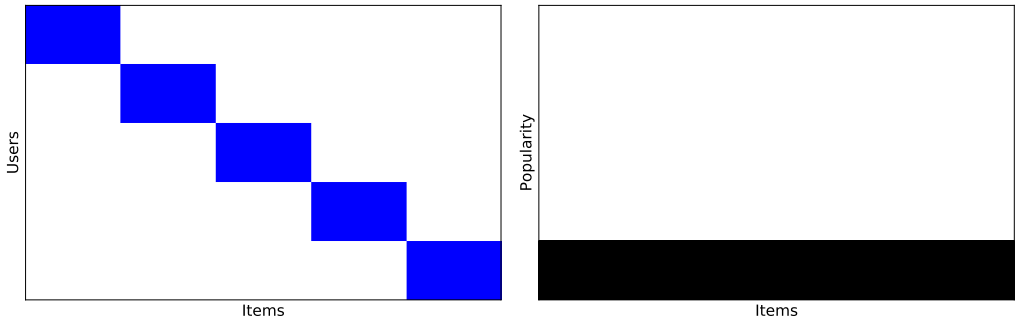


(a) Setting 1 (b) Setting 2

Figure 5.8: Algorithm performances on the *linear descending matrix*.

plots in figure 5.8 show that the MW algorithm performs almost as good as the quasi-offline algorithms, while the block algorithm places between the strong quasi-offline algorithms and the UAR algorithm. We chose $p_0 = 0.9$, $p_1 = 0$ and $q = 0.65$ in this case.

5.1.5 Block Matrix



(a) Preference matrix (b) Likes distribution

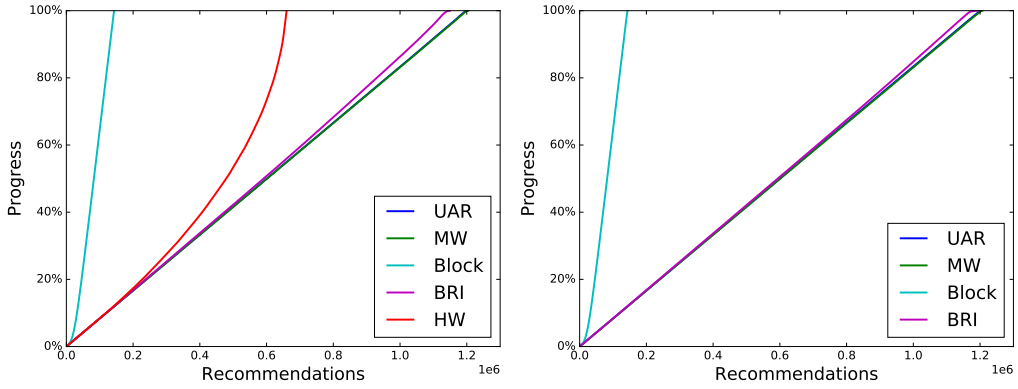
Figure 5.9: *Block* matrix

On the previous matrices the algorithms' performances roughly followed the same pattern; the two quasi-offline algorithms performed best, closely followed by the MW algorithm. The Block algorithm's performance varied, but it usually was somewhere between the UAR algorithm and the other algorithms. *Block* matrices are a type of preference matrices where all algorithms except the Block and HW algorithms perform poorly, i.e., their performance is comparable with the UAR algorithm. The Block algorithm is tailored to perform well on these types of matrices.

The BRI and MW algorithms are good at finding items which are relatively popular. The algorithms are not expected to perform better than the UAR algorithm on matrices where all items are equally popular. Such matrices may still contain a clear structure which can be exploited by algorithms that are designed for these types of data. In fact, similar preference matrices are encountered in real-world. For example, a preference matrix for movies shows a clustered structure since movies can be categorized and users have an affinity for some genres and a dislike for others.

In the extreme case, each user and each item is assigned to exactly one cluster. In other words, a user likes only items from her cluster and an item is only liked by users of its cluster. In addition, we want to make these clusters equally big. A visual representation of such a matrix with 5 clusters is given in figure 5.9. We measured the algorithms' performances on a matrix with 10 blocks. We can see in figure 5.10 that the block algorithm performs significantly better than all

other algorithms. Interestingly, the Harmonic Weights algorithm outperforms the BRI algorithm. The reason can be described as follows. Since all users like equally many items, the weights are the same at the beginning of the Harmonic Weights algorithm. After the first item has been added to the ordering, users of the corresponding cluster have a higher weight, i.e., the Harmonic Weights algorithm appends items of the same cluster to the ordering until all users of the cluster would become satisfied. In the end, the ordering consists of length $\lceil f \cdot l \rceil$ sequences where each sequence consists of items of the same block. l denotes the total number of 1-entries in a user's preference vector, which is the width of a block in the preference matrix.



(a) Setting 1 (b) Setting 2

Figure 5.10: Algorithm performances on the *block matrix*.

5.2 Real-world Preference Matrices

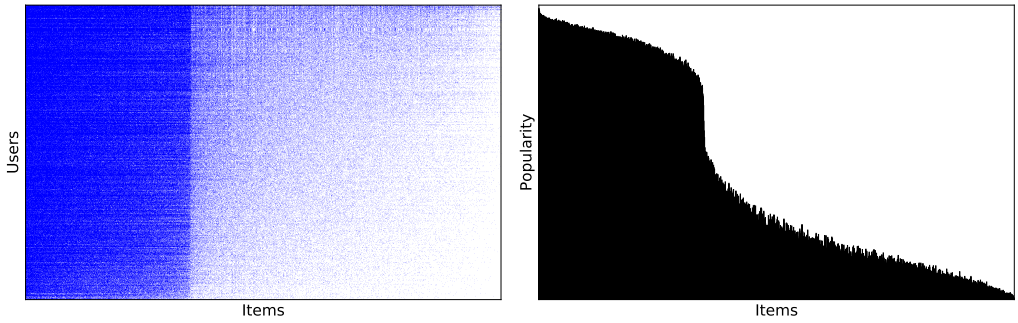
In this section we compare the algorithms on real-world datasets. The biggest issue with real-world data is that the resulting preference matrices are usually sparse. Another issue is that many of these real-world datasets do not use binary ratings. We describe how we create the preference matrices in the corresponding sections.

5.2.1 MovieLens

The MovieLens dataset [9] is a movie dataset made available by GroupLens Research. The dataset consists of 24 million ratings from 260'000 users and 40'000 movies. Although 24 million ratings seems large, the preference matrix is sparse. In fact, 24 million is less than 0.25% of all possible ratings. Matrix completion is an active area of study and there exist many algorithms that try to approximate a matrix as good as possible, i.e., fill the missing values. We

focus only on a subset consisting of the 3000 most popular movies and 800 users who have rated the most movies. This sub-matrix has still missing entries, but it is not that sparse anymore. We apply two methods to create the preference matrix. Once the preference matrix is filled, a last modification needs to be made. Since we work with binary matrices and the ratings of the MovieLens dataset range from 1 to 5, we specify a threshold; ratings below the threshold value are translated to 0, in the other case we write a 1. In both cases we work with a threshold of 3.5. We observe that the resulting preference matrices do not have a cluster-like structure. The main reason for this is that we work only on a small subset of the data.

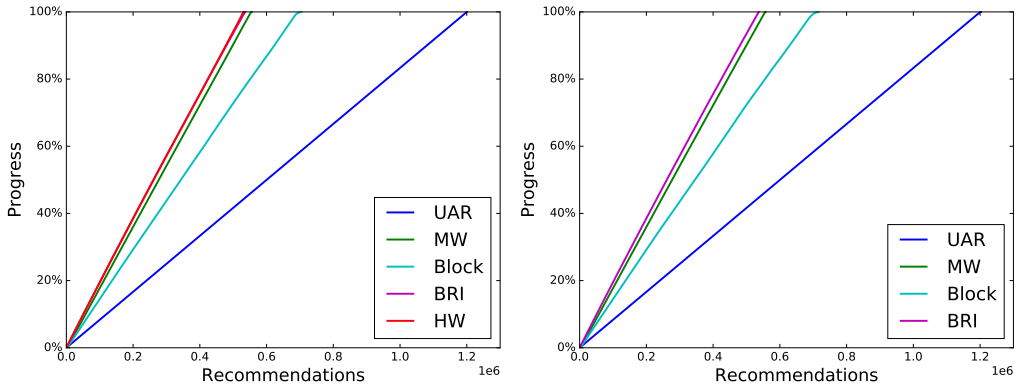
MovieLens: mean-rating



(a) Preference matrix (b) Likes distribution

Figure 5.11: *MovieLens* matrix, mean-rating filled with threshold 3.5.

A very simple approach to fill the missing entries is the average rating of an item. We chose a threshold value of 3.5. The resulting preference matrix is depicted in figure 5.11. The matrix looks similar to a high-low matrix, i.e., there are two types of items that differ by their popularity. Within each type, we see a linearly descending pattern, similar to linear descending matrices. The performance plots in figure 5.12 show that all algorithms except UAR perform well.

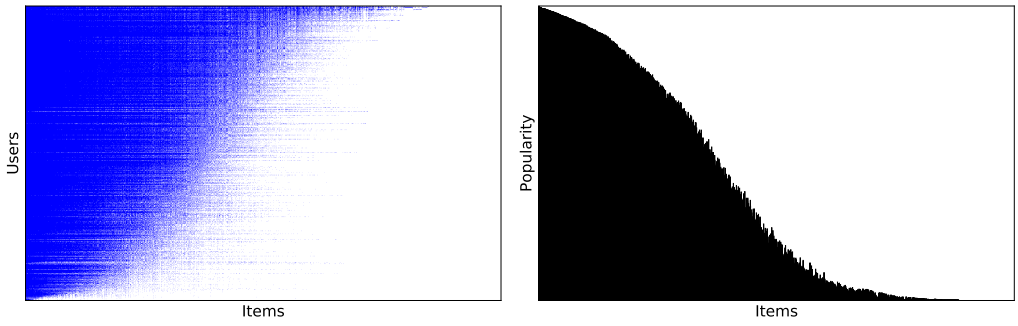


(a) Setting 1 (b) Setting 2

Figure 5.12: Algorithm performances on a subset of the *MovieLens* dataset, missing values filled with mean-ratings.

One disadvantage of filling with mean-ratings is that items are either fairly popular or unpopular. For example, consider a movie with some good ratings, some bad ones, and a majority of users who have not rated the movie. We fill all missing entries of that movie with the same value. The movie becomes either extremely popular or unpopular, depending on what the mean-rating was.

MovieLens: SVD

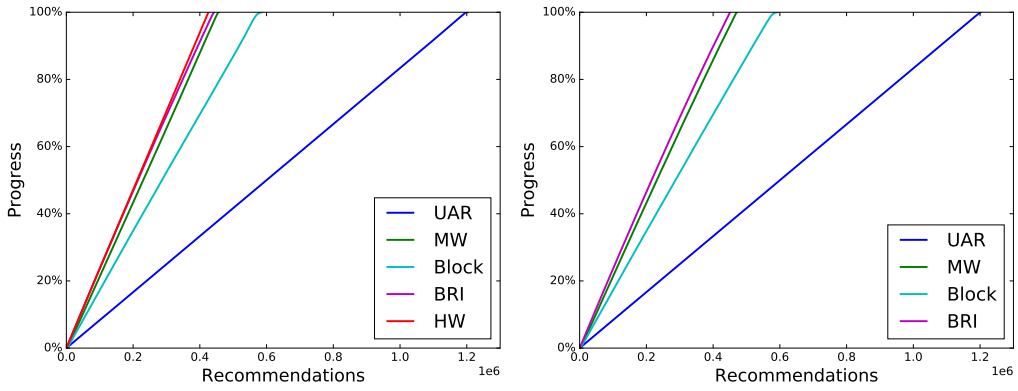


(a) Preference matrix (b) Likes distribution

Figure 5.13: MovieLens matrix, mean-rating pre-filled, SVD filled with threshold 3.5.

The *Singular Value Decomposition (SVD)* is a well known method for matrix dimensionality reduction and reconstruction [12, 7, 11]. SVD is a factorization of a matrix, and a low-rank approximation of the matrix is obtained by selecting the most relevant dimensions. SVD requires the matrix to be filled, and filling

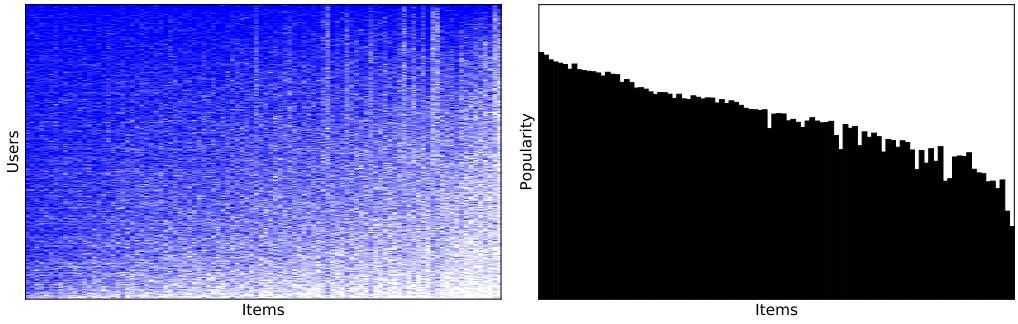
with mean-ratings has proved to be reasonable. Since our matrix is relatively dense, we expect a good approximation for the missing values. We still need to translate this matrix to a binary matrix, but since we do not use the same value for all missing entries of a single movie anymore, we do not get this strong divide into popular and unpopular movies, as depicted in figure 5.13. The preference matrix as well as the performance plots in figure 5.14 look similar to those of the linear descending matrix.



(a) Setting 1 (b) Setting 2

Figure 5.14: Algorithm performances on a subset of the MovieLens dataset. The matrix is an approximation through Singular Value Decomposition on a matrix where missing values were filled with mean-ratings.

5.2.2 Jester

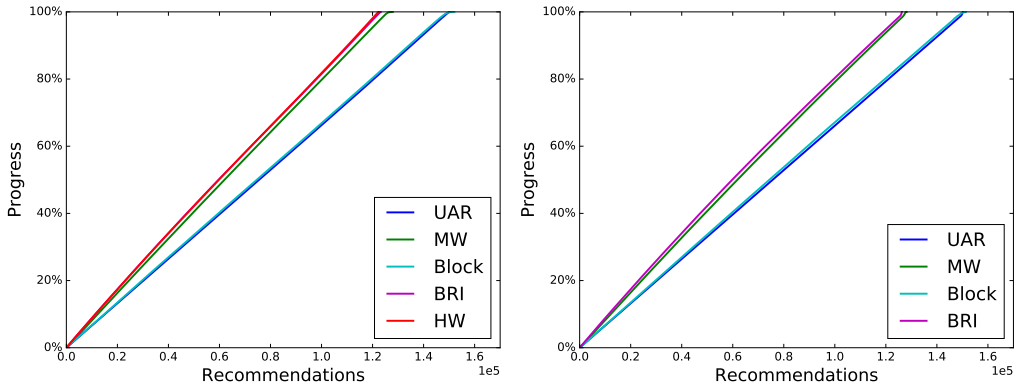


(a) Preference matrix (b) Likes distribution

Figure 5.15: Jester preference matrix, 3000 users rated 100 jokes.

Jester is a joke recommendation system developed at UC Berkeley [6]. In the Jester dataset we worked with, 100 jokes were rated with real values ranging

from -10 to 10. Fortunately, there are many users who have rated all jokes; we randomly selected 3000 users who have rated all jokes for the preference matrix. Ratings from 0 to 10 are translated to a 1-entry in the preference matrix, and ratings below 0 to a 0-entry. Since there are only 100 items, figure 5.15a is scaled accordingly. We see in figure 5.16 that the quasi-offline algorithms and the Multiplicative Weights algorithm perform better than the UAR and Block algorithms. Surprisingly, the Block algorithm's performance comparable with the UAR algorithm. We assume that a user-based approach may perform better with more items.



(a) Setting 1 (b) Setting 2

Figure 5.16: Algorithm performances on a subset of the Jester dataset.

Chapter 6

Summary

We have introduced two settings for the recommendation problem. We presented several algorithms; two quasi-offline algorithms with different information about the preference matrix, an online algorithm which recommends items through sampling, the Multiplicative Weights algorithm which maintains weights on the items and samples items according to these weights, and a third online algorithm which performs well on clustered data. The comparison was made on different types of preference matrices. We observed that the strength of the Multiplicative Weights algorithm lies in detecting items which are more popular than other items. We presented a matrix where the clustering algorithm outperformed all other algorithms; on this matrix, the Multiplicative Weights algorithm performed as bad as the UAR algorithm. Interestingly, the HW quasi-offline algorithm outperformed the BRI quasi-offline algorithm on this matrix. We did not notice any difference in the algorithms' performances on the different settings, where the same total number of recommendations had to be made.

Bibliography

[1] Noga Alon, Baruch Awerbuch, Yossi Azar, and Boaz Patt-Shamir. Tell me who I am: an interactive recommendation system. In *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 1–10. ACM, 2006.

[2] Sanjeev Arora, Elad Hazan, and Satyen Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012.

[3] Yossi Azar, Iftah Gamzu, and Xiaoxin Yin. Multiple intents re-ranking. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 669–678. ACM, 2009.

[4] Nikhil Bansal, Anupam Gupta, and Ravishankar Krishnaswamy. A constant factor approximation algorithm for generalized min-sum set cover. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 1539–1545. Society for Industrial and Applied Mathematics, 2010.

[5] Amotz Bar-Noy, Mihir Bellare, Magnús M Halldórsson, Hadas Shachnai, and Tami Tamir. On chromatic sums and distributed resource allocation. *Information and Computation*, 140(2):183–202, 1998.

[6] UC Berkely. Jester dataset. http://eigentaste.berkeley.edu/dataset/.

[7] Daniel Billsus and Michael J Pazzani. Learning collaborative information filters. In *Icml*, volume 98, pages 46–54, 1998.

[8] Uriel Feige, László Lovász, and Prasad Tetali. Approximating min sum set cover. *Algorithmica*, 40(4):219–234, 2004.

[9] GroupLens. MovieLens dataset. https://grouplens.org/datasets/movielens/.

[10] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Recommendation systems: A probabilistic analysis. In *Foundations of Computer Science, 1998. Proceedings. 39th Annual Symposium on*, pages 664–673. IEEE, 1998.

[11] Michael H Pryor. The effects of singular value decomposition on collaborative filtering. 1998.

[12] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Application of dimensionality reduction in recommender system-a case study. Technical report, DTIC Document, 2000.

[13] Martin Skutella and David P Williamson. A note on the generalized min-sum set cover problem. *Operations Research Letters*, 39(6):433–436, 2011.

[14] Jara Uitto and Roger Wattenhofer. On competitive recommendations. *Theor. Comput. Sci.*, 620(C):4–14, March 2016.