Dynamic Boolean Matrix Factorizations

Pauli Miettinen
Max Planck Institute for Informatics
Saarbrücken, Germany
pauli.miettinen@mpi-inf.mpg.de

***Abstract*—Boolean matrix factorization is a method to decompose a binary matrix into two binary factor matrices. Akin to other matrix factorizations, the factor matrices can be used for various data analysis tasks. Many (if not most) real-world data sets are dynamic, though, meaning that new information is recorded over time. Incorporating this new information into the factorization can require a re-computation of the factorization – something we cannot do if we want to keep our factorization up-to-date after each update.**

This paper proposes a method to dynamically update the Boolean matrix factorization when new data is added to the data base. This method is extended with a mechanism to improve the factorization with a trade-off in speed of computation. The method is tested with a number of real-world and synthetic data sets including studying its efficiency against off-line methods. The results show that with good initialization the proposed online and dynamic methods can beat the state-of-the-art offline Boolean matrix factorization algorithms.

***Keywords*-Boolean matrix factorization; On-line algorithms; Dynamic algorithms**

I. Introduction

Dynamic binary data sets are common in data mining. A popular example is a data base of which movies (or web pages or any other items) users have 'liked' (and signaled this liking by giving good reviews, clicking a button on the web page or by other means we have recorded). When the user likes an item, we add a '1' in our data; the zeros represent the case when user has not liked the item. Note that there are two different reasons why the user has not liked the item: either she does not know about it (and might or might not like it should she know it) or she knows about it and simply does not like it (a third reason is that she knows and likes it, but has not yet expressed her liking – we consider this to be a special case of the first reason). Unfortunately, we typically do not have any means of telling these two reasons apart, and therefore we encode them with the same symbol, 0. This also means that we cannot compute the error only on the known values, as is typical with recommendation systems, for example: if we consider only 1s as known values, any factorization that represents the whole matrix with 1s would be optimal.

What to do with such data? In this paper, we represent the data as a binary matrix and apply the Boolean matrix factorization (BMF) on it. The BMF finds a representation of the given binary matrix as a Boolean product of two (smaller) binary matrices. The restriction to binary matrices and the Boolean matrix product, similar to the normal matrix product, but with addition defined as $1 + 1 = 1$, give BMF some desirable properties, such as interpretability [1] and sparsity [2], although with a cost of increased computational complexity. The factorization can be used for various purposes, ranging from manual data analysis to an input for further data analysis or recommendation algorithms.

II. Background and Basic Definitions

A. Notation

We identify datasets as binary matrices. Matrices are denoted by upper-case bold letters ($\boldsymbol{A}$). Vectors are lower-case bold letters ($\boldsymbol{a}$). If $\boldsymbol{A}$ is an n-by-m binary matrix, $|\boldsymbol{A}|$ denotes the number of 1s in it, i.e. $|\boldsymbol{A}| = \sum_{i,j} a_{ij}$. We extend the same notation to binary vectors. The scalar product of two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is denoted as $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$.

If $\boldsymbol{X}$ and $\boldsymbol{Y}$ are two n-by-m binary matrices, we have the following element-wise matrix operations. The *Boolean sum* $\boldsymbol{X} \vee \boldsymbol{Y}$ is the normal matrix sum with addition defined as $1 + 1 = 1$. The *Boolean subtraction* $\boldsymbol{X} \ominus \boldsymbol{Y}$ is the normal element-wise subtraction with $0 - 1 = 0$. Notice that this does not define an inverse of Boolean sum, as $1 + 1 - 1 = 0$. The *Boolean element-wise product* $\boldsymbol{X} \wedge \boldsymbol{Y}$ is defined as normal element-wise matrix product. The *exclusive or* $\boldsymbol{X} \oplus \boldsymbol{Y}$ is the normal matrix sum with addition defined as $1 + 1 = 0$ (i.e. addition is done over the field $\mathbb{Z}_2$).

Let $\boldsymbol{X}$ be n-by-k and $\boldsymbol{Y}$ be k-by-m binary matrices (i.e. $\boldsymbol{X}$ and $\boldsymbol{Y}$ take values from $\{0, 1\}$). Their *Boolean matrix product*, $\boldsymbol{X} \circ \boldsymbol{Y}$, is the binary matrix $\boldsymbol{Z}$ with $z_{ij} = \bigvee_{l=1}^{k} x_{il} y_{lj}$, that is, Boolean matrix product is the normal matrix product using the Boolean addition.

The *Boolean rank* of an n-by-m binary matrix $\boldsymbol{A}$, $\operatorname{rank}_B(\boldsymbol{A})$, is the least integer k such that there exists an n-by-k binary matrix $\boldsymbol{B}$ and a k-by-m binary matrix $\boldsymbol{C}$ for which $\boldsymbol{A} = \boldsymbol{B} \circ \boldsymbol{C}$. Matrices $\boldsymbol{B}$ and $\boldsymbol{C}$ are the *factor matrices* of $\boldsymbol{A}$, and the pair $(\boldsymbol{B}, \boldsymbol{C})$ is the (exact) *Boolean factorization* of $\boldsymbol{A}$. If $\boldsymbol{A} \neq \boldsymbol{B} \circ \boldsymbol{C}$ (but the dimensions match), the factorization is approximate.

If $p = (i, j)$ is a pair of nonnegative integers and $\boldsymbol{A} = (a_{ij})$ is an n-by-m binary matrix (with $i \leq n$ and $j \leq m$), we write $p \in \boldsymbol{A}$ if $a_{ij} = 1$ and $p \notin \boldsymbol{A}$ otherwise.

Let $\mathtt{s} = (s_1, s_2, \ldots)$ be an ordered sequence of arbitrary items. We use the following slice notation: $\mathtt{s}(1 : n)$ stands for the first n elements of $\mathtt{s}$ while $\mathtt{s}(n\colon)$ stands for the elements of $\mathtt{s}$ from element n onwards, including n.

Now let $\mathtt{s}$ be an ordered sequence of k pairs of positive integers, $\mathtt{s} = \big((i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)\big)$. We define operator $\mathcal{M}_{n\times m}(\mathtt{s})$ to produce an n-by-m binary matrix $\boldsymbol{M} = (m_{ij})$ such that

$$m_{ij} = \begin{cases} 1 & \text{if } (i,j) \in \mathtt{s} \\ 0 & \text{otherwise.} \end{cases}$$

We omit the subscript of $\mathcal{M}$ when it is clear from the context.

B. A Brief Introduction to BMF

In Boolean matrix factorization, the goal is to (approximately) represent a binary matrix as the Boolean product of two binary matrices. The crux is the Boolean product: as the product is not over a field, but over a semiring $(0, 1, \vee, \wedge)$, Boolean matrix factorizations have some unique properties. For example, the Boolean rank of a matrix $\boldsymbol{A}$ can be only a logarithm of the normal matrix rank of $\boldsymbol{A}$ [3]. As a consequence, Boolean factorizations can yield smaller reconstruction error than factorizations of same size done under the normal arithmetic. Unfortunately, unlike normal rank, computing the Boolean rank is NP-hard [4], and even approximation is hard [5] (although recent work shows that logarithmic approximations can be obtained by assuming sparsity [2]).

But even assuming we could compute the Boolean rank efficiently, this is rarely what we actually want. Similarly to normal rank, one would assume that most of the real-world data matrices have full or almost full Boolean rank, due to noise; instead, we often want to have a low-rank approximation of a matrix. Such approximation is usually interpreted to contain the latent structure of the data, while the error it causes is regarded as the noise. When the target rank is given, we have the Boolean matrix factorization problem:

Problem 1 (BMF)**.** Given an n-by-m binary matrix $\boldsymbol{A}$ and integer k, find an n-by-k binary matrix $\boldsymbol{B}$ and a k-by-m binary matrix $\boldsymbol{C}$ such that $\boldsymbol{B}$ and $\boldsymbol{C}$ minimize

$$|\boldsymbol{A} \oplus (\boldsymbol{B} \circ \boldsymbol{C})| \ . \qquad (1)$$

Unsurprisingly, also this optimization problem is NP-hard, and has strong inapproximability results in terms of multiplicative and additive errors (see [1]).

C. The Problem

Informally, the *Dynamic Boolean Matrix Factorization* problem (DBMF) asks us to keep up a factorization of changing data that is a good approximation at any time. For the purposes of this paper, we restrict our attention to *additive* changes to the data, that is, we can only add new 1s, but we cannot remove them. We argue that this is the most common, and most important, case: consider, for example, the aforementioned example where users 'like' items. These likings are usually irreversible: you cannot unlike something you have liked earlier. With other types of data, reversing the action can be even physically impossible – once the user has seen a movie or visited a web page, for example, there is no undoing it. Therefore, we define the problem as follows.

Problem 2 (DBMF)**.** Given an n-by-m binary matrix $\boldsymbol{A}$, its (approximate) rank-k Boolean factorization $(\boldsymbol{B}, \boldsymbol{C})$, and, for any given time t, a prefix $\mathtt{s}(1:t)$ of an unknown sequence $\mathtt{s}$, find a rank-k Boolean factorization $(\boldsymbol{B}_t, \boldsymbol{C}_t)$ that minimizes

$$\big|\big(\boldsymbol{A} \vee \mathcal{M}(\mathtt{s}(1:t))\big) \oplus (\boldsymbol{B}_t \circ \boldsymbol{C}_t)\big| \ . \qquad (2)$$

Notice that DBMF is not a prediction problem. The quality of our result is not based on how well it forecasts the future; it is based on how well it explains the ever-changing present. In principle, we could re-factor the matrix after every addition to answer the problem. In practice, however, that approach would be computationally infeasible. Our goal, therefore, is to find a fast algorithm that can update the existing factorization without completely re-computing it. For that purpose it definitely helps if the original factorization is indeed good at predicting the upcoming additions.

In the definition of Problem 2, both the size of the data matrix and the rank of the factorization are fixed. Both constraints can be relaxed, though. Adding new rows and columns to the data matrix is straight forward, and all algorithms we are going to present can handle that. Allowing the rank of the decomposition to change, however, is a more complex issue. If we let the data size to increase, it does sound reasonable that we let also the rank to increase. But how much should the rank change and how should we define the new rank? We study these problems more in Section III-E.

We can also restrict the factor matrices to additive changes, that is, the algorithm can add new 1s to them, but never remove any. In that case our problem can be seen as an *online problem*: the task is to gradually build a factorization without ever backtracking the decisions. We call this restricted version the *Online BMF* problem.

D. Computational Complexity

The offline version of DBMF, that is, the normal BMF, is computationally hard problem, and its dynamic or online versions are no easier. That is to say, dynamic and online BMF are both NP-hard problems that are NP-hard to approximate well (where we measure the quality of the approximation via the competitive factor against an optimal offline algorithm).

Even some important sub-problems, that are in P in normal linear algebra, are NP-hard with Boolean matrix algebra. Perhaps the most important for this paper is the Basis Usage problem [5]: given a binary vector $\boldsymbol{a}$ and binary matrix $\boldsymbol{B}$, find a binary vector $\boldsymbol{c}$ such that $|\boldsymbol{a} \oplus (\boldsymbol{B} \circ \boldsymbol{c})|$ is minimized. We would like to solve this problem when we are given a new, non-empty column of $\boldsymbol{A}$, $\boldsymbol{a}$. If we could compute the vector $\boldsymbol{c}$, we could simply add that as the last column of $\boldsymbol{C}$. Alas, the Basis Usage problem is NP-hard, too [1].

III. Algorithms

In what follows, we present an algorithm that, given a data matrix and its Boolean factorization, dynamically updates the factorization as new 1s (and new rows and columns) are added. As discussed earlier, we believe this is the most interesting setting. We also think that it is reasonable to expect an initial factorization; after all, most data mining tasks start with some data at hand.

We will start this section by briefly explaining some existing algorithms for solving the BMF. These algorithms are used to compute the initial decomposition which is given to the dynamic algorithm as a starting point. We will then explain the basic online algorithm. Next, we extend the online algorithm to give better factorization, albeit with the cost of increased computational complexity and the loss of the online behaviour. Finally, we discuss on the problem of dynamically changing the rank of the factorization.

A. Algorithms for BMF

`Asso`. We start by explaining how the `Asso` algorithm [5] works. For more detailed explanation, see [5]. The name of `Asso` stems from the algorithm using pairwise association accuracies to generate so-called candidate columns. More precisely, `Asso` generates an n-by-n matrix $\boldsymbol{X} = (x_{ij})$ with $x_{ij} = \langle \boldsymbol{a}_i, \boldsymbol{a}_j \rangle \, / \, \langle \boldsymbol{a}_j, \boldsymbol{a}_j \rangle$, where $\boldsymbol{a}_i$ is the jth row of $\boldsymbol{A}$. That is, x_{ij} is the association accuracy for rule $\boldsymbol{a}_j \Rightarrow \boldsymbol{a}_i$. Matrix $\boldsymbol{X}$ is then rounded to have binary values. The rounding is done from a user-specified threshold $\tau \in (0, 1]$.

The columns of $\boldsymbol{B}$ are selected from the columns of $\boldsymbol{X}$. The selection of columns of $\boldsymbol{B}$ happens in a greedy fashion: each not-used column of rounded $\boldsymbol{X}$ is tried, and the selected column is the one that maximizes the gain, defined being the number of newly-covered 1s of $\boldsymbol{A}$ minus the number of newly-covered 0s of $\boldsymbol{A}$. Element a_{ij} is newly-covered if $(\boldsymbol{B} \circ \boldsymbol{C})_{ij} = 0$ before adding the new column to $\boldsymbol{B}$. The row of $\boldsymbol{C}$ corresponding to the column of $\boldsymbol{B}$ is build using the same technique: if the gain of using the new column of $\boldsymbol{B}$ to cover a column of $\boldsymbol{A}$ is positive, then the corresponding element of the new row of $\boldsymbol{C}$ is set to 1; otherwise it is 0. The gain is computed by the function cover:

$$\begin{aligned} \mathsf{cover}(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}) = &\left|\{(i,j) : a_{ij} = 1 \wedge (\boldsymbol{B} \circ \boldsymbol{C})_{ij} = 1\}\right| \\ &- \left|\{(i,j) : a_{ij} = 0 \wedge (\boldsymbol{B} \circ \boldsymbol{C})_{ij} = 1\}\right| \ . \end{aligned} \tag{3}$$

`Panda`. The second algorithm we present here is the `Panda` algorithm [6]. Similar to `Asso`, it also aims at finding a Boolean factorization of the given binary matrix, but instead of just minimizing the error (1), it tries to minimize the error and the number of 1s in the factor matrices. In other words, `Panda` tries to find factor matrices $\boldsymbol{B}$ and $\boldsymbol{C}$ that minimize

$$|\boldsymbol{A} \oplus (\boldsymbol{B} \circ \boldsymbol{C})| + |\boldsymbol{B}| + |\boldsymbol{C}| \ . \tag{4}$$

The aim of this changed optimization function is to avoid overfitting, i.e. to avoid using the factors to explain what is essentially noise. The `Panda` algorithm can be provided with a maximum number of factors to return, but it can return fewer if it finds that adding more factors would only increase the cost.

The algorithmic approach utilized by `Panda` also differs from that of `Asso` (see [6] for more information): `Panda` starts by finding a frequent itemset, i.e. a noise-free factor called *core pattern*, which it then extends to cover more rows and columns until the cost starts to increase. The algorithm then finds the next core pattern, extends it, and so on.

`Asso+MDL`. The third algorithm we consider is in many ways a combination of the `Asso` algorithm with the idea of `Panda`: `Asso+MDL` [7], [8] uses the Minimum Description Length (MDL) principle [9] to decide the best parameters τ and k for the `Asso` algorithm. The `Asso+MDL` differs from `Panda` in some significant ways, however. First, instead of counting number of errors and 1s in the factor matrices, `Asso+MDL` tries to minimize the number of bits it takes to encode the data matrix exactly using the factor matrices and the error (see [7], [8] for various encoding schemes for doing this). Second, as the name implies, it uses the `Asso` algorithm for finding the factorization. In particular, the `Asso` algorithm tries to find a factorization that minimizes the error, and the MDL part is only used to select the proper number of factors (and the parameter τ). For our algorithm, the Typed XOR DtM encoding method (see [8]) was used, as it was found to perform best in the experiments by Miettinen and Vreeken [8]. For the sake of brevity we will abuse the terminology and call `Asso+MDL` briefly just `MDL`.

B. The Online Algorithm

The first algorithm we present will never backtrack its decisions: if it adds a 1 in one of the factor matrices, it can never remove it. This behaviour makes it suitable for Online BMF, hence the name. In addition to never backtracking, the algorithm does not change the rank of the factorization, either. It is worth noting that once the representation $\boldsymbol{B} \circ \boldsymbol{C}$ has 1 at some location, the online algorithm cannot remove it, as there is no inverse operation of the addition in Boolean algebra.

The algorithm gets as an input the original data matrix $\boldsymbol{A}$, its Boolean factorization $(\boldsymbol{B}, \boldsymbol{C})$, and a sequence s of indices denoting the locations of new 1s. This initial factorization can be provided by any BMF algorithm, and the online algorithm will extend it to handle upcoming 1s as appropriate. There are several cases the algorithm analyses when a new 1 is added. For the following analysis, let t be an arbitrary time step, let $\boldsymbol{A} = \boldsymbol{A}_t$, $\boldsymbol{B} = \boldsymbol{B}_t$, and $\boldsymbol{C} = \boldsymbol{C}_t$ be the configuration of the algorithm at time t, and let $\mathsf{s}(t) = s_t = (i_t, j_t)$ be the index currently under consideration.

Case 1. The added 1 is already covered by the factorization, i.e. $s_t \in \boldsymbol{B} \circ \boldsymbol{C}$. In this case we do nothing.

Case 2. No column of $\boldsymbol{B}$ has 1 at row i_t and no row of $\boldsymbol{C}$ has 1 at column j_t. In this case, we have no previous data that would support the addition of this 1 in any of the factors. Therefore, we do nothing.

Case 3. There exists a column (or columns) in $\boldsymbol{B}$ (and/or row(s) in $\boldsymbol{C}$) that have 1 at row i_t (column j_t). Now we can try to extend the factors to row i_t or column j_t. First, we compute how much the factors would reduce the error if they are extended to the new row or column. Let f be a factor (column of $\boldsymbol{B}$) that has 1 in row i_t (i.e. $b_{i_t f} = 1$). The row f of $\boldsymbol{C}$ must then have 0 at column j_t, or s_t would fall into case 1. To see if we should extend the factor to column j_t, we compute the change in the cover function (3) for column j_t if we set $c_{fj_t} = 1$. We do this for all the factors that have 1 either in row i_t or column j_t and select the factor that yields the highest change in cover. If the highest change of cover is negative, it means we are increasing the error, and we stop. If the highest change is non-negative, we do not increase the error, and we proceed by extending this factor to the new row or column. That is, if f is the factor with highest increase in cover, we set $b_{i_t f} = c_{fj_t} = 1$ (with one of them already being 1).

After we have extended the factor to a new row or column, we see if we can extend it further. Assume we extended the factor f to a new row i_t. We now see if adding this row allows us to extend f to a new column. For this, we study all columns that are not yet included in f (i.e. columns j for which $c_{fj} = 0$) and see if including some column would yield non-negative change in the cover function. After that, we use similar technique to check if we can extend the (possibly extended) factor to new rows. We continue to alternatively extending the factor to new columns and rows until there are no new columns or rows that would give non-negative cover value. The process is guaranteed to end as in each iteration we reduce the error by at least 1.

A pseudo-code for the update method of the on-line algorithm is presented as Algorithm 1. This update method is called for every new index pair (i_t, j_t) and the result is used as the input for the next call.

Algorithm 1 An online algorithm for DBMF

```
Input: A binary matrix $\boldsymbol{A}$, its rank-$k$ Boolean factorization $(\boldsymbol{B}, \boldsymbol{C})$,
    and index pair $(i_t, j_t)$.
Output: Updated factorization $(\boldsymbol{B}_t, \boldsymbol{C}_t)$.
 1: function UpdateOnline($\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, (i_t, j_t)$)
 2:     if $(i_t, j_t) \in \boldsymbol{B} \circ \boldsymbol{C}$ then
 3:         return $\boldsymbol{B}$ and $\boldsymbol{C}$
 4:     else if $b_{i_t f} = c_{f j_t} = 0$ for all $f = 1, \dots, k$ then
 5:         return $\boldsymbol{B}$ and $\boldsymbol{C}$
 6:     else
 7:         set $\boldsymbol{B}_t = \boldsymbol{B}$ and $\boldsymbol{C}_t = \boldsymbol{C}$
 8:         find $f$ s.t. setting $(\boldsymbol{B}_t)_{i_t f} = (\boldsymbol{C}_t)_{f j_t} = 1$ increases
            cover$(\boldsymbol{A}, \boldsymbol{B}_t, \boldsymbol{C}_t)$ the most
 9:         if increase in cover is negative then
10:             return $\boldsymbol{B}$ and $\boldsymbol{C}$
11:         else
12:             set $(\boldsymbol{B}_t)_{i_t f} = (\boldsymbol{C}_t)_{f j_t} = 1$
13:         end if
14:         repeat
15:             try to extend $f$ to new rows and columns
16:         until no new extensions are possible
17:     end if
18:     return $\boldsymbol{B}_t$ and $\boldsymbol{C}_t$
19: end function
```

C. Iterative Updates to the Factors

As the online update method never backtracks its decisions, the effects of wrong decisions start to add up over time. The easiest solution to this problem is to compute a fresh factorization of the data, and start over. In this section we present an in-between solution that is cheaper than computing a completely new factorization, but can still improve the reconstruction error and remove the wrong choices made by the online update.

The idea of the iterative updates is similar to the `Asso+iter` algorithm presented by Miettinen et al. [5]. First, we fix matrix $\boldsymbol{B}$ and iterate over every factor $f = 1, \dots, k$. For each factor f and each column j, we compute if flipping the value of c_{fj} would increase the value of the cover function and flip the value accordingly. After we have processed through the matrix $\boldsymbol{C}$, we fix it and try similarly each value of matrix $\boldsymbol{B}$. We repeat this process until we cannot anymore improve. Again, this process will end, if no earlier, then by latest when there are no error left.

The process of iteratively updating the factors is akin to the way the online update method increases the factors. There are some notable differences, though. First, the online method will never remove any 1s from the factors, and consequently, it does not compute the cover function for those rows and columns that are already included in the factor. Second, the online update method will add a new row or column to the factor if the change to the cover is non-negative, that is, it will add the row or column even if there is no change in the cover (and consequently, in the error). The iterative update method, on the other hand, will only include a row or column if doing so actually reduces the error.

The motivation behind the latter difference is that it allows the online method to extend the factors to new rows and columns as soon as possible, hopefully providing better factorizations along the way. For the iterative update, however, the goal is to remove the spurious 1s that do not decrease the error, thereby – hopefully – providing better starting position for the new round of online updates. This behaviour is most notable when two factors overlap. The online update algorithm will grow both factors, even if one of them already covers some area (almost) completely. The iterative update will select one of the factors to cover the area and remove the overlap where it does not reduce the error.

This iterative process allows the algorithm to revert its decisions (by removing some 1s from the factors) and can yield to significant decrease in error. But it is also an expensive operation, limiting its usability. To balance the (possible) improvements in the error and the increase in computation time, we let the user to define the frequency of the iterative updates. For example, the user could decide that the iterative update procedure should be run after every 10 000 additions.

D. Implementation Details and Time Complexity

There are some implementation details that greatly affect the (practical) time and space complexity of the algorithm. The first is how the matrices are stored. Most (or all) real-world applications to DBMF use very sparse data, so the matrices (data and factor) should be stored using some sparse representation. But due to the nature of the dynamic algorithms, all matrices will be changed during the computation, and the sparse representation should make the updates reasonably effective. For our implementation, we selected a hybrid of list-of-lists and dictionary-of-keys: each column is represented by a set of indices, and the sets are stored in a list. The sets of indices can be implement using standard techniques, for example, cuckoo hashing [10] giving amortized constant addition and deletion times and constant query time. This can be improved even further if one allows for some probability of false 1s by using Bloom filters[1] (see e.g. [11]).

Assuming we store the product $\boldsymbol{B} \circ \boldsymbol{C}$, we can check case 1 of the online update method in constant time and case 2 in time $O(k)$. Computing the change in the cover function if factor f is extended to a new column takes time $O(|\boldsymbol{b}_f|)$, the number of nonzero entries on the fth column of $\boldsymbol{B}$ (again assuming we store $\boldsymbol{B} \circ \boldsymbol{C}$). Thus, in the worst case finding the factor that increases cover most takes $O\left(\sum_{f=1}^{k} \max(|\boldsymbol{b}_f|, |(\boldsymbol{c}^T)_f|)\right) = O(|\boldsymbol{B}| + |\boldsymbol{C}|)$ time.

Extending the factors can be very costly, though. For n-by-m data matrix, computing the cover values for factor f and each data column can take $O(m\,|\boldsymbol{b}_f|)$ time. As this computation might need to be done after (almost) each addition, and for rows too, we need to find a way to make it faster. To that end, we do some more clever bookkeeping. When initializing the algorithm, we compute and store the cover values for each factor and each row and column of the data not included in the factor. Along these values, we store the timestamp when we computed the value and we order them in decreasing order by the cover value.

At each time step (addition), the cover value can change by at most 1. Thus, when searching for the columns or rows to extend the factor, we traverse this list of pre-computed cover values, starting from the largest one. If c is the value stored and Δt is the difference in time between when the value was computed and the current time, we consider this column (or row) only if $c + \Delta t \geq 0$. If the condition holds, we compute the actual cover value: if it is non-negative, the column (or row) is added into the factor; if it is negative, the value along with the updated timestamp is put back to the list. After we see the first column (or row) for which $c + \Delta t < 0$, we know that no more columns (rows) can have non-negative cover value. After the first iteration, the list is kept sorted by $c + \Delta t$.

At first glance it might look like keeping the list sorted is going to be very expensive ($O(m \log m)$ for columns). Note, however, that we only need to move the elements for which we re-compute the cover. As moving these elements in a pre-ordered list is (in practice) much faster than $O(m)$ (they rarely move to the very end of the list), this bookkeeping actually saves us considerable time.

Iteratively updating the factors is more expensive operation, as stated earlier. For fixed $\boldsymbol{B}$, the time complexity to update $\boldsymbol{C}$ is $O(m\,|\boldsymbol{B}|)$. We cannot use the pre-computed cover values, as removing 1s from the factors breaks the bound for maximum change. Rather, we have to re-compute the values for each row and each column and each factor. After the iterative update ends, we store these values as new pre-computed values, to be used by the online update method.

E. Changing the Rank

The final question is, what to do if we want to change the rank of the factorization during the updates. This has many problems: when to change the rank; if new factor is added, which rows and columns should be included; if a factor is removed, what to do with the ones only that factor covered; and so on.

For choosing when to change the rank we could use the minimum description length principle: we should increase the rank if doing so decreases the encoding length of the data, and if reducing the rank decreases the encoding length, we should do that instead. The problem is that before we know how the encoding length changes, we must know how to add or remove a factor – something that can easily become a very slow process. Therefore we do not consider changing the rank in this paper. If that is needed, it should be done when a new factorization is computed from the scratch.

IV. Experimental Evaluation

We tested our algorithm using both synthetic and real-world data. Before going to the results, we present the algorithms and error measures we used in the experiments.

A. Algorithms and Error Measures

The main algorithm was the same online update algorithm for each test. In some tests, we used the iterative updated to the factors. In addition, we varied the algorithm used to compute the initial factorization. We used four methods and named the different variants based on the

[1]We can use Bloom filters as the online algorithm never removes any 1s from the factors. The iterative update might remove 1s, but it can re-build the Bloom filter after the updates.

initialization algorithm. `Asso` and `Panda` use the respective algorithms for initialization. For `Asso`, the rank was fixed and the algorithm selected the best τ among the given possibilities ($\{0.3, 0.5, 0.7, 0.9\}$ for synthetic data and $\{0.1, 0.2, 0.3, \ldots, 0.9\}$ for real-world data). For `Panda`, we used the same maximum rank as with `Asso`, but the algorithm can return smaller ranks. With synthetic data we also used method called `Opt` that initialized the algorithm using the actual factors used to create the data. The purpose of this method was to remove the effects of bad initial selection. With real-world data we used also the `MDL` algorithm; this algorithm selected both the rank and τ, and could return higher-rank factorizations than the other methods.

The motivation behind using both `Panda` and `MDL` was that both methods are supposed to combat overfitting. As overfitting can seriously harm the online update method (as it cannot remove the extra 1s introduced in the initialization) we wanted to see if either of these methods could provide better initial solutions than the `Asso`. It should be noted, however, that as these methods do not (directly) aim at minimizing the error, they can provide initial solutions with very high error; it is only after a number of additions we can say whether the initial solutions were good or bad.

To measure the quality of the factorizations we used the simple *reconstruction error* (2) at the end of the input sequence. In addition, we also computed the *relative error*: if e_i is the error caused by the initialization, e_f is the final error, and $|\mathsf{s}|$ the length of the input sequence, the relative error is defined as $(e_f - e_i)/\,|\mathsf{s}|$. That is, the relative error explains how much error we do, on average, per each addition.

We also computed the offline factorization, that is, the factorization of the data after the full sequence of additions has been applied. With `Asso` algorithm, we used this to compute the *empirical competitive factor*: the reconstruction error of the dynamic method divided by the reconstruction error of the off-line method. Note that we cannot compute the true competitive factor as even the offline version of the problem is NP-hard.

Finally, it is worth noting that the problem setting in this paper differs from those typical in machine learning, as the goal here is not to predict but to adapt. Therefore, we cannot use separate testing data to assess the quality of our algorithm.

For `Asso`, `MDL`, and `Panda`, we used the implementations freely available from the authors of [5], [7], [8], and [6], respectively. The online update algorithm and the iterative factor update algorithm were implemented using Matlab and Python and the source code together with the synthetic data generators is freely available for research purposes.[2]

B. Synthetic Data

The purpose of the synthetic experiments is to test the effects various data characteristics have on the algorithms in a controlled manner. We studied three characteristics: the Boolean rank of the factorization, the density of the data, and the amount of data revealed at the initialization phase. All synthetic matrices were 500-by-700 and for each data point, we generated 10 random binary matrices with identical parameters. In figures, we report the mean over these 10 matrices as well as the standard deviation. The rank parameter was set to the correct rank of the synthetic data. The iterative updates, if used, were done after each $1\,000$ additions.

1) Rank: The rank of the matrices varied from 5 to 25 in steps of 5. The data was generated so that the expected density of the data matrix was 10%. The results are in Figures 1(a) (absolute reconstruction error) and 2(a) (relative error).

The first thing to note from Figure 1(a) is that `Opt` and `Asso+iter` are perfect, and `Panda+iter` is very close of being perfect. `Asso` with only online updates performs quite well, too, being ever so slightly worse than off-line `Asso`. Initializing with `Panda` without using the iterative updates is considerably worse, and the offline `Panda` is the worst, which is no surprise, given that it does not try to minimize the error itself. On the relative error, `Panda+iter` obtains the best improvement (with $k = 25$, each addition results on average an improvement of 15 in the error). Such high numbers are explained by the very high starting error, and also the online `Panda` obtains good relative error. Finally, it should be noted that any relative error below 0 should be considered very good, as it shows that the dynamic method is able to reduce the error compared to the initialization.

2) Density: When we vary the density of the data, from 1% to 30% (Figures 1(b) and 2(b)), we can observe very similar behaviour as with varying the rank. Indeed, the most notable difference is that increasing the density has a notable effect on the `Panda`'s reconstruction error (both offline and online without iterative updates), but almost no effect to the rest of the tested methods.

3) Sequence length: The results with the varying input sequence length (Figures 1(c) and 2(c)) mirror the other results: methods initialized with `Asso` obtain very good results, while `Panda` without iterative updates yields much higher absolute error and much smaller relative error. And again initialization with `Panda` and using the iterative updates gives perfect results.

4) Conclusions: The synthetic experiments suggest that the methods initialized with `Asso` or using the iterative updates are very robust to different data characteristics. The poor performance of offline `Panda` is to be expected as it does not try to minimize the reconstruction error, but the also-poor performance of `Panda`-initialized online update method was more surprising. Given the considerably improved results when the iterative updates are used, we assume that the initial factorization provided by `Panda` is not very suitable for online updates.

[2] http://www.mpi-inf.mpg.de/~pmiettin/dbmf/

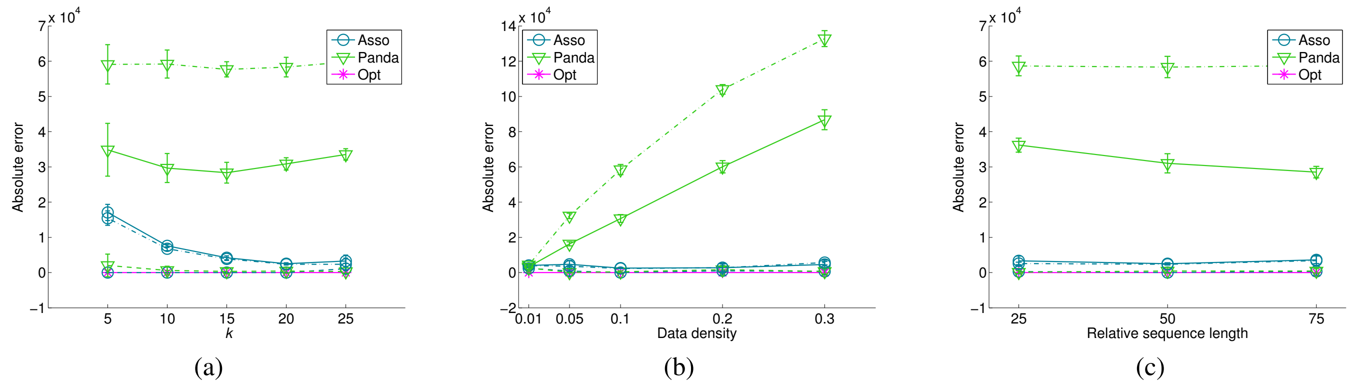

Figure 1. Results for synthetic data using absolute errors. (a) Using different rank k. (b) Using different data density. (c) Using different length of on-line sequence. Solid line represent the online algorithm; dashed line represents the online algorithm with iterated updates; dashed-and-dotted line shows the results with the offline algorithm. The markers show the mean over ten random samples and the width of the error bars is twice the standard deviation.

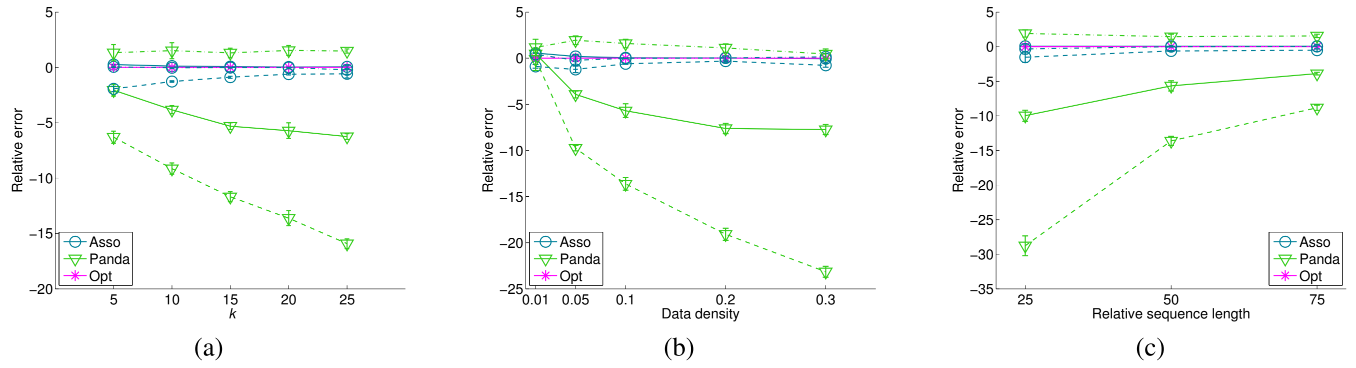

Figure 2. Results for synthetic data using relative errors. (a) Using different rank k. (b) Using different data density. (c) Using different length of on-line sequence. Solid line represent the online algorithm; dashed line represents the online algorithm with iterated updates; dashed-and-dotted line shows the results with the offline algorithm. The markers show the mean over ten random samples and the width of the error bars is twice the standard deviation.

C. *Real-World Data*

1) Data Sets: For real-world experiments, we used three timestamped data sets from the HetRec 2011 collection.[3]

I. The `Delicious` data[4] contains information about which bookmarks users have tagged. As each user can tag a bookmark with multiple tags, we only considered the first tag each user gave to each bookmark. After removing all bookmarks with less than five tags and all users with less than 3 bookmarks tagged, we were left with a 1053-by-1203 binary matrix with 7 717 ones (density 0.6%).

II. The `LastFM` data[5] contains information about which artist which user has tagged. We again removed repeated tags and artists with less than five tags and users with less than five tagged artists. This left us with a 1348-by-3708 binary matrix with 53 676 ones (density 1%).

III. The `Movielens` data[6] contains information about which user has rated which movie. We removed movies with less than ten reviews and users with less than five movies reviewed to get a 2113-by-6829 matrix with 841 910 ones (density 5.8%).

We generated three variations of each of these three data sets. As each of the items (ones) in the data sets comes with a timestamp, we ordered them by time and generated the three variations by retaining $1/3$, $1/2$, and $2/3$ of the ones in the initial data matrix. The rest were put into the input sequence.

When iterative update of factors was used, it was triggered after every 1 000 (for `Delicious`), 10 000 (for `LastFM`), or 100 000 (for `Movielens`) additions.

2) Behaviour of the Error over Time: We start by examining how the reconstruction error behaves as we add more and more ones to the data. For this, we report the results with `Movielens` data and $2/3$ of the ones in the initial matrix using `Asso` as the initialization method and updating the factors iteratively after every 100 000 additions. The results can be seen in Figure 3.

[3]http://www.grouplens.org/node/462
[4]http://www.delicious.com
[5]http://www.lastfm.com
[6]www.grouplens.org, www.imdb.com, and http://www.rottentomatoes.com

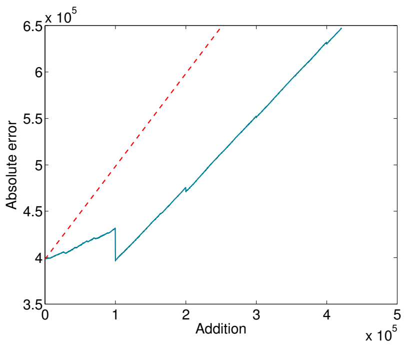

Figure 3. The behaviour of the error in each iteration for Movielens data with 2/3 of the data given initially. The algorithm used is Asso with iterative updates in every 100 000 additions. The dashed line gives the reference where every addition adds one more error.

During the first 100 000 additions, the algorithm adds very moderate amounts of error per added 1. The first iterative update of the factors yields in great reduction of the error, pushing it below the initial level. After that, however, the error curve becomes steeper, although the gradient is still less than 1 (the dashed red line shows a line with gradient 1). The subsequent iterative updates give only small improvements on the overall error.

It seems obvious that during the first 100 000 additions the initial factorization was generalizing rather well. The first iterative update seems to have destroyed some of this generalization, but at the same time it removed all the error caused by the first 100 000 additions, and more. This shows that while the iterative updates can provide a major improvement on the reconstruction error, they should not be computed too often.

3) Numerical Results: The main body of the results with the real-world data is gathered in Tables I, II, and III. Table I gives the absolute reconstruction error, Table II gives the relative error[7], and Table III gives the empirical competitive ratio for the Asso.

The first thing to note from Table I is that the errors (mainly) behave as expected: the more there is data in the initialization matrix (and the shorter the input sequence), the smaller the reconstruction error, and using the iterative updates always improves the results, sometimes substantially, sometimes not so. The first exception to this rule is MDL which, with Delicious and Movielens data, has smallest reconstruction error with half of the data in initialization. This is explained by the fact that the rank MDL uses varies from data version to data version. In these case, the data with small reconstruction error is also the data with high rank. More interestingly, with Movielens data, online Asso and Asso+iter are better than the offline Asso (the same holds true for Panda, too, but again, Panda aims at

[7]The results with MDL+iter on Movielens data did not finish in reasonable time due to the extensive number of factors used by MDL.

Table I

Results for real-wold data sets, absolute error. '+iter' means that iterative updates were used with the DBMF algorithm and the initialization was the same as in the line above.

		Fraction of 1s given			
Data	Algorithm	1/3	1/2	2/3	All
I	Asso	7201	7115	7068	6935
	+iter	7133	7060	7028	
	MDL	7629	7420	7545	7629
	+iter	7583	7376	7514	
	Panda	7717	7717	7717	7717
	+iter	7717	7717	7717	
II	Asso	45383	44264	43267	42004
	+iter	44613	43885	43258	
	MDL	31433	26170	17741	14014
	+iter	26355	22135	15926	
	Panda	51331	51897	51509	51599
	+iter	48858	50755	50280	
III	Asso	678482	672673	675565	682877
	+iter	658573	647866	646691	
	MDL	416735	329884	675846	670310
	+iter	—	—	631218	
	Panda	684139	690042	699955	717325
	+iter	643362	643192	643811	

minimizing different error). This means that we can obtain better reconstruction error by using the online algorithm than what we can by using the offline algorithm!

In absolute terms, MDL+iter obtains the smallest reconstruction error; a testament to the power of selecting the rank using the minimum description length principle.

Considering the relative errors in Table II, we see that with the small-and-sparse Delicious the relative error with all methods is almost 1, i.e. every added 1 adds almost one more error on average. The results are better with LastFM, especially for MDL+iter. With the largest and densest data set, Movielens, all methods are producing rather good relative error. Perhaps the (relatively) higher density allows the initial factorizations to find good structures that generalize.

The final table, Table III, gives the empirical competitive factor for the Asso-initialized methods. Overall, we see that the dynamic methods are very competitive with the offline method, being even better with Movielens data.

4) Scalability: Table IV shows the computation time (in seconds) for computing the updates for the full sequence (excluding the time used for the initialization) and the time for computing the factorization for the full matrix.

As we can see, computing the factorization of the full matrix always takes longer than computing the online update for the full sequence, but doing the iterative updates causes a severe hit on the speed and scalability. On the other hand, if one would compute the full factorization for each update, the resulting time would be much larger than any of the reported times.

Table II
RESULTS FOR REAL-WOLD DATA SETS, RELATIVE ERROR. '+ITER' MEANS THAT ITERATIVE UPDATES WERE USED WITH THE DBMF ALGORITHM AND THE INITIALIZATION WAS THE SAME AS IN THE LINE ABOVE.

		Fraction of 1s given		
Data	Algorithm	1/3	1/2	2/3
I	Asso	0.9821	0.9733	0.9631
	+iter	0.9689	0.9590	0.9475
	MDL	0.9994	0.9876	0.9969
	+iter	0.9905	0.9762	0.9848
	Panda	0.9994	1.0000	1.0000
	+iter	0.9994	1.0000	1.0000
II	Asso	0.9613	0.9721	0.9591
	+iter	0.9397	0.9579	0.9586
	MDL	0.8160	0.8143	0.6431
	+iter	0.6741	0.6640	0.5416
	Panda	0.9662	0.9719	0.9568
	+iter	0.8971	0.9293	0.8881
III	Asso	0.8170	0.8020	0.7985
	+iter	0.7816	0.7431	0.6957
	MDL	0.6716	0.6284	0.7753
	+iter	—	—	0.6163
	Panda	0.7931	0.7897	0.7860
	+iter	0.7204	0.6784	0.5859

Table III
RESULTS FOR REAL-WOLD DATA SETS, ASSO AND EMPIRICAL COMPETITIVE FACTOR. '+ITER' MEANS THAT ITERATIVE UPDATES WERE USED WITH THE DBMF ALGORITHM AND THE INITIALIZATION WAS THE SAME AS IN THE LINE ABOVE.

		Fraction of 1s given		
Data	Algorithm	1/3	1/2	2/3
I	Asso	1.0384	1.0260	1.0192
	+iter	1.0286	1.0180	1.0134
II	Asso	1.0804	1.0538	1.0301
	+iter	1.0621	1.0448	1.0299
III	Asso	0.9936	0.9851	0.9893
	+iter	0.9644	0.9487	0.9470

Table IV
TIMING RESULTS (IN SECONDS) FOR LASTFM AND MOVIELENS WITH ASSO INITIALIZATION. '+ITER' MEANS THAT ITERATIVE UPDATES WERE USED WITH THE DBMF ALGORITHM AND THE INITIALIZATION WAS THE SAME AS IN THE LINE ABOVE.

		Fraction of 1s given			
Data	Algorithm	1/3	1/2	2/3	All
II	Asso	47	24	39	225
	+iter	1610	805	1387	
III	Asso	5683	3868	1954	5818
	+iter	62815	66638	65842	

5) Conclusions: Overall, the results with real-world data are very good. The absolute reconstruction errors might look high, but compared to the error caused by the offline method – the only reasonable comparison point – they are very competitive. That an online method is better than the comparable offline method is rather surprising, but as we have seen, with the heuristics involved here, it is the case. Furthermore, computing the online addition (excluding the initialization) is actually faster than computing the factorization of the full matrix.

V. RELATED WORK

Boolean matrix factorizations have gained interest in data mining community during the past few years. The use of Boolean matrix factorizations in data mining was proposed in [5], although related concepts, such as tiles and formal concepts, were studied much earlier. *Tiling* a database [12] refers to the task of covering all 1s of a binary matrix using few[8] itemsets. The Boolean matrix factorization can be seen as a generalization of this task, each rank-1 binary matrix defining a 'tile'. The difference is that tiling does not allow any 0s to be represented as 1s, whereas the Boolean matrix factorization allows this type of errors. Before that, Boolean matrix factorizations were mostly studied by combinatorics; see [3] and references therein. For some applications and variations of Boolean matrix factorizations, see [1].

Boolean matrix factorization is not the only type of matrix factorization dealing with binary matrices. Methods using normal algebra [13] or probabilistic modeling [14], [15], for example, have been proposed. The characteristics and behaviour of such methods are very different to Boolean matrix factorization, though.

Extending a matrix factorization is a common problem in Information Retrieval (IR) when latent factor models, such as Latent Semantic Indexing [16], are used. These models represent the given corpus as a (non-negative) matrix, and apply a factorization on it. When a new document arrives to the corpus, it has to be *fold in*. The folding-in is performed by projecting the document vector into the lower-dimensional latent factor space (e.g. by multiplying it with the inverse of one of the factor matrices). As noted earlier, this folding-in is an NP-hard problem with the Boolean matrix factorization.

Recently Saha and Sindhwani [17] proposed an algorithm for dynamic non-negative matrix factorization. Their method, as most of those in IR, allows adding new rows and columns (typically, terms and documents), but not changing the already-observed values. This restriction makes sense in the framework of IR, as the contents of the documents rarely gets changed. Our setup, however, asks specifically for handling the changes in the already-factored part of the matrix.

[8]When the goal is to cover all 1s and minimize the number of tiles, it is equivalent to computing the Boolean rank [2]; when the number of tiles is given and the goal is to minimize the number of uncovered 1s, the problem is more akin to standard Boolean matrix factorization.

While this paper is, to the best of the author's knowledge, first to address the problem of dynamic (and online) Boolean matrix factorization, similar problems, such as clustering, have already been studied (see, for example, [18] and citations therein). The differences between Boolean matrix factorization and biclustering, say, are however so fundamental that the methods proposed in that line of work do not seem to be applicable in dynamic and online BMF.

VI. Conclusions

We have presented algorithms for doing dynamic and online Boolean matrix factorizations. As already the offline version is NP-hard even to approximate well, out algorithm is naturally a heuristic. Yet, our tests with real-world data show that the dynamic and online algorithms are, not only faster, but in some cases more accurate than their offline counterparts. The fact that we can keep the reconstruction error low with a fast dynamic algorithm should also help on adapting the Boolean matrix factorization for many new problems, such as social netwok analysis and recommendation systems.

In this paper we considered the dynamic model with only additive changes and without changing the rank of the factorization. While we consider this being the most important case, developing algorithms that can efficiently handle the removing of 1s or change the rank on-the-fly is an interesting and important future problem.

The methods proposed in this paper can also be used to compute a dynamic tiling of a data base by changing requirement that the factor is only extended to a new row (or column) if that induces no error. As the tiling algorithm in [12] uses the set cover algorithm, it might be more effective to dynamically update the set system and then use dynamic set cover algorithm [19] to update the tiling.

References

[1] P. Miettinen, "Matrix Decomposition Methods for Data Mining: Computational Complexity and Algorithms," Ph.D. dissertation, Department of Computer Science, University of Helsinki, 2009.

[2] ——, "Sparse Boolean Matrix Factorizations," in *ICDM '10*, 2010, pp. 935–940.

[3] S. D. Monson, N. J. Pullman, and R. Rees, "A Survey of Clique and Biclique Coverings and Factorizations of $(0,1)$-Matrices," *Bull. ICA*, vol. 14, pp. 17–86, 1995.

[4] D. S. Nau, G. Markowsky, M. A. Woodbury, and D. B. Amos, "A Mathematical Analysis of Human Leukocyte Antigen Serology," *Math. Biosci.*, vol. 40, pp. 243–270, 1978.

[5] P. Miettinen, T. Mielikäinen, A. Gionis, G. Das, and H. Mannila, "The Discrete Basis Problem," *IEEE TKDE*, vol. 20, no. 10, pp. 1348–1362, Oct. 2008.

[6] C. Lucchese, S. Orlando, and R. Perego, "Mining Top-K Patterns from Binary Datasets in presence of Noise," in *SDM '10*, 2010, pp. 165–176.

[7] P. Miettinen and J. Vreeken, "Model Order Selection for Boolean Matrix Factorization," in *KDD '11*, 2011, pp. 51–59.

[8] ——, "MDL4BMF: Minimum Description Length for Boolean Matrix Factorization," Max-Planck-Institut für Informatik, Tech. Rep. MPI-I-2012-5-001, Jun. 2012.

[9] J. Rissanen, "Modeling by shortest data description," *Automatica*, vol. 14, no. 5, pp. 465–471, Sep. 1978.

[10] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithm*, vol. 51, no. 2, pp. 122–144, 2004.

[11] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[12] F. Geerts, B. Goethals, and T. Mielikäinen, "Tiling databases," in *DS '04*, 2004, pp. 77–122.

[13] Z.-Y. Zhang, T. Li, C. Ding, X.-W. Ren, and X.-S. Zhang, "Binary matrix factorization for analyzing gene expression data," *Data Min. Knowl. Discov.*, vol. 20, no. 1, pp. 28–52, 2010.

[14] E. Bingham, A. Kabán, and M. Fortelius, "The aspect Bernoulli model: multiple causes of presences and absences," *Pattern Anal. Appl.*, vol. 12, no. 1, pp. 55–78, 2009.

[15] A. Streich, M. Frank, D. Basin, and J. M. Buhmann, "Multi-assignment clustering for Boolean data," in *ICML '09*, 2009.

[16] S. C. Deerwester *et al.*, "Indexing by latent semantic analysis," *J. Am. Soc. Inform. Sci.*, vol. 41, no. 6, pp. 391–407, 1990.

[17] A. Saha and V. Sindhwani, "Learning evolving and emerging topics in social media: A dynamic NMF approach with temporal regularization," in *WSDM '12*, 2012, pp. 693–702.

[18] D. Duan, Y. Li, R. Li, and Z. Lu, "Incremental K-clique clustering in dynamic social networks," *Artif. Intell. Rev.*, May 2011.

[19] J. Chrissis, R. Davis, and D. Miller, "The dynamic set covering problem," *Appl. Math. Model.*, vol. 6, no. 1, pp. 2–6, 1982.