Hashing for Fast Pattern Set Selection

Maiju Karjalainen[0000−0001−5885−3143] (✉) and Pauli Miettinen[0000−0003−2271−316X]

University of Eastern Finland, {firstname.lastname}@uef.fi

Abstract. Pattern set mining, which is the task of finding a good set of patterns instead of all patterns, is a fundamental problem in data mining. Many different definitions of what constitutes a good set have been proposed in recent years. In this paper, we consider the reconstruction error as a proxy measure for the goodness of the set, and concentrate on the adjacent problem of how to find a good set efficiently. We propose a method based on bottom-k hashing for efficiently selecting the set and extend the method for the common case where the patterns might only appear in approximate form in the data. Our approach has applications in tiling databases, Boolean matrix factorization, and redescription mining, among others. We show that our hashing-based approach is significantly faster than the standard greedy algorithm while obtaining almost equally good results in both synthetic and real-world data sets.

Keywords: Hashing · Pattern Set Mining · BMF · Redescription Mining.

1 Introduction

The goal of data mining is to find surprising, new information about the data. A common problem with many methods that find local patterns from the data is that they find *too many* of those patterns—so-called pattern explosion problem. The popular solution to the pattern explosion problem is to switch to *pattern set mining*; that is, to the task of finding a good set of patterns instead of all patterns.

The definition of what is a good set is a topic of active study, and approaches such as MDL [26] or subjective interestingness [5] have been proposed for that. In this paper we approach the problem from a different perspective. We take the reconstruction error as the means of defining a good set, and ask the question: how fast can we find the set of patterns that minimizes the reconstruction error?

The motivation for our approach is twofold: For one, we argue that the reconstruction error is often a good proxy for the more sophisticated pattern set selection methods. Indeed, they all ask the question: how much yet unexplained data this new pattern explains, the main difference being how the "how much" part is defined. Of these, the "plain" reconstruction error is also the fastest to compute, which is why we prefer it.

For two, we argue that selecting the set of patterns is typically slow. This can be shadowed by the fact that *generating* the patterns can be slower and hence

speeding up the task of selecting the patterns does not feel important. But this is often not so straightforward. In many cases, each pattern is independent of the others, and they can be generated efficiently in parallel. With modern multi-core CPUs and GPUs the pattern generation, even for a huge number of patterns, can be very effective. *Selecting* the patterns in the set, however, is inherently sequential process: in a good pattern set every selected pattern depends on the others. The selection of the patterns can easily become the slower part of the process.

To obtain our fast pattern set selection algorithm, we utilize the bottom-k hashing idea [4, 3, 25] using the efficient computation proposed in [23]. While [23] studies, in some sense, the quality of a pattern set, our approach will build the set, and furthermore, we propose a way to handle inexact patterns, that is, patterns that do not appear as such in the data.

In the next section we will formally define our problems and analyze their computational complexity. We will also discuss how our problems model different pattern set mining problems, although we postpone the other related work to Sect. 5. We will end Sect. 2 with a brief analysis of the standard greedy algorithm before introducing our algorithm in Sect. 3. The experiments are presented in Sect. 4.

2 Notation, Problems, and Theoretical Analysis

In this paper, we study the problem of selecting a good subset of patterns to explain the data. There are two variants: in *exact patterns* the patterns are exact subsets of the data; in *inexact patterns* the patterns can cover things that are not in the data. We formalise the problems using binary matrices. For that, we need the following concepts. We denote a matrix (Boolean or otherwise) with a bold-face upper-case letter, such as $\mathbf{M}$. Vectors are denoted with bold-face lower-case letters, such as $\mathbf{v}$. All vectors are considered to be column vectors. The (i, j) element of matrix $\mathbf{M}$ is m_{ij} and the ith element of vector $\mathbf{v}$ is v_i. We say that a binary matrix $\mathbf{S} \in \{0,1\}^{m \times n}$ is *dominated by* a binary matrix $\mathbf{D} \in \{0,1\}^{m \times n}$ if $s_{ij} \leq d_{ij}$ for all i and j. We denote that by $\mathbf{S} \leq \mathbf{D}$. Given a collection of binary matrices $\mathcal{C} = \{\mathbf{S}_i : i = 1, \ldots, \ell, \mathbf{S}_i \in \{0,1\}^{m \times n}\}$, we define $\bigvee_{\mathbf{S} \in \mathcal{C}} \mathbf{S} = \mathbf{C}$ as the element-wise logical *or* over matrices $\mathbf{S}$, that is, $c_{ij} = \bigvee_{\mathbf{S} \in \mathcal{C}} s_{ij}$. Matrix $\mathbf{M}$, of size m-by-n, is *rank-1* if it can be expressed as an outer product of two vectors, $\mathbf{M} = \mathbf{u}\mathbf{v}^T$. Matrix $\mathbf{M} \in \{0,1\}^{m \times n}$ is *Boolean rank-1* if $\mathbf{M} = \mathbf{u}\mathbf{v}^T$ where $\mathbf{u} \in \{0,1\}^m$ and $\mathbf{v} \in \{0,1\}^n$. Finally, we use $|\cdot|$ to denote the number of non-zero entries in a matrix or a vector, that is, $|\mathbf{M}| = |\{(i, j) : m_{ij} \neq 0\}|$, where the latter $|\cdot|$ is the usual set cardinality operator.

2.1 Problem Statement

We give the exact definitions of our problems below. For an explanation of how these definitions relate to pattern set mining tasks, see Section 2.3.

Problem 1 (Exact Patterns). Given a binary data matrix $\mathbf{D} \in \{0,1\}^{m \times n}$, a collection of (rank-1) binary matrices of the same size, $\mathcal{S} = \{\mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_\ell : \mathbf{S}_i \in \{0,1\}^{m \times n}, \mathbf{S}_i \leq \mathbf{D}\}$, and an integer k, find a subcollection $\mathcal{C} \subseteq \mathcal{S}$ of size k that minimizes

$$\left| \mathbf{D} - \bigvee_{\mathbf{S} \in \mathcal{C}} \mathbf{S} \right| . \tag{1}$$

Alternatively, the size of collection $\mathcal{C}$ can be unlimited and (1) can be replaced with a requirement that

$$\bigvee_{\mathbf{S} \in \mathcal{C}} \mathbf{S} = \mathbf{D} , \tag{2}$$

where we now assume that $\cup\mathcal{S} = \bigcup_{\mathbf{S} \in \mathcal{S}} \mathbf{S} = \mathbf{D}$. Notice that as $\mathbf{S}$ are dominated by $\mathbf{D}$, the union can never have 1s in locations where $\mathbf{D}$ does not have them. Consequently, the subtraction in (1) cannot produce negative entries. We say that $\cup\mathcal{S}$ does not *cover* any 0s of $\mathbf{D}$.

When the pattern matrices are not dominated by the data, the problem definition is as follows.

Problem 2 (Inexact Patterns). Given a binary data matrix $\mathbf{D} \in \{0,1\}^{m \times n}$ and a collection of (rank-1) binary matrices of the same size, $\mathcal{S} = \{\mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_\ell : \mathbf{S}_i \in \{0,1\}^{m \times n}\}$, find a subcollection $\mathcal{C} \subseteq \mathcal{S}$ that minimizes

$$\left| \mathbf{D} - \bigvee_{\mathbf{S} \in \mathcal{C}} \mathbf{S} \right| . \tag{3}$$

Contrary to (1), in (3) the matrices $\mathbf{S}$ are not necessarily dominated by $\mathbf{D}$ and hence the subtraction can yield negative values. Similarly to Problem 1, it is also possible to limit the size of $\mathcal{C}$ in Problem 2 by some user-defined constant k.

In the above definitions, we have limited the patterns to be rank-1 binary matrices. As we shall see below, this still allows us to represent many different kinds of patterns commonly encountered in data mining. Our algorithm will utilize the rank-1 structure for efficiency; however, we can still handle arbitrary binary matrices as patterns, with a cost in the running time of the algorithm (see Sect. 3.1). We will use terms *pattern*, *tile*, and *rank-1 matrix* interchangeably in this paper.

2.2 Computational Complexity

Both problems (and their variants) are NP-hard. The Max k-Cover problem can be reduced to Problem 1, and the Set Cover problem to the variant of Problem 1 with unlimeted cover size (see, e.g. [11]). The simplest form of this reduction is to consider a case where $\mathbf{D}$ has only one m-dimensional column full of 1s. Each $\mathbf{S}$ is also an m-dimensional column vector and they correspond to incidence vectors of the sets in Set Cover or Max k-Cover. It is trivial to see that solutions that maximize (1) or (2) maximize Max k-Cover or Set Cover, respectively.

In the Positive–Negative Partial Set Cover [18], the input consists of positive and negative elements and a collection of sets of those elements and the goal is

to select a subcollection of the sets that minimizes the sum of covered negative elements plus the sum of uncovered positive elements. Consider again a case where $\mathbf{D}$ is m-dimensional vector. Each entry d_i of the input vector corresponds to an element in the input; $d_i = 1$ if the element is positive and $d_i = 0$ otherwise. The matrices $\mathbf{S}$ are again incidence vectors of the sets. It is straightforward to see that minimizing (3) is the same as minimizing the sum of uncovered 1s plus the sum of covered 0s.

Furthermore, these reductions are approximation-preserving as the values of cost functions (number of used matrices/sets or number of uncovered 1s plus covered 0s/uncovered positive and covered negative elements) are exactly the same after the reductions. Hence, Problem 1 with optimization goal (1) cannot be approximated better than $\Omega(1 - 1/e)$. With optimization goal (2) the approximation lower bound is $\Omega(\ln(m))$ [8]. For Problem 2, the approximation lower bound is much higher $\Omega(2^{\log^{1-\varepsilon} \ell^4})$, where ℓ is the number of sets in $\mathcal{S}$ [18]. The intuition behind the vast difference in inapproximability lower bounds is that in the case of exact patterns, we should aim to select maximally disjoint patterns, whereas in the case of inexact patterns, we should aim to select patterns that cover maximally disjoint set of 1s but maximally similar set of 0s, as we only "pay" the cost of covering a 0 once. This latter problem is clearly much harder.

2.3 Connections to Other Forms of Pattern Set Mining

While we described our problems in the terms of binary matrices, it is relatively easy to see that the description applies to many different pattern set mining tasks. Here we describe three such tasks: tiling databases, Boolean matrix factorization, and redescription mining.

Tiling databases [11] is the task of selecting a subset of (frequent or closed) itemsets to cover all the items in a transaction database. In our setting, the transaction database is the data matrix $\mathbf{D}$ with items as columns and transactions as rows, and each rank-1 matrix is one itemset: the columns with 1s correspond to the items in the itemset and the rows with 1s correspond to the transactions that contain the itemset. As every item of the itemset must appear in a transaction for the transaction to contain the itemset, the patterns are dominated and we have the exact patterns of Problem 1.

Equation (1) is a natural optimization goal for tiling. Equation (2) requires that every item in every transaction is included in some pattern; this is achieved most easily by including the singleton itemsets for each item in the set $\mathcal{S}$. Such exact cover is also required when one uses alternative means of selecting the pattern set, such as MDL [26] (see also Sect. 5).

Boolean matrix factorization [19] is the task of (approximately) expressing a given binary matrix as a Boolean product of two binary factor matrices. Equivalently, it is the task of expressing the input as a element-wise Boolean *or* of rank-1 binary matrices. With this definition, the connection to our problem is clear. As Boolean factorization can be approximated both by not covering some 1s and by covering some 0s, the appropriate formulation is typically that of Problem 2. For

$q_L = ([a \leq 0.4] \wedge [b \geq 0.6]) \vee [d = 1.0]$ $\quad q_R = [x = 1] \vee [y = 1]$

$$\mathbf{D}_L = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 0.3 & 0.7 & 0.2 & 0.9 \\ 0.5 & 0.4 & 0.6 & 0.8 \\ 0.4 & 0.6 & 0.3 & 0.9 \\ 0.3 & 0.5 & 0.3 & 1.0 \end{pmatrix} \end{array} \quad \mathbf{D}_R = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cc} x & y \end{array} \\ \begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \end{array} \quad \mathbf{S} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & x & y \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

Fig. 1. Example of how redescriptions can be expressed in the framework of Problem 1. **Top:** A pair of queries, forming a redescripition. **Bottom left and middle:** Original data matrices $\mathbf{D}_L$ and $\mathbf{D}_R$ for redescription mining. Cells shaded in gray will be 1 in $\mathbf{u}_L\mathbf{v}_L^T$ and $\mathbf{u}_R\mathbf{v}_R^T$. **Bottom right:** The final matrix $\mathbf{S} = (\mathbf{u}_L \wedge \mathbf{u}_R)[\mathbf{v}_L^T, \mathbf{v}_R^T]$.

computing the smallest exact Boolean factorization (so-called *Boolean rank* [21]), however, Problem 1 with optimization goal (2) is the appropriate formulation.

Redescription mining [22, 10] is a data analysis method that, unlike the aforementioned ones, can, and often does, use numerical data. The goal of redescription mining is to mine *redescriptions*, that is, pairs of queries (q_L, q_R) over two sets of data, $\mathbf{D}_L$ and $\mathbf{D}_R$, that provide two different views to the same entities. A classical example of redescription mining is species presence–absence data and climate data. These provide different "views" to the same geographical locations and a good redescription on such data would describe a query (or a rule) over the species (i.e. the presence or absence of certain species) and a query over the climate variables (e.g. temperature or precipitation) such that the two queries hold in approximately same geographic areas. The quality of a redescription is often measured using the Jaccard coefficient between the sets of entities where the individual queries hold.

While the data in redescription mining can be numerical, every individual query can be seen as a rank-1 binary matrix. The 1s in the columns correspond to the attributes that appear in the query; the 1s in the rows correspond to the entities where the query holds. For a pair of queries, we can concatenate these two matrices next to each other (note that they must have the same number of rows) and consider only those rows where both matrices have a 1. That is, if $\mathbf{u}_L\mathbf{v}_L^T$ is the rank-1 matrix corresponding to q_L and $\mathbf{u}_R\mathbf{v}_R^T$ is that for q_R, the rank-1 matrix corresponding to pair (q_L, q_R) is $(\mathbf{u}_L \wedge \mathbf{u}_R)[\mathbf{v}_L^T, \mathbf{v}_R^T]$, where $\wedge$ is the element-wise *and* and the right-hand vectors are concatenated. Figure 1 shows an example of one pair of queries over numerical and Boolean data and the resulting pattern matrix $\mathbf{S}$.

2.4 The Greedy Algorithm

By far the most common algorithm to do the pattern set selection for minimum error case is the greedy algorithm. This algorithm takes first the pattern that minimizes the error the most, then the pattern from the remaining ones that minimizes the error the most given the already selected pattern, and so on and

so forth until it has either selected a predefined number of patterns or no further pattern reduces the error. This approach is used in tiling [11], in BMF [19], and in redescription mining[1] [10] for selecting a set of patterns. For the exact case, the greedy algorithm has approximation ratio of $O(1 - 1/e)$ (for (1)) or $O(\ln(m))$ (for (2)), matching the lower bounds. For inexact patterns, however, the greedy algorithm can be arbitrarily bad, although it seems to work well in practice [19].

The time complexity of the greedy algorithm is $\tilde{O}(mn\ell^2)$ for data matrix of size m-by-n and $|\mathcal{S}| = \ell$: after selecting every new pattern, the algorithm must re-compute how good the remaining patterns are. We use $\tilde{O}(\cdot)$ to denote the time complexity that ignores the logarithmic factors (that are needed to find the best pattern after the updates). For sparse patterns, tighter analysis can be obtained, as updating each pattern takes at most as much time as the number of 1s in the pattern.

3 The Algorithm

Our algorithm, called `HaPSi` (Hashing for Pattern Selection), is based on using hash values to estimate the sizes of rank-1 matrices and their combinations. The main idea, presented in [1] and [23], is to calculate hash values uniformly distributed in $[0, 1]$ for all 1s in a rank-1 matrix, and to use the kth smallest hash value v to estimate the number of 1s in the rank-1 matrix as k/v. This can be used as an estimate, because, if there are z values uniformly distributed in $[0, 1]$, the expected value of the kth smallest value is $k/(z + 1)$. This method is used by [23] to estimate the product of two Boolean matrices, by iterating efficiently over the rank-1 matrices and keeping track of the k smallest hash values seen at each point.[2] Next we will explain a technique for estimating sizes of overlapping tiles before explaining the algorithm in Sect. 3.2. We analyse the time complexity in Sect. 3.3.

3.1 Estimating Overlapping Tile Size

This section describes a hashing-based technique of estimating the number of 1s in a single tile and in a collection of overlapping tiles. It is based on [23], although with some changes to make the approach suitable for our application.

The hash value $h(x, y)$ for a point (x, y) in a rank-1 matrix is computed based on two pairwise independent hash functions $h_1, h_2 \colon U \to [0, 1]$ which map the row and column indices to $[0, 1]$. The hash value for the point is then defined as $h(x, y) = (h_1(x) - h_2(y)) \mod 1$.

To find the k smallest hash values of a rank-1 matrix $\mathbf{u}\mathbf{v}^T$, $h(x, y)$ are arranged into a matrix of size $|\mathbf{u}|$-by-$|\mathbf{v}|$. The rows are ordered by increasing values of h_1 and the columns by increasing values of h_2. Going down a column the values of $h(x, y)$ will increase, except at one point where the value drops to the minimum

[1] In redescription mining the approaches in literature have some technical differences to the approach outlined above.

[2] Notice that this k is different to the number of selected tiles in problem definitions.

value of that column. Similarly on the rows, the values decrease going to the right, except for one increase to the maximum value of that row.

The algorithm is given a value p which serves as an initial maximum hash value to be stored. The values of the hash matrix are visited starting from the top of the left-most column, first finding the smallest value of the column. After that, going through the column, all values that are smaller than p are stored. After all such values have been found, the next column is searched, this time starting one row below the row where the previous column had its smallest value. When k values have been found, if the kth value is smaller than p, p is updated to this new smallest value.

The idea of using the kth smallest hash value as a size estimate can be applied to a matrix of any rank, but the efficiency of this algorithm comes from iterating over the row and column indices which only works on rank-1 matrices. Hence, if we want to use non-rank-1 patterns, we have to give up on this optimization.

3.2 The Algorithm for the Exact Patterns

The first step of our algorithm is to compute the hash values for each tile. This is done as explained above. For pairwise independent hash functions, we use random affine transformations in $\mathbb{Z}/p$, where $p \in \mathbb{N}$ is a prime number larger than largest dimension of the matrix. That is, each hash function is of form $h(x) = (ax + b \mod p)/p$ for some random a and b. Unlike [23], we store the k smallest hash values separately for each tile instead of maintaining a single collection of size k over all tiles.

The pseudocode for `HaPSi` is given in Algorithm 1. The algorithm starts by choosing the largest tile. Then, it goes through rest of the tiles, trying to find the one that increases the number of ones covered the most. For each tile that has not been chosen yet, we concatenate its hash values and those of the already-chosen tiles' in line 9. The combined hash values are sorted and we use the kth smallest value to get the estimated size of these tiles together. In our preliminary experiments we noticed that this method gives generally very accurate estimates, but sometimes gives an exceptionally large or small result. This is why we get the final size estimate as the median over several ($|\mathcal{H}|$, typically 10 in our experiments) estimates (line 12).

When we have the estimated sizes for all tiles, we sort the tiles in a descending order according to the estimate. Then, in line 16, we go through a limited number m (typically 30) of top candidates, and calculate the actual reconstruction error (or the number of 1s in the resulting matrix in the exact case) we would get when adding that tile to the already chosen ones. Once we find a tile that lowers the reconstruction error, we stop the search and add that tile to the collection. The process is repeated until the error no longer improves, or the maximum number of tiles is met (constraint set by the user).

Handling inexact tiles. In the case where the tiles can cover zeros, the first tile chosen is the one with the smallest reconstruction error, meaning it has the least amount of uncovered ones and the least amount of covered zeros. In addition

Algorithm 1 Exact Patterns (Problem 1)

Input: Data matrix $\mathbf{D}$, tiles $\mathcal{T}$, collection of vectors of hashvalues for each tile $\mathcal{H}$, maximum number of tiles returned $t_{\max}$, k for bottom-k hashing, and maximum number of tiles searched in each iteration m

Output: A set of tiles $\mathcal{Q}$

```
1: function HaPSi(T, H, t)
2:     Q ← tile t_0 with the smallest reconstruction error
3:     T ← T \ t_0
4:     while len(Q) < t_max do
5:         C ← ∅
6:         for t ∈ T do
7:             E ← ∅
8:             for h ∈ H[t] do
9:                 h ← Sort(H[Q] ∪ h)[: k]
10:                E ← E ∪ {k/h[k]}
11:            end for
12:            v ← median(E)
13:            C ← C ∪ {(v, t)}
14:        end for
15:        C ← Sort(C) in descending order by v
16:        for (v, c) ∈ C[: m] do
17:            if error(Q ∪ {c}) < error(Q) then
18:                Q ← Q ∪ {c}
19:                break
20:            else
21:                return Q
22:            end if
23:        end for
24:    end while
25:    return Q
26: end function
```

to estimating the combined size with the already chosen tiles, we subtract the number of zeros covered by the tile from the joined size estimate. This means that the line 12 changes to median$(\mathcal{E}) - \mathcal{Z}_t$, where $\mathcal{Z}_t$ is the number of zeros the tile t covers.

We note that by subtracting every 0 from every tile that covers it, we are probably over-penalizing the tiles; if an already-chosen tile covers some of the same 0s as this tile, using this tile does not increase the cost for those 0s. While this approach might seem naïve, the inapproximability of the inexact case indicates that much better algorithms are unlikely to exist.

3.3 Time Complexity

For an input matrix of size m-by-n and number of tiles ℓ, the time complexity of `HaPSi` is $\tilde{O}((n+m)\ell + \ell^2 + mn\ell)$. Computing the hash values for one tile $\mathbf{u}\mathbf{v}^T$ takes amortized time $O(|\mathbf{u}| + |\mathbf{v}|)$ [23], where $|\mathbf{u}|$ is the number of non-zeros in $\mathbf{u}$.

This explains the first term. After each tile has been chosen, we need update how well the others do given the chose tile (the second term), and to choose the tile, we need to compute its exact error (the final term). We could omit calculating the actual error, but we would then be susceptible to cascading errors in size estimates.

4 Experiments

We evaluate the performance of our algorithm on both synthetic and real-world data. The aim of the synthetic data experiments is to assess how different data features affect the algorithm; experiments with real-word data then validate these results.

4.1 Experimental Setup

We generate the synthetic data as follows. First, we make 100 random rank-1 matrices (aka tiles or patterns) of size m-by-n by sampling vectors $\mathbf{u}_i \in \{0,1\}^m$ and $\mathbf{v}_i \in \{0,1\}^n$ from m- and n-dimensional Bernoulli distributions with parameter $p \in (0,1)$. Each "original" tile is $\mathbf{u}_i\mathbf{v}_i^T$. We make 5 copies of each of these tiles, and for each copy $\mathbf{u}'_i\mathbf{v}'^T_i$, we move a fraction d of the 1s from $\mathbf{u}'_i$ and $\mathbf{v}'_i$ to other locations. This gives us 100 mostly non-overlapping rank-1 matrices, each of which have 5 copies that have a given amount overlap with the original and each other.

Then, we have two ways of creating a data matrix out of these 600 patterns, either by multiplying the original 100 patterns, or by multiplying all 600. After this we can add noise to the data matrix by randomly flipping a portion of the elements.

For the exact pattern experiments we only created the data matrix by multiplying all 600 tiles and did not add any noise, to ensure that the tiles are dominated by the data. For the inexact pattern experiments, we used both methods of creating the data matrix and also added noise.

There are four features we test with the synthetic matrices: overlap between the tiles, density of the data matrix and the tiles, size of the matrix, and the added noise. We tested all of these with the inexact patterns, and the first two (overlap and density) also with the exact patterns.

We created the data 5 times for each set of parameters tested, and the results are the average over these five runs. As our own algorithm has some randomness in creating the hash values, we initially tested it with three restarts on the same data. However, since the algorithm uses the median estimate over multiple hash values on a single run, running it more times did not have much variance in the results.

We compared `HaPSi` to two baselines. `Greedy` is the standard greedy algorithm for Set Cover (see Sect. 2.4). `Naïve` is a simpler and faster algorithm that always selects the tile from the unselected tiles that alone covers the most 1s (or reduces the error most, in inexact case), irrespective of what has been selected before.

These two algorithms represent two extremes. `Greedy`, while not being optimal, is the state of the art solution for minimizing the error, while `Naïve` is about as fast as it can get: it's time complexity is $\tilde{O}(nm\ell)$, that is, it only requires one linear scan over the input data.

For all experiments, we report *relative reconstruction error*, that is, reconstruction error divided by the matrix size. The algorithms were implemented on Python and are freely available.[3] The experiments were conducted on a machine with 2 AMD EPYC 7702 processors with 64 cores each. All experiments were run single-threaded.

4.2 Setting the Hyperparameters for `HaPSi`

We evaluated the effect of the hyperparameters of `HaPSi` on the reconstruction error and the running time. The following hyperparameters were tested: number of hash values used in bottom-k hashing, k; number of times the hashing is repeated for the data, $|\mathcal{H}|$; and the maximum number of tiles for which the reconstruction error is computed before stopping, m. The other parameter for Algorithm 1 is the maximum number of tiles returned, $t_{\max}$, but it was not tested here since it only affects the final error and running time.

The default values used for the experiments were $m = 30$, $k = 30$, $|\mathcal{H}| = 10$. The tests were ran on a synthetic data of size 1000-by-1200 with data density 0.06; fraction of rows and columns changed in copied tiles by 0.1 and noise added to the data matrix was 10 %. Each hyperparameter was varied separately, while keeping others constant.

The results of the tests are shown in Fig. 2. The results show that two of the parameters had a noticeable effect on the reconstruction error: $|\mathcal{H}|$, the number of times the hashing was repeated, and the k for the bottom-k hashing method (Fig. 2(a) and (c)). Naturally as the hash values are computed several times, the final estimate (calculated as the median over all of the estimates) will be more accurate. A larger value here also significantly affects the running time. The choice of optimal value of k is explained in detail in [23], and our results show that a larger value improves the error at least to some point. However, we chose not to use a large value of k in our experiments, as in real-world applications the rank-1 matrices can have varying densities; in particular, rank-1 matrices with less than k 1s would be automatically discarded. Another approach for choosing the value of k would be to choose it after all of the rank-1 matrices have had their hash values calculated, and their sizes are known, but testing this is left for future work.

4.3 Results on Synthetic Data

In each test, all other data generation parameters except the one being tested for were kept constant. The default values used in the data generation were as follows. Data size: 1000-by-1200; data density: 0.3; fraction of rows and columns

[3] https://doi.org/10.5281/zenodo.15629578

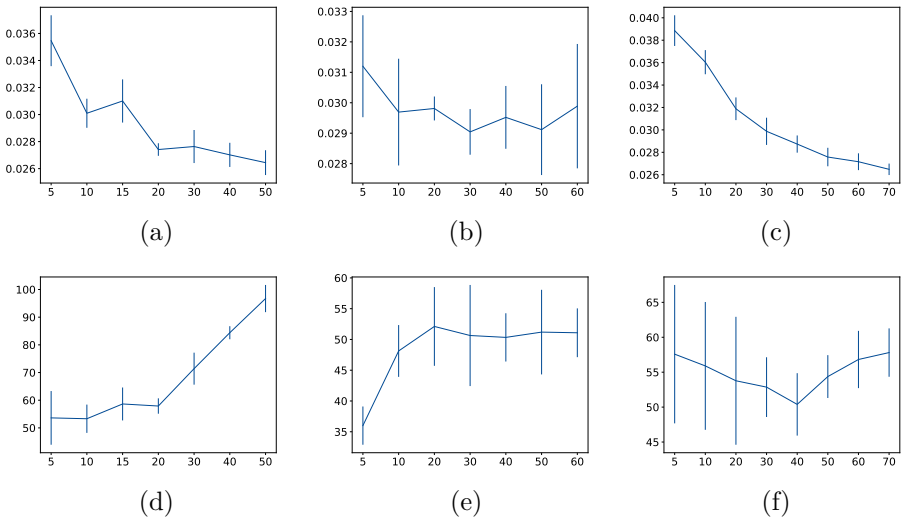

Fig. 2. Reconstruction errors and running times from the hyperparameter tests. **Top row:** x-axis: (a) $|\mathcal{H}|$, number of times the hashing is repeated; (b) m, maximum number of tiles for which the reconstruction error is computed; (c) k, number of hash values. Relative reconstruction error on y-axis. **Bottom row:** Running time in seconds on y-axis. (d)–(f) as (a)–(c).

changed in copied tiles: 0.1; noise added to the data matrix: 10 %. Each algorithm searched for a maximum of 200 tiles unless otherwise specified.

Overlap between the tiles. We created the data as explained above. In the step where the 100 tiles are copied, we vary the fraction of rows and columns that are changed. The fractions we used were 0.05, 0.1, 0.25, and 0.5.

Reconstruction errors for these experiments are shown in Fig. 3(a)–(c). There the overlap *decreases* as y-axis increases. `Naïve` can include tiles that increase the error. With 100 tiles, we also show the best result it obtained. This is depicted as a dashed line in Fig. 3(a).

With 600 tiles, the results show that the error increases as the overlap reduces. This is most likely due to the fact that there are more 1s to be covered as each tile is more disjoint. In case of 100 tiles, situation is different. Now, there are basically 100 "correct" tiles and 500 "incorrect" tiles to select, with increasing non-overlap making the incorrectness worse. Here we see that `Greedy` always selects the correct tiles. `HaPSi` selects good tiles initially, but starts deteriorating as the tiles get more disjoint. `Naïve` is always the worst eventually, but its best result is actually better than `HaPSi` with least overlap.

Data density. We considered data with density 0.06, 0.1, 0.15, 0.2, 0.25, 0.3, and 0.4. The tile density was adjusted depending whether 100 or 600 tiles were used,

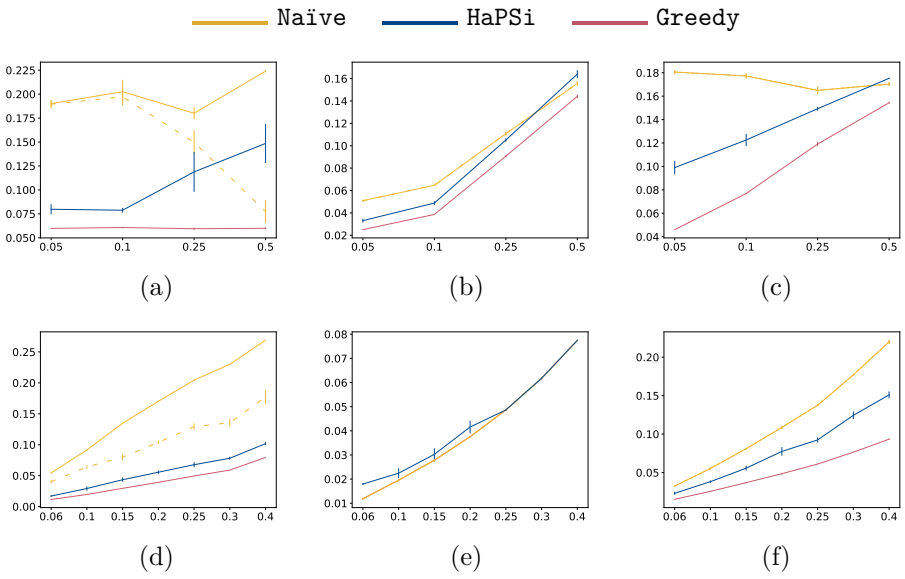

Fig. 3. Relative reconstruction error on y-axis. **Top row:** The amount d of changed 1s on x-axis. (a) Inexact case with 100 tiles. (b) Inexact case with 600 tiles. (c) Exact case with 600 tiles. In (a), the dashed line indicates the best error `Naïve` obtained. **Bottom row:** Data density on x-axis. (d)–(f) as (a)–(c).

so that the data matrix has the desired density. The results are depicted in Fig. 3(d)–(f). All algorithms perform worse as density increases with `HaPSi` being close to `Greedy` in all cases and `Naïve` being clearly worse in Fig. 3(d) and (f), even when considering its best result in (d).

Added noise. We added 5 %, 10 %, 20 %, and 30 % noise to the data matrix. Results are shown in Fig. 4(a)–(b). Again, as noise increases all methods get worse, and the order is clear: `Naïve` (worst), `HaPSi`, and `Greedy`, with `HaPSi` being relatively close to `Greedy`.

Data size. Finally we tested data size by generating matrices of size x-by-$(x+200)$ for x values of 1000, 2000, 3000, and 4000. The reconstruction errors are depicted in Fig. 4(d)–(e), showing that `HaPSi` performs very similarly to `Greedy` in terms of reconstruction error.

Running time. Running times of the algorithms with respect to number of tiles and size of the data are shown in Fig. 4(c) and (f), respectively. These plots show clearly that `Greedy` is significantly slower that `HaPSi`, which in turn is very close to `Naïve`, showing that `HaPSi` reaches a good balance between result quality and running time.

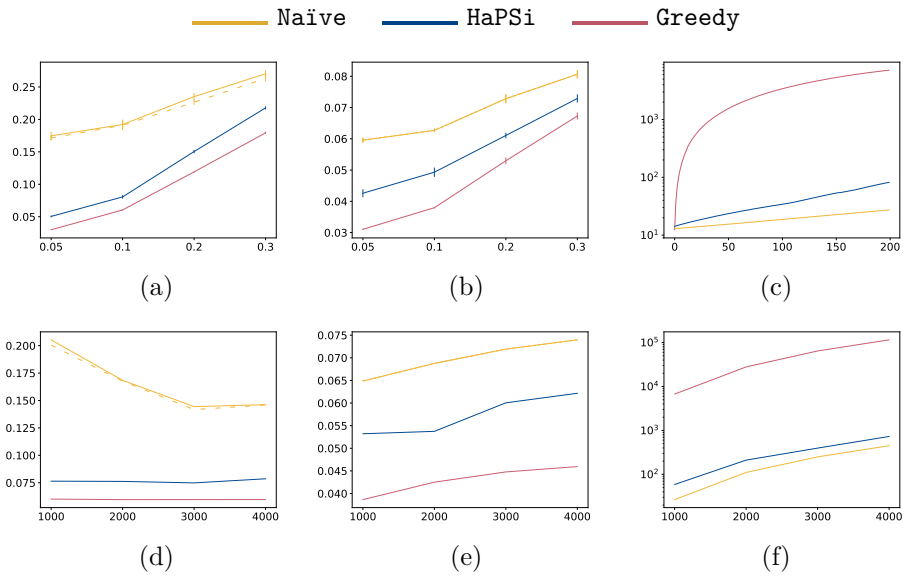

Fig. 4. Top row, (a)–(b): The amount of noise on x-axis, relative reconstruction error on y-axis. (a) Inexact, 100 tiles. (b) Inexact, 600 tiles. In (a), the dashed line indicates the best error `Naïve` obtained. **Top row, (c):** Running time (seconds, logarithmic scale) as number of selected patterns increases. **Bottom row, (d)–(e):** Number of rows in data on x-axis, relative reconstruction error on y-axis. (d)–(e) as (a)–(b). **Bottom row, (f):** Running time (seconds, logarithmic scale) as matrix size increases.

4.4 Results on Real-World Data

Data sets. We used four real-world data sets in our experiments. Some statistics about them are presented in Table 1. The Mammals dataset contains information about which mammal species inhabit which areas of the world on one side, and climate information on the other side [12]. The Dialect data contains features of spoken dialects of Finnish over different geographical regions [6, 7]. The 20Newsgroups data[4] is a corpus of 1000 posts from 20 different newsgroups, stemmed and with rare terms removed. The Abstracts data[5] is another corpus, this time of project abstracts. Terms are stemmed and rare ones are removed.

Redescription mining. We used redescriptions mined using the fast redescription mining algorithm Fier [14]. These results contained 161 redescriptions, some of them very similar to each other. The results are shown in Fig. 5(a) and (d). The results show that `HaPSi` is able to find very fast a good set of approximately 50 redescriptions. It should be noted that this data can be fully covered using all 161 tiles, so all methods converge towards the end. Notice also that `Greedy` is very slow.

[4] https://archive.ics.uci.edu/dataset/113/twenty+newsgroups
[5] https://archive.ics.uci.edu/dataset/134/nsf+research+award+abstracts+1990+2003

Table 1. Properties of real-world data sets

Data	Rows	Columns	Density (%)	Note
Mammals	54 013	4 802	—	Numerical data
Dialect	1 334	506	16.14	
20Newsgroups	5 163	19 997	0.89	
Abstracts	4 894	12 841	0.90	

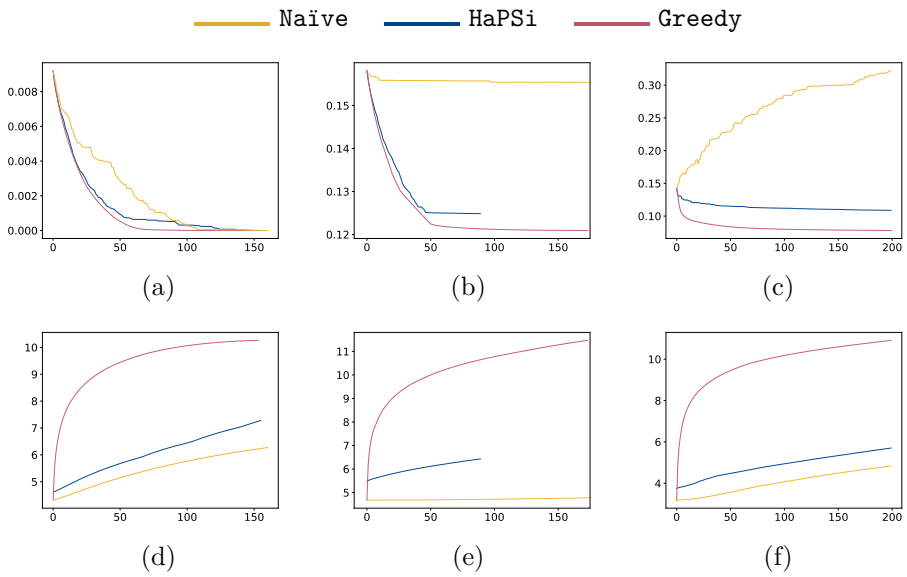

Fig. 5. Top row: Relative reconstruction error on the y-axis and number of tiles on the x-axis. (a) Tiles are redescriptions from the Mammals data. (b) Tiles are frequent itemsets from the Dialect data. (c) Tiles are rank-1 Boolean matrices for BMF from the Dialect data. **Bottom row:** Time (in seconds) on y-axis and number of tiles on x axis. (d)–(f) as (a)–(c).

Tiling databases. We mine frequent itemsets from the Dialect dataset with minimum frequency of 15 %. This gave us 28 535 frequent itemsets. We used all of these as tiles, and searched for a set of maximum 1000 tiles. The results are in Fig. 5(b) and (e). We see that `HaPSi` is again very good up to about 50 itemsets, after which it has converged. It is again significantly faster than `Greedy` and significantly better than `Naïve`.

Boolean matrix factorization. We created the rank-1 Boolean matrices using the association matrix technique used by the Asso algorithm and the restarted random walks technique [19]. These rank-1 matrices can cover 0s in the data, unlike in the previous cases. The results using the Dialect data are shown in Fig. 5(c) and (f). The results are similar to the other real-world use cases, except that in this case the `Naïve` algorithm increases its error consistently. But again `HaPSi` is competitive with `Greedy`, especially when we consider the significant advance it has on running time.

We also ran experiments with tiles found only by the association matrix technique using the two large corpus matrices. This created approximately 5000 tiles for 20Newsgroups and 12 000 for Abstracts. `HaPSi` took 850 s for Abstracts, while `Greedy` took 101 h. For 20Newsgroups `HaPSi` took 665 s and `Greedy` 74 h. This shows that `HaPSi` can find results efficiently with input sizes where `Greedy` is unable to produce the results in a reasonable amount of time.

5 Related Work

Tiling databases was first studied in [11], where also the use of the greedy algorithm was proposed. The problem was later independently studied in [27]. The tiling was later extended to hierarchical representations (where tiles contain other tiles) [24], data streams [15], and ranked data [16], among others.

Boolean matrix factorization (BMF) can be seen as a variant of tiling. It has been originally studied in combinatorics (see [21] and references therein), but it has also seen significant research interest in data mining (see [19] and references therein). An exact Boolean matrix factorization is the same as exact tiling, and [2] propose an algorithm for exact BMF based on greedy Set Cover algorithm.

The aforementioned methods use reconstruction error as the primary means of deciding which tiles or rank-1 matrices to use. Minimum Description Length (MDL) principle is another popular method for selecting the pattern set (see, e.g. [26, 17, 20]). In MDL, the idea is to select those tiles that compress the data the best.

Another method for selecting the set of patterns was proposed by De Bie [5]. His proposal is based on modelling the data with a maximum-entropy distribution, and selecting the tiles that are the most surprising (i.e. least likely) under this distribution. After each selected tile, the distribution is updated by constraining it with the selected tile such that the new distribution has the maximum entropy over all those distributions where the found tiles appear. Later, [13] extended that approach to selecting a set of redescriptions.

Redescription mining [10] was proposed in [22] and has seen various algorithms and applications proposed since (see [10]). The first redescription set mining approach was proposed in [9] and later [13] proposed a version based on subjective interestingness.

The idea of using bottom-k hashing for size estimation was first proposed in [1], while [23] extended it to the size of Boolean matrix product and explained the efficient algorithm we use as the basis of `HaPSi`.

6 Conclusions

Bottom-k hashing can provide significant speedups for selecting a good set of patterns with minimal effects on the quality of patterns selected. We showed how our algorithm can be applied to tiling, Boolean matrix factorization, and redescription set mining, but we believe that there are other pattern set mining problems where our approach can be useful, as well.

The hashing techniques presented in this paper are limited to reconstruction error. Algorithms like `HaPSi` are not directly good for selecting itemsets based on measures like MDL, as they pay no attention to the "cost" of the selected patterns. Indeed, we can see in Fig. 5 that `HaPSi` sometimes selects patterns that improve the error very little if at all; such patterns are very unsuitable for optimizing MDL, say. In principle, the description length could be taken into account when selecting the patterns, but it is not clear if the bottom-k hashing would ever find patterns that are good in MDL-sense. Developing these algorithms is an interesting topic for a future work.

Being able to select the pattern set faster can also facilitate novel algorithms that build sets of pattern sets. Such algorithms can provide a hierarchical view to the data, supporting exploration with few high-level pattern-sets-as-patterns, each of which can be "zoomed in" to see the actual patterns.

Our method does not allow novel unethical uses of data mining techniques although the vast speed improvements do allow existing potentially unethical uses to be scaled to much larger data sets.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Bar-Yossef, Z., Jayram, T., Kumar, R., Sivakumar, D., Trevisan, L.: Counting distinct elements in a data stream. In: Proc. 6th International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM). pp. 1–10. Springer (2002). https://doi.org/10.1007/3-540-45726-7_1
2. Bělohlávek, R., Vychodil, V.: Discovery of optimal factors in binary data via a novel method of matrix decomposition. Journal of Computer and System Sciences **76**(1), 3–20 (2010). https://doi.org/10.1016/j.jcss.2009.05.002

3. Cohen, E.: Size-estimation framework with applications to transitive closure and reachability. Journal of Computer and System Sciences **55**(3), 441–453 (1997). https://doi.org/https://doi.org/10.1006/jcss.1997.1534, https://www.sciencedirect.com/science/article/pii/S0022000097915348
4. Cohen, E., Kaplan, H.: Summarizing data using bottom-k sketches. In: Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing. p. 225–234. PODC '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1281100.1281133, https://doi.org/10.1145/1281100.1281133
5. De Bie, T.: Maximum entropy models and subjective interestingness: an application to tiles in binary databases. Data Mining and Knowledge Discovery **23**(3), 407–446 (2011). https://doi.org/10.1007/s10618-010-0209-3
6. Embleton, S.M., Wheeler, E.S.: Finnish dialect atlas for quantitative studies. Journal of Quantitative Linguistics **4**(1–3), 99–102 (1997). https://doi.org/10.1080/09296179708590082
7. Embleton, S.M., Wheeler, E.S.: Computerized dialect atlas of Finnish: Dealing with ambiguity. Journal of Quantitative Linguistics **7**(3), 227–231 (2000). https://doi.org/10.1076/jqul.7.3.227.4109
8. Feige, U.: A threshold of $\ln n$ for approximating set cover. Journal of the ACM **45**(4), 634–652 (1998). https://doi.org/10.1145/285055.285059
9. Galbrun, E., Miettinen, P.: Mining redescriptions with siren. ACM Transactions on Knowledge Discovery from Data **12**(1), 1–30 (2018). https://doi.org/10.1145/3007212
10. Galbrun, E., Miettinen, P.: Redescription Mining. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-72889-6
11. Geerts, F., Goethals, B., Mielikäinen, T.: Tiling databases. In: Proc. 7th International Conference on Discovery Science (DS). p. 278–289 (2004). https://doi.org/10.1007/978-3-540-30214-8_22
12. Hijmans, R.J., Cameron, S.E., Parra, J.L., Jones, P.G., Jarvis, A.: Very high resolution interpolated climate surfaces for global land areas. International Journal of Climatology **25**(15), 1965–1978 (2005). https://doi.org/10.1002/joc.1276
13. Kalofolias, J., Galbrun, E., Miettinen, P.: From sets of good redescriptions to good sets of redescriptions. Knowledge and Information Systems **57**, 21–54 (2018). https://doi.org/10.1007/s10115-017-1149-7
14. Karjalainen, M., Galbrun, E., Miettinen, P.: Fast redescription mining using locality-sensitive hashing. In: Proc. European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD). pp. 124–142. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-70368-3_8
15. Lam, H.T., Pei, W., Prado, A., Jeudy, B., Fromont, É.: Mining top-k largest tiles in a data stream. In: Proc. European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD). pp. 82–97 (2014). https://doi.org/10.1007/978-3-662-44851-9_6
16. Le Van, T., van Leeuwen, M., Nijssen, S., Fierro, A.C., Marchal, K., De Raedt, L.: Ranked tiling. In: Proc. European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD). pp. 98–113. Springer Berlin Heidelberg (2014). https://doi.org/10.1007/978-3-662-44851-9_7
17. Mampaey, M., Tatti, N., Vreeken, J.: Tell me what i need to know: Succinctly summarizing data with itemsets. In: Proc. 20th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD). pp. 573–581 (2011). https://doi.org/10.1145/2020408.2020499

18. Miettinen, P.: On the positive-negative partial set cover problem. Information Processing Letters **108**(4), 219–221 (2008). https://doi.org/10.1016/j.ipl.2008.05.007
19. Miettinen, P., Neumann, S.: Recent developments in Boolean matrix factorization. In: Proc. 29th International Joint Conference on Artificial Intelligence (IJCAI). pp. 4922–4928 (2020). https://doi.org/10.24963/ijcai.2020/685
20. Miettinen, P., Vreeken, J.: MDL4BMF: Minimum description length for Boolean matrix factorization. ACM Transactions on Knowledge Discovery from Data **8**(4) (2014). https://doi.org/10.1145/2601437
21. Monson, S.D., Pullman, N.J., Rees, R.: A survey of clique and biclique coverings and factorizations of (0,1)-matrices. Bulletin of the ICA **14**, 17–86 (1995)
22. Ramakrishnan, N., Kumar, D., Mishra, B., Potts, M., Helm, R.F.: Turning CARTwheels: An alternating algorithm for mining redescriptions. In: Proc. 10th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD). pp. 266–275 (2004). https://doi.org/10.1145/1014052.1014083
23. Resen Amossen, R., Campagna, A., Pagh, R.: Better size estimation for sparse matrix products. Algorithmica **69**, 741–757 (2014). https://doi.org/10.1007/s00453-012-9692-9
24. Tatti, N., Vreeken, J.: Discovering descriptive tile trees by mining optimal geometric subtiles. In: Proc. 2012 European Conference on Machine Learning and Priciples and Practice on Knowledge Discovery in Databases (ECML PKDD). pp. 9–24 (2012). https://doi.org/10.1007/978-3-642-33460-3_6
25. Thorup, M.: Bottom-k and priority sampling, set similarity and subset sums with minimal independence. In: Proc. forty-fifth annual ACM symposium on Theory of Computing (STOC). pp. 371–380 (2013). https://doi.org/10.1145/2488608.2488655
26. Vreeken, J., van Leeuwen, M., Siebes, A.: Krimp: mining itemsets that compress. Data Mining and Knowledge Discovery **23**(1), 169–214 (2011). https://doi.org/10.1007/s10618-010-0202-x
27. Xiang, Y., Jin, R., Fuhry, D., Dragan, F.F.: Summarizing transactional databases with overlapped hyperrectangles. Data Mining and Knowledge Discovery **23**(2), 215–251 (2010). https://doi.org/10.1007/s10618-010-0203-9