technische universität
dortmund

Diplomarbeit

Untersuchung von Code-Tabellen zur Kompression von binären Datenbanken

Sibylle Hess
April

Gutachter:
Katharina Morik
Nico Piatkowski

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Künstliche Intelligenz (LS-8)
http://www-ai.cs.uni-dortmund.de

Zusammenfassung

Mit zunehmender Integration informationsverarbeitender Systeme in das alltägliche Leben steigt die Menge verfügbarer Daten. Ein Großteil dieser Daten ist binärer Natur. Sei es bei der Repräsentation natürlicher Sprache mittels *Bag of Words* Modellen, der Darstellung von Transaktionen, Likes oder Dislikes in sozialen Netzwerken oder der quantitativen Analyse von Genexpressionsdaten. Die Repräsentation mittels binärer Datenbanken findet in vielerlei Hinsicht ihre Bedeutung.

Als beliebte Methode zur Extraktion der zugrundeliegenden Information binärer Datenbanken hat sich das Frequent Pattern Mining erwiesen. Häufig zusammen vorkommende Muster sollen die notwendigen Indizien liefern um Aussagen über die zugrundeliegende Thematik treffen zu können. Redundanz in den auftretenden Mustern macht eine Interpretation der erhaltenen Information jedoch schwierig. Eine Methode zur Filterung der relevanten Aussagen ist von Notwendigkeit.

In dieser Diplomarbeit sollen die Möglichkeiten zur Auswahl solcher Muster mittels Code-Tabellen untersucht werden. Eine Code-Tabelle stellt die elementaren Informationen einer Datenbank in der möglichst kompaktesten und präzisesten Variante zusammen. Ihre Qualität bemisst sich in der Fähigkeit den vorliegenden Sachverhalt geeignet zu komprimieren. Es werden vorhandene Methoden zur Berechnung von Code-Tabellen analysiert und eine geeignete Erweiterung dieser Methoden besprochen. Die Verbindungen zu anderen bekannten Verfahren, beispielsweise zum Clustering, werden herausgestellt. Des Weiteren wird eine vollkommen neue Herangehensweise zur Bestimmung der besten Kodierung, basierend auf Matrix-Faktorisierung vorgestellt. Dies ermöglicht eine Herleitung der Kodierung mithilfe bekannter numerischer Methoden.

Die besprochenen Ansätze werden auf realen Datensätzen evaluiert, sowohl in Hinblick auf Qualität der Code-Tabelle als auch der Performanz entsprechender Algorithmen.

Contents

Chapter 1

Introduction

Basic Patterns, grouped and changed
Sequence frequent, seek and gain
Break, break for narration

– LL Cool J, *I need a Beat*

Nowadays, we experience an enormous growth of the information that is available. Large amounts of data are collected and require efficient mechanisms to extract its information. It arises the question how intrinsic structures, reflecting the important characteristics of a large data collection, can be derived. One popular method is to summarize the inherent structure of a dataset by its patterns. Patterns shall reflect particularly interesting or at least reoccurring parts of the data. As described by Mannila and Toivonen [33] we seek for the theory of the dataset, represented by the subsets that satisfy a given predicate of interest.

The most common practice is the frequent pattern mining [1] that identifies the relevance of a pattern with its frequency. The monotonic property of frequent sets, that all subsets of a frequent pattern are also frequent, induces however a high redundancy between the returned patterns. In addition, the threshold that denotes the minimal number of occurrences at which a pattern may be called frequent, is hard to set. Determining this threshold too high results in an output of a small set of patterns that reveals nothing but common knowledge. A small decrease is however likely to let the number of issued patterns literally explode. Mining such an enormous amount of patterns poses the question again how the information of these patterns can be extracted, which connections or correlations are given between them.

In this work, we review existing approaches and develop new methods to filter and summarize the information of a dataset by a simple principle. This principle relies on the observation that any regularity of the data can be used to compress the data [21] and is known as the *Minimum Description Length* (MDL) principle. We discuss the relations of this approach to other ones that intend to discover the underlying characteristics of

datasets and spot the specific differences. Furthermore, we develop new methods to derive the models that compress a database by means of numerical optimization. Experiments on common binary datasets complete the comparison of the regarded endeavors to obtain a precise summarization of the data.

1.1 Preliminaries

Throughout this work, we assume that we are given a set of $n \in \mathbb{N}$ items $\mathcal{I} = \{x_1, \ldots, x_n\}$ and a database $\mathcal{D} = \{t_1, \ldots, t_m\}$ that consists of $m \in \mathbb{N}$ transactions $t_j \subseteq \mathcal{I}$ for $1 \leq j \leq m$. Let $X \subseteq \mathcal{I}$ be an itemset. The cover of X is denoted as the set $\mathrm{cov}(X)$ that comprises the transactions containing the itemset X, i.e.,

$$\mathrm{cov}(X) = \{t \in \mathcal{D} \,|X \subseteq t\}\,.$$

The frequency of X is defined as the number of transactions that cover X

$$\mathrm{freq}(X) = |\,\mathrm{cov}(X)|.$$

The relative frequency $\mathrm{supp}(X)$ is denoted as the support that puts the number of occurrences in relation to the number of transactions. For a given minimum support threshold $minsup \in [0, 1]$, the set X is called frequent if

$$\mathrm{supp}(X) = \frac{\mathrm{freq}(X)}{|\mathcal{D}|} \geq minsup.$$

We define $\mathcal{F}$ as the set of all frequent patterns.

A database $\mathcal{D}$ may also be described by a binary matrix $D \in \{0,1\}^{n \times m}$, where the entry D_{ij} is set to one iff the item x_i is contained in transaction t_j. In this notation, a transaction $t_j \in \mathcal{D}$ is represented by the column $D_{\cdot j}$.

We denote by $\circ$ the boolean matrix product, i.e., for a matrix $C \in \{0,1\}^{n \times m}$ it holds that $C = A \circ B$ iff

$$C_{ij} = A_{i\cdot} \circ B_{\cdot j} = \bigvee_{s=1}^{r} A_{is} \wedge B_{sj}.$$

The Hadamard product is referred by $*$, i.e., for matrices $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{n \times m}$ it holds that

$$(A * B)_{ij} = A_{ij} B_{ij}.$$

The Frobenius norm is notated by $\|\cdot\|$ and with $|\cdot|$ we denote the matrix 1-norm, i.e., for a matrix $A \in \mathbb{R}^{n \times m}$ the respective norms can be calculated by

$$\|A\| = \left(\sum_{i=1}^{n} \sum_{j=1}^{m} A_{ij}^2 \right)^{\frac{1}{2}}$$

$$|A| = \sum_{i=1}^{n} \sum_{j=1}^{m} |A_{ij}|.$$

Throughout this paper all logarithms are to base 2 and by convention we use $0 \log 0 = 0$. Further, we denote by $X \preceq Y$ that X is preceding to Y with respect to an order that will be clear from the context.

1.2 Related Work

First attempts to cope with the overflow of information that is obtained by frequent pattern mining algorithms try to shorten the returned results by a restrictive generation of closed or maximal frequent itemsets [37, 11, 3]. An itemset X is called closed if there is no superset of X that has the same support. Therewith, it is possible to describe the set of frequent patterns losslessly by a subset. The size of this subset is yet sensitive to the nature of the database, in particular to the length of the frequent patterns, i.e., the cardinality of the itemset, and the density of the database [38]. For this occasion, algorithms have been developed to mine only the set of maximal frequent patterns. Those are the frequent itemsets that do not have any frequent superset [11, 3]. In this representation, the information about the support of many frequent itemsets is however lost.

There are many related approaches to describe a set of frequent or generally interesting patterns in a compact way. Free frequent itemsets [10], non-derivable itemsets [13] or sampling frequent itemsets [14] are further examples of this category. However, these methods focus either on deriving all sets that satisfy a specified property or on finding a specified subset that shall describe the set of interesting patterns. The constellation of the returned patterns is thereby not taken into account. None of the approaches above set the interestingness of a pattern in relation to the set of patterns that has already been considered to be useful.

That's why Siebes et al. introduced another idea with KRIMP [45] where the most interesting patterns are selected according to the MDL principle [21]. The patterns are retained in a dictionary for binary code words that is used to encode the database. The best set of patterns is identified as the one that yields the best database compression. By this strategy, the number of returned patterns is reduced drastically, e.g. from billions of frequent patterns to less than 700 itemsets for the mushroom database, and has a variety of applications [43, 44, 26]. KRIMP has as input a preferable large set of frequent patterns, i.e., the set of frequent patterns given a small minimum support. These are the code word candidates, a set that is likely to be orders of magnitudes larger than the input dataset, due to the aforementioned pattern explosion. KRIMP is designed to find the best compressing selection of patterns from its candidate set by a greedy procedure. Every candidate pattern is regarded once and added to the set of code words if the inclusion of the considered pattern improves the compression size. The achieved compression quality is thus increasing with the number of regarded candidates. However, a generation of all occurring itemsets in the database is often not possible, referable to time and especially

to storage limits. The exponential increase of frequent patterns with decreasing minimum support, poses the difficult problem to find a suitable minimum support that generates roughly the right amount of candidates.

The algorithm SLIM [41] encounters this problem by a candidate generation that results directly from the current selection of code words. In every iteration, candidates are generated and sorted by their estimated compression gain and the first pattern that actually enhances the compression size is accepted. The estimation of the compression size requires however an identification of the affected parts of the database and for some candidates the actual compression size has to be computed as well. These operations are performed for most of the time and the combinatorial possibilities regarding the candidate generation are numerous. Thus, an indexing structure that supports these operations and that gives insight into the consequences of integrating a pattern into the encoding, is desirable.

With the algorithm SHrimp[23] a tree structure is proposed to facilitate a fast identification of the interesting parts of the dataset. Therewith, a direct determination of the consequences that result from a change of the current pattern selection shall be enabled. In addition, the tree representation can be utilized to visualize the encoding of the dataset and the dependencies of patterns in certain parts of the database.

As pointed out in [32] the derived patterns composing the set of code words are often variants on the same theme although redundancy is reduced with respect to the compression. In this context, Siebes et al. considered a new methodology to derive the set of code words. They propose to mine the set of the k patterns that compresses the database best [40]. Their developed algorithm Groei uses a beam search to obtain the desired patterns, but this procedure is very expensive and only small datasets can be regarded. Therefore, we will not discuss this approach in detail. However, we will have a look at another approach that is more suitable to solve this problem definition.

1.3 Outline

We proceed as follows: In Chapter 2 we give a theoretical introduction to the principles of MDL and related concepts. We present the calculation of the compression size and reflect the possibilities that are given to derive an optimal encoding. We discuss the relation to the method of tiling that draws a connection to the task of matrix factorization. Common methods and applications are summarized that are applied to obtain accurate factorizations with regard to different constraints.

In Chapter 3 we review and discuss the approaches to derive a suitable encoding by a code table. Existing algorithms are described and new methods presented. We consider several contributions to the algorithms. Both in theoretical as well as in algorithmic views.

The methodology to represent an encoding by a matrix factorization is introduced in this chapter and two algorithms are proposed that rely on this principle.

The conducted experiments are discussed in Chapter 4. We evaluate the discussed algorithms with regard to their performance and quality of the returned code table on seven selected binary databases.

Finally, we conclude our observations in Chapter 5.

Chapter 2

Compressing a Database and related Foundations

We explain in the following the terms in that a good encoding of the database can be derived. We discuss the complexity of the problem and give an introduction to related variants. In particular, we provide an overview on the task of matrix factorization and the methods of numerical optimization that can be applied to approximate a matrix by a factorization.

2.1 The MDL principle

MDL has been introduced by Rissanen et al. [39] as an applicable version of the Kolmogorov complexity [27]. Given a set of models $\mathcal{M}$, the best model is identified as the one that minimizes the compression size

$$L(\mathcal{D}, M) = L(\mathcal{D}|M) + L(M),$$

whereby $L(\mathcal{D}|M)$ denotes the compression size of the database in bits, assuming that model M is used for the encoding and $L(M)$ is the description size in bits of the model M itself.

2.1.1 Encoding a Database

We define a coding set $CS \subseteq \mathcal{P}(\mathcal{I})$ as a set of patterns that contains at least all singleton itemsets $\{\{x\}|x \in \mathcal{I}\} \subseteq CS$. Let $code : CS \to \{0,1\}^*$ be a mapping from patterns in the coding set to a finite, unique and prefix-free code. A *code table* is denoted by a set of pairs

$$CT \subseteq \{(X, code(X))|X \in CS\},$$

or equivalently by a two column table where the patterns are listed on the left column and the assigned codes on the right. Code tables represent the compressing models in Krimp and can be interpreted as dictionaries for code words. Once the *code* function is

CT	$usage_{CT}(X)$
$\{b, d, e\}$	4
$\{a, c\}$	4
$\{a, g\}$	2
$\{e\}$	2
$\{f\}$	4
$\{g\}$	2

$\mathcal{D}$	encoding set
a, b, d, e, f, g	$\{b, d, e\}\{a, g\}\{f\}$
a, b, c, d, e, g	$\{b, d, e\}\{a, c\}\{g\}$
a, c, e, f	$\{a, c\}\{e\}\{f\}$
a, b, c, d, e, f	$\{b, d, e\}\{a, c\}\{f\}$
a, c, e, g, f	$\{a, c\}\{e\}\{g\}\{f\}$
a, b, d, e, g	$\{b, d, e\}\{a, g\}$

Figure 2.1: An example code table as it is induced by a coding set is depicted on the left, the usage of the patterns is stated as well. The database and the corresponding encoding set is shown on the right.

determined, the coding set induces a code table by itself. We use therefore the term code table simultaneously to describe a coding set, since the association is given more directly by the former designation. The problem we want to solve can thus be stated as the task to find the best compressing coding set of a database. The explicit *code* function that transfers the coding set into a set of concrete codes is introduced in Section 2.1.2.

The function $encode : \mathcal{P}(\mathcal{I}) \to \mathcal{P}(CS)$ determines the way in that a database is encoded. It selects the patterns of a coding set, and with that the code words that are used to encode a specified transaction. Given a transaction t, the set $encode(t)$ is called the *encoding of transaction* t and the items $x \in X \in encode(t)$ are called *encoded by* X. The representation of an encoding of the database, as it is given by the patterns that are used to constitute its transactions, is called the encoding set. Figure 2.1 shows and example code table, respectively only the left side of the table that denotes the selection of patterns that are used for the encoding. The concrete codes can then be deduced by the *code* function. The function $usage_{CT}(X)$ calculates the number of occurrences of the pattern X in the encoding set of the database. This function is discussed more detailed in the next section. On the right of Figure 2.1, we see the respective database and its encoding set.

2.1.2 Computing the Compression Size

We assume that codes are constructed such that more frequently used codes are smaller in size. The existence of such a set of codes is guaranteed by Theorem 5.4.1 in Cover & Thomas [15]. The theorem states that for a given distribution P over a finite set $\mathcal{X}$ there exists an optimal set of prefix-free codes such that the length of the codes measured in the number of required bits $L(code(X))$ for $X \in \mathcal{X}$ is given by

$$L(code(X)) = -\log(P(X)). \tag{2.1}$$

Codes that satisfy this property are e.g. the Shannon-Fano or Huffman code. We emphasize that the derivation of an optimal code table does not require a realization of actual codes. Throughout this work, we simply assume that codes are given that satisfy Eq. (2.1).

Since we want that more frequently used codes have a smaller size, the probability distribution that describes how likely a code word is used for the encoding has to be stated. This is induced as follows. Let CT be a code table, respectively the coding set that induces the code table, and $X \in CT$ an itemset of the code table. We denote with $usage_{CT}(X)$ the number of transactions that are encoded using X, i.e., $usage_{CT}(X) = |\{t \in \mathcal{D} \,|\, X \in encode(t)\}|$. Thus, the probability that X is used for the encoding of the database is given as

$$P(X) = \frac{usage_{CT}(X)}{\sum_{Y \in CT} usage_{CT}(Y)}.$$

The usage of an itemset $X \in CT$ is equal to the frequency of $code(X)$ in the encoded database. In this respect, the compression size of a database $\mathcal{D}$ given a code table CT is computed by

$$\begin{aligned} L(\mathcal{D}|CT) &= \sum_{t \in \mathcal{D}} \sum_{X \in encode(t)} L\big(code_{CT}(X)\big) \\ &= -\sum_{X \in CT} usage_{CT}(X) \cdot \log\big(P(X)\big). \end{aligned} \tag{2.2}$$

Formulation (2.2) as a sum over code table elements allows for a faster computation of the compression size than the formulation above (2.2) suggests. Since the code table contains assumable much less sets than transactions in the database exist, the sum in Eq. (2.2) is computed with low expenses if the usage function is computed capably.

A code table CT is represented in terms of the *standard code table* that provides codes for singleton itemsets, i.e., for sets $\{x\}$ with $x \in \mathcal{I}$. The patterns of a code table CT are described by concatenated singleton-codes as provided by the standard code table ST. So, the description length of the code table is given by

$$\begin{aligned} L(CT) &= \sum_{X \in CT} \left(L\big(code_{CT}(X)\big) + \sum_{x \in X} L\big(code_{ST}(x)\big) \right) \\ &= -\sum_{X \in CT} \left(\log\big(P(X)\big) + \sum_{x \in X} \log\big(\,\mathrm{supp}(\{x\})\big) \right). \end{aligned} \tag{2.3}$$

We note that the relative usage of codes in the standard encoding is equal to the support of the respective singleton.

Applying the MDL principle, the total compression size is calculated as the sum of the description size of the encoded database given the code table CT and the description size of the model itself:

$$L(D, CT) = L(\mathcal{D}|CT) + L(CT)$$

The compression size depends therewith mainly on the obtained usage of codes in the database. That is in turn determined by the applied encoding or more specifically by the definition of the function *encode*. The possibilities to define this function are however numerous. We summarize briefly how complex it actually is to calculate the best possible encoding.

2.1.3 Complexity of the Encoding

Since there are at most $2^{|\mathcal{I}|} - |\mathcal{I}| - 1$ patterns with a cardinality greater than one, the number of possible coding sets with k non-singleton itemsets is equal to $\binom{2^{|\mathcal{I}|}-|\mathcal{I}|-1}{k}$. In total there are at most

$$\sum_{k=0}^{2^{|\mathcal{I}|}-|\mathcal{I}|-1} \binom{2^{|\mathcal{I}|} - |\mathcal{I}| - 1}{k}$$

possible coding sets [45]. For each of these coding sets, all possible encoding functions have to be considered to obtain the best compression. Assuming that codes must not overlap, i.e., the encoding of a transaction is disjunctive, the best compression for a single transaction can be determined by an order of the code table for this transaction. Since there are $(k + |\mathcal{I}|)!$ possible orders for a coding set with k non-singleton patterns, the number of possible encodings of a database $\mathcal{D}$ is given as

$$|\mathcal{D}| \sum_{k=0}^{2^{|\mathcal{I}|}-|\mathcal{I}|-1} \binom{2^{|\mathcal{I}|} - |\mathcal{I}| - 1}{k} (k + |\mathcal{I}|)!.$$

This number is quite large. That is also why all algorithms known so far tackle this problem by a heuristic determination of the encoding function by a static order. If overlap of codes is allowed, the possibilities to define an encoding are even more numerous.

2.2 Tiling

The encoding of a database has also an interpretation as a tiling of the database. A tiling is a finite set of pairs $T = (X, Y)$, consisting of a pattern $X \subseteq \mathcal{I}$ and a subset of covering transactions $Y \subseteq \text{cov}(X)$, called tiles. A tile describes also a selection of rows $X_r = \{i | x_i \in X\}$ and columns $Y_c = \{j | t_j \in Y\}$ of the matrix representation $D \in \{0,1\}^{n \times m}$ of the database. Since all transactions in Y contain the pattern X, extracting the rows and columns as described by the tile $(D_{ij})_{i \in X, j \in Y}$ yields a $|X| \times |Y|$ matrix full of ones. If we reorder the columns and rows of D such that the columns of Y_c and rows of X_r are side by side, we obtain a block full of ones in the data matrix D. Consequently, if we order the rows and columns of a matrix for a given tiling suitably in a way that all rows and columns of each tile are next to each other, the tiles can be identified as blocks of ones in the matrix. Those might also be overlapping. Figure 2.2 shows such a transformation of

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\!\!\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \left(\begin{array}{ccccc} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{array}\right) \end{array} \qquad \begin{array}{c} \\ 3 \\ 1 \\ 2 \\ 4 \end{array}\!\!\begin{array}{c} \begin{array}{ccccc} 2 & 4 & 3 & 5 & 1 \end{array} \\ \left(\begin{array}{ccccc} 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{array}\right) \end{array}$$

Figure 2.2: Permutation of rows and columns such that items and transactions of the tiles $T_1 = (\{1,3\},\{2,3,4,5\})$ (green) and $T_2 = (\{1,2,4\},\{1,3,5\})$ (red) are next to each other.

a binary matrix with two tiles from its original representation to the one with permuted rows and columns that reveals the inherent structure of the tiling.

There are various challenges that can be formulated with respect to tiling [19]. Those that intend to find a collection of tiles that describes the database are *Maximum k-Tiling* and *Minimum Tiling*. In Maximum k-Tiling, the goal is to find for a specified number of at most k tiles a tiling that covers as many ones in the database as possible. By contrast, a Minimum Tiling is a decomposition of the database into as few tiles as possible. In this respect, the encoding of a database yields also a tiling. An itemset X of a code table indicates also a pattern of a tile $T = (X, Y)$ whose respective transactions Y are defined as the transactions that are encoded by this pattern. The derivation of an encoding that compresses the database best is thus related to the derivation of a Minimum Tiling. The difference is that not only as few tiles as possible shall be used, but the assignment of columns to patterns that denote the usage, shall induce a small compression size. As pointed out in [41], a Minimum Tiling does not yield good code tables. However, the first few tiles mined by a greedy procedure, that chooses in every iteration the tile that covers most of the ones, typically do compress well. The relationship to this kind of problems indicates also the inherent clustering properties of an MDL related compression as we now point out.

2.2.1 Relations to Boolean Matrix Factorization

A tiling poses a boolean matrix factorization of the regarded data matrix [34]. That is an approximation of a binary matrix $D \approx X \circ Y$ by the boolean matrix product of two factor matrices $X \in \{0,1\}^{n \times r}$ and $Y \in \{0,1\}^{r \times m}$ of a given rank $r \in \mathbb{N}$. As such, the tiling displayed in Figure 2.2 yields the following factorization.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \circ \begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix} = X \circ Y$$

We can see that every tile corresponds to the product of a column of X and a row of Y, i.e., for $T_1 = (\{1,3\},\{2,3,4,5\})$ colored in green, the first and third entry of the corresponding

column in X and the second to the fifth entry of the corresponding row in Y are set to one. Similarly, a tiling with non overlapping tiles poses a binary matrix factorization (BMF) $D \approx X \cdot Y$ with $X \in \{0,1\}^{n \times r}$ and $Y \in \{0,1\}^{r \times m}$.

The problem of finding a good approximation by a boolean matrix factorization is also referred to as the *Discrete Basis Problem* (DBP), which is known to be NP-hard and NP-hard to approximate [34]. The word *basis* refers thereby to the interpretation of the columns of X as basis vectors whose linear combinations denoted by the coefficients in Y construct the columns of the data matrix D. E.g., the last column of our example matrix is computed by

$$\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \circ \begin{pmatrix} 1 \\ 1 \end{pmatrix} = 1 \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} \vee 1 \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}.$$

This representation establishes an interpretation of the DBP as a possibly overlapping clustering of the columns in D. This is justified by the conjunction of Nonnegative Matrix Factorization and the DBP.

2.3 Matrix Factorization

The most common application in the field of Matrix Factorization is the task of Nonnegative Matrix Factorization (NMF). It can generally be described as the approximation of a matrix $D \in \mathbb{R}^{n \times m}$ with $n \in \mathbb{N}$ features and $m \in \mathbb{N}$ observations by the product of two matrices $X \in \mathbb{R}_+^{n \times r}$ and $Y \in \mathbb{R}_+^{r \times m}$ for a given rank $r \in \mathbb{N}$. The quality of the factorization is often measured by the residual sum of squares (RSS), i.e., a small RSS results in a good factorization. The objective is therefore given as

$$\min_{X,Y} F(X,Y) = \frac{1}{2}\|D - XY\|^2. \tag{2.4}$$

Initially, the differences between NMF and clustering have been emphasized [24]. Further research affirmed however the inherent clustering properties [29]. In that view, basis vectors resemble cluster centroids and the coefficient Y_{sj} indicates the degree with that the j-th observation $D_{\cdot j}$ is associated to the s-th cluster $X_{\cdot s}$. Binary restrictions on $Y \in \{0,1\}^{r \times m}$ make the cluster memberships definite. Orthogonality constraints $Y^T Y = I$ enforce additionally unique cluster assignments and the corresponding matrix factorization problem is equivalent to the problem of k-means clustering [18, 17]. If the data matrix $D \in \{0,1\}^{n \times m}$ is binary, it might be desirable to restrict both matrices X and Y to binary values, at least to get interpretable results for the cluster centroids [28]. This formulation has also a reading as a simultaneous clustering of data points and features, also known as *biclustering*.

Biclustering is especially applied to identify groups of highly related genes in sample subsets from gene expression data [6]. Other applications include textmining as a simultaneous clustering of words and documents, and collaborative filtering, as a simultaneous clustering of users and opinions [12]. Similarly, Miettinen et al. consider the DBP with respect to a topic extraction from documents as it is described for topic models [16, 7]. As evaluated by Zhang et al. [47, 48], Binary Matrix Factorization (BMF) is a favorable technique that outperforms common biclustering algorithms on synthetic as well as real world datasets.

We review now common methods of NMF and discuss afterwards how these attempts can be extended to BMF.

2.3.1 Algorithms for Nonnegative Matrix Factorization

NMF has been introduced by Paatero et al. [36] initially under the name Positive Matrix Factorization and received much attention since the publication of the easily implementable multiplicative update algorithm by Lee and Seung [25]. This procedure is sketched in Alg. 1. For each of the K iterations, the matrices are updated alternating by an elementwise

Algorithm 1 Multiplicative Update NMF

procedure MULTMF(D, r, K)
  Initialize $X_0 \in \mathbb{R}^{n \times r}$ and $Y_0 \in \mathbb{R}^{r \times m}$ randomly
  for $k \in \{1, \dots, K\}$ **do**
    $(X_k)_{is} = (X_k)_{is} \frac{(Y_k^T D)_{is}}{(Y_k^T Y_k X_k)_{is}}$
    $(Y_k)_{sj} = (Y_k)_{sj} \frac{(D X_{k+1}^T)_{sj}}{(Y_k X_{k+1} X_{k+1}^T)_{sj}}$
  end for
  return (X_K, Y_K)
end procedure

multiplication with respect to one matrix while the other one is fixed. That way, the generated sequence (X_k, Y_k) preserves nonnegativity of the factor matrices and it can be proven that it is nonincreasing with respect to the objective function values $F(X_k, Y_k)$. Furthermore, the sequence is proven to converge to a critical point [30], but convergence is slow in practice [31, 4]. By contrast, good convergence rates are obtained by an application of the Gauss-Seidel scheme, also known as block-coordinate descent. This procedure is outlined in Alg. 2. In this procedure, the function values are alternatingly minimized with respect to one matrix while the other one is fixed. The convergence of the generated sequence (X_k, Y_k) is therefore granted, because the objective F of Eq. (2.4) is convex in X if Y is fixed and the other way round. In this case, a sequence (X_k, Y_k) generated by the scheme above, converges to a stationary or critical point [5]. The limit is however not necessarily a local minimizer of the objective function and finding the global minimum

Algorithm 2 Block Coordinate Descent NMF

1: **procedure** BLOCKCOORDDESCENTMF(D, r, K)
2: Initialize $X_0 \in \mathbb{R}^{n\times r}$ and $Y_0 \in \mathbb{R}^{r\times m}$ randomly
3: **for** $k \in \{1, \dots, K\}$ **do**
4: $X_k \in \arg\min_{X\in\mathbb{R}_+^{n\times r}} F(X, Y_{k-1})$
5: $Y_k \in \arg\min_{Y\in\mathbb{R}_+^{n\times r}} F(X_k, Y)$
6: **end for**
7: **return** (X_K, Y_K)
8: **end procedure**

is NP-hard [42]. We point out that, whether the subproblems at lines 3 and 6 can be solved efficiently is crucial to the performance of the algorithm. In general, we could apply gradient descent to solve these subproblems. This method relies on the first Taylor approximation of a function. We summarize concisely this basic optimization algorithm.

Gradient Descent Let $f : \mathbb{R}^n \to \mathbb{R}$ be a continuously differentiable function and $p \in \mathbb{R}^n$. According to Taylor's theorem, it holds that

$$f(x+p) = f(x) + \nabla f(x)^T p + o(\|p\|). \tag{2.5}$$

Given a scalar $\alpha > 0$ and $p = -\alpha\nabla f(x)$, the function value of x decreases in the direction of $-\nabla f(x)$

$$\begin{aligned} & f\big(x - \alpha\nabla f(x)\big) = f(x) - \alpha\|\nabla f(x)\|^2 + o(\|\alpha\nabla f(x)\|) \\ \Leftrightarrow\ & f\big(x - \alpha\nabla f(x)\big) - f(x) = -\alpha\|\nabla f(x)\|^2 + o(\alpha\|\nabla f(x)\|) \\ \Leftrightarrow\ & \frac{f(x) - f\big(x - \alpha\nabla f(x)\big)}{\alpha\|\nabla f(x)\|} = -\|\nabla f(x)\| + \frac{o(\alpha\|\nabla f(x)\|)}{\alpha\|\nabla f(x)\|} < 0, \end{aligned}$$

if α is sufficiently small. A minimum of a function can thus be approximated by taking iterative steps that point into the direction of the negative gradient. This procedure is shown in Alg. 3. For a specified number of maximal iterations, the steps are taken with a stepsize α (line 8). The algorithm stops if the critical point is approximated sufficiently, i.e., $\|\nabla f(x_k)\| \approx 0$ (line 5) or a number of maximal iterations K is reached.

The crucial part of this algorithm is the determination of the stepsize. In the ideal case it is determined such that the function value is minimized in the direction of the negative gradient

$$\alpha^* = \arg\min_{\alpha>0} f\big(x_k - \alpha\nabla f(x_k)\big).$$

However, the approximation of this optimal stepsize is too expensive. In practice, a method is used that determines a suitable stepsize in few iterations. Examples are the Wolfe or backtracking linesearch [46]. The backtracking linesearch algorithm is denoted in Alg. 4.

Algorithm 3 Gradient Descent

1: **procedure** GradientDescent(f, x_1, ϵ, K)
2: $k \leftarrow 1$
3: **while** $k < K$ **do**
4: **if** $\|\nabla f(x_k)\| < \epsilon$ **then**
5: **break**
6: **end if**
7: Choose a stepsize $\alpha_k > 0$
8: $x_{k+1} \leftarrow x_k - \alpha_k \nabla f(x_k)$
9: $k \leftarrow k + 1$
10: **end while**
11: **return** x_k
12: **end procedure**

It is an easily implementable method to derive a stepsize that decreases the function value and that is perhaps not too small. It starts with a specified stepsize α_0 and decreases it for maximal T iterations by a factor ρ until the sufficient decrease condition (line 4) is fulfilled. The sufficient decrease condition, also known as *Armijo condition*, shall guarantee that the generated sequence of iterates x_k does not converge to a non stationary point. This might happen if the sequence of stepsizes $\alpha_k \to 0$ decreases too rapidly. Therefore, the stepsize is chosen depending on the size of the gradient. If the gradient is large, i.e., a stationary point is not yet approached, the function value in the descent direction shall also decrease much more in relation to the stepsize. The sufficient decrease condition is however satisfied for all sufficiently small stepsizes. That's why the stepsize is only decreased as much as necessary, beginning with a possibly high value of α_0. The parameters are usually set to $\alpha_0 = 1, \rho = 0.5$ and $c = 10^{-4}$.

Algorithm 4 Backtracking Linesearch

1: **procedure** BacktrackingLS($f, x_k, \alpha_0, \rho, c, T$)
2: $t \leftarrow 0$
3: **while** $t < T$ **do**
4: **if** $f\big(x_k - \alpha_t \nabla f(x_k)\big) \leq f(x_k) - c\alpha_t \|\nabla f(x_k)\|$ **then**
5: **break**
6: **end if**
7: $\alpha_{t+1} \leftarrow \rho \alpha_t$
8: $t \leftarrow t + 1$
9: **end while**
10: **return** α_t
11: **end procedure**

In contrast to the aforementioned Wolfe linesearch, the backtracking linesearch does not require an evaluation of the gradient at each iteration. It does however compute the function value in the descent direction for every step. Depending on the complexity of the function, this procedure might also be very expensive.

We might see that gradient descent is not an appropriate approach to solve the subproblems in Alg. 2. The nested determination of optimal values results in a ridiculously slow performance. As a naive and practical version of Alg. 2 we can perform instead only a single gradient descent step to decrease the function value. We refer to this method as the *projected gradient descent* which is displayed in Alg. 5. For K iterations, the matrices

Algorithm 5 Projected Gradient Descent NMF

procedure ProjGradDescentMF(D, r, K, T, F)
  Initialize $X_0 \in \mathbb{R}_+^{n\times r}$ and $Y_0 \in \mathbb{R}_+^{r\times m}$ randomly
  for $k \in \{1, \dots, K\}$ **do**
    $\alpha_k \leftarrow$ BacktrackingLS($F, (X_k, Y_k), T$)
    $X_{k+1} \leftarrow P_+\big(X_k - \alpha_k \nabla_X F(X_k, Y_k)\big)$
    $\alpha_k \leftarrow$ BacktrackingLS($F, (X_{k+1}, Y_k), T$)
    $Y_{k+1} \leftarrow P_+\big(X_{k+1} - \alpha_k \nabla_Y F(X_{k+1}, Y_k)\big)$
  end for
  return (X_K, Y_K)
end procedure

are updated in an alternating fashion by a projected gradient descent step. The function $P_+(\cdot)$ maps thereby negative entries of a matrix to zero. If the function F is equal to the RSS (2.4), the gradients are given as

$$\begin{aligned} \nabla_X F(X, Y) &= (XY - D)Y^T \\ \nabla_Y F(X, Y) &= X^T(XY - D). \end{aligned} \tag{2.6}$$

2.3.2 Algorithms for Binary Matrix Factorization

Considering BMF, few attempts have been made to derive algorithms that solve this kind of problem. The only algorithms known to me are the ones of Zhang et al. [47, 48]. They propose two algorithms. The first one has the resulting factor matrices of a NMF as input and derives a suitable threshold for each of the matrices at which the entries can be rounded by the heavyside step function $\theta : \mathbb{R} \times \mathbb{R} \to \{0, 1\}$ to one or zero

$$\theta(x, a) = \begin{cases} 1 & x > a \\ 0 & x \leq a. \end{cases}$$

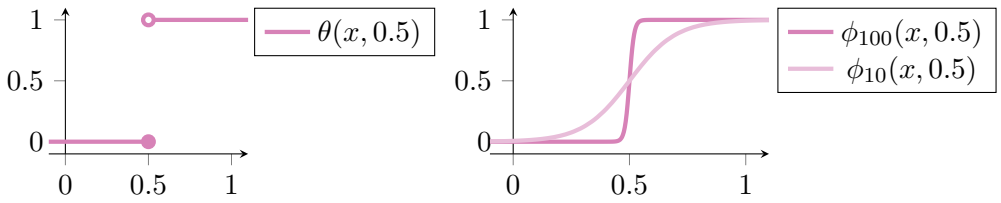


Figure 2.3: Plots of the heavyside function θ and its approximation ϕ.

We denote by the uppercase $\Theta : \mathbb{R}^{n\times r} \times \mathbb{R} \to \{0,1\}^{n\times r}$ the corresponding function applied to the entries of a matrix $\Theta(X, a) = (\theta(X_{is}, a))_{is}$. Given two NMF factor matrices X and Y, the algorithm tries to find the

$$\min_{a,b} F_\theta(a, b) = \frac{1}{2}\|D - \Theta(X, a)\Theta(Y, b)\|_F^2$$

The thresholds are obtained by a gradient descent procedure on a smooth approximation of the heavyside function $\phi_\eta : \mathbb{R} \times \mathbb{R} \to \{0,1\}$ displayed on the right in Figure 2.3

$$\phi_\eta(x, a) = \frac{1}{1 + e^{\eta(a-x)}}$$

with Wolfe or backtracking linesearch. Again, we denote by the uppercase formulation $\Phi_\eta : \mathbb{R}^{n\times r} \to \{0,1\}^{n\times r}$, $\Phi_\eta(X, a) = (\phi_\eta(X_{is}, a))_{is}$ the equivalent function applied to a matrix. Therewith, the objective of the algorithm changes to

$$\min_{a,b} F_\phi(a, b) = \frac{1}{2}\|D - \Phi_\eta(X, b)\Phi_\eta(Y, b)\|^2.$$

The minimization of the objective function is drafted in Alg. 6. The matrices X and Y are initialized by an arbitrary nonnegative matrix factorization algorithm NMF and suitable thresholds are derived by a common gradient descent procedure. The derivative of F_ϕ can

Algorithm 6 Threshold BMF

procedure ThresholdBMF(D, r, K, T)
  $(X, Y) \leftarrow \mathrm{NMF}(D, r)$
  $(a_0, b_0) \leftarrow (0.5, 0.5)$
  for $k \in \{1, \ldots, K\}$ **do**
    $\alpha_k \leftarrow$ BacktrackingLS$(F_\phi, (a_k, b_k), T)$
    $(a_{k+1}, b_{k+1}) \leftarrow (a_k, b_k) - \alpha_k \nabla F_\phi(a_k, b_k)$
  end for
  return $\big(\Theta(X, a_K), \Theta(Y, b_K)\big)$
end procedure

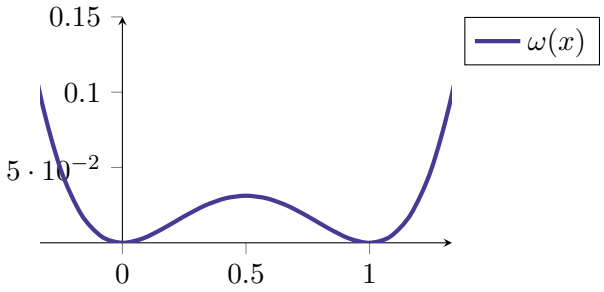


Figure 2.4: Plot of the function ω.

be calculated by a simple application of the chain rule. We state here only the derivative of the function ϕ_η which we will need later

$$\frac{\partial}{\partial x}\phi_\eta(x,a) = \phi_\eta(x,a)\big(1-\phi_\eta(x,a)\big). \tag{2.7}$$

As evaluated in [48] this algorithm yields good results if the data matrix is sparse. The second algorithm aims at solving the BMF problem in terms of nonlinear programming. The function $\omega : \mathbb{R} \to \mathbb{R}_+$, $\omega(x) = \frac{1}{2}(x^2 - x)^2$ attains its minimum at binary values (see Figure 2.4) and can be utilized to determine if a matrix is binary

$$\begin{aligned} \min_{X,Y} \quad & \frac{1}{2}\|D - XY\|^2 \\ s.t. \quad & \sum_{i,s} \omega(X_{is}) = 0 \\ & \sum_{s,j} \omega(Y_{sj}) = 0. \end{aligned}$$

Given a weighting parameter $\lambda \in \mathbb{R}^+$, this problem can be solved by the introduction of penalizing terms

$$\begin{aligned} \min_{X,Y} F_\omega(X,Y) &= \frac{1}{2}\|D - XY\|^2 + \frac{\lambda_1}{2}\|X * X - X\|^2 + \frac{\lambda_2}{2}\|Y * Y - Y\|^2 \\ &= F(X,Y) + \lambda_1 \sum_{i,s} \omega(X_{is}) + \lambda_2 \sum_{s,j} \omega(Y_{sj}). \end{aligned} \tag{2.8}$$

The minimization is performed by alternating multiplicative updates that are equivalent

to gradient descent steps with the longest stepsize that preserves nonnegativity. An outline of this procedure is given in Alg. 7. Since no linesearch method is applied, the generated sequence of function values $F_\omega(X_k, Y_k)$, $k \in \mathbb{N}$ is not necessarily nonincreasing. Zhang et

Algorithm 7 Penalizing BMF

1: **procedure** PENALBMF(D, r, λ, K)
2: Initialize $X_0 \in \mathbb{R}^{n\times r}$ and $Y_0 \in \mathbb{R}^{r\times m}$ randomly
3: **for** $k \in \{1, \dots, K\}$ **do**
4: $(X_{k+1})_{is} = (X_k)_{is} \frac{(DY_k^T)_{is} + 3\lambda(X_k)_{is}^2}{(X_k Y_k Y_k^T)_{is} + 2\lambda(X_k)_{is}^3 + \lambda(X_k)_{is}}$
5: $(Y_{k+1})_{sj} = (Y_k)_{sj} \frac{(X_{k+1}^T D)_{sj} + 3\lambda(Y_k)_{sj}^2}{(X_{k+1}^T X_{k+1} Y_k)_{sj} + 2\lambda(Y_k)_{sj}^3 + \lambda(Y_k)_{sj}}$
6: **end for**
7: **return** $(\Theta(X_K, 0.5), \Theta(Y_K, 0.5))$
8: **end procedure**

al. pointed out that this algorithm performs superior on rather dense matrices [48]. The derivative of the function ω is stated as

$$\frac{d}{dx}\omega(x) = (x^2 - x)(2x - 1). \tag{2.9}$$

Chapter 3

Algorithms

In this chapter we review existing approaches and develop new views on the derivation of the best encoding by a code table. First, I describe the heuristic procedures Krimp, Slim and SHrimp. We will see how a set of frequent patterns can be filtered, how the contribution of a pattern to the encoding can be estimated and what insights are provided if we apply a tree data structure to represent the encoding. We discuss how these procedures might be improved, either in speed or with respect to the quality of the returned result. Then, we have a look at the representation of an encoding by a matrix factorization. This introduces a new point of view on Krimp related problems and draws a connection to the task of clustering.

3.1 Krimp

Krimp [45] is the first algorithm that has been proposed to find the code table that compresses the database best. To determine a suitable encoding out of the enormous amount of possible ones (see Section 2.1.3), Siebes et al. indicate to apply an heuristic fixed order of the coding set in that the items of each transaction are encoded. In a greedy procedure, every itemset from a pool of candidates is considered once and added to the coding set if that improves the compression size. To refine the coding set during the candidate selection, mechanisms of pruning are exerted.

Let's have a look at the procedure of Krimp as it is stated in Alg. 8. The input is given by the database $\mathcal{D}$ and the frequent patterns of a preferably low minimum support $\mathcal{F}$. The frequent patterns are sorted in the order in that they are proceeded, the *standard candidate order*. That is first decreasing on support, second decreasing on cardinality and at last lexicographically (line 2). Therewith, the most frequent patterns with the potential to get shorter codes by a high usage, are considered first. Longer patterns that cover many items by one code, are preferably regarded under those with equal support. The code table is initialized to the standard code table (line 3) and revised while the set of

Algorithm 8 Krimp

```
1: procedure KRIMP(𝒟, ℱ)
2:     ℱ ← sort(ℱ)                          ▷ in standard candidate order
3:     CT ← STANDARDCODETABLE(𝒟)
4:     for X̂ ∈ ℱ \ ℐ do
5:         CT̂ ← CT ∪ X̂
6:         if L(𝒟, CT̂) < L(𝒟, CT) then
7:             CT ← PRUNING(CT̂, CT)
8:         end if
9:     end for
10:    return CT
11: end procedure
```

candidate patterns is traversed. A candidate is added to the code table if this reduces the compression size (lines 4-9).

Since the compression size decreases monotonically in the number of regarded candidates, this procedure achieves the best compression if as many candidates as possible are regarded, i.e., the minimum support is set to $\frac{1}{|\mathcal{D}|}$. This set of occurring patterns is difficult to maintain in memory for many databases and in practice, a suitable minimum support has to be put.

Computing the Usage. To compute the resulting compression size for every candidate pattern, the corresponding usage function has to be computed. The usage calculation relies on the method ENCODE (Alg. 9). It is invoked for every transaction that contains the currently regarded candidate pattern. Assuming that the code table is sorted by a total order, the algorithm traverses the code table and selects the first pattern that is contained in the specified transaction (line 2). The used order for this procedure is called *standard encoding order* that is first decreasing on cardinality, second decreasing on support and at last lexicographically. If the transaction is encoded completely by elements of the code table, the algorithm stops (line 3), otherwise the procedure is called recursively for the uncovered part of the transaction (line 6). This procedure is called for every transaction and each of the numerous candidate patterns in the worst case. An efficient implementation of this procedure is thus crucial to the performance of the algorithm.

3.1.1 Pruning

The insertion of an itemset to the code table might decrease the usage of other code table elements. A decrease in the usage results in an increase of the code length. It is therefore possible that a removal of those patterns with a decreased usage improves the compression

Algorithm 9 Encoding of a transaction

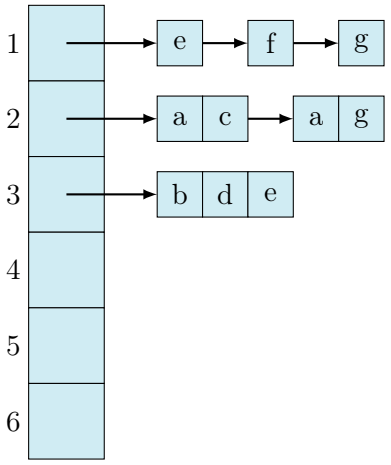
1: **procedure** ENCODE(t, CT)
2: $X^* \leftarrow \min\{X | X \subseteq t \wedge X \in CT\}$
3: **if** $t \setminus X^* \leftarrow \emptyset$ **then**
4: **return** $\{X^*\}$
5: **else**
6: **return** $\{X^*\} \cup$ ENCODE($t \setminus X^*$)
7: **end if**
8: **end procedure**

size. The pruning method Alg. 10 regards all patterns whose usage has been decreased and decides for each of these patterns if it should be removed. The input of this algorithm is a code table CT and a modified version $\widehat{CT}$. The set of pruning candidates is denoted by P, that is initialized to all patterns that have a lower usage in the modified code table (line 2). The pruning candidates are regarded in an order such that the ones with the lowest usage are regarded first. If the removal of a pruning candidate improves the compression, the candidate is took off and the pruning candidates are extended to the ones whose usage is decreased by this action (lines 7- 10). This procedure is repeated until all pruning candidates have been considered. As it has been evaluated for 27 datasets in [45], the

Algorithm 10 Pruning of a Code Table

1: **procedure** PRUNING($\widehat{CT}, CT$)
2: $P \leftarrow \{X \in CT | usage_{\widehat{CT}}(X) < usage_{CT}(X)\}$
3: **while** $P \neq \emptyset$ **do**
4: $\widehat{X} \leftarrow \arg\min_{X \in \widehat{CT}}\{usage_{\widehat{CT}}(X)\}$
5: $P \leftarrow P \setminus \widehat{X}$
6: $\widetilde{CT} \leftarrow \widehat{CT} \setminus \widehat{X}$
7: **if** $L(\mathcal{D}, \widetilde{CT}) < L(\mathcal{D}, \widehat{CT})$ **then**
8: $P \leftarrow P \cup \{X \in \widetilde{CT} | usage_{\widetilde{CT}}(X) < usage_{\widehat{CT}}(X)\}$
9: $\widehat{CT} \leftarrow \widetilde{CT}$
10: **end if**
11: **end while**
12: **return** $\widehat{CT}$
13: **end procedure**

application of pruning enhances the resulting compression size for all examined datasets at least a bit. More significant is however the reduction in the number of used code table elements that can be achieved by pruning, e.g. from 4046 elements to 467 elements for the *Accidents* dataset.



CT	freq
$\{b, d, e\}$	4
$\{a, c\}$	4
$\{a, g\}$	4
$\{e\}$	6
$\{f\}$	4
$\{g\}$	4

Figure 3.1: Representation of a code table by a list of lists. The vertical array stores pointers to the lists of code table elements whose cardinality is equal to the number of the cell in the array.

3.1.2 Computational Details

The method Encode as it is displayed in Alg. 9 is easy to implement, but that is not how the usage should be determined in practice. There are multiple tricks that can be used to reduce the computation time of the compression size with respect to the candidate code table. First, an implementation of transactions and patterns as bitvectors with $|\mathcal{I}|$ bits allows for a fast check if a candidate pattern is contained in a transaction. Second, an implementation of the code table as an array of lists as displayed in Figure 3.1 enables a fast determination of the insertion point of a candidate set. The array contains at position i a list with all the code table elements that have a cardinality of i. The elements in the list are stored decreasingly on their support. Thus, the insertion point of a candidate pattern is always at the end of the list at the array position which is equal to its cardinality. That is because candidate patterns are regarded decreasingly on their frequency. Third, for a transaction t and a candidate pattern $\widehat{X} \subseteq t$, it should be ascertained first if $\widehat{X}$ is used for the encoding of t at all. If there is a pattern $X \in CT$ that is preceding to $\widehat{X}$ with regard to the standard encoding order, that is used for the encoding $X \in \mathit{encode}(t)$ and that has a nonempty intersection $X \cap \widehat{X} \neq \emptyset$, then $\widehat{X} \notin \mathit{encode}(t)$. Otherwise, the changes in the usage for itemsets that are preceded by $\widehat{X}$ can be determined afterwards.

3.1.3 Complexity

The computational expensive part in Krimp is the determination of the compression size for each candidate code table. Therefore, the candidate usage function has to be obtained and the whole code table has to be traversed for every transaction in the worst case. Checking whether a code table element is used for the encoding of t or not is done in $\mathcal{O}(|I|)$ time by checking if $X \subseteq t$ in Alg. 9. Thus, the computation of the usage function

given a code table CT takes $\mathcal{O}(|\mathcal{D}||CT||\mathcal{I}|)$ time. The resulting compression size can be obtained in $\mathcal{O}(|CT|)$ when the database description size is computed by Eq. (2.2). The overall computation time, regarding all of the candidates $\mathcal{F}$ results therewith in a complexity of

$$\mathcal{O}(|\mathcal{F}||\mathcal{D}||CT||\mathcal{I}|).$$

CT denotes the output code table, which is the largest code table obtained during the algorithm. If we apply pruning, all elements of the code table have to be reconsidered for every candidate in the worst case. Taking the time $\mathcal{O}(|\mathcal{F}|\log(\mathcal{F}))$ for sorting the candidates in the beginning into account, we obtain an overall used time of

$$\mathcal{O}(|\mathcal{F}|(\log|\mathcal{F}| + |\mathcal{D}||\overline{CT}|^2|\mathcal{I}|)),$$

when $\overline{CT}$ denotes the set of all candidate patterns that have been accepted during the runtime of the algorithm.

Siebes et al. state in [45], that the factor $|\mathcal{D}||CT||\mathcal{I}|$ can be neglected in the big-O notation. They argue that for transactions and patterns that are implemented as bitvectors, i.e., as an array of 64 bit integers, a subset check is done in constant time and the identification of relevant transactions for a candidate pattern takes therefore also constant time. Further, they indicate that the number of relevant transactions d whose encoding changes with the integration of a candidate and the number of elements in the code table are comparably small $|CT|, d << |\mathcal{D}| << |\mathcal{F}|$ and can thus be regarded as a constant. These considerations result in a time complexity of $\mathcal{O}(|\mathcal{F}|\log|\mathcal{F}|+|\mathcal{F}|)$, but the conclusions are questionable.

A bitvector implementation of a set with at most $|\mathcal{I}|$ elements needs still $\mathcal{O}\left(\frac{|\mathcal{I}|}{64}\right) = \mathcal{O}(|\mathcal{I}|)$ operations to make a subsetcheck. Thus, an identification of the relevant transactions needs at least $|D|$ operations and overall, at least $\mathcal{O}(|\mathcal{F}|(|D|+\log|\mathcal{F}|))$ operations are required. An even more severe issue, that is difficult to handle in practice, is the memory requirement of KRIMP that is dominated by the storage of frequent patterns. The algorithm SLIM , discussed in Section 3.2, and the matrix factorization formulations discussed in Section 3.4 overcome this issue.

3.2 Slim

Smets et al. propose in [41] the algorithm SLIM , that mines the candidate patterns directly from the current code table. The same heuristic assumptions are made as in KRIMP , i.e., the encoding of a transaction is determined by the standard encoding order and codes may not overlap. Candidates are generated by a combination of code table elements and a heuristic estimation of the compression improvement that induces a candidate to the code table is applied to determine the next candidate. To reduce the number of considered combinations, heuristic bounds are applied and only the set of top-k candidates is derived in each iteration.

3.2.1 Mining good Candidates

The intuition behind SLIM is that the pattern that decreases the compression size most should be added to the code table in every iteration. The decrease of the compression size that goes along with the insertion of a candidate pattern $\widehat{X}$ to a code table CT is measured as the *compression gain* of the candidate code table $\widehat{CT} = CT \cup \widehat{X}$ and is defined as

$$\begin{aligned}\Delta L(\widehat{CT}, \mathcal{D}) &= L(CT, \mathcal{D}) - L(\widehat{CT}, \mathcal{D}) \\ &= \Delta L(\mathcal{D} \,|\widehat{CT}) + \Delta L(\widehat{CT}|\, \mathcal{D}).\end{aligned}$$

The computation of the gain involves the computation of the compression size with regard to all possible candidates as it is done in KRIMP for the frequent patterns. This results in an algorithm that requires about as many operations as KRIMP , but for every iteration when a pattern is merged to the code table. This is not computationally feasible for many databases. Hence, the impact that an integrated pattern has on the usage function and therewith on the compression size, is estimated. Furthermore, the pool of candidates is restricted to the ones that can be generated by the union of two code table elements. Given two code table entries X_1 and X_2, the usage of the candidate pattern $\widehat{X} = X_1 \cup X_2$ is bounded below by the number of transactions that are encoded by both patterns X_1 and X_2

$$usage(\widehat{X}) \geq |\{t \in \mathcal{D} \,|\{X_1, X_2\} \subseteq encode(t)\}|. \tag{3.1}$$

That is, because the pattern $\widehat{X}$ is preceding to the elements X_1, X_2 in the encoding order as it contains more items, and therewith all simultaneous encodings of X_1 and X_2 are replaced by $\widehat{X}$. However, there might also be some transactions that are encoded by a pattern $Y \preceq X_{1/2}$ that prevents an encoding by $X_{1/2}$, but that would however be dominated by the candidate $\widehat{X} \preceq Y$. These transactions are not taken into account by the set on the right size of Eq. (3.1), and the estimation is therefore not exact. The bound can however be used to estimate the compression improvement that comes in with a candidate pattern.

Estimating the Compression Gain For a brief notation, we denote in the following by u_{CT} the usage with respect to a code table CT and by $\tilde{u}_{\widehat{CT}}$ the estimated usage with respect to the candidate code table. We assume that the usage changes only for the candidate and the joined patterns

$$\tilde{u}_{\widehat{CT}}(X) = \begin{cases} |\{t \in \mathcal{D} \,|\{X_1, X_2\} \subseteq encode(t)\}| & X = \widehat{X} \\ u_{CT}(X_i) - \tilde{u}_{\widehat{CT}}(\widehat{X}) & X \in \{X_1, X_2\} \\ u_{CT}(X_j) & X \in CT \setminus \{X_1, X_2\}. \end{cases}$$

That way, the cascading effects on the usage of code table elements that are succeeding to X_1 or X_2 are not taken into account, but those effects are hard to predict and not often dramatic [41]. With the uppercase formulations $\widetilde{U}_{\widehat{CT}} = \sum_{X \in \widehat{CT}} \tilde{u}_{\widehat{CT}}(X)$ and $U_{CT} = \sum_{X \in CT} u_{CT}(X)$, we denote the sums of the usage of all corresponding code table elements. The compression gain of the description size of the database is by these assumptions estimated as

$$\begin{aligned} \Delta\widetilde{L}(\mathcal{D} \,|\widehat{CT}) &= L(\mathcal{D} \,|CT) - \widetilde{L}(\mathcal{D} \,|\widehat{CT}) \\ &= -\sum_{X \in CT} u_{CT}(X) \log\left(\frac{u_{CT}(X)}{U_{CT}}\right) + \sum_{X \in \widehat{CT}} \tilde{u}_{\widehat{CT}}(X) \log\left(\frac{\tilde{u}_{\widehat{CT}}(X)}{\widetilde{U}_{\widehat{CT}}}\right) \\ &= -\sum_{X \in \{X_1, X_2\}} \left(u_{CT}(X) \log\left(u_{CT}(X)\right) - \tilde{u}_{\widehat{CT}}(X) \log\left(\tilde{u}_{\widehat{CT}}(X)\right)\right) \\ &\quad + \tilde{u}_{\widehat{CT}}(\widehat{X}) \log\left(\tilde{u}_{\widehat{CT}}(\widehat{X})\right) + U_{CT} \log\left(U_{CT}\right) - \widetilde{U}_{\widehat{CT}} \log\left(\widetilde{U}_{\widehat{CT}}\right). \end{aligned} \tag{3.2}$$

Correspondingly, the compression gain of the model description size is estimated by

$$\begin{aligned} \Delta\widetilde{L}\left(\widehat{CT} |\, \mathcal{D}\right) &= L(CT |\, \mathcal{D}) - \widetilde{L}\left(\widehat{CT} |\, \mathcal{D}\right) \\ &= -\sum_{X \in CT} \left(\log\left(\frac{u_{CT}(X)}{U_{CT}}\right) + \sum_{x \in X} \log(\text{supp}(x))\right) \\ &\quad + \sum_{X \in \widehat{CT}} \left(\log\left(\frac{u_{\widehat{CT}}(X)}{\widetilde{U}_{\widehat{CT}}}\right) + \sum_{x \in X} \log(\text{supp}(x))\right) \\ &= -\sum_{X \in \{X_1, X_2\} \cap \widehat{CT}} \left(\log\left(u_{CT}(X)\right) - \log\left(\tilde{u}_{\widehat{CT}}(X)\right)\right) \\ &\quad - \sum_{X \in \{X_1, X_2\} \setminus \widehat{CT}} \left(\log\left(u_{CT}(X)\right) + \sum_{x \in X} \log(\text{supp}(x))\right) \\ &\quad + \log\left(\tilde{u}_{\widehat{CT}}(\widehat{X})\right) + \sum_{\hat{x} \in \widehat{X}} \log(\text{supp}(\hat{x})) \\ &\quad + |CT| \log\left(U_{CT}\right) - |\widehat{CT}| \log\left(\widetilde{U}_{\widehat{CT}}\right). \end{aligned} \tag{3.3}$$

The formulation (3.3) describes the changes in the code length of X_1 and X_2 if the patterns is still used in the encoding by the candidate code table (first line) or the changes

that take place if one code is not used anymore, which might happen if $\tilde{u}_{\widehat{CT}}(\hat{X}) = \min\{u_{CT}(X_1), u_{CT}(X_2)\}$ (second line). The last two lines indicate the description size of the candidates code and the affects on the sum of the usage.

The Algorithm The algorithm SLIM (Alg. 11) is very similar to KRIMP , the only difference is that candidate patterns are obtained dynamically and depending on the current code table. Therefore, SLIM needs as input only the database $\mathcal{D}$ and no frequent patterns. The code table is initialized to the standard code table (line 2) and candidates are accepted greedily whenever that improves the compression size. The candidates are generated by unions of code table patterns and selected according to the gain order, i.e., the candidate that improves the approximate compression size most is examined first (line 3). A candidate pattern is added to the code table if that improves the compression size (lines 4-7). If a candidate is accepted, the method of pruning as described in Section 3.1.1 is applied (line 6) and the set of candidate patterns is updated.

Algorithm 11 Slim

```
1:  procedure SLIM(D)
2:      CT ← STANDARDCODETABLE(D)
3:      for X̂ ∈ {X1 ∪ X2 | X1, X2 ∈ CT} do                ▷ in gain order
4:          CT̂ ← CT ∪ X̂
5:          if L(D, CT̂) < L(D, CT) then
6:              CT ← PRUNING(CT̂, CT)
7:          end if
8:      end for
9:      return CT
10: end procedure
```

Determining the best Candidate

The identification of the next best candidate in line 3 is computational expensive. New candidates have to be generated and sorted by the estimated compression gain after one of them has been accepted. Since the first few candidates with a high estimated compression gain are likely to improve the compression size indeed, only the top-k candidates that have not been regarded for the current code table, are provided in the for loop of line 3. To discover the top-k candidates in a reasonable time, further heuristics are applied.

The possible patterns $X_1 \in CT$ are considered descending on usage and candidates $\widehat{X} = X_1 \cup X_2$ are generated by a union with a pattern X_2 succeeding to X_1 with respect to its usage, i.e., $u_{CT}(X_1) \geq u_{CT}(X_2)$. Therewith, the usage of a candidate $u_{\widehat{CT}}(\widehat{X}) \leq u_{CT}(X_2) \leq u_{CT}(X_1)$ is bounded above by the usage of X_1 or X_2 and the lower the usage

of X_2 is, the less impact does the insertion of the candidate have on the compression. This observation is used to determine which combinations of patterns are potential top-k candidates.

For every pattern X_1, it is decided if the usage is large enough to grant a sufficiently good estimated compression gain. Assuming that all occurrences of X_1 or X_2 are encoded by $\widehat{X}$, and that the usage of X_2 is maximal $u_{CT}(X_1) = u_{CT}(X_2)$, an upper bound on the estimated compression gain of the database description size is approximated by

$$\widetilde{\Delta}\widetilde{L}(\mathcal{D}\,|CT \cup \{X_1\}) = -2u_{CT}(X_1)\log\frac{u_{CT}(X_1)}{U_{CT}} + u_{CT}(X_1)\log\frac{u_{CT}(X_1)}{U_{CT} - u(X_1)}. \tag{3.4}$$

We note, that this guess on the maximal compression gain is not equal to the estimated compression gain (3.2) under the above assumptions, since changes of the usage sum are not reflected. The formula provides however a reasonable estimate on the achievable compression gain and if this maximal compression gain $\widetilde{\Delta}\widetilde{L}(\mathcal{D}\,|CT \cup \{X_1\}) < \Delta\widetilde{L}(\mathcal{D}, CT \cup \{\widehat{X}_k\})$ is less than the estimated compression gain for the the last candidate $\widehat{X}_k$ of the current collection of top-k candidates, then the obtained set of top-k patterns is assumed to be complete and the candidate generation algorithm stops.

A similar threshold is applied to decide whether a candidate $\widehat{X} = X_1 \cup X_2$ can be excluded from being in the top-k if the usage of X_2 is small. Herefore, it is assumed that all occurrences of X_2 are encoded by $\widehat{X}$ and thus the maximal database description gain is approximated by

$$\begin{aligned}\widetilde{\Delta}\widetilde{L}(\mathcal{D}\,|CT \cup \{X_1 \cup X_2\}) = &-u_{CT}(X_1)\log\frac{u_{CT}(X_1)}{U_{CT}} - u_{CT}(X_2)\log\frac{u_{CT}(X_2)}{U_{CT}} \\ &+ (u_{CT}(X_1) - u_{CT}(X_2))\log\frac{u_{CT}(X_1) - u_{CT}(X_2)}{U_{CT} - u(X_2)} \\ &+ u_{CT}(X_2)\log\frac{u_{CT}(X_2)}{U_{CT} - u(X_2)}.\end{aligned} \tag{3.5}$$

The terms of the first row describe the fact that the old codes of X_1 and X_2 are not used anymore. The usage of X_2 decreases to zero and the usage of X_1 decreases by the usage of X_2 while the candidate is introduced with the usage of X_2, so the sum of all usages in the candidate code table is equal to $U_{CT} - u_{CT}(X_2)$. The corresponding affects on the description size by the candidate code table are given in the second (encoding size of X_2) and third line (encoding size of the candidate pattern). Again, the affect on the code lengths of other codes by a decreased usage sum is not taken into account. Also, the effects on the compression size of the code table, i.e., on the model description size, are hard to predict and are also not considered in this estimate.

If the maximal achievable compression gain (3.5) is greater than the lowest estimated compression size of the top-k candidates $\widetilde{\Delta}\widetilde{L}(\mathcal{D}\,|CT \cup \{X_1\}) > \Delta\widetilde{L}(\mathcal{D}, CT \cup \{\widehat{X}_k\})$, the actual estimated compression size of $\widetilde{L}(\mathcal{D}, CT \cup \{\widehat{X}\})$ is calculated and the candidate is added to the set of top-k patterns depending on the result. Otherwise, the next pattern X_1 and its combinations are regarded.

3.2.2 Bounding the maximal achievable gain

The code tables of SLIM yield superior compressions for many databases in comparison to the ones of KRIMP. However, the algorithm uses several heuristics and heuristic approximations of the used heuristics. Therefore, we derive now an actual upper bound on the estimated compression gain. Afterwards, we have a look on the differences to the used heuristic.

3.2.1 Proposition. *Given a code table CT and two patterns $X_1, X_2 \in CT$, we denote by $\widehat{CT} = CT \cup \{\widehat{X}\}$ the candidate code table that is created by the insertion of a pattern $\widehat{X} = X_1 \cup X_2$ into CT. Let $X \in \{X_1, X_2\}$ be one of the joined patterns, the estimated compression gain of the database description size (3.2) is then bounded above by*

$$\Delta\widetilde{L}\left(\mathcal{D} \mid \widehat{CT}\right) \leq (U_{CT} - u_{CT}(X)) \log\left(\frac{U_{CT}}{U_{CT} - u_{CT}(X)}\right) - u_{CT}(X) \log\left(\frac{u_{CT}(X)}{U_{CT}}\right). \quad (3.6)$$

Proof. A reformulation of the compression gain in Eq. (3.2), where the logarithm of a fraction is not calculated by a sum, yields

$$\begin{aligned}\Delta\widetilde{L}(\mathcal{D}\,|\widehat{CT}) = &\sum_{X \in CT \setminus \{X_1, X_2\}} \left(u_{CT}(X) \left(\log\left(\frac{u_{CT}(X)}{\widetilde{U}_{\widehat{CT}}}\right) - \log\left(\frac{u_{CT}(X)}{U_{CT}}\right)\right)\right) \\ &+ \sum_{X \in \{X_1, X_2\}} \left(\tilde{u}_{\widehat{CT}}(X) \log\left(\frac{\tilde{u}_{\widehat{CT}}(X)}{\widetilde{U}_{\widehat{CT}}}\right) - u_{CT}(X) \log\left(\frac{u_{CT}(X)}{U_{CT}}\right)\right). \\ &+ \tilde{u}_{\widehat{CT}}(\widehat{X}) \log\left(\frac{\tilde{u}_{\widehat{CT}}(\widehat{X})}{\widetilde{U}_{\widehat{CT}}}\right)\end{aligned}$$

The logarithmic terms in the first line can be shortened by an application of the logarithmic properties and the sum over all usages excluding X_1 and X_2, can be calculated by means of the usage sum U_{CT}. The usage estimations in the argument of the logarithm function of the second line can be bounded above by $u_{CT}(X_1)$ or $u_{CT}(X_2)$, such that

$$\begin{aligned}\Delta\widetilde{L}(\mathcal{D}\,|\widehat{CT}) \leq &(U_{CT} - u_{CT}(X_1) - u_{CT}(X_2)) \log\left(\frac{U_{CT}}{\widetilde{U}_{\widehat{CT}}}\right) \\ &+ \sum_{X \in \{X_1, X_2\}} \left(\tilde{u}_{\widehat{CT}}(X) \log\left(\frac{u_{CT}(X)}{\widetilde{U}_{\widehat{CT}}}\right) - u_{CT}(X) \log\left(\frac{u_{CT}(X)}{U_{CT}}\right)\right) \\ &+ \tilde{u}_{\widehat{CT}}(\widehat{X}) \log\left(\frac{\tilde{u}_{\widehat{CT}}(\widehat{X})}{\widetilde{U}_{\widehat{CT}}}\right).\end{aligned}$$

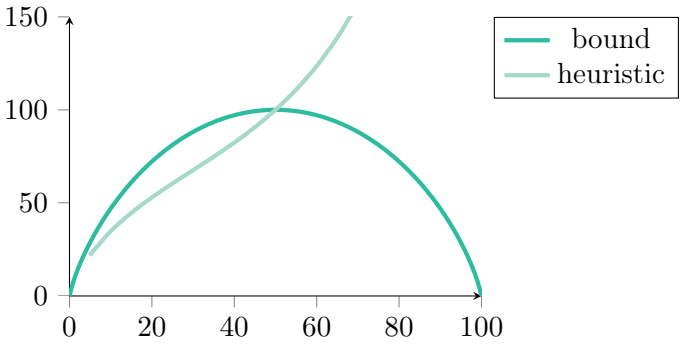


Figure 3.2: Plot of the heuristic and the actual bound on the maximal achievable gain if the usage of one of the joined patterns is given by the values of the x-axis and the overall sum of usages is equal to 100.

Sorting the logarithmic terms in the first line according to the usage factors yields

$$\begin{aligned}\Delta\widetilde{L}(\mathcal{D}\,|\widehat{CT}) \leq\; & U_{CT}\log\left(\frac{U_{CT}}{\widetilde{U}_{\widehat{CT}}}\right) + \tilde{u}_{\widehat{CT}}(\widehat{X})\log\left(\frac{\tilde{u}_{\widehat{CT}}(\widehat{X})}{\widetilde{U}_{\widehat{CT}}}\right) \\ & + \sum_{X\in\{X_1,X_2\}} (\tilde{u}_{\widehat{CT}}(X) - u_{CT}(X))\log\left(\frac{u_{CT}(X)}{\widetilde{U}_{\widehat{CT}}}\right) \\ =\; & U_{CT}\log\left(\frac{U_{CT}}{\widetilde{U}_{\widehat{CT}}}\right) \\ & + \tilde{u}_{\widehat{CT}}(\widehat{X})\left(\log\left(\frac{\tilde{u}_{\widehat{CT}}(\widehat{X})}{\widetilde{U}_{\widehat{CT}}}\right) - \sum_{X\in\{X_1,X_2\}}\log\left(\frac{u_{CT}(X)}{\widetilde{U}_{\widehat{CT}}}\right)\right),\end{aligned}$$

where the last equation is obtained by the fact that the number of transactions that are not encoded by X_1 or X_2 is equal to the estimated usage of the candidate $\tilde{u}_{\widehat{CT}}(X_{1/2}) - u_{CT}(X_{1/2}) = \tilde{u}_{\widehat{CT}}(\widehat{X})$. The following steps are exchangeable for X_1 and X_2 because the estimated candidate usage is bounded above by the usage of X_1 or X_2. Depending on the chosen bound, the estimated compression gain is bounded in terms of X_1 or X_2. We bound the candidate usage that determines the code length of the candidate pattern now by X_2 and get

$$\Delta\widetilde{L}(\mathcal{D}\,|\widehat{CT}) \leq U_{CT}\log\left(\frac{U_{CT}}{\widetilde{U}_{\widehat{CT}}}\right) - \tilde{u}_{\widehat{CT}}(\widehat{X})\log\left(\frac{u_{CT}(X_1)}{\widetilde{U}_{\widehat{CT}}}\right).$$

The estimated usage of the candidate in the sum $\widetilde{U}_{\widehat{CT}} = U_{CT} - \tilde{u}_{\widehat{CT}}(\widehat{X})$ and as the factor of the code length, can then be limited above by $u_{CT}(X_1)$ and a summarization of the terms yields the final upper bound (3.6). □

Figure 3.2 shows the differences between the heuristic bound and the one of Eq. (3.6). The symmetry of the proven bound reflects the fact that if one of the joined patterns has a larger usage than half of the overall usage sum, the other pattern must have a respectively

smaller one. This case might however occur seldom in practice. If the usage of one joined pattern is smaller than half of the usage sum, the heuristic bound is smaller. The heuristic bound might therefore be too close and neglect some patterns that are able to improve the estimated compression gain much more than guessed. The tendencies are however similar for both bounds. The proof of an actual bound in this setting is solely a first approach to consolidate the applied heuristics theoretically.

3.3 SHrimp

SHRIMP [23] is an algorithm that uses a different data structure to represent the database and that aims at enabling an efficient computation of changes in the underlying encoding for a candidate code table. The database is maintained in a tree that reflects the present encoding and provides therewith different possibilities to evaluate the impact that the insertion of a pattern has on the compression size. Furthermore, a visualization of the encoded database is obtained. This can be used to explore the database in different ways than it was possible before. The constitution of the database tree is similar to that of the *FP-tree*, introduced by Han et al. in the popular FP-Growth algorithm [22]. The nodes of an FP-tree contain yet single items while sets of items are represented by nodes in SHRIMP. The order of items in the FP-tree is determined by the frequency, similarly the standard encoding order that is used in KRIMP and SLIM, induces the placement of the nodes in the SHRIMP -tree.

3.3.1 Utilizing Tree Structures

The lack of insight how code tables can be extended such that the compression size decreases, forces the algorithms regarded so far to try different constellations of patterns out and to preserve the one that encodes best. The algorithm SLIM is thereby able to reduce the number of regarded candidates by 2 orders-of-magnitude [41], but still, the number of materialized candidates is extremely large in comparison to the amount of accepted patterns in the code table. Therefore, the underlying changes in the encoding have to be examined again and again for candidate after candidate. The determination of these changes includes always an identification of affected parts of the database: the transactions whose encoding changes with an alteration of the coding set, to calculate or estimate resulting changes in the compression size.

With SHRIMP , we examine to what extend the indexing nature of trees can be exploited in order to identify the parts of a database concerned to a change of the coding set. Furthermore, we consider next to the representation of the database as a tree, a constitution of the code table as a tree, reflecting the slightly recursive encoding of codes by the standard code table in the model description.

The Database Tree

The tree is built such that each branch from the root to a leaf represents a transaction and provides the information about all possible and the currently applied encoding, given a code table. We notate nodes and respective sets in the fraktur font. The tree that represents the database is defined by its constituting nodes as follows.

3.3.1 Definition (database tree). Given a total order on itemsets $\preceq$, a database tree is a tree structure with the following properties

1. The root of the tree is labeled as *null*.
2. Each node $\mathfrak{n} \neq null$ is described by a pattern $X_\mathfrak{n}$, a set of inactive items $inact(\mathfrak{n})$, a counter $use(\mathfrak{n})$ and the pointers to its children $children(\mathfrak{n})$ and the parent node $parent(\mathfrak{n})$. $use(\mathfrak{n})$ denotes the number of transactions represented by the branch from the root to that node.
3. For a node $\mathfrak{n}$ and the parent node $\mathfrak{p} = parent(\mathfrak{n}) \neq null$ it holds that

$$X_\mathfrak{p} \preceq X_\mathfrak{n}$$

The total order that is used to determine the database tree hierarchy is given by the standard encoding order. In general, it is possible to reflect any way in that transactions are encoded by the application of different orders. The definition poses that a transaction is described by the nodes from the root to a node whose counter *use* is larger than the sum of all *use* counters from its children. Nodes that satisfy this property are called *leaves*, although these nodes may also have children nodes. Nevertheless, at least one branch ends at these nodes. We describe now the meaning of the field *inact* technically.

3.3.2 Definition. Let $\mathfrak{n}$ be a node with a non-singleton pattern $|X_\mathfrak{n}| \geq 2$ and let $\text{anc}(\mathfrak{n})$ denote the ancestor nodes of $\mathfrak{n}$. For an item $x \in X_\mathfrak{n}$ it holds that

$$x \in inact(X_\mathfrak{n}) \Leftrightarrow \exists \mathfrak{a} \in \text{anc}(\mathfrak{n}) : inact(\mathfrak{a}) = \emptyset \wedge x \in X_\mathfrak{a}.$$

Items $x \in X_\mathfrak{n}$ that are present in the set $inact(\mathfrak{n})$ are called inactive, otherwise active. Accordingly, nodes that have inactive items are called inactive, otherwise active.

Inactive nodes denote transactions that would use the respective pattern for their encoding if a subset of the pattern was not already encoded by a preceding code. These nodes summarize the information about contained singletons, but their presence in the tree may enlarge the complexity due to the reflection of redundant information. Inactive nodes are integrated into the tree because they define the consequences of a change in the pattern selection and enable accordingly an efficient computation of the impacts that the removal or insertion of patterns have on the encoding.

To illustrate what is said so far, an example database and its depiction as a tree is displayed in Figure 3.3. The database on the left is encoded with the code table given in Figure 3.1, containing the patterns $\{b, d, e\} \preceq \{a, c\} \preceq \{a, g\}$ in standard encoding order besides of singleton itemsets. The resulting encoding of transactions is displayed at the right column of the table and the corresponding tree representation can be seen in the picture. The transactions are characterized by the branches up to a leaf, that are

TID	transaction	encoding set
1	a, b, d, e, f, g	$\{b, d, e\}\{a, g\}\{f\}$
2	a, b, c, d, e, g	$\{b, d, e\}\{a, c\}\{g\}$
3	a, c, e, f	$\{a, c\}\{e\}\{f\}$
4	a, b, c, d, e, f	$\{b, d, e\}\{a, c\}\{f\}$
5	a, c, e, g, f	$\{a, c\}\{e\}\{g\}\{f\}$
6	a, b, d, e, g	$\{b, d, e\}\{a, g\}$

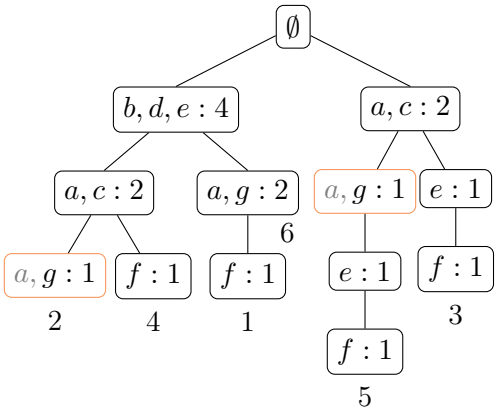


Figure 3.3: The encoding of a database (left table) and the induced tree representation of the database (right).

annotated with the identifier of the equivalent transaction. Inactive items are greyed out, which occurs only if one of the items is also present in a preceding active node of the corresponding branch.

We utilize this structure now to compute the usage function concerning an exemplary regarded candidate pattern as presented in Figure 3.4. The considered candidate is given by the pattern $\{c, e, f\}$ and it holds that $\{b, d, e\} \preceq \{c, e, f\} \preceq \{a, c\} \preceq \{a, g\}$. The computation of emerging usage differences starts with an identification of branches that would use the designated pattern for their encoding. The search begins at the minimum child of the root node, i.e. the node with the pattern $\{b, d, e\}$. Since the items b and e of the candidate pattern would be encoded by this node furthermore, the candidate pattern is not used in this sub-tree. The search proceeds with the right sub-tree, finding that transactions 5 and 3 would make use of the candidate and the corresponding leaves containing the singleton $\{f\}$ are stored. For these branches, the effects on the encoding resulting from the insertion of the candidate are calculated. It means in this case, that the pattern $\{a, c\}$ is not used there anymore, but the node with the pattern $\{a, g\}$ becomes active again. If the resulting usage function yields a better compression size, the pattern is integrated into the tree. Therefore, the branches that contain the candidate pattern but do not use it for the encoding, have to be identified. These are the branches that contain the nodes colored in light peach, that determine the insertion of inactive candidate nodes. By contrast, the nodes that indicate the insertion of an active candidate node have a darker peach color. The consequent tree is displayed in Figure 3.4 on the right.

The Code Table Tree

The structure of the database tree induces multiple ways in that the usage of a candidate pattern can be determined. Smets et al. state in [41] that there is no monotonicity or structure that can be used to find the optimal code table efficiently. Although that is true

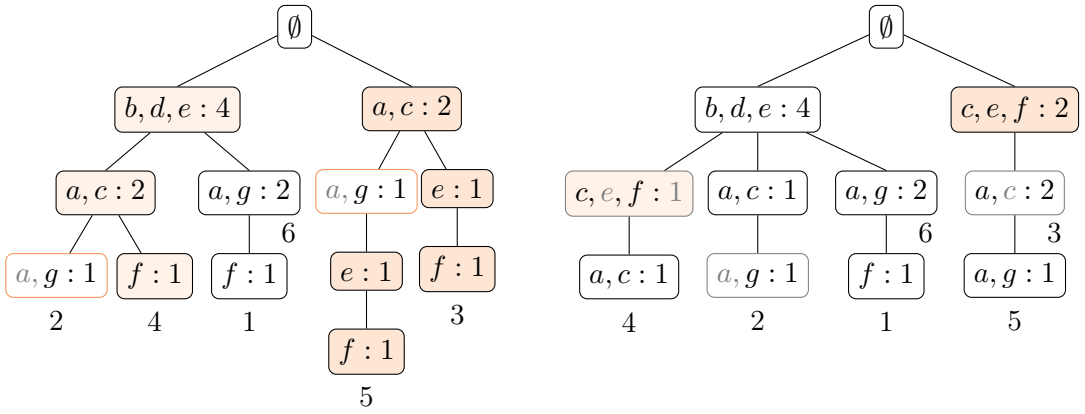


Figure 3.4: The database tree of Figure 3.3 (left) and the resulting tree when the pattern $\{c, e, f\}$ is inserted (right).

CT	freq
$\{b, d, e\}$	4
$\{c, e, f\}$	3
$\{a, c\}$	4
$\{a, g\}$	4
$\{f\}$	4
$\{g\}$	4

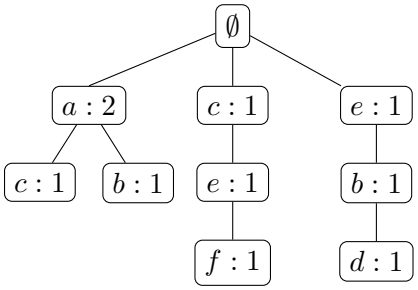


Figure 3.5: Representation of a code table (left) as a tree (right). Nodes are ordered according to the frequency of items.

with respect to isolated code tables, the structure of the encoding can at least be used to estimate the impact of changes to the present code table as it is done for instance in Slim. With regard to the tree structure, the affected patterns in the code table identify also the affected nodes in the database tree. Therefore, I propose to receive the patterns in the code table that are proper subsets of a candidate pattern, first. Every transaction that would use a designated pattern for their encoding must also contain all subset patterns. If the subset patterns are available and sorted in the standard encoding order, a top-down search in the database tree might be conducted more efficiently. The subsets serve as some kind of checkpoint nodes, whose absence indicates that the candidate is not contained in the corresponding branch.

Let's have a look at the representation of our example code table as a tree, ordered in standard encoding order, i.e., on frequency first and then lexicographically for singleton itemsets. The code table and the resulting tree are displayed in Figure 3.5. We assume that the next candidate pattern $\{a, c, e, f\}$ is given. A top-down search in the code table

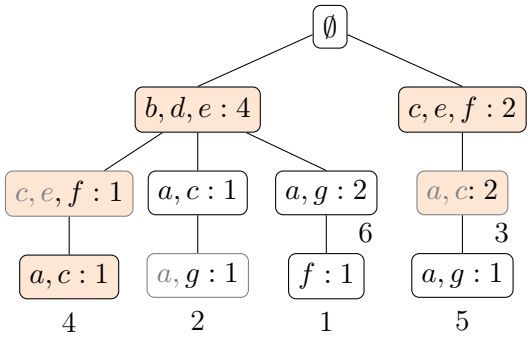


Figure 3.6: The database tree of Figure 3.4, nodes that indicate the part of the database where the candidate pattern $\{a, c, e, f\}$ would be applied are colored in green.

tree yields that the patterns $\{a, c\}$ and $\{c, e, f\}$ are the only code table entries that are also a subset of the candidate. That is, every branch in the database tree that indicates affected transactions must contain the nodes $\{c, e, f\}$ and $\{a, c\}$ in that order. The search in the database tree to identify those branches is then proceeded as follows and depicted in Figure 3.6.

Again, the minimum child of the root with the pattern $\{b, d, e\}$ is regarded first. This node reveals yet no indications to the absence or presence of the candidate in this branch, so the search is proceeded at its children. The first child node contains the checkpoint pattern $\{c, e, f\}$. Thus, the localization of the candidate in the branch with root $\{b, d, e\}$ is restricted to the transactions represented by descendant nodes of this checkpoint node. The only child of this node contains the other checkpoint pattern, which completes the search of the candidate in the left subtree. The node regarded next is the other child of the root node, which contains the first checkpoint pattern and that has again the second checkpoint pattern as its child. This finishes the identification of concerned branches, no matter how many succeeding children of the root node or of the checkpoint nodes exist. Therefore, the interesting parts of the tree are likely to be determined much faster this way.

We note that the code table might also be preserved by any other data structure that enables a fast subset identification. However, similar to the database tree, the standard encoding order induces an elegant way to determine the interesting patterns in a tree efficiently. We regard now the algorithm in a more formal way.

3.3.2 Identifying the affected Parts of the Database

The procedure of SHrimp (Alg. 12) is similar to that of Krimp. In fact, it returns the same code tables and uses all heuristic assumptions that are also made in Krimp. The computation of central functions like usage or compression size is however different and these parts may be adapted to other algorithms like Slim. We describe now exemplary

the procedure to examine candidates and to calculate the usage function in the KRIMP framework.

The input of SHRIMP is the database $\mathcal{D}$ and a set of frequent patterns $\mathcal{F}$. The frequent patterns are sorted in standard candidate order (line 2) and the database respectively code table tree is initialized (lines 3, 4). Since the code table is initialized by the standard

Algorithm 12 SHrimp

```
1:  procedure SHRIMP(D, F)
2:      F ← sort(F)                                  ▷ in Standard Candidate Order
3:      D ← initTree(D)
4:      CT ← initTree(∅)
5:      U ← {(x, freq(x)) | x ∈ I}
6:      for X̂ ∈ F \ I do
7:          checkpts ← {X ∈ CT | X ⊂ X̂}
8:          𝔑 ← AffectedParts(X̂, root(D), ∅, checkpts)
9:          ΔU ← DeltaUsage(X̂, 𝔑, D)
10:         if L(U ⊕ ΔU) < L(U) then
11:             U ← U ⊕ ΔU
12:             (D, CT, U) ← Pruning(D, CT, U, ΔU)
13:         end if
14:     end for
15:     return CT, D
16: end procedure
```

code table, containing only singletons, the initial database tree is equal to the FP-tree. Accordingly, the usage function is set as the function that maps to each singleton itemset its frequency (line 5). Candidates are regarded in standard candidate order. For each of the candidates, the transactions that would use the candidate for their encoding are computed and stored representative by the corresponding nodes in the set $\mathfrak{N}$ (line 8). The differences in the encoding are computed (line 9) and the compression size as defined by the resulting usage function $U \oplus \Delta U$ is calculated. The symbol $\oplus$ denotes hereby a sum of the assigned usages for each pattern, i.e., $(U \oplus \Delta U)(X) = U(X) + \Delta U(X)$. If the inclusion of the pattern improves the compression, the pattern is integrated into the tree (line 12). The method PRUNING can easily be adapted from its original formulation in Alg. 10, using the mechanisms to identify changes in the usage function and comparing the compression size afterwards, as presented here. In the end, the algorithm returns the code table and the database tree (line 15). This way, the set of interesting patterns as well as their actual composition in the database can be inspected in a compact representation.

Identifying the Differences of the Encoding

The computation of nodes whose descendant leaves describe the transactions that are affected by an insertion of the candidate pattern is displayed in Alg. 13. It is a recursive function, that calls itself for selected children nodes of the specified node $\mathfrak{n}$. The set $\widehat{X}$ maintains the items of the candidate pattern that are not contained in any ancestor node of $\mathfrak{n}$. The method is invoked for the first time by Alg. 12 having the candidate $\widehat{X}$, the root node of the database tree, an empty set of indicating nodes $\mathfrak{N}$ and the checkpoint patterns *checkpts* from the code table tree as arguments. As discussed in Section 3.3.1,

Algorithm 13 Computing the affected Transactions for a Candidate

```
1:  procedure AffectedParts(X̂, n, N, checkpts)
2:      for c ∈ {c ∈ children(n) | c ⪯ min{checkpts}} do
3:          if X_c ∩ X̂ ≠ ∅ ∧ X_c ≺ X̂ then
4:              continue
5:          end if
6:          X̂ ← X̂ \ X_c
7:          if X̂ = ∅ then
8:              N ← N ∪ c
9:          else
10:             checkpts ← checkpts \ X_c
11:             N ← N ∪ AffectedParts(X̂, c, N, checkpts)
12:         end if
13:     end for
14:     return N
15: end procedure
```

only those children that are preceding or equal to the next checkpoint pattern indicate the branches that might contain the candidate pattern and are therefore regarded (line 2). If some of the candidate items are encoded by a preceding pattern, the branch is not further investigated (lines 3-5). Otherwise, the items of the child node are removed from the set of not yet spotted items of the candidate (line 6). If this set is empty, the branch of the current child node would contain an active node with the candidate pattern and the child node is added to the set $\mathfrak{N}$ (lines 7,8). If this is not the case, the branch has to be traversed further and the function is therewith called for the child node again (lines 9-12). In the end, the obtained set of nodes $\mathfrak{N}$ is returned.

The method DeltaUsage (Alg. 14) identifies the changes in the encoding and therewith also in the usage that accompany the insertion of a candidate. For each of the nodes $\mathfrak{m}$ that are descendant leaves of a change indicating node $\mathfrak{n} \in \mathfrak{N}$, the respective changes of encodings from transactions denoted by that leaf are determined by the function Prec-

Algorithm 14 Computing the Usage Differences for a Candidate

1: **procedure** DELTAUSAGE($\widehat{X}, \mathfrak{N}, D$)
2: **for** $\mathfrak{m} \in \{\mathfrak{m} | \exists \mathfrak{n} \in \mathfrak{N} : \mathfrak{m} \in \text{desc}(\mathfrak{n}), \mathfrak{m} \text{ is a leaf}\}$ **do**
3: $(\Delta U, bound, free) \leftarrow$ PRECCHANGES($\mathfrak{m}, \emptyset, \{\widehat{X}\}, \emptyset, usage(m)$)
4: **end for**
5: **return** ΔU
6: **end procedure**
7: **procedure** PRECCHANGES($\mathfrak{m}, \Delta U, bound, free, u$)
8: **if** $\widehat{X} \prec X_{\mathfrak{m}}$ **then**
9: $(\Delta U, bound, free) \leftarrow$ PRECCHANGES($parent(\mathfrak{m}), bound, free$)
10: **end if**
11: **if** $inact(\mathfrak{m}) = \emptyset \wedge (bound \cap X_{\mathfrak{m}} \neq \emptyset)$ **then**
12: $\Delta U(X_{\mathfrak{m}}) \leftarrow \Delta U(X_{\mathfrak{m}}) - u$
13: $free \leftarrow free \cup X_{\mathfrak{m}}$
14: **else if** $(\emptyset \neq inact(\mathfrak{m}) \subseteq free) \wedge (bound \cap X_{\mathfrak{m}} = \emptyset)$ **then**
15: $\Delta U(X_{\mathfrak{m}}) \leftarrow \Delta U(X_{\mathfrak{m}}) + u$
16: $bound \leftarrow bound \cup X_{\mathfrak{m}}$
17: **end if**
18: **return** $(\Delta U, bound, free)$
19: **end procedure**

CHANGES (lines 7-19). This procedure calculates the occurring changes for the ancestor nodes, by a recursive call for parent nodes up to the first node that is preceding to the candidate pattern (line 9). Changes may only occur in nodes that are succeeding to the candidate. The maintained set *bound* stores items that are encoded by ancestor nodes of the specified node $\mathfrak{m}$, while $free$ contains those items that are in contrast to the current encoding up to this point not encoded anymore. By means of these sets, it is checked whether $\mathfrak{m}$ would change its status of activity, either from active to inactive (line 11) or the other way round (line 14). Such status changes indicate usage changes for every transaction that is represented by the leaf for that the method has been invoked first. The usage changes therewith by the number of affected transactions u (lines 12,15).

3.4 Matrix Factorization for Compression

In this section, we conclude the discussion about the relations of the MDL-based approach to identify the set of patterns that describes the database in the most compressed way, to clustering methods. We derive a formulation of the compression size in terms of matrix factorization, that introduces the possibility to gain the best compressing patterns without making use of any heuristic. Furthermore, the number of obtained patterns in the code table can be specified.

While the approaches discussed so far are related to the Minimum Tiling problem, the task of finding the k patterns that describe the database best is more similar to a Maximum k-Tiling. The adjustment of the parameter that denotes the number of obtained patterns makes it also possible to obtain different views on the same database. The description of a database with a high number of compressing patterns might be more fine grained than one using only few patterns. Depending on the insights one wants to obtain, a rough overview can be derived as well as a detailed description of the database.

In the following two subsections, I propose two methods to solve relaxed versions of the task to obtain the best compressing code table by a matrix factorization. As a standard technique that is applied to NMF and BMF problems, gradient descent is applied here as well. The values of the matrices are then rounded by a suitable threshold, to derive the actual encoding. We begin with a description of the representation of an encoding and develop the algorithm Pimp. We discuss then a different method to model the encoding by matrix factorization with the algorithm Mimikri.

3.4.1 Pimp

We recall from Section 2.3 that a matrix decomposition into two factor matrices yields a description by basis vectors and their coefficients. The coefficients define the linear combinations of basis vectors that compose the observations, respectively transactions. Accordingly, we can interpret basis vectors as dictionary entries, or codewords, that put the transactions as indicated by the coefficient matrix together. In other terms, a binary matrix can be factorized into a matrix that represents a code table and a matrix that denotes the usage of the code words.

Let $D \in \{0,1\}^{n \times m}$ denote the binary matrix representation of the given database and let $X \in \{0,1\}^{n \times r}$ and $U \in \{0,1\}^{r \times m}$ be factor matrices for a given rank $r \in \mathbb{N}$, i.e., $D \approx XU$. Simplifying the notation, we assume for the following descriptions that the factorization is exact, i.e., $D = XU$. To understand the synergy between the code table matrix X and the matrix of usages U more detailed, we take a look at the decomposition of the j-th transaction $D_{\cdot j}$ into X and U

$$D_{\cdot j} = X \cdot U_{\cdot j} = \sum_{s=1}^{r} U_{sj} X_{\cdot s}.$$

We can see how the encoding of the j-th transaction is described by the patterns $X_{\cdot s}$ and the usage entries U_{sj} for $1 \leq s \leq r$. A pattern $X_{\cdot s}$ is used for the encoding of $D_{\cdot j}$ iff $U_{sj} = 1$, whereas the entries $X_{is} = 1$ indicate the items with index $1 \leq i \leq n$ that are present in the pattern $X_{\cdot s}$. That way, the usage of a pattern $X_{\cdot s}$ can be computed by the sum of all entries in $U_{s\cdot}$ that are equal to one

$$usage(X_{\cdot s}) = \sum_{j=1}^{m} U_{sj} = |U_{s\cdot}|. \tag{3.7}$$

The notation as a sum of absolute values $|U_{s\cdot}|$ in Eq. (3.7) is valid as long it may be assumed that the entries of U are nonnegative. The compression size of the database (2.2) can therewith be calculated depending solely on the usage matrix U by the function

$$\mathcal{L}_D(U) = -\sum_{s=1}^{r} |U_{s\cdot}| \log\left(\frac{|U_{s\cdot}|}{|U|}\right).$$

Equivalently, we can state the compression size of the model (2.3) by means of the vector of standard code lengths $v = -(\log(\text{supp}(x_1)), \ldots, \log(\text{supp}(x_n)))$ by the function

$$\begin{aligned}\mathcal{L}_M(X,U) &= -\sum_{s=1}^{r} \mathrm{I}_{|U_{s\cdot}|>0} \left(\log\left(\frac{|U_{s\cdot}|}{|U|}\right) + \sum_{i=1}^{n} X_{is} \log\left(\text{supp}(x_i)\right)\right) \\ &= -\sum_{s=1}^{r} \mathrm{I}_{|U_{s\cdot}|>0} \left(\log\left(\frac{|U_{s\cdot}|}{|U|}\right) - X_{\cdot s}^T v\right).\end{aligned}$$

The indicator function $\mathcal{I}_{|U_{s\cdot}|>0}$ models the fact that only used patterns are stated in the code table and contribute to the model description size.

With these observations, we can describe the originally declared task of finding the code table that compresses the database best as

Problem 1.

$$\begin{aligned}\min_{X,U}\ & \mathcal{L}(X,U) = \mathcal{L}_D(U) + \mathcal{L}_M(X,U) \\ & s.t.\quad \frac{1}{2}\|D - XU\|^2 = 0, \\ & \qquad X \in \{0,1\}^{n\times r}, \\ & \qquad U \in \{0,1\}^{r\times m}.\end{aligned} \tag{3.8}$$

We note that algorithms that are designed to solve Problem 1 are not restricted to the standard encoding order, as any algorithm is known so far. Solving the problem is however not trivial.

Similar to the Penalizing BMF algorithm [48] which is summarized in Section 2.3.2 we use penalizing terms to derive an unconstrained formulation of the objective (3.8). Also, we use the function ω to penalize non-binary values. The objective is given as

$$\min_{X,U} F_\lambda(X,U) = \frac{1}{2}\|D - XU\|^2 + \mu\mathcal{L}(X,U) + \lambda\sum_{i,s}\omega(X_{is}) + \lambda\sum_{s,j}\omega(Y_{sj}), \tag{3.9}$$

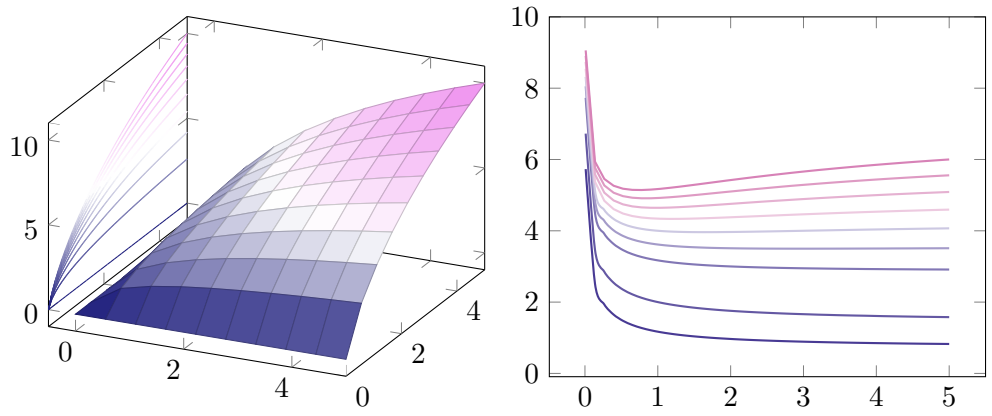


Figure 3.7: Plots of the functions $f(x,y) = -x\log\left(\frac{x}{x+y}\right) - y\log\left(\frac{y}{x+y}\right)$ on the left side, and $g(x;y) = -(x+1)\log\left(\frac{x}{x+y}\right)$ for increasing values of $y \in [1,5]$ on the right.

for parameters $\mu, \lambda \in \mathbb{R}_+$. We have now a closer look on this function and discuss its properties.

The function $F_\lambda(X,U)$ is nonconvex, in particular it is nonconvex in X and U and is therefore likely to contain multiple minimums or stationary points in general. The function $\mathcal{L}_D(U)$ is concave and has a single minimum at the origin. Its curve progression is depicted exemplary for a rank of two on the left in Figure 3.7. The values on the x- and y-axis denote the values of $|U_{\cdot 1}|$ and $|U_{\cdot 2}|$. We can see that the function values are rather small if one of the patterns is not used at all, i.e., $|U_{\cdot i}| = 0$ for $i \in \{1,2\}$. Moreover, the larger the usage of one pattern is, the stronger increases the function strictly monotonically in the usage of the other pattern.

The function that takes also the description size of the code table into account is sketched on the right of Figure 3.7. The values on the x-axis denote the usage of a pattern, assuming that the usage of other patterns is fixed. The depicted curves belong to different statically determined usage sums of other patterns that are from 0.5 to 5 assignable from bottom to top. It is particularly noticeable that the compression size tends to infinity if the usage of one pattern approaches zero. Although this is negligible in the discrete setting of the encoding, the continuous representation of the description size has a point of discontinuity at the origin.

Therefore, the function $\mathcal{L}(X,U)$ is not differentiable at matrices X and U where one pattern is not used at all, i.e., if

$$U \in \mathcal{S} := \{V \in \mathbb{R}_+^{r\times m} | \exists s \in \mathbb{N}, s \leq r : |V_{s\cdot}| = 0\}.$$

Since the negative gradient is not defined at those points, a different search direction has to be used. A common method is to use a negative subgradient instead [35]. The negative subgradient is yet not necessarily a descent direction, since Taylor's theorem can not be

applied. Therefore, a line search algorithm is not guaranteed to find a suitable stepsize and the sequence of obtained iterates $F_\lambda(X_k, U_k)$ might increase at certain iterations.

Computing the Gradient We derive now the gradient of the function F_λ with respect to X and U at those matrices where it exists, i.e., $U \notin \mathcal{S}$. F_λ is composed by a sum of the functions F (2.4), $\mathcal{L}$ describing the compression size and ω. The gradients of F_λ are thus calculated by

$$\nabla_X F_\lambda(X,Y) = \nabla_X F(X,Y) + \left(\frac{\partial}{\partial X_{is}} F_\lambda(X,Y)\right)_{is} + \left(\frac{d}{dX_{is}}\omega(X_{is})\right)_{is} \tag{3.10}$$

$$\nabla_Y F_\lambda(X,Y) = \nabla_Y F(X,Y) + \left(\frac{\partial}{\partial Y_{sj}} F_\lambda(X,Y)\right)_{sj} + \left(\frac{d}{dY_{sj}}\omega(Y_{sj})\right)_{sj}. \tag{3.11}$$

The respective derivatives of F and ω are already stated in Section 2.3. It remains to deduce the gradient of the compression size denoting function. The partial derivative of $\mathcal{L}(X,U)$ with respect to $U_{\hat{s}\hat{j}} \geq 0$ is given as follows:

$$\frac{\partial \mathcal{L}(X,U)}{\partial U_{\hat{s}\hat{j}}} = -\sum_{s=1}^{r}\left[\frac{\partial}{\partial U_{\hat{s}\hat{j}}}\left(|U_{s\cdot}|+1\right)\right]\log\left(\frac{|U_{s\cdot}|}{|U|}\right) + \left(|U_{s\cdot}|+1\right)\left[\frac{\partial}{\partial U_{\hat{s}\hat{j}}}\log\left(\frac{|U_{s\cdot}|}{|U|}\right)\right] \tag{3.12}$$

$$= -\log\left(\frac{|U_{\hat{s}\cdot}|}{|U|}\right) - \sum_{s=1}^{r}\left(|U_{s\cdot}|+1\right)\left[\frac{\partial}{\partial U_{\hat{s}\hat{j}}}\left(\log|U_{s\cdot}| - \log|U|\right)\right] \tag{3.13}$$

$$= -\log\left(\frac{|U_{\hat{s}\cdot}|}{|U|}\right) - \frac{|U_{\hat{s}\cdot}|+1}{|U_{\hat{s}\cdot}|} + \sum_{s=1}^{r}\frac{|U_{s\cdot}|+1}{|U|} \tag{3.14}$$

$$= -\log\left(\frac{|U_{\hat{s}\cdot}|}{|U|}\right) - 1 - \frac{1}{|U_{\hat{s}\cdot}|} + 1 + \frac{r}{|U|} \tag{3.15}$$

$$= -\log\left(\frac{|U_{\hat{s}\cdot}|}{|U|}\right) - \frac{1}{|U_{\hat{s}\cdot}|} + \frac{r}{|U|} \tag{3.16}$$

We apply the product rule in Eq. (3.12) and use in Eq. (3.13) the properties of the logarithm and the fact that

$$\frac{\partial}{\partial U_{\hat{s}\hat{j}}}|U_{s\cdot}| = \frac{\partial}{\partial U_{\hat{s}\hat{j}}}\sum_{j=1}^{m} U_{sj} = \begin{cases} 1 & \text{if } s = \hat{s} \\ 0 & \text{otherwise.} \end{cases}$$

The logarithmic terms are derived in Eq. (3.14) and in Eq. (3.15) we apply that $\sum_{s=1}^{r}|U_{s\cdot}| = |U|$. Therewith, we obtain the final derivative (3.16). If $|U_{\hat{s}\cdot}| = 0$, we choose the partial subderivative zero.

By contrast, the partial derivative of $\mathcal{L}(X,U)$ with respect to $X_{\hat{\imath}\hat{s}} \geq 0$ can be calculated straightforwardly as

$$\frac{\partial \mathcal{L}(X,U)}{\partial X_{\hat{\imath}\hat{s}}} = \frac{\partial}{\partial X_{\hat{\imath}\hat{s}}}\sum_{s=1}^{r} X_{\cdot s}^T v$$

$$= v_{\hat{\imath}}.$$

We apply projected gradient descent, as it is stated in Alg. 5 to obtain the approximate result. We call the resulting algorithm PIMP as it uses penalizing terms to solve the problem that is tackled by KRIMP. For the sake of completeness, we sketch the procedure in Alg. 15. It uses the method THRESHOLD to round the matrices suitably and that yields the finally derived encoding of the database. This method is discussed in Section 3.4.3.

Algorithm 15 Pimp

procedure PIMP($D, r, K, T, \mathcal{A}$)
  Initialize $X_0 \in \mathbb{R}_+^{n\times r}$ and $Y_0 \in \mathbb{R}_+^{r\times m}$ randomly
  for $k \in \{1, \dots, K\}$ **do**
    $\alpha_k = \text{BACKTRACKINGLS}(\mathcal{L}, (X_k, Y_k), T)$
    $X_{k+1} \leftarrow P_+\big(X_k - \alpha_k \nabla_X \mathcal{L}(X_k, Y_k)\big)$
    $\alpha_k = \text{BACKTRACKINGLS}(\mathcal{L}, (X_{k+1}, Y_k), T)$
    $Y_{k+1} \leftarrow P_+\big(X_{k+1} - \alpha_k \nabla_Y \mathcal{L}(X_{k+1}, Y_k)\big)$
  end for
  $a \leftarrow \text{THRESHOLD}(X_K, Y_K\mathcal{A}, true)$
  return $(\Theta(X_K; a), \Theta(Y_K; a))$
end procedure

3.4.2 Mimikri

Siebes et al. state in [45] that as optimal compression is the goal, it makes intuitive sense that overlapping elements may lead to shorter encodings, as then fewer itemsets may be required to describe the data. However, it is not immediately clear how to achieve this in a fast heuristic, which is why they do not allow overlap.

In this section, we pursue a different approach to model the encoding by a matrix factorization, allowing codes to overlap. For this purpose, we utilize the function Φ introduced in Section 2.3.2 that approximates the heavyside step function. Here, we overload the notation of the function Φ and set the threshold parameter to one half, i.e., $\phi(x) = \phi(x, 0.5)$ and also $\Phi(X) = \Phi(X, 0.5)$. Applying Φ to the matrices X and U simulates the factorization by binary matrices. The overlap of codes can then be facilitated by an additional application of the function Φ to the product of the factor matrices. This maps the values of the factorization that are greater than one, i.e., where multiple codes describe the same item, near to one. The function that models the quality of the factorization is thereby for parameters $\eta_1, \eta_2 > 0$ given as

$$F_\phi(X, U) = \|D - \Phi_{\eta_1}\big(\Phi_{\eta_2}(X)\Phi_{\eta_2}(U)\big)\|^2$$
$$F_{\phi i}(X, U) = \|D_{i\cdot} - \Phi_{\eta_1}\big(\Phi_{\eta_2}(X_{i\cdot})\Phi_{\eta_2}(U)\big)\|^2.$$

We state the function that describes the quality of the factorization with respect to a single row of the data matrix explicitly as the function $F_{\phi i}$. We need this formulation later. The function F_ϕ approximates the amount of incorrectly approximated entries. We assume that items that are not covered by the factorization are appended to the encoding by their corresponding singleton codes. Similarly, items that are present in the factorization but do not occur in the original transactions can be removed to make the database description exact. We imagine a second column of the encoding that contains the codes of items that have to be removed. Assuming that the patterns described by the matrix X contain more than one item, the usage of singletons is approximated by the function F_ϕ. More precisely, the usage of a singleton $\{x_i\}$ is approximated by the function value $F_{\phi i}(X, U)$. For a shorter notation, we denote the function Φ_{η_2} as the function Φ. By this means, we can model the compression size of the database directly by the functions

$$\mathcal{L}_U(X,U) = -\sum_{r=1}^{s} |\Phi(U_{r\cdot})| \log\left(\frac{|\Phi(U_{r\cdot})|}{|\Phi(U)| + F_\phi(X,U)}\right)$$
$$\mathcal{L}_X(X,U) = -\sum_{i=1}^{n} F_{\phi i}(X,U) \log\left(\frac{F_{\phi i}(X,U)}{|\Phi(U)| + F_\phi(X,U)}\right).$$

The function $\mathcal{L}_U(X,U)$ models the description size of the database as it is given by the usage matrix and the function $\mathcal{L}_X(X,U)$ accounts the addition or removal of singletons to represent the database exactly. The compression size of the database can thus be calculated by the function

$$\begin{aligned}\mathcal{L}_\phi(X,U) &= \mathcal{L}_U(X,U) + \mathcal{L}_X(X,U) \\ &= -\sum_{r=1}^{s} |\Phi(U_{r\cdot})| \log |\Phi(U_{r\cdot})| - \sum_{i=1}^{n} F_{\phi i}(X,U) \log F_{\phi i}(X,U) \\ &\quad + \big(|\Phi(U)| + F_\phi(X,U)\big) \log \big(|\Phi(U)| + F_\phi(X,U)\big).\end{aligned} \tag{3.17}$$

This function is smooth for real valued matrices X and U. So, we can apply a simple gradient descent procedure to minimize (3.17). The description size of the model is neglected in this case, since this would introduce points of discontinuity. Furthermore, the reason why the description size of the model is taken into account originally is to provide some mechanism against overfitting. In our formulation, the specification of the rank r bounds the complexity of the model and it is therefore reasonable to ignore the part that describes the model description size.

Calculating the Gradient To apply the gradient descent algorithm, we have to infer the derivative of the objective. The partial derivatives of F_ϕ are given as

$$\frac{\partial F_\phi(X,U)}{\partial U_{\hat{s}\hat{j}}} = -2\phi'(U_{\hat{s}\hat{j}}) \sum_{i=1}^{n} \left(D_{i\hat{j}} - \phi(\Phi(X_{i\cdot})\Phi(U_{\cdot\hat{j}}))\right) \phi'(\Phi(X_{i\cdot})\Phi(U_{\cdot\hat{j}}))\phi(X_{i\hat{s}}) \tag{3.18}$$

$$\frac{\partial F_\phi(X,U)}{\partial X_{\hat{i}\hat{s}}} = -2\phi'(X_{\hat{i}\hat{s}}) \sum_{j=1}^{m} \left(D_{\hat{i}j} - \phi(\Phi(X_{\hat{i}\cdot})\Phi(U_{\cdot j}))\right) \phi'(\Phi(X_{\hat{i}\cdot})\Phi(U_{\cdot j}))\phi(U_{\hat{s}j}), \tag{3.19}$$

which can be derived by an elementary application of the chain rule. The function ϕ' denotes thereby the derivative of the function ϕ in Lagrange's notation. Less straightforward is the calculation of the gradient of the objective (3.17). We observe first, that the function F_ϕ is decomposable into a sum of respective values of $F_{\phi i}$

$$\begin{aligned}\sum_{i=1}^{n} F_{\phi i}(X,U) &= \sum_{i=1}^{n} \|D_{i\cdot} - \Phi(\Phi(X_{i\cdot})\Phi(U))\|^2 \\ &= \sum_{i=1}^{n}\sum_{j=1}^{m} (D_{ij} - \phi(\Phi(X_{i\cdot})\Phi(U_{\cdot j})))^2 \\ &= \|D - \Phi(\Phi(X)\Phi(U))\|^2. \end{aligned} \tag{3.20}$$

The partial derivative with respect to $U_{\hat{s}\hat{j}}$ can be calculated by

$$\begin{aligned}\frac{\partial \mathcal{L}(X,U)}{\partial U_{\hat{s}\hat{j}}} &= -\phi'(U_{\hat{s}\hat{j}}) \log|\Phi(U_{\hat{s}\cdot})| - |\Phi(U_{\hat{s}\cdot})| \frac{\phi'(U_{\hat{s}\hat{j}})}{|\Phi(U_{\hat{s}\cdot})|} \\ &\quad - \sum_{i=1}^{n} \left(\left[\frac{\partial}{\partial U_{\hat{s}\hat{j}}} F_{\phi i}(X,U)\right] \log F_{\phi i}(X,U) + F_{\phi i}(X,U) \frac{\left[\frac{\partial}{\partial U_{\hat{s}\hat{j}}} F_{\phi i}(X,U)\right]}{F_{\phi i}(X,U)} \right) \\ &\quad + \left(\phi'(U_{\hat{s}\hat{j}}) + \left[\frac{\partial}{\partial U_{\hat{s}\hat{j}}} F_\phi(X,U)\right]\right) \log\left(|\Phi(U)| + F_\phi(X,U)\right) \\ &\quad + \left(|\Phi(U)| + F_\phi(X,U)\right) \frac{\phi'(U_{\hat{s}\hat{j}}) + \left[\frac{\partial}{\partial U_{\hat{s}\hat{j}}} F_\phi(X,U)\right]}{|\Phi(U)| + F_\phi(X,U)} \end{aligned} \tag{3.21}$$

$$\begin{aligned} &= -\phi'(U_{\hat{s}\hat{j}}) \log \frac{|\Phi(U_{\hat{s}\cdot})|}{|\Phi(U)| + F_\phi(X,U)} \\ &\quad - \sum_{i=1}^{n} \left[\frac{\partial}{\partial U_{\hat{s}\hat{j}}} F_{\phi i}(X,U)\right] \log \frac{F_{\phi i}(X,U)}{|\Phi(U)| + F_\phi(X,U)} \end{aligned} . \tag{3.22}$$

The first equation (3.21) describes thereby the application of the product rule, while Eq. (3.22) summarizes the terms suitably. The rightmost terms of the first two lines in Eq. (3.21) are canceled out by the term on the last line. To describe the other terms in a compact way, the properties of the logarithm are applied. We use thereby the decomposability of the function F_ϕ (3.20) to combine the expressions that contain the derivative of F_ϕ.

The partial derivative with respect to X can be calculated accordingly

$$\begin{aligned}\frac{\partial \mathcal{L}(X,U)}{\partial X_{\hat{\imath}\hat{s}}} &= -\left[\frac{\partial}{\partial X_{\hat{\imath}\hat{s}}} F_{\phi\hat{\imath}}(X,U)\right] \log F_{\phi\hat{\imath}}(X,U) - F_{\phi\hat{\imath}}(X,U) \frac{\left[\frac{\partial}{\partial X_{\hat{\imath}\hat{s}}} F_{\phi\hat{\imath}}(X,U)\right]}{F_{\phi\hat{\imath}}(X,U)} \\ &\quad + \left[\frac{\partial}{\partial U_{\hat{s}\hat{\jmath}}} F_{\phi}(X,U)\right] \log\left(|\Phi(U)| + F_{\phi}(X,U)\right) \\ &\quad + \left(|\Phi(U)| + F_{\phi}(X,U)\right) \frac{\left[\frac{\partial}{\partial X_{\hat{\imath}\hat{s}}} F_{\phi\hat{\imath}}(X,U)\right]}{|\Phi(U)| + F_{\phi}(X,U)} \\ &= -\left[\frac{\partial}{\partial X_{\hat{\imath}\hat{s}}} F_{\phi\hat{\imath}}(X,U)\right] \log \frac{F_{\phi\hat{\imath}}(X,U)}{|\Phi(U)| + F_{\phi}(X,U)}.\end{aligned}$$

The first equation describes again the application of the product rule and the second one summarizes the result.

We summarize the proceeded steps in Alg. 16. The name MIMIKRI is chosen as the objective mimics the factorization of binary matrices by the function ϕ and also all letters of the word *Krimp* are covered, allowing overlappping, except for the letter p which is to be added by its singleton code in this view. How the final encoding is computed by the

Algorithm 16 Mimikri

```
1: procedure MIMIKRI(D, r, K, T, A)
2:     Initialize X_0 ∈ R_+^{n×r} and Y_0 ∈ R_+^{r×m} randomly
3:     for k ∈ {1, ..., K} do
4:         α_k = BACKTRACKINGLS(F_φ, (X_k, Y_k), T)
5:         X_{k+1} ← X_k − α_k ∇_X F(X_k, Y_k)
6:         α_k = BACKTRACKINGLS(F, (X_{k+1}, Y_k), T)
7:         Y_{k+1} ← X_{k+1} − α_k ∇_Y F(X_{k+1}, Y_k)
8:     end for
9:     a ← THRESHOLD(X_K, Y_k, A, false)
10:    return (Θ(X_K; a), Θ(Y_K; a), Θ(D − X_K Y_K; a))
11: end procedure
```

function THRESHOLD is described in the following section.

3.4.3 Thresholding the Matrices

The description of the database by a factorization is not lossless. If we round the matrix entries to binary values in the end, the computation of an actual encoding requires that imperfect representations of transactions are repaired. As described for the algorithm MIMIKRI, we can assume that incorrectly represented items can be added or removed by their singleton codes. The difference between the algorithms PIMP and MIMIKRI is that PIMP takes also the description size of the model into account. Depending on the function

that describes the compression size, we seek for a threshold at which the matrices can be rounded such that the encoding is minimal in size.

To obtain the threshold at which the entries of X and U can be rounded suitably, equidistant values between zero and one are tested and the one that yields the smallest compression size is applied. The compression size of the database is calculated by substituting the function ϕ of the objective F_ϕ used in MIMIKRI, with the heavyside function θ.

$$\begin{aligned}\mathcal{L}_\theta^D(X,U;a) = &-\sum_{r=1}^{s} |\Theta(U_{r\cdot};a)| \log\left(\frac{|\Theta(U_{r\cdot};a)|}{|\Theta(U;a)| + F_\theta(X,U;a)}\right) \\ &-\sum_{i=1}^{n} F_{\theta i}(X,U;a) \log\left(\frac{F_{\theta i}(X,U;a)}{|\Theta(U;a)| + F_\theta(X,U;a)}\right).\end{aligned}$$

The function that returns the description size of the model for a threshold a can adequately be stated as

$$\begin{aligned}\mathcal{L}_\theta^M(X,U;a) = &-\sum_{s=1}^{r} \mathrm{I}_{|\Theta(U_{s\cdot};a)|>0}\left(\log\left(\frac{\Theta(|U_{s\cdot}|;a)}{\Theta(|U|;a) + F_\theta(X,U;a)}\right) - \Theta(X_{\cdot s})^T v\right) \\ &-\sum_{i=1}^{n} \mathrm{I}_{F_{\theta i}(X,U;a)>0}\left(\log\left(\frac{F_{\theta i}(X,U;a)}{\Theta(|U|;a) + F_\theta(X,U;a)}\right) - e_i^T v\right).\end{aligned}$$

The vector e_i denotes thereby the i-th standard basis vector. The procedure that obtains the best threshold with respect to the description size is outlined in Alg. 17.

Algorithm 17 Obtain the threshold that minimizes the compression size

```
1:  procedure THRESHOLD(X_K, Y_K A, twoPart)
2:      a* ← 0
3:      L* ← ∞
4:      for a ∈ A do
5:          L ← L_θ^D(X, U; a)
6:          if twoPart then
7:              L ← L + L_θ^M(X, U; a)
8:          end if
9:          if L < L* then
10:             a* ← a
11:             L* ← L
12:         end if
13:     end for
14:     return a*
15: end procedure
```

3.4.4 Relations to Clustering

The formulation of the problem to derive the shortest possible encoding (3.9) as a constrained BMF completes our discussion about the relationship of KRIMP to clustering methods. Since a BMF yields a simultaneous clustering of items and transactions, inducing a small compression size on the encoding that is represented by a BMF has an effect on the cluster centroids. We recall that a binary matrix factorization $D \approx XU$ induces a clustering of the transactions with centroids $X_{\cdot s}$ and cluster assignments U_{sj}. Transposing the factorization $D^T \approx U^T X^T$ returns cluster centroids with respect to items $U_{s\cdot}$ and the assignments X_{is}.

To derive the connection of the objectives with regard to Problem 1 more explicitly, we reformulate the term that describes the compression size of the database slightly:

$$\begin{aligned}\mathcal{L}_D(U) &= -|U| \sum_{s=1}^{r} \frac{|U_{s\cdot}|}{|U|} \log\left(\frac{|U_{s\cdot}|}{|U|}\right) \\ &= |U| \cdot H\left(\frac{|U_{1\cdot}|}{|U|}, \ldots, \frac{|U_{r\cdot}|}{|U|}\right)\end{aligned}$$

H denotes thereby the entropy function. The minimization of the database description size $\mathcal{L}_D(U)$ minimizes also the overall number of cluster assignments $|U|$ as well as the entropy of the probabilities $\frac{U_{s\cdot}}{|U|}$ with that an observation is assigned to the a cluster. Thereby, assuming that all of the r clusters contain at least one data point, clusterings with few large clusters and multiple clusters that are as tiny as possible are preferably returned by KRIMP.

The view of the rows of U as feature clusters yields that the 1-norm of feature cluster centroids is comparably large for few centroids and small for most of the other ones. We further notice that KRIMP and SLIM return for the regarded databases code tables that contain at least twice as many code words than items in the database exist [41, 45]. Therewith, the returned code tables are massively overfitting the clustering on the features.

As such, a code table as it is returned by some of the heuristic algorithms discussed so far, yields a clustering of the data that does not require a specification of the number of expected clusters.

Chapter 4

Experiments

In this chapter, we discuss the performance and ability to compress the data by the aforementioned algorithms. The used datasets are displayed in Table 4.1, showing basic characteristics like size and density. The description size with respect to the standard code table is also depicted. All datasets except for the BMS-webview 1 originate from the UCI Machine Learning Repository[1] and have been prepared for itemset mining tasks. The prepared binary datasets are publicly available from the LUCS/KDD data set repository[2]. The BMS-webview 1 dataset can be downloaded from the FIMI repository[3]. It contains

D	n	m	$\lvert D\rvert\%$	$L(\mathcal{D}, ST)$	$L(\mathcal{D}\,\lvert ST)$
BMS-webview 1	497	59602	0.51	1184484	1173962
Chess (k-k)	75	3196	49.33	1083791	1083046
Chess (kr-k)	58	28056	12.07	688180	687120
Connect	129	67557	33.33	17777083	17774814
Ionosphere	157	351	22.29	84170	81630
Mushroom	119	8124	19.3	1113311	1111287
Pen digits	86	10992	19.77	1141982	1140795

Table 4.1: Characteristics of the used datasets. Stated are the number of attributes n, the number of transactions m, the density and the size of the standard encoding.

clickstream data from an e-commerce website of a legwear and legcare retailer and has been used in the KDD-Cup 2000 competition. A detailed description of the dataset can be found in [49].

All experiments are conducted as single threaded RapidMiner[4] processes on a Linux Intel Xeon X7550 machine with 2GHz. RapidMiner is a popular data mining software

[1] https://archive.ics.uci.edu/ml/index.html
[2] http://cgi.csc.liv.ac.uk/ frans/KDD/Software/LUCS-KDD-DN/DataSets/dataSets.html
[3] http://www.cs.rpi.edu/ zaki/Workshops/FIMI/data/
[4] `http://rapidminer.com`

written in Java, that is easily extendable by the integration of plugins. I implemented the algorithms KRIMP, SHRIMP and the two proposed algorithms based on Matrix Factorization, PIMP and MIMIKRI as operators in RapidMiner. An implementation of the algorithm SLIM is provided in C++[5]. To mine the frequent patterns that are passed as input to the algorithms KRIMP and SHRIMP, we use the FP-Growth implementation of RapidMiner. The minimum support is adjusted to every dataset such that the set of frequent patterns does not exceed a storage limit of 50GiB. This parameter is hard to set in practice. As previously mentioned, a small drop in the minimum support results often in an enormous increase in the number of obtained patterns.

Parameter settings The parameters of the algorithms that rely on matrix factorization are adjusted with respect to experiments that have been conducted on smaller datasets. We summarize briefly the specifications.

The parameters of PIMP are set as follows. The weight of the binary penalizers λ is put to 1.1, which is also the default value in the Penalizing BMF implementation available by the Python library nimfa[6]. A justification of this setting is not given by the authors, but my own experiments confirmed that this value yields comparably well results. The parameter μ that determines the impact of the compression size to the objective function is fixed to 0.2. Since the compression size is the lowest if the usage matrix and the code matrix are equal to the zero matrix, assigning a higher weight to the compression size often returns an unsatisfactory result with one very sparse matrix. Furthermore, the intermediate increase of function values at non differentiable points is more likely to occur if the usage matrix is too sparse. In order to prevent those situations, a weight of $\mu = 0.2$ is appropriate for most databases.

The parameters η_1 and η_2 of the algorithm MIMIKRI are set to $\eta_1 = 5$ and $\eta_2 = 10$. Although the function $\phi_\eta(x)$ approximates the heavyside function more accurately if the value of η is high, the convergence rate of the algorithm suffers from such an alignment. This is due to the factor $\phi'_\eta(x) = \phi_\eta(x)(1-\phi_\eta(x))$ in the gradient of the objective function F_ϕ. Therewith, if the matrices $\Phi_\eta(X)$ and $\Phi_\eta(Y)$ approach binary matrices as desired, the gradient is basically zero and the algorithm gets stuck in a stationary point that is not likely to minimize the compression size.

We discuss now the quality of the obtained code tables and the characteristics of the used datasets that might induce the observed results. Then we have a look at the performance of the Java implementations of KRIMP, SHRIMP, PIMP and MIMIKRI. Finally, we explore some selected databases more in detail by having a glimpse at the tree that composes the encoded database.

[5]http://adrem.ua.ac.be/slim

[6]http://nimfa.biolab.si/nimfa.methods.factorization.bmf.html

4.1 Comparing the Compression Quality

We elaborate in this section the differences in the achieved compression of the algorithms. The quality of the compression is usually measured by the relative compression size. This sets the compression size attained by the computed model in relation to the compression size of the standard encoding

$$\%L(\mathcal{D}, CT) = \frac{L(\mathcal{D}, CT)}{L(\mathcal{D}, ST)} \cdot 100.$$

As MIMIKRI takes only the description size of the data into account, its relative compression is measured by $\%L(\mathcal{D}\,|CT)$, putting only the description sizes of the data in relation. The algorithm SHRIMP is excluded from this analysis as it returns the same code tables as KRIMP.

	KRIMP			SLIM		PIMP		MIMIKRI
$\mathcal{D}$	*minsup*	$\lvert CT\rvert$	$\%L$	$\lvert CT\rvert$	$\%L$	$\lvert CT\rvert$	$\%L$	$\%L$
BMS-webview 1	$1 \cdot 10^{-3}$	217	93.9	965	**84.0**	20	96.6±0.1	97.3±0.5
	$5 \cdot 10^{-5}$	736	86.2			50		
Chess (k-k)	$5 \cdot 10^{-1}$	110	44.9	292	**14.7**	20	30.2±2.0	41.2±0.5
	$9 \cdot 10^{-2}$	280	27.3			50	29.0±1.9	36.8±0.5
						100	28.5±3.3	34.6±0.5
Chess (kr-k)	$1 \cdot 10^{-4}$	1740	61.7	1060	**57.5**	20	99.9±0.0	99.7±0.0
	$3 \cdot 10^{-5}$	1684	61.6			50	97.6±1.8	99.9±0.0
						100	91.8±4.6	99.8±0.0
Connect	$8 \cdot 10^{-1}$	29	54.2	1670	12.3	20	57.1±1.7	50.2±0.8
	$1 \cdot 10^{-5}$	2036	**10.9**			50	–	48.8±0.2
Ionosphere	$2 \cdot 10^{-1}$	62	64.7	240	49.7	20	53.4±1.0	54.7±1.2
	$1 \cdot 10^{-1}$	164	61.3			50	53.2±1.6	46.4±0.6
						100	49.0±1.0	**35.4±0.7**
Mushroom	$5 \cdot 10^{-2}$	524	34.7	340	**18.5**	20	56.2±3.0	50.2±1.4
	$1 \cdot 10^{-4}$	442	20.6			50	44.3±0.9	44.1±0.7
						100	38.8±4.2	41.5±0.4
Pen digits	$1 \cdot 10^{-2}$	1003	50.9	1347	**39.4**	20	83.5±0.4	70.4±1.2
	$9 \cdot 10^{-5}$	1247	42.3			50	75.4±1.7	66.2±0.5
						100	66.8±1.7	63.4±0.7

Table 4.2: Comparison of the relative compression size $\%L$ with respect to the number of maintained patterns in the code table for the regarded datasets. $|CT|$ denotes the number of elements maintained in the code table that are non-singletons.

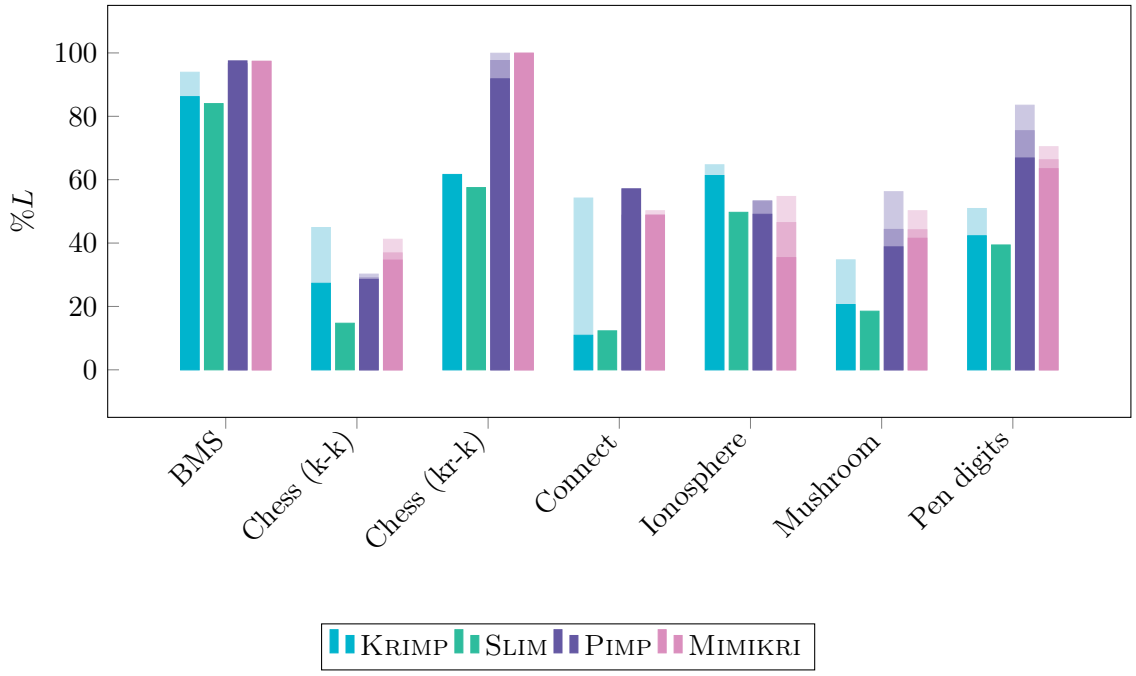


Figure 4.1: The relative compression size as it is achieved by the considered algorithms. Different shades of the barplots indicate different parameter settings, that are stated in Table 4.2. The compression size of KRIMP code tables is depicted in the paler color for the higher minimum support. The colors denoting the matrix factorization outcomes are paler for a lower rank.

Table 4.2 shows the results of the conducted experiments. For every algorithm, the number of entries in the code table that do not belong to singleton itemsets $|CT|$ and the relative compression size is denoted. For the experiments of KRIMP, the parameter *minsup* is indicated as well. There are two different values of the minimum support for every dataset. The first one is used for the Java processes and the second one is the value that has been used in the experiments with the C++ implementation of the authors [7]. Since items are represented as short integers in the C++ implementation and strings are used by the RapidMiner FP-Growth operator, the storage capability of the frequent pattern set is much higher in the first representation. The compression size with respect to the lower value of the minimum support denotes therewith a bound on the quality that can be achieved by KRIMP.

The parameter $|CT|$ refers for matrix factorization based algorithms to the rank of the factorization. The obtained compression sizes are averaged over 5 runs for each of the specified ranks 20, 50 and 100 using 1000 iterations. For the larger databases Connect and BMS-webview 1, the computation of ranks greater than 20 required more than one week and have been aborted.

[7] https://people.mmci.uni-saarland.de/ jilles/prj/krimp/

With regard to the number of elements that are selected by the algorithms KRIMP and SLIM, the databases can roughly be divided into three categories. The first one contains databases that are described sufficiently using only few codes, let's say up to 500. Prototypes of this category are the Chess (k-k), Ionosphere and Mushroom datasets. The second category consists of databases that require a larger vocabulary and have code tables that contain more than 1000 code words. To this category belong the datasets Pen digits and the Chess (kr-k). The Connect dataset is somewhere between those categories, it can be described acceptably by few codes but extending the set of codes by orders of magnitudes decreases the compression size significantly. The third category covers the datasets that are barely compressible, like the BMS-webview 1 dataset.

The obtained compression quality is visualized in Figure 4.1. We can see that SLIM code tables provide the shortest compressions for almost all datasets. The differences of realized compressions by KRIMP and SLIM code tables are often not drastic, if the parameter *minsup* is sufficiently low. Increasing the minimum support as depicted in Table 4.2 increases the compression size visibly, but never as extremely as for the Connect dataset.

The matrix factorizations do not differ in their compression size that much. Depending on the dataset, one or the other approach is describing the data more suitably. Datasets that can be expressed by a few hundreds of codes can also be summarized well using much smaller code tables by PIMP and MIMIKRI. The Ionosphere dataset attains even its lowest compression size for 100 non-singleton codes generated by MIMIKRI. This dataset seems to have a structure that can much easier be encoded if overlapping is allowed. No other algorithm achieves such a good compression.

By contrast, an improvement of the compression in relation to that of the standard code table can barely be observed using only few codes for the BMS-webview 1 and Chess (kr-k) datasets. This suggests that tilings of those datasets contain multiple smaller tiles, that do not allow for a compact representation by few submatrices full of ones. The rank would have to be set much higher to obtain compressions comparable to those of KRIMP and SLIM.

The adjustment of the different ranks has a comparably small effect on the MIMIKRI factorizations. To understand this phenomenon, we have a look at the convergence plots for selected datasets.

Exploring the Convergence Figure 4.2 shows the averaged function values with respect to the number of conducted gradient descent steps. On the left, we can see the convergence of the objective function values for PIMP and on the right those of the MIMIKRI algorithm. We display these progresses for the Ionosphere, Chess (kr-k) and Mushroom dataset, since they exhibit apparently different structures. We might notice that the curves of MIMIKRI are nonincreasing while especially for a higher rank of 100, the function value

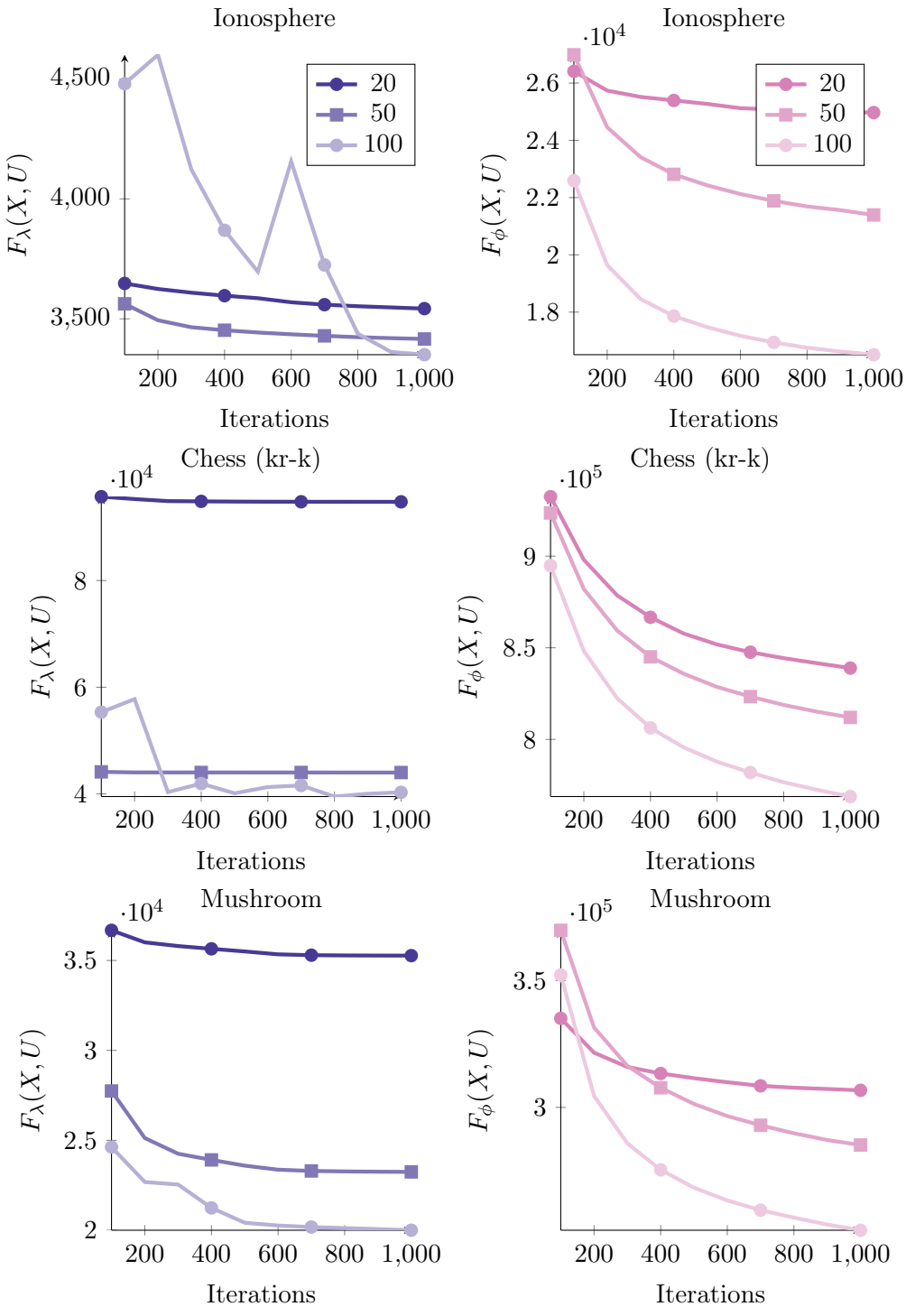


Figure 4.2: Convergence plots of the PIMP (left) and MIMIKRI (right) algorithm. The displayed function values with respect to the number of iterations are averaged over two runs on the respective dataset. The values in the legend denote the specified rank of the factorization.

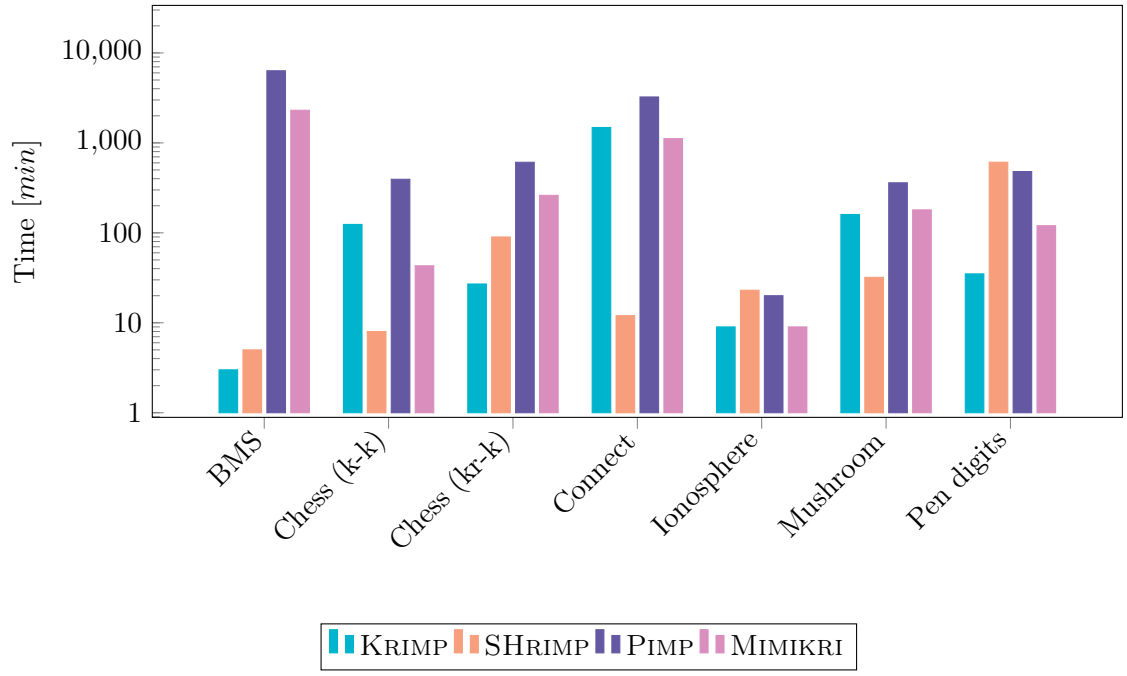


Figure 4.3: Runtime in minutes of the algorithms KRIMP, SHRIMP, PIMP and MIMIKRI. The rank of the factorization is given as 20 and the minimum support parameter is specified as denoted for the Java processes in Table 4.2. The time is displayed on a logarithmic axis.

F_λ increases for some steps of PIMP. The number of 1000 iterations seems to be sufficient in most cases with regard to PIMP. After 400 iterations, the function values decrease barely for ranks of 20 or 50.

Concerning the curve progressions of MIMIKRI, especially for a higher rank of 100, using more iterations than 1000 is likely to decrease the function value remarkably. The Chess (kr-k) dataset is the largest of the depicted ones. The convergence rate of this dataset is rather small and a much higher number of iterations appear to be beneficial to factorizations of all ranks.

Since the factorization of such a dataset already takes quite its time for lower ranks, a procedure that chooses the stepsize appropriately and also faster than the expensive backtracking linesearch algorithm, would be desirable. The possibility to apply alternate methods depends however strongly on the characteristics of the objective function like Lipschitz continuity. The compression size, that is derived with the logarithmic function is however not Lipschitz continuous. Further research is thereby needed to obtain more accurate results of the MIMIKRI algorithm

4.2 Runtime Evaluation

We evaluate in the following the performance of the Java implementations of KRIMP, SHRIMP, PIMP and MIMIKRI. We focus especially on the comparison of the efficiency of

the tree-based algorithm SHrimp to its role model Krimp. Figure 4.3 shows the overall runtime on a logarithmic scale. This includes the generation of frequent patterns for the algorithms Krimp and SHrimp and the thresholding of the matrices with respect to 100 equidistant values between zero and one for Pimp and Mimikri. We observe that the runtime of the matrix factorizations depends solely on the size of the database, while different criteria hold for the heuristic algorithms. The calculation of a factorization requires for the large dataset BMS-webview 1 several days. On the smallest dataset Ionosphere, a factorization can be derived in 10 minutes. The computation of a factorization by Pimp takes for all datasets a noticeably longer time. This indicates that the linesearch requires more iterations to derive a suitable stepsize than in the smooth case of the algorithm Mimikri.

To have a closer look at the performances of Krimp and SHrimp we display in Figure 4.4 the required time to compute the usage and the integration of a pattern into the code table. We select six out of the seven databases to be examined in this view. It is eye-catching that SHrimp has an exceptionally shorter runtime on the Chess (k-k), Connect and Mushroom datasets. By contrast, for the BMS-webview 1, Chess (kr-k), Ionosphere and Pen digits datasets, Krimp performs superior.

The latter datasets are also not very well compressible. In fact, we observe a strong correlation between the minimal achievable relative compression size and the performance of SHrimp in comparison to Krimp. The datasets that have a minimal compression size of less that 20% are also the ones that are depicted on the right of Figure 4.4, where SHrimp is able to decide about the contribution of a pattern to the encoding much more efficiently. Ionosphere has a minimal relative compression size of 35%, obtained by the algorithm Pimp. With respect to the heuristic encoding, the best compression size is given by about 50%. Chess (kr-k) and BMS-webview 1 have at least a relative compression size of about 60% and Krimp is accordingly eminently faster.

This observation is also reasonable since the compression size is dependent on the sum and entropy of usages as discussed in Section 3.4.4. Having a small sum of usages means also that less codes are used to describe a transaction. Therewith, there are also less active nodes in a branch from the root to the leaves in the database tree and the tree can be traversed more efficiently. A low entropy in the usage of codes is transferable to a high usage for few codes and small usages for as few codes as possible. This means also that the nodes of codes with a high usage summarize likely many transactions in one branch and the tree is less ramified.

To picture these thoughts, excerpts of the database trees are depicted for the exceptionally well compressible Connect dataset an the barely compressible BMS-webview 1 dataset in Figure 4.5. The two branches with the highest usage of the root nodes children are displayed up to a depth of four. The information of the nodes is summarized to the number of inactive and active items and the usage of the node.

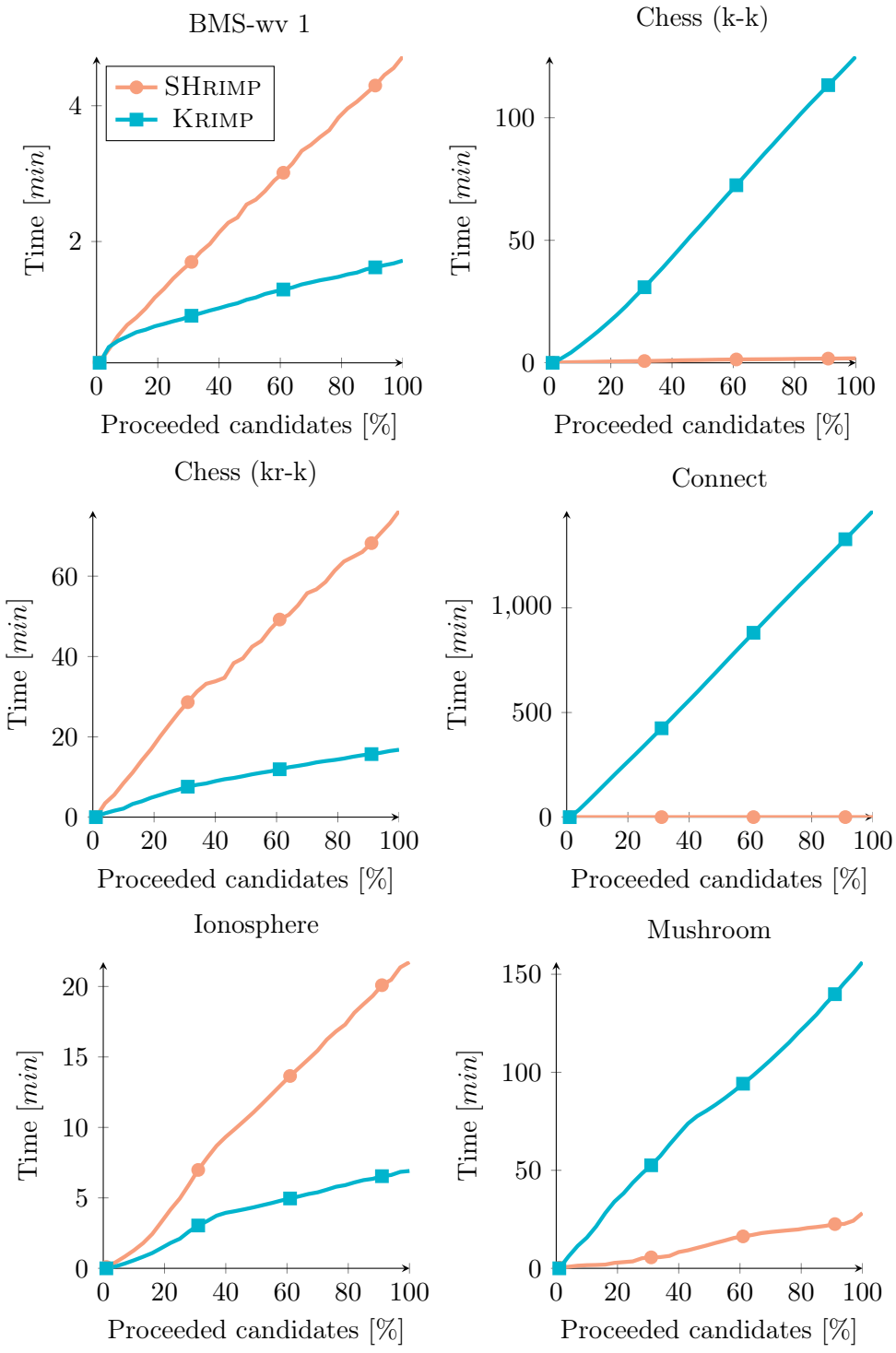


Figure 4.4: Runtime for KRIMP (blue quadratic marks) and SHRIMP (orange circle marks) in relation to the percentage of examined patterns for the different datasets.

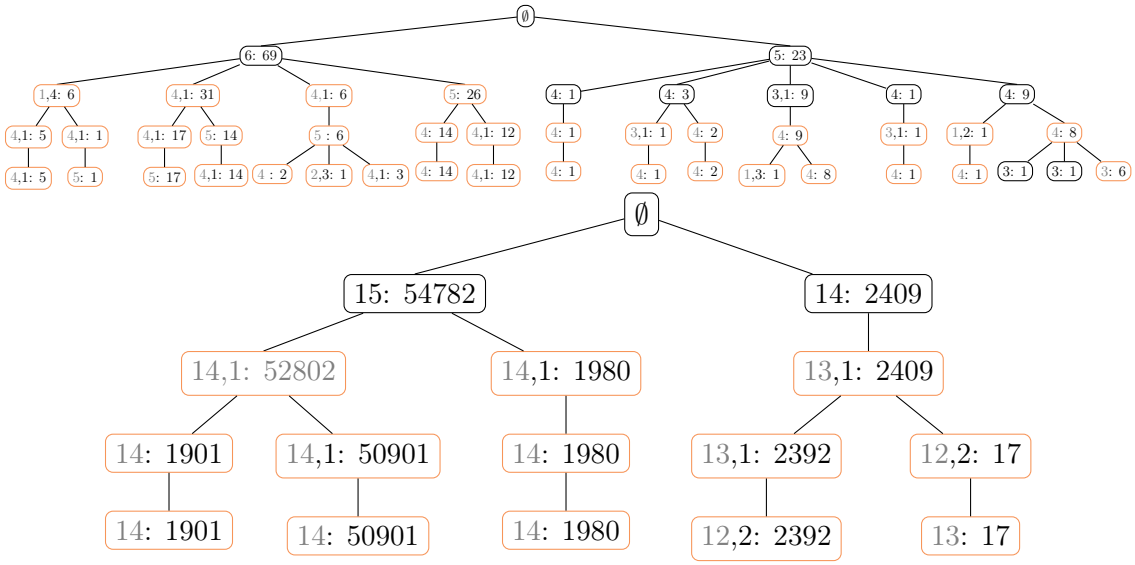


Figure 4.5: A sample of the database tree from the BMS-webview 1 (above) and the Connect dataset (below). Nodes are depicted up to a depth of 4 and the branches of the root nodes children with the highest usage are selected. Every node contains the cardinality of active items (black) and inactive items if existing (grey). The number after the colon denotes the usage of the node.

It is striking how branched the tree of the BMS dataset appears in relation to the Connect database tree. The usage of the nodes on top of the tree is also indicative. The node with the highest usage in the Connect dataset tree is the one depicted in the left branch with a usage of 54,782. That means that the encoding of about 55,000 from the less than 70,000 transactions in the Connect dataset is determined by this node that contains a quite long pattern of 15 items.

The usage of nodes in the BMS-webview 1 dataset is on the contrary more evenly distributed. The node with the highest usage has a usage of 69, while the BMS dataset contains also about 60,000 transactions. The contained patterns have also remarkably less items, up to 6. Considering these observations, it is understandable why SHRIMP performs better on well compressible datasets.

Chapter 5

Conclusion and further Work

In this work, we had a look at the challenges that arrive with large amounts of binary data. A binary database is like a cryptic document that has to be deciphered if one wants to extract its information. We regarded this information in terms of patterns that describe the database in a compact, probably non-redundant way. Code tables are the utilized objects that yield a description of the database, like a dictionary of the vokabulary in that a document is expressed. The (code-)words with that the database can be phrased are determined such that the representation, the encoded database, is as short as possible. This methodology is embodied by the MDL principle.

In this work, I investigated the different approaches that derive the desired information by code tables. Existing techniques include Krimp, Slim and the recently proposed algorithm SHrimp. These algorithms rely on heuristic procedures to face the problem of the exponentially many possibilities to encode a database.

With regard to Krimp we went through the multiple technical details that make an implementation of this algorithm efficient. The applied techniques often rely on fundamental insights on the nature of the used encoding. Those provide a more profound view on this procedure. While Krimp tries to reduce the enormous amount of patterns by a suitable selection, we have also seen how a code table can be created by a successive integration of the patterns that make a good contribution to the current modeling with Slim. For the heuristic determination of the patterns with the greatest potential, we have obtained a theoretical justification. This has been achieved by the derivation of a bound that provides a lower threshold on the contribution of a pattern to the encoding. With the algorithm SHrimp we explored the representation of an encoding by a tree. In this representation, the calculation of the contribution of a pattern to the encoding required further insights on the underlying mechanisms of encoding. Especially the modeling of the code table as a tree made an efficient identification of the affected parts of the encoded database possible.

A completely new approach to determine an encoding of a database has been introduced by the application of matrix factorization. This approach is not restricted to any

heuristic assumptions, but introduces parameters that have to be specified by the user. In this variant, objective functions have been formulated that describe the quality of the compression in the space of real numbers. The objectives can be minimized using common methods of numerical optimization.

I proposed two different versions in that the objective function can be formulated. The first one is based on nonlinear programming, where we used penalizing terms to solve the problem description. This is the PIMP algorithm. The second one allows for overlapping codes and approximates the obtained compression size by the value of the objective function. We called this procedure the MIMIKRI algorithm.

The conducted experiments on 7 popular datasets for pattern mining tasks with different characteristics affirmed that SLIM yields a superior compression. We have seen that SHRIMP has an exceptionally better performance for well compressible datasets in relation to KRIMP. The utilization of the properties that are induced by the standard encoding in a tree structure, might also be beneficial applicable to SLIM. Determining the estimated compression gain by patterns that are created by a join of two nodes is likely to speed up the calculation of top-k compressing patterns.

The approaches relied on matrix factorization were capable to compress the datasets in a similar order of magnitude as SLIM and KRIMP , sometimes even better, using much fewer itemsets. We saw that the convergence rate of MIMIKRI can be rather slow for larger datasets. By contrast, PIMP suffers from its discontinuous properties. There are multiple ways in that the convergence of these algorithms might be improved. First it might be worthwhile investigating how PIMP performs using a systematically decreasing stepsize, e.g. $\alpha_k = \frac{1}{k}$ as it is often performed for subgradient methods [46]. Although this will not improve the convergence rate, such a stepsize strategy reduces the costs of the linesearch. We have seen that particularly for higher ranks, a stepsize can often not be derived that decreases the function value. Determining the stepsize in a more static way enables us to take more (subgradient descent) steps in fewer time.

Another approach to minimize a nonsmooth and nonconvex function by alternating updates has been introduced by Bolte et al. [9]. They apply the proximal operator to obtain the exact minimum of a linearized version of the objective function. This method requires however the analytical derivation of the proximal operator, which is not trivial for the rather complicated function that describes the compression size.

With regard to larger databases, one has to pose the question how they can be compressed efficiently. Especially for sparse databases, which often go along with a large size, no encoding is able to yield a good performance as we have seen for the BMS-webview 1 dataset. The possibility to reflect the encoding by a matrix factorization offers yet multiple possibilities to encode the database. As we have seen for the MIMIKRI algorithm, a modular construction system where one can add and remove codes for every transaction is

also possible. This might be the key idea to describe sparse datasets that inhibit a noisy structure.

Further applications that might make use of compressing patterns include for instance the field of collaborative filtering. In this area of recommender systems, biclustering is applied to cluster users and opinions simultaneously. Recommendations can then be obtained by an assignment of new observations to the respective cluster by a nearest neighbour procedure [20]. As we have seen that the compression of a database exhibits strong relations to biclustering, the use of compressing patterns in this area might offer new opportunities.

Another research area that makes use of biclustering algorithms is the field of textmining. Biclusters identify the vocabulary that is used with respect to groups of documents. The identification of correlating words and documents finds its interpretation in the broad field of topic modeling [8]. The state of the art algorithm to extract the topics of a document collection is the Latent Dirichlet Allocation [7]. This is a generative approach that assumes that documents are generated by an underlying distribution. The aim of related algorithms is to discover the parameters of this distribution. A different view on the encoding by MDL is therefore given if one assumes that the probability function as defined by the usage *generates* a database. As the algorithms that rely on the estimation of the underlying probability distribution yield superior results to the ones that are obtained by matrix factorization approaches until now [2], investigating this approach with regard to Krimp seems very interesting.

List of Figures

List of Algorithms

Bibliography

[1] Agrawal, Rakesh, Tomasz Imieliński and Arun Swami: *Mining Association Rules Between Sets of Items in Large Databases.* SIGMOD Rec., 22(2):207–216, 1993.

[2] Arora, Sanjeev, Rong Ge and Ankur Moitra: *Learning topic models - Going beyond SVD.* In *Proceedings - Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 1–10, 2012.

[3] Bayardo Jr, Roberto J: *Efficiently mining long patterns from databases.* In *ACM Sigmod Record*, volume 27, pages 85–93. ACM, 1998.

[4] Berry, Michael W, Murray Browne, Amy N Langville, V Paul Pauca and Robert J Plemmons: *Algorithms and applications for approximate nonnegative matrix factorization.* Computational statistics & data analysis, 52(1):155–173, 2007.

[5] Bertsekas, Dimitri P: *Nonlinear programming.* Athena Scientific, 1999.

[6] Blachon, Sylvain, Ruggero G Pensa, Jérémy Besson, Céline Robardet, Jean-François Boulicaut and Olivier Gandrillon: *Clustering formal concepts to discover biologically relevant knowledge from gene expression data.* In silico biology, 7(4):467–483, 2007.

[7] Blei, David, Lawrence Carin and David Dunson: *Probabilistic topic models.* IEEE Signal Processing Magazine, 27(6):55–65, 2010.

[8] Blei, David M and John D Lafferty: *Topic models.* Text mining: classification, clustering, and applications, 10:71, 2009.

[9] Bolte, Jérôme, Shoham Sabach and Marc Teboulle: *Proximal alternating linearized minimization for nonconvex and nonsmooth problems.* Mathematical Programming, 146(1-2):459–494, 2014.

[10] Boulicaut, Jean-François, Artur Bykowski and Christophe Rigotti: *Free-sets: a condensed representation of boolean data for the approximation of frequency queries.* Data Mining and Knowledge Discovery, 7(1):5–22, 2003.

[11] Burdick, Douglas, Manuel Calimlim and Johannes Gehrke: *MAFIA: A maximal frequent itemset algorithm for transactional databases.* In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 443–452. IEEE, 2001.

[12] Busygin, Stanislav, Oleg Prokopyev and Panos M Pardalos: *Biclustering in data mining.* Computers & Operations Research, 35(9):2964–2987, 2008.

[13] Calders, Toon and Bart Goethals: *Mining all non-derivable frequent itemsets.* In *Principles of Data Mining and Knowledge Discovery*, pages 74–86. Springer, 2002.

[14] Cohen, Edith, Mayur Datar, Shinji Fujiwara, Aristides Gionis, Piotr Indyk, Rajeev Motwani, Jeffrey D Ullman and Cheng Yang: *Finding interesting associations without support pruning.* Knowledge and Data Engineering, IEEE Transactions on, 13(1):64–78, 2001.

[15] Cover, T.M. and J.A. Thomas: *Elements of information theory.* Wiley-Interscience, 2006.

[16] Deerwester, Scott, Susan T. Dumais, George W. Furnas, Thomas K. Landauer and Richard Harshman: *Indexing by latent semantic analysis.* Journal of the American Society for Information Science, 41:391–407, 1990.

[17] Ding, Chris, Tao Li, Wei Peng and Haesun Park: *Orthogonal nonnegative matrix t-factorizations for clustering.* In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 126–135. ACM, 2006.

[18] Ding, Chris HQ, Xiaofeng He and Horst D Simon: *On the Equivalence of Nonnegative Matrix Factorization and Spectral Clustering.* In *SDM*, volume 5, pages 606–610. SIAM, 2005.

[19] Geerts, Floris, Bart Goethals and Taneli Mielikäinen: *Tiling databases.* In *Discovery science*, pages 278–289. Springer, 2004.

[20] Gong, Songjie: *A collaborative filtering recommendation algorithm based on user clustering and item clustering.* Journal of Software, 5(7):745–752, 2010.

[21] Grünwald, P.D.: *Minimum Description Length Principle.* MIT press, Cambridge, MA, 2007.

[22] Han, Jiawei, Jian Pei and Yiwen Yin: *Mining Frequent Patterns Without Candidate Generation.* SIGMOD Rec., 29(2):1–12, 2000.

[23] Hess, Sibylle, Nico Piatkowski and Katharina Morik: *SHrimp: Descriptive Patterns in a Tree.* 2014.

[24] Lee, Daniel D and H Sebastian Seung: *Learning the parts of objects by non-negative matrix factorization.* Nature, 401(6755):788–791, 1999.

[25] Lee, Daniel D and H Sebastian Seung: *Algorithms for non-negative matrix factorization.* In *Advances in neural information processing systems*, pages 556–562, 2001.

[26] Leeuwen, Matthijs van, Francesco Bonchi, Börkur Sigurbjörnsson and Arno Siebes: *Compressing tags to find interesting media groups.* In *CIKM*, pages 1147–1156. ACM, 2009.

[27] Li, Paul Vitányi Ming: *An Introduction to Kolmogorov Complexity and Its Applications.* Springer, 1997.

[28] Li, Tao: *A general model for clustering binary data.* In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 188–197. ACM, 2005.

[29] Li, Tao and Chris Ding: *The relationships among various nonnegative matrix factorization methods for clustering.* In *Data Mining, 2006. ICDM'06. Sixth International Conference on*, pages 362–371. IEEE, 2006.

[30] Li, Tao and Chris Ding: *The relationships among various nonnegative matrix factorization methods for clustering.* In *Data Mining, 2006. ICDM'06. Sixth International Conference on*, pages 362–371. IEEE, 2006.

[31] Lin, Chih-Jen: *Projected gradient methods for nonnegative matrix factorization.* Neural computation, 19(10):2756–2779, 2007.

[32] Mampaey, Michael, Nikolaj Tatti and Jilles Vreeken: *Tell me what i need to know: succinctly summarizing data with itemsets.* In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 573–581. ACM, 2011.

[33] Mannila, Heikki and Hannu Toivonen: *Levelwise Search and Borders of Theories in Knowledge Discovery*, 1997.

[34] Miettinen, Pauli, Taneli Mielikainen, Aristides Gionis, Gautam Das and Heikki Mannila: *The discrete basis problem.* Knowledge and Data Engineering, IEEE Transactions on, 20(10):1348–1362, 2008.

[35] Mordukhovich, Boris S: *Variational Analysis and Generalized Differentiation I: Basic Theory*, volume 330. Springer Science & Business Media, 2006.

[36] Paatero, Pentti and Unto Tapper: *Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values.* Environmetrics, 5(2):111–126, 1994.

[37] Pasquier, Nicolas, Yves Bastide, Rafik Taouil and Lotfi Lakhal: *Discovering frequent closed itemsets for association rules.* In *Database Theory—ICDT'99*, pages 398–416. Springer, 1999.

[38] Pudi, Vikram and Jayant R Haritsa: *Generalized closed itemsets for association rule mining.* In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 714–716. IEEE, 2003.

[39] Rissanen, J.: *Modeling By Shortest Data Description.* Automatica, 14:465–471, 1978.

[40] Siebes, Arno and René Kersten: *A Structure Function for Transaction Data.* In *SDM*, pages 558–569. SIAM, 2011.

[41] Smets, Koen and Jilles Vreeken: *Slim: Directly Mining Descriptive Patterns.* In *SDM*, pages 236–247. SIAM / Omnipress, 2012.

[42] Van Benthem, Mark H. and Michael R. Keenan: *Fast algorithm for the solution of large-scale non-negativity-constrained least squares problems.* Journal of Chemometrics, 18(10), 2004.

[43] Vreeken, Jilles, Matthijs van Leeuwen and Arno Siebes: *Characterising the difference.* In *KDD*, pages 765–774. ACM, 2007.

[44] Vreeken, Jilles, Matthijs van Leeuwen and Arno Siebes: *Preserving Privacy through Data Generation.* In *ICDM*, pages 685–690. IEEE Computer Society, 2007.

[45] Vreeken, Jilles, Matthijs van Leeuwen and Arno Siebes: *Krimp: mining itemsets that compress.* Data Min. Knowl. Discov., 23:169–214, 2011.

[46] Wright, Stephen J and Jorge Nocedal: *Numerical optimization*, volume 2. Springer New York, 1999.

[47] Zhang, Zhong-Yuan, Tao Li, Chris Ding, Xian-Wen Ren and Xiang-Sun Zhang: *Binary matrix factorization for analyzing gene expression data.* Data Mining and Knowledge Discovery, 20(1):28–52, 2010.

[48] Zhang, Zhongyuan, Chris Ding, Tao Li and Xiangsun Zhang: *Binary matrix factorization with applications.* In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 391–400. IEEE, 2007.

[49] Zheng, Zijian, Ron Kohavi and Llew Mason: *Real world performance of association rule algorithms.* In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 401–406. ACM, 2001.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kentlich gemacht habe.

Dortmund, den April 22, 2015

Sibylle Hess