KRIMP: mining itemsets that compress

**Jilles Vreeken · Matthijs van Leeuwen ·
Arno Siebes**

Abstract   One of the major problems in pattern mining is the explosion of the num-
ber of results. Tight constraints reveal only common knowledge, while loose con-
straints lead to an explosion in the number of returned patterns. This is caused by large
groups of patterns essentially describing the same set of transactions. In this paper
we approach this problem using the MDL principle: the best set of patterns is that set
that compresses the database best. For this task we introduce the KRIMP algorithm.
Experimental evaluation shows that typically only hundreds of itemsets are returned;
a dramatic reduction, up to seven orders of magnitude, in the number of frequent
item sets. These selections, called code tables, are of high quality. This is shown with
compression ratios, swap-randomisation, and the accuracies of the code table-based

J. Vreeken (✉) · M. van Leeuwen · A. Siebes
Algorithmic Data Analysis, Department of Information and Computing Sciences, Faculty of Science,
Universiteit Utrecht, Utrecht, The Netherlands
e-mail: jillesv@cs.uu.nl

M. van Leeuwen
e-mail: mleeuwen@cs.uu.nl

A. Siebes
e-mail: arno@cs.uu.nl

J. Vreeken
ADReM, Department of Mathematics and Computer Science, Faculty of Science,
University of Antwerp, Antwerp, Belgium

KRIMP classifier, all obtained on a wide range of datasets. Further, we extensively evaluate the heuristic choices made in the design of the algorithm.

Keywords MDL · Pattern mining · Pattern selection · Itemsets

1 Introduction

1.1 Patterns

Without a doubt, *pattern mining* is one of the most important concepts in data mining. In contrast to *models*, patterns describe only part of the data (see e.g. Hand et al. 2002; Morik et al. 2005). In this paper, we consider one class of pattern mining problems, viz., *theory mining* (Mannila and Toivonen 1997). In this case, the patterns describe interesting subsets of the database.

Formally, this task has been described by Mannila and Toivonen (1996) as follows. Given a database $\mathcal{D}$, a language $\mathcal{L}$ defining subsets of the data and a selection predicate q that determines whether an element $\phi \in \mathcal{L}$ describes an interesting subset of $\mathcal{D}$ or not, the task is to find

$$\mathcal{T}(\mathcal{L}, \mathcal{D}, q) = \{ \phi \in \mathcal{L} \mid q(\mathcal{D}, \phi) \text{ is true } \}.$$

That is, the task is to find *all* interesting subsets.

The best known instance of theory mining is *frequent set mining* (Agrawal et al. 1996); this is the problem we will consider throughout this paper. The standard example for this is the analysis of shopping baskets in a supermarket. Let $\mathcal{I}$ be the set of items the store sells. The database $\mathcal{D}$ consists of a set of *transactions* in which each transaction t is a subset of $\mathcal{I}$. The pattern language $\mathcal{L}$ consists of *itemsets*, i.e. again sets of items. The *support* of an itemset X in $\mathcal{D}$ is defined as the number of transactions that contain X, i.e.

$$supp_{\mathcal{D}}(X) = |\{t \in \mathcal{D} \mid X \subseteq t\}|.$$

The 'interestingness' predicate is a threshold on the support of the itemsets, the *minimal support: minsup*. In other words, the task in frequent set mining is to compute

$$\{X \in \mathcal{L} \mid supp_{\mathcal{D}}(X) \geq \text{minsup}\}.$$

The itemsets in the result are called *frequent* itemsets. Since the support of an itemset decreases w.r.t. set inclusion, the *A Priori* property,

$$X \subseteq Y \Rightarrow supp_{\mathcal{D}}(Y) \leq supp_{\mathcal{D}}(X),$$

a simple level-wise search algorithm suffices to compute all the frequent itemsets. Many efficient algorithms for this task are known, see, e.g. Goethals and Zaki (2003). Note, however, that since the size of the output can be exponential in the number

of items, the term efficient is used w.r.t. the size of the output. Moreover, note that whenever $\mathcal{L}$ and q satisfy an A Priori like property, similarly efficient algorithms exist (Mannila and Toivonen 1996).

1.2 Sets of patterns

A major problem in frequent itemset mining, and pattern mining in general, is the so-called *pattern explosion*. For tight interestingness constraints, e.g. high minsup thresholds, only few, but well-known, patterns are returned. However, when the constraints are loosened, pattern discovery methods quickly return humongous amounts of patterns; the number of frequent itemsets is often many orders of magnitude larger than the number of transactions in the dataset.

This pattern explosion is caused by the locality of the minimal support constraint; each individual itemset that satisfies the constraint is added to the result set, independent of the already returned sets. Hence, we end up with a rather redundant set of patterns, in which many patterns essentially describe the same part of the database. One could impose additional constraints on the individual itemsets to reduce their number, such as closed frequent itemsets (Pasquier et al. 1999). While this somewhat alleviates the problem, redundancy remains an issue.

We take a different approach: rather than focusing on the individual frequent itemsets, we focus on the resulting set of itemsets. That is, we want to find the *best set* of (frequent) itemsets. The question is, of course, what is the best set? Clearly, there is no single answer to this question. For example, one could search for small sets of patterns that yield good classifiers, or show maximal variety (Knobbe and Ho 2006a,b).

One of the main reasons for mining a dataset is to gain insight in the data. Hence, for us, the best set of patterns is the set of patterns that *describes the data best*. Clearly, "describing the data best" is still an ill-defined concept. To make it precise, we use the *Minimal Description Length Principle* (MDL) (Rissanen 1978; Grünwald 2005, 2007).

One could summarize this approach by the slogan: *the best model compresses the data best*. By taking this approach we do not try to compress the set of frequent itemsets, rather, we want to find that set of frequent itemsets that yields the best lossless compression of the *database*.

The MDL principle provides us a fair way to balance the complexities of the compressed database and the encoding. Note that both need to be considered in the case of lossless compression. Intuitively, we can say that if the encoding is too simple, i.e. it consists of too few itemsets, the database will hardly be compressed. On the other hand, if we use too many, the code table for coding/decoding the database will become too complex.

Considering the combination of the complexities of the compressed data and the encoding is the cornerstone of the MDL principle; it ensures that the model will not be overly elaborate or simplistic w.r.t. the complexity of the data.

While MDL removes the need for user defined parameters, it comes with its own problems: only heuristics, no guaranteed algorithms. However, our experiments show that these heuristics give a dramatic reduction in the number of itemsets. Moreover, the

set of patterns discovered is characteristic of the database as independent experiments verify; see Sect. 7.

We are not the first to address the pattern explosion, nor are we the first to use MDL. We are the first, however, to employ the MDL principle to select the best pattern set. For a discussion of related work, see Sect. 2.

A primary version of the KRIMP algorithm (although not yet under that name) was published as Siebes et al. (2006) and a primary version of the KRIMP classifier as van Leeuwen et al. (2006). Here, we thoroughly discuss the theory and choices, as well as providing extensive experimental validation of the methods on 27 datasets. In particular, we further evaluate the heuristic choices made in the KRIMP algorithm, show that the selected itemsets model relevant structure in the data and that the method is robust w.r.t. noise.

The paper is organised as follows. First, we discuss related work in Sect. 2. Next, we cover the theory of using MDL for selecting itemsets, after which we define our problem formally and analyse its complexity. We introduce the heuristic KRIMP algorithm for solving the problem in Sect. 4. In a brief interlude we provide a small sample of the results. We continue with theory on using MDL for classification, and introduce the KRIMP classifier in Sect. 6. Section 7 provides extensive experimental validation of our method, as well as an evaluation of the heuristic choices made in the design of the KRIMP algorithm. We round up with discussion in Sect. 8 and conclude in Sect. 9.

2 Related work

2.1 MDL in data mining

MDL was introduced by Rissanen (1978) as a noise-robust model selection technique. In the limit, refined MDL is asymptotically the same as the Bayes Information Criterion (BIC), but the two may differ (strongly) on finite data samples (Grünwald 2007). We are not the first to use MDL, nor are we the first to use MDL in data mining or machine learning. Many, if not all, data mining problems can be related to Kolmogorov Complexity, which means they can be practically solved through compression (Faloutsos and Megalooikonomou 2007), e.g. clustering (unsupervised learning), classification (supervised learning), distance measurement. Other examples include feature selection (Pfahringer 1995), finding temporally surprising patterns (Chakrabarti et al. 1998), defining a parameter-free distance measure on sequential data (Keogh et al. 2004, 2007), discovering communities in matrices (Chakrabarti et al. 2004), and evolving graphs (Sun et al. 2007).

2.2 Summarizing frequent itemsets

Most, if not all pattern mining approaches suffer from the pattern explosion. As discussed before, its cause lies primarily in the large redundancy in the returned pattern sets. This has long since been recognised as a problem and has received ample attention.

To address this problem, closed (Pasquier et al. 1999) and non-derivable (Calders and Goethals 2002) itemsets have been proposed, which both provide concise lossless representations of the original collection of frequent itemsets. That is, the frequencies of all itemsets in the original collection can be reconstructed from the reduced collection. However, the reduction provided by these exact methods deteriorates even under small amounts of noise. Similar in goal, but providing a partial (i.e. lossy) representation, are maximal itemsets (Bayardo 1998) and δ-free sets (Crémilleux and Boulicaut 2002). Along these lines, Yan et al. (2005) proposed a method that selects k representative patterns that together summarize the frequent pattern set. To the same end, Wang and Parthasarathy (2006) propose to use Markov Random Fields to select those itemsets of which the frequencies cannot be reconstructed within a specified accuracy threshold. The problem is approached similarly by Xin et al. (2005), albeit by clustering the itemsets and using the 'centroids' itemsets as representatives for calculating frequencies.

2.3 Summarizing data

Recently, the approach of finding small subsets of informative patterns that describe the database has attracted a significant amount of research (Mielikäinen and Mannila 2003; Knobbe and Ho 2006a; Bringmann and Zimmermann 2007). KRIMP clearly also falls in this category.

First, there are the methods that provide a lossy description of the data. These strive to describe just part of the data, and may so by definition overlook interesting or even important patterns. Summarization as proposed by Chandola and Kumar (2007) is such a compression-based approach that identifies a group of itemsets such that each transaction is summarized by one itemset with as little loss of information as possible. Wang and Karypis (2006) find summary sets, sets of itemsets such that each transaction is (partially) covered by the largest itemset that is frequent.

Pattern Teams (Knobbe and Ho 2006b) are groups of most-informative length-k itemsets (Knobbe and Ho 2006a). These are exhaustively selected through an external criterion, e.g. joint entropy or classification accuracy. As this approach is computationally intensive, the number of team members is typically <10. Bringmann and Zimmermann (2007) proposed a similar selection method that can consider larger pattern sets. However, it also requires the user to choose a quality measure to which the pattern set has to be optimized, unlike our parameter-free and lossless method.

Second, in the category of lossless data description, we recently (Siebes et al. 2006) introduced the MDL-based KRIMP algorithm. In this paper we extend the theory, tune the pruning techniques and thoroughly verify the validity of the chosen heuristics, as well as provide extensive experimental evaluation of the quality of the returned code tables.

Tiling (Geerts et al. 2004) is closely related to our approach. A tiling is a cover of the database by a group of (overlapping) item sets. Itemsets with maximal uncovered area are selected, i.e. as few as possible itemsets cover the data. Unlike our approach, model complexity is not explicitly taken into account. Another major difference in the outcome is that KRIMP selects more specific (longer) itemsets. Xiang et al. (2008)

proposed a slight reformulation of Tiling that allows tiles to also cover transactions in which not all its items are present.

Two approaches inspired by KRIMP are Pack (Tatti and Vreeken 2008) and LESS (Heikinheimo et al. 2009). Both approaches consider the data 0/1 symmetric, unlike here, where we only regard items that are present (1s). LESS employs a generalised KRIMP encoding to select only tens of low-entropy sets (Heikinheimo et al. 2007) as lossless data descriptions, but attains worse compression ratios than KRIMP. Pack does provide a significant improvement in that regard. It employs decision trees to succinctly transmit individual attributes, and these models can be built from data or candidate sets. Typically, Pack selects many more itemsets than KRIMP.

Our approach seems related to the set cover problem (Karp 1972), as both try to cover the data with sets. Although NP-complete, fast approximation algorithms exist for set cover. These are not applicable for our setup though, as in set cover the complexity of the model is not taken into account. Another difference is that we do not allow overlap between itemsets. 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 here we do not allow overlap.

3 Theory

In this section we state our problem formally. First we briefly discuss the MDL principle. Next we introduce our models: code tables. We show how we can encode a database using such a code table, and what the total size of the coded database is. With these ingredients, we formally state the problems studied in this paper. Throughout the paper all logarithms have base 2.

3.1 MDL

MDL (Minimum Description Length) (Rissanen 1978; Grünwald 2005), like its close cousin MML (Minimum Message Length) (Wallace 2005), is a practical version of Kolmogorov Complexity (Li and Vitányi 1993). All three embrace the slogan *Induction by Compression*. For MDL, this principle can be roughly described as follows.

Given a set of models[1] $\mathcal{H}$, the best model $H \in \mathcal{H}$ is the one that minimises

$$L(H) + L(\mathcal{D} \mid H)$$

in which

- $L(H)$ is the length, in bits, of the description of H, and
- $L(\mathcal{D} \mid H)$ is the length, in bits, of the description of the data when encoded with H.

---

[1] MDL-theorists tend to talk about *hypothesis* in this context, hence the $\mathcal{H}$; see (Grünwald 2005) for the details.

Fig. 1 Example code table. The widths of the codes represent their lengths. $\mathcal{I} = \{A, B, C\}$. Note that the *usage* column is not part of the code table, but shown here as illustration: for optimal compression, codes should be shorter the more often they are used

Code table *CT*

Itemset	Code	Usage
A B C	□	5
A B	▭	1
A	▭	1
B	▭	1
C	—	0

This is called two-part MDL, or *crude* MDL. As opposed to *refined* MDL, where model and data are encoded together (Grünwald 2007). We use two-part MDL because we are specifically interested in the compressor: the set of frequent itemsets that yields the best compression. Further, although refined MDL has stronger theoretical foundations, it cannot be computed except for some special cases.

To use MDL, we have to define what our models $\mathcal{H}$ are, how a $H \in \mathcal{H}$ describes a database, and how all of this is encoded in bits.

3.2 MDL for itemsets

The key idea of our compression based approach is the *code table*. A code table is a simple two-column translation table that has itemsets on the left-hand side and a code for each itemset on its right-hand side. With such a code table we find, through MDL, the set of itemsets that together optimally describe the data.

Definition 1 Let $\mathcal{I}$ be a set of items and $\mathcal{C}$ a set of code words. A code table *CT* over $\mathcal{I}$ and $\mathcal{C}$ is a two-column table such that:

1. The first column contains itemsets, that is, subsets over $\mathcal{I}$. This column contains at least all singleton itemsets.
2. The second column contains elements from $\mathcal{C}$, such that each element of $\mathcal{C}$ occurs at most once.

An itemset X, drawn from the powerset of $\mathcal{I}$, i.e. $X \in \mathcal{P}(\mathcal{I})$, occurs in *CT*, denoted by $X \in CT$ iff X occurs in the first column of *CT*, similarly for a code $C \in \mathcal{C}$. For $X \in CT$, $code_{CT}(X)$ denotes its *code*, i.e. the corresponding element in the second column. We call the set of itemsets $\{X \in CT\}$ the *coding set*, denoted CS. For the number of itemsets in the code table we write $|CT|$, i.e. we define $|CT| = |\{X \in CT\}|$. Likewise, $|CT \setminus \mathcal{I}|$ indicates the number of non-singleton itemsets in the code table.

Example 1 An example code table is shown in Fig. 1. The first column shows the itemsets in the code table, the second column the corresponding codes. Each bar represents a code, its width represents the code length. The usage column is not actually part of the code table—it is only shown for illustrative purposes and will be explained shortly. Note that $\mathcal{I} = \{A, B, C\}$ and thus that all singleton itemsets are in the code table, even though C has no code associated. $|CT| = 5$, $|CT \setminus \mathcal{I}| = 2$.

To encode a transaction t from database $\mathcal{D}$ over $\mathcal{I}$ with code table *CT*, we require a cover function $cover(CT, t)$ that identifies which elements of *CT* are used to encode
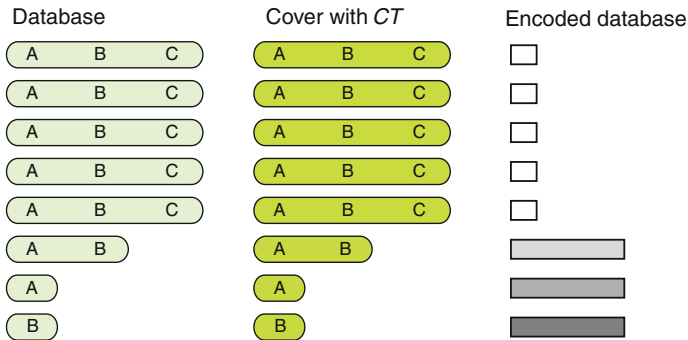
Fig. 2 Example database, and the cover and encoded database obtained by using the code table shown in Fig. 1. $\mathcal{I} = \{A, B, C\}$

t. The parameters are a code table CT and a transaction t, the result is a disjoint set of elements of CT that cover t. Or, more formally, a cover function is defined as follows.

Definition 2 Let $\mathcal{D}$ be a database over a set of items $\mathcal{I}$, t a transaction drawn from $\mathcal{D}$, let $\mathcal{CT}$ be the set of all possible code tables over $\mathcal{I}$, and CT a code table with $CT \in \mathcal{CT}$. Then, $cover : \mathcal{CT} \times \mathcal{P}(\mathcal{I}) \mapsto \mathcal{P}(\mathcal{P}(\mathcal{I}))$ is a *cover function* iff it returns a set of itemsets such that

1. $cover(CT, t)$ is a subset of CS, the coding set of CT, i.e. $X \in cover(CT, t) \rightarrow X \in CT$
2. if $X, Y \in cover(CT, t)$, then either $X = Y$ or $X \cap Y = \emptyset$
3. the union of all $X \in cover(CT, t)$ equals t, i.e.
   $t = \bigcup_{X \in cover(CT, t)} X$

We say that $cover(CT, t)$ covers t. Note that there exists at least one well-defined *cover* function on any code table CT over $\mathcal{I}$ and any transaction $t \in \mathcal{P}(\mathcal{I})$, since CT contains at least the singleton itemsets from $\mathcal{I}$.

By not allowing itemsets in the cover of a transaction to overlap, we ensure that it is always unambiguous what the cover of a transaction is. If we would allow overlap, it can easily happen that multiple covers are possible and computing and testing all of them would be a computational burden.

Example 2 In Fig. 2, an example database is shown. This database will be used as running example from now on. It consists of 8 itemsets, of which five are identical. Also shown is the cover of this database with example code table CT (see Fig. 1). In this example, each transaction is covered by only a single itemset from the code table.

To encode a database $\mathcal{D}$ using code table CT we simply replace each transaction $t \in \mathcal{D}$ by the codes of the itemsets in the cover of t,

$$t \rightarrow \{code_{CT}(X) \mid X \in cover(CT, t)\}.$$

Note that to ensure that we can decode an encoded database uniquely we assume that $\mathcal{C}$ is a *prefix code*, in which no code is the prefix of another code (Cover and Thomas
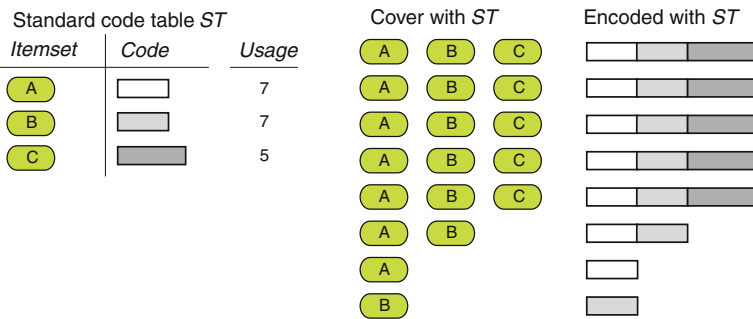
Fig. 3 Example standard code table for the database in Fig. 2, with associated cover and encoded database

2006). (Confusingly, such codes are also known as *prefix-free* codes (Li and Vitányi 1993).)

Example 3 Figure 3 shows how the cover of a database can be translated into an encoded database: replace each itemset in the cover by its associated code.

Since MDL is concerned with the best compression, the codes in *CT* should be chosen such that the most often used code has the shortest length. That is, we should use an optimal prefix code. Note that in MDL we are never interested in materialised codes, but only in the complexities of the model and the data. Therefore, we are only interested in the *lengths* of the codes of itemsets $X \in CT$. As there exists a nice correspondence between code lengths and probability distributions (see e.g. Li and Vitányi 1993), we can calculate the optimal code lengths through the Shannon entropy. So, to determine the complexities we do not have to operate an actual prefix coding scheme such as Shannon-Fano or Huffman encoding.

Theorem 1 *Let P be a distribution on some finite set $\mathcal{D}$, there exists an optimal prefix code C on $\mathcal{D}$ such that the length of the code for $d \in \mathcal{D}$, denoted by $L(d)$ is given by*

$$L(d) = -\log(P(d)).$$

Moreover, this code is optimal in the sense that it gives the smallest expected code size for data sets drawn according to P. (For the proof, please refer Theorem 5.4.1 in Cover and Thomas 2006)

The optimality property means that we introduce no bias using this code length. The probability distribution induced by a cover function is, of course, simply given by the relative usage frequency of each of the item sets in the code table. To determine this, we need to know how often a certain code is used. We define the *usage* count of an itemset $X \in CT$ as the number of transactions t from $\mathcal{D}$ where X is used to cover. Normalised, this frequency represents the probability that that code is used in the encoding of an arbitrary $t \in \mathcal{D}$. The optimal code length (Li and Vitányi 1993) then is $-\log$ of this probability, and a code table is optimal if all its codes have their optimal length. Note that we use fractional lengths, not integer-valued lengths of materialised codes. This ensures that the length of a code accurately represents its usage

probability, and since we are not interested in materialised codes, only relative lengths are of importance. After all, our ultimate goal is to score the optimal code table and not to actually compress the data. More formally, we have the following definition.

Definition 3 Let $\mathcal{D}$ be a transaction database over a set of items $\mathcal{I}$, $\mathcal{C}$ a prefix code, *cover* a cover function, and CT a code table over $\mathcal{I}$ and $\mathcal{C}$. The *usage* count of an itemset $X \in CT$ is defined as

$$usage_{\mathcal{D}}(X) = |\{t \in \mathcal{D} \mid X \in cover(CT, t)\}|.$$

This implies a *probability* distribution of $X \in CT$ for $\mathcal{D}$, given by

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

The $code_{CT}(X)$ for $X \in CT$ is *optimal* for $\mathcal{D}$ iff

$$L(code_{CT}(X)) = |code_{CT}(X)| = -\log(P(X \mid \mathcal{D})).$$

A code table CT is *code-optimal* for $\mathcal{D}$ iff all its codes,

$$\{code_{CT}(X) \mid X \in CT\},$$

are optimal for $\mathcal{D}$.

From now onward we assume that code tables are code-optimal for the database they are induced on, unless we state differently.

Example 4 Figure 1 shows usage counts for all itemsets in the code table. For example, itemset $\{A, B, C\}$ is used five times in the cover of the database. These usage counts are used to compute optimal code lengths. For $X = \{A, B, C\}$:

$$P(X \mid \mathcal{D}) = \tfrac{5}{8}$$
$$L(code_{CT}(X)) = -\log(\tfrac{5}{8}) = 0.68$$

And for $Y = \{A\}$:

$$P(Y \mid \mathcal{D}) = \tfrac{1}{8}$$
$$L(code_{CT}(Y)) = -\log(\tfrac{1}{8}) = 3$$

So, $\{A, B, C\}$ is assigned a code of length 0.68 bits, while $\{A, B\}$, $\{A\}$ and $\{B\}$ are assigned codes of length 3 bits each.

Now, for any database $\mathcal{D}$ and a code table CT over the same set of items $\mathcal{I}$ we can compute $L(\mathcal{D} \mid CT)$. It is simply the summation of the encoded lengths of the transactions. The encoded size of a transaction is simply the sum of the sizes of the codes of the itemsets in its cover. In other words, we have the following trivial lemma.

Lemma 1 *Let $\mathcal{D}$ be a transaction database over $\mathcal{I}$, CT be a code table over $\mathcal{I}$ and code-optimal for $\mathcal{D}$, cover a cover function, and usage the usage function for cover.*

1. For any $t \in \mathcal{D}$ its encoded length, in bits, denoted by $L(t \mid CT)$, is

$$L(t \mid CT) = \sum_{X \in cover(CT, t)} L(code_{CT}(X)).$$

2. The encoded size of $\mathcal{D}$, in bits, when encoded by CT, denoted by $L(\mathcal{D} \mid CT)$, is

$$L(\mathcal{D} \mid CT) = \sum_{t \in \mathcal{D}} L(t \mid CT).$$

With Lemma 1, we can compute $L(\mathcal{D} \mid H)$. To use the MDL principle, we still need to know what $L(H)$ is, i.e. the encoded size of a code table.

Recall that a code table is a two-column table consisting of itemsets and codes. As we know the size of each of the codes, the encoded size of the second column is easily determined: it is simply the sum of the lengths of the codes. For encoding the itemsets, the first column, we have to make a choice.

A naïve option would be to encode each item with a binary integer encoding, that is, using $\log(\mathcal{I})$ bits per item. Clearly, this is hardly optimal; there is no difference in encoded length between highly frequent and infrequent items.

A better choice is to encode the itemsets using the codes of the simplest code table, i.e. the code table that contains only the singleton itemsets $X \in \mathcal{I}$. This code table, with optimal code lengths for database $\mathcal{D}$, is called the *standard code table* for $\mathcal{D}$, denoted by ST. It is the optimal encoding of $\mathcal{D}$ when nothing more is known than just the frequencies of the individual items; it assumes the items to be fully independent. As such, it provides a practical bound: ST provides the simplest, independent, description of the data that compresses much better than a random code table. This encoding allows us to reconstruct the database up to the names of the individual items.

Example 5 Figure 3 shows the standard code table for the database in Fig. 2. It contains all singleton itemsets and usage counts are obtained by covering the database. Code lengths are based on these usages. It is clear that the standard code table does not provide a good compression of the data.

Definition 4 Let $\mathcal{D}$ be a transaction database over $\mathcal{I}$ and CT a code table that is code-optimal for $\mathcal{D}$. The size of CT in bits, denoted by $L(CT \mid \mathcal{D})$, is given by

$$L(CT \mid \mathcal{D}) = \sum_{X \in CT : usage_{\mathcal{D}}(X) \neq 0} L(code_{ST}(X)) + L(code_{CT}(X)).$$

Note that we do not take itemsets with zero usage into account. Such itemsets are not used to code. We use $L(CT)$ wherever $\mathcal{D}$ is clear from context.

With these results we know the total size of our encoded database. It is simply the size of the encoded database plus the size of the code table. That is, we have the following result.

Definition 5 Let $\mathcal{D}$ be a transaction database over $\mathcal{I}$, let CT be a code table that is code-optimal for $\mathcal{D}$ and *cover* a cover function. The *total compressed size* of the encoded database and the code table, in bits, denoted by $L(\mathcal{D}, CT)$ is given by

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

Now that we know how to compute $L(\mathcal{D}, CT)$, we can formalise our problem using MDL. Before that, we discuss three design choices we did not mention so far, because they do not influence the total compressed size of a database.

First, when encoding a database $\mathcal{D}$ with a code table CT, we do not mark the end of a transaction, i.e. we do not use stop-characters. Instead, we assume a given framework that needs to be filled out with the correct items upon decoding. Since such a framework adds the same additive constant to $L(\mathcal{D} \mid CT)$ for any CT over $\mathcal{I}$, it can be disregarded.

Second, for more detailed descriptions of the items in the decoded database, one could add an ASCII table giving the names of the individual items to a code table. Since such a table is the same for all code tables over $\mathcal{I}$, this is again an additive constant we can disregard for our purposes.

Last, since we are only interested in the complexity of the content of the code table, i.e. the itemsets, we disregard the complexity of its structure. That is, like for the database, we assume a static framework that fits any possible code table, consisting of up to $|\mathcal{P}(\mathcal{I})|$ itemsets, and is filled out using the above encoding. The complexity of this framework is equal for any code table CT and dataset $\mathcal{D}$ over $\mathcal{I}$, and therefore we can also disregard this third additive constant when calculating $L(\mathcal{D}, CT)$.

3.3 The problem

Our goal is to find the set of itemsets that best describe the database $\mathcal{D}$. Recall that the set of itemsets of a code table, i.e. $\{X \in CT\}$, is called the coding set CS. Given a coding set, a cover function and a database, a (code-optimal) code table CT follows automatically. Therefore, each coding set has a corresponding code table; we will use this in formalising our problem.

Given a set of itemsets $\mathcal{F}$, the problem is to find a subset of $\mathcal{F}$ which leads to a minimal encoding; where minimal pertains to all possible subsets of $\mathcal{F}$. To make sure this is possible, $\mathcal{F}$ should contain at least the singleton item sets $X \in \mathcal{I}$. We will call such a set, a candidate set. By requiring the smallest coding set, we make sure the coding set contains no unused non-singleton elements, i.e. $usage_{DB}(X) > 0$ for any non-singleton itemset $X \in CT$.

More formally, we define the problem as follows.

Minimal Coding Set Problem

Let $\mathcal{I}$ be a set of items and let $\mathcal{D}$ be a dataset over $\mathcal{I}$, cover a cover function, and $\mathcal{F}$ a candidate set. Find the smallest coding set $CS \subseteq \mathcal{F}$ such that for the corresponding code table CT the total compressed size, $L(\mathcal{D}, CT)$, is minimal.

A solution for the Minimal Coding Set Problem allows us to find the 'best' coding set from a given collection of itemsets, e.g. (closed) frequent itemsets for a given minimal support. If $\mathcal{F} = \{X \in \mathcal{P}(\mathcal{I}) \mid supp_{\mathcal{D}}(X) > 0\}$, i.e. when $\mathcal{F}$ consists of all itemsets that occur in the data, there exists no candidate set $\mathcal{F}'$ that results in a smaller total encoded size. Hence, in this case the solution is truly the minimal coding set for $\mathcal{D}$ and *cover*.

In order to solve the Minimal Coding Set Problem, we have to find the optimal code table and cover function. To this end, we have to consider a humonguous search space, as we will detail in the next subsection.

3.4 How hard is the problem?

The number of coding sets does not depend on the actual database, and nor does the number of possible cover functions. Because of this, we can compute the size of our search space rather easily.

A coding set contains the singleton itemsets plus an almost arbitrary subset of $\mathcal{P}(\mathcal{I})$. Almost, since we are not allowed to choose the $|\mathcal{I}|$ singleton itemsets.

In other words, there are

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

possible coding sets. In order to determine which one of these minimises the total encoded size, we have to consider all corresponding (code-optimal) code tables using every possible cover function. Since every itemset $X \in CT$ can occur only once in the cover of a transaction and no overlap between the itemsets is allowed, this translates to traversing the code table once for every transaction. However, as each possible code table order may result in a different cover, we have to test every possible code table order per transaction to cover. Since a set of n elements admits $n!$ orders, the total size of the search space is as follows.

Lemma 2 *For one transaction over a set of items $\mathcal{I}$, the number of possible ways to cover it is given by*

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

So, even for a rather small set $\mathcal{I}$ and a database of only *one* transaction, the search space we are facing is already huge. Table 1 gives an approximation of the number of cover possibilities for the first few sizes of $\mathcal{I}$. Clearly, the search space is far too large to consider exhaustively.

To make matters worse, there is no useable structure that allows us to prune levelwise as the attained compression is not monotone w.r.t. the addition of itemsets. So, without calculating the *usage* of the itemsets in CT, it is generally impossible to call

Table 1 The number of cover possibilities for a database of one (1) transaction over $\mathcal{I}$

$|\mathcal{I}|$	# Cover possibilities	$|\mathcal{I}|$	# Cover possibilities
1	1	4	2.70×10^{12}
2	8	5	1.90×10^{34}
3	8742	6	4.90×10^{87}

the effects (improvement or degrading) on the compression when an itemset is added to the code table. This can be seen as follows.

Suppose a database $\mathcal{D}$, itemsets X, Y and Z such that $X \cap Y \neq \emptyset$, $X \subset Z$, and a code table CT, all over $\mathcal{I}$. Further, suppose $Y \in CT$. The addition of X to CT can lead to a larger compressed size for two reasons. First and foremost as X may add more complexity to the code table than is compensated for by using X in the encoding of $\mathcal{D}$. Second, because we do not allow overlap in a cover, X may get 'in the way' of itemsets already in CT. Say *cover* prefers using X over Y, then for those transactions $t \in \mathcal{D}$ where we had $Y \in cover(CT, t)$ but also have $X \subseteq t$, then after adding X we will get $X \in cover(CT, t)$ and $Y \notin cover(CT, t)$, as no overlap is allowed. In turn, this reduces $usage(Y)$, which leads to longer codes. Hence, even if X is used a good number of times, by this effect, adding X to CT may still lead to a (much) worse compression.

Alternatively, let us consider adding Z to CT. As Z is a superset of X, it can cover more items with one code. If Z 'gets in the way' of Y like X does above, fewer codes (itemsets) might be necessary to encode the transactions at hand, because Z covers more items. As a result, the total compression might improve; especially if Z is a superset of Y, or when we happen to have a $W \in CT$ that covers (at least) $Y \setminus Z$. As such, depending on the situation, Z may avoid the effect of getting 'in the way' of Y, and thus lead to an improved compression. However, the effect can just as well be negative again, as W may again 'get in the way' of some other sets, etc.

In short, before adding an element to a code table, there is no simple way to predict what the effect will be on the overall compression.

4 Algorithms

In this section we present algorithms for solving the problem formulated in the previous section. As shown above, the search space one needs to consider for finding the optimal code table is far too large to be considered exhaustively. We therefore have to resort to heuristics.

4.1 Basic heuristic

To cut down a large part of the search space, we use the following simple greedy search strategy:

– Start with the standard code table ST, containing only the singleton itemsets $X \in \mathcal{I}$.
– Add the itemsets from $\mathcal{F}$ one by one. If the resulting codes lead to a better compression, keep it. Otherwise, discard the set.

Algorithm 1 The STANDARD CODE TABLE Algorithm

Input: A transaction database $\mathcal{D}$ over a set of items $\mathcal{I}$.
Output: The standard code table CT for $\mathcal{D}$.
1: $CT \leftarrow \emptyset$
2: **for all** $X \in \mathcal{I}$ **do**
3:    insert X into CT
4:    $usage_{\mathcal{D}}(X) \leftarrow supp_{\mathcal{D}}(X)$
5:    $code_{CT}(X) \leftarrow$ optimal code for X
6: **end for**
7: **return** CT

To turn this sketch into an algorithm, some choices have to be made. Firstly, in which order are we going to encode a transaction? So, what cover function are we going to employ? Secondly, in which order do we add the itemsets? Finally, do we *prune* the newly constructed code table before we continue with the next candidate itemset or not?

Before we discuss each of these questions, we briefly describe the initial encoding. This is, of course, the encoding with the standard code table. For this, we need to construct a code table from the elements of $\mathcal{I}$. The algorithm called **Standard Code Table**, given in pseudo-code as Algorithm 1, returns such a code table. It takes a set of items and a database as parameters and returns a code table. Note that for this code table all cover functions reduce to the same, namely the cover function that replaces each item in a transaction with its singleton itemset. As the singleton itemsets are mutually exclusive, all elements $X \in \mathcal{I}$ will be used $supp_{\mathcal{D}}(X)$ times by this cover function.

4.2 Standard cover function

From the problem complexity analysis in the previous section it is quite clear that finding an optimal cover of the database is practically impossible, even if we are given the optimal set of itemsets as the code table: examining all $|CT|!$ possible permutations is already virtually impossible for one transaction, let alone expanding this to all possible combinations of permutations for all transactions.

We therefore employ a heuristic and introduce a standard cover function which considers the code table in a fixed order. The pseudo-code for this **Standard Cover** function is given as Algorithm 2. For a given transaction t, the code table is traversed in a fixed order. An itemset $X \in CT$ is included in the cover of t iff $X \subseteq t$. Then, X is removed from t and the process continues to cover the uncovered remainder, i.e. $t \setminus X$. Using the same order for every transaction drastically reduces the complexity of the problem, but leaves the choice of the order.

Again, considering all possible orders would be best, but is impractical at best. A more prosaic reason is that our algorithm will need a definite order; random choice does not seem the wisest of ideas. When choosing an order, we should take into account that the order in which we consider the itemsets may make it easier or more difficult to insert candidate itemsets into an already sorted code table.

Algorithm 2 The STANDARD COVER Algorithm

Input: Transaction $t \in \mathcal{D}$ and code table CT, with CT and $\mathcal{D}$ over a set of items $\mathcal{I}$.
Output: A cover of t using non-overlapping elements of CT.
1: $S \leftarrow$ smallest element X of CT in **Standard Cover Order** for which $X \subseteq t$
2: **if** $t \setminus S = \emptyset$ **then**
3:     $Res \leftarrow \{S\}$
4: **else**
5:     $Res \leftarrow \{S\} \cup$ STANDARDCOVER$(t \setminus S, CT)$
6: **end if**
7: **return** Res

We choose to sort the elements $X \in CT$ first decreasing on cardinality, second decreasing on support in $\mathcal{D}$ and thirdly lexicographically increasing to make it a total order. To describe the order compactly, we introduce the following notation. We use ↓ to indicate that an attribute is sorted descending, and ↑ to indicate it is sorted ascending:

$$|X| \downarrow \quad supp_{\mathcal{D}}(X) \downarrow \quad \text{lexicographically} \uparrow$$

We call this the **Standard Cover Order**. The rationale is as follows. To reach a good compression we need to replace as many individual items as possible, by as few and short as possible codes. The above order gives priority to long itemsets, as these can replace as many as possible items by just one code. Further, we prefer those itemsets that occur frequently in the database to be used as often as possible, resulting in high *usage* values and thus short codes. We rely on MDL not to select overly specific itemsets, as such sets can only be infrequently used and would thus receive relatively long codes.

4.3 Standard candidate order

Next, we address the order in which candidate itemsets will be regarded. Preferably, the candidate order should be in concord with the cover strategy detailed above. We therefore choose to sort the candidate itemsets such that long, frequently occurring itemsets are given priority. Again, to make it a total order we thirdly sort lexicographically. So, we sort the elements of $\mathcal{F}$ as follows:

$$supp_{\mathcal{D}}(X) \downarrow \quad |X| \downarrow \quad \text{lexicographically} \uparrow$$

We refer to this as the **Standard Candidate Order**. The rationale for it is as follows. Itemsets with the highest support, those with potentially the shortest codes, end up at the top of the list. Of those, we prefer the longest sets first, as these will be able to replace as many items as possible. This provides the search strategy with the most general itemsets first, providing ever more specific itemsets along the way.

A welcome advantage of the standard orders for both the cover function and the candidate order is that we can easily keep the code table sorted. First, the length of an itemset is readily available. Second, with this candidate order we know that any candidate itemset for a particular length will have to be inserted after any already

present code table element with the same length. Together, this means that we can insert a candidate itemset at the right position in the code table in O(1) if we store the code table elements in an array (over itemset length) of lists.

4.4 The KRIMP algorithm

We now have the ingredients for the basic version of our compression algorithm:

- Start with the standard code table ST;
- Add the candidate itemsets from $\mathcal{F}$ one by one. Each time, take the itemset that is maximal w.r.t. the standard candidate order. Cover the database using the standard cover algorithm. If the resulting encoding provides a smaller compressed size, keep it. Otherwise, discard it permanently.

This basic scheme is formalised as the KRIMP algorithm given as Algorithm 3. For the choice of the name: 'krimp' is Dutch for 'to shrink'. The KRIMP pattern selection process is illustrated in Fig. 4.

KRIMP takes as input a database $\mathcal{D}$ and a candidate set $\mathcal{F}$. The result is the best code table the algorithm has seen, w.r.t. the Minimal Coding Set Problem.

Now, it may seem that each iteration of KRIMP can only lessen the *usage* of an itemset in CT. For, if $F_1 \cap F_2 \neq \emptyset$ and F_2 is used before F_1 by the standard cover function, the usage of F_1 will go down (provided the support of F_2 does not equal

---

Algorithm 3 The KRIMP Algorithm

Input: A transaction database $\mathcal{D}$ and a candidate set $\mathcal{F}$, both over a set of items $\mathcal{I}$
Output: A heuristic solution to the Minimal Coding Set Problem, code table CT
1: $CT \leftarrow$ **Standard Code Table**$(\mathcal{D})$
2: $\mathcal{F}_o \leftarrow \mathcal{F}$ in **Standard Candidate Order**
3: **for all** $F \in \mathcal{F}_o \setminus \mathcal{I}$ **do**
4:     $CT_c \leftarrow (CT \cup F)$ in **Standard Cover Order**
5:     **if** $L(\mathcal{D}, CT_c) < L(\mathcal{D}, CT)$ **then**
6:        $CT \leftarrow CT_c$
7:     **end if**
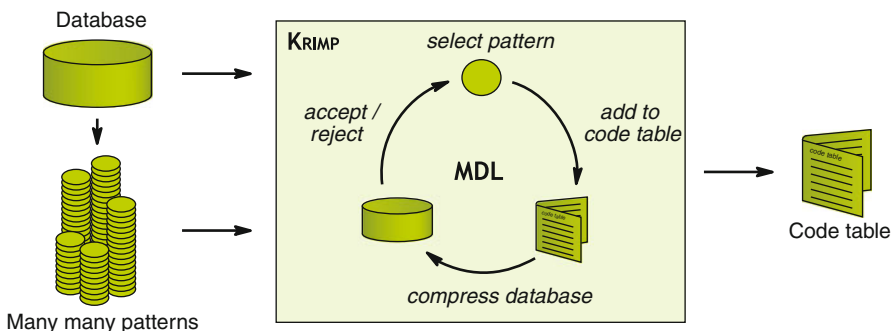8: **end for**
9: **return** CT

Fig. 4 KRIMP in action

zero). While this is true, it is not the whole story. Because, what happens if we now add an itemset F_3, which is used before F_2 such that:

$$F_1 \cap F_3 = \emptyset \quad \text{and} \quad F_2 \cap F_3 \neq \emptyset$$

The usage of F_2 will go down, while the usage of F_1 will go up again; by the same amount, actually. So, taking this into consideration, even code table elements with zero usage cannot be removed without consequence. However, since they are not used in the actual encoding, they are not taken into account while calculating the total compressed size for the current solution.

In the end, itemsets with zero usage can be safely removed though. After all, they do not code, so they are not part of the optimal answer that should consist of the smallest coding set. Since the singletons are required in a code table by definition, these remain.

4.5 Pruning

That said, we cannot be sure that leaving itemsets with a very low usage count in CT is the best way to go. As these have a very small probability, their respective codes will be very long. Such long codes may make better code tables unreachable for the greedy algorithm; it may get stuck in a local optimum. As an example, consider the following three code tables:

$$CT_1 = \{\{X_1, X_2\}, \{X_1\}, \{X_2\}, \{X_3\}\}$$
$$CT_2 = \{\{X_1, X_2, X_3\}, \{X_1, X_2\}, \{X_1\}, \{X_2\}, \{X_3\}\}$$
$$CT_3 = \{\{X_1, X_2, X_3\}, \{X_1\}, \{X_2\}, \{X_3\}\}$$

Assume that $supp_{\mathcal{D}}(\{X_1, X_2, X_3\}) = supp_{\mathcal{D}}(\{X_1, X_2\}) - 1$. Given these assumptions, standard KRIMP will never consider CT_3, but it is very well possible that $L(\mathcal{D}, CT_3) < L(\mathcal{D}, CT_2)$ and that CT_3 provides access to a branch of the search space that is otherwise left unvisited. To allow for searching in this direction, we can *prune* the code table that KRIMP is considering.

There are many possibilities to this end. The most obvious strategy is to check the attained compression of all valid subsets of CT including the candidate itemset F, i.e. $\{CT_p \subseteq CT \mid F \in CT_p \wedge \mathcal{I} \subset CT_p\}$, and choose CT_p with minimal $L(\mathcal{D}, CT_p)$. In other words, prune when a candidate itemset is added to CT, but before the acceptance decision. Clearly, such a pre-acceptance pruning approach implies a huge amount of extra computation. Since we are after a fast and well-performing heuristic we do not consider this strategy.

A more efficient alternative is post-acceptance pruning. That is, we only prune when F is accepted: when candidate code table $CT_c = CT \cup F$ is better than CT, i.e. $L(\mathcal{D}, CT_c) < L(\mathcal{D}, CT)$, we consider its valid subsets. This effectively reduces the pruning search space, as only few candidate itemsets will be accepted.

To cut the pruning search space further, we do not consider all valid subsets of CT, but iteratively consider for removal those itemsets $X \in CT$ for which

---

Algorithm 4 Code Table Post-Acceptance Pruning

Input: Code tables CT_c and CT, for a transaction database $\mathcal{D}$ over a set of items $\mathcal{I}$, where $\{X \in CT\} \subset$
  $\{Y \in CT_c\}$ and $L(\mathcal{D}, CT_c) < L(\mathcal{D}, CT)$.
Output: Pruned code table CT_p, such that $L(\mathcal{D}, CT_p) \leq L(\mathcal{D}, CT_c)$ and $CT_p \subseteq CT_c$.
1: $PruneSet \leftarrow \{X \in CT \mid usage_{CT_c}(X) < usage_{CT}(X)\}$
2: **while** $PruneSet \neq \emptyset$ **do**
3:    $PruneCand \leftarrow X \in PruneSet$ with lowest $usage_{CT_c}(X)$
4:    $PruneSet \leftarrow PruneSet \setminus PruneCand$
5:    $CT_p \leftarrow CT_c \setminus PruneCand$
6:    **if** $L(\mathcal{D}, CT_p) < L(\mathcal{D}, CT_c)$ **then**
7:      $PruneSet \leftarrow PruneSet \cup \{X \in CT_p \mid usage_{CT_p}(X) < usage_{CT_c}(X)\}$
8:      $CT_c \leftarrow CT_p$
9:    **end if**
10: **end while**
11: **return** CT_c

$usage_{\mathcal{D}}(X)$ has decreased. The rationale is that for these itemsets we know that their code lengths have increased; therefore, it is possible that these sets now harm the compression.

In line with the standard order philosophy, we first consider the itemset with the smallest usage and thus the longest code. If by pruning an itemset the total encoded size decreases, we permanently remove it from the code table. Further, we then update the list of prune candidates with those item sets whose usage consequently decreased. This post-acceptance pruning strategy is formalised in Algorithm 4. We refer to the version of KRIMP that employs this pruning strategy (which would be on line 6 of Algorithm 3) as KRIMP with pruning. In Sect. 7 we will show that employing pruning improves the performance of KRIMP.

Example 6 Two example code tables obtained with KRIMP are shown in Fig. 5. Without pruning, KRIMP selects 2 itemsets of length > 1 to encode the database. The size of the database encoded with this code table is 12.4 bits. The total encoded size, including the size of the code table, is 33 bits. When pruning is enabled, $\{A, B\}$ is pruned from the code table. It was accepted into the code table earlier than $\{A, B, C\}$, but does no longer contribute to compression now that this larger set has been added. With the pruned code table, the size of the encoded database is 12.9 bits. However, since the code table is smaller, the total encoded size becomes 26 bits; 7 bits smaller than without pruning.

4.6 Complexity

Here we analyse the complexity of the KRIMP algorithms step–by–step. We start with time-complexity, after which we cover memory-complexity.

Given a set of (frequent) itemsets $\mathcal{F}$, we first order this set, requiring $O(|\mathcal{F}| \log |\mathcal{F}|)$ time. Then, every element $F \in \mathcal{F}$ is considered once. Using a hash-table implementation we need only $O(1)$ to insert an element at the right position in CT, keeping CT ordered. To calculate the total encoded size $L(\mathcal{D}, CT)$, the *cover* function is applied to each $t \in \mathcal{D}$. For this, the standard cover function considers each $X \in CT$ once for a

Fig. 5 Example code tables obtained with KRIMP, with and without pruning, on the database shown in Fig. 2. Also shown are the associated covers and encoded databases

t. Checking whether X is an (uncovered) subset of t takes at most $O(|\mathcal{I}|)$. Therefore, covering the full database takes $O(|\mathcal{D}| \times |CT| \times |\mathcal{I}|)$ time. Then, optimal code lengths and the total compressed size can be computed in $O(|CT|)$.

Note that we know the code table will grow to at most $|\mathcal{F}|$ elements. So, given a set of (frequent) itemsets $\mathcal{F}$ and a *cover* function that considers the elements of the code table in a static order, the worst-case time-complexity of the KRIMP algorithm without pruning is

$$O(\, |\mathcal{F}| \log |\mathcal{F}| + |\mathcal{F}| \times (|\mathcal{D}| \times |\mathcal{F}| \times |\mathcal{I}| + |\mathcal{F}|) \,).$$

When we do employ pruning, in the worst-case we have to reconsider each element in CT after accepting each $F \in \mathcal{F}$,

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

This looks quite horrendous. However, it is not as bad as it seems.

First of all, due to MDL, the number of elements in the code table is very small, $|CT| \ll |\mathcal{D}| \ll |\mathcal{F}|$, in particular when pruning is enabled. In fact, this number (typically 100 to 1000) can be regarded as a constant, removing it from the big-O notation. Therefore,

$$O(\, |\mathcal{F}| \log |\mathcal{F}| + |\mathcal{D}| \times |\mathcal{F}| \times |\mathcal{I}| \,)$$

is a better estimate for the time-complexity for KRIMP with or without pruning enabled.

Next, for $\mathcal{I}$ of reasonable size (say, up to 1000), bitmaps can be used to represent the itemsets. This allows for subset checking in O(1), again removing a term from the complexity. Further, for any new candidate code table itemset $F \in \mathcal{F}$, the database needs only to be covered partially; so instead of all $|\mathcal{D}|$ transactions only those d transactions in which F occurs need to be covered. If $\mathcal{D}$ is large and the *minsup* threshold is low, d is generally very small ($d \ll |\mathcal{D}|$) and can be regarded as a constant. So, in the end we have

$$O(|\mathcal{F}| \log |\mathcal{F}| + |\mathcal{F}|).$$

Now, we consider the order of the memory requirements of KRIMP. The worst-case memory requirements of the KRIMP algorithms are

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

Again, as the code table is dwarfed by the size of the database, it can be regarded a (small) constant. The major part is the storage of the candidate code table elements. Sorting these can be done in place. As it is iterated in order, it can be handled from the hard drive without much performance loss. Preferably, the database is kept resident, as it is covered many times.

5 Interlude

Before we continue with more theory, we will first present some results on a small number of datasets to provide the reader with some measure and intuition on the performance of KRIMP. To this end, we ran KRIMP with post-acceptance pruning on six datasets, using all frequent itemsets mined at *minsup* $= 1$ as candidates. The results of these experiments are shown in Table 2. Per dataset, we show the number of transactions and the number of candidate itemsets. From these latter figures, the problem of the pattern explosion becomes clear: up to 5.5 billion itemsets can be mined from the *Mushroom* database, which consists of only 8124 transactions. It also shows that KRIMP successfully battles this explosion, by selecting only hundreds of itemsets from millions up to billions. For example, from the 5.5 billion for *Mushroom*, only 442 itemsets are chosen; a reduction of seven orders of magnitude.

For the other datasets, we observe the same trend. In each case, fewer than 2000 itemsets are selected, and reductions of many orders of magnitude are attained. The number of selected itemsets depends mainly on the characteristics of the data. These itemsets, or the code tables they form, compress the data to a fraction of its original size. This indicates that very characteristic itemsets are chosen, and that the selections are non-redundant. Further, the timings for these experiments show that the compression-based selection process, although computationally complex, is a viable approach in practice. The selection of the above mentioned 442 itemsets from 5.5 billion itemsets takes under 4 h. For the *Adult* database, KRIMP considers over 400,000 itemsets per second, and is limited not by the CPUs but by the rate with which the itemsets can be read from the harddisk.

Table 2 Results of running KRIMP on a few datasets

Dataset	$|\mathcal{D}|$	$|\mathcal{I}|$	$|\mathcal{F}|$	KRIMP		Time
				$|CT \setminus \mathcal{I}|$	$\frac{L(\mathcal{D},CT)}{L(\mathcal{D},ST)}\%$	
Adult	48842	97	58461763	1303	24.4	0:02:25
Chess (kr–k)	28056	58	373421	1684	61.6	0:00:13
Led7	3200	24	15250	152	28.6	0:00:00
Letter recognition	20000	102	580968767	1780	35.7	0:52:33
Mushroom	8124	119	5574930437	442	20.6	3:40:25
Pen digits	10992	86	459191636	1247	42.3	0:31:33

For all datasets the candidate set $\mathcal{F}$ was mined with *minsup* $= 1$, and KRIMP with post-acceptance pruning was used. For KRIMP, the size of the resulting code table (minus the singletons), the compression ratio and the runtime is given. The compression ratio is the encoded size of the database with the obtained code table divided by the baseline encoded size, in percentages. Timings, in hours, minutes and seconds (h: mm:ss), as recorded with the parallel implementation on quad-core 3.0 Ghz Xeon machines using four threads.

Given this small sample of results, we now know that indeed few, characteristic and non-redundant itemsets are selected by KRIMP, in number many orders smaller than the complete frequent itemset collections. However, this leaves the question of how *good* are the returned pattern sets?

6 Classification by compression

In this section, we describe a method to verify the quality of the KRIMP selection in an independent way. To be more precise, we introduce a simple classification scheme based on code tables, previously published as van Leeuwen et al. (2006). We answer the quality question by answering the question: how well do KRIMP code tables classify? For this, classification performance is compared to that of state-of-the-art classifiers in Sect. 7.6.

6.1 Classification through MDL

If we assume that our database of transactions is an i.i.d. sample from some underlying data distribution, we expect the optimal code table for this database to compress an arbitrary transaction sampled from this distribution well. We make this intuition more formal in Lemma 3.

We say that the itemsets in *CT* are *independent* if any co-occurrence of two itemsets $X, Y \in CT$ in the cover of a transaction is independent. That is, $P(XY) = P(X)P(Y)$. Clearly, this is a Naïve Bayes (Warner et al. 1961) like assumption.

Lemma 3 *Let $\mathcal{D}$ be a bag of transactions over $\mathcal{I}$, cover a cover function, CT the optimal code table for $\mathcal{D}$ and t an arbitrary transaction over $\mathcal{I}$. Then, if the itemsets*

$X \in cover(CT, t)$ *are independent,*

$$L(t \mid CT) = -\log\left(P(t \mid \mathcal{D})\right).$$

Proof

$$
\begin{aligned}
L(t \mid CT) &= \sum_{X \in cover(CT,t)} L(code_{CT}(X)) \\
&= \sum_{X \in cover(CT,t)} -\log\left(P(X \mid \mathcal{D})\right) \\
&= -\log\left(\prod_{X \in cover(CT,t)} P(X \mid \mathcal{D})\right) \\
&= -\log\left(P(t \mid \mathcal{D})\right).
\end{aligned}
$$

$\square$

The last equation is only valid under the Naïve Bayes like assumption, which might be violated. However, if there are itemsets $X, Y \in CT$ such that $P(XY) > P(X)P(Y)$, we would expect an itemset $Z \in CT$ such that $X, Y \subset Z$. Therefore, we do not expect this assumption to be overly optimistic.

Now, assume that we have two databases generated from two different underlying distributions, with corresponding optimal code tables. For a new transaction that is generated under one of the two distributions, we can now decide to which distribution it most likely belongs. That is, under the Naïve Bayes assumption, we have the following lemma.

Lemma 4 *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two bags of transactions over $\mathcal{I}$, sampled from two different distributions, cover a cover function, and t an arbitrary transaction over $\mathcal{I}$. Let CT_1 and CT_2 be the optimal code tables for respectively $\mathcal{D}_1$ and $\mathcal{D}_2$. Then, from Lemma 3 it follows that*

$$L(t \mid CT_1) > L(t \mid CT_2) \ \Rightarrow\ P(t \mid \mathcal{D}_1) < P(t \mid \mathcal{D}_2).$$

Hence, the Bayes optimal choice is to assign t to the distribution that leads to the shortest code length.

6.2 The Krimp classifier

The previous subsection, with Lemma 4 in particular, suggests a straightforward classification algorithm based on KRIMP code tables. This provides an independent way to assess the quality of the resulting code tables. The KRIMP Classifier is given in Algorithm 5. The KRIMP classification process is illustrated in Fig. 6.

The classifier consists of a code table per class. To build it, a database with class labels is needed. This database is split according to class, after which the class labels are

Algorithm 5 The KRIMP Classifier

Input: A database $\mathcal{D}$ with class labels and a transaction t, both over a set of items $\mathcal{I}$
Output: The class label assigned to t
1: $K \leftarrow \{$ class labels of $\mathcal{D}\}$
2: $\{\mathcal{D}_k\} \leftarrow$ split $\mathcal{D}$ on K, remove each $k \in K$ from each $t \in \mathcal{D}$
3: **for all** $\mathcal{D}_k$ **do**
4:   $\mathcal{F}_k \leftarrow$ MineCandidates($\mathcal{D}_k$)
5:   $CT_k \leftarrow$ KRIMP ($\mathcal{D}_k, \mathcal{F}_k$)
6:   **for each** $X \in CT_k : usage_{CT_k}(X) \leftarrow usage_{CT_k}(X) + 1$
7: **end for**
8: **return** arg $\min_{k \in K} L(t \mid CT_k)$
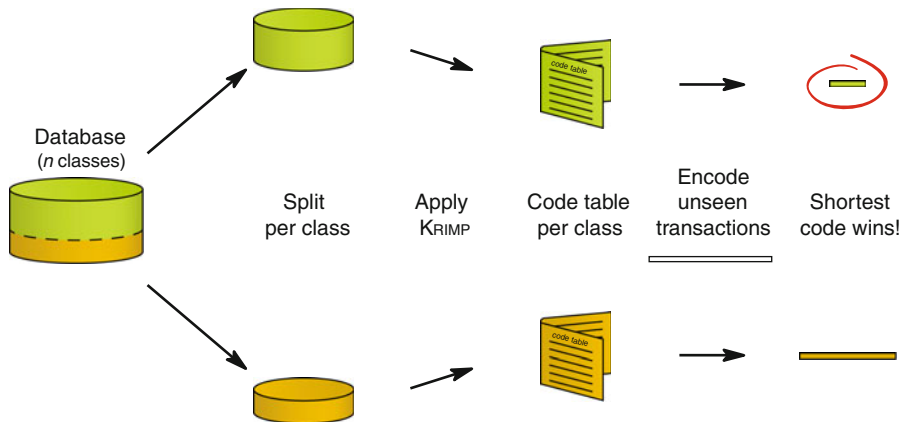
Fig. 6 The KRIMP Classifier in action

removed from all transactions. KRIMP is applied to each of the single-class databases, resulting in a code table per class. At the very end, after all pruning has been done, each code table is Laplace corrected: the *usage* of each itemset in CT_k is increased by one. This ensures that all itemsets in CT_k have non-zero usage, therefore have a code, i.e. their code length can be calculated, and thus, that any arbitrary transaction $t \subseteq \mathcal{I}$ can be encoded. (Recall that we require a code table to always contain all singleton itemsets.)

When the compressors have been constructed, classifying a transaction is trivial. Simply assign the class label belonging to the code table that provides the minimal encoded length for the transaction.

7 Experiments

In this section we experimentally evaluate the KRIMP algorithms and the underlying heuristics, and assess the quality of the resulting code tables.

We describe our setup in Sect. 7.1 and the datasets we use in the experiments in Sect. 7.2. Then, we start our evaluation of KRIMP by looking at how many itemsets are selected and what compression ratios are attained in Sect. 7.3. The stability of these results, and whether these rely on specific itemsets is explored in Sect. 7.4. In

Sect. 7.5 we test through swap-randomisation whether the code tables model relevant structure. The quality of the code tables is independently validated through classification in Sect. 7.6. Last, in Sect. 7.7, we evaluate the cover and candidate order heuristics of KRIMP.

7.1 Setup

We use the shorthand notation $L\%$ to denote the relative total compressed size of $\mathcal{D}$, so

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

wherever $\mathcal{D}$ is clear from context. Since $0 < L(\mathcal{D}, CT) \leq L(\mathcal{D}, ST)$, $L\%$ is a value between 0 and 100. As candidates, $\mathcal{F}$, we typically use all frequent itemsets mined at $minsup = 1$, unless indicated otherwise. We use the AFOPT miner (Liu et al. 2004), taken from the FIMI repository (Goethals and Zaki 2003), to mine (closed) frequent itemsets. The reported KRIMP timings are of the selection process only and do not include the mining and sorting of the candidate itemset collections. All experiments were conducted on quad-core Xeon 3.0 GHz systems running Windows Server 2003. Timings reported in this section are based on single-threaded runs.

7.2 Data

For the experimental validation of our methods we use a wide range of freely available datasets. From the LUCS/KDD discretised data set repository (Coenen 2003) we take some of the largest databases. We transformed the *Connect-4* dataset to a less dense format by removing all 'empty-field' items. From the FIMI repository (Goethals and Zaki 2003) we use the BMS datasets[2] (Kohavi et al. 2000). Further, we use the *Mammals* presence and *DNA Amplification* databases. The former consists of presence records of European mammals[3] within geographical areas of 50×50 kilometers (Mitchell-Jones et al. 1999). The latter is data on DNA copy number amplifications. Such copies activate oncogenes and are hallmarks of nearly all advanced tumors (Myllykangas et al. 2006). Amplified genes represent attractive targets for therapy, diagnostics and prognostics.

The details for these datasets are depicted in Table 3. For each database we show the number of attributes, the number of transactions and the density: the percentage of 'present' attributes. Last, we provide the total compressed size in bits as encoded by the singleton-only standard code tables ST.

---

[2] We wish to thank Blue Martini Software for contributing the KDD Cup 2000 data.

[3] The full version of the mammal dataset is available for research purposes upon request from the Societas Europaea Mammalogica. http://www.european-mammals.org

Table 3 Statistics of the datasets used in the experiments

Dataset	$|\mathcal{D}|$	$|\mathcal{I}|$	Density	# Classes	$L(\mathcal{D} \mid ST)$
Accidents	340183	468	7.22	–	74592568
Adult	48842	97	15.33	2	3569724
Anneal	898	71	20.15	5	62827
BMS-pos	515597	1657	0.39	–	25321966
BMS-webview 1	59602	497	0.51	–	1173962
BMS-webview 2	77512	3340	0.14	–	3747293
Breast	699	16	62.36	2	27112
Chess (k–k)	3196	75	49.33	2	687120
Chess (kr–k)	28056	58	12.07	18	1083046
Connect–4	67557	129	33.33	3	17774814
DNA amplification	4590	392	1.47	–	212640
Heart	303	50	27.96	5	20543
Ionosphere	351	157	22.29	2	81630
Iris	150	19	26.32	3	3058
Led7	3200	24	33.33	10	107091
Letter recognition	20000	102	16.67	26	1980244
Mammals	2183	121	20.5	–	320094
Mushroom	8124	119	19.33	2	1111287
Nursery	12960	32	28.13	5	569042
Page blocks	5473	44	25	5	216552
Pen digits	10992	86	19.77	10	1140795
Pima	768	38	23.68	2	26250
Pumsbstar	49046	2088	2.42	–	19209514
Retail	88162	16470	0.06	–	10237244
Tic–tac–toe	958	29	34.48	2	45977
Waveform	5000	101	21.78	3	656084
Wine	178	68	20.59	3	14101

Per dataset the number of transactions, the number of attributes, the density (average percentage of items) and the number of bits required by KRIMP to compress the data using the singleton-only standard code table *ST*

7.3 Selection

We first evaluate the question whether KRIMP provides an answer to the pattern explosion. To this end, we ran KRIMP on 27 datasets, and analysed the outcome code tables, with and without post-acceptance pruning. The results of these experiments are shown as Table 4. As candidates itemset collections we mined frequent itemsets of the indicated *minsup* thresholds. These were chosen as low as possible, either storage-wise or computationally feasible.

The main result shown in the table is the reduction attained by the selection process: up to seven orders of magnitude. While the candidate sets contain millions up to billions of itemsets, the resulting code tables typically contain hundreds to thousands

Table 4 Results of KRIMP with and without post-acceptance pruning

Dataset	minsup	$|\mathcal{F}|$	KRIMP w/o pruning		KRIMP with pruning	
			$|CT \setminus \mathcal{I}|$	L%	$|CT \setminus \mathcal{I}|$	L%
Accidents	50000	2881487	4046	55.4	467	55.1
Adult	1	58461763	1914	24.9	1303	24.4
Anneal	1	4223999	133	37.5	102	35.3
BMS-pos	100	5711447	14628	82.7	1657	81.8
BMS-wv1	32	1531980297	960	86.6	736	86.2
BMS-wv2	10	4440334	5475	84.4	4585	84.0
Breast	1	9919	35	17.4	30	17.0
Chess (k–k)	319	4603732933	691	30.9	280	27.3
Chess (kr–k)	1	373421	2203	62.9	1684	61.6
Connect–4	1	233142539	4525	11.5	2036	10.9
DNA amp	9	312073710	417	38.6	326	37.9
Heart	1	1922983	108	61.4	79	57.7
Ionosphere	35	225577741	235	63.4	164	61.3
Iris	1	543	13	48.2	13	48.2
Led7	1	15250	194	29.5	152	28.6
Letter	1	580968767	3758	43.3	1780	35.7
Mammals	200	93808243	597	50.4	316	48.4
Mushroom	1	5574930437	689	22.2	442	20.6
Nursery	1	307591	356	45.9	260	45.5
Page blocks	1	63599	56	5.1	53	5.0
Pen digits	1	459191636	2794	48.8	1247	42.3
Pima	1	28845	72	36.3	58	34.8
Pumsbstar	11120	272580786	734	51.0	389	50.9
Retail	4	4106008	7786	98.1	6264	97.7
Tic–tac–toe	1	250985	232	65.0	160	62.8
Waveform	5	465620240	1820	55.6	921	44.7
Wine	1	2276446	76	80.9	63	77.4

Per dataset, the *minsup* for mining frequent itemsets, and the size of the resulting candidate set $\mathcal{F}$. For KRIMP without and with post-acceptance pruning enabled, the number of non-singleton elements in the returned code tables and the attained compression ratios.

of non-singleton itemsets. These selected itemsets compress the data well, typically requiring only a quarter to half of the bits of the independent *ST* encoding. Dense datasets are compressed very well. For *Adult* and *Mushroom*, ratios of respectively 24 and 21% are noted. Sparse data, on the other hand, typically contains little structure. We see that such datasets (e.g. the *Retail* and *BMS* datasets) indeed prove difficult to compress; relatively many itemsets are required to provide a meagre compression.

Comparing between KRIMP with and without post-acceptance pruning, we see that enabling pruning provides the best: fewer itemsets (∼ 1000, on average) are returned, which provide better compression (avg. 2% improvement). For *Accidents* and

BMS-pos the difference in the number of selected itemsets is a factor of 10. The average length of the itemsets in the code tables is about the same, with respectively 5.9 and 5.7 with and without pruning. However, the average of the *usage* of these itemsets differs more, with averages of respectively 80.7 and 48.2.

As post-acceptance pruning provides improved performance, from now onward we employ KRIMP with post-acceptance pruning, unless indicated otherwise. Further, due to the differences in code table size, experiments with pruning typically execute faster than those without pruning.

Next, we examine the development in number of selected itemsets w.r.t. the number of candidate itemsets. For the *Mushroom* database, the top graph of Fig. 7 shows the size of the candidate set and size of the corresponding code table for varying *minsup* thresholds. While we see that the number of candidate itemsets grows exponentially, to 5.5 billion for *minsup* = 1, the number of selected itemsets stabilises at around 400. This stabilisation is typical for all datasets, with the actual number being dependent on the characteristics of the data.

This raises the question whether the total compressed size also stabilises. In the bottom graph of Fig. 7, we plot the total compressed size of the database for the same range of *minsup*. From the graph it is clear that this is not the case: the compressed size decreases continuously, it does not stabilise. Again, this is typical behaviour; especially for sparse data we have recorded steep descents at low *minsup* values. As the number of itemsets in the code table is stable, we thus know that itemsets are being replaced by better ones. Further, note that the compressed size of the code table is dwarfed by the compressed size of the database. This is especially the case for large datasets.

In Fig. 8 we display histograms of the support of the selected itemsets. These plots shows that these itemsets are typically frequent. That is, KRIMP does not just choose itemsets of low-support, but rather selects a range of specific and more general itemsets. The top graph, for the *Accidents* database, shows that the itemsets with supports close to the chosen *minsup* of 50000 are favored, which is in line with what we saw in Fig. 7: compression increases if itemsets up till lower *minsup* are considered. However, note the long tail of the graph, for also many itemsets of higher support are included in the code table. We see the same behaviour when we set *minsup* to 1, like in the bottom graph for the much smaller *Tic–Tac–Toe* database.

Next, in Fig. 9, we regard the relation between the support and the length of the used itemsets. We see that KRIMP selects a broad spectrum of itemsets, and does not simply focus on short and frequent or long and infrequent. We recorded highly similar graphs for the other datasets.

Back to the top graph of Fig. 7, we see a linear correlation between the runtime and the number of candidate sets. The correlation factor depends heavily on the data characteristics; its density, the number of items and the number of transactions. For this experiment, we observed 150,000 to 350,000 candidates considered per second, the performance being limited by IO.

In the top graph of Fig. 10 we provide an overview of the differences in the sizes of the candidate sets and code tables, and in the bottom graph the runtimes recorded for these experiments. Those experiments for which the runtime bars are missing finished within one second. The bottom graph shows that the runtimes are low. Letting
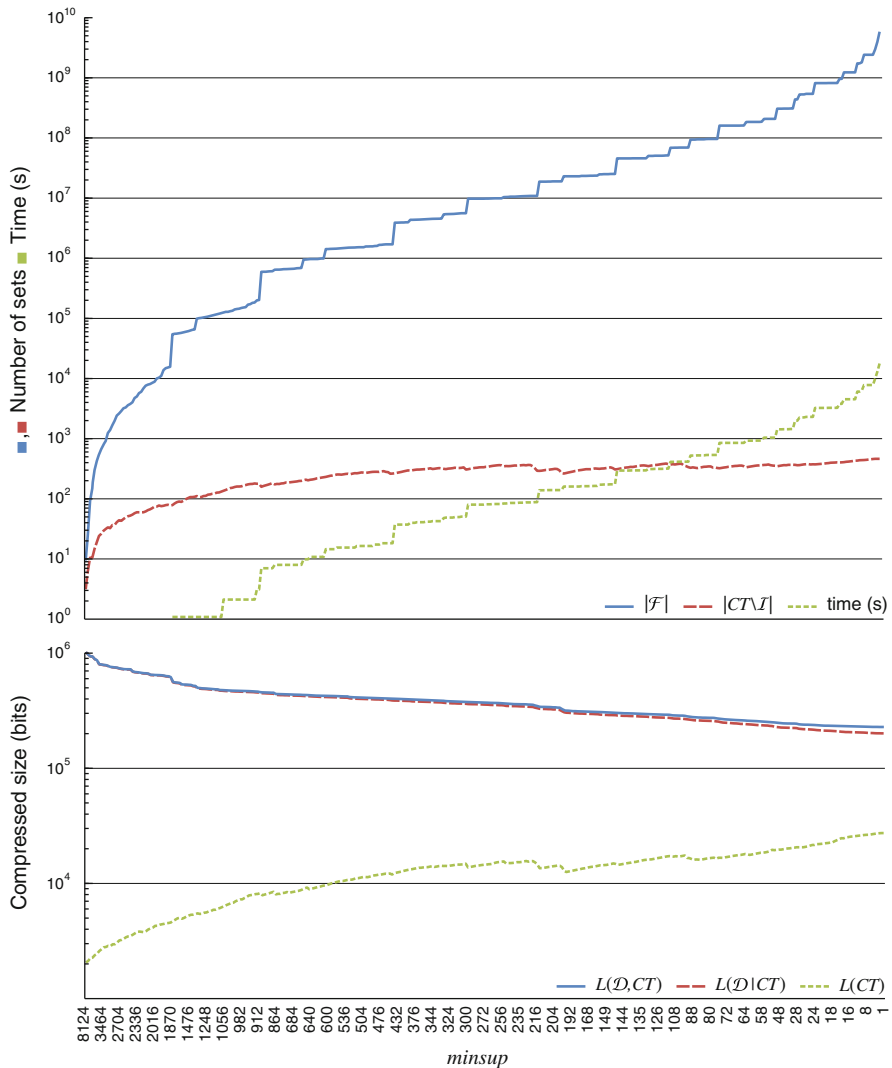
Fig. 7 Running KRIMP with post-acceptance pruning on the *Mushroom* dataset, using 5.5 billion frequent itemsets as candidates (*minsup* = 1). (*top*) Per *minsup*, the size of the candidate set, $|\mathcal{F}|$, the size of the code table, $|CT \setminus \mathcal{I}|$, and the runtime in seconds in respectively the continuous, the long-dashed and short-dashed lines. (*bottom*) Per *minsup*, in bits the size of the code table, the data, and the total compressed size (respectively $L(CT)$, $L(\mathcal{D} \mid CT)$ and $L(\mathcal{D}, CT)$) in respectively the short-dashed, long-dashed and continuous lines

KRIMP consider the largest candidate sets, billions of itemsets, takes up to a couple of hours. The actual speed (candidates per second) mainly depends on the support of the itemsets (the number of transactions that have to be covered). The speed at which our current implementation considers candidate itemsets typically ramps up to thousands, even hundreds of thousands, per second.
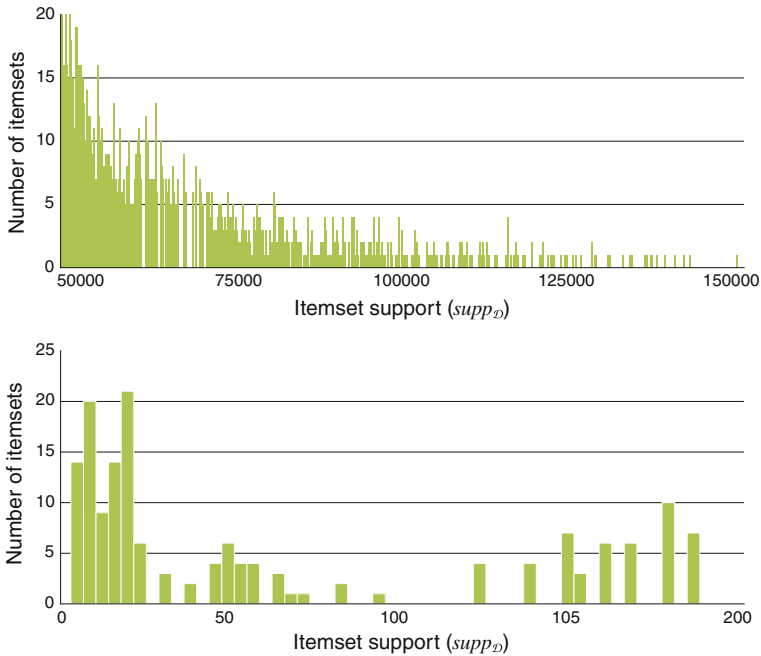
Fig. 8 Histograms of the support, $supp_{\mathcal{D}}$, of the non-singleton itemsets in the code tables of the experiments on the *Accidents* (*top*) and *Tic–Tac–Toe* (*bottom*) datasets with, respectively, *minsup* thresholds of 50000 and 1, employing online pruning
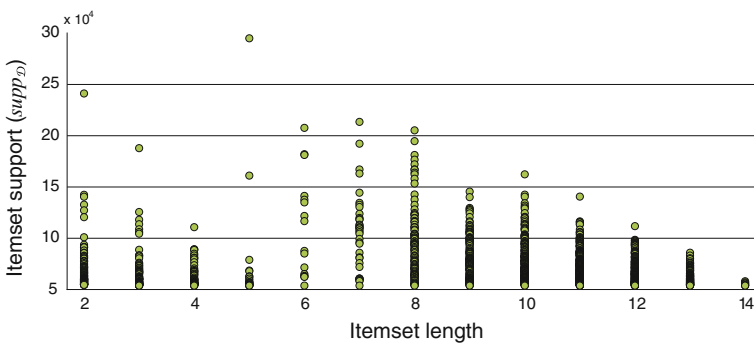
Fig. 9 Scatterplot of the supports and lengths of the non-singleton itemsets in the code table for the *Accidents* dataset, using *minsup* = 50000 and employing online pruning

7.4 Stability

Here, we verify the stability of KRIMP w.r.t. different candidate sets. First we investigate whether good results can be attained without the itemsets normally chosen in the code table. Given the large redundancy in the frequent pattern set, one would expect so.
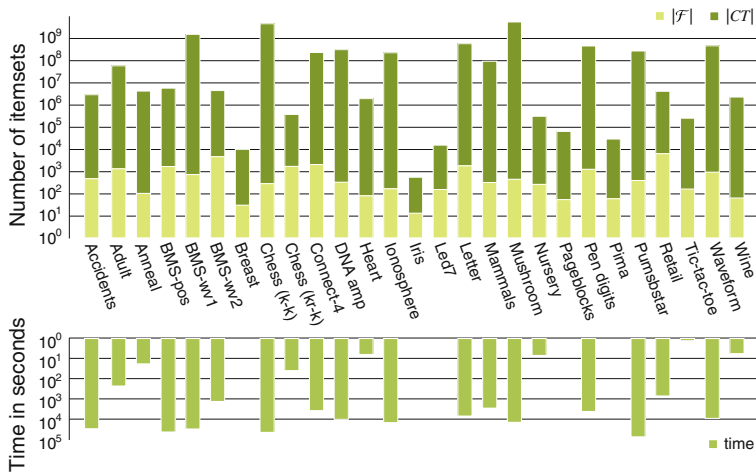
Fig. 10 The results of KRIMP with post-acceptance pruning on the 27 datasets using the *minsup* thresholds of Table 4. The dark coloured bars show the number of itemsets in the candidate set, $|\mathcal{F}|$, the lighter coloured bars the number of non-singleton itemsets selected in the code table, $|CT \setminus \mathcal{I}|$, and the *bottom* graph the associated runtimes for the single-threaded implementation

To this end, we first ran KRIMP using candidate set $\mathcal{F}$ to obtain CT. Then, for each $X \in \{CT \setminus \mathcal{I}\}$ we ran KRIMP using the candidate set $\mathcal{F} \setminus \{X\}$. In addition, we also ran KRIMP using $\mathcal{F} \setminus \{X \in \{CT \setminus \mathcal{I}\}\}$.

As a code table typically consists of about 100 to 1000 elements, a considerable set of experiments is required per database. Therefore, we use five of the datasets. The results of these experiments are presented in Table 5. The minute differences in compression ratios show that the performance of KRIMP does not rely on just those specific itemsets in the original code tables. Excluding all elements from the original code table results in compression ratios up till only .5% worse than normal. This is expected, as in this setting all structure in the data captured by the original code table has to be described differently. The results of the individual itemset exclusion experiments, on the other hand, are virtually equal to the original. In fact, sometimes shorter (better) descriptions are found this way: the removed itemsets were in the way of ones that offer a better description.

Next, we consider excluding far more itemsets as candidates. That is, we use closed, as opposed to all, frequent itemsets as candidates for KRIMP. For datasets with little noise the closed frequent pattern set can be much smaller and faster to mine and process. For the *minsup* thresholds depicted in Table 4, we mined both the complete and closed frequent itemset collections, and used these as candidate sets $\mathcal{F}$ for running KRIMP. Due to crashes of the closed frequent itemset miner, we have no results for the *Chess (k–k)* dataset. Figure 11 shows the comparison between the results of either candidate set in terms of the relative compression ratio. The differences in performance are slight, with an average increase in $L\%$ of about 1%. The only exception seems *Ionosphere*, where a 12% difference is noted, the reason of which is unclear. (The reduction in the number of candidate itemsets obtained by using closed itemsets is not exceptional compared to the other datasets. It may be an artifact of discretisation,

Table 5 Stability of the KRIMP given candidate sets with exclusions

Dataset	minsup	L% given candidates		
		$\mathcal{F}$	$\mathcal{F} \setminus X$	$\mathcal{F} \setminus CT$
Chess (kr–k)	1	61.6	61.7 ± 0.21	61.6
Mushroom	1	24.7	24.7 ± 0.01	25.0
Nursery	1	45.5	45.4 ± 0.36	46.0
Pen digits	50	46.7	46.7 ± 0.12	47.2
Wine	1	77.4	77.4 ± 0.26	78.0

Per dataset, the *minsup* threshold at which frequent itemsets were mined as candidates $\mathcal{F}$ for KRIMP. Further, the relative compression $L\%$ for running KRIMP with $\mathcal{F}$, the average relative compression attained by excluding single original code table elements from $\mathcal{F}$, and the relative compression attained by excluding all itemsets normally chosen from $\mathcal{F}$, i.e. using $\mathcal{F} \setminus \{X \in CT \mid X \notin \mathcal{I}\}$ as candidates for KRIMP. For *Mushroom* and *Pen digits* the closed frequent itemset collections were used as candidates.
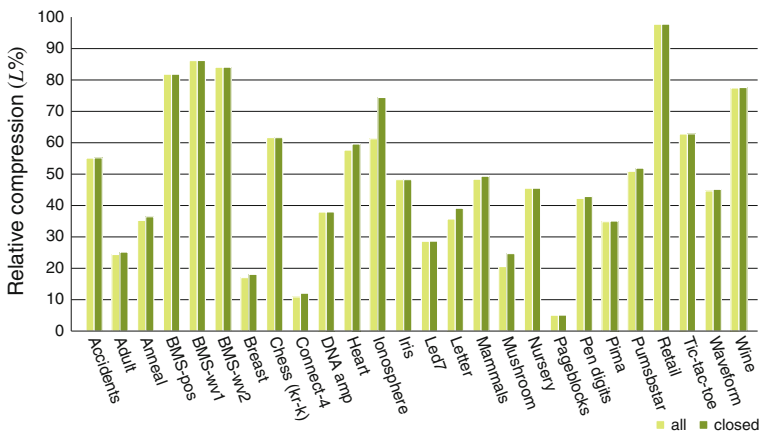
Fig. 11 The results of KRIMP with post-acceptance pruning on 26 datasets using the *minsup* thresholds of Table 4. Per dataset, the upward bars indicated the relative total compressed size ($L\%$). As candidates, $\mathcal{F}$, all (*left bars*), and closed (*right bars*) frequent itemsets were used

leading to dependent items.) Further, the resulting code tables are of approximately the same size; the 'closed' code tables consist of fewer itemsets: 10 on average. From these experiments we can conclude that closed frequent itemsets are a good alternative to be used as input for KRIMP, especially if otherwise too many frequent itemsets are mined.

7.5 Relevance

To evaluate whether KRIMP code tables model relevant structure, we employ swap randomisation (Gionis et al. 2007). Swap randomisation is the process of randomising data to obscure the internal dependencies, while preserving the row and column margins of the data. This is achieved by applying individual swap operations that maintain these margins. That is, one randomly finds and replaces two items I_i and I_j
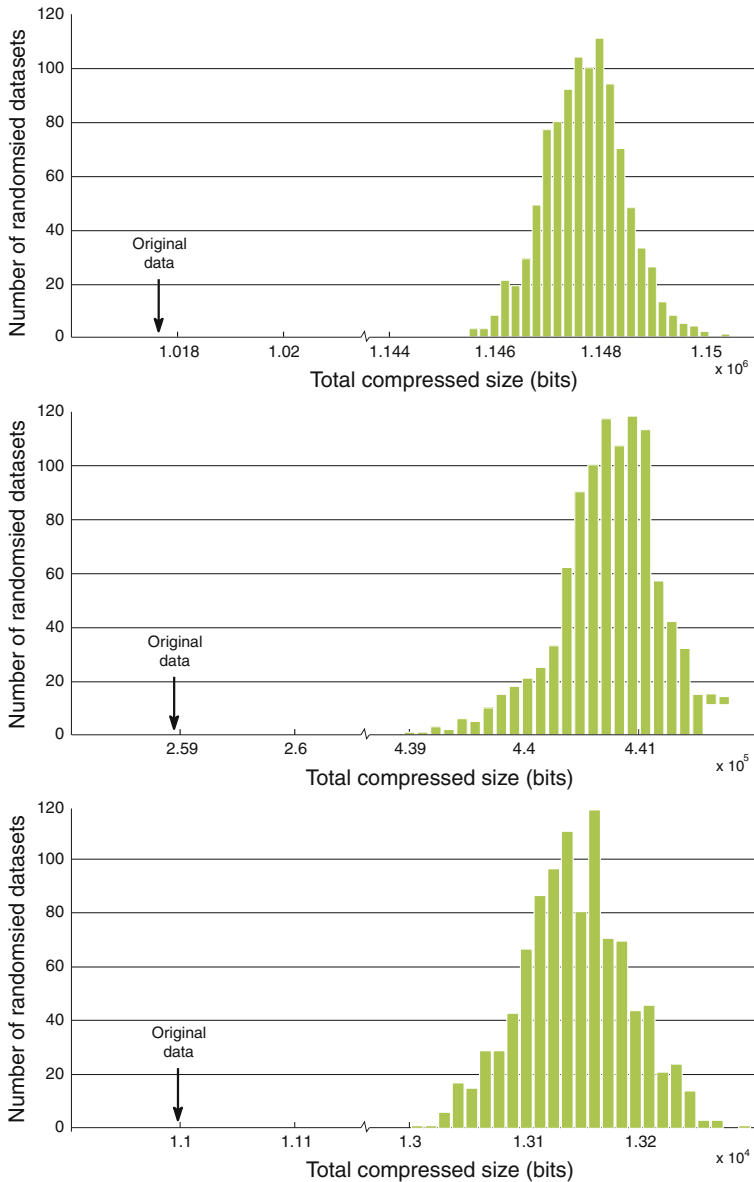
Fig. 12 Histograms of the total compressed sizes of 1000 swap randomised *BMS-wv1* (*top*), *Nursery* (*middle*) and *Wine* (*bottom*) datasets, using all frequent itemsets as candidates for KRIMP with post-acceptance pruning. Total compressed sizes of the original datasets are indicated by the arrows. Note the jumps in compressed size on the x-axes

(with $i \neq j$), and two transactions t_k and t_l (with $k \neq l$), such that I_i occurs in t_k but not in t_l, i.e. $I_i \in t_k$ and $I_j \notin t_l$, and analogously for I_j and t_l, i.e. $I_j \in t_l$ and $I_i \notin t_k$. Then, a swap operation swaps the items, i.e. $t_k \leftarrow \{t_k \cup I_j\} \setminus I_i$ and $t_l \leftarrow \{t_l \cup I_i\} \setminus I_j$.

Table 6 Swap randomisation experiments

Dataset	Original data			Swap randomised		
	$|CT \setminus \mathcal{I}|$	$\overline{|X|}$	$L\%$	$|CT \setminus \mathcal{I}|$	$\overline{|X|}$	$L\%$
BMS-wv1	718	2.8	86.7	277.9 ± 7.3	2.7 ± 0.0	97.8 ± 0.1
Nursery	260	5.0	45.5	849.2 ± 19.9	4.0 ± 0.0	77.5 ± 0.1
Wine	67	3.8	77.4	76.8 ± 3.4	3.1 ± 0.1	93.1 ± 0.4

Results of 1000 independent swap randomisation experiments per dataset. As many swaps were applied as there are 1's in the data. By $\overline{|X|}$ we denote the average cardinality of itemsets $X \in CT \setminus \mathcal{I}$. The results for the swap randomisations are averaged over the 1000 experiments per dataset. As candidates for KRIMP with post-acceptance pruning we used all frequent itemsets, *minsup* = 1, except for *BMS-wv1* for which we set *minsup* = 35.

This process, effectively a Markov chain, has to be repeated numerous times, as often as is required to break down the significant structure of the data, i.e. the mixing time of the chain. No hard results exist on the optimal value of swaps, so we follow Gionis et al. and apply as many swaps as there are 1s in the data.

The idea is that if the true structure of the data is captured, there should be significant differences between the models found in the original and randomised datasets.

To this end, we compare the total compressed size of the actual data to those of 1000 swap randomised versions of the data. As this implies a large number of experiments, we have to restrict ourselves to a small selection of datasets. To this end, we chose three datasets with very different characteristics: *BMS-wv1*, *Nursery* and *Wine*. To get reasonable numbers of candidate itemsets (i.e. a few million) from the randomised data we use *minsup* thresholds of 1 for the latter two datasets and 35 for *BMS-wv1*.

Figure 12 shows the histogram of the total compressed sizes of these 1000 randomisations. The total compressed sizes of the original databases are indicated by arrows. The standard encoded size for these three databases, $L(\mathcal{D}, ST)$, are 1173962, 569042 and 14100 bits, respectively. The graphs show that the original data can be compressed significantly better than the randomised datasets (p-value of 0). Further quantitative results are shown in Table 6. Besides much better compression, we see that for *Nursery* and *Wine* the code tables induced on the original data contain fewer, but more specific (i.e. longer) itemsets. For *BMS-wv1* the randomised data is virtually incompressible with KRIMP ($L\% \approx 98\%$), and as such much fewer itemsets are selected.

7.6 Classification

As an independent evaluation of the quality of the itemsets picked by KRIMP, we compare the performance of the KRIMP classifier (detailed in Sect. 6.2) to the performance of a wide range of well-known and top-performing classifiers. We consider rule-induction-based methods such as C4.5 (Quinlan 1993a), FOIL (Quinlan 1993b) and CPAR (Yin and Han 2003). Mehta et al. (1996) use MDL to prune decision trees for classification. However, we are more interested in the comparison to association-rule-based algorithms like iCAEP (Zhang et al. 2000), HARMONY

(Wang and Karypis 2005), CBA (Liu et al. 1998) and LB (Meretakis et al. 2000) as these also employ a collection of itemsets for classification. Because we argued that our method is strongly linked to the principle of Naïve Bayes (NB) (Duda and Hart 1973) it is imperative we compare to it. Our hypothesis is that, if the code table-based classifier performs on-par, KRIMP selects itemsets that are characteristic for the data. Further, because these methods were devised with the goal of classification in mind, opposed to KRIMP, we would expect them to (slightly) outperform the KRIMP classifier.

We use the same *minsup* thresholds as we used for the compression experiments, listed in Table 4. Although the databases are now split on class before they are compressed by KRIMP, these thresholds still provide large numbers of candidate itemsets and result in code tables that compress well.

It is not always beneficial to use the code tables obtained for the lowest *minsup*, as class sizes are often unbalanced or more structure is present in one class than in another. Also, overfitting may occur for very low values of *minsup*. Therefore, we store code tables at fixed support intervals during the pattern selection process. During classification, we need to select one code table for each class. To this end, we 'pair' the code tables using two methods: *absolute* and *relative*. In absolute pairing, code tables that have been generated at the same support levels are matched. Relative pairing matches code tables of the same relative support between 100% and 1% of the maximum support value (per class, equals the number of transactions in a class). We evaluate all pairings and choose that one that maximises accuracy.

All results reported in this section have been obtained using 10-fold cross-validation. As performance measure we use accuracy, the percentage of true positives on the test data. We compare to results obtained with six state-of-the-art classifiers. All scores for Naïve Bayes, C4.5 and SVM have been obtained using Weka (Witten and Frank 2005), with default parameter settings. For CBA we used the implementation that is part of the LUCS-KDD software library (Coenen 2004) and the settings recommended by Liu et al. (1998) (minsup = 1%, minimum confidence = 50%). With CBA, the parameters can make a large difference and we had to fine-tune these for two datasets to get reasonable results. For all scores for NB, C4.5, SVM and CBA, exactly the same discretisation was used as for KRIMP. The scores for iCAEP and HARMONY have been taken from Zhang et al. (2000) and Wang and Karypis (2005) respectively. Note that, in contrast with the other methods, we used the sparse version of *Connect*–4 for KRIMP, as described in Sect. 7.2.

Before we compare the KRIMP classification performance to other methods, we verify whether there is a qualitative difference between using all or only closed frequent itemsets as candidates and using post-acceptance pruning or not. Table 7 shows

Table 7 Results of KRIMP classification, for all/closed frequent itemsets and without/with pruning

Candidates	w/o Pruning	With pruning
All	84.3 ± 14.2	84.5 ± 13.1
Closed	84.0 ± 13.9	83.7 ± 14.2

For each combination of candidates and pruning, the average accuracy (%) over the 19 datasets from Table 8 is given. Standard deviation is high as a result of the diverse range of datasets used.

Table 8 Results of KRIMP classification, compared to six state-of-the-art classifiers

Dataset	Baseline	KRIMP	NB	C4.5	CBA	HRM	iCAEP	SVM
Adult	76.1	84.3	80.2	**85.5**	84.2	81.9	80.9	84.7
Anneal	76.2	96.6	96.3	**97.8**	94.7		95.1	95.3
Breast	65.5	94.1	93.3	94.1	94.0		**97.4**	93.7
Chess (k–k)	52.2	90.0	87.6	**99.4**	72.8		94.6	93.9
Chess (kr–k)	16.2	57.9	35.9	**78.5**	25.8[a]	44.9		46.3
Connect–4	65.8	69.4	67.9	**80.1**	68.7	68.1	69.9	77.6
Heart	54.1	61.7	55.1	54.8	57.3		**80.3**	58.4
Ionosphere	64.1	**91.0**	90.9	90.9	87.2[b]		90.6	90.9
Iris	33.3	**96.0**	94.7	94.0	94.0	94.7	93.3	**94.0**
Led7	11.0	75.3	75.4	75.3	66.6	74.6		**75.8**
Letter	4.1	70.9	57.2	**77.5**	28.6	76.8		69.8
Mushroom	51.8	**100**	94.0	**100**	46.4	99.9	99.8	99.9
Nursery	33.3	92.3	92.2	**99.5**	90.1	92.8	84.7	97.6
Page blocks	89.8	**92.6**	91.8	92.5	90.9	91.6		92.2
Pen digits	10.4	95.0	84.2	95.6	87.4	96.2		**96.6**
Pima	65.1	72.7	73.8	72.7	**75.0**	73.0	72.3	74.0
Tic–tac–toe	65.3	88.7	68.8	93.3	**100**	81.0	92.1	87.9
Waveform	33.9	77.1	77.4	74.1	77.6	80.5	**81.7**	80.1
Wine	39.9	**100**	95.5	96.6	53.2	63.0	98.9	97.2
Average	47.8	84.5	79.6	87.0	75.4			84.5

For each dataset, baseline accuracy and accuracy (%) obtained with KRIMP classification is given, as well as accuracies obtained with six other classifiers are given. We use HRM as abbreviation for HARMONY. The highest accuracy per dataset is displayed in boldface. Additionally, the average score is given for all methods for which all scores are available. For KRIMP with post-acceptance pruning, per class, frequent itemsets mined at thresholds found in Table 4 were used as candidates. All results are 10-fold cross-validated. Non-default settings for CBA: [a] *minconf* = 25%. [b] *minsup* = 10%.

that the variation in average accuracy is very small, but using all frequent itemsets as candidates with pruning gives the highest accuracy, making it an obvious choice for inspection of more detailed results in the rest of this subsection.

Classification results with all frequent itemsets as candidates and post-acceptance pruning are presented in Table 8, together with accuracies obtained with 6 competitive classifiers. Baseline accuracy is the (relative) size of the largest class in a dataset. For about half of the datasets, the maximum score is provided by an absolute pairing, in the other cases the scores are obtained with a relative pairing. As expected, relative pairing performs well especially for datasets with small classes or unbalanced class sizes. In general though, the difference in maximum accuracy between the two types of pairings is very small, i.e. <1%. For a few datasets, the difference is more notable, 2–10%.

Looking at the scores, we observe that performance on most datasets is very similar for most algorithms. For *Pima*, for example, all accuracies are within a very small range. Because of this, it is important to note that performance may vary up to a few

percent depending on the (random) partitioning used for cross-validation, especially for datasets having smaller classes. Although the same partitioning was used for all results obtained with Weka, this is not the case for the other results. Therefore, we cannot conclude that a particular classifier is better on a certain dataset if the difference is not larger than by a margin of, e.g. 2–3%.

We used a two-tailed paired t-test to analyse whether the results obtained by the KRIMP classifier are significantly *different* (with a *p*-value of 0.05). This analysis shows us that this is indeed the case for the comparisons to Naïve Bayes and CBA. However, there is no significant difference between KRIMP and C4.5, HARMONY, or iCAEP (respective test scores of 0.10, 0.20 and 0.22). Moreover, the comparison between the KRIMP classifier and SVM shows they perform highly similar, with a score of 0.99. For our goal, these are excellent scores, as they show that the results obtained by KRIMP are on par with the best scoring classifiers around—which says a lot about how well the patterns that KRIMP selects describe the data distribution.

If we go back to the plain accuracy scores, we note that the KRIMP classifier scores 5 wins, indicated in boldface, which is only beaten by C4.5 with 8 wins. Additionally, the achieved accuracy is close to the winning score in five cases, and average for the remaining 9 datasets.

Compared to Naïve Bayes, to which our method is closely related, we observe that the obtained scores are indeed quite similar, with a slight advantage for KRIMP. This also shows in the average accuracy, which is equal to that of SVM, only C4.5 performs better. Note that CBA was ran with the default parameter settings: minimum support 1% and minimum confidence 50%.

We also looked at the performance of FOIL, PRM and CPAR (Quinlan 1993b; Yin and Han 2003) reported in Coenen (2004). These classifiers perform sub-par in comparison to those in Table 8 though. A comparison to LB and/or LB-chi2 (Meretakis et al. 2000) is problematic, as no implementation is publically available, only few accuracies are reported for the (large) datasets we use, and for those that are available, the majority of the LB results is based on train/test, and are not 10-fold cross-validated.

To provide further insight in the classification results, confusion matrices for three datasets are given in Table 9. The confusion matrix for *Heart* shows us why the KRIMP classifier is unable to perform well on this dataset: it contains four very small classes. For such small databases, the size of the code table is dominant, precluding the discovery of the important frequent itemsets. This is obviously the case for some *Heart* classes. If we consider *Mushroom* and *Iris*, then the bigger the classes, the better the results. In other words, if the classes are big enough, the KRIMP classifier performs very well.

We can zoom in even further to show how the classification principle works in practice. Take a look at the effect of coding a transaction with the 'wrong' code table, which is illustrated in Fig. 13. The rounded boxes in this figure visualise the itemsets that make up the cover of the transaction. Each of the itemsets is linked to its code by the dashed line. The widths of the black and white encodings represent the actual computed code lengths. From this figure, it is clear that code tables can be used to both characterise and recognise data distributions.

Recall that KRIMP was not designed with classification in mind. We designed the KRIMP classifier primarily to test the quality of the selected patterns. Because the

Table 9 Confusion matrices

	Mushroom		Iris			Heart				
	1	2	1	2	3	1	2	3	4	5
1	4208	0	47	2	0	142	22	9	6	2
2	0	3916	2	48	1	17	23	8	9	5
3			1	0	40	3	2	12	4	2
4						0	7	5	10	4
5						2	1	2	6	0

The values denote how many transactions with class *column* are classified as class *row*
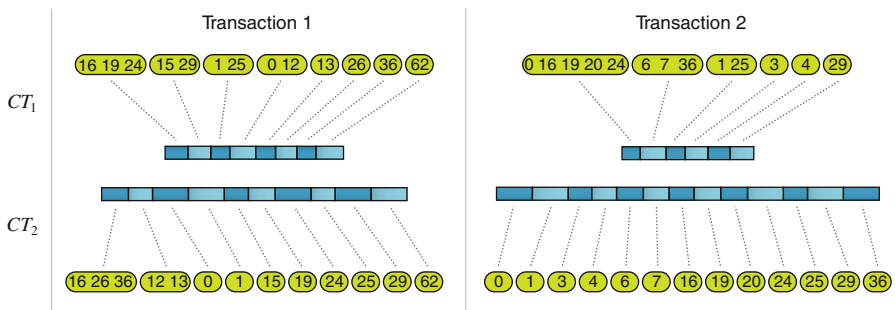

Fig. 13 *Wine*; two transactions from class 1, $\mathcal{D}_1$, encoded by the code tables for class 1, CT_1 (*top*), and class 2, CT_2 (*bottom*)

results of the code table-based classifier are on par to those of the state-of-the-art, we conclude that the itemsets selected by KRIMP are very characteristic for the data. That a high quality selection of itemsets is crucial for good classification performance can be deduced from the sub-par performance of CBA, a classifier based on all association-rules that satisfy a given minsup and minimum confidence threshold.

7.7 Order

Next, we investigate the order heuristics of the KRIMP algorithm. Both the standard cover order and standard candidate order are rationally made choices, but choices nevertheless. Here, we consider a number of alternatives for either and evaluate the quality of possible combinations through compression ratios and classification accuracies. The outcome of these experiments are shown in Table 10. Before we cover these results, we discuss the orders. As before, ↓ indicates the property to be sorted descending, and ↑ ascending.

For the **Standard Cover Algorithm**, we experimented with the following orders of the coding set.

– **Standard Cover Order**:

$$|X| \downarrow \quad supp_{\mathcal{D}}(X) \downarrow \quad \text{lexicographically} \uparrow$$

Table 10 Evaluation of candidate and cover orders

$\mathcal{F}\downarrow$	Cover order							
	Standard		Entry		Area		Random	
	L%	acc.(%)	L%	acc.(%)	L%	acc.(%)	L%	acc.(%)
Standard	44.2	88.6	43.7	88.7	51.1	88.1	49.5	88.1
Standard'	44.2	88.5	43.7	88.8	51.6	88.1	49.5	88.3
Length	45.2	88.0	43.9	87.9	55.4	87.1	49.1	78.8
Area	48.6	88.0	64.5	88.4	64.4	88.1	65.0	88.0
Random	49.4	86.8	51.2	87.0	57.5	86.8	50.8	87.0

Results for 20 combinations of candidate and cover orders for Krimp with post-acceptance pruning. Shown are average relative Krimp compression, $L\%$, and average classification accuracy (%) on a number of datasets. Results for compression and classification are averaged over 16 respectively 11 datasets. Table 11 in Appendix shows which datasets were used and at what *minsup* thresholds the candidate sets, $\mathcal{F}$, were mined for these experiments.

– **Entry**:

$$\text{position of } X \text{ in } \mathcal{F}, \downarrow$$

– **Area ascending**:

$$|X| \times supp_{\mathcal{D}}(X) \uparrow \quad \text{position of } X \text{ in } \mathcal{F}, \downarrow$$

We also consider the **Random** cover order, where new itemsets are entered at a random position in the code table. For all the above orders, we sort the singleton itemsets below itemsets of longer length. In Table 10 we refer to these orders, respectively, as Standard, Entry, Area and Random. We further experimented with a number of alternatives of the above, but the measured performance did not warrant their inclusion in the overall comparison.

As for the lineup in which the itemsets are considered by the Krimp algorithm, the candidate order, we experimented with the following options.

– **Standard Candidate Order**:

$$supp_{\mathcal{D}}(X) \downarrow \quad |X| \downarrow \quad \text{lexicographically} \uparrow$$

– **Standard, but length ascending**:

$$supp_{\mathcal{D}}(X) \downarrow \quad |X| \uparrow \quad \text{lexicographically} \uparrow$$

– **Length ascending, support descending**:

$$|X| \uparrow \quad supp_{\mathcal{D}}(X) \downarrow \quad \text{lexicographically} \uparrow$$

– **Area descending, support descending**:

$$|X| \times supp_{\mathcal{D}}(X) \downarrow \quad supp_{\mathcal{D}}(X) \downarrow \quad \text{lexicographically} \uparrow$$

A **Random** candidate order is also considered, which is simply a random permutation of the candidate set. In Table 10 we refer to these orders as, respectively, Standard, Standard', Length, Area and Random. Again, we considered a number of variants of the above, for none of which the performance was good enough to be included here. This includes the order with which the (depth-first) frequent itemset mining algorithm we employ outputs the itemsets. Most likely, this is because the most specific itemsets come first in this order, disallowing the algorithm to incrementally refine compression.

For all 20 combinations of the above cover and candidate orders we ran compression experiments on 16 datasets and classification experiments for 11 datasets. As candidate itemsets, we ran experiments with both the complete and closed frequent itemset collections. Due to the amount of experiments required by this evaluation we only used single samples for the random orders. Classification scores are 10-fold cross-validated, as usual. Per dataset, the score of the best performing pairing (absolute or relative) was chosen. The details on which datasets and what *minsup* thresholds were used can be found in Appendix Table 11.

The results of these experiments are depicted in Table 10. Shown are, per combination of orders, the total compression ratio $L\%$ and the classification accuracy, both of which are averaged over all databases and both all and closed frequent itemsets as candidate sets.

Between the orders, we measure considerable differences in compression ratio, up to 15%. For the candidate orders, the standard cover order is the best choice. The difference between the two variants Standard and Standard', is negligible, while the other options perform significantly worse for cover orders Standard and Entry. The same can be said for classification. We note that, although intuitively it seems a good choice, all variants of the area-descending order we further tested perform equally, but sub-par.

For the standard cover algorithm, order-of-entry performs best, with the standard cover order second at a slight margin of half a percent. Covering in order of area shows outright bad performance, loosing even to random. As order-of-entry shows the best compression ratios, it is preferred from a MDL point of view. However, the standard order has the practical benefit of being (much) faster in practice. As it does not always insert new elements at the 'top' of the code table, partially covered transactions can be cached, speeding up the cover process significantly. The differences between the two, both in terms of compression and classification accuracies, are small, warranting the choice of the 'suboptimal' standard cover order for the practical reason of speed.

The scores for the random orders show that the greedy covering and MDL-based selection are most important for attaining good compression ratios. With either the candidate and/or cover order being random, KRIMP still typically attains ratios of 50% and average accuracies are far above the baseline of 47.7%. This is due to the redundancy in the candidate sets and the cover order being fixed, even when the insertion position for a candidate is random. This allows the selection process to still pick sets of high-quality itemsets, albeit sub-optimal.

8 Discussion

The experimental evaluation shows that KRIMP provides a practical solution to the well-known explosion in pattern mining. It reduces the highly redundant frequent itemset collections with many orders of magnitude to sets of only hundreds of high-quality itemsets. High compression ratios indicate that these itemsets are characteristic for the data and non-redundant in-between. Swap randomisation experiments show that the selections model relevant structure, and exclusion of itemsets shows that the method is stable with regard to noise. The quality of the itemsets is independently validated through classification, for which we introduced theory to classify by code-table based compression. While the patterns are chosen to compress well, the KRIMP classifier performs on par with state-of-the-art classifiers.

KRIMP is a heuristic algorithm, as is usual with MDL: the search space is by far too large to consider fully, especially since it is unstructured. The empirical evaluation of the choices made in the design of the algorithm show that the standard candidate order is the best, both from a compression and a classification perspective. The standard order in which itemsets are considered for covering a transaction is near-optimal; the order-of-entry approach, where new itemsets are used maximally, achieves slightly better compression ratios and classification accuracies. However, the standard order allows for efficient caching, speeding up the cover process considerably while hardly giving in on quality. Post-acceptance pruning is shown to improve the results: fewer itemsets are selected, providing better compression ratios and higher classification accuracies. Although pruning requires itemsets in the code table to be reconsidered regularly, its net result is a speed-up as code tables are kept smaller and the cover process thus needs to consider fewer itemsets to cover a transaction.

The timings reported in this study show that compression is not only a good, but also a realistic approach, even for large databases and huge candidate collections; the single-threaded implementation already considers up to hundreds of thousands of itemsets per second. While highly efficient frequent itemset miners were used, we observed that mining the candidates sometimes takes (much) longer than the actual KRIMP selection. Also, the algorithm can be easily parallelised, both in terms of covering parts of the database and of checking the candidate itemsets. The implementation[4] we used for the experiments in this chapter uses the latter option, as experiments showed that the performance of the former deteriorates rapidly for candidate itemsets with low support.

In general, the larger the candidate set, the better the compression ratio. The total compressed size decreases continuously, even for low *minsup* values, i.e. it never converges. Hence, $\mathcal{F}$ should be mined at a *minsup* threshold as low as possible. Given a suited frequent itemset miner, experiments could be done iteratively, continuing from the so-far optimal code table and corresponding previous *minsup*. For many datasets, it is computationally feasible to set *minsup* $= 1$.

When mining all frequent itemsets for a low *minsup* is infeasible, using closed frequent itemsets as candidate set is a good alternative instead. For most datasets,

---

[4] Our implementation of KRIMP is freely available for research purposes from
http://www.cs.uu.nl/groups/ADA/krimp/

results obtained with closed are almost as good as with all frequent itemsets, while for some datasets this makes the candidate set much smaller and thus computationally attractive.

KRIMP selects both specific and more general itemsets to describe the data. The best data descriptions are attained when it can consider all possible itemsets in the database. However, if only itemsets of particular supports or characteristics are desired, the input can be adjusted accordingly: by simply providing KRIMP only those itemsets, e.g. mined at a particular *minsup*.

KRIMP can be regarded a parameter-free algorithm and used as such in practice. The candidate set is a parameter, but since larger candidate sets give better results this can always be set to all frequent itemsets with *minsup* = 1. Only when this default candidate set turns out to be too large to handle for the available implementation and hardware, this needs to be tuned. Additionally, using post-acceptance pruning always improves the results and even results in a speed-up in computation time, so there is no reason not to use this.

Although code tables are made to just compress well, it turns out they can easily be used for classification. Because other classifiers have been designed with classification in mind, we expected these to outperform the KRIMP classifier. We have shown this is not the case: KRIMP performs on par with the best classifiers available. We draw two conclusions from this observation. Firstly, KRIMP selects itemsets that are very characteristic for the data. Secondly, the compression-based classification scheme works very well.

While this paper covers a large body of work done, there remains plenty of future work left to do. For example, KRIMP could be further improved by directly generating candidate itemsets from the data and its current cover. Or, all frequent itemsets could be generated on-the-fly from a closed candidate set. Both extensions would address the problems that occur with extremely large candidate sets, i.e. crashing itemset miners and IO being the bottleneck instead of CPU time.

9 Conclusions

In this paper we have shown how MDL gives a dramatic reduction in the number of frequent itemsets that one needs to consider. For twenty-seven data sets, the reductions reached by the KRIMP algorithm ranges up to seven orders of magnitude; only hundreds of itemsets are required to succinctly describe the data. The algorithm shows a high stability w.r.t. different candidate sets. It is parameter-free for all practical purposes; for the best result, use as large as possible candidate sets and enable pruning.

Moreover, by designing a simple classifier we have shown that KRIMP picks itemsets that matter. This is all the more telling since the selection of code table elements does not take predictions into account. The small sets that are selected characterise the database accurately, as is also indicated by small compressed sizes and swap randomisation experiments.

In this paper, we verified the heuristic choices we made for the KRIMP algorithm in Siebes et al. (2006). We extensively evaluated different possible orders for both the candidate set and code table. The outcome is that the standard orders are very good: no

combination of orders was found that performs significantly better, while the standard orders offer good opportunities for optimisation.

Because we set the frequent pattern explosion, the original problem, in a wide context but discussed only frequent itemsets, the reader might wonder: does this also work for other types of patterns? The answer is affirmative, in Bathoorn et al. (2006) we have shown that our MDL-based approach also works for pattern-types such as *frequent episodes* for sequence data and *frequent subgraphs* for graph data. In Koopman and Siebes (2008, 2009), we extended the approach to multi-relational databases, i.e. to select patterns over multiple tables. Also, the LESS algorithm (Heikinheimo et al. 2009) (see also Sect. 2) introduces an extension of the encoding such that it can be used to select more generic patterns, e.g. low-entropy sets.

Like detailed in Faloutsos and Megalooikonomou (2007), there are many data mining tasks for which compression, and thus the foundations presented in this paper, can be used. e.g. we have independently shown that compression (or, more specifically, KRIMP) can be successfully employed for (but not limited to), including characterising differences (Vreeken et al. 2007a), generating data and preserving privacy (Vreeken et al. 2007b), detecting change in data streams (van Leeuwen and Siebes 2008), imputing missing values (Vreeken and Siebes 2008), and identifying components in a database (van Leeuwen et al. 2009).

Appendix

See Table 11

Table 11 Datasets and settings used for the candidate and cover order experiments

Dataset	*minsup*		Used for	
	All	Closed	Compression	Classification
Adult	20	1	✓	✓
Anneal	1	1	✓	✓
Breast	1	1	✓	✓
Chess (kr–k)	1	1	✓	
DNA amplification	10	1	✓	
Ionosphere	50	1	✓	✓

Table 11 continued

Dataset	*minsup*		Used for	
	All	Closed	Compression	Classification
Iris	1	1	✓	✓
Led7	1	1	✓	✓
Letter recognition	50	20	✓	
Mammals	545	545	✓	
Mushroom		1	✓	✓
Nursery	1	1	✓	
Pen digits	20	1	✓	✓
Pima	1	1	✓	✓
Waveform	50	5	✓	✓
Wine	1	1	✓	✓

Details for which datasets and which settings were used for the candidate and cover order experiments. Per dataset, shown are the *minsup* thresholds at which candidate sets, $\mathcal{F}$, were mined for all and closed frequent itemsets. Ticks indicate which datasets were used for compression experiments and which for the classification experiments. In total, tens of thousands of individual KRIMP compression runs were required for these order experiments

References

Agrawal R, Mannila H, Srikant R, Toivonen H, Verkamo AI (1996) Fast discovery of association rules. In: Advances in knowledge discovery and data mining, AAAI, pp 307–328

Bathoorn R, Koopman A, Siebes A (2006) Reducing the frequent pattern set. In: Proceedings of the ICDM-workshops'06, pp 55–59

Bayardo R (1998) Efficiently mining long patterns from databases. In: Proceedings of SIGMOD'98, pp 85–93

Bringmann B, Zimmermann A (2007) The chosen few: on identifying valuable patterns. In: Proceedings of the ICDM'07, pp 63–72

Calders T, Goethals B (2002) Mining all non-derivable frequent itemsets. In: Proceedings of the ECML PKDD'02, pp 74–85

Chakrabarti D, Papadimitriou S, Modha DS, Faloutsos C (2004) Fully automatic cross-associations. In: Proceedings of KDD'04, pp 79–88

Chakrabarti S, Sarawagi S, Dom B (1998) Mining surprising patterns using temporal description length. In: Proceedings of VLDB'98, Morgan Kaufmann, San Francisco, pp 606–617

Chandola V, Kumar V (2007) Summarization—compressing data into an informative representation. Knowl Inf Syst 12(3):355–378

Coenen F (2003) The LUCS–KDD discretised/normalised ARM and CARM data library. http://www.csc.liv.ac.uk/~frans/KDD/Software/LUCS-KDD-DN/DataSets/dataSets.html

Coenen F (2004) The LUCS–KDD software library. http://www.csc.liv.ac.uk/~frans/KDD/Software

Cover T, Thomas J (2006) Elements of information theory, 2nd edn. John Wiley and Sons, New York

Crémilleux B, Boulicaut JF (2002) Simplest rules characterizing classes generated by δ-free sets. In: Proceedings of KBSAAI'02, pp 33–46

Duda R, Hart P (1973) Pattern classification and scene analysis. John Wiley and Sons, New York

Faloutsos C, Megalooikonomou V (2007) On data mining, compression and Kolmogorov complexity. Data Min Knowl Discov 15(1):3–20

Geerts F, Goethals B, Mielikäinen T (2004) Tiling databases. In: Proceedings of DS'04, pp 278–289

Gionis A, Mannila H, Mielikäinen T, Tsaparas P (2007) Assessing data mining results via swap randomization. ACM Trans Knowl Discov Data 1(3):14

Goethals B, Zaki MJ (2003) Frequent itemset mining implementations repository (FIMI). http://fimi.cs.helsinki.fi

Grünwald PD (2005) Minimum description length tutorial. In: Grünwald P, Myung I (eds) Advances in minimum description length. MIT Press, Cambridge

Grünwald PD (2007) The minimum description length principle. MIT Press, Cambridge

Hand D, Adams N, Bolton R (eds) (2002) Pattern detection and discovery. Springer, New York

Heikinheimo H, Hinkkanen E, Mannila H, Mielikäinen T, Seppänen JK (2007) Finding low-entropy sets and trees from binary data. In: Proceedings of KDD'07, pp 350–359

Heikinheimo H, Vreeken J, Siebes A, Mannila H (2009) Low-entropy set selection. In: Proceedings of SDM'09, pp 569–579

Karp RM (1972) Reducibility among combinatorial problems. In: Miller R, Thatcher J (eds) Proceedings of a symposium on the complexity of computer computations. Plenum Press, New York, USA, pp 85–103

Keogh E, Lonardi S, Ratanamahatana CA (2004) Towards parameter-free data mining. In: Proceedings of KDD'04, pp 206–215

Keogh E, Lonardi S, Ratanamahatana CA, Wei L, Lee SH, Handley J (2007) Compression-based data mining of sequential data. Data Min Knowl Discov 14(1):99–129

Knobbe AJ, Ho EKY (2006a) Maximally informative *k*-itemsets and their efficient discovery. In: Proceedings of KDD'06, pp 237–244

Knobbe AJ, Ho EKY (2006b) Pattern teams. In: Proceedings of the ECML PKDD'06, pp 577–584

Kohavi R, Brodley C, Frasca B, Mason L, Zheng Z (2000) KDD-Cup 2000 organizers' report: peeling the onion. SIGKDD Explor 2(2):86–98. http://www.ecn.purdue.edu/KDDCUP

Koopman A, Siebes A (2008) Discovering relational items sets efficiently. In: Zaki M, Wang K (eds) Proceedings of SDM'08, SIAM, pp 108–119

Koopman A, Siebes A (2009) Characteristic relational patterns. In: Proceedings of KDD'09, pp 437–446

Li M, Vitányi P (1993) An introduction to Kolmogorov complexity and its applications. Springer, New York

Liu B, Hsu W, Ma Y (1998) Integrating classification and association rule mining. In: Proceedings of KDD'98, pp 80–86

Liu G, Lu H, Yu JX, Wei W, Xiao X (2004) AFOPT: an efficient implementation of pattern growth approach. In: Proceedings of the 2nd workshop on frequent itemset mining implementations

Mannila H, Toivonen H (1996) Multiple uses of frequent sets and condensed representations. In: Proceedings of KDD'96, pp 189–194

Mannila H, Toivonen H (1997) Levelwise search and borders of theories in knowledge discovery. Data mining and knowledge discovery, pp 241–258

Mehta M, Agrawal R, Rissanen J (1996) Sliq: a fast scalable classifier for data mining. In: Advances in database technology. Springer, NY, pp 18–32

Meretakis D, Lu H, Wüthrich B (2000) A study on the performance of large bayes classifier. In: Proceedings of the ECML'00, pp 271–279

Mielikäinen T, Mannila H (2003) The pattern ordering problem. In: Proceedings of the ECML PKDD'03, pp 327–338

Mitchell-Jones AJ, Amori G, Bogdanowicz W, Krystufek B, Reijnders PJH, Spitzenberger F, Stubbe M, Thissen JBM, Vohralik V, Zima J (1999) The atlas of European mammals. Academic Press, London

Morik K, Boulicaut JF, Siebes A (eds) (2005) Local pattern detection. Springer, New York

Myllykangas S, Himberg J, Böhling T, Nagy B, Hollmén J, Knuutila S (2006) Dna copy number amplification profiling of human neoplasms. Oncogene 25(55)

Pasquier N, Bastide Y, Taouil R, Lakhal L (1999) Discovering frequent closed itemsets for association rules. In: Proceedings of the ICDT'99, pp 398–416

Pfahringer B (1995) Compression-based feature subset selection. In: Proceedings of the IJCAI'95 workshop on data engineering for inductive learning, pp 109–119

Quinlan J (1993b) C4.5: programs for machine learning. Morgan-Kaufmann, Los Altos

Quinlan J (1993b) FOIL: a midterm report. In: Proceedings of the ECML'93

Rissanen J (1978) Modeling by shortest data description. Automatica 14(1):465–471

Siebes A, Vreeken J, van Leeuwen M (2006) Item sets that compress. In: Proceedings of SDM'06, pp 393–404

Sun J, Faloutsos C, Papadimitriou S, Yu PS (2007) Graphscope: parameter-free mining of large time-evolving graphs. In: Proceedings of KDD'07, pp 687–696

Tatti N, Vreeken J (2008) Finding good itemsets by packing data. In: Proceedings of the ICDM'08, pp 588–597

van Leeuwen M, Siebes A (2008) Streamkrimp: detecting change in data streams. In: Proceedings of ECMLPKDD'08, Springer, Heidelberg, pp 672–687

van Leeuwen M, Vreeken J, Siebes A (2006) Compression picks the item sets that matter. In: Proceedings of the ECML PKDD'06, pp 585–592

van Leeuwen M, Vreeken J, Siebes A (2009) Identifying the components. Data Min Knowl Discov 19(2):173–292

Vreeken J, Siebes A (2008) Filling in the blanks—KRIMP minimisation for missing data. In: Proceedings of the ICDM'08, pp 1067–1072

Vreeken J, van Leeuwen M, Siebes A (2007a) Characterising the difference. In: Proceedings of KDD'07, pp 765–774

Vreeken J, van Leeuwen M, Siebes A (2007b) Preserving privacy through data generation. In: Proceedings of the ICDM'07, pp 685–690

Wallace C (2005) Statistical and inductive inference by minimum message length. Springer, New York

Wang J, Karypis G (2005) HARMONY: efficiently mining the best rules for classification. In: Proceedings of SDM'05, pp 205–216

Wang J, Karypis G (2006) On efficiently summarizing categorical databases. Knowl Inf Syst 9(1):19–37

Wang C, Parthasarathy S (2006) Summarizing itemset patterns using probabilistic models. In: Proceedings of KDD'06, pp 730–735

Warner H, Toronto A, Veasey L, Stephenson R (1961) A mathematical model for medical diagnosis, application to congenital heart disease. J Am Med Assoc 177:177–184

Witten I, Frank E (2005) Data mining: practical machine learning tools and techniques. 2nd edn. Morgan Kaufmann, San Francisco

Xiang Y, Jin R, Fuhry D, Dragan FF (2008) Succinct summarization of transactional databases: an overlapped hyperrectangle scheme. In: Proceedings of KDD'08, pp 758–766

Xin D, Han J, Yan X, Cheng H (2005) Mining compressed frequent-pattern sets. In: Proceedings of VLDB'05, pp 709–720

Yan X, Cheng H, Han J, Xin D (2005) Summarizing itemset patterns: a profile-based approach. In: Proceedings of KDD'05, pp 314–323

Yin X, Han J (2003) CPAR: Classification based on predictive association rules. In: Proceedings of SDM'03, pp 331–335

Zhang X, Guozhu D, Ramamohanarao K (2000) Information-based classification by aggregating emerging patterns. In: Proceedings of IDEAL'00, pp 48–53