SLIM: Directly Mining Descriptive Patterns

Koen Smets Jilles Vreeken

Advanced Database Research and Modelling
Universiteit Antwerpen
{koen.smets,jilles.vreeken}@ua.ac.be

Abstract

Mining small, useful, and high-quality sets of patterns has recently become an important topic in data mining. The standard approach is to first mine many candidates, and then to select a good subset. However, the pattern explosion generates such enormous amounts of candidates that by post-processing it is virtually impossible to analyse dense or large databases in any detail.

We introduce SLIM, an any-time algorithm for mining high-quality sets of itemsets directly from data. We use MDL to identify the best set of itemsets as that set that describes the data best. To approximate this optimum, we iteratively use the current solution to determine what itemset would provide most gain—estimating quality using an accurate heuristic. Without requiring a pre-mined candidate collection, SLIM is parameter-free in both theory and practice.

Experiments show we mine high-quality pattern sets; while evaluating orders-of-magnitude fewer candidates than our closest competitor, KRIMP, we obtain much better compression ratios—closely approximating the locally-optimal strategy. Classification experiments independently verify we characterise data very well.

1 Introduction

Pattern mining is one of the central topics in data mining, and is focused on discovering interesting local structure in data. Starting from frequent sets and association rules [1], one of the key goals in pattern mining has been completeness: discovering all patterns that satisfy certain conditions. In a way, this is a useful goal, as we know that all returned patterns meet the requirements—and as such, are potentially interesting.

Completeness, however, also has its drawbacks. Typically, there exist extremely many patterns fulfilling the constraints, many of which convey the same information. Mining all patterns hence leads to prohibitively large and strongly redundant results, which are in turn difficult to use or interpret. To address this, researchers have recently instead focused on discovering small and useful, high-quality *sets of patterns* [12, 8, 22].

We identify the best set of patterns as those patterns that together describe the data best. That is, by the Minimum Description Length (MDL) principle [6], we are after that set of patterns that provides the optimal lossless compression of the data. By definition, this set has many desirable properties: it is non-redundant, not overly simple, nor complex for the data—as otherwise, bits could have been saved. The main question is, how do we mine this optimal result?

In this paper, we present SLIM, an efficient heuristic for directly mining high-quality data descriptions on transaction data. SLIM is a fast, one-phase, any-time alternative to KRIMP [17]. Besides obtaining better compression, it can handle large and dense datasets, and is parameter-free in both theory and practice.

The MDL approach to pattern mining was pioneered by Siebes et al. [17, 22], who gave the KRIMP algorithm to approximate the optimal set of itemsets, or, *code table*. Subsequent research showed these sets of patterns to be very useful. Besides characterising the distribution of the data very well, they have been shown to provide high performance in a wide range of tasks, including difference measurement [21], clustering [9], and outlier detection [18].

Like many pattern set mining approaches [7, 2, 16], KRIMP follows a straightforward two-phase approach to mining code tables. First, it mines a collection of frequent itemsets. Second, it considers these candidates in a static order, accepting a pattern if it improves compression. This simplicity has some drawbacks.

First of all, mining candidates is expensive. As more candidates correspond to a larger search space, lower support thresholds correspond to better final results. Often, however, it is difficult to keep the number of candidates feasible—for large and dense databases especially, a small drop of the threshold can lead to an enormous increase in patterns. Moreover, as most candidates will be rejected, this step is quite wasteful.

Second, by considering candidates only once, and in a fixed order, KRIMP sometimes rejects candidates that it could have used later on. A more powerful strategy would be to iteratively select the best addition out of all candidates—which naively, however, quickly becomes infeasible for larger databases or candidate collections.

With Slim, we address these issues, and give an efficient any-time algorithm for mining descriptive pattern sets *directly* from data. We greedily construct pattern sets in a bottom-up fashion, iteratively joining co-occurring patterns such that compression is maximised. To reduce computation and database scans, it employs a simple yet accurate heuristic to estimate the gain or cost of introducing a candidate. Importantly, Slim is parameter-free in both theory and practice.

Experiments show we discover high-quality pattern sets. The Slim code tables attain very high compression ratios; in particular on large and dense datasets, we obtain tens of percentages better compression than Krimp, while considering orders-of-magnitude fewer candidates. Classification experiments show high accuracy, verifying that we characterise the data well. Convergence plots show Slim closely approximates the powerful greedy approach of selecting the best candidate out of all candidates; all while being much more efficient.

The remainder of this paper is organised as follows. First, we cover the preliminaries in Section 2 including notation, and short introductions to, resp. MDL, code tables, and Krimp. Next, in Section 3 we discuss how to mine and identify good candidate patterns. In Section 4 we introduce the Slim algorithm for mining high quality code tables directly from data. Section 5 discusses related work, and we experimentally evaluate our method in Section 6. Finally, we round up with discussion in Section 7 and conclude in Section 8.

2 Preliminaries

In this Section we give the notation used throughout the paper, and provide an introduction to MDL, code tables and the Krimp algorithm.

2.1 Notation In this paper we consider transaction databases. Let $\mathcal{I}$ be a set of items, e.g. the products for sale in a shop. A transaction $t \in \mathcal{P}(\mathcal{I})$ is a set of items that, e.g. represent the items a customer bought. A database $\mathcal{D}$ over $\mathcal{I}$ is then a bag of transactions, e.g. the different sale transactions on a given day. We say that a transaction $t \in \mathcal{D}$ supports an itemset $X \subseteq \mathcal{I}$, iff $X \subseteq t$. The support of X in $\mathcal{D}$ is the number of transactions in the database where X occurs.

Note that any binary or categorical dataset can be trivially converted into a transaction database.

Throughout this paper all logarithms are to base 2, and by convention we use $0 \log 0 = 0$.

2.2 MDL, a brief introduction The Minimum Description Length principle (MDL) [6], like its close cousin MML (Minimum Message Length) [23], is a practical version of Kolmogorov Complexity [10]. All three embrace the slogan *Induction by Compression*. For MDL, this can be roughly described as follows.

Given a set of models $\mathcal{M}$, the best model $M \in \mathcal{M}$ is the one that minimises

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

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

This is called two-part MDL, or *crude* MDL—as opposed to *refined* MDL, where model and data are encoded together [6]. We use two-part MDL because we are specifically interested in the model: the patterns that give the best description. 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 $\mathcal{M}$ are, how a $M \in \mathcal{M}$ describes a database, and how we encode these in bits. Note, that in MDL we are only concerned with code lengths, not actual code words.

2.3 Code tables As our itemset-based models we use code tables [17, 22]. A code table is a simple dictionary: a two-column table with itemsets on the left-hand side, and corresponding codes on its right-hand side. The itemsets in the code table are ordered first descending on cardinality, second descending on support, and third lexicographically. We refer to this as the **Standard Cover Order**.

The actual codes on the right-hand side are of no importance: their lengths are. To explain how these lengths are computed, the coding algorithm needs to be introduced. A transaction t is encoded by searching for the first itemset X in the code table for which $X \subseteq t$. The code for X becomes part of the encoding of t. If $t \setminus X \neq \emptyset$, the algorithm continues to encode $t \setminus X$. Since it is insisted that each code table contains at least all single items, this algorithm gives a unique encoding to each (possible) transaction over $\mathcal{I}$.

The set of itemsets used to encode a transaction is called its *cover*. Note that the coding algorithm implies that a cover consists of non-overlapping itemsets. The length of the code of an itemset in a code table CT depends on the database at hand; the more often a code is used, the shorter it should be. To this end, we use an optimal prefix code. To compute the code lengths, we have to cover every transaction in the database.

The *usage* of an itemset $X \in CT$ is the number of transactions $t \in \mathcal{D}$ which have X in their cover. The relative usage of $X \in CT$ is the probability that X is used in the encoding of an arbitrary $t \in \mathcal{D}$. For optimal compression of $\mathcal{D}$, the higher $\Pr(X \mid \mathcal{D})$, the shorter its code should be. In fact, from Information Theory [4], we have the Shannon entropy, which gives us the length of the optimal prefix code for X as

$$L(X \mid \mathcal{D}) = -\log \Pr(X \mid \mathcal{D}) \ ,$$

where

$$\Pr(X \mid \mathcal{D}) = \frac{usage(X)}{\sum\limits_{Y \in CT} usage(Y)} \ .$$

The length of the encoding of transaction is now simply the sum of the code lengths of the itemsets in its cover,

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

The size of the encoded database is then the sum of the sizes of the encoded transactions,

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

To find the optimal code table using MDL, we need to take both the compressed size of the database, and the size of the code table into account. For the size of the code table, we only consider those itemsets that have a non-zero usage. The size of the right-hand side column is obvious; it is simply the sum of all the different code lengths. For the size of the left-hand side column, note that the simplest valid code table consists only of the single items. This code table we refer to as the **Standard Code Table**, or ST. We encode the itemsets in the left-hand side column using the codes of ST. This allows us to decode up to the names of items. The encoded size of the code table is then given by

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

We define the optimal set of itemsets as the one whose associated code table minimises the total encoded size

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

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 algorithm, and $\mathcal{F}$ a collection of candidate patterns $\mathcal{F} \subseteq \mathcal{P}(\mathcal{I})$. Find the smallest set of patterns $\mathcal{S} \subseteq \mathcal{F}$ such that for the corresponding code table CT the total compressed size, $L(CT, \mathcal{D})$, is minimal.*

Using our cover algorithm only patterns occurring in $\mathcal{D}$ can be used in describing the data. As such, $\mathcal{P}(\mathcal{I})$ is a clear overestimate for what candidates the optimal CT can consist of. For reasons of efficiency, and without loss of generality, we can hence limit $\mathcal{F}$ to the collection of all itemsets in $\mathcal{D}$ with a support of at least 1.

Even for small $\mathcal{F}$, however, the search space of all possible code tables is rather large—and moreover, it does not exhibit structure nor monotonicity we can use to efficiently find the optimal code table [22]. Hence, we resort to heuristics.

2.4 Introducing Krimp The Krimp algorithm was introduced by Siebes et al. [17, 22] as a straightforward approach for mining good approximations of the optimal code table. Subsequent research showed these code tables to indeed be of high quality, and useful for a wide range of data mining tasks [21, 9, 22, 18].

The pseudo-code of Krimp is given as Algorithm 1. It starts with the singleton code table (line 1), and a candidate collection $\mathcal{F}$ of frequent itemsets up to a given *minsup*. The candidates are ordered first descending on support, second descending on itemset length and third lexicographically. Each candidate F is considered in turn by inserting it in CT (3), denoted by $CT \oplus F$, and calculating the new total compressed size (4). It is only accepted if compression improves (4). If accepted, all elements $X \in CT$ are reconsidered wrt. their contribution to compression (5).

The *minsup* parameter is used to control the number of candidates: the lower *minsup*, the more candidates, the larger the search space, and hence the better the approximation of the optimal code table. As by MDL we are after the optimal compressor, *minsup* should be set as low as practically feasible. Hence, technically *minsup* is not a parameter.

In practice, however, keeping the number of itemsets feasible is difficult. Especially for dense or large databases, minute decreases in threshold give rise to enormous increases in patterns. Mining, sorting, and storing these in large numbers takes non-trivial time and effort, even before Krimp can begin selecting.

Its robust empirical results aside, Krimp is a rather rough greedy heuristic: it considers each candidate only once, in a static order, raising the question how good its approximations really are.

A standard approach to hard optimisation problems, with provable approximation bounds in case of sub-modular set functions [15], is to locally optimise the target function. For Krimp, local optimisation translates to iteratively finding the $F \in \mathcal{F}$ that maximally increases compression. By considering $\mathcal{F}$ quadratically instead of linearly, however, the running time quickly

Algorithm 1 The KRIMP Algorithm [22]

```
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. for $F \in \mathcal{F}$ in Standard Candidate Order do
3.   $CT_c \leftarrow (CT \oplus F)$ in Standard Cover Order
4.   if $L(\mathcal{D}, CT_c) < L(\mathcal{D}, CT)$ then
5.     $CT \leftarrow$ post-prune($CT_c$)
6.   end if
7. end for
8. return $CT$
```

grows out of hand. In Section 6 we will refer to this variant as KRAMP. While there are no results on submodularity for code tables, and hence no approximation bounds, we can use it as a gold-standard to compare to.

3 Identifying good candidates

Instead of filtering pre-mined candidates, we rather mine code tables directly from data. To do so, we need to be able to identify good candidates on the fly.

3.1 Covering Data Intuitively, SETCOVER seems like a suited approach for approximating the optimal code table. Its goal is to find the smallest set of itemsets that cover all 1s in the data. This problem is NP-hard, but good approximation bounds exists for the greedy approach. That approach, also known as Tiling [5], iteratively finds that itemset (or, tile) by which the most uncovered 1s in the data are covered.

In practice, however, Tiling does not mine good code tables. Whereas KRIMP refines its cover by replacing general patterns with more specific ones, Tiling only focuses on covering data; quickly leading to the selection of overly general (individual items), or overly specific itemsets (complete transactions), that do not contribute towards good compression.

Interestingly, though, we do see that the first few discovered tiles typically do compress well. This is especially interesting, as we further note that both methods initially regard the data similarly: at first, every 1 in the data corresponds to a separate code, and hence, covering many 1s with one tile means replacing many codes with one code. This suggests finding tiles may be worthwhile, if we consider the right search space.

(Note that as in our problem itemset costs develop non-monotonically with changes of CT, the weighted variant of SETCOVER does not apply trivially.)

3.2 Covering Codes Whereas Tiling only regards the uncovered part of the 0/1 data matrix, in our problem we always consider complete covers. That is, the introduction of a new code table element X into CT induces a different covering of the data—for which we determine its quality by $L(CT, \mathcal{D})$.

We can regard the cover of a database $\mathcal{D}$ by a code table CT as follows. Let $\mathcal{C}$ be the $|\mathcal{D}|$-by-$|CT|$ binary matrix, where the rows correspond to transactions $t \in \mathcal{D}$ and the columns to elements $X \in CT$. The cell values are 1 when $X \in \mathit{cover}(t)$, and 0 otherwise.

In this cover matrix, or *cover space*, an itemset XYZ represents a group of code table elements $X, Y, Z \in CT$ that co-occur in the covers of transactions in $\mathcal{D}$. As such, *frequent* itemsets in $\mathcal{C}$ make for good code table candidates: as instead of writing multiple codes together many times, we can gain bits if we can write one short code for their combination instead.

Introducing (or removing) an element X to CT changes the cover matrix, however, both in numbers of columns, and values. As the co-occurrences change, so does what makes a good addition to the current CT. This suggests that to find good candidates, we have to iteratively search for frequent itemsets in cover space, updating for every change in CT.

3.3 Estimating candidate quality As simple an observation it is, mining good itemsets in cover space to find good code table candidates is the key idea of this paper. In the remainder, we will show that by iteratively considering such itemsets, we can optimise compression far beyond KRIMP, and do this much more quickly too. Before we can do so, we first have to discuss how to measure the quality of a candidate.

As we are optimising compression, the quality of a candidate X essentially is the gain in total compression $L(CT, \mathcal{D})$ we obtain when we add X to CT. The most straightforward approach is to calculate these gains for every candidate and then to select the best one. As such, every candidate X to be considered corresponds to the combination of code table elements identified by an itemset Y in $\mathcal{C}$. The items in Y are in fact indexes to elements in CT, i.e. $Y \subset \{1, \ldots, |CT|\}$. Hence, X is simply the union of the code table elements $X_i \in CT$ identified by the $i \in Y$, i.e. $X = \bigcup_{i \in Y} X_i$.

Next, once we know the gains in compression for every candidate, we simply locally optimise and accept that X into CT which maximises compression. Although powerful, this is expensive, since to calculate the gain for a candidate, we have to cover the database.

Instead, we can *estimate* the gain first, and only calculate exact gain for the best estimated candidate. To maximise compression, we observe we want candidates

1) with high *usage* (i.e. short codes), 2) replacing many codes in $\mathcal{C}$, while 3) not adding much complexity. The first two of these properties are simply approximated by finding frequent itemsets in $\mathcal{C}$.

However, we can say more on what itemsets we want: highly frequent sets of only few items. Besides that these sets will likely compress well, they add little complexity to CT. However, keeping in mind that by iteratively refining, if we consider too large itemsets in $\mathcal{C}$, we strongly reduce our search space and are more likely to end up in a local minimum. Moreover, calculating the gain in compression is most accurate for itemsets of length 2. Hence, we restrict the cardinality of the itemsets we find in $\mathcal{C}$ to length 2. Note however, that more specific code table elements, i.e. large itemsets in $\mathcal{D}$, can (and are) be constructed in only few iterations.

Suppose X and Y are itemsets in CT. Let CT' be the code table after adding the union of these two itemsets, i.e. $CT' = CT \oplus X \cup Y$. We can estimate the *usage* of $X \cup Y$ in CT' as

$$|usage(X) \cap usage(Y)| ,$$

where by $usage(X)$ we slightly abuse notation to refer to the *tid* list of transactions with X in their cover. Note this gives an upper bound to $usage(X \cup Y)$, where equality only holds if $X \cup Y$ is considered for covering right before when X or Y would be used. As code table elements are ordered first on length, and the union of two non-equal itemsets produces a longer set, this is implicitly enforced—making the bound quite tight.

Although useful, as a gain estimate it is quite rough: it disregards the effects on $L(CT \mid \mathcal{D})$ of adding $X \cup Y$ to CT, the increase of the code lengths of X and Y as their *usage* decreases, as well as the scaling effect on all other code lengths. We can improve our estimate by taking these effects into account as follows, directly estimating the total compressed size for adding a candidate.

Let ΔL be the difference in encoded size between CT and CT'; or, in other words, the gain in bits for candidate $X \cup Y$. It is easy to see that ΔL for the total encoded size consists of the difference in compressed sizes of the database, and the difference between the encoded sizes of the code tables,

$$\begin{aligned}&\Delta L(CT \oplus X \cup Y, \mathcal{D}) \\ &\quad = L(CT, \mathcal{D}) - L(CT \oplus X \cup Y, \mathcal{D}) \\ &\quad = \Delta L(\mathcal{D} \mid CT \oplus X \cup Y) + \Delta L(CT \oplus X \cup Y \mid \mathcal{D}) .\end{aligned}$$

For notational brevity, we use the lower case, x, of an itemset $X \in CT$ to indicate $usage(X)$, i.e. $x = usage(X)$. Then, let s be the sum of usages of CT, i.e. $s = \sum_{X \in CT} x$. Similarly, we use x' and s' for CT'. Using this notation, we can write the difference in bits between encoding $\mathcal{D}$ by either CT or CT' as follows,

$$\begin{aligned}\Delta L(\mathcal{D} \mid CT \oplus X \cup Y) = s \log s - s' \log s' + xy' \log xy' \\ - \sum_{\substack{C \in CT \\ c \neq c'}} (c \log c - c' \log c') ,\end{aligned}$$

and the difference in the model complexity as

$$\begin{aligned}&\Delta L(CT \oplus X \cup Y \mid \mathcal{D}) = \log xy' - L(X \cup Y \mid ST) \\ &\quad + |CT| \log s - |CT'| \log s' + \sum_{\substack{C \in CT \\ c' \neq c \\ c' c \neq 0}} \log c' - \log c \\ &\quad + \sum_{\substack{C \in CT \\ c' \neq c \\ c = 0}} \log c' - L(C \mid ST) + \sum_{\substack{C \in CT \\ c' \neq c \\ c' = 0}} L(C \mid ST) - \log c ,\end{aligned}$$

where $xy' = usage(X \cup Y)$ when using CT'. This means that we can express the gain in bits when adding $X \cup Y$ to CT only in terms of the usages of code table elements for which the usage changes between CT and CT'.

In practice, however, it is hard to predict how $usage(Z)$ for all $Z \in CT$ will develop when a new element $X \cup Y$ is inserted into CT. While we know the usages of $Z \in CT$ ordered above $X \cup Y$ remain static, a cascading effect may occur for those below it: in some $t \in \mathcal{D}$, where some $Z \in CT$ with $Z \cap (X \cup Y) \neq \emptyset$ was previously part of the cover of t, $X \cup Y$ may now prevent Z from being used; hence changing *usage* of Z, as well as the *usage* of those Z' now used to cover $Z \setminus (X \cup Y)$ of t, which in turn can block other previously used elements, etc.—potentially changing all usages.

Although unpredictable, in practice the effect is not often dramatic. Hence, for practical reasons, for estimating the gain we can assume only the *usage* of X, Y, and $X \cup Y$ will change, using our above usage estimate for $X \cup Y$. For calculating the estimated gain in total compressed size, we then simply use $xy' = |usage(X) \cap usage(Y)|$, $x' = x - xy'$, $y' = y - xy'$ and $s' = s - xy'$. As such, we obtain a very easily calculable, and generally accurate estimate of ΔL for $X \cup Y$.

4 Directly Mining Descriptive Patterns

We can now combine the above results, and construct the Slim algorithm for mining heuristic solutions to the Minimal Coding Set Problem directly from data.[1]

We give the pseudo-code as Algorithm 2. Slim starts with the singleton-only code table ST (line 1). Every iteration (2) we consider all pairwise combinations of $X, Y \in CT$ as candidates in **Gain Order**, i.e. descending on $\Delta L(CT \oplus (X \cup Y), \mathcal{D})$. Iteratively, we add a candidate to CT in **Standard Cover Order**

[1]Slim is Dutch for *smart*, whereas Krimp means *to shrink*.

Algorithm 2 The Slim Algorithm

```
Input: A transaction database $\mathcal{D}$ 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. for $F \in \{X \cup Y : X, Y \in CT\}$ in Gain Order do
3.   $CT_c \leftarrow (CT \oplus F)$ in Standard Cover Order
4.   if $L(\mathcal{D}, CT_c) < L(\mathcal{D}, CT)$ then
5.     $CT \leftarrow$ post-prune($CT_c$)
6.   end if
7. end for
8. return $CT$
```

(3), cover the data, and compute total encoded size (4). If compression improves, we accept the candidate, otherwise reject it. If accepted, we reconsider every element in CT to whether it still contributes towards compression (5), and update the candidate list (2). We continue considering pairwise combinations of $X, Y \in CT$ to refine the current code table until no candidate decreases the total compressed size, after which we are done.

Note that, if desired, extra constraints on individual candidates (e.g. a minimum support, or length) can be checked when constructing the candidate list, or before adding them to the code table at line 3.

In practice, we do not need to materialise all candidates on line 2. Instead, we traverse CT ordered on usage, employing branch-and-bound to find the $X \cup Y$ with highest estimated gain; as we traverse the elements descending on usage, we do not need to consider any element V or W with lower usage than the current best candidate $X \cup Y$. Moreover, suppose X is considered before Y. Therefore $\mathit{usage}(Y) \leq \mathit{usage}(X)$, and we can first bound using $\mathit{usage}(X \cup Y) = \mathit{usage}(X)$. Second, we can bound using $\mathit{usage}(X \cup Y) = \mathit{usage}(Y)$. Then, if this bound is met, we need to calculate the expected usage $\mathit{usage}(X \cup Y)$ by intersecting the usage lists of X and Y. Moreover, to speed up computation, we store the top-k best estimates, allowing us to quickly suggest the next-best candidate when a candidate is rejected.

Slim is well-suited for any-time computation, as it iteratively refines the current code table. As such, it allows for interactive data analysis and time-budgeted computation, providing good intermediate results. Given a result, Slim can simply continue refining.

Next, we analyse the complexity of Slim. Considering the candidates (line 2) maximally takes $O(|CT|^2)$ steps. A code table for $\mathcal{D}$ could contain all $|\mathcal{F}|$ itemsets occurring in $\mathcal{D}$. At worst, we re-evaluate each candidate $|\mathcal{F}|$ times. The complexity of steps 3–6 is $O(|\mathcal{F}| \times |\mathcal{D}| \times |\mathcal{I}|)$. Together, the worst-case time-complexity is $O(|\mathcal{F}|^3 \times |\mathcal{D}| \times |\mathcal{I}|)$. We will see in the experiments this estimate is quite pessimistic; in practice, code tables are small (ranging from 10s to 1000s) and Slim evaluates 2 orders-of-magnitude fewer candidates than all $|\mathcal{F}|$ itemsets occurring in $\mathcal{D}$.

Regarding memory complexity we can be brief. As we need to store a code table of maximally $|\mathcal{F}|$ itemsets, and the database $\mathcal{D}$, memory complexity is $O(|\mathcal{F}| + |\mathcal{D}|)$.

5 Related Work

Since the seminal paper by Agrawal and Srikant [1] on frequent pattern mining, a lot of research is aimed at reducing the pattern explosion; mining the most interesting and useful patterns in manageable amounts. The traditional approach is to mine concise representations for collections of patterns, either lossless, such as non-derivable [3] itemsets, or lossy, as for self-sufficient itemsets [26] and probabilistic summaries [27]. This is different from our approach in that we summarise data, instead of pattern collections.

Webb argues [25] not to condense set of mined patterns, but to rank patterns according to their statistical significance, and let the end-user consider the top-k. However, as patterns are considered individually, redundancy among significant patterns remains.

Considering patterns as binary features on rows, Knobbe and Ho [7], and Bringmann and Zimmermann [2], resp. exhaustively and heuristically select those groups of patterns by which data rows are partitioned optimally, using an external criterion such as joint-entropy or accuracy. Unlike Slim, both post-process materialised collections of candidate patterns, and partition the data instead of summarising it.

Tilings [5] are sets of itemsets that together efficiently cover the data, and are hence strongly related to SetCover. Although tilings can be mined directly from data, as area is not (anti-)monotonic with set inclusion, efficiency is an issue. Related, Kontonasios and De Bie [8] propose a two-phase approach to select the most informative noisy tiles from a collection of fault-tolerant itemsets, using MDL and a maximum entropy data model. Both methods require a number of patterns to select, as well as a minimum area threshold.

Slim is strongly related to the Krimp algorithm [22]. Both aim at finding the set of itemsets that together describe the data best. Krimp code tables have been shown to capture data distributions very well, and have been used successfully for a wide range of data mining tasks [21, 9, 22]. Krimp post-processes a candidate collection, filtering it in a static order. By iteratively considering the current data description, Slim avoids redundant patterns, explores a larger search space, does not mine and sort huge numbers of candidates, and does not require a minimal support threshold.

Recently, Siebes and Kersten [16] proposed the GROEI algorithm for finding the best k-element KRIMP code table by beam search. By considering a much larger search space, they improve over KRIMP at expense of efficiency—for beam-width 1 it coincides with KRAMP—and hence only small datasets are considered. Although beyond the scope of this paper, the SLIM search strategy and estimation heuristics can likely speed up GROEI significantly.

The PACK algorithm [19] makes a connection between decision trees and itemsets, mines good decision trees, and returns the itemsets that follow from these. It can mine its trees either from a candidate itemset collection, or directly from the data. Unlike here, PACK considers the data 0/1 symmetrically. As a result, it requires fewer bits, but returns many more itemsets.

Wang and Partharsarthy [24] incrementally build probabilistic models for predicting itemset frequencies. Iteratively they update the model by those itemsets for which the estimate deviated more than the threshold. For efficiency, itemsets are considered in level-wise batches. Mampaey et al. [11] propose a convex heuristic to efficiently iteratively find the least-well predicted itemset overall, using BIC to control complexity. As constructing and querying maximum entropy models is computationally very expensive, only high-level summaries can be mined. Furthermore, by only considering frequency, co-occuring patterns cannot be detected [8].

6 Experiments

Here we experimentally evaluate SLIM, its heuristics, and the quality of the discovered code tables.

6.1 Setup We implemented a prototype in C++, and provide the source code for research purposes[2].

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

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

wherever $\mathcal{D}$ is clear from context.

As candidates $\mathcal{F}$ for KRIMP, we use all frequent itemsets mined at the *minsup* thresholds depicted in Table 1. These thresholds were chosen as low as feasible, while making the processing finish within 24 hours. KRIMP includes one of the fastest miners from the FIMI repository [22]. Timings reported for KRIMP include mining and sorting of the candidate collections.

All experiments were conducted as single-threaded runs on Linux machines with Intel Xeon X5650 processors (2.66GHz) and 12 GB of memory.

[2]`http://adrem.ua.ac.be/implementations/`

6.2 Datasets We consider a wide range of benchmark and real datasets. The base statistics for each are depicted in Table 1. We show for each database the number of attributes, number of transactions, the density (relative number of 1s), and the number of classes.

From the LUCS/KDD dataset repository we take some of the largest and most dense databases. From the FIMI repository we use the *Accidents*, *BMS* and *Pumsb* datasets. The *Chess (kr–k)* and *Plants* datasets were obtained from the UCI repository.

We use the real *Mammals* presence and *DNA Amplification* databases. The former consists of presence records of European mammals[3] within geographical areas of 50×50 kilometers [13]. The latter contains DNA copy number amplifications. Such copies activate oncogenes and are hallmarks of advanced tumours [14].

From the Antwerp University Hospital we obtained *MCADD* data [18]. Medium-Chain Acyl-coenzyme A Dehydrogenase Deficiency (MCADD) is a deficiency all newborns are screened for [20].

The *Abstracts* dataset contains the abstracts of all accepted papers at the ICDM conference up to 2007, where words are stemmed and stop words removed [8].

6.3 Compression First we investigate how well SLIM describes data. In Table 1 we give the relative compression rates ($L\%$) for both SLIM and KRIMP. To ease interpretation we also plot these in Figure 1. The shorter the bars in the plot, the shorter the discovered descriptions of the data.

Overall, we see SLIM outperforms KRIMP with an average gain in compression ratio of 11%, up to over 50% for *Pumsb*. The only exception is *BMS-pos*, for which both methods fail to find succinct descriptions.

Analysing these results in more detail, we see that, unsurprisingly, the largest improvements are made on the large and/or dense datasets, such as *Accidents*, *Connect-4*, and *Ionosphere*. By their characteristics, these datasets give rise to very large numbers of patterns, and hence can only be considered by KRIMP if we set *minsup* relatively high—implicitly limiting the detail at which KRIMP can describe the data.

The ability to compress a dataset depends on the amount of recognisable structure. For sparse datasets, like *Abstracts*, and the two *BMS* datasets, neither SLIM nor KRIMP can identify much structure, whereas SLIM can describe *Pumsb(star)* quite succinctly by considering lower-frequency itemsets. For dense data, such as *Chess (k-k)*, *Connect-4*, and *Mushroom*, both algorithms ably find structure.

[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 1: Overview statistics.

	General statistics				Krimp				Slim		
Dataset	$\|\mathcal{D}\|$	$\|\mathcal{I}\|$	%1s	$\|cl\|$	*minsup*	$L\%$	$\|CT\|$	$\|\mathcal{F}\|$	$L\%$	$\|CT\|$	$\|\mathcal{F}\|$
Abstracts	859	3933	1.2	-	3	91.9	1102	6 M	**84.3**	1815	13 M
Accidents	340183	468	7.2	-	50000	55.1	1583	3 M	**31.1**	2018	21 k
Adult	48842	97	15.3	2	1	24.4	1303	58 M	**22.8**	1201	199 k
BMS-pos	515597	1657	0.4	-	100	**81.7**	9478	6 M	83.1	1964	15 k
BMS-webview 1	59602	497	0.5	-	35	85.9	718	1 M	**84.0**	965	2 M
Chess (k-k)	3196	75	49.3	2	500	30.0	275	846 M	**14.7**	292	20 k
Chess (kr-k)	28056	58	12.1	18	1	61.6	1684	373 k	**57.5**	1060	28 k
Connect-4	67557	129	33.3	3	40000	42.9	56	24 M	**12.3**	1670	297 k
DNA amplification	4590	391	1.5	-	9	36.7	326	312 M	**35.7**	359	67 k
Ionosphere	351	157	22.3	2	35	59.8	170	226 M	**49.7**	240	294 k
Letter recognition	20000	102	16.7	26	1	35.7	1780	581 M	**33.4**	1599	521 k
Mammals	2183	121	20.5	-	200	48.1	316	94 M	**39.9**	434	235 k
MCADD	31924	198	11.1	2	35	55.4	2280	2 M	**51.0**	4067	924 k
Mushroom	8124	119	19.3	2	1	20.5	442	6 G	**18.5**	340	16 k
Pen digits	10992	86	19.8	10	1	42.2	1247	459 M	**39.4**	1347	394 k
Plants	34781	70	12.4	-	2000	46.4	511	913 k	**36.1**	840	179 k
Pumsb	49046	2113	3.5	-	35000	70.0	175	2 M	**19.1**	3299	151 k
Pumsbstar	49046	2088	2.4	-	12500	56.0	331	2 M	**25.1**	4274	383 k
Waveform	5000	101	21.8	3	5	44.5	921	466 M	**39.0**	734	134 k

The first four columns provide general statistics per dataset; number of transactions, attributes, density (in percentage of 1s) and number of classes (if any). For Krimp we give relative compression ($L\%$), number of non-singleton code table elements, and number of candidates for the given *minsup* threshold. For Slim we give relative compression ($L\%$), number of non-singleton code table elements, and number of evaluated candidates.

When we look at the number of (non-singleton) itemsets in CT, we see very similar results. In general, for both Slim and Krimp, depending on the data, the code tables contain between a hundred up to a few thousand itemsets. Three datasets, *Connect-4* and *Pumsb-(star)*, stand-out, with Slim returning 10 times more patterns. However, for these Krimp can only run with very high *minsup*—whereas Slim does not have this restriction, and can better capture the structure by using more fine-grained patterns.

6.4 Greedy vs. Greedy Next, we compare Slim to two variants of the powerful standard greedy optimisation algorithm for complex combinatorial problems. Kramp is a variant of Krimp that in each iteration chooses that F out of all $\mathcal{F}$ that locally maximises compression. Analogously, Slam is a variant of Slim calculating exact compression gain to order the candidates at every iteration—instead of estimating it heuristically.

Clearly, Slam and Kramp are computationally expensive. Hence, we first compare on some small, well-known UCI benchmark datasets: *Anneal, Breast, Heart, Iris, Led7, Nursery, Page blocks, Pima, Tic-tac-toe* and *Wine*.

All these datasets are easily mined and processed using a *minsup* threshold of 1. Comparing the total compressed sizes, we observe that Slam is always ranked best, Slim second, Kramp third and Krimp last, with an average relative compression ($L\%$) of respectively 36.5, 36.8, 37.8 and 40.1. Note that although Kramp considers the largest search-space, it does not always obtain the best result. In the remainder, we do not consider these datasets further.

When we consider some larger databases, i.e. *Adult, Chess (k-k), DNA*, and *Letter recog.*, we see the same pattern: Slam obtains the best average $L\%$ of 26.5, Slim a close second at 26.7, and Krimp is behind with 31.7—Kramp does not finish within reasonable time.

Last, using *Adult* as a typical example, we plot the development of $L\%$ per itemset accepted into CT as Figure 2. As the plot shows, Slim closely follows Slam and Kramp, quickly converging to good compression—much more directly than Krimp. Note that as itemsets can be pruned from CT, the final x-coordinate does not necessarily match $|CT|$ in Table 1. Further, we note that while Slim only needs 35 minutes to converge, Slam requires one week, and Kramp two months. We will discuss convergence and runtime in more detail below.

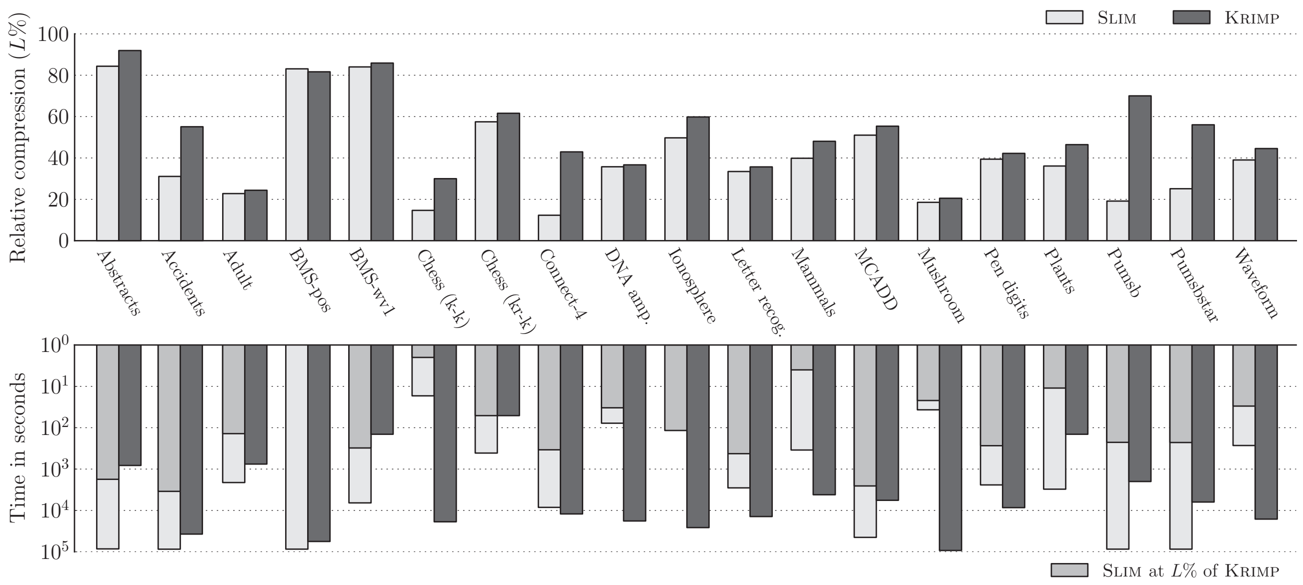
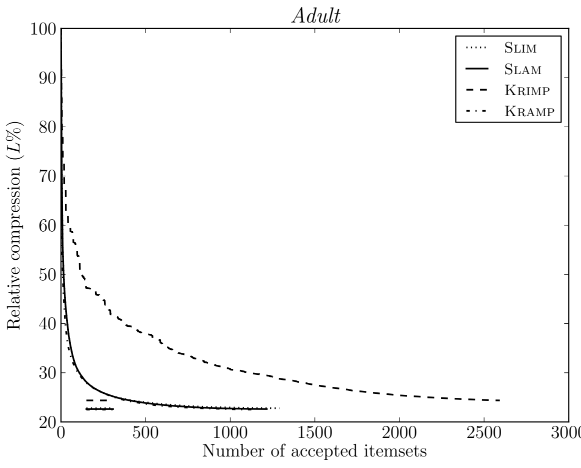


Figure 1: Comparing Slim and Krimp: relative compression ($L\%$) (top), overall running time in seconds (bottom), and time needed by Slim to reach the compression attained by Krimp (mid-grey). Note that besides obtaining better compression, the Slim prototype is nearly always faster than the optimised Krimp implementation.



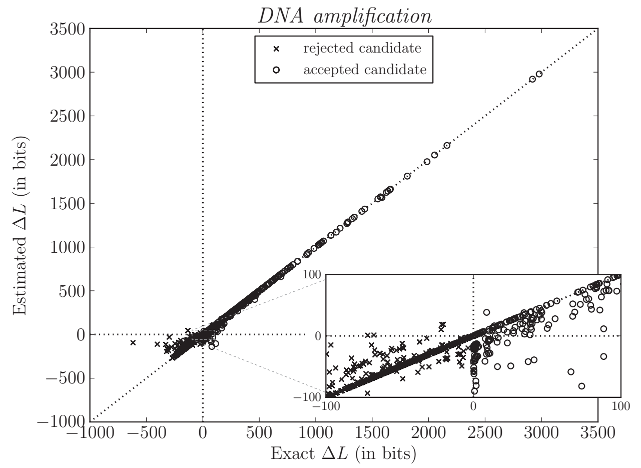


Figure 2: Convergence plot for *Adult*. The lines in lower left corner show the final attained relative compression.

Figure 3: Correlation plot of the estimated versus exact ΔL (in bits) on the *DNA amplification* dataset.

6.5 Number of Candidates Next, we compare the number of instantiated candidates $|\mathcal{F}|$, shown in Table 1. This is the number of itemsets for which we calculate the total compressed size by covering the data. For Krimp this equals the number of frequent itemsets at the listed *minsup* threshold. For Slim this reflects the number of materialised unions of code table elements. As Table 1 shows, Slim evaluates 2 orders-of-magnitude fewer candidates. In general, Slim considers between 10 thousand and 10 million itemsets, whereas Krimp processes millions to billions of itemsets.

Only for *Abstracts* and *BMS-webview 1* Slim evaluates more candidates than Krimp—datasets for which a minute decrease in *minsup* leads to an explosion of candidates. As such, also for this sparse data Slim can consider candidates at lower support than Krimp can handle, while for the other datasets Slim only requires a fraction to reach better compression.

6.6 Timings and convergence We now inspect run-times and convergence. We plot the total wall-clock running time for Slim and Krimp as the bottom bar plot of Figure 1. For Krimp (darkest bars), this includes mining frequent itemsets, sorting, and selecting from them. For Slim we mark two timestamps: in mid-grey we show the time it takes Slim to overtake Krimp; the light bars display the time to convergence, with a maximum run-time of 24 hours.

First we inspect how long Slim requires to match the compression of Krimp. We see that for 16 datasets Slim reaches this point faster than Krimp—in fact, several orders-of-magnitude faster for many of these, particularly for dense datasets. For *BMS-pos*, the bar is not shown, as Slim does not reach the same compression. For *Ionosphere* the bar is simply not visible, as Slim overtakes Krimp in less than one second. As such, Slim is generally much faster than Krimp in obtaining Krimp-level descriptions.

Second, we compare overall runtime. We see Slim is still faster than Krimp for 9 datasets, including huge improvements for *Chess (k-k)*, *DNA amplification*, *Ionosphere* and *Mushroom*. For the other datasets, the current implementation requires more time, yet it acquires much more succinct descriptions.

To inspect these cases, we again consider Figure 2. By optimising (estimated) gain the compressed size converges much faster for Slim than for Krimp. After the early large gain, the line quickly levels. Although not converged, now only very few bits are gained per candidate. This goes general, where for *Pumbsb(star)* runtime is further increased by the relatively large code table. As Slim is worst-case quadratic in $|CT|$, the tail of convergence is where most time is invested: CT grows, while few candidates can be pruned. Caching evaluations between iterations will likely speed up the implementation. A more efficient encoding would make selection more strict, providing better descriptions *and* do away with the small-gain candidates.

In Figure 3 we plot the correlation between our estimate of ΔL to the actual gain in total compressed size. We see the two show strong correlation, as most of the measurements lie on the diagonal, especially for gain (upper right quadrant). Moreover, we see almost no false positives (upper left quadrant), and only few examples where few bits are gained while we estimated small loss (lower right quadrant). This strong correlation explains why the convergence of Slim and Slam are similar in Figure 2.

6.7 Classification Above, we saw Slim describes data more succinct than Krimp. We here independently validate how well it captures data distributions by classification: Krimp showed performance on par with 6 state-of-the-art classifiers on a wide range of datasets [22]. If Slim performs at least as well, we can say it mines code tables at least as characteristic for the data.

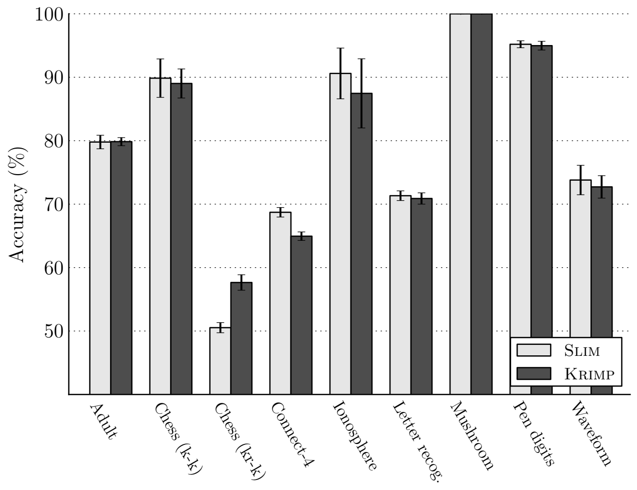


Figure 4: Comparing accuracy on classification tasks.

For this, we reuse the simple classification scheme based on code tables [22]. To use it, we need a code table per class. To build those we split the database according to class, after which the class labels are removed from all transactions. We then apply Slim and Krimp to each of these class-databases, resulting in a code table per class. When the compressors have been constructed, classifying a transaction t is trivial: simply assign the class label belonging to the code table providing the minimal encoded length for t.

We measure performance by accuracy, the percentage of true positives on the test data. All reported results have been obtained using 10-fold cross-validation. Note that we do not maximise accuracy by choosing good pairings between intermediately reported code tables [22], but simply use the final code tables.

We give the accuracy scores as Figure 4. We observe high similarity between Slim and Krimp, and that on average Slim performs slightly better. A pair-wise Student t-test reveals that only the results on the *Connect-4* and *Chess (kr-k)* differ significantly, at significance levels lower than 0.1%, respectively in favour of Slim and Krimp. The performance for *Chess (kr-k)* is due to Slim much better describing the large classes, whereas the more general Krimp code tables balance better against those for the small classes.

6.8 Outlier Detection To see whether Slim is as useful as Krimp, we consider a case study on detecting carriers of MCADD, a rare metabolic disease [18].

Previous work [18] discusses how compression can be used to identify cases that differ from the norm. Informally said, by building a compressor on a database, we can decide whether a sample is an outlier by its encoded length: if more bits are required than expected, the sample is likely an outlier. Experiments show Krimp performs on par with the best outlier detection methods for binary and categorical data [18].

Experiments show Slim provides similar performance to Krimp: all 8 positive cases are ranked among the top-15 using the Slim code tables, and the obtained performance indicators (100% sensitivity, 99.9% specificity and a positive predictive value of 53.3%) correspond with the state-of-the-art results [20].

The highly-ranked non-MCADD cases show combinations of attribute-values very different from the general population, and are therefore outliers by definition. Analysing the encoding of normal cases reveals that Slim code tables correctly identify attribute-value combinations commonly used in diagnostics by experts [20].

6.9 Manual Inspection Finally, we subjectively evaluate the selected itemsets. To this end, we take ICDM *Abstracts* dataset. Considering the top-k itemsets with highest *usage* we see Slim and Krimp provide highly similar results: both provide patterns related to topics in data mining—e.g. 'mine association rules [in] databases', 'support vector machines (SVM)', 'algorithm [to] mine frequent patterns'.

One can argue, however, that experts should not only look at the top-ranked itemsets, as the patterns are selected together to describe the data. When we browse the code tables a whole, we see Slim and Krimp select roughly the same patterns of 2 to 5 items. As Krimp only considers patterns of at least *minsup* occurrences, in this data it does not consider more specific itemsets, whereas Slim may consider any itemset in the data.

For this dataset, Slim selects a few highly specific patterns, such as *'femal, ecologist, jane, goodal, chimpanze, pan, troglodyt, gomb, park, tanzania, riplei, stuctur'*. Inspection shows these itemsets all represent small groups of papers sharing (domain) specific terminology—an application paper in this case—that is not used in any of the other abstracts. As such, these itemsets make sense from both interpretation, and MDL perspective; since the likelihood of these words is very low, yet they strongly interact, and hence can best be described using a single (rare) pattern, saving bits by not requiring codes for the individual rare words.

Note however, that if such level of detail is not desired, a domain expert can prune the search space by specifying additional constraints, e.g. by specifying *minsup*, on the candidates Slim may consider.

7 Discussion

The experiments show Slim finds sets of itemsets that describe the data well, characterising it in detail. In particular on large and dense datasets, Slim code tables obtain tens of percents better compression ratio than Krimp. Classification results show high accuracy, verifying that high-quality pattern sets are discovered.

Dynamically reconsidering the set of candidates while traversing the space leads to better compression. In particular, Slim closely approximates the expensive greedy algorithms Kramp and Slam, that select the best candidate of all current candidates. By employing a branch-and-bound strategy using an efficient and tight heuristic, Slim is much more efficient. The good convergence adds to its any-time property, providing users good results within a time-budget, while allowing further refinements given more time.

Slim generally evaluates two orders-of-magnitude fewer candidates than Krimp, while obtaining more succinct descriptions. Moreover, although the Krimp implementation is optimised, the prototype implementation of Slim is often quicker too. Slim can be optimised in many ways. For example, we currently do not re-use information from previous iterations; nor do we cache whether items co-occur at all. Furthermore, Slim is trivial to parallelise, including parallel branch-and-bound search, evaluation of top-ranked candidates, and covering the data.

Although the exact effects on *usage* of a change in CT can only be calculated by covering the data, better estimates of ΔL may lead to a further decrease in the number of evaluated candidates. Moreover, if the estimate would be sub modular, efficient optimisation strategies with provable bounds could be employed [15].

Our main goal is to provide a faster alternative to Krimp that can operate on large and dense data, while finding even better sets of patterns. Both for describing succinctly, and classification we have seen Slim is indeed at least as good as Krimp. Further research needs to verify whether Slim can also match or surpass Krimp on other data mining tasks [21, 9, 18].

Slim, Slam and Kramp all spend much of their time searching for candidates that only lead to minute improvements in compression. A natural heuristic to stop search early, without much expected loss of quality, would be to stop as soon as the gain estimate becomes negative or when a finite difference approximation of the derivative of $L\%$ indicates we reach a plateau. Another option is to make selection more strict by refining the encoding model.

Although beyond the scope of this paper, the encoding model can be improved in several ways. First, by allowing overlap during covering we can likely gain compression, although covering becomes more complex. Also, dynamic codes could be used; taking into account what itemsets can or cannot be used to encode the remainder of t. These changes will make selection more strict, increasing convergence, possibly avoiding small-gain candidates, yet will make encoding more complex. Future work includes investigating whether this indeed leads to better, i.e. more useful, code tables.

8 Conclusion

In this paper, we introduced SLIM, an any-time algorithm for mining small, useful, high-quality sets of patterns directly from data. We use MDL to identify the best set of itemsets as that set that describes the data best. To approximate this optimum, we iteratively consider what refinement provides most gain—estimating quality using a light-weight and accurate heuristic. Importantly, SLIM is completely parameter-free.

Experiments show SLIM discovers high-quality pattern sets, resulting in high compression rates and high accuracy scores. Furthermore, SLIM closely approximates the common greedy approach of selecting the best candidate overall, while being several orders faster.

There are at least two directions we will explore in future work. Firstly, extending the compression model to allow overlap and use smarter codes. Secondly, optimising our code to analyse even larger data.

Acknowledgements

The authors thank i-ICT of Antwerp University Hospital (UZA) for providing data and expertise on MCADD. Koen Smets and Jilles Vreeken are respectively supported by a Ph.D. and Postdoctoral Fellowship of the Research Foundation – Flanders (FWO).

References

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, pp. 307–328. AAAI/MIT Press, 1996.

[2] B. Bringmann and A. Zimmermann. The chosen few: On identifying valuable patterns. In *Proc. ICDM*, pp. 63–72, 2007.

[3] T. Calders and B. Goethals. Non-derivable itemset mining. *Data Min. Knowl. Disc.*, 14(1):171–206, 2007.

[4] T. Cover and J. Thomas. *Elements of Information Theory.* Wiley-Interscience New York, 2006.

[5] F. Geerts, B. Goethals, and T. Mielikäinen. Tiling databases. In *Proc. DS*, pp. 278–289, 2004.

[6] P. D. Grünwald. *The Minimum Description Length Principle.* MIT Press, 2007.

[7] A. Knobbe and E. Ho. Pattern teams. In *Proc. PKDD*, pp. 577–584, 2006.

[8] K.-N. Kontonasios and T. De Bie. An information-theoretic approach to finding noisy tiles in binary databases. In *Proc. SDM*, pp. 153–164, 2010.

[9] M. van Leeuwen, J. Vreeken, and A. Siebes. Identifying the components. *Data Min. Knowl. Disc.*, 19(2):173–292, 2009.

[10] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications.* Springer, 1993.

[11] M. Mampaey, N. Tatti, and J. Vreeken. Tell me what I need to know: Succinctly summarizing data with itemsets. In *Proc. KDD*, pp. 573–581, 2011.

[12] T. Mielikäinen and H. Mannila. The pattern ordering problem. In *Proc. PKDD*, pp. 327–338, 2003.

[13] A. Mitchell-Jones, G. Amori, W. Bogdanowicz, B. Krystufek, P. H. Reijnders, F. Spitzenberger, M. Stubbe, J. Thissen, V. Vohralik, and J. Zima. *The Atlas of European Mammals.* Ac. Press, 1999.

[14] S. Myllykangas, J. Himberg, T. Böhling, B. Nagy, J. Hollmén, and S. Knuutila. DNA copy number amplification profiling of human neoplasms. *Oncogene*, 25(55):7324–7332, 2006.

[15] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functions—I. *Math. Program.*, 14:265–294, 1978.

[16] A. Siebes and R. Kersten. A structure function for transaction data. In *Proc. SDM*, pp. 558–569, 2011.

[17] A. Siebes, J. Vreeken, and M. van Leeuwen. Item sets that compress. In *Proc. SDM*, pp. 393–404, 2006.

[18] K. Smets and J. Vreeken. The Odd One Out: Identifying and characterising anomalies. In *Proc. SDM*, pp. 804–815, 2011.

[19] N. Tatti and J. Vreeken. Finding good itemsets by packing data. In *Proc. ICDM*, pp. 588–597, 2008.

[20] T. Van den Bulcke, P. Broucke, V. Hoof, K. Wouters, S. Broucke, G. Smits, E. Smits, S. Proesmans, T. Genechten, and F. Eyskens. Data mining methods for classification of Medium-Chain Acyl-CoA dehydrogenase deficiency (MCADD) using non-derivatized tandem ms neonatal screening data. *J Biomed Inf*, 44:319–325, 2011.

[21] J. Vreeken, M. van Leeuwen, and A. Siebes. Characterising the difference. In *Proc. KDD*, pp. 765–774, 2007.

[22] J. Vreeken, M. van Leeuwen, and A. Siebes. KRIMP: mining itemsets that compress. *Data Min. Knowl. Disc.*, 23(1):169–214, 2011.

[23] C. S. Wallace. *Statistical and Inductive Inference by Minimum Message Length.* Springer, 2005.

[24] C. Wang and S. Parthasarathy. Summarizing itemset patterns using probabilistic models. In *Proc. KDD*, pp. 730–735, 2006.

[25] G. I. Webb. Discovering significant patterns. *Mach. Learn.*, 68(1):1–33, 2007.

[26] G. I. Webb. Self-sufficient itemsets: An approach to screening potentially interesting associations between items. *ACM TKDD*, 4(1):1–20, 2010.

[27] X. Yan, H. Cheng, J. Han, and D. Xin. Summarizing itemset patterns: a profile-based approach. In *Proc. KDD*, pp. 314–323, 2005.