Snor: Simpler Descriptions through Overlapping Patterns

Matthijs van Leeuwen[1] and Jilles Vreeken[2]

[1] Universiteit Leiden, Netherlands
m.van.leeuwen@liacs.leidenuniv.nl
[2] CISPA Helmholtz Center for Information Security, Germany
jv@cispa.de

Abstract. The pattern explosion is a well-known problem that refers to the humongous numbers of results discovered by traditional pattern mining algorithms. One line of research that combats this, originally started with Krimp, and is to employ the minimum description length (MDL) principle for selecting small sets of characteristic patterns.
When Krimp was proposed twenty years ago, it was shown to work well but also had several limitations. Over time its major shortcomings have been addressed, except for one: the cover function still only considers non-overlapping patterns to describe transactions. While efficient, this may result in redundant pattern sets, as we need *combinations of* rather than just the *true patterns* to succinctly describe the data.
In this paper, we propose Snor, a cover function that enables simpler descriptions by allowing overlapping patterns in the cover of a transaction. Our key observation is that the cover problem is an instance of (weighted) set cover, and we can hence use a greedy method to approximate it. Snor can be straightforwardly combined with existing search algorithms for finding itemset code tables, such as Krimp and Slim.
In experiments on 18 benchmark datasets, we demonstrate that Snor leads to simpler descriptions, i.e., allowing overlap leads to smaller code tables while compression stays the same. Further, classification accuracy and computation time remain almost the same.

1 Introduction

Exactly 20 years ago, in September 2005, Prof. Dr. Arno Siebes coined an idea to the authors of this paper, who had then just started as PhD students under his supervision. The idea was simple, let's use the Minimum Description Length (MDL) principle [27,12] to defeat the pattern explosion in frequent itemset mining [1]. After sorting out a few further details, this resulted in Krimp– originally published at SIAM SDM 2006 [30] as a method and algorithm without a name, and later extended and given its name in 2011 [36].

Krimp[3] was the first approach to combine the MDL principle and pattern mining. It introduced the notion of a 'code table', a model containing pairs of

[3] Krimp means 'to shrink' in Dutch

patterns and code words that can be used to losslessly compress data. Compressing data was never the goal but rather a means to an end: following the MDL principle, the slogan of the approach became that 'the best set of patterns is the one that compresses the data best'. As such, KRIMP introduced a principled and rigorous approach for model selection in (unsupervised) data mining tasks such as data characterisation using frequent patterns.

A series of papers on related but data mining different tasks [21,22,31] followed, and the approach was also picked up by others (see Section 2). Over time, MDL-based pattern mining became a successful research area on itself, as can be witnessed from, for example, the 2022 survey dedicated to the topic by Galbrun [11], and the mention of the research area as a 'flourishing application' of MDL in the 2019 'MDL revisited' paper by Grünwald and Roos [13].

As the first method for MDL-based pattern mining, KRIMP had clear limitations. First, although the MDL principle prescribes using a lossless encoding, the initial proposal [30] implicitly, and the later version explicitly [36], swept some bits under the rug. Second, the filtering approach employed by the KRIMP search heuristic had an obvious downside in needing frequent itemsets as candidates. Not only does this require choosing a minimum support threshold, but also incurs long runtimes thanks to the same pattern explosion that KRIMP aims to solve. This problem was addressed by SLIM[4] [32], which generates candidates on-the-fly by considering how patterns currently in the code table are being used.

Third, the explicit inclusion of code words in the code table was suboptimal; not only did this lead to the first-mentioned problem, but in fact turned out to be unnecessary. That is, in 2015, Budhathoki and Vreeken [6] showed how we can use prequential codes to (near) optimally encode data without having to first explicitly transmit the codes in the code table. Importantly, they showed that using their scheme the total encoded length can still be computed efficiently, maintaining efficient code table search with the improved encoding.

Fourth, KRIMP disallows overlap between patterns that describe a transaction; it employs a 'cover' function that iteratively selects the longest and most frequent pattern in the model that does not overlap with previously selected patterns, until the entire transaction is described. This choice was because of efficiency (maintaining a sorted list is fast), because the ideas we tried at the time did not work, and because it was unclear whether overlap makes sense in light of MDL (why describe items multiple times if once suffices?). Ever since, non-overlapping cover functions have been taken for granted, and cover functions for other pattern and data types have followed the original idea. This is a pity, because allowing overlap does result in *smaller* pattern sets.

In this paper, we once and for all address the problem of overlapping covers. We propose an alternative cover function called SNOR[5] that allows for the cover of a transaction to contain overlapping patterns. Our key observation is that the cover problem—i.e., finding a subset of code table elements that together cover a given transaction—can be regarded as an instance of the (weighted) set

[4] SLIM means 'smart' in Dutch
[5] SNOR means 'moustache' in Dutch

cover problem. Following this observation, we propose to replace the original cover function by the well-known greedy algorithm for approximating the set cover problem, iteratively finding that pattern that covers most of the (currently uncovered part of the) transaction.

The main contributions of this paper are:

(a) We revisit KRIMP 20 years after its introduction, discuss its strengths and limitations, and describe improvements that have been proposed.
(b) We propose SNOR, a greedy approximation to the set cover problem that does cover a transaction with overlapping itemsets.
(c) We instantiate KRIMP and SLIM using SNOR and empirically demonstrate that this leads to simpler descriptions. Averaged over 18 benchmark datasets, it leads to smaller code tables that compress and classify as well as before.

We review related work in Section 2, and revisit MDL for pattern mining in Sec. 3. We discuss the classic COVER function and introduce SNOR for covering with overlap in Sec. 4. After that, we will briefly describe the KRIMP and SLIM algorithms (Sec. 5), followed by experiments (Sec. 6) and conclusions (Sec. 7).

2 Related Work

In this section we will briefly describe some notable papers that are closely related to KRIMP, either as a direct improvement, by using KRIMP code tables for another task, or by extending KRIMP to another pattern type. For a comprehensive survey on MDL for pattern mining until 2022, see Galbrun [11].

Krimp improvements. As already mentioned, SLIM [32] and DIFFNORM [6] directly built and improved on KRIMP. SLIM maintained the exact same code table model and coding as was introduced for KRIMP, but vastly improved the search algorithm by eliminating the need for a candidate set of itemsets as input. Instead, it generates candidate patterns on-the-fly by combining pairs of patterns currently in the code table and iteratively adds the generated candidate pattern that improves compression most. In addition, it uses heuristics that estimate compression gain to speed-up the candidate generation and search process.

DIFFNORM [6] uses KRIMP code tables to characterise multiple databases by looking for their 'norm' and 'differences', represented by a set of code tables modelling both sets of databases and individual databases. Moreover, it introduced prequential coding for encoding the data given the code table, replacing the Shannon-Fano coding and resulting in better overall compression by removing the need to explicitly encode code words (or usages) in the code tables.

Widened KRIMP [28] uses the concept of 'widening' to diversify the search for code tables. Instead of using a single greedy hillclimber, the goal is to pursue multiple paths in the search space in parallel, thereby hoping to find a better solution. By employing parallelisation, the goal is to achieve this without increasing runtime compared to the hillclimber.

Tasks using Krimp and Slim. MDL-optimal models have desirable properties, but when KRIMP was first introduced, it was only shown to find pattern sets

that compress the data well. To demonstrate that KRIMP (and later SLIM) code tables are indeed characteristic for the data, they were used for a series of different data mining tasks: classification [21], difference characterisation [35], change detection in data streams [20], clustering [22], structure functions [16], anomaly detection [31,2,5], denoising [17], and providing insights into models [10,14].

In this paper we revisit the classification task, to investigate the effect of allowing overlap within the cover of a transaction on classification accuracy. It would be interesting to also revisit other tasks with our proposed method.

Other pattern and data types. Over the years the MDL for pattern mining approach that started with KRIMP has been extended to numerous other pattern and data types. One direction that has received considerable attention, in several settings, is considering richer pattern languages such as including rules [34,26,39], or general dependencies [15]. Other extensions often consider more complex data and corresponding pattern types, e.g., relational data [18], sequences [33,4,8], process logs [38], graphs [19,29,3], and numerical data [24].

3 MDL for Itemset Mining — Revisited

In this section we introduce notation, the MDL principle, and the problem of finding the MDL-optimal code table for a given transaction database.

3.1 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 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 $t \in D$ supports an itemset $X \subseteq \mathcal{I}$, iff $X \subseteq t$. The support of X in D is the number of transactions in the database where X occurs.

All logarithms are to base 2, and by convention we say $0 \log 0 = 0$.

3.2 MDL, a brief introduction

The Minimum Description Length principle (MDL) [12], like its close cousin MML (Minimum Message Length) [37], is a practical version of Kolmogorov Complexity [23]. 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

$$L(D, M) = L(M) + L(D \mid M) \ ,$$

in which $L(M)$ is the length in bits of the description of M, and $L(D \mid M)$ 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 [12]. 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.

3.3 The MDL-optimal Code Table

As models we use the exact same itemset-based code tables as those introduced for Krimp [30,36]. That is, a code table is a two-column table with itemsets in the left-hand column, and corresponding code words in the right-hand column.[6]

The actual codes are of no importance: their lengths are. To explain how these lengths are computed, we first need to understand the coding algorithm. A transaction t is encoded by a subset of the itemsets in the code table, which is called its *cover*. A cover function takes a transaction (and code table) as input, and outputs the corresponding cover. We discuss how to instantiate a cover function in Section 4. For now, it suffices that a cover $C \subseteq CT$ of a transaction t permits its lossless reconstruction, i.e., $\bigcup_{X \in C} X = t$. Each code table is required to contain at least all singletons, i.e., all single items, to ensure that a cover can be constructed for each (possible) transaction over $\mathcal{I}$. A database is encoded by encoding each transaction $t \in D$ independently.

The *usage* of an itemset $X \in CT$ is the number of transactions $t \in D$ that 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 $t \in D$. For optimal compression of D, the higher $\Pr(X \mid D)$, the shorter its code should be. Shannon entropy [7] gives us the length of the optimal prefix code for X as

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

where

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

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,

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

[6] Note that we are not using prequential coding for the data, as introduced by DiffNorm [6]. This is a deliberate choice, as we aim to compare Krimp and Slim with and without Snor. It will make, as the adage goes, for engaging future research to explore how overlap works with more advanced coding schemes.

To find the optimal code table, 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 column is simply the sum of all the different code lengths. For the size of the left-hand column, note that the simplest valid code table consists only of the single items, to which we refer as the standard code table ST. We encode the itemsets in the left-hand column using the codes of ST.

The encoded size of the code table is then given by

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

We define the MDL-optimal code table as the one that minimises the total encoded size

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

More formally, we define the problem as follows.

Minimal Coding Set Problem [30] *Let $\mathcal{I}$ be a set of items and let D be a dataset over $\mathcal{I}$. Find the smallest set of patterns $\mathcal{S} \subseteq \mathcal{P}(\mathcal{I})$ such that for the corresponding code table CT the total compressed size, $L(CT, D)$, is minimal.*

The search space of all possible code tables is doubly exponential in the number of unique items. It does not exhibit structure we can use to efficiently find the optimal code table [36]. Hence, we resort to heuristics.

4 Cover Functions

Having defined code table models, code length computations, and the minimal coding set problem, the only missing link is the cover function. We first describe the original cover function, without overlap, and then introduce the new cover function, with overlap.

The idea of a cover function is that we select a set $\mathcal{C} \subseteq CT$ such that the union of the patterns $X \in \mathcal{C}$ form the transaction t at hand, i.e. $\bigcup_{X \in \mathcal{C}} X = t$. The goal of MDL is to describe the data as succinctly as possible, and hence we should choose that $\mathcal{C}$ that minimises the encoded length. If we interpret the code lengths $L(X)$ of sets X as the weights, the problem is hence an instance of Weighted Set Cover, which is known to be NP-hard. In our case, however, things are worse, as the weights depend on how often these codes are used [9]. Rather than exhaustively considering the exponentially many covers of the entire data, or trying to converge to a good solution through expectation-maximization, we resort to heuristics for finding good covers for a transaction t.

4.1 Covering without Overlap

The standard cover function, denoted Cover, was introduced for Krimp [30,36] and subsequently used by Slim [32], DiffNorm [6], and others. It finds a non-overlapping cover for a given transaction and code table. The idea it pursues

is to use as few (and hence long) and as short (and hence often-used) codes as possible to describe t.

To this end, COVER starts with an empty cover, i.e., $\mathcal{C} = \emptyset$, and iteratively adds the first $X \in CT$ such that $X \subseteq t$ and $X \cap C = \emptyset$ where $C = \bigcup \mathcal{C}$. It does so by considering itemsets $X \in CT$ in **Standard Cover Order**, defined as first preferring X of higher cardinality, then those with higher support, and finally, by lexicographical ordering (i.e., using alphabet $\mathcal{I}$, so depending on the original data). By doing so it prefers using long itemsets (such that we need few codes) with high support (such that the chance of these having a short code word is high). By optimising the total encoded size, MDL weeds out candidates that break this idea.

This straightforward cover function is easy to implement in an efficient way: as long as we maintain CT in Standard Cover Order, we can simply traverse over it and add those $X \in CT$ to $\mathcal{C}$ for which $X = (t \setminus C) \cap X$. This way, covering a single transaction t with CT has a worst case complexity of $O(|CT|)$.

4.2 Covering with Overlap

Here we introduce the main contribution of this section, dubbed SNOR. Although the standard cover function has been empirically shown to work well, it is limited to finding non-overlapping covers, which results in overly large code tables. Although intuitively appealing, overlap is a more subtle issue than it seem at first. Or, as Peter Grünwald asked us a long time ago: If you are after the minimal description, why describe an item more than once?

Let us start with an example where overlap makes sense. Consider a transaction $t = \{ABCDE\}$ and a code table consisting of patterns ABC, BCD, and CDE with high usage and singletons with low usage. COVER happily describes t using ABC and singletons D and E. While correct, this cover is much more costly in terms of bits than when we simply use ABC and CDE.

Next, we illustrate why we should not expect too much from overlap. Consider a dataset where we plant ABC, BCD, and CDE with overlap. It seems obvious that these are the only patterns we need to most succinctly describe the data. This is correct, *iff* they are strictly independent. The moment the empirical frequency of a *combination* of these deviates from the marginals, it will quickly becomes beneficial to include that combination (e.g. $ABCD$) into the pattern set. That is, overlap *only* provides a gain in terms of MDL if patterns are (sufficiently) independent, but otherwise there is not much difference. This means it is unlikely that by allowing overlap, we will see much better compression or much smaller code tables. What it will do is prevent combinations of independent and overlapping patterns to be unnecessarily included in the code table. In short, overlap will lead to (somewhat) smaller code tables.

Covering with overlap is, however, not as simple as removing the condition $X \cap C = \emptyset$ from COVER. To illustrate what goes wrong, suppose we have a transaction $t = ABCDEF$ and a code table consisting of patterns ABC, BCD, CDE, DEF all with approximately equally high usage, and singletons with low usage. COVER without overlap returns $C = \{ABC, DEF\}$, which is a good

result. It is concise and has a short encoded length $L(t)$. Obviously, we would like to obtain the same, or similarly good result if we do allow for overlap. If we remove the overlap condition from Cover, however, it returns all four patterns in the code table, which is highly redundant (C is described thrice!) and has a much longer encoded length. We encountered this problem back when developing Krimp [36] but found that straightforward ideas, such as a alternate orders, or more complicated ones, such as penalizing doubly-described items, did not solve this problem. Focusing on fixed cover orders for efficiency, we did not realize that Cover resembles a greedy algorithm for weighted set cover *without overlap*, and that by replacing it with an efficient greedy algorithm for weighted set cover *with overlap*, we obtain the solution we were after.

That is, the key idea to making overlap work is to follow the underlying idea of Cover to iteratively select that $X \in CT$ that describes the most *uncovered* elements of t in (likely) the fewest number of bits. To this end, Snor starts just like Cover with an empty cover, $\mathcal{C} = \emptyset$. It then iteratively chooses and adds that $X \in CT$ such that $X \subseteq t$ and $|t \setminus C|$ is maximal. That is, Snor iteratively adds that code table element that covers most of the items in the transaction that have not been covered yet. Further, it break ties by using the same Standard Cover Order as Cover, preferring X of higher cardinality, then those with higher support, and finally, determining preference lexicographically as to prefer those patterns with (the highest probability of having) a short encoded length.

Snor is a bit harder to implement efficiently than Cover. To cover a given transaction t, we lazily maintain a list L of patterns $X \in CT$ that can still be added to cover $\mathcal{C}$, i.e., for which $X \subseteq t$ and $X \notin \mathcal{C}$. We order the elements $X \in L$ by how many uncovered items of t they can cover, i.e., $|(t \cap X) \setminus C|$. This way we can quickly access those $X \in CT$ that cover the most uncovered elements of t. This way, covering a single transaction t with CT has a worst case complexity of $O(|\mathcal{C}| \cdot |CT|)$. Luckily this is much faster in practice.

5 Search Algorithms

We will use the original Krimp and Slim algorithms for finding good tables and experiment with both cover functions, Cover and Snor, to assess the impact of covering with or without overlap.

5.1 Krimp

The Krimp algorithm was introduced by Siebes et al. [30,36] for mining good code tables given a set of candidate patterns. We give the pseudo-code of Krimp as Algorithm 1. Krimp starts with the singleton code table ST (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 X is considered in turn by inserting it in CT (l.3) which we maintain ordered first descending on itemset length, second descending on support, and third lexicographically. We accept a candidate only

if it improves compression (l.4). If accepted, we post-prune the code table by reconsidering all elements $X \in CT$ w.r.t. their contribution to compression (l.5). Post-pruning is a pivotal step as it helps to weed out patterns that are no longer useful.

Algorithm 1: The KRIMP Algorithm [36]

input : Database D and Candidate Set $\mathcal{F}$
output : Code Table CT

1 $CT \leftarrow ST$;
2 **foreach** $X \in \mathcal{F}$ in **Standard Candidate Order do**
3 $\quad CT' \leftarrow CT \cup X$;
4 $\quad$ **if** $L(D, CT') < L(D, CT)$ **then**
5 $\quad\quad CT \leftarrow$ *post-prune*(CT');
6 **return** CT;

5.2 Slim

The SLIM algorithm was introduced by Smets and Vreeken [32] for mining good code tables directly from data. We give the pseudo-code of SLIM as Algorithm 2. SLIM starts with the singleton-only code table ST (line 1). In every iteration (line 2) we consider all pairwise combinations of $X, Y \in CT$ and estimate the gain in compression, $L(D, CT) - \hat{L}(D, CT \oplus Z)$, if we were to add $Z = X \cup Y$ to the current code table. Iteratively, in descreasing order of estimated gain, we add a candidate Z to CT (line 3), cover the data, and compute the actual gain (line 4). If compression improves, we accept the candidate, otherwise we reject it. If accepted, we post-prune by reconsidering weeding out every pattern in CT that no longer contributes towards compression (line 5). We then update the candidate list (line 2), and continue until no candidate decreases the total compressed size, after which we are done.

Algorithm 2: The SLIM Algorithm [32]

input : Database D
output : Code Table CT

1 $CT \leftarrow ST$;
2 **foreach** $Z \in \{X \cup Y : X, Y \in CT\}$ in **Gain Order do**
3 $\quad CT' \leftarrow CT \cup Z$;
4 $\quad$ **if** $L(D, CT') < L(D, CT)$ **then**
5 $\quad\quad CT \leftarrow$ *post-prune*(CT');
6 **return** CT;

Dataset	$\|D\|$	$\|I\|$	$\|C\|$	*minsup*	$L(D, ST)$
Adult	48842	97	2	20	3569724
Anneal	898	71	5	1	62827
Breast	699	16	2	1	27113
Chess (k-k)	3196	75	2	1200	687120
Chess (kr-k)	28056	56	18	1	1083046
DNA Amplification	4590	392	-	1	212640
Heart	303	50	5	1	20543
Hepatitis	155	52	-	1	14959
Iris	150	19	3	1	3058
Led7	3200	24	10	1	107091
Letter Recognition	20000	102	26	50	1980244
Mushroom	8124	119	2	1	1111287
Nursery	12960	32	5	1	569042
Pageblocks	5473	44	5	1	216552
Pen Digits	10992	86	10	50	1140795
Pima	768	38	2	1	26250
TicTacToe	958	29	2	1	45977
Wine	178	68	3	1	14101

Table 1. Datasets. Given are the numbers of transactions ($|D|$), items ($|I|$), resp. classes ($|C|$), the minimal support threshold (*minsup*) such that KRIMP finishes within a reasonable time, and the encoded size using the singleton-only code table ($L(D, ST)$).

6 Experiments

In this section we empirically investigate the benefit of using SNOR instead of COVER in either KRIMP [36] or SLIM [32]. For conciseness, we will write KRIMP and SLIM for their original setup using COVER. We write KRAMP and SLAM to refer to their respective setup using SNOR, where the 'a' comes from 'overlap'.

6.1 Setup

We evaluate the algorithms on 18 datasets that together cover a wide range of characteristics. We give the base statistics in Table 1. We use the largest and most dense databases from the LUCS/KDD dataset repository, and obtain *Chess (kr-kp)* and *Chess (kr-k)* from the UCI repository. As real data we consider the DNA Amplification database [25], which contains DNA copy number amplifications. Such copies activate oncogenes and are hallmarks of advanced tumours.

KRIMP and KRAMP require a set of frequent patterns as candidates. For these, we mine closed frequent patterns with a minimum support threshold (*minsup*) set such that they finish within an hour. We give these minsup values in Table 1. SLIM and SLAM permit setting a *minsup*, but by default use *minsup* 1, which is what we use unless stated differently. For all algorithms, we enable pruning the code table after every accepted new pattern.

			Krimp		Kramp		*diff*	
Dataset	$\|D\|$	$\|I\|$	$\|CT\|$	$L(D,M)$	$\|CT\|$	$L(D,M)$	$\Delta\|CT\|$	$\Delta L(D)$
Adult	48842	97	1346	907803	1043	870096	-22.5%	-4.2%
Anneal	898	71	141	22873	131	21128	-7.1%	-7.6%
Breast	699	16	32	4896	37	4921	8.8%	0.5%
Chess (kr-kp)	3196	75	213	293176	149	280824	-30.0%	-4.2%
Chess (kr-k)	28056	56	1721	667339	1386	643654	-19.5%	-3.5%
DNA Amp.	4590	392	615	79829	586	78454	-4.7%	-1.7%
Heart	303	50	99	12241	107	11829	8.1%	-3.4%
Hepatitis	155	52	92	8317	89	8146	-3.3%	-2.1%
Iris	150	19	22	1475	23	1487	4.5%	0.8%
Led7	3200	24	168	30664	161	31310	-4.5%	2.1%
LetRecog	20000	102	1317	832483	1138	856730	-13.6%	2.9%
Mushroom	8124	119	268	274220	376	242017	40.3%	-11.7%
Nursery	12960	32	262	258898	302	270634	15.3%	4.5%
Pageblocks	5473	44	75	10967	71	10934	-5.3%	-0.3%
Pen Digits	10992	86	1072	523805	811	546219	-24.3%	4.3%
Pima	768	38	84	9199	84	9171	0.0%	-0.3%
TicTacToe	958	29	181	28880	172	28285	-5.0%	-2.1%
Wine	178	68	117	10965	123	10809	5.1%	-1.4%
Average			435	221002	377	218147	-3.2%	-1.5%

Table 2. Kramp *mines better pattern sets than* Krimp. For 18 benchmark datasets, we show the number of transactions ($|D|$), number of items ($|I|$), and give the number of discovered patterns ($|CT|$, lower is better) and the total encoded size ($L(D,M)$, lower is better) for Krimp resp. Kramp using the *minsup* values given in Table 1. We also give the relative differences between the number of discovered patterns and total encoded sizes. Green is better, red is worse. On average, Kramp mines smaller code tables that compress better than those mined by Krimp.

We implemented all algorithms in C++ and make our code available for research purposes.[7] To compare fairly, we use equally (un)optimised implementations of the cover functions[8] and report wall-clock running times. All experiments were conducted as single-threaded runs on a Windows 11 machine with an AMD Ryzen 5 3600X processor (3.8GHz) and 32 GB of memory.

6.2 Compression

First, we investigate the effect of Snor in terms of the models that Krimp and Slim find. In particular, we are interested in whether allowing overlap leads to better compression. To this end we run each method on all datasets. We show the results for Krimp and Kramp in Table 2, and for Slim and Slam in Table 3.

[7] `https://vreeken.eu/prj/snor/`

[8] We use the more general 'cpoul' branch of cover functions, rather than the heavily optimised 'cccp' branch that does not allow overlap.

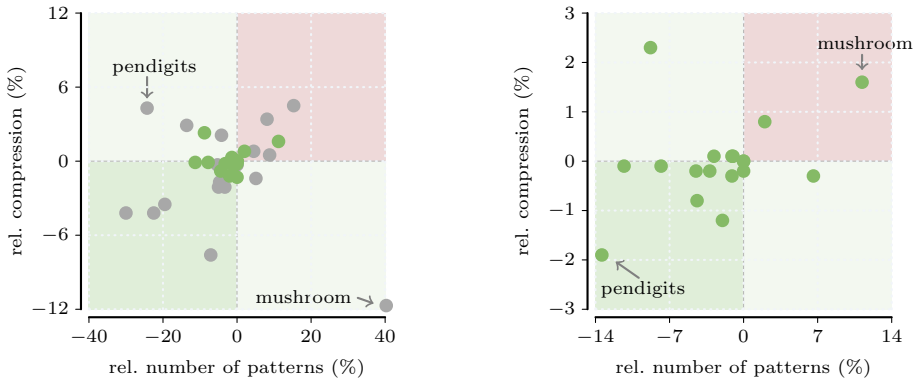

Fig. 1. SNOR *and* KRAMP *mine smaller code tables.* On the left, we consider KRIMP versus KRAMP (grey) and SLIM versus SNOR (green) using *minsup* values such that KRIMP finishes within reasonable time (see Tab. 1). On the right, we consider SLIM and SNOR using *minsup* = 1. The bottom left quadrants contain datasets where overlap leads to *smaller* pattern sets that compress *better*. The bottom right those where overlap leads to *larger* pattern sets that compress *better*, the top left *smaller* pattern sets that compress *worse*, and the top right *larger* pattern sets that compress *worse*. With only few exceptions, KRAMP and SLAM provide a benefit over KRIMP and SLIM.

Before we discuss these results in detail, we consider Figure 1. There, we plot the relative gain in compression, i.e.,

$$\Delta L(D) = \frac{L_{\text{COVER}}(D, CT) - L_{\text{SNOR}}(D, CT)}{L_{\text{COVER}}(D, CT)} \ ,$$

against the relative number of discovered patterns, i.e.,

$$\Delta |CT| = \frac{|CT_{\text{COVER}}| - |CT_{\text{SNOR}}|}{|CT_{\text{COVER}}|} \ ,$$

between using KRIMP or SLIM with COVER resp. SNOR. In both cases, larger negative numbers are better.

On the left, we plot the difference between KRIMP and KRAMP, resp. SLIM and SLAM, using the *minsup* thresholds from Table 1. We see that for the majority of datasets, SNOR leads to smaller pattern sets that compress better. The effects of SNOR are more pronounced for KRIMP than for SLIM. For *Mushroom*, KRAMP mines a larger code table (40% more patterns) that compresses much better (12% better), while for *Pen Digits* it mines a smaller code table (24% fewer patterns) that compresses a bit worse (4.3%).

On the right, we plot the difference between SLIM and SLAM when we set the minimal support threshold to 1. We see again that for the majority of datasets, SNOR leads to smaller pattern sets that compress better. The effect is less pronounced than for KRIMP, with improvements up to 14% fewer patterns and up to 2% better compression. Interestingly, we find that SLAM performs much worse

			Slim		Slam		*diff*	
Dataset	$\|D\|$	$\|I\|$	$\|CT\|$	$L(D,M)$	$\|CT\|$	$L(D,M)$	$\Delta\|CT\|$	$\Delta L(D)$
Adult	48842	97	1264	813778	1229	814599	-2.8%	0.1%
Anneal	898	71	147	20679	134	21146	-8.8%	2.3%
Breast	699	16	28	4251	28	4251	0.0%	0.0%
Chess (kr-kp)	3196	75	243	102730	232	102568	-4.5%	-0.2%
Chess (kr-k)	28056	56	1056	622946	1044	623665	-1.1%	0.1%
DNA Amp.	4590	392	613	78933	606	78660	-1.1%	-0.3%
Heart	303	50	99	10210	97	10089	-2.0%	-1.2%
Hepatitis	155	52	90	6798	86	6742	-4.4%	-0.8%
Iris	150	19	22	1489	22	1489	0.0%	0.0%
Led7	3200	24	147	29408	150	29648	2.0%	0.8%
LetRecog	20000	102	1635	664262	1527	666218	-6.6%	0.3%
Mushroom	8124	119	384	208608	427	211936	11.2%	1.6%
Nursery	12960	32	207	247390	205	247742	-1.0%	0.1%
Pageblocks	5473	44	77	10911	71	10898	-7.8%	- 0.1%
Pen Digits	10992	86	1409	452528	1220	443946	-13.4%	-1.9%
Pima	768	38	76	8342	76	8327	0.0%	-0.2%
TicTacToe	958	29	133	23090	118	23074	-11.3%	-0.1%
Wine	178	68	124	10676	120	10652	-3.2%	-0.2%
Average			431	184279	411	184203	-3.1%	0.0%

Table 3. Slam *mines smaller pattern sets than* Slim. For 18 benchmark datasets, we show the number of transactions ($|D|$), number of items ($|I|$), and give the number of discovered patterns ($|CT|$, lower is better) and the total encoded size ($L(D,M)$, lower is better) for Slim resp. Slam for *minsup* = 1. We also give the relative differences between the number of discovered patterns and total encoded sizes. Green is better, red is worse. On average, Slim with Snor compresses as well as vanilla Slim, but does so with smaller code tables.

on *Mushroom* than Slim; the former returns a much larger code table (13% more patterns) that compresses worse (1.5% more bits). For *Pen Digits*, however, we see an improvement, as Slam finds a code table that is much smaller (14% fewer patterns) that compresses better (2% fewer bits). It is unclear why the relative compression gains for these two datasets are reversed between Krimp and Kramp resp. Slim and Slam.

Next, we consider the detailed results for Krimp and Kramp in Table 2. On average, Kramp mines code tables that contain 3.2% fewer patterns yet compress 1.5% better. There is no clear trend, but it seems that datasets where *minsup* is set larger than 1, Snor has a stronger effect.

Next, we consider the results of Slim and Slam in Table 3. We here find that overlap leads to smaller code tables (3.1% on average) that compress as well as those without overlap. We attribute this to the advanced search scheme that Slim employs, which enables it to mine candidates from how patterns are being used to cover the data. It is unclear why exactly overlap has such a strongly negative effect for *Mushroom*. We postulate that this also has to do with the

SLIM search scheme, in that it initially finds overlapping 'building blocks' and then refines these into variations of a theme, rather than discovering new themes.

6.3 Classification

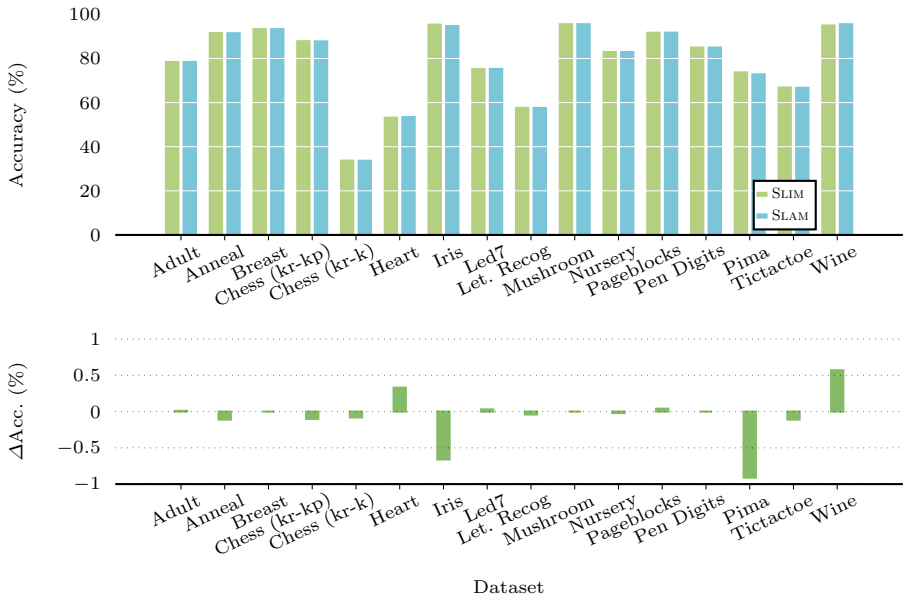

Fig. 2. SLAM *performs almost as well as* SLIM *in classification.* We plot classification accuracies (top) and absolute differences in accuracy (bottom) for SLIM and SNOR. For many datasets there is little to no difference in accuracy.

Next, we evaluate the effect of SNOR on how well the mined code tables generalise. To this end, we consider the task of classification and follow the setup of Van Leeuwen et al. [21], in which we classify an unseen transaction t as label $l \in L$, according to $\arg\min_{l \in L} L(t \mid CT_l)$, where CT_l is the code table inferred for D_l. As data, we consider 15 benchmark datasets. As SLIM has been shown to outperform KRIMP in terms of classification [32], we focus on SLIM and SLAM.

We report the average accuracy over 10-fold cross-validation in Figure 2 and Table 4. We see that both methods perform very similarly. If we count carefully, we find that SLIM beats SLAM in 8 cases, SLAM beats SLIM in 5, but then also find that the differences in accuracy are very small (0.07% on average) and well within the standard deviations. The results on the very small *Iris* and *Pima* datasets skew the average somewhat; if we disregard these, SLAM in fact outperforms SLIM with a difference of 0.03% (77.68 ± 2.69 versus 77.71 ± 2.58).

Dataset	$\|D\|$	$\|I\|$	$\|C\|$	Slim *acc* ±*std*	Slam *acc* ±*std*	*diff*
Adult	48842	97	2	78.40 ±0.63	**78.41** ±0.52	0.01%
Anneal	898	71	5	**91.54** ±2.17	91.43 ±3.37	-0.11%
Breast	699	16	2	**93.28** ±3.83	**93.28** ±2.63	0.00%
Chess (kr-kp)	3196	75	2	**87.83** ±2.03	87.73 ±2.86	-0.10%
Chess (kr-k)	28056	56	18	**33.70** ±0.61	33.62 ±0.75	-0.08%
Heart	303	50	5	53.14 ±9.34	**53.47** ±11.18	0.33%
Iris	150	19	3	**95.33** ±4.27	94.67 ±8.33	-0.66%
Led7	3200	24	10	75.22 ±2.87	**75.25** ±2.36	0.03%
Letter Recog.	20000	102	26	**57.59** ±1.20	57.55 ±1.04	-0.04%
Mushroom	8124	119	2	**95.51** ±0.67	**95.51** ±0.57	0.00%
Nursery	12960	32	5	**82.91** ±0.91	82.89 ±0.87	-0.02%
Pageblocks	5473	44	5	91.65 ±1.02	**91.69** ±0.82	0.04%
Pen Digits	10992	86	10	**84.93** ±1.03	**84.93** ±0.95	0.00%
Pima	768	38	2	**73.70** ±6.09	72.79 ±2.88	-0.91%
TicTacToe	958	29	2	**66.81** ±5.94	66.70 ±4.90	-0.11%
Wine	178	68	3	94.94 ±5.42	**95.51** ±3.36	0.57%
Avg. Acc				**78.53** ±3.00	78.46 ±2.96	-0.07%
Avg. Rank				**1.41**	1.59	

Table 4. Slam *performs almost as well as* Slim *in classification.* For sixteen benchmark datasets, we show the number of transactions ($|D|$), number of items ($|I|$), number of classes ($|C|$), and average accuracy (*acc*) with standard deviation (*std*) as obtained using 10-fold cross validation using Slim resp. Slam. Bold numbers are better.

6.4 Runtime

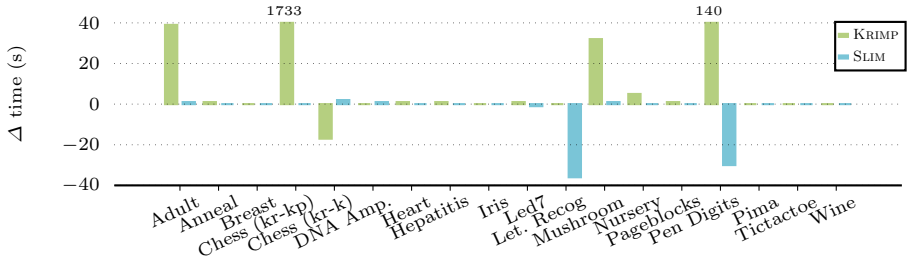

Fig. 3. Snor *slows down* Krimp *but speeds up* Slim. We plot the difference in runtime between Krimp and Kramp, resp. Slim and Slam. For many datasets there is little to no difference in runtime. For Slim, we find cases with significant speed-up, while for Krimp we find the opposite.

Next, we consider the effect of Snor in terms of runtime. In Figure 3, we show the differences in runtime of Krimp and Kramp, resp. Slim and Slam. We see that for most datasets, the overhead incurred by Snor is negligible. For Kramp we have four notable exceptions: it is much slower on *Adult*, *Chess (kr-kp)*, *Mushroom*, and *Pen Digits*. For Slam we have two notable exceptions: it is notably faster than Slim on *Letter Recognition* and *Pen Digits*. It is unclear whether this is because these datasets focus on a computer vision task, or another reason.

7 Conclusion

With the goal of obtaining smaller pattern sets that describe the data even better, we studied how to enable Krimp and Slim to describe the data using sets of patterns that overlap. We studied the original Cover function used by both, and saw that it is an efficient but crude approximation of greedy weighted set cover. We saw that because of the interplay between usage and code lengths an optimal solution is infeasible, and that naively extending Cover to allow overlap leads to bad results. Taking inspiration from the aspects that make Cover work well, we then proposed Snor, an efficient heuristic for succinctly describing a transaction using overlapping patterns. The key idea is that rather then considering the patterns in a fixed order, we iteratively select that pattern that covers the most uncovered items.

Through an extensive set of experiments, we showed that Krimp and Slim obtain better results with Snor than without. On average, Snor leads to smaller pattern sets that compress the data at least as well, permit equally good classification, and in case of Slim even speeds up the inference of the code tables. In other words, pattern mining is simply better with Snor.

While the experiments show that Snor leads to smaller pattern sets, whether these patterns are also simpler is left for future investigation. We postulate that Snor speeds Slim up exactly because of this reason, i.e., once Slim has found the (overlapping) building blocks ABC, CDE, EFG that make up the data, thanks to Snor it can accurately estimate if these are used (sufficiently) independently to expect any gain in compression by combining them; this, as opposed to Slim having to consider combining ABC with D, and then $ABCD$ with E, etc. to describe the data equally well without overlap.

We focused on Krimp and Slim, by which it remains a question whether Snor would also improve DiffNorm, and related, it is interesting to investigate variants of Snor for sequential or graph data. Another challenge for the future is to how to enable Krimp and Slim to effectively deal with destructive noise.

All in all, even after 20 years plenty of opportunities to revisit remain.

Acknowledgements

The authors thank Arno Siebes for suggesting to use MDL for pattern mining, the many great discussions that followed, but above all on delivering on the 'fun guarantee' he gave them at the start of their Ph.D.

References

1. C. C. Aggarwal and J. Han, editors. *Frequent Pattern Mining*. Springer, 2004.
2. L. Akoglu, H. Tong, J. Vreeken, and C. Faloutsos. CompreX: Compression based anomaly detection. In *Proceedings of the 21st ACM Conference on Information and Knowledge Management (CIKM), Maui, HI*. ACM, 2012.
3. F. Bariatti, P. Cellier, and S. Ferré. GraphMDL: Graph pattern selection based on minimum description length. In *Proceedings of Advances in Intelligent Data Analysis (IDA)*, pages 54–66. Springer, 2020.
4. R. Bertens, J. Vreeken, and A. Siebes. Keeping it short and simple: Summarising complex event sequences with multivariate patterns. In *Proceedings of the 22nd ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD), San Francisco, CA*, pages 735–744, 2016.
5. R. Bertens, J. Vreeken, and A. Siebes. Efficiently discovering unexpected pattern co-occurrences. In *Proceedings of the SIAM International Conference on Data Mining (SDM'17)*. SIAM, 2017.
6. K. Budhathoki and J. Vreeken. The difference and the norm – characterising similarities and differences between databases. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), Porto, Portugal*. Springer, 2015.
7. T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience New York, 2006.
8. J. Cueppers and J. Vreeken. Below the surface: Summarizing event sequences with generalized sequential patterns. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2023.
9. I. Farag. Efficient data summarization with overlapping patterns. Msc. thesis, Saarland University, 2018.
10. J. Fischer, A. Oláh, and J. Vreeken. What's in the box? exploring the inner life of neural networks with robust rules. In *Proceedings of the International Conference on Machine Learning (ICML)*. PMLR, 2021.
11. E. Galbrun. The minimum description length principle for pattern mining: a survey. *Data Mining and Knowledge Discovery*, 36(5):1679–1727, 2022.
12. P. Grünwald. *The Minimum Description Length Principle*. MIT Press, 2007.
13. P. Grünwald and T. Roos. Minimum description length revisited. *International Journal of Mathematics for Industry*, 11(1), 2019.
14. M. Hedderich, J. Fischer, D. Klakow, and J. Vreeken. Label-descriptive patterns and their application to characterizing classification errors. In *Proceedings of the International Conference on Machine Learning (ICML)*. PMLR, 2022.
15. H. Heikinheimo, J. Vreeken, A. Siebes, and H. Mannila. Low-entropy set selection. In *Proceedings of the 9th SIAM International Conference on Data Mining (SDM), Sparks, NV*, pages 569–580, 2009.
16. R. Kersten and A. Siebes. A structure function for transaction data. In *Proceedings of the SIAM International Conference on Data Mining (SDM)*, 2011.

17. R. Kersten and A. Siebes. Smoothing categorical data. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, 2012.
18. A. Koopman and A. Siebes. Characteristic relational patterns. In *Proceedings of the 15th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD), Paris, France*, pages 437–446, 2009.
19. D. Koutra, U. Kang, J. Vreeken, and C. Faloutsos. VoG: Summarizing graphs using rich vocabularies. In *Proceedings of the SIAM International Conference on Data Mining (SDM), Philadelphia, PA*, pages 91–99. SIAM, 2014.
20. M. van Leeuwen and A. Siebes. StreamKrimp: Detecting change in data streams. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), Antwerp, Belgium*, pages 672–687, 2008.
21. M. van Leeuwen, J. Vreeken, and A. Siebes. Compression picks the item sets that matter. In *Proceedings of the 10th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD), Berlin, Germany*, pages 585–592, 2006.
22. M. van Leeuwen, J. Vreeken, and A. Siebes. Identifying the components. *Data Mining and Knowledge Discovery*, 19(2):173–292, 2009.
23. M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer, 1993.
24. T. Makhalova, S. O. Kuznetsov, and A. Napoli. Mint: MDL-based approach for Mining INTeresting numerical pattern sets. *Data Mining and Knowledge Discovery*, 36(1):108–145, 2022.
25. 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.
26. H. M. Proenca and M. van Leeuwen. Interpretable multiclass classification by mdl-based rule lists. *Information Sciences*, 512:1372–1393, 2020.
27. J. Rissanen. Modeling by shortest data description. *Automatica*, 14(1):465–471, 1978.
28. O. Sampson and M. R. Berthold. Widened krimp: Better performance through diverse parallelism. In *Proceedings of Advances in Intelligent Data Analysis (IDA)*, 2014.
29. N. Shah, D. Koutra, T. Zou, B. Gallagher, and C. Faloutsos. Timecrunch: Interpretable dynamic graph summarization. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 1055–1064. ACM, 2015.
30. A. Siebes, J. Vreeken, and M. van Leeuwen. Item sets that compress. In *Proceedings of the 6th SIAM International Conference on Data Mining (SDM), Bethesda, MD*, pages 393–404. SIAM, 2006.
31. K. Smets and J. Vreeken. The odd one out: Identifying and characterising anomalies. In *Proceedings of the 11th SIAM International Conference on Data Mining (SDM), Mesa, AZ*, pages 804–815. Society for Industrial and Applied Mathematics (SIAM), 2011.
32. K. Smets and J. Vreeken. Slim: Directly mining descriptive patterns. In *Proceedings of the 12th SIAM International Conference on Data Mining (SDM), Anaheim, CA*, pages 236–247. Society for Industrial and Applied Mathematics (SIAM), 2012.
33. N. Tatti and J. Vreeken. The long and the short of it: Summarizing event sequences with serial episodes. In *Proceedings of the 18th ACM International Conference on*

Knowledge Discovery and Data Mining (SIGKDD), Beijing, China, pages 462–470. ACM, 2012.
34. M. van Leeuwen and E. Galbrun. Association discovery in two-view data. *IEEE Transactions on Knowledge and Data Engineering*, 27(12):3190–3202, 2015.
35. J. Vreeken, M. van Leeuwen, and A. Siebes. Characterising the difference. In *Proceedings of the 13th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD), San Jose, CA*, pages 765–774, 2007.
36. J. Vreeken, M. van Leeuwen, and A. Siebes. KRIMP: Mining itemsets that compress. *Data Mining and Knowledge Discovery*, 23(1):169–214, 2011.
37. C. S. Wallace. *Statistical and inductive inference by minimum message length*. Springer, 2005.
38. B. Wiegand, B. Klakow, and J. Vreeken. Mining easily understandable models from complex event data. In *Proceedings of the SIAM Conference on Data Mining (SDM)*. SIAM, 2021.
39. L. Yang and M. van Leeuwen. Truly unordered probabilistic rule sets for multi-class classification. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*. Springer, 2022.