Utrecht University

Department of Information and Computing Sciences

Low Entropy Attribute Set Partitioning

Author:
Niels Rustenburg
ICA-3539792

Supervisor:
prof. dr. A.P.J.M. Siebes

May 17, 2019

Contents

1 Introduction **3**
1.1 Outline 4

2 Related Work **5**
2.1 Important Concepts from Information Theory 5
2.1.1 Shannon Entropy 5
2.1.2 Minimum Description Length Principle 8
2.2 KRIMP 9
2.2.1 Frequent Itemset Mining 9
2.2.2 KRIMP: MDL for Frequent Itemsets 10
2.3 SLIM 13
2.4 Identifying the Components 15
2.4.1 Model-Driven Partitioning 15
2.4.2 Data-Driven Partitioning 16
2.5 Low Entropy Set Selection 16
2.5.1 Low Entropy Set Mining 16
2.5.2 LESS: MDL for (Low Entropy) Attribute Sets 18

3 Research Question **22**
3.1 Subquestions 22

4 Improvements to LESS **23**
4.1 Candidate Generation 23
4.2 Biased Code Lengths in CT_I 23
4.2.1 LESS-Per-Length 25
4.2.2 LESS-Individual-Instantiations 25
4.2.3 Which variant should we use? 26
4.3 Estimating Candidate Compression Improvement 26
4.3.1 Consequences of the Cover Function 26
4.3.2 Using A Different Cover Function? 28
4.4 Conclusions 28

5 Partitioning with LESS(-variants) **29**
5.1 Data-Driven Partitioning for LESS 29
5.2 Model-Driven Partitioning for LESS 30
5.3 Partitioning Algorithm for LESS 31
5.4 Conclusions 33

6 Experiments **34**
6.1 Datasets 34
6.2 Results 35
6.2.1 Comparison with van Leeuwen et al. 2009 35
6.2.2 Visualized Results on European Mammals dataset 37
6.2.3 Entropy of Class Distribution 38

6.3 Conclusions 39

7 Conclusion **41**
7.1 Summary 41
7.2 Discussion and Future Work 41

1 Introduction

The KRIMP algorithm [16] was introduced by Siebes et al. as a way to combat pattern explosion, a commonly encountered problem in Frequent Itemset Mining [1]. A classic example for frequent itemset mining is that of trying to find patterns among the transactions made in a super market, where the patterns, frequent itemsets, are sets of products which are often bought together. The problem known as pattern explosion is that if one wishes to find more than the obvious and well-known patterns, one is confronted with a huge amount of mostly redundant patterns, as many patterns have some level of overlap with each other. The KRIMP algorithm manages to reduce the huge set of frequent itemsets to a small set of patterns characteristic to the dataset. It does so by following the Minimum Description Length principle [14, 6], which can be summarized as model selection through compression. KRIMP uses the frequent itemsets to give a lossless description of the original data, and tries to find the set of patterns which gives the most succinct description. Further validating the KRIMP approach, the models it produces have been shown to be useful at various other tasks, such as classification [21], outlier detection [18], difference measurement [20] and partitioning [19].

Low entropy sets [8] are a type of pattern similar to frequent itemsets. The difference between low entropy sets and frequent itemsets is that frequent itemsets only consider the co-occurrence of items, whereas low entropy sets signify strong interaction between items in general, by considering items symmetrically. Both patterns have a monotonic property which allows for efficient pattern mining algorithms. Many of the pattern mining algorithms for frequent itemset mining have been converted to the context of low entropy set mining. Low entropy set mining is also plagued by pattern explosion, it should come as no surprise that an algorithm similar to KRIMP has been proposed to solve this. This algorithm is called Low Entropy Set Selection [9], or LESS for short, and was introduced by Heikinheimo et al. Unlike with KRIMP the models produced by LESS have not been used for other applications yet.

Partitioning, or clustering is the task of creating groups of data points without being provided some ground truth grouping through class labelling. Algorithms for this task often require some form of distance or similarity metric in order to create these groupings. When it is hard to define a meaningful distance metric ad hoc approaches are employed. The benefit of the KRIMP-based approach to partitioning by van Leeuwen et al. [19] is that no distance measure needs to be defined, so no prior knowledge of the data is required. In addition their models provide insight into the patterns underlying each grouping, whereas other well-known clustering methods such as k-means [11] merely give the grouping itself.

The subject of this thesis has been inspired by an ad hoc approach to partitioning, known as Minimum Entropy Decomposition, by Eren et al. [5]. This al-

gorithm comes from the field of bio-informatics where it is used to group together genetic sequences into high resolution Operational Taxonomic Units, while filtering out erroneous sequences created during laboratorial analysis. While their algorithm makes use of an ad hoc stopping criterion it shows that entropy can be a very effective tool for partitioning.

Because LESS is an entropy-based variant of KRIMP which has yet to be applied to other tasks it seemed like an interesting subject for a thesis to apply the LESS algorithm to the partitioning task, similar to how van Leeuwen et al. do this with KRIMP models.

1.1 Outline

This thesis is outlined as follows: in Section 2 I will discuss the literature relevant to my thesis subject. In Section 3 I formulate the main research question, and three sub questions. These subquestions are answered in Sections 4, 5 and 6. Lastly in Section 7 I answer the main research question while summarizing the contents of this thesis, followed by a discussion and suggestions for future work.

2 Related Work

I begin the related work section covering some important concepts from the field of Information Theory in Section 2.1, the first being Shannon entropy [15] and its relation to compression, the second being the Minimum Description Length principle [14, 6]. After this, Section 2.2 will discuss frequent itemset mining and the KRIMP models and algorithm [16, 21]. The SLIM algorithm [17] is discussed in Section 2.3 because it works with the same models as KRIMP but uses a smarter heuristic to find better solutions in less time. I will then discuss the KRIMP-based partitioning algorithms suggested by van Leeuwen et al. [19] in Section 2.4. This is followed by an explanation of low entropy set mining [8], and the KRIMP-like Low Entropy Set Selection algorithm [9] in Section 2.5.

2.1 Important Concepts from Information Theory

For this subsection I will mostly refer to explanations from the book *Elements of Information Theory* [4] by Cover and Thomas.

2.1.1 Shannon Entropy

Shannon entropy [15] is a function on probability distributions, used as a measure of the information produced by some data source. Consider a source to be a random variable X which produces information by taking values $x \in \mathcal{X}$, the entropy $H(X)$ is defined by the following formula, where $p(x) = P(X = x)$:

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x) \tag{1}$$

The logarithm is base 2, meaning entropy is measured in bits. Unless explicitly stated otherwise all logarithms in this thesis are base 2. Further, it is appropriate to follow the convention that $0 \log 0 = 0$, as values with a probability of zero should not affect a measure of the information produced by X. Aside from the (for now) vague notion of being a measure of information it is clear from this formula that entropy tells us something about the skewness of a probability distribution, the more skewed the distribution, the lower the entropy.

There are multiple ways to justify using Shannon entropy as a measure of information, for this thesis I focus on the relation between entropy and compression. A more thorough account of this can be found in chapter 5 of *Elements of Information Theory* [4].

Suppose the string “21131241” represents a sequence of states output by a random variable X from a set of possible states $\mathcal{X} = \{1, 2, 3, 4\}$ and consider the task of storing the output produced by X concisely in a binary format. To translate the output of X in binary we need a mapping from possible states of X to binary strings, $C(x) : \mathcal{X} \to \{0, 1\}^*$. We call such a mapping a *code*, the states $x \in \mathcal{X}$ *source words* and the binary strings *code words*. A code can be

represented by a binary tree. Source words are placed on nodes of the tree, and the corresponding code word is given by the path from the root of the tree to the node of the source word. At every branching of the tree, going left adds a '0' to the code word and going right a '1'.

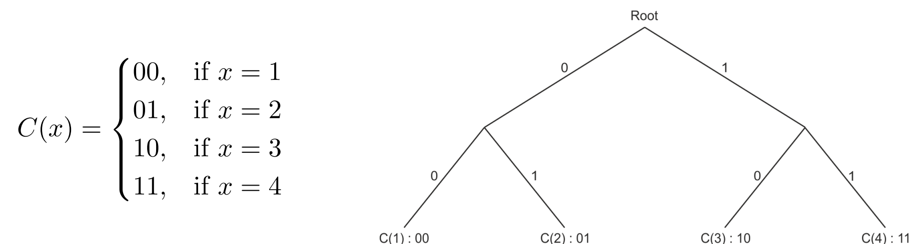

Figure 1: Example prefix code $C(x)$ and its tree representation

A special kind of code is the *prefix code* this is a code where no code word is the prefix of another code word. As a result any string encoded by $C(x)$ is uniquely decodable, meaning it can only be produced by (and decoded into) one possible source string. In addition to being uniquely decodable, for a prefix code we can immediately recognize when we've arrived at the end of a code word, allowing for instantaneous decoding. Prefix codes provide the most efficient encoding when we require these features, as other codes would require additional symbols to indicate the ending of a code word.

In the binary tree representation of a prefix code all source words must be located at leaf nodes. If this were not the case the path to one source word would be contained by the path to another, causing its code word to be prefix of the other code word. Figure 1 gives an example of a prefix code and its corresponding binary tree.

The prefix constraint has an effect on the length of the code words l_x, this effect is captured by Kraft's Inequality:

$$\sum_i 2^{-l_i} \leq 1 \tag{2}$$

Intuitively it can be seen as a budget for code words in a prefix code, where using a code word of length l_x takes up 2^{-l_x} of your budget, as none of the nodes below it in the tree representation can be used for coding anymore. A more formal explanation of Kraft's inequality can be found in section 5.2 of Cover et al. [4].

Using the code from Figure 1 the expected code length L_X to encode a

source word produced by X is 2 bits as all code words require 2 bits.

$$L_X = \sum_{x \in \mathcal{X}} p(x) l_x \tag{3}$$

If we know something about the probability distribution of X we could use this knowledge to lower the expected code length, by assigning shorter codes to more frequent outcomes. Suppose the outputs from X are distributed according to the probability distribution in (4). We can then lower the expected code length to 1.75 bits using the code $C'(x)$ from Figure 2.1.1, which is actually the optimal code for X.

$$\begin{aligned} P(X=1) &= \tfrac{1}{2} \\ P(X=2) &= \tfrac{1}{4} \\ P(X=3) &= \tfrac{1}{8} \\ P(X=4) &= \tfrac{1}{8} \end{aligned} \tag{4}$$

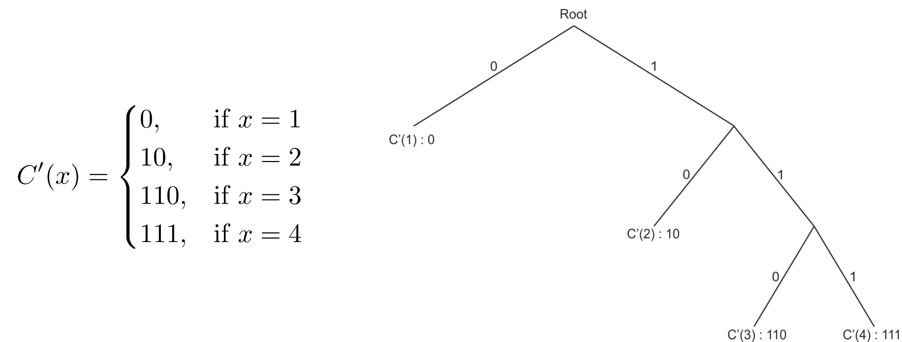

Figure 2: Optimal code $C'(x)$ for probability distribution in (4)

For any probability distribution the optimal code is given by Huffman coding [10]. However in this thesis we are not interested in actual codes, but in theoretical compression and the relation to Shannon Entropy.

We can consider the requirement for code lengths l_x to be integer values as a practical limitation of binary codes. Dropping this requirement, to find the optimal code lengths l_x^* is to minimize the expected code length (3) with the code lengths satisfying Kraft's inequality. In section 5.3 of their book Cover et al. [4] show that this leads to the following optimal code lengths.

$$l_x^* = -\log p(x)$$

Inserting this into the formula for the expected code length, we find that the expected code length for a (non-integer) optimal prefix code for X is the same as the entropy of X.

$$L_X = -\sum_{x \in \mathcal{X}} p(x) \log p(x) = H(X) \tag{5}$$

Shannon entropy thus gives us the shortest expected code length we can achieve with a prefix code, when unhindered by the practical limitations of actualized binary codes. This makes Shannon entropy a measure of information in that it gives us a lower bound for the average amount of bits required to describe data.

2.1.2 Minimum Description Length Principle

The second important concept from Information Theory to discuss is the Minimum Description Length (MDL) principle [14, 6]. The MDL principle uses compression to choose between multiple models for explaining some observed data set. The idea is that the model which can describe the data using the fewest bits is the model which best fits the data. MDL avoids selecting complex and likely overfitted models by requiring the models themselves to be compressed as well. The version of MDL discussed and used in this thesis is known as two-part MDL or crude MDL [6]. Here the data and model are compressed separately. Given a set of models $\mathcal{M}$ for some data set $\mathcal{D}$ the best model $M \in \mathcal{M}$ for $\mathcal{D}$ is the one which minimizes:

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

Where $L(M)$ is the length in bits required to describe the model M, and $L(\mathcal{D}|M)$ is the length of the description of the data $\mathcal{D}$ when encoded with M. Many of the algorithms I discuss in this thesis use the MDL principle for model selection, the first example of how to put MDL into practice will be the KRIMP algorithm in subsection 2.2.2.

2.2 KRIMP

In this section I will present the KRIMP algorithm [16, 21], which is based on the MDL principle. The algorithm was designed to solve a problem in Frequent Itemset Mining known as pattern explosion. In the next subsection I will explain frequent itemset mining and pattern explosion, after which I present the KRIMP algorithm.

2.2.1 Frequent Itemset Mining

The subject of frequent itemset mining was introduced by Agrawal et al. [1]. As mentioned earlier, the problem can be summarized as finding patterns of frequently co-occurring products in a database of supermarket transactions. These patterns are called *frequent itemsets* and can be used to give insight into buying habits of customers.

The problem can more formally be described as follows, consider the supermarket products to be represented by a set of literals $\mathcal{I}$ referred to as items

$$\mathcal{I} = \{I_1, I_2, ..., I_m\}$$

Transactions can be seen as a set of purchased products, which is a subset of the items $t \subseteq \mathcal{I}$. A transaction database $\mathcal{D}$ is a bag of transactions

$$\mathcal{D} = \{t_1, t_2, ..., t_n\}$$

The objective of frequent itemset mining is to find frequently occurring patterns in a transaction database.

The patterns we are looking for are called *itemsets*, an itemset X is a subset of items, $X \subseteq \mathcal{I}$. If an itemset X is a subset of a transaction t, we say t *supports* X. The support of an itemset X by a database $\mathcal{D}$ is written as $supp_{\mathcal{D}}(X)$ and gives us the number of transactions $t \in \mathcal{D}$ which support X.

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

Table 1 gives an example of a transaction database, and Table 2 shows the support $supp_{\mathcal{D}}(X)$ of this database for some itemsets.

$\mathcal{D}$	$\mathcal{I} = \{A, B, C, D, E\}$
t_0	$\{A, B, D\}$
t_1	$\{C, E\}$
t_2	$\{C, D, E\}$
t_3	$\{B, D\}$
t_4	$\{A, B, C, E\}$
t_5	$\{D\}$

Table 1: Example Database

X	$supp_{\mathcal{D}}(X)$
$\{B, D\}$	2
$\{C, E\}$	3
$\{A\}$	2
$\{B\}$	3
$\{C\}$	3
$\{D\}$	4
$\{E\}$	3

Table 2: $supp_{\mathcal{D}}(X)$ for some example itemsets

The objective of frequent itemset mining is to find the set $\mathcal{F}$ of all *frequent* itemsets X, where an itemset X being frequent means $supp_{\mathcal{D}}(X)$ is above some minimum support threshold *minsup*.

$$\mathcal{F}_{\mathcal{D}} = \{X \in \mathcal{P}(\mathcal{I}) \mid supp_{\mathcal{D}}(X) \geq minsup\}$$

To find the set of frequent itemsets we can make use of a monotonic property known as the A Priori property, which I will briefly discuss.

The A Priori Property

For a database over $\mathcal{I}$ there are $2^{|\mathcal{I}|} - 1$ itemsets[1], which could all potentially be frequent depending on the *minsup* threshold. The size of the search space makes it important to have efficient algorithms. The first efficient algorithms for finding frequent itemsets [2] make use of the A Priori property. For an itemset Y which is a superset of X, the set of transactions supporting Y is a subset of the set of transactions supporting X.

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

The implications of this property are that if a set X is infrequent then none of its supersets are frequent. Conversely for a set Y to be frequent all of its subsets must be frequent. This property allows for a bottom up level-wise search, only checking support for itemsets of size k for which all subsets of size $k-1$ were found to be frequent in the previous stage. The A Priori property has since been used to design more efficient algorithms such as Eclat [22] (depth-first search) and FP-Growth [7] (transforming $\mathcal{D}$ to an efficient tree structure).

While the A Priori property allows for efficient mining algorithms it comes with the downside of pattern explosion. If we mine itemsets with a high *minsup* we get very few, but likely well-known, and thus not very interesting patterns. But if we lower the *minsup* threshold the amount of resulting patterns grows rapidly, resulting in more patterns than can be manually interpreted. Because of the subset/superset relation between frequent itemsets there will be a lot of redundancy in the results. In the next subsection I will discuss the KRIMP algorithm's approach to filtering out redundant itemsets.

2.2.2 KRIMP: MDL for Frequent Itemsets

KRIMP's [16, 21] solution to pattern explosion is to use itemsets for compression of a database. The redundancy of an itemset is evaluated by how helpful it is in achieving better compression. Itemsets which don't aid compression must be redundant, and are thus not interesting enough to report. This allows Siebes et al. [16] to reduce the output of frequent itemset mining algorithms to a much smaller and more interesting subset.

[1] not counting the empty set

Recall that the objective of Minimum Description Length is to find the model M which minimizes:

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

The models KRIMP is concerned with are called *code tables*. A code table CT is a two-column table, with the left column containing itemsets and the right column containing code words. It represents a prefix code for itemsets, but is only used as a means to calculate the theoretically achievable compression of a model. Because of this we never look for actual code words in the right column, instead we consider their (not necessarily integer) code lengths.

In the following subsections I will explain how a code table is used to encode a transaction, how to calculate the amount of bits required to encode a database $\mathcal{D}$, how to compress the code tables themselves, and how to find code tables with good compression. The MDL objective written in terms of code tables is to minimize:

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

Cover Function

We can describe a transaction by selecting a non-overlapping set of itemsets from the left column of a code table, until the union of these itemsets matches the items in the transaction. Such a description is called a *cover*. Any valid cover must satisfy the following conditions:

$$X \in cover(CT, t) \implies X \in CT$$
$$X, Y \in cover(CT, t) \implies X \cap Y = \emptyset \text{ or } X = Y$$
$$t = \bigcup_{X \in cover(CT,t)} X$$

To ensure that there is always at least one way for a code table to cover any transaction $t \subseteq \mathcal{I}$ it is required that CT contains the singleton itemsets for all items $i \in \mathcal{I}$. There are often multiple ways to encode a transaction with a code table, to have a non-ambiguous way of encoding transactions we require a *cover function*. KRIMP's cover function is quite simple, the itemsets of the code table are ordered based on what they call the *Standard Cover Order* and every transaction is encoded by greedily picking itemsets from the code table, provided they don't overlap with previously selected cover elements. The standard cover order is to sort the itemsets first descending on cardinality, second descending on their support by $\mathcal{D}$ and third lexicographically. This is represented by the following notation [21]:

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

Compressing the Database

After covering all transactions $t \in \mathcal{D}$ we can count how often each itemset gets used.

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

I will occasionally abuse notation to avoid cluttered equations, writing $usage_{CT}(X)$ when I mean $|usage_{CT}(X)|$. It should generally be clear from context whether I'm referring to the set of transactions or the amount of transactions in the set. The usage counts of all itemsets in CT together form a probability distribution.

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

As discussed in section 2.1.1 the code lengths for minimizing the average required bits to encode, given a probability distribution, coincide with the formula for Shannon entropy. Therefore the optimal code length for an itemset X is

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

We know how often each itemset is used to describe $\mathcal{D}$ with CT and we know the code lengths for each itemset, so we can calculate the compressed size of $\mathcal{D}$ using CT by multiplying the code lengths with the usage counts.

$$L(\mathcal{D} \mid CT) = \sum_{X \in CT} usage_{CT}(X) L(X \mid CT)$$

Compressing the Code Table

Two-part MDL not only requires compression of the data, but also of the model itself. In order to compress a code table CT we need to compress both columns, the itemsets and the codes.

The (unmaterialized) codes are already in binary format and don't need further compression, their code lengths are simply added to the compression of the model. The itemsets are compressed using another, simpler code table, this is possible since itemsets just like transactions are simply sets of items. They are compressed with a code table using only singleton itemsets, called the *Standard Code Table* CT_{ST}. The code lengths of CT_{ST} are determined by covering $\mathcal{D}$ with only the singleton itemsets, as a result items with high support get shorter codes.

Itemsets in the code table with a usage of 0 are not used for compressing $\mathcal{D}$ and are therefore not included in the compression of the model. The above leads to the following formula for the amount of bits required to encode CT

$$L(CT \mid \mathcal{D}) = \sum_{\substack{X \in CT \\ usage_{CT}(X) \neq \emptyset}} L(X \mid CT_{ST}) + L(X \mid CT)$$

Finding Good Code Tables

We've discussed how to use a code table to unambiguously encode a transaction database $\mathcal{D}$, and how to calculate the corresponding amount of bits required for this encoding. The remaining task is to find the model giving the shortest encoding $L(\mathcal{D}, CT)$. Naïvely one could try all possible code tables, but this is not a feasible approach. According to Vreeken et al. [21] there are

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

possible code tables. This search space is considered too large and not structured enough to allow for an efficient way of finding the optimal code table. Instead the KRIMP algorithm relies on a greedy search heuristic.

The algorithm begins with the standard code table CT_{ST} and requires a set of candidate itemsets $\mathcal{F}$ to be mined from $\mathcal{D}$ beforehand. These candidates are sorted in *Standard Candidate Order*, first descending on support, second descending on size and then lexicographically.

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

In this order candidates are iteratively added to the code table to see if their addition improves compression. If a candidate itemset improves the total compression it is accepted into the code table, otherwise it is permanently discarded. After each acceptance KRIMP tries to prune previously added code table elements. This keeps the code table small and lowers redundancy among the itemsets in the code table. Pruning is also done in the same greedy manner, but in order of the usage of itemsets, beginning with the itemsets with lowest usage. Itemsets which no longer help with compression are discarded.

2.3 SLIM

Vreeken et al. [21] suggested that the search space of all possible code tables is unstructured. While it may be too unstructured to find a guaranteed best code table the SLIM algorithm [17] shows that there is enough structure for improvement over KRIMP. The SLIM algorithm does not use a previously mined set of candidate itemsets, and therefore requires no *minsup* threshold to be chosen. Instead SLIM generates its candidate itemsets on the fly based on itemsets present in the code table. Because candidates are based on code table elements they are able to make accurate estimates of how adding a candidate would change compression. They manage to gain better compression, while nearly always being faster than KRIMP as well.

At every step of the algorithm the candidate set $\mathcal{F}$ is based on the itemsets in CT at that time. Each candidate is the union of two elements of the code table:

$$\mathcal{F} = \{X \cup Y \mid X, Y \in CT\}$$

This list is generally smaller than the candidate lists KRIMP is run with, allowing SLIM to also reconsider old candidates rather than permanently discarding them. SLIM manages to outperform KRIMP because their accurate estimates allow them to focus on good candidates first.

Estimating Compression Gain

The gain in compression after adding an itemset $X \cup Y$ is simply old compression, $L(\mathcal{D}, CT)$ minus the new compression $L(\mathcal{D}, CT')$. This compression relies almost[2] entirely on the usage counts of the itemsets in both code tables. While for most itemsets $Z \in CT$ it is hard to predict the difference from $usage_{CT}(Z)$ to $usage_{CT'}(Z)$ there are three exceptions. For candidate $X \cup Y$ we know that it will (at least) be used in every cover where previously both X and Y were used, as the candidate places higher in the standard cover order. This gives us three bounds for usage counts in CT'.

$$usage_{CT'}(X \cup Y) \geq usage_{CT}(X) \cap usage_{CT}(Y)$$
$$usage_{CT'}(X) \leq usage_{CT}(X) - usage_{CT'}(X \cup Y)$$
$$usage_{CT'}(Y) \leq usage_{CT}(Y) - usage_{CT'}(X \cup Y)$$

The authors of SLIM found that assuming equality, with all other usage counts remaining the same allowed them to make very accurate estimates of the compression gain, while requiring little computation.

Finding the usage intersection $usage_{CT}(X) \cap usage_{CT}(Y)$ is a potentially expensive computation, in the worst case one would have to go over the entire database. Because the amount of transactions in the usage intersection of two itemsets is bounded by the usage of the individual itemsets SLIM can avoid computing many intersections.

$$|usage_{CT}(X) \cap usage_{CT}(Y)| \leq usage_{CT}(X)$$
$$|usage_{CT}(X) \cap usage_{CT}(Y)| \leq usage_{CT}(Y)$$

Generating the candidates in order of their parent-sets usage counts SLIM keeps track of the top-k candidate estimates, and only computes the usage intersection if it has the potential to improve on the top-k estimates.

[2] Almost, as compressing the itemsets themselves does not rely on $usage_{CT}(X)$

2.4 Identifying the Components

As mentioned at the end of section 2.2.2 KRIMP can be used for other tasks than filtering out redundant itemsets. van Leeuwen et al. [19] use code tables to partition a database into what they call "highly characteristic components". The authors state that databases often consist of samples of different distributions. And that a model which takes into account the underlying distributions is often superior to one that does not. Following the MDL principle, a better model should lead to better compression.

The basic idea of their approach is to let k partitions be encoded by different code tables and iteratively refine the partitioning/code tables to allow for better compression. The authors give two separate approaches for partitioning with KRIMP code tables. Both approaches require running the partitioning algorithm for all $k \in [2, |\mathcal{D}|]$ and selecting the partitioning with the shortest compression, assuming that compression is better than the original code table which corresponds to $k = 1$.

Both methods are reported to provide partitionings with much shorter data descriptions while dividing the data into highly dissimilar components. Because the optimal partitioning is determined by MDL no parameter k has to be chosen, making this an attractive approach when there is little to no prior knowledge about the data. Additionally the authors suggest that their framework can be adapted to function with other MDL methods without much work.

2.4.1 Model-Driven Partitioning

The first algorithm they propose is what they call a *Model-Driven* approach. A code table induced for a complete database should capture the entire distribution, so the distributions of the multiple underlying components should be modelled by this code table implicitly. This suggests that each of the underlying components could be modelled efficiently using subsets of the original code table.

The Model-Driven algorithm creates k copies of the original code table and iteratively removes itemsets from the code tables to improve compression. To calculate the total compression of the database every transaction is assumed to be encoded by the code table which gives it the shortest encoded length. All code lengths are calculated with a Laplace correction[3] to ensure every code table can encode transactions from other partitions.

For removing code table elements the algorithm searches for the best element to remove exhaustively, until no removal further improves compression. This exhaustive search for the best removal makes this method relatively slow compared to the next approach.

[3]increasing $usage_{CT}(X)$ by one

2.4.2 Data-Driven Partitioning

They call their second algorithm a Data-Driven approach. This Data-Driven approach is similar to k-means clustering, but without the need for a distance metric. The algorithm starts by randomly dividing the data over k partitions. After initialization the algorithm iteratively makes code tables for each partition, and reassigns transactions to the code table which provides the shortest encoding. This process is repeated until the partitioning becomes stable, meaning no transactions require reassigning. Because of the random initialization step the algorithm is run multiple times to ensure finding a good solution, however the authors claim that the results of each run are very similar, suggesting that the algorithm is very robust despite the initial randomness.

2.5 Low Entropy Set Selection

Low Entropy Set Selection (LESS) [9] is a variant of KRIMP which works with *low entropy sets* rather than frequent itemsets. Low entropy sets were introduced by Heikinheimo et al. [8] and are close relatives to frequent itemsets. While frequent itemsets only look at the co-occurrence of items low entropy sets look at both presence and absence relations of items. This symmetrical approach should allow for more powerful descriptions of data.

First I will briefly go over the differences and similarities between low entropy sets and frequent itemsets, after which I will discuss the LESS MDL approach [9].

2.5.1 Low Entropy Set Mining

In this context we are still concerned with transaction databases $\mathcal{D}$, bags of transactions, but the transactions and items are represented differently. Previously the items were literals, which were either present or absent in a transaction, here the items $\mathcal{I}$, usually referred to as attributes, are considered to be binary attributes which can take values 0-1. Transactions are no longer subsets of $\mathcal{I}$, but binary vectors of length $|\mathcal{I}|$. The value of a transaction t at the xth index is written as $\pi_x(t)$ and should coincide with an item's presence in the old representation.

$$
\begin{aligned}
I_x \in t_{old} &\iff \pi_x(t) = 1 \\
I_x \notin t_{old} &\iff \pi_x(t) = 0
\end{aligned}
$$

Previously we called sets of items $X \subseteq \mathcal{I}$ itemsets, now we call them *attribute sets*. Because every transaction now contains all attributes we don't look at the support of an attribute set, but at the entropy. An attribute set X and a transaction t together determine a binary vector $\pi_X(t)$ of values $\pi_A(t)$ defined by the attributes $A \in X$. Ω_X is defined to be the set $\{0,1\}^{|X|}$ of all 0-1 vectors

of length $|X|$ [8]. The vectors $i \in \Omega_X$ are called the instantiations of X. An instantiation i fits a transaction t iff $i = \pi_X(t)$. Counting how often each instantiation $i \in \Omega_X$ fits a transaction $t \in \mathcal{D}$ we get a probability distribution.

$$p_X(i) = \frac{|\{t \in \mathcal{D} \mid i = \pi_X(t)\}|}{|\mathcal{D}|}$$

From the probability distribution over the instantiations of an attribute set X we can compute the entropy of X.

$$H(X) = -\sum_{i \in \Omega_X} p_X(i) \log p_X(i)$$

$\mathcal{D} \setminus \mathcal{I}$	A	B	C	D	E
t_0	1	1	0	1	0
t_1	0	0	1	0	1
t_2	0	0	1	1	1
t_3	0	1	0	1	0
t_4	1	1	1	0	1
t_5	0	0	0	1	0

Table 3: Example Database

X	$H(X)$
$\{B, D\}$	1.92
$\{C, E\}$	1
$\{A\}$	0.92
$\{B\}$	1
$\{C\}$	1
$\{D\}$	0.92
$\{E\}$	1

Table 4: Entropy example for Table 3

Attribute sets with low entropy have a skewed probability distribution, meaning certain value combinations occur significantly more often than others. Such attribute sets might provide valuable insight into the data, similar to how frequent itemsets can be interesting. Given a maximum entropy threshold ϵ we call an attribute set X a low entropy set if $H(X) \leq \epsilon$.

A Priori Property for Low Entropy Sets

Similar to frequent itemsets, low entropy sets have a monotonicity property [8]. The entropy of an attribute set Y is greater or equal to the entropy of its subsets X.

$$X \subseteq Y \Rightarrow H(Y) \geq H(X)$$

As with itemset mining, this property allows for very similar efficient pattern mining algorithms, Heikinheimo et al. [8] propose a level-wise search, Heikinheimo et al. [9] use a depth-first search similar to Eclat [22], and very recently Pennerath [13] proposed an approach inspired by FP-growth [7]. While this property allows for similar efficient mining strategies, it also causes low entropy set mining to suffer from pattern explosion. Similar to how frequent itemset mining algorithms have been adapted to work with low entropy sets LESS is an adaptation of KRIMP to solve pattern explosion for low entropy sets.

2.5.2 LESS: MDL for (Low Entropy) Attribute Sets

Like KRIMP, LESS [9] also uses code tables for models, now with attribute sets in the left column, and codes in the right column. Unlike KRIMP this is not enough to describe a database. The content of a transaction can't be described by attribute sets alone, as an attribute set says nothing of the values of attributes. Instead LESS uses pairs of attribute sets and instantiations to cover transactions. These instantiations are part of a second code table, with instantiations in the left column and code lengths on the right. The code table containing the attribute sets will be referred to as CT_{LE}, the code table for instantiations as CT_I. Valid covers for a transaction t can't contain overlapping attributes between attribute sets, for a cover element (X, i) instantiations are determined by the values of t for the attributes of X, all attributes of t must be covered.

$$
\begin{aligned}
&(X, i) \in cover(t) \implies X \in CT_{LE} \text{ and } i \in CT_I \\
&(X, i), (Y, j) \in cover(t) \implies X \cap Y = \emptyset \text{ or } X = Y,\ i = j \\
&t = \bigcup\nolimits_{(X,i) \in cover(t)} \pi_X(t) = i
\end{aligned}
$$

To make sure any possible transaction for attributes $\mathcal{I}$ can be encoded the authors require CT_{LE} to contain all singleton attribute sets, and CT_I all instantiations from length 1 to the largest attribute set in CT_{LE}.

$$
CT_I = \bigcup_{l=1}^{\max_{X \in CT_{LE}} |X|} \{0, 1\}^l
$$

Because the cover function from Heikinheimo et al. [9] is more complicated than the one used for KRIMP I will first discuss the models and how to do compression with them, before I explain how the covers are determined.

Compressing the Database

Since LESS uses two code tables we break the objective of MDL into four parts, the task is to minimize:

$$
\begin{aligned}
&L(M) + L(\mathcal{D} \mid M) \\
&Where \\
&L(\mathcal{D} \mid M) = L(\mathcal{D} \mid CT_{LE}) + L(\mathcal{D} \mid CT_I) \\
&L(M) = L(CT_{LE} \mid \mathcal{D}) + L(CT_I \mid \mathcal{D})
\end{aligned}
$$

The usage of attribute sets X and instantiations i is determined by how often they occur in the cover of a transaction, written as follows[4]:

$$usage_{CT_{LE}}(X) = \{t \in \mathcal{D} \mid X \in cover(t)\}$$
$$usage_{CT_I}(i) = \{t \in \mathcal{D} \mid i \in cover(t)\}$$

As with KRIMP the usage counts of the code tables form probability distributions, which are used to calculate the optimal code lengths. The attribute sets are treated the same as itemsets in KRIMP.

$$P(X \mid \mathcal{D}, CT_{LE}) = \frac{usage_{CT_{LE}}(X)}{\sum_{Y \in CT_{LE}} usage_{CT_{LE}}(Y)}$$
$$L(X \mid CT_{LE}) = -\log(P(X \mid \mathcal{D}, CT_{LE}))$$

For the instantiations in CT_I remember that to ensure we can encode any transaction, CT_I must contain all instantiations of size 1 to the largest set in CT_{LE}, even if they are not used in any covers for $\mathcal{D}$. To give these zero-usage instantiations a non-infinite code length we have to apply a Laplace correction[5]. All usages are increased by 1 in the definition of the probability distribution for the instantiations.

$$P(i \mid \mathcal{D}, CT_I) = \frac{usage_{CT_I}(i) + 1}{|CT_I| + \sum_{j \in CT_I} usage_{CT_I}(j)}$$
$$L(i \mid CT_I) = -\log(P(i \mid \mathcal{D}, CT_I))$$

With this we can calculate the encoded size of $\mathcal{D}$ as

$$L(\mathcal{D} \mid CT_{LE}) = \sum_{X \in CT_{LE}} usage_{CT_{LE}}(X) L(X \mid CT_{LE})$$

$$L(\mathcal{D} \mid CT_I) = \sum_{i \in CT_I} usage_{CT_I}(i) L(i \mid CT_I)$$

Note that the Laplace corrections are only present in the definition for $P(i \mid \mathcal{D}, CT_I)$ so we can assign codes to unused instantiations, these do not affect the actual usage counts.

Compressing the Code Tables

The attribute set code table CT_{LE} is compressed analogously to how KRIMP encodes the code table for itemsets. The code lengths are summed, and the attribute sets are encoded by the standard code table, the simplest code table containing only singleton attribute sets. In this context the standard code table

[4] for $usage_{CT_I}(i)$ interpret it to be a bag of transactions rather than a set, considering that a transaction can use the same instantiation multiple times in its cover

[5] This is not mentioned in[9], but I have verified that this is indeed what they do

will assign equal length codes to each attribute, as every transaction contains every attribute $A \in \mathcal{I}$.

$$L(code_{ST}(X)) = |X| \log(|\mathcal{I}|)$$

$$L(CT_{LE} \mid \mathcal{D}) = \sum_{x \in CT_{LE}} L(X \mid CT_{LE}) + L(code_{ST}(X))$$

For compressing CT_I again the code lengths are summed, the instantiations are encoded by their bit-representation. This means the instantiation $i = \langle 0, 1, 0 \rangle$ gets "compressed" to '010' thus $bit(i) = 3$. All instantiations are taken into account when encoding the model $L(CT_I)$, including instantiations with $usage_{CT_I}(i) = 0$.

$$L(CT_I \mid \mathcal{D}) = \sum_{i \in CT_I} L(i \mid CT_I) + bit(i)$$

Cover Function

The cover function for LESS is very different from the one used by KRIMP. The reason for this is that for every transaction t and attribute set X there is an instantiation i which can make X fit t. If we were to simply impose an ordering on CT_{LE} and cover transactions with non-overlapping attribute sets we would always use the same attribute sets in every cover. Instead the authors propose a cover function which considers the attribute sets in a different order for each transaction. For this they use the maximum likelihood principle. Taking the cover C that maximizes the conditional probability $p(t \mid C)$ of the transaction. The loglikelihood of a transaction t given a cover C is:

$$llh(t, C) = \sum_{X \in C} \log p(\pi_X(t))$$

The cover which maximizes the likelihood is called the optimal cover

$$C^{opt} = \arg\max_{C} llh(t, C)$$

The code length for the instantiation matching $\pi_X(t)$ should then be proportional to its negative loglikelihood. This would make minimizing the negative loglikelihood a way to optimize the code lengths and thus a way to optimize compression.

Because finding the cover which optimizes the likelihood is an NP-complete problem [9] they use a greedy heuristic to approximate the optimal cover. Given a transaction t each attribute set X is assigned a weight $w(t, X)$ called the *per attribute likelihood addition*, which is the loglikelihood of $\pi_X(t)$ divided by the amount of attributes in X.

$$w(t, X) = \frac{\log\left(p(\pi_X(t))\right)}{|X|}$$

For each transaction t the cover function orders the attribute sets descendingly on these weights, greedily selecting $(X, \pi_X(t))$ to cover if none of the previously selected attribute sets overlap with X. The LESS cover function makes an exception for singleton attribute sets [6]. Singleton sets are not taken into account in the cover order, instead a non-singleton set X can only be used to cover to cover t iff the likelihood of $\pi_X(t)$ is higher than the likelihood of the combined singletons.

$$useful(t) = \{X \mid p(\pi_X(t)) \geq \prod_{a \in X} p(\pi_a(t))\}$$

Any attributes not yet covered after considering all useful non-singleton attribute sets will be covered by the singleton attribute sets. Without this feature singleton attribute sets will be used more often, leading to many low usage attribute sets being added to the solution. Doing this requires little extra work and can only improve the likelihood of a cover, making it a better approximation of the optimal cover.

Finding Good Code Tables

The only remaining topic is how LESS attempts to find a good code table. This is done in the same way KRIMP does. Candidate attribute sets are mined beforehand with some threshold ϵ and iteratively added to the code table, accepting candidates if compression improves, rejecting them if not. The order in which candidates are considered by LESS is entropy over size, $H(X)/|X|$. This should put attribute sets with high likelihood addition at the top of the candidate list, meaning candidates with high expected usage will be considered first. LESS also prunes the code table after each addition, attempting to remove potentially redundant attribute sets from CT_{LE}.

The authors claim LESS finds very short, high quality descriptions of the data, needing one to two orders of magnitude fewer attribute sets than KRIMP requires itemsets.

[6] A crucial part of the algorithm, unmentioned in the original paper [9]

3 Research Question

In Section 2 I've discussed the KRIMP algorithm [16, 21] which provides a solution to pattern explosion for frequent itemset mining. This algorithm was improved on by the SLIM algorithm [17], and used for partitioning by van Leeuwen et al. [19]. The LESS algorithm [9] provides an approach similar to KRIMP for low entropy attribute sets. There are many similarities between frequent itemset mining and low entropy set mining, and many algorithms for frequent itemsets can be applied to low entropy sets. This makes it interesting to investigate whether approaches which build on the KRIMP algorithm can be adapted to work with the LESS algorithm. The MED algorithm [5] has shown that entropy can be used as an effective tool for partitioning, but provides an ad hoc approach. This inspired the following research question:

Can we create a generally applicable entropy-based partitioning method using LESS code tables?

3.1 Subquestions

To answer this question I have divided my research into three sections aiming to answer the following three subquestions.

- **Can LESS be improved with runtime candidate generation?**
  The SLIM algorithm improved on KRIMP, providing a faster approach with better compression rates. The way SLIM achieved this is through runtime candidate generation and accurate estimation of compression gain. I want to investigate whether the same improvements can be made for LESS.

- **Is it possible to translate the Data-Driven and Model-Driven methods by van Leeuwen et al. [19] to the context of LESS code tables?**
  van Leeuwen et al. suggest their partitioning methods, especially the Data-Driven one, should be straightforward to adapt to other MDL approaches. To use LESS code tables for partitioning I will investigate how straightforward this adaptation is.

- **How does partitioning with LESS code tables compare to partitioning with KRIMP code tables?**
  Frequent itemsets only consider presence relations between items, whereas low entropy sets consider data symmetrically. This allows low entropy sets to capture more types of variable interactions than frequent itemsets. It's interesting to explore whether this makes LESS code tables better at partitioning than KRIMP code tables.

4 Improvements to LESS

In this section I aim to answer the first subquestion:
Can LESS be improved with runtime candidate generation?

In order to answer this question I will use the SLIM algorithm [17] as a guideline, attempting to implement their changes to KRIMP to LESS in similar fashion. I will discuss how to make LESS suitable for runtime candidate generation, and what is required to make accurate estimates for compression gain.

4.1 Candidate Generation

Following SLIM's approach we can generate candidate attribute sets by combining code table elements from CT_{LE}. The set of candidates $\mathcal{F}$ is then defined as

$$\mathcal{F} = \{X \cup Y \mid X, Y \in CT_{LE}\}$$

KRIMP, SLIM and LESS have all used the standard code table as the starting point of their algorithm, it makes sense to do the same here. As a result, initially all sets $X \in CT_{LE}$ will be singletons. This means for our first set of candidates $\mathcal{F}$, all attribute sets $Z \in \mathcal{F}$ contain two attributes $|Z| = 2$.

The reason SLIM is an improvement over KRIMP is because of their compression gain estimates, allowing them to mostly try good candidates. However the algorithm should work without these estimates as well, albeit suboptimally. We might not always pick the optimal candidate, or reject more candidates before accepting, but adding a candidate to the code table will have the same effect on compression. This means we should be able to run LESS using this dynamic candidate set, but this already proved to be problematic. In all of my initial tests, none of the candidates $Z \in \mathcal{F}$ of size $|Z| = 2$ gave any improvement in compression, even when used in every cover of every transaction. The reason for this is that the code lengths in CT_I are causing unnecessary bias in the compression gain of candidates.

4.2 Biased Code Lengths in CT_I

I will try to provide an explanation for the problem. Consider we have a database $\mathcal{D}$ with n transactions over m attributes. Initially each attribute set is used in the cover of each transaction leading to all code lengths being $\log(m)$. After adding a candidate $X \cup Y$, assuming the best case scenario, X and Y are no longer used and all other code lengths become $\log(m-1)$. This will always improve $L(\mathcal{D} \mid CT_{LE})$.

For CT_I the effect of adding $X \cup Y$ is very different, even in this best case scenario. Where previously CT_I contained only two instantiations, $\langle 0 \rangle$ and $\langle 1 \rangle$, it now contains six instantiations, adding $\langle 0, 0 \rangle$, $\langle 0, 1 \rangle$, $\langle 1, 0 \rangle$ and $\langle 1, 1 \rangle$,

all sharing the same prefix code. If we ignore Laplace corrections to simplify matters, we can write $L(\mathcal{D}|CT_I)$ as the total usage of the code tables, multiplied by the average expected bits of the code table, its entropy, the same for the new "best case" code table CT'_I.

$$L(\mathcal{D} \mid CT_I) \approx nmH(CT_I)$$
$$L(\mathcal{D} \mid CT'_I) \approx n(m-1)H(CT'_I)$$

For CT'_I to improve on CT_I, $L(\mathcal{D} \mid CT'_I) \leq L(\mathcal{D} \mid CT_I)$ this reduces to the following inequality

$$H(CT'_I) \leq \frac{mH(CT_I)}{m-1}$$

While this is far from a formal proof, it shows that unless $\mathcal{D}$ consists of very few attributes[7] this means $H(CT'_I)$ must be near equal to $H(CT_I)$ (or lower). But the new CT'_I contains six instantiations where previously CT_I contained two. Generally speaking the only way for the entropy to become lower is that one of the two singleton instantiations is (nearly) entirely replaced by one of the new instantiations. In that case, the rest of the database must have only used the other singleton instantiation, that is obviously not an interesting database to analyse. In this explanation I ignored that the gain in $L(\mathcal{D} \mid CT'_{LE})$ could in theory outweigh this loss in compression, because I found that in practice this did not happen.

The reason LESS avoids this problem is because it considers larger candidates first. Larger candidates are more likely to replace enough singletons in CT_{LE} to outweigh the initial negative effects on CT_I. After a while many singletons have been replaced and sets of size 2 no longer harm the compression and can be added to the code table. However if we would inflate a database with extra attributes, while keeping the candidate set the same, LESS would also never accept a candidate.

The way CT_I is defined causes LESS to be biased towards attribute sets which have the same cardinality as others already in CT_{LE}. A singleton instantiation can never be paired with a non-singleton attribute set Z, yet it still influences the code lengths for instantiations of length $|Z|$. Having many singleton attribute sets in CT_{LE} leads to longer code lengths for non-singleton instantiations. I consider this bias to be unnecessary, leading to suboptimal compression, and likely worse code tables. In the next subsection I will discuss two alternate definitions for CT_I to reduce this bias.

[7] In which case why use LESS as the number of low entropy sets would be small enough to inspect manually

4.2.1 LESS-Per-Length

The first solution to remove bias from CT_I stays close to the original definition from LESS, but accounts for the fact that an attribute set of size $|Z|$ can only be paired with instantiations $i \in \{0,1\}^{|Z|}$. The idea is to split CT_I into multiple code tables IT, such that there is a separate code table for each length of instantiation up to the largest set $X \in CT_{LE}$.

$$IT^l = \{0,1\}^l$$

$$CT_I = \{IT^l \mid l \in \bigcup_{X \in CT_{LE}} |X|\}$$

Because instantiations will share a prefix code only with instantiations of equal length the code lengths will be shorter, determined by the next distribution:

$$P(i \mid \mathcal{D}, IT^{|i|}) = \frac{usage_{IT^{|i|}}(i) + 1}{|IT^{|i|}| + \sum_{j \in IT^{|i|}} usage_{IT^{|i|}}(j)}$$

Using this definition of CT_I adding a two-attribute candidate to the singleton code table is no longer (practically) guaranteed to worsen compression. I will refer to the version of LESS using such code tables as LESS-Per-Length or LESS-PL for short.

4.2.2 LESS-Individual-Instantiations

Even LESS-PL has some potentially unnecessary bias in the code lengths of CT_I, which can be illustrated as follows. Take three attribute sets X, Y, Z of equal cardinality, suppose the cover function only uses each attribute set for one (separate) instantiation of high likelihood. Because of their high likelihood these instantiations are supposed to get relatively short code lengths. Since each attribute set is used with only one instantiation, you could even argue that a good model should assign a code length of (near) zero to the instantiations. In LESS-PL, because the attribute sets use different instantiations from the same code table $IT^{|X|}$ the instantiations all negatively affect each other's code lengths. As a result, despite the high likelihoods, it could very well have been better to use some differently sized attribute sets with a lower likelihood addition, to create code tables with better compression.

My solution for this problem is once again to use even more code tables, redefining CT_I to contain a separate code table for each attribute set in CT_{LE}.

$$CT_I = \{IT^X \mid X \in CT_{LE}\}$$

Code lengths will now be based on the following probabilities

$$P(i \mid \mathcal{D}, IT^X) = \frac{usage_{IT^X}(i) + 1}{|IT^X| + \sum_{j \in IT^X} usage_{IT^X}(j)}$$

This version of LESS will be referred to as LESS-Individual-Instantiations or LESS-II.

There are often multiple attribute sets of the same size, giving each attribute set its own code table sounds like a lot of extra work, but in practice it's not. An implementation of these code tables will not have to explicitly model all instantiations, it suffices to only keep track of the ones which are actually used. The LESS cover function only uses a handful of different instantiations per attribute set, so this should remain manageable. In addition pruning should also keep CT_{LE} small.

4.2.3 Which variant should we use?

Both of the variants just introduced solve the bias problem which obstructed us from using runtime candidate generation, but which of them is better?

Given the same CT_{LE} the compression with LESS-PL is always better than LESS, since CT_I still contains exactly the same instantiations, only the codes are shorter. The same can't be guaranteed for LESS-II, as each attribute set creates some overhead by needing a new code table, increasing the model size $L(CT_I \mid \mathcal{D})$. Due to the Laplace corrections it is also not guaranteed that the data compression $L(\mathcal{D} \mid CT_I)$ will be more compact. However in general, given that an attribute set is used often enough, the data compression of LESS-II should be shorter due to code lengths of CT_I being tailored to specific attribute sets.

Because it is difficult to objectively say which variant provides better models I will from this point onward consider both, though this does not matter for most of the choices made in the coming sections. The variants are experimentally compared in Section 6.

4.3 Estimating Candidate Compression Improvement

To make good estimates on the change in compression after adding a candidate we rely on good estimates of the changes in usage counts after adding a candidate. For LESS we must accurately estimate both the changes in CT_{LE} and in CT_I, as compression gained in $L(\mathcal{D}, CT_{LE})$ can easily be undone by changes in CT_I.

4.3.1 Consequences of the Cover Function

The usage counts of attribute sets and instantiations depend on the cover function. The simplicity of the cover function used by KRIMP [21] has multiple

benefits for the SLIM algorithm [17]. Because itemsets are ordered by cardinality a candidate will always be used in a cover if previously both parent itemsets were used. The ordering on cardinality also make its impossible for a candidate to replace a larger itemset in a cover, as the larger itemsets are considered first. This means a candidate can only lower the usage of smaller itemsets, generally this is a good thing for KRIMP code tables. Fewer itemsets required per cover lead to lower total usage, which will generally lead to shorter codes. The exception where adding a candidate leads to a higher total usage count is when a candidate partially replaces a smaller itemset in a cover, but the remainder has to be replaced by multiple smaller itemsets. This situation is either rare or has no large effect on compression, as SLIM manages to make good estimates without taking this into account.

The Likelihood Cover Function

The cover function for LESS is more complicated, giving no guarantee that a candidate attribute set replaces its parent sets in every cover where both parents were previously used. Instead a candidate needs to have a better likelihood addition than both parent sets.

$$w(t, X \cup Y) \geq \max\left(w(t, X), w(t, Y)\right)$$

To know these weights we need to know the likelihoods of all instantiations of $X \cup Y$, requiring us to pass over the entire database. This is expensive, but it does tell us exactly for which covers the parent attribute sets will be replaced.

This allows us to make good estimates for the first few additions to the code table, but after CT_{LE} contains a few overlapping attribute sets the estimates become very inaccurate. Candidates with large expected gain are rejected for having negative gain, and at some point candidates with no expected gain will end up improving the solution.

The main reason for this occurs when a candidate overlaps with a larger attribute set. If the candidate has a higher likelihood addition for some transactions it will replace the larger set. Because the candidate set is smaller, the remaining attributes need to be covered by other sets, which increases the total usage count of the code tables, causing longer codes. For SLIM this situation was impossible, as a large itemset could never be replaced in a cover by a new smaller set.

Taking into account this situation would require comparing $w(t, X \cup Y)$ to the likelihood addition of all (larger) overlapping attribute sets. Doing this and then "estimating" the change in compression is effectively the same as adding the candidate to the code table and calculating the actual compression. This shows that the likelihood cover function does not provide enough structure for accurate and efficient estimation.

4.3.2 Using A Different Cover Function?

If we wish to make good estimates to use with runtime candidate generation we are forced to use a different cover function, but should we? And what should be the new cover function?

One cover function I've tried is similar to the *Area ascending* cover function [21], tried for KRIMP. This cover function considers the database as an area that needs to be covered and each cover element as a tile covering certain cells of the database. A cover element (X, i) would be ordered based on the support of (X, i) multiplied by its cardinality, this is the maximum area a cover element can potentially cover. This cover function turned out to be too similar to the likelihood cover function, providing too little structure. I suspect that unless we use the same cover order on CT_{LE} for all transactions we won't be able to make good estimates.

As explained in Section 2.5.2 we can't use the same cover order for every transaction, as we would end up only using the same greedily picked non-overlapping large sets in every cover. This would result in only a handful of patterns, a vastly different kind of reduction in patterns compared to what KRIMP does for itemset mining.

It appears that any cover function will either lack exploitable structure for making good estimates, or significantly simplify the code tables to the point where the algorithm no longer serves its initially intended purpose.

4.4 Conclusions

Can LESS be improved with runtime candidate generation?
The original LESS algorithm is not suitable for runtime candidate generation, as small attribute sets are extremely unlikely to improve compression of the standard code table. This can be helped by redefining CT_I as I've done in my suggested variants of LESS-PL and LESS-II. Using either of these variants LESS code tables can be found using runtime candidate generation, however the likelihood cover function provides too little structure to allow for accurate compression estimation. Using a different cover function will change the algorithm to the point where it becomes unsuitable for its original application. I have to conclude that LESS can't be improved with runtime candidate generation, because of this I will continue with the original heuristic for finding code tables, using a set of mined candidates.

5 Partitioning with LESS(-variants)

In this section I aim to answer the most important subquestion:
Is it possible to translate the Data-Driven and Model-Driven methods from van van Leeuwen et al. [19] to the context of LESS code tables?

The section will be split into three different parts. In the first subsection I will investigate how to integrate LESS into the Data-Driven partitioning algorithm [19]. The second subsection similarly considers how to translate the Model-Driven method to the two code table approach of LESS. In the third subsection I will propose what I consider to be the most elegant solution.

5.1 Data-Driven Partitioning for LESS

The Data-Driven approach is supposedly easy to translate to other MDL methods, which would include LESS. The first consideration for this approach is whether we should mine new candidate sets for each partition. Candidates mined for the entire dataset might no longer have low enough entropy in a partition and vice versa. While the problem is slightly different for itemsets[8] I imagine [19] faced the same question and uses the old set of candidates as mining new candidates for each iteration is very computationally expensive.

Formulating a Data-Driven algorithm for LESS should then be trivial. Begin with an initial random partitioning, and iteratively build code tables and reassign transactions, until no transactions need to be reassigned. This turned out to be more problematic than expected. The partitioning often did not stabilize as transactions would continuously change partitions. It was also not simply that transactions would swap between two partitions in a small loop. Instead each iteration lead to a new partitioning, so there were no easily detectable loops. This makes it difficult to justify halting the algorithm prematurely, so instead I investigated what caused this behaviour.

I found that, given some partition with some code table built on the partition, even if only a handful of transactions are added to the partition, the CT_{LE} built on that partition would be completely different. Generally less than 1 out of 10 attribute sets from the old code table would still be used in the new code table. A counterintuitive consequence is that a model M_D built for a partition D, with $t \in D$ can be worse at compressing t than the model $M_{D \setminus t}$ built for the same partition, except t is removed from D. This inconsistency means that similar partitions don't necessarily require a similar amount of bits to compress.

Another observation I made was that $L(\mathcal{D} \mid CT_{LE})$ tends to be much larger than $L(\mathcal{D} \mid CT_I)$ when we partition. One could say this is a good result as low

[8] support can only go down from partitioning data

values for $L(\mathcal{D} \mid CT_I)$ suggest that we use few different instantiations, meaning the data being partitioned well. I would argue this is not the case, and that having CT_{LE} dominate the compression rate is a bad thing. The elements of CT_{LE} are seemingly a bit arbitrary as evidenced by small changes in partitions having large changes on CT_{LE}. To have this arbitrary factor dominate compression, the measure by which we determine the 'optimal' partitioning, can't be desirable.

5.2 Model-Driven Partitioning for LESS

The Model-Driven approach [19] makes copies of the original code table for the entire dataset and prunes the copies so that they can efficiently encode different parts of the data. To translate this approach to LESS code tables some choices have to be made, such as whether to copy only CT_{LE}, CT_I or both.

Copying CT_{LE} seems like a logical first step, as it is the code table most similar to those in KRIMP. Most importantly, it can actually be pruned, whereas CT_I must contain all instantiations per definition.

One problem with copying CT_{LE} stems from a seemingly incomplete formulation of the Model-Driven approach. Transactions are assigned to the code table providing the shortest encoded length, but the cover order of a code table depends secondarily on the support of the data it describes, but initially all partitions are empty. For LESS this problem is even worse as the cover order depends primarily on likelihood. Without a cover order there is no cover, and in turn no way to calculate the encoded length of a transaction. I assume they decided to to use the cover order of the original code table regardless of changes to partitions.

Another problem area is that there is no suggestion on how to break ties when two code tables give identical encoded length, which will be the case for the initial partitioning. One might choose randomly between the best partitions, but this is effectively a random initialization as with the Data-Driven method, as initially all code tables are equal. But for the Model-Driven approach the authors don't suggest doing multiple runs to cancel out randomness. An alternative is to order the partitions and pick the first in the order, but this means one partition will initially contain all transactions. Another way to handle this is to break ties by iteratively assigning transactions to the smallest partition in the tie to more evenly spread the data. If the transactions are ordered this is technically not random, but for LESS this order will have a big influence on the outcome.

At first glance copying CT_I appears strange, part of the definition of CT_I, regardless of which LESS-variant we use is that it contains every instantiation. Surely it can't be very efficient to have multiple completely filled code tables. On the other hand, if we use a single CT_I for all partitions then $L(\mathcal{D} \mid CT_I)$

is unlikely to get much shorter. Because we use the cover order from the original code table most covers remain the same, attribute sets get shorter codes from the CT_{LE} copies, but instantiations remain the same. As a result we are grouping transactions based on using the same attribute sets, while ignoring the valuations on those attribute sets. This means, we should make copies of CT_I.

Even if we ignore the problems that van Leeuwen et al. [19] leave undescribed, following this approach we are likely to run into the same problem we had with the Data-Driven method, which was that CT_{LE} would dominate the compression rates. For this reason I suggest a different approach in the next section, which could be considered a hybrid approach, taking elements from both algorithms.

5.3 Partitioning Algorithm for LESS

The algorithm I propose for finding a good partitioning with LESS code tables is given in pseudocode below. The algorithm is the same for all variants of LESS, but will obviously give different compression rates and results.

Algorithm 1 LESS Partitioning

```
FindOptimalPartitioning(D, F)
 1: CT_OG = LESS(D, F)
 2: for k = 2 to |D| : r_k = FindKPartitioning(D, CT_OG)
 3: r_best = argmin_{k∈[1,|db|]} CompressOptimally(r_k)
 4: return r_best

FindKPartitioning(D, CT_OG)
 1: parts = RandomKPartitioning(D, k)
 2: do
 3:     for each p_i ∈ parts
 4:         CT_i = copy CT_I ∈ CT_OG, set all usage = 0
 5:         for each t ∈ p_i
 6:             IncrementUsageCounts(CT_i, cover_{CT_OG}(t))
 7:         SetLaplaceCorrectedCodeLengths(CT_i)
 8:     for each t ∈ D find i s.t. CT_i gives t shortest code, assign t to p_i
 9: while transactions have been swapped
10: return (parts, tables)
```

The first step is to build a code table CT_{OG} for the entire database $\mathcal{D}$ given some previously mined set of candidates $\mathcal{F}$. This is followed by the partitioning algorithm, executed for $k = [2, |\mathcal{D}|]$.

The partitioning algorithm initially splits $\mathcal{D}$ into k random partitions. Every k-partitioning uses a single attribute set code table CT_{LE}, which is the one

used by the original code table. All covers are the same as they were in CT_{OG}. Because of this $L(D, CT_{LE})$ will be the same for any partitioning, so we can ignore it entirely when comparing compression between solutions.

Every partition p_i gets its own instantiation code table CT_I, which contains every instantiation from $CT_I \in CT_{OG}$. Usage counts of instantiations depend on the instantiations used in the covers $cover_{CT_{OG}}(t)$ for all transactions $t \in p_i$. To ensure transactions can be encoded by code tables from other partitions all instantiations are Laplace corrected, even if an entire subtable in LESS-PL or LESS-II has 0-usage. Transactions are reassigned to the partition which gives the shortest encoded length. At some point transactions will no longer swap partitions, the partitioning and code table are returned.

The final step is to find the partitioning with the best compression. This is not as straightforward as one would expect. Due to the Laplace corrections, the more homogeneous the partition the bigger $L(CT_I|\mathcal{D})$, as there will be more and more low- to zero-usage instantiations which get very long codes. This should not necessarily be a problem, the shorter code instantiations should make up for it in $L(\mathcal{D} \mid CT_I)$. Still, almost always the model with $k = 1$ turned out to have the best compression. This implicates there is still some redundancy in the compression calculation, which gets amplified in models of higher k.

A source of redundancy seems to be the that we compress the bit-representation of CT_I separately for every partition. Because CT_I is supposed to contain all instantiations every code table should contain the exact same instantiations. I've decided instead to count compression $bit(i)$ of the instantiations only once per partitioning. A downside is that this will be a constant amount of bits required, depending on the attribute sets in CT_{LE}, regardless of k. However I expect that this advantage for partitionings of large k is cancelled out by the amount of codes introducing an extra CT_I comes with.

Encoding the Partition Assignment

The methods suggested by van Leeuwen et al. [19] do not mention taking into account the partition assignments in their compression rates, I consider this a mistake. For encoding a transaction it is not strictly necessary, one can simply try all code tables and use the shortest encoding, but for decoding it is a problem. Each code table forms its own prefix code, which if it were materialized, might use some of the same code words for different source words. As a result there might be multiple possible decodings depending on which code table one uses. To ensure that the transaction encoding remains uniquely decodable we should always first specify the code table with which to decode the subsequent code words. The codes specifying which code table to use obviously will become longer if there are more code tables to choose from. At the very least we should add $\log(k)$ to the encoded length of each transaction. The MDL principle would suggest we instead take into account the size of partitions as well, increasing

compressed size by $-|p|\log(|p|/|\mathcal{D}|)$.

5.4 Conclusions

Is it possible to translate the Data-Driven and Model-Driven methods by van Leeuwen et al. [19] to the context of LESS code tables?

A direct translation of the approaches by van Leeuwen et al. [19] is not possible. Despite it being supposed to work with any MDL method the Data-Driven approach often does not lead to a stable partitioning, preventing it from terminating. In addition to this the compression rates are dominated by the attribute set code tables CT_{LE}, which have the undesirable property of changing drastically for small changes in partitions. It is possible to define something similar to the Model-Driven approach, however van Leeuwen et al. [19] leave some important parts of their method undescribed. Such an approach would also likely fall victim to similar problems as the Data-Driven variant, with CT_{LE} dominating compression. Instead I have defined a partitioning algorithm which takes ideas from both approaches, and eliminates the influence of CT_{LE} by fixing it over all partitionings. In short, it is possible to translate the ideas from van Leeuwen et al. [19] to LESS code tables, but less directly than expected.

6 Experiments

This section contains various experimental results to evaluate the partitioning algorithm defined in Section 5.3 in order to answer the last of the three subquestions: **How does partitioning with LESS code tables compare to partitioning with KRIMP code tables**

In the first subsection I will give a summary of the datasets used in the experiments. In the second subsection I compare the results from the three variations of LESS with those from van Leeuwen et al. [19]. In addition to this comparison I will give a visualization of the results for one dataset to gain more insight into the solutions of the LESS partitioning algorithm. I also ran similar experiments to van Leeuwen et al. [19] with a different quality measure to get a more complete view of the quality of my solutions.

6.1 Datasets

For the experiments in Section 6.2.1 I use various UCI datasets [3]. For fair experimentation the class labels are separated from the data before building code tables. In Section 6.2.2 I use the European Mammals dataset[9] [12] to allow for a visual inspection of the output. This dataset contains the presence records of 121 mammals in geographical areas of 50×50 km in Europe.

Table 5 contains basic statistics for the datasets used in the experiments[10]. The fourth column contains the purity of the dataset, which van Leeuwen et al. [19] used as a quality measure for their results. The purity of a partitioning is the percentage of transactions in $\mathcal{D}$ belonging to the majority class of their assigned partition. The baseline purity is therefore the amount of transactions belonging to the majority class of the entire dataset.

The right side of Table 5 contains the settings I used for mining low entropy sets. Attributes with extremely skewed probability distributions were excluded from candidate mining based on parameter σ to avoid early pattern explosion. If an attribute value had a probability above σ it was excluded from candidate mining. These attributes are still part of the code tables as singletons, so that the compression remains lossless. These parameters were manually determined in order to get a manageable amount of candidates.

[9] available upon request from the European Mammal Foundation at https://www.european-mammals.org

[10] Pageblocks and nursery values differ from van Leeuwen et al. [19] I assume they made some errors while typing

Dataset	Basic statistics				Candidate Mining		
	$\|\mathcal{D}\|$	$\|\mathcal{I}\|$	$\|C\|$	Purity	ϵ	σ	$\|\mathcal{F}\|$
Adult	48842	95	2	76.1	2.9	0.95	108133
Anneal	898	65	6	76.2	2.8	0.95	310777
Chess (kr-k)	28056	40	18	16.2	2.5	0.99	144622
Mammals	2183	121	-	-	3.3	0.9	591208
Mushroom	8124	117	2	51.8	2.8	0.9	220259
Nursery	12960	27	5	33.3	3.8	0.99	26114
PageBlocks	5473	39	5	89.8	2.0	0.9985	2091730

Table 5: Statistics on datasets used in experiments

6.2 Results

6.2.1 Comparison with van Leeuwen et al. 2009

In Table 6 I compare the results of my partitioning algorithm with those of the KRIMP-based algorithms by van Leeuwen et al. [19] based on the achieved purity. For each LESS-variant code tables are built beforehand to determine the CT_{LE} with which each partitioning run is performed.

To ensure a fair comparison I looked for k-partitions where k matched the results reported by van Leeuwen et al. [19]. It would be unfair to compare results on purity with different k, since a partitioning assigning each transaction its own partition, in other words $k = |\mathcal{D}|$, will have a purity of 100%, but is clearly overpartitioned.

Results of the LESS-variants are determined by running the algorithm 25 times to cancel out the random initialization. The result with the best compression is reported. In some cases the algorithm ends with less than k partitions, these results are marked with an asterisk to indicate that the comparison is not entirely fair, favoring the KRIMP-based partitioning.

Dataset	van Leeuwen et al.		Purity			
	Method	Optimal k	LESS	LESS-PL	LESS-II	KRIMP
Adult	*Data-D*	177	80.4*	81.3*	81.9*	**82.2**
	Model-D	2	76.1	76.1	76.1	76.1
Anneal	*Data-D*	2	76.2	76.2	76.2	76.2
	Model-D	19	76.2*	**82.2***	81.3*	80.8
Chess (kr-k)	*Data-D*	13	19.9	**20.6**	18.5	17.8
	Model-D	6	18.2	**19.4**	18.0	18.2
Mushroom	*Data-D*	20	88.9*	96.0*	**96.6***	75.6
	Model-D	12	93.9*	95.7	**95.8**	88.2
Nursery	*Data-D*	8	46.9	42.6	**47.5**	43.4
	Model-D	14	**62.1**	50.6	42.9	45.0
PageBlocks	*Data-D*	30	92.1*	92.3*	92.1*	**92.5**
	Model-D	6	91.0*	90.8	90.9	**91.5**

Table 6: Comparison of results between Model-Driven and Data-Driven methods for KRIMP [19] and the LESS-variants

Overall the level of purity achieved by the LESS-variants is comparable to the results from the KRIMP-based algorithms, with the exception of the mushroom dataset, where the LESS-variants achieve much higher purity. But Table 6 does not tell the whole story, as it shows only the output with the best compression. Considering all 25 runs the output from the LESS-variants is far from robust. Each run producing an often significantly different output both in terms of compression and purity, whereas the Data-Driven algorithm from van Leeuwen et al. [19] is supposedly very robust. In addition to the lack of robustness, the LESS-variants often select sub-optimal solutions in terms of purity. This is evidenced by Figure 3 showing there is no correlation between the purity and compression of a partitioning. The plots for the other datasets[11] all look similar, ranging from a Pearson correlation coefficient of 0.3 to -0.3.

[11] the ones not marked with asterisk

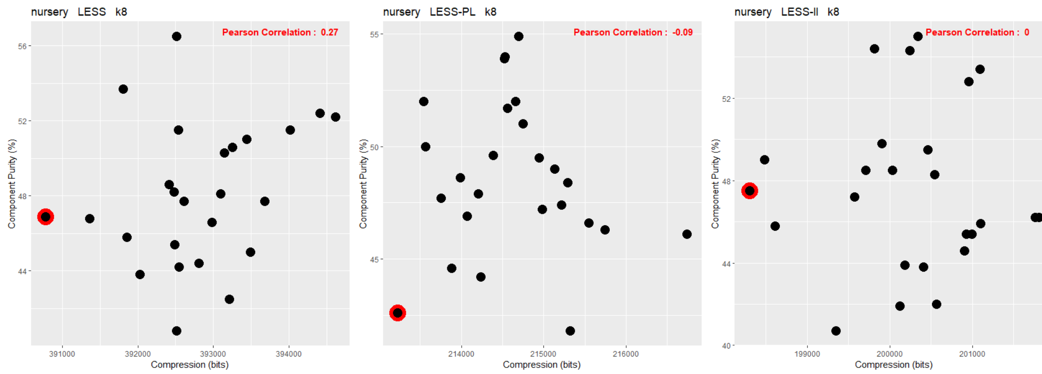

Figure 3: Scatterplots for Nursery dataset k = 8 : compression vs. purity

6.2.2 Visualized Results on European Mammals dataset

To visually inspect the quality of the results and gain insight into the lack of robustness I've plotted the results of the partitioning with LESS-variants on the European Mammals dataset onto a map of Europe. Figures 4, 5 and 6 show the results of 3 experiments per LESS-variant, with 25 runs per experiment for $k = 5$, plotting the best of the 25 runs.

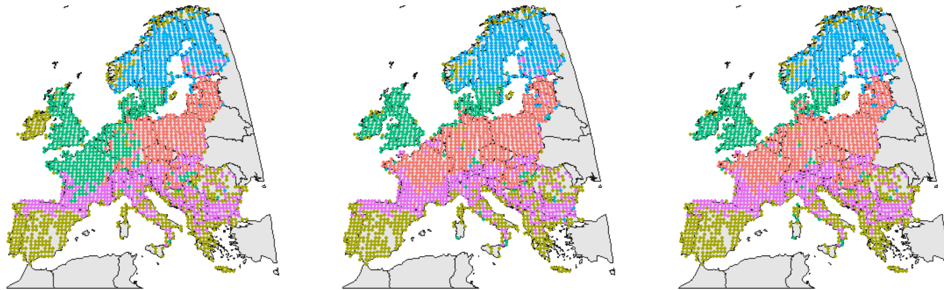

Figure 4: Partitionings made with LESS

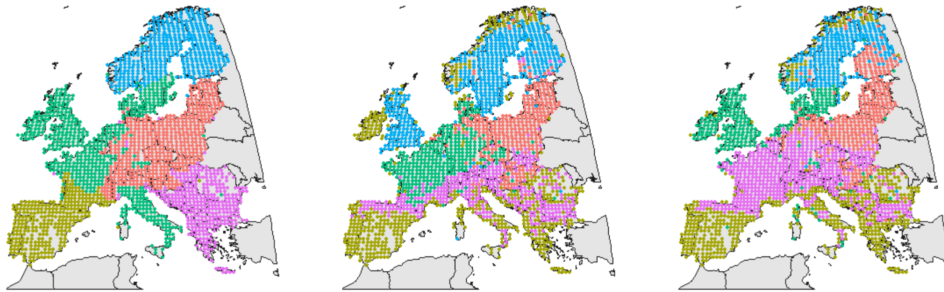

Figure 5: Partitionings made with LESS-PL

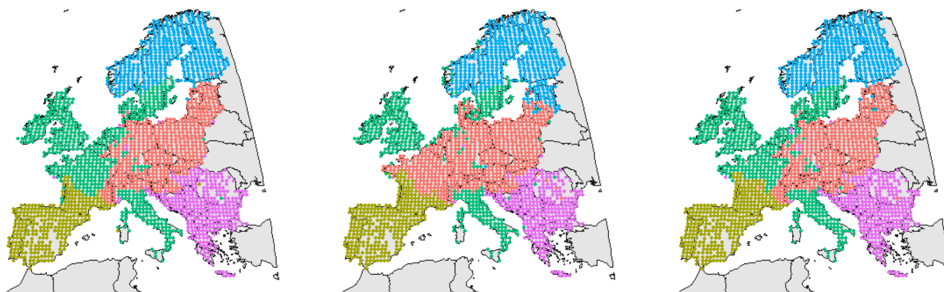

Figure 6: Partitionings made with LESS-II

I expected the results after 25 runs to at least look similar, but as Figures 4,5 and 6 show, each of the variants has at least one significantly different result. In terms of solution quality it is worth noting that LESS-II in Figure 6 manages to divide the data into quite spatially coherent clusters. This is quite promising considering that the algorithm isn't provided any geographical data. What does seem odd is that Italy and the British Isles are in the same cluster in all three results. Perhaps this is also caused by the lack of correlation between compression and our solution quality measure, as more intuitively correct groupings were part of the 25 results.

6.2.3 Entropy of Class Distribution

I decided to run more tests on another quality measure and to see if it has a better correlation with the compressed size of the solutions. Because purity only looks at the majority classes of partitions it gives limited insight into how well the data gets separated. If class labels aren't distributed evenly it is likely that the majority class for the entire dataset will be the majority class in multiple partitions, the distribution of the remaining classes in those partitions will be ignored in the purity score. Therefore it might be more interesting to look at

the entropy of the class distribution within the partitions, weighted by the size of the partitions. I calculate this weighted entropy as follows:

$$H_w(Q) = \sum_{p_i \in Q} \frac{|p|}{|\mathcal{D}|} H(C_p)$$

Where Q is a partitioning and C_{p_i} is the class distribution within a partition p_i.

I ran the experiments on the UCI datasets again, this time with $k = |C|$ as some of the experiments in Table 6 showed the algorithm does not always reach the k chosen earlier. The amount of runs are also increased to 100 to make sure we have enough data points for reliable analysis of the correlation.

Dataset	$H_w(Q)$			Pearson's r		
	LESS	LESS-PL	LESS-II	LESS	LESS-PL	LESS-II
Adult	0.79	0.79	**0.77**	-0.29	-0.25	**0.32**
Anneal	0.96	0.85	**0.71**	-0.08	**0.33**	0.32
Chess (kr-k)	3.20	**3.19**	3.30	0.40	**0.42**	0.19
Mushroom	0.98	**0.37**	0.48	0.30	**0.86**	0.18
Nursery	1.40	1.38	**1.34**	**0.48**	0.27	0.18
PageBlocks	0.48	**0.47**	0.48	-0.80	**-0.67**	-0.72

Table 7: Left: Entropy of class distribution for partitioning with best compression, Right: Pearson correlation coefficient between compression and weighted class entropy

Table 7 shows that the results vary per dataset and LESS-variant, though it shows there is at least some correlation between class entropy and compression for most datasets. The results also show that the LESS-PL and LESS-II models are mostly better at producing solutions with low class entropy than the original LESS code tables.

6.3 Conclusions

How does partitioning with LESS code tables compare to partitioning with KRIMP code tables?

It is difficult to give a decisive answer to this question. The resulting partitionings look acceptable, and in terms of purity the LESS algorithm slightly outperforms the KRIMP methods, suggesting it is indeed a more powerful approach. However the KRIMP methods are supposedly much more robust, outputting roughly the same results for multiple runs. The lack of robustness with LESS could be remedied by performing even more runs per experiment, but it is uncertain whether this will necessary lead to better results. The problem is that compression seems to have little correlation with the selected quality

measures. Perhaps the data contains more interesting groupings than the class labels provide, but I would still expect at least a moderate correlation with the tried quality measures. Combining this with the lack of robustness makes the algorithm unreliable to say the least. Table 7 suggests that adapting the original LESS code tables was justified, as both variants produce models with better distinctive capabilities. All in all I think using low entropy attribute sets for the partitioning task has potential, but the approach needs to be made more robust before it can be considered useful.

7 Conclusion

7.1 Summary

In order to answer the research question:
Can we create a generally applicable entropy-based partitioning method using LESS code tables? My research was split into three parts.

In Section 4 I tried to improve LESS through runtime candidate generation, similar to how SLIM [17] gave a better algorithm for finding KRIMP code tables. The definition of the instantiation code tables from LESS was fundamentally problematic for any bottom up runtime candidate generation approach. I gave two variants of LESS to avoid this problem. Despite this it was not possible to improve LESS through runtime candidate generation. The cover function provided too little structure, making it impossible to efficiently make good estimates for compression gain.

In Section 5 I attempted to translate the partitioning approaches from van Leeuwen et al. [19] to LESS code tables. Surprisingly the Data-Driven approach, which was expected to work with any MDL approach, did not work with LESS. The algorithm would often not terminate, and exhibited counter-intuitive behavior. I also considered how to convert the Model-Driven variant to the LESS setting, however decided against this approach as it would likely exhibit the same problems as the Data-Driven variant. Instead in Section 5.3 I gave a partitioning algorithm which incorporated both approaches, while avoiding most of their downsides.

In Section 6 my partitioning algorithm was compared to the Data-Driven and Model-Driven approaches for KRIMP. The algorithm produces acceptable partitions, both visibly and in terms of component purity. The final output of my algorithm was generally comparable with the KRIMP-based partitioning, if not better. A major problem however was inconsistency. Separate runs produce significantly different results, and good compression rates did not seem to correlate much to what I would consider higher quality solutions. The models of my LESS-variants slightly outperform the original LESS code tables in distinctive capabilities, giving justification for the adaptations.

Because of the inconsistency problems of my partitioning algorithm I must conclude that neither LESS code tables as defined by Heikinheimo et al. [9] or my two variants allow for a "generally applicable partitioning method". Though the approach does show some promise.

7.2 Discussion and Future Work

The results from Section 6.2 have shown that despite decent results the partitioning algorithm for LESS is very inconsistent between runs, variants and

datasets. I believe this might be due to some underlying problems within the LESS algorithm. I suspect that LESS finds overfitted code tables[12]. This would affect LESS-II the most, because it uses more and smaller code tables for the instantiations. To verify this I think it would be interesting to test LESS, for all three variants, on a more clearly defined task such as classification.

Another reason for my suspicions is the fact that LESS is not compatible with the Data-Driven partitioning [19] despite the seemingly valid claims that it should work with any MDL method. I believe the problem is caused by the likelihood cover function. This cover function is aimed at optimizing CT_I while ignoring CT_{LE}, but CT_{LE} tends to be the major contributor to the compressed size of databases with LESS. The cover function also prohibited us from making good compression estimates for a faster SLIM-like algorithm. It could be worth exploring different cover functions even if the results would be less interesting from a pattern mining perspective as noted at the end of Section 4.

If LESS proves to have problems on other tasks where KRIMP does not then perhaps it is wise to take a simpler approach to compression, as LESS might be overcomplicating the problem. A naïve compression of a binary database would require one bit per cell of the database, and perhaps some small amount of bits to model that each column is linked to an attribute. Models produced by LESS always require many more bits to encode than this naïve encoding[13]. This is because LESS requires that for each cell we add a code for the attribute-id, creating the need for CT_{LE}, and allowing for overlapping attribute sets to be used in covers. However considering that an attribute set can fit any transaction, maybe using overlapping sets is redundant in and of itself. A more pure compression approach, although less interesting from a pattern mining perspective, would be to work with solutions of non-overlapping attribute sets, each assigned an instantiation code table. This should be a more structured search space, with code lengths directly related to the Shannon entropy of the attribute sets, likely allowing for fast heuristics and Data-Driven partitioning.

[12]While MDL should prevent overfitting, maximum likelihood estimation is known to have problems with overfitting, perhaps the likelihood cover function also causes overfitting

[13]In Heikinheimo et al. [9] their final solutions require 3 to 7 times as many bits as this naïve encoding

References

[1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. "Mining association rules between sets of items in large databases". In: *Acm sigmod record*. Vol. 22. 2. ACM. 1993, pp. 207–216.

[2] Rakesh Agrawal, Ramakrishnan Srikant, et al. "Fast algorithms for mining association rules". In: *Proc. 20th int. conf. very large data bases, VLDB*. Vol. 1215. 1994, pp. 487–499.

[3] F. Coenen. *The LUCS-KDD Discretised/normalised ARM and CARM Data Library*. `http://www.csc.liv.ac.uk/~frans/KDD/Software/LUCS_KDD_DN/`. Department of Computer Science, The University of Liverpool, UK, 2003.

[4] Thomas M Cover and Joy A Thomas. *Elements of Information Theory*. John Wiley & Sons, 2012.

[5] A Murat Eren et al. "Minimum entropy decomposition: unsupervised oligotyping for sensitive partitioning of high-throughput marker gene sequences". In: *The ISME journal* 9.4 (2015), pp. 968–979.

[6] Peter Grünwald. "A tutorial introduction to the minimum description length principle". In: *Advances in minimum description length: Theory and applications* (2005), pp. 3–81.

[7] Jiawei Han, Jian Pei, and Yiwen Yin. "Mining frequent patterns without candidate generation". In: *ACM sigmod record*. Vol. 29. 2. ACM. 2000, pp. 1–12.

[8] Hannes Heikinheimo et al. "Finding low-entropy sets and trees from binary data". In: *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2007, pp. 350–359.

[9] Hannes Heikinheimo et al. "Low-entropy set selection". In: *Proceedings of the 2009 SIAM International Conference on Data Mining*. SIAM. 2009, pp. 569–580.

[10] David A Huffman. "A method for the construction of minimum-redundancy codes". In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[11] James MacQueen et al. "Some methods for classification and analysis of multivariate observations". In: *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. Vol. 1. 14. Oakland, CA, USA. 1967, pp. 281–297.

[12] Anthony J Mitchell-Jones et al. *The atlas of European mammals*. Vol. 3. Academic Press London, 1999.

[13] Frédéric Pennerath. "An Efficient Algorithm for Computing Entropic Measures of Feature Subsets". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2018, pp. 483–499.

[14] Jorma Rissanen. "Modeling by shortest data description". In: *Automatica* 14.5 (1978), pp. 465–471.

[15] Claude Elwood Shannon. "A mathematical theory of communication". In: *Bell system technical journal* 27.3 (1948), pp. 379–423.

[16] Arno Siebes, Jilles Vreeken, and Matthijs van Leeuwen. "Item sets that compress". In: *Proceedings of the 2006 SIAM International Conference on Data Mining*. SIAM. 2006, pp. 395–406.

[17] Koen Smets and Jilles Vreeken. "Slim: Directly mining descriptive patterns". In: *Proceedings of the 2012 SIAM International Conference on Data Mining*. SIAM. 2012, pp. 236–247.

[18] Koen Smets and Jilles Vreeken. "The odd one out: Identifying and characterising anomalies". In: *Proceedings of the 2011 SIAM international conference on data mining*. SIAM. 2011, pp. 804–815.

[19] Matthijs van Leeuwen, Jilles Vreeken, and Arno Siebes. "Identifying the components". In: *Data Mining and Knowledge Discovery* 19.2 (Oct. 2009), pp. 176–193. ISSN: 1573-756X. DOI: `10.1007/s10618-009-0137-2`. URL: `https://doi.org/10.1007/s10618-009-0137-2`.

[20] Jilles Vreeken, Matthijs Van Leeuwen, and Arno Siebes. "Characterising the difference". In: *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2007, pp. 765–774.

[21] Jilles Vreeken, Matthijs Van Leeuwen, and Arno Siebes. "Krimp: mining itemsets that compress". In: *Data Mining and Knowledge Discovery* 23.1 (2011), pp. 169–214.

[22] Mohammed J Zaki et al. *New Algorithms for Fast Discovery of Association Rules*. Tech. rep. Rochester, NY, USA, 1997.