Chapter 1

# MINING AND USING SETS OF PATTERNS THROUGH COMPRESSION

Matthijs van Leeuwen
*KU Leuven*
*Leuven, Belgium*
matthijs.vanleeuwen@cs.kuleuven.be

Jilles Vreeken
*Max-Planck Institute for Informatics and Saarland University*
*Saarbrücken, Germany*
jilles@mpi-inf.mpg.de

**Abstract** In this chapter we describe how to successfully apply the MDL principle to pattern mining. In particular, we discuss how pattern-based models can be designed and induced by means of compression, resulting in succinct and characteristic descriptions of the data.

As motivation, we argue that traditional pattern mining asks the wrong question: instead of asking for *all* patterns satisfying some interestingness measure, one should ask for a small, non-redundant, and interesting *set* of patterns—which allows us to avoid the pattern explosion. Firmly rooted in algorithmic information theory, the approach we discuss in this chapter states that the best set of patterns is that set that compresses the data best. We formalize this problem using the Minimum Description Length (MDL) principle, describe useful model classes, and briefly discuss algorithmic approaches to inducing good models from data. Last but not least, we describe how the obtained models—in addition to showing the key patterns of the data—can be used for a wide range of data mining tasks; hence showing that MDL selects *useful* patterns.

**Keywords:** Compression, MDL, pattern set mining, data summarization

# 1. Introduction

What is the ideal outcome of pattern mining? Which patterns would we really like to find? Obviously, this depends on the task at hand, and possibly even on the user. When we are exploring the data for new insights the ideal outcome will be different than when the goal is to build a good pattern-based classifier.

There are, however, a few important general observations to be made. For starters, we are not interested in patterns that describe noise—we only want patterns that identify important associations in the data. In pattern mining, the function that usually determines the importance of a pattern in this regard is called an *interestingness* measure.[1]

The traditional pattern mining question is to ask for *all* patterns in the data that satisfy some interestingness constraint. For example, all patterns that occur at least $n$ times, or, those that are so-and-so significant according to a certain statistical test. Intuitively this makes sense, yet in practice, this approach rarely leads to satisfactory results.

The primary cause is the pattern explosion. While strict constraints only result in few patterns, these are seldom informative: they are the most obvious patterns, and hence often long-since common knowledge. However, when we loosen the constraints—to discover novel associations—the pattern explosion occurs and we are flooded with results. More often than not orders of magnitude *more* patterns are returned than there are rows in the data. In fact, even for modest amounts of data billions of patterns are discovered for non-trivial constraints. Clearly, in such numbers these patterns are impossible to consider by hand, as well as very difficult to use in any other task—therewith effectively negating the goal of mining these patterns in the first place. Not quite the ideal result.

It does, however, provide us a second observation on the ideal outcome: we do not want to have too many results. In particular, we want to avoid *redundancy*: every pattern should be interesting or useful with regard to all of the other patterns in the result.

Simply put, traditional pattern mining has been asking the wrong question. In most situations, what we really want is a small, non-redundant, and as interesting possible group of patterns. As such, instead of asking for *all* patterns that satisfy some constraint, we should ask for the *set of patterns* that is optimal with regard to a *global* interestingness criterion. This means evaluating groups of patterns indirectly, i.e. by first constructing a model using these patterns, and then scoring the quality of that model. The main questions are then how to construct

[1] See Chapter 4 for a detailed overview of interestingness measures.

such a model, and which criterion should be used? Clearly, this depends on the task at hand.

In this chapter, we focus on exploratory data analysis. That is, our goal is to explore the data for new insights, to discover any local structure in the data—in other words, to discover patterns that describe the most important associations of the data, patterns that capture the distribution of the data. As such, we are looking for a set of patterns that models the data well. To this end, we need a criterion that measures both how well the patterns capture the distribution of the data, and—to avoid overfitting and redundancy—how complex the set of patterns is. Given a global interestingness criterion we can perform model selection and identify the best model. There are a few such criteria available, including Akaike's Information Criterion (AIC) [2] and the Bayesian Information Criterion (BIC) [53]. For pattern mining, the Minimum Description Length (MDL) principle [52] is the most natural choice. It provides a principled, statistically well-founded, yet practical approach for defining an objective function for *descriptive* models—which, as patterns *describe* part of the data, fits our setting rather well.

MDL allows us to unambiguously identify the best set of patterns as that set that provides the best lossless compression of the data. This provides us with a means to mine small sets of patterns that describe the distribution of the data very well: if the pattern set at hand would contain a pattern that describes noise, or that is redundant with regard to the rest, removing it from the set will *improve* compression. As such, the MDL optimal pattern set automatically balances the quality of fit of the data with the complexity of the model—without the user having to set any parameters, as all we have to do is minimize the encoding cost.

In this chapter we will give an overview of how MDL—or, compression—can be used towards mining informative pattern sets, as well as for how to use these patterns in a wide range of data mining tasks.

In a nutshell, we first discuss the necessary theoretical foundations in Section 2. In Section 3 we then use this theory to discuss constructing pattern-based models we can use with MDL. Section 4 covers the main approaches for mining good pattern sets, and in Section 5 we discuss a range of data mining tasks that pattern-based compression solves. We discuss open challenges in Section 6, and conclude in Section 7.

## 2. Foundations

Before we go into the specifics of MDL for pattern mining, we will have to discuss some foundational theory.

Above, we stated that intuitively our goal is to find patterns that describe *interesting structure* of the data—and want to avoid patterns that overfit, that describe noise. This raises the questions, what is significant structure, and where does structure stop and noise begin?

Statistics offers a wide range of tests to determine whether a result is significant, including via Bayesian approaches such as calculating confidence intervals, as well as frequentist approaches such as significance testing [17]. Loosely speaking, these require us to assume a background distribution or null hypothesis, and use different machinery to evaluate how likely the observed structure is under this assumption.

While a highly successful approach for confirming findings in science, in our exploratory setting this raises three serious problems. First and foremost, there are no off-the-shelf distributions for data and patterns that we can test against. Second, even if we could, by assuming a distribution we strongly influence which results will be deemed significant—a wrong choice will lead to meaningless results. Third, the choice for the significance or confidence thresholds is arbitrary, yet strongly influences the outcome. We want to avoid such far-reaching choices.

These problems were acknowledged by Ray Solomonoff, Andrey Kolmogorov, and Gregory Chaitin, whom independently invented and contributed to what is now known as algorithmic information theory [12]. In a nutshell, instead of using the probability under a distribution, in algorithmic information theory we consider the *algorithmic complexity* of the data. That is, to measure the amount of information the data contains by the amount of algorithmic 'effort' is required to generate the data using a universal Turing machine. There are different ways of formalizing such 'effort'. Here, we focus on Kolmogorov complexity.

## 2.1 Kolmogorov complexity

Kolmogorov complexity measures the information content of a string $s$; note that any database $\mathcal{D}$ can be serialized into a string. The Kolmogorov complexity of $s$, $K_{\mathcal{U}}(s)$, is defined as the length in bits of the shortest program $p$ for a Universal Turing machine $\mathcal{U}$ that generates $s$ and then halts. Formally, we have

$$K_{\mathcal{U}}(s) = \min_{p:\mathcal{U}(p)=x} |p| \quad .$$

Intuitively, program $p$ can be regarded as the ultimate compressor of $s$.

Let us analyze what this entails. First of all, it is easy to see that every string $s$ has at least one program that generates it: the program $p_0$ that simply prints $s$ verbatim. Further, we know that if the string is fully random, there will be no shorter program than $p_0$. This gives us an upper bound. In fact, this allows us to define what structure is, and what not. Namely, any (subset of) the data for which $K(s)$ is smaller than the length of $p_0$ exhibits structure—and the program $p$ is the shortest description of this structure.

Loosely speaking, the lower bound for $K$ is zero, which will only be approximated when the data $s$ is very simple to express algorithmically. Examples include a long series of one value, e.g., $000000000\ldots$, but also data that seems complex at first glance, such as the first $n$ digits of $\pi$, or a fractal, have in fact a very low Kolmogorov complexity—which matches the intuition that, while the result may be complex, the process for generating this data can indeed be relatively simple.

In fact, we can regard $p$ as two parts; the 'algorithm' that describes the compressible structure of $s$, and the 'input' to this algorithm that express the incompressible part of $s$. Separating these two components in a given dataset is exactly the goal of exploratory data analysis, and as such Kolmogorov Complexity institutes the ideal. Sadly, however, $K(s)$ is not computable. Apart from the fact that the space of possible programs is enormous, we face the problem that $p$ has to generate $s$ and then halt. By the halting problem we are unable to make that call.

This does not mean Kolmogorov complexity is useless. Quite the contrary, in fact. While beyond the scope of this chapter, it provides the theoretical foundations to many aspects of data analysis, statistics, data mining, and machine learning. We refer the interested reader to Li and Vitány [40] for a detailed discussion on these foundations.

Although Kolmogorov complexity itself is not computable, we can still put it to practice by approximating it. With $p$ we have the ultimate compressor, which can exploit *any* structure present in $s$. The incomputability of $K(s)$ stems from this infinite 'vocabulary', as we have to consider all possible programs. We can, however, constrain the family of programs we consider to a set for which we know they halt, by limiting this vocabulary to a fixed set of regularities. In other words, by considering lossless compression algorithms.

## 2.2 MDL

Minimum Description Length (MDL) [52, 20], like its close cousin Minimum Message Length (MML) [69], is in this sense a practical version of Kolmogorov Complexity [40]. All three embrace the slogan *Induction by Compression*, but the details on how to compress vary. For MDL, this principle can be roughly described as follows.

Given a set of models[2] $\mathcal{M}$, the best model $M \in \mathcal{M}$ is the one that minimizes

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

in which

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

This is called two-part MDL, or *crude* MDL—as opposed to *refined* MDL, where model and data are encoded together [20]. We consider two-part MDL because we are specifically interested in the compressor: the set of patterns that yields the best compression. Further, although refined MDL has stronger theoretical foundations, it cannot be computed except for some special cases.

**2.2.1 MDL and Kolmogorov.** The MDL-optimal model $M$ has many of the properties of the Kolmogorov optimal program $p$. In fact, two-part MDL and Kolmogorov complexity have a one-to-one connection [1, 20]. Loosely speaking, the two terms respectively express the structure in the data, and the deviation from that structure: $L(M)$ corresponds to the 'algorithm' part of $p$, which generates the structure. $L(\mathcal{D} \mid M)$, on the other hand, does not contain any structure—as otherwise there would be a better $M$—and can be seen as the 'parameter' part of $p$. One important difference is that $L(\mathcal{D}, M)$ happily ignores the length of the decompression algorithm—which would be needed to reconstruct the data given the compressed representation of the model and data. The reason is simple: its length is constant, and hence does not influence the selection of the best model.

**2.2.2 MDL and Probabilities.** Any MDL-based approach encodes both the data and the models, for which *codes* are required. It is well-known that there is a close relation between probability distributions and optimal codes. That is, Shannon's source coding theorem states that the optimal code length for a given symbol in a string is equal to the $-\log$ of the probability of observing it in the string [12].

As such, an alternate interpretation of MDL is to interpret $L(\mathcal{D} \mid M)$ as the (negative) log-likelihood of the data under the model, $-\log \Pr(\mathcal{D} \mid M)$, and to regard $L(M)$, as the negative log-likelihood of the model, $-\log \Pr(M)$, or, a regularization function. Hence, looking for the model

[2] MDL-theorists tend to talk about *hypotheses* in this context

that gives the best compression is similar to looking for the maximum likelihood model under a budget. As such it has a similar shape to Akaike's Information Criterion (AIC) [2] and the Bayesian Information Criterion (BIC) [53]. This of course assumes that there is a distribution for models, as well as that we have a generative model for data that can be parameterized by $M$. This is often not the case.

In MDL, however, we are concerned with *descriptive* models—not necessarily generative ones. As such, different from Bayesian learning, in both Kolmogorov complexity and MDL we evaluate only the data and explicit model at hand—we do not 'average' over all models, so to speak, and hence do not need access to a generative model. Moreover, MDL is different in that it requires a complete, lossless encoding of both the model and the data while BIC and AIC penalize models based only on the number of parameters.

In practice, while (refined) MDL and BIC are asymptotically the same, the two may differ (strongly) on finite data samples. Typically, MDL is a bit more conservative. For a detailed discussion on the differences between BIC and MDL we refer to Grünwald [20].

**Using MDL in Practice.** To use MDL in practice, one has to define the model class $\mathcal{M}$, how a single model $M \in \mathcal{M}$ describes a database, and how all of this is encoded in bits. That is, we have to define a compression scheme. In addition, we need an algorithm to mine—or approximate—the optimal model.

A key advantage of MDL is that it removes the need for user-defined parameters: the best model minimizes the total encoded size. Unfortunately, there are also disadvantages: 1) contrary to Kolmogorov Complexity, a model class needs to be defined in advance, and 2) finding the optimal model is often practically infeasible. Consequently, important design choices have to be made, and this is one of the challenges of the compression-based approach to exploratory data mining.

A standard question regarding the use of MDL concerns the requirement of a lossless encoding: if the goal is to find very short descriptions, why not use a lossy encoding? The answer is two-fold.

First, and foremost, lossless encoding ensures fair comparison between models: we know that every model is evaluated based on how well it describes the complete dataset. With lossy compression, this is not the case: two models could have the same $L(\mathcal{D}, M)$—one describing only a small part of the data in high detail, and the other describing all the data in low detail—and unlike for lossless compression, we would have no (principled) way of choosing which one is best.

Second of all, we should point out that compression is *not* the goal, but only a *means* to select the best model. By MDL, the best model provides the shortest description out of all models in the model class, and it is that model that we are interested in—in the end, the length of the description is often not of much interest. When compression is the goal, a general purpose compressor such as ZIP often provides much better compression, as it can exploit many types of statistical dependencies.

In a similar vein, is it also important to note that in MDL we are *not* concerned with materialized codes, but only interested in their *lengths*—again, as model *selection* is the goal. Although a complete overview of all useful codes—for which we can compute the optimal lengths in practice—is beyond the scope of this chapter, we will discuss a few instances in the next chapter, where we will discuss how to use MDL for pattern mining. Before we do so, however, let us quickly go into the general applicability of MDL in data mining.

### 2.3 MDL in data mining

Faloutsos and Megalooikonomou [15] argue that Kolomogorov Complexity and Minimum Description Length [52, 20] provide a powerful and well-founded approach to data mining. There exist many examples where MDL has been successfully employed in data mining, including, for example, for classification [50, 38], clustering [31, 6, 39], discretization [16, 30], defining parameter-free distance measures [28, 29, 11, 66], feature selection [48], imputation [65], mining temporally surprising patterns [9], detecting change points in data streams [37], model order selection in matrix factorization [46], outlier detection [58, 3], summarizing categorical data [43], transfer learning [54], discovering communities in matrices [8, 47, 63] and evolving graphs [60], finding sources of infection in large graphs [49], and for making sense of selected nodes in graphs [4].

We will discuss a few of these instances in Section 5, but first cover how to define an MDL score for a pattern-based model.

## 3. Compression-based pattern models

In this section we introduce how to use the above foundations for mining small sets of patterns that capture the data distribution well. We will give both the high level picture and illustrate it with concrete instances and examples. Before we go into details, let us briefly describe the basic ingredients that are required for any pattern-based model.

We assume a dataset $\mathcal{D}$ to be a bag of elements $t$ of some data type—which we, for simplicity, will refer to as *tuples.* In the context of frequent itemset mining, each $t$ is a transaction over a set of items $\mathcal{I}$, i.e., $t \subseteq \mathcal{I}$.

Similarly, we can consider sequences, trees, graphs, time series, etc. Let us write $\mathcal{T}$ to denote the universe of possible tuples for a given data type. Clearly, all tuples in $\mathcal{D}$ are elements from $\mathcal{T}$.

Given a dataset, on of the most important choices is the *pattern language* $\mathcal{X}$. A pattern language is the set of all possible patterns that we can discover for a given data type. In principle, a pattern can be any structure that describes the distribution of (a subset of) the data. Given the topic of the book, we focus on frequent patterns; e.g., when we consider itemsets, $\mathcal{X}$ can be the set of all possible itemsets, while for structured data, $\mathcal{X}$ can consist of sequential patterns, subgraphs, etc.

Clearly, the choice of $\mathcal{X}$ is highly important, as it determines the type of structure that we will be able to discover in the data. Another way of thinking about $\mathcal{X}$ is that it defines the 'vocabulary' of the compressor. If one chooses a pattern language that is highly specific, it may be impossible to find relevant structure of that type in the data. On the other hand, if a very rich, i.e., more complex pattern language is chosen, the encoding and search for the model can become rather complicated.

## 3.1 Pattern models for MDL

Given a class of data and a pattern language, we can start to construct a pattern-based model. Note that by defining a model class, we essentially fix the set of possible models $\mathcal{M}$, the possible descriptions, for a given dataset $\mathcal{D}$. Given this space of possible descriptions, we can employ the MDL principle to select the best model $M \in \mathcal{M}$ for $\mathcal{D}$ simply by choosing the model that minimizes the total compressed size. In order to do so, however, we need to be able to compute $L(\mathcal{D}, M)$, the encoded length of the model and the data given the model.

We start with the latter, i.e., we first formally define how to compute $L(\mathcal{D} \mid M)$, the encoded length in bits of the data given the model. Generally speaking, there are many different ways to describe the same data using one model. However, by the MDL principle, our encoding should be such that we use the minimal amount of bits to do so. This helps us to make principled choices when defining the encoding scheme. Some of these may impose additional constraints and requirements on the design of the compressor, as well as determine how the score can be used. This is particularly important in light of subsequently using the pattern-based models in data mining tasks other than summarization. Here we describe three important properties that a compressor may have.

**Dataset-level compression** At the highest level we need to be able to compare the encoded sizes of different databases. The most trivial way to do so is by comparing the total encoded size, $L(\mathcal{D}, M)$, where we induce the MDL-optimal model $M$ for each $\mathcal{D}$.

This property alone allows us to use compression as a 'black box': without paying any attention to the contents of the models or how datasets are compressed, the MDL principle can be used to select appropriate models for a given dataset. In fact, this property does not even require datasets to consist of individual tuples that can be distinguished, nor does it require models to consist of patterns.

Moreover, it allows us to *use* compression for data mining tasks, such as for computing data dissimilarity. Note that this property generally holds for any generic compressor, and therefore compression algorithms like those in ZIP, GZIP, BZIP, etc, can also be used for such tasks. As a concrete example, the family of Normalized Compression Distance measures [41] rely on this.

As a slight variant, we can also fix the model $M$ and see how well it compresses another dataset. That is, we require that $L(\mathcal{D} \mid M)$ is explicitly calculable for any $\mathcal{D}$ of the specified data universe $\mathcal{T}$ and any $M \in \mathcal{M}$. This allows us to calculate how well a dataset matches the distribution of another dataset. See also Section 5.

**Tuple-level compression** In addition, a rather useful property is when each tuple $t \in \mathcal{D}$ can be compressed individually, independent of all other tuples. That is, $L(t \mid M)$ can be computed for a given $M$. This also implies we can simply calculate $L(\mathcal{D} \mid M)$ as

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

This property simplifies many aspects related to the induction and usage of the models. For example, as a consequence, calculating the encoded size can now be trivially parallelized. More important, though, is that common data mining tasks such as classification and clustering are now straightforward. More on this later.

**Pattern-level inspection** The third and final property that we discuss here is that of *sub-tuple*, or, *pattern-level* inspection. That is, beyond computing $L(t \mid M)$ we also want to be able to inspect *how* a given tuple is encoded: what structure, in the form of a set of patterns, is used to compress it?

With this property, it becomes possible to provide explanations for certain outcomes (e.g., explain why is a certain tuple compressed better by one model than by another), but also to exploit this

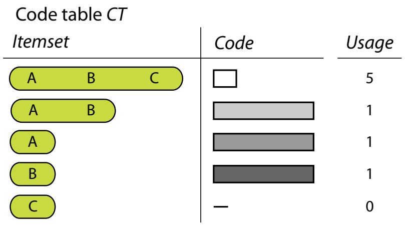

*Figure 1.1.* Example code table. The widths of the codes represent their lengths. $\mathcal{I} = \{A, B, C\}$. Note that the *usage* column is not part of the code table, but shown here as illustration: for optimal compression, codes should be shorter the more often they are used.

information to improve the model (e.g., patterns that often occur together in a tuple should probably be combined). Effectively, it is this property that makes pattern-based solutions so powerful, as it ensures that in addition to decisions, we can offer *explanations*.

## 3.2 Code tables

The conceptually most simple, as well as most commonly used pattern-based model for MDL are so-called *code tables* (see e.g., [57, 23, 56, 68, 59, 64]). Informally, a code table is a dictionary, a translation table between patterns and codes. Each entry in the left column contains a pattern and corresponds to exactly one code word in the right column. Such a code table can be used to compress the data by replacing occurrences of patterns with their corresponding codes, and vice versa to decode an encoded dataset and reconstruct the original the data. Using the MDL principle, the problem can then be formulated as finding that code table that gives the best compression.

Next, we describe both the general approach, as well as cover a specific instance for transaction data. First, we formally define a code table.

Definition 1.1 *Let $\mathcal{X}$ be a set of patterns and $\mathcal{C}$ a set of code words. A code table $CT$ over $\mathcal{X}$ and $\mathcal{C}$ is a two-column table such that:*

*1 The first column contains patterns, that is, elements from $\mathcal{X}$.*

*2 The second column contains elements from $\mathcal{C}$, such that each element of $\mathcal{C}$ occurs at most once.*

*We write code$(X \mid CT)$ for the code corresponding to a pattern $X \in CT$. Further, we say $PS$ for $\{X \in CT\}$, the* pattern set *of $CT$.*

Example 1.2 *Throughout we will use* Krimp *[57, 68] as running example. It was the first pattern set mining method using code tables and MDL, and considers itemset data. In all examples, a dataset $\mathcal{D}$ is a bag of transactions over a set of items $\mathcal{I}$, i.e., for each $t \in \mathcal{D}$ we have $t \subseteq \mathcal{I}$. Patterns are also itemsets and the pattern language is the set of all possible itemsets, i.e., $\mathcal{X} = 2^{\mathcal{I}} = \{X \subseteq \mathcal{I}\}$.*

*Figure 1.1 shows an example* Krimp *code table of five patterns. The left column lists the itemsets, the second column contains the codes. Each bar represents a code, its width represents the code length. (Note, these are obviously not real codes, but a simplified representation; for our purposes representing code lengths suffices.) The usage column is not part of the code table, but only used to determine the code lengths.*

**3.2.1 Encoding the data.** Given a dataset $\mathcal{D}$ and a code table $CT$, we need to define how to encode $\mathcal{D}$ with $CT$. As already mentioned, encoding a dataset is done by replacing occurrences of patterns in the code table by their corresponding codes. To achieve lossless encoding of the data, we need to *cover* the complete dataset with patterns from pattern set $PS$. In practice, covering a dataset is usually done on a per-tuple basis, such that each tuple is covered by a subset of the patterns in the code table. Hence, a code table normally has all three properties discussed in the previous subsection: it allows for *dataset-level compression*, *tuple-level compression*, and *sub-tuple inspection*.

To encode a tuple $t$ from database $\mathcal{D}$ with code table $CT$, a cover function $cover(CT, t)$ is required that identifies which elements of $CT$ are used to encode $t$. The parameters are a code table $CT$ and a tuple $t$, the result is a disjoint set of elements of $CT$ that cover $t$. Or, more formally, a cover function is defined as follows.

Definition 1.3 *Let $\mathcal{D}$ be a database over a universe of possible tuples $\mathcal{T}$, $t$ a tuple drawn from $\mathcal{D}$, let $\mathcal{CT}$ be the set of all possible code tables over $\mathcal{X}$, and $CT$ a code table with $CT \in \mathcal{CT}$. Then, $cover : \mathcal{CT} \times \mathcal{T} \mapsto \mathcal{P}(\mathcal{X})$ is a* cover function *iff it returns a set of patterns such that*

1 *$cover(CT, t)$ is a subset of $PS$, the pattern set of $CT$, i.e.,*
  *$X \in cover(CT, t) \rightarrow X \in CT$*

2 *together all $X \in cover(CT, t)$ cover $t$ completely, i.e., $t$ can be fully reconstructed from $cover(CT, t)$*

*We say that $cover(CT, t)$ covers $t$.*

Observe that this cover function is very generic and allows many different instances. In general, finding a subset of a pattern set that covers

**Algorithm 1** The KrimpCover Algorithm

```
Require: Transaction $t \in \mathcal{D}$ and code table $CT$, both over $\mathcal{I}$.
Ensure: A cover of $t$ using non-overlapping elements of $CT$.
$S \leftarrow$ first element $X$ of $CT$ for which $X \subseteq t$
if $t \setminus S = \emptyset$ then
    $Res \leftarrow \{S\}$
else
    $Res \leftarrow \{S\} \cup$ KrimpCover($t \setminus S, CT$)
end if
return $Res$
```

a tuple can be a hard combinatorial problem. Depending on the data universe $\mathcal{T}$, pattern language $\mathcal{X}$ and requirements imposed by the task, it may therefore be beneficial to impose additional constraints to make the cover function fast and efficient to compute. Also, note that without any further requirements on code tables, it may be possible that a code table cannot cover any tuple. To remedy this, a common approach is to require that any 'valid' code table should contain at least all primitive patterns, i.e., singletons, required to cover any tuple from $\mathcal{T}$.

Example 1.4 *We continue the example of* Krimp *and present its cover function in Algorithm 1. To allow for fast and efficient covering of transactions,* Krimp *considers non-overlapping covers. Its mechanism is very simple: look for the first element in the code table that occurs in the tuple, add it to the cover and remove it from the tuple, and repeat this until the tuple is empty. Recalling that tuples and patterns are both itemsets, we have that a cover is a set of itemsets, s.t.*

$$\forall_{X,Y \in cover(t,CT)} X \cap Y = \emptyset,$$

*and*

$$\cup_{X \in cover(t,CT)} X = t.$$

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

*To ensure that each code table is 'valid', each $CT$ is required to contain at least all singleton itemsets from $\mathcal{I}$, i.e., $PS \supseteq \{\{i\} \mid i \in \mathcal{I}\}$. This way, any transaction $t \in \mathcal{P}(\mathcal{I})$ can always be covered by any $CT \in \mathcal{CT}$.*

*Figure 1.2 shows an example database consisting of 8 itemsets, of which 5 are identical. Also shown is the cover of this database with the example code table from Figure 1.1. In this example, each transaction*

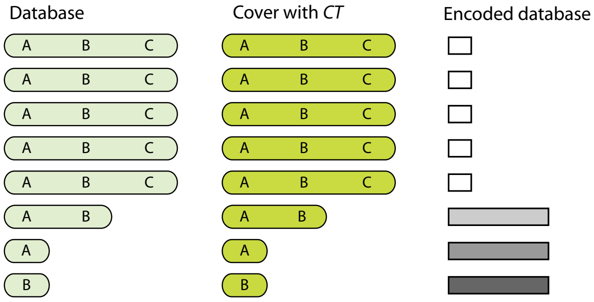

*Figure 1.2.* Example database, cover and encoded database obtained by using the code table shown in Figure 1.1. $\mathcal{I} = \{A, B, C\}$

*is covered by only a single itemset from the code table, resulting in very good compression. Obviously it is often not the case that complete transactions can be covered with a single itemset. For example, if itemset* $\{ABC\}$ *had not been in the code table, the first five transactions would have been covered by* $\{AB\}$ *and* $\{C\}$.

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

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

Note that to ensure that we can decode an encoded database uniquely we assume that $\mathcal{C}$ is a *prefix code*, in which no code is the prefix of another code [12].

Example 1.5 *Figure 1.2 shows how the cover of a database can be translated into an encoded database: replace each itemset in the cover by its associated code.*

**3.2.2 Computing encoded lengths.** Since MDL is concerned with the best compression, the codes in $CT$ should be chosen such that the most often used code has the shortest length. That is, we should use *optimal* prefix codes. As there exists a nice correspondence between code lengths and probability distributions (see, e.g., [40]), the optimal code lengths can be calculated through the Shannon entropy. In MDL we are only interested in measuring complexity, and not in materialized codes. As such we do not have to require round code lengths, nor do we have to operate an actual prefix coding scheme such as Shannon-Fano or Huffman encoding.

Theorem 1.6 *Let* $P$ *be a distribution on some finite set* $\mathcal{D}$, *there exists an optimal prefix code* $\mathcal{C}$ *on* $\mathcal{D}$ *such that the length of the code for* $t \in \mathcal{D}$, *denoted by $L(t)$ is given by*

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

*Moreover, this code is optimal in the sense that it gives the smallest expected code size for data sets drawn according to $P$. (For the proof, please refer to Theorem 5.4.1 in [12].)*

The optimality property means that we introduce no bias using this code length. The probability distribution induced by a cover function is, of course, given by the relative usage frequency of each of the patterns.

To determine this, we need to know how often a certain code is used. We define the *usage* count of a pattern $X \in CT$ as the number of tuples $t$ from $\mathcal{D}$ where $X$ occurs in its cover. Normalized, this frequency represents the probability that that code is used in the encoding of an arbitrary $t \in \mathcal{D}$. The optimal code length [40] then is $-\log$ of this probability, and a code table is optimal if all its codes have their optimal length. More formally, we have the following definition.

Definition 1.7 *Let $\mathcal{D}$ be a database drawn from a tuple universe $\mathcal{T}$, $\mathcal{C}$ a prefix code, cover a cover function, and $CT$ a code table over $\mathcal{X}$ and $\mathcal{C}$. The* usage *count of a pattern $X \in CT$ is defined as*

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

*This implies a* probability *distribution over the usage of patterns $X \in CT$ in the cover of $\mathcal{D}$ by $CT$, which is given by*

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

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

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

*A code table $CT$ is* code-optimal *for $\mathcal{D}$ iff all its codes, $\{code(X \mid CT) \mid X \in CT\}$, are optimal for $\mathcal{D}$.*

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

Example 1.8 *Figure 1.1 shows usage counts for all itemsets in the code table. For example, itemset $\{A, B, C\}$ is used 5 times in the cover of the database. These usage counts are used to compute optimal code lengths. For $X = \{A, B, C\}$:*

$$\begin{aligned} P(X \mid \mathcal{D}, CT) &= \frac{5}{8} \\ L(code(X \mid CT)) &= -\log(\frac{5}{8}) = 0.68 \end{aligned}$$

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

$$\begin{aligned} P(Y \mid \mathcal{D}, CT) &= \frac{1}{8} \\ L(code(Y \mid CT)) &= -\log(\frac{1}{8}) = 3.00 \end{aligned}$$

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

Now, for any database $\mathcal{D}$ and code table $CT$ over the same set of patterns $\mathcal{X}$ we can compute $L(\mathcal{D} \mid CT)$ according to the following lemma.

Lemma 1.9 *Let $\mathcal{D}$ be a database, $CT$ be a code table over $\mathcal{X}$ and code-optimal for $\mathcal{D}$,* cover *a cover function, and* usage *the usage function for* cover.

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

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

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

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

With Lemma 1.9, we can compute $L(\mathcal{D} \mid M)$, but we also need to know what $L(M)$ is, i.e., the encoded size of a code table.

Recall that a code table is a two-column table consisting of patterns and codes. As we know the size of each of the codes, the encoded size of the second column is easily determined: it is simply the sum of the lengths of the codes. The encoding of the first column, containing the patterns, depends on the pattern type; a lossless and succinct encoding should be chosen.

Definition 1.10 *Let $\mathcal{D}$ be a database, $CT$ a code table over $\mathcal{X}$ that is code-optimal for $\mathcal{D}$, and* encode *an encoding for elements of $\mathcal{X}$. The size of $CT$ in bits, denoted by $L(CT \mid \mathcal{D})$, is given by*

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

*Note that we do not take patterns with zero usage into account, because they are not used to code and do not get a finite code length.*

With these results we have the total size of the encoded database.

Definition 1.11 *Let $\mathcal{D}$ be a database with tuples drawn from $\mathcal{T}$, let $CT$ be a code table that is code-optimal for $\mathcal{D}$ and* cover *a cover function. The* total compressed size *of the encoded database and the code table, in bits, denoted by $L(\mathcal{D}, CT)$ is given by*

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

**3.2.3 The Problem.** The overall problem is now to find the set of patterns that best describe a database $\mathcal{D}$. Given a pattern set $PS$, a cover function and a database, a (code-optimal) code table $CT$ follows automatically. Therefore, each coding set has a corresponding code table and we can use this to formalize the problem.

Given a set of patterns $\mathcal{F} \subseteq \mathcal{X}$, the problem is to find a minimal subset of $\mathcal{F}$ which leads to a minimal encoded size $L(\mathcal{D}, CT)$. By requiring the smallest possible pattern set, we make sure it does not contain any unused patterns, i.e., $usage_{CT}(X) > 0$ for any pattern $X \in CT$.

More formally, in general terms, we define the problem as follows.

Problem 1 (Minimum Description Length Pattern Set) *Let $\mathcal{D}$ be a dataset of tuples drawn from $\mathcal{T}$, $\mathcal{F} \subseteq \mathcal{X}$ a candidate set, and* enc *an encoding for datasets over $\mathcal{T}$ and models over $\mathcal{X}$. Find the smallest pattern set $PS \subseteq \mathcal{F}$ such that for the corresponding model $M$ the total compressed size with encoding* enc*, $L_{enc}(\mathcal{D}, M)$, is minimal.*

Naively, one might say that the solution for this problem can be found by simply enumerating all possible pattern sets given a collection of patterns $\mathcal{X}$. As such, the search space is already huge: a pattern set contains an arbitrary subset of $\mathcal{X}$, excluding only the empty set. Hence, there are

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

possible pattern sets. To determine which pattern set minimizes the objective function, we have to know the optimal cover function. Even for a greedy strategy such as covering the data using a fixed order, this explodes to having to consider all possible orders of all possible pattern sets. To make matters worse, the score typically exhibits no (weak) (anti-)monotone structure that we can exploit. As such, we relax the problem and resort to heuristics to find good models instead of the optimum.

## 3.3 Instances of compression-based models

Code tables form a generic model class that can be used with virtually any pattern and data type, given a suitable encoding. Of course there are also other compression-based model classes, and we will now discuss instances of both types.

**3.3.1 Code table instances.** The best-known instance of code tables is the one used as running example in this chapter, i.e., KRIMP code tables over itemsets and often used in conjunction with the cover function presented in Algorithm 1. As we will see in the next section, there also exist more sophisticated algorithms for inducing code tables.

In practice, we find that KRIMP returns pattern sets in the order of hundreds to a few thousand of patterns [68], which have been shown to describe the distribution of the data very well. In the next section we will discuss some of the applications in which these pattern sets have been successfully put to use.

Akoglu et al. [3] proposed the COMPREX algorithm, which describes a categorical dataset by a *set* of KRIMP code tables—by partitioning the attributes into parts that correlate strongly, and inducing a KRIMP code table for each part directly from the data.

In frequent pattern mining, and hence KRIMP, we only regard associations between 1s of the data as potentially interesting. This is mostly a matter of keeping matters computational feasible—clearly there are cases where associations between occurrences and absences are rather interesting. LESS [24] is an algorithm that describes data not using frequent itemsets, but using low-entropy sets [23]. These are itemsets for which we see the distribution of its occurrences is strongly skewed. LESS code tables consist of low-entropy sets, and it uses these to identify areas of the data of where the attributes strongly interact. LESS code tables typically contain only tens to hundreds of low-entropy sets. Attribute clustering [43] provides even more succinct code tables, with the goal to provide good high-level summaries of categorical data, only up to tens of patterns are selected.

Code table instances for richer data include those for sequential patterns, i.e., serial episodes. Bathoorn et al. [5] gave a variant of KRIMP for sequential patterns without gaps, wheras the SQS [64] and GOKRIMP [35] algorithms provide fast algorithms for descriptions in terms of serial episodes where gaps are allowed. Like KRIMP, these algorithms find final selections in the order of hundreds of patterns.

Koopman and Siebes [32, 33] discussed the KRIMP framework in light of frequent patterns over multi-relational databases.

**3.3.2 Other model classes.** Like Less, Pack [62] considers binary data symmetrically. Its patterns are itemsets, but they are modeled in a decision tree instead of a code table. This way, probabilities can be calculated more straightforwardly and refined MDL can be used for the encoding. Mtv [44] also constructs a probabilistic model of the data, and aims to find that set of itemsets that best predicts the data. The framework allows both BIC and MDL to be used for model selection. Typically, between tens and hundred of itemsets are selected.

Stijl [63] describes data hierarchically in terms of dense and sparse tiles, rectangles in the data which contain surprisingly many/few 1s.

We also find compression-based models in the literature that employ lossy compression. While this contradicts MDL in principle, as long as the amount of 'lost' data is not too large, relatively fair comparisons between models can still be made.

Summarization [10] is such an approach, which identifies a group of itemsets such that each transaction is summarized by one itemset with as little loss of information as possible. Wang et al. [70] find summary sets, sets of itemsets such that each transaction is (partially) covered by the largest itemset that is frequent.

There are also model classes where the link to compression exists, but is hidden from plain sight. Tiling [18] should be mentioned: a tiling is the cover of the database by the smallest set of itemsets, and is related to Set Cover [27], Minimum Entropy Set Cover [22], and matrix factorization problems [42, 46].

## 4. Algorithmic approaches

So far we have discussed in detail the motivation, theoretical foundations, and models for compression-based pattern mining. Given the previous, the natural follow-up question is: *given a dataset, how can we find that model that minimizes the total compressed size?*

In this section we aim to give a brief overview of the main algorithmic strategies for inducing good code tables from data. There are two main approaches we need to discuss: candidate filtering and direct mining.

In our concise discussion on the complexity of the *Minimum Description Length Code Table* problem, we already mentioned that the search space will generally be too large to consider exhaustively. Hence, as is common with MDL-based approaches, the common solution is to resort to heuristic search strategies. This obviously implies that we usually cannot guarantee to find the best possible model, and experimental evaluation will have to reveal how useful induced models are.

In this section, we will outline common techniques. For a more in-depth discussion of the individual algorithms, we refer to the original papers; algorithmic aspects are not the main focus of this chapter.

## 4.1 Candidate set filtering

The definition of Problem 1 already hints at the most often used approach: candidate filtering. While the set of candidates $\mathcal{F}$ could consist of all possible patterns $\mathcal{X}$, it can also be a subset defined by some additional constraints. Typically, $\mathcal{F}$ is generated in advance and given as argument to the algorithm used for model induction.

For example, when inducing itemset-based models, it is common practice to use closed itemsets with a given minimum support threshold as candidate set. A large advantage of using smaller candidate sets, i.e., keeping $|\mathcal{F}|$ small, is that model induction can be done relatively quickly.

Given a dataset $\mathcal{D}$ and candidate set $\mathcal{F}$, a candidate set filtering method returns a model $M$ corresponding to a pattern set $PS \subset \mathcal{F}$ for which $L(\mathcal{D}, M)$ is 'small'. (Note that we cannot claim that the compressed size is minimal due to the heuristic nature of filtering methods.)

**4.1.1 Single-pass filtering.** The simplest filtering approach uses the following greedy search strategy:

1 Start with an 'empty' model $M$.

2 Add patterns $F \in \mathcal{F}$ to $M$ one by one. If the addition leads to better compression, keep it, otherwise, permanently discard $F$.

Although the basic principle of this approach is very simple, note that there are important details that need to be worked out depending on the specific model and encoding. For example, it is often impossible to start with a model that is truly empty: if a model does not contain any patterns at all, it may be impossible to encode the data at hand and hence there is no compressed size to start from. Also, adding a pattern to a model is not always straightforward: how and where in the model should it be added? Depending on design choices, there may be many possibilities and if these need all to be tested this can become a computational burden. Finally, in what order should we consider the candidates in $\mathcal{F}$? Since single-pass filtering considers every candidate only once, this choice will have a large impact on the final result.

EXAMPLE 1.12 KRIMP *employs single-pass filtering with several heuristic choices to ensure that it can induce good code tables from relatively large datasets and candidate sets in reasonable time.*

*To ensure any transaction can be encoded, the induction process departs from the code table containing all singleton itemsets, i.e.,* $\{\{i\} \mid i \in \mathcal{I}\}$*. Candidate itemsets are considered in a fixed order, on frequencies and lengths, maximizing the probability that we encounter candidates that aid compression. Finally, with the same goal, we imposed an order on the itemsets in the code table. Together with the cover function, which does not allow overlap, this means that each candidate itemset can be efficiently evaluated. To further illustrate this example, the* Krimp *algorithm is given as Algorithm 2.*

**Algorithm 2** The Krimp Algorithm

```
Require: A transaction database $\mathcal{D}$ and a candidate set $\mathcal{F}$, both over
    a set of items $\mathcal{I}$
Ensure: Code table $CT$, a heuristic solution to the MDL Pattern Set
    problem
$CT \leftarrow$ Standard Code Table($\mathcal{D}$)
$\mathcal{F}_o \leftarrow \mathcal{F}$ in Standard Candidate Order
for all $F \in \mathcal{F}_o \setminus \mathcal{I}$ do
    $CT_c \leftarrow (CT \cup F)$ in Standard Cover Order
    if $L(\mathcal{D}, CT_c) < L(\mathcal{D}, CT)$ then
        $CT \leftarrow CT_c$
    end if
end for
return $CT$
```

Other examples of compression based pattern mining algorithms employing single-pass filtering include R-Krimp [32], RDB-Krimp [33], Less [24], Pack [62], and SQS [64].

**4.1.2 Iterative candidate selection.** Single-pass filtering is a very greedy search strategy. One particular point of concern is that it considers every candidate only once, in fixed order, deciding acceptance or rejection on the candidate's quality in relation to only the model mined up to that time. This means that unless the candidate order is perfect, we will see that candidates get rejected that would have been ideal later on, and hence that sub-optimal candidates will be accepted because we do not have access to the optimal candidate at that time.

The reason this strategy still provides good results is exactly the problem it aims to resolve: redundancy. For every rejected 'ideal' candidate we will (likely) see a good enough variant later on.

The optimal result, however, may be a much smaller set of patterns that describe the data much better. One way to approximate the optimal result better is to make the search less greedy. Smets and Vreeken [59] showed that iteratively greedily adding the locally optimal candidate leads to much better code tables.

1 Start with an 'empty' model $M$.

2 Select that $F \in \mathcal{F}$ that minimizes $L(\mathcal{D}, M \cup F)$.

3 Add $F$ to $M$ and remove it from $\mathcal{F}$.

4 Repeat steps 2-3 until compression can no longer be improved.

Naively, this entails iteratively re-ranking all candidates, and taking the best one. That is, with regard to Chapter 4, this approach can be viewed as the dynamic ranking approach to pattern set mining.

The naive implementation of this strategy is computationally much more demanding than single-pass filtering, with a complexity of $O(|\mathcal{F}|^2)$ opposed to $O(|\mathcal{F}|)$. On the upside, it is less prone to local minima. If one desires to explore even a larger parts of the search space, one could maintain the top-$k$ best models after each iteration instead of only the single best model. Such a strategy would essentially be a beam search and is employed by the Groei algorithm, as proposed by Siebes and Kersten [56] to find good approximations to the problem of finding the best description of the data in $k$ patterns.

Instead of exactly calculating the quality of each candidate per iteration, which requires a pass over the data and is hence expensive, we can also employ a quality estimate. To this end, the Mtv algorithm uses a convex quality estimate [44], which allows both to effectively prune a large part of the candidate space, as well as to identify the best candidate without having to calculate the actual score. Slim [59] uses an optimistic estimate, and only calculates the actual score for the top-$k$ candidates until one is accepted by MDL.

**4.1.3 Pruning.** Another improvement that can be used by any candidate filtering approach is to add a *pruning* step: patterns that were added to the model before may become obsolete later during the search. That is, due to other additions previously added patterns may no longer contribute to improved compression. To remedy this, we can *prune* the model, i.e., we can test whether removing patterns from the model results in improved compression.

Again, there are many possibilities. The most obvious strategy is to check the attained compression of all valid subsets of the current pattern set and choose the corresponding model with minimal compressed size. One could even include a new candidate pattern in this process, yet this requires considerable extra amount of computation.

A more efficient alternative is to prune only directly after a candidate $F$ is accepted. To keep the pruning search space small, one could consider each pattern in the current model for removal once after acceptance of another pattern, in a heuristic order. If pruning a pattern does not result in an increased encoded size of data and model, it apparently no longer contributes to compression. When this is the case, it is permanently removed from the model. Even simple pruning techniques like this can vastly improve the compression ratios attained by pattern-based models found by candidate filtering methods.

Pruning has been shown to be one of the key elements of Krimp [68], as it allows to remove patterns from the model for which better replacements have been found. If we would keep these now obsolete patterns, they would be in the way (in terms of cost) of other patterns. Pruning practically always improves performance, both in terms of speed, compression rates, and pattern set size [68, 23, 64].

## 4.2 Direct mining of patterns that compress

Candidate filtering is conceptually easy and generally applicable. It allows us to mine any set of candidate patterns, and then select a good subset. However, the reason for mining code tables, the pattern explosion, is also the Achilles heel of this two-stage approach. Mining, storing, and sorting candidate patterns is computationally demanding for non-trivial data. In particular as lower thresholds correspond to better models: larger candidate sets induce a larger model space, and hence allow for better models to be discovered. However, the vast majority of these patterns will never be selected or make it into the final model, the question is: can't we mine a good code table directly from data?

The space of models $\mathcal{M}$ is too erratic to allow direct sampling of high-quality code tables. We can, however, adapt the iterative candidate selection scheme above. In particular, instead of iteratively identifying the best candidate from $\mathcal{F}$, we use the current model $M$ to generate candidates that are likely good additions to the model.

What is likely to make a good addition to the model? A pattern that helps to reduce redundancy in the encoding. In our setting, this means correlations between code usages. If the code for pattern $A$ and the code for pattern $B$ often co-occur, we can gain bits using a new code $C$ meaning '$A$ and $B$'. We can hence find good candidates by mining frequent patterns in 'encoding space'. Moreover, by employing an optimistic estimate we can prune large parts of the search space, and efficiently identify the best pattern [59]. In general terms, we have

1 Start with an 'empty' model $M$.

2 Find that $F \in \mathcal{X}$ that minimizes $L(\mathcal{D}, M \cup F)$.

3 Add $F$ to $M$.

4 Repeat steps 2-3 until compression can no longer be improved.

Because of the strong dependence on the specific encoding and pattern type, providing a universal algorithmic strategy for step 2 is hardly possible—in itemset data correlations mean co-occurrences [59], in sequential data it means close-by occurrences [64], etc. In general, the current encoding of the data will have to be inspected to see if there are any 'patterns' in there that can be exploited to improve compression.

The Slim algorithm [59] was the first to implement this strategy for MDL, and induces Krimp code tables by iteratively searching for pairs of itemsets that often occur together. The union of the pair that results in the best improvement in compression is added to the code table. Although it hence considers a search space of only $O(|CT|^2)$ instead of $O(|\mathcal{F}|^2)$, its results very closely approximate the ideal local greedy strategy, or, Kramp. In particular for dense data, Slim can be orders of magnitude faster than Krimp, obtaining smaller code tables that offer much more succinct descriptions of the data.

To save computation, SQS does not iteratively identify the best candidate, but instead iteratively generates a set of candidates given the current model, considers all these candidates in turn, then generates new candidates, etc, until MDL tells it to stop.

## 5. MDL for data mining

So far, we considered compression for model selection, but it has been argued [15] and shown in the literature that it can also be used for many (data mining) tasks. For example, we already referred to the Normalized Compression Distance [41]. Another concrete example is the usage of MPEG video compression for image clustering [29].

In these examples, existing compression algorithms are used as 'black boxes' to approximate Kolmogorov complexity, and usually only dataset-level compression is required (to be precise, individual strings/objects are considered as 'datasets').

In this chapter, we are particularly interested in compression-based models that allow for inspection, so that any discovered local structure can be interpreted by domain experts. For that purpose pattern-based models that can be selected by means of the MDL principle have been developed. However, we have not yet discussed if and how these models can be used for tasks other than describing and summarizing the data.

In the following we will show how many learning and mining tasks can be naturally formalized in terms of compression, using the pattern-based models and MDL formulation described in this chapter. In particular, to be able to give more concrete details we will focus on using code tables as models. Again, it is important to note that the overall approach can be applied to other compression- and pattern-based models as well.

## 5.1 Classification

Classification is a traditional task in machine learning and data mining. Informally, it can be summarized as follows: given a training set of tuples with class labels and an 'unseen' tuple $t$ without class label, use the training data to infer the correct class label for $t$. Next, we describe a simple classification scheme based on the MDL principle [38].

**5.1.1 Classification through MDL.** If we assume that a database $\mathcal{D}$ is an i.i.d. sample from some underlying data distribution, we expect that the optimal model for this database, i.e., optimal in MDL sense, to compress an arbitrary tuple sampled from this distribution well. For this to work, we need a model that supports tuple-level compression.

In the context of code tables, we make this intuition more formal in Lemma 1.13. We say that the patterns in $CT$ are *independent* if any co-occurrence of two patterns $X, Y \in CT$ in the cover of a tuple is independent. That is, $P(XY) = P(X)P(Y)$, a Naïve Bayes [71] like assumption.

LEMMA 1.13 *Let $\mathcal{D}$ be a bag of tuples drawn from $\mathcal{T}$,* cover *a cover function, $CT$ the optimal code table for $\mathcal{D}$ and $t$ an arbitrary transaction from $\mathcal{T}$. Then, if the patterns $X \in cover(CT, t)$ are independent,*

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

*(See [38] for the proof.)*

This lemma is only valid under the Naïve Bayes like assumption, which in theory might be violated. However, by MDL, if there would be patterns $X, Y \in CT$ such that $P(XY) > P(X)P(Y)$, there will be a pattern $Z$ in the optimal code table $CT$ that covers both $X$ and $Y$.

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

LEMMA 1.14 *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two bags of tuples from $\mathcal{T}$, sampled from two different distributions, $CT_1$ and $CT_2$ the optimal code tables for $\mathcal{D}_1$ and $\mathcal{D}_2$, and $t$ an arbitrary tuple over $\mathcal{T}$. Then, by Lemma 1.13 we have*

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

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

**5.1.2 The code table classifier.** The above suggests a straightforward classification algorithm based on code tables. This classification scheme is illustrated in Figure 1.3.

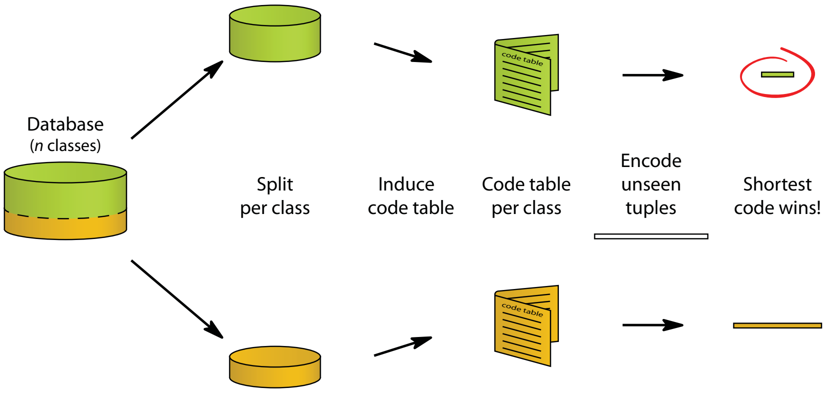

*Figure 1.3.* The code table classifier in action.

The classifier consists of a code table per class. Given a database with class labels, this database is split according to class, after which the class labels are removed from all tuples. Then, some induction method is used to obtain a code table for each single-class database. When the per-class compressors have all been constructed, classifying unseen tuples is trivial: simply assign the class label belonging to the code table that provides the minimal encoded length for the transaction.

Note that this simple yet effective scheme requires a code table to be able to compress any possible tuple, i.e., it should be possible to compute $L(t \mid CT)$ for any $t \in \mathcal{T}$. For this it is important to keep all 'primitive' patterns in the code table, i.e., those that are in the 'empty' code table. Further, to ensure valid codes all patterns should have non-zero usage, which can be achieved by, e.g., applying a Laplace correction: add one to the usage of each pattern in the code table.

Results of this scheme on itemset data, using KRIMP [38] and PACK [62], show this simple classifier performs *on par* with the best classifiers in the literature, including Naïve Bayes and SVMs. This may be considered an unexpectedly positive result, as the sole goal of each code table is to characterize and describe an individual class-based database. The fact that these code tables can in practice also be used for distinguishing samples drawn from the different distributions means they indeed capture these very well.

## 5.2 A dissimilarity measure for datasets

Comparing datasets to find and explain differences is a frequent task in many organizations. The two databases can, e.g., originate from different branches of the same organizations, such as sales records from different stores of a chain or the "same" database at different points in time. A first step towards identifying differences between datasets is to *quantify* how different two datasets are.

Although this may appear to be a simple task at first sight, in practice it turns out to be far from trivial in many cases. In particular, this is true when considering data types for which no obvious distance measures are available, such as for categorical data. In this subsection we describe a compression-based difference measure for datasets (based on [66]).

**5.2.1 Code length differences.** Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two databases with tuples drawn from the same data universe $\mathcal{T}$. The MDL principle implies that the optimal compressor induced from a database $\mathcal{D}_1$ will generally provide shorter encodings for its tuples than the optimal compressor induced from another database $\mathcal{D}_2$. This is the same principle as used by the classifier described in the previous subsection, and again we assume and exploit the tuple-level compression property.

Formally, let $M_i$ be the optimal model induced from database $\mathcal{D}_i$, and $t$ a transaction in $\mathcal{D}_1$. Then, the MDL principle implies that

$$|L(t \mid M_2) - L(t \mid M_1)|$$

- is small if $t$ is equally likely under the distributions of $\mathcal{D}_1$ and $\mathcal{D}_2$;
- is large if $t$ is more likely under the distribution of one database than under the distribution underlying the other.

Furthermore, the MDL principle implies that for the MDL-optimal models $M_1$ and $M_2$ and $t$ from $\mathcal{D}_1$, the expected average value of $L(t \mid M_2) - L(t \mid M_1)$ is positive. The next step towards a dissimilarity measure is to aggregate these code length differences over the dataset.

If we would do this naively, the resulting aggregate would depend on the size of the data. To avoid this, we normalize by dividing by the 'native' encoded size of the database, $L(\mathcal{D}_1 \mid M_1)$, and arrive at

$$ACLD(\mathcal{D}_1, M_2) = \frac{L(\mathcal{D}_1 \mid M_2) - L(\mathcal{D}_1 \mid M_1)}{L(\mathcal{D}_1 \mid M_1)}.$$

Like Kullback-Leibler divergence, $ACLD$ is asymmetric: it measures how different $\mathcal{D}_2$ is from $\mathcal{D}_1$, not vice versa. While it is reasonable to expect these to be in the same ballpark, this is not a given.

**5.2.2 The database dissimilarity measure.** The asymmetric measure allows measuring similarity of one database to another. To make it a practical measure we would like it to be symmetric. We do this by taking the maximum value of two aggregated differences, i.e., $\max\{ACLD(\mathcal{D}_1, M_2), ACLD(\mathcal{D}_2, M_1)\}$. This can easily be rewritten in terms of compressed database sizes, as follows.

Definition 1.15 *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two databases drawn from $\mathcal{T}$, and let $M_1$ and $M_2$ be their corresponding MDL-optimal models. Then, define the* dissimilarity *measure $DS$ between $\mathcal{D}_1$ and $\mathcal{D}_2$ as*

$$DS(\mathcal{D}_1, \mathcal{D}_2) = \max\left(\frac{L(\mathcal{D}_1|M_2) - L(\mathcal{D}_1|M_1)}{L(\mathcal{D}_1|M_1)}, \frac{L(\mathcal{D}_2|M_1) - L(\mathcal{D}_2|M_2)}{L(\mathcal{D}_2|M_2)}\right).$$

Using this measure, we'll obtain a score of 0 iff the databases are identical, and higher scores indicate higher dissimilarity. In theory, using MDL-optimal models we find that $DS$, like NCD [11] is a metric: the symmetry axiom holds by definition, scores cannot be negative, and it holds that $DS(\mathcal{D}_1, \mathcal{D}_2) = 0$ iff $\mathcal{D}_1 = \mathcal{D}_2$. The advantage of $DS$ over NCD is that we only have to induce two models, as opposed to four.

For heuristic model induction algorithms the metric property is difficult to prove. However, instantiating this measure for itemset data using Krimp, we obtain very good results [66]: dataset pairs drawn from the same distribution have very low dissimilarities, whereas dataset pairs from different distributions have substantially larger dissimilarities.

## 5.3 Identifying and characterising components

Though most databases are mixtures drawn from different distributions, we often assume only one distribution. Clearly, this leads to suboptimal results: the distributions need to be modeled individually.

Clustering addresses part of this problem by trying to separate the source components that make up the mixture. However, as we do not know upfront what distinguishes the different components, the appropriate distance metric is hard to define. Furthermore, in clustering we are only returned the object assignment, and not any insight in the characteristics per cluster. For example, what is typical for that cluster, and how do the different ingredients of the mixture compare to each other?

The pattern- and compression-based models described in this chapter provide all prerequisites required for data characterization, classification, and difference measurement. If a compression-based approach can be used to identify the components of a database, each represented by a pattern model, all these advantages can be obtained 'for free'.

**5.3.1 MDL for component identification.** On a high level, the goal is to discover an optimal partitioning of the database; optimal, in the sense that the characteristics of the different components are different, while the individual components are homogeneous. Translating this to MDL, the task is to partition a given database such that the total compressed size of the components is minimized—where each component is compressed by its own MDL-optimal model.

The intuition is that similar tuples of a database can be better compressed if they are assigned to the same partition and hence compressed by the same model. However, having multiple components, with corresponding models, allows models to be more specific and hence can be expected to provide better overall compression. By minimizing the total compressed size, including the sizes of the models, the different distributions of the mixture are expected to be divided over the partitions.

Following [39], we have the following problem statement:

Problem 2 (Identifying Database Components) *Let $\mathcal{D}$ be a bag of tuples drawn from $\mathcal{T}$. Find a partitioning $\mathcal{D}_1, \cdots, \mathcal{D}_k$ of $\mathcal{D}$ and associated models $M_1, \cdots, M_k$, such that* the total compressed size *of $\mathcal{D}$,*

$$\sum_{i \in \{1, \cdots, k\}} L(M_i, \mathcal{D}_i),$$

*is minimized.*

There are a few of observations we should make with regard to this problem. First of all, note that it is parameter-free: MDL determines the optimal number of components. Second, asking for both the partitioning and the models is in a sense redundant. For any partitioning, the best associated models are, of course, the optimal ones. The other way around, given a set of models, a database partitions naturally: each tuple goes to the model that compresses it best, as with classification.

The search space, however, is enormous, and solving the problem hard. An effective and efficient heuristic is to take an EM-like approach [14], starting with a random partitioning, and iteratively inducing models and re-assigning tuples to maximize compression, until convergence. Besides automatically determining the optimal number of components, this approach has been shown to find sound groupings [39].

## 5.4 Other data mining tasks

So far we covered some of the most prominent tasks in data mining. However, many more tasks have been formulated in terms of MDL and pattern-based models. Below, we briefly describe five examples.

**5.4.1 Data generation — and privacy preservation.** The MDL principle is primarily geared towards descriptive models. However, these models can also be employed as predictive models, such as in the classification example above. Furthermore, under certain conditions, compression-based models can also be used as *generative models*.

By exploiting the close relation between code lengths and probability distributions, code tables can be used for data generation. For categorical data, *synthetic* data generated from a Krimp code table has the property that the deviation between the observed and original frequencies is very small in expectation for *all* itemsets [67]. One application is privacy preservation: the generated data has the same characteristics as the original data, yet individual details are lost and specified levels of anonymity can be obtained.

**5.4.2 Missing value imputation.** Many datasets have missing values. Under the assumption these are missing without correlation to the data, they do not affect the observed overall distribution. Consequently, despite those missing values, a model of reasonable quality can be induced given sufficient data. Given such a database and corresponding model, the best estimation for a single missing value is the one that minimizes the total compressed size. We can do so both for individual tuples, a well as for databases with many missing values: by iteratively imputing the values, and inducing the model, completed datasets with very high accuracy are obtained [65].

**5.4.3 Change detection in data streams.** A database can be a mixture of different distributions, but in data streams concept drift is common: one distribution is ‘replaced’ by another distribution. In this context, it is important to detect when such change occurs. Complicating issues are that streams are usually infinite, can have high velocity, and only limited computation time is available for processing.

By first assuming that the data stream is sampled from a single distribution, a model can be induced on only few samples; how many are needed can be deduced from the attained compression ratios. Once we have a model, we can observe the compressed size of the new data; if this is considerably larger than for the earlier samples, a change has occurred and a new model should be induced. In particular for sudden distribution shifts, this scheme is highly effective [37].

**5.4.4 Coherent group discovery.** Whereas the *Identifying Database Components* problem assumes that we are interested in a partitioning of the complete database, this task aims at the discovery of coherent subsets of the data that deviate from the overall distribution. As such, it is an instance of subspace clustering. In terms of MDL, this means that the goal is to find groups that can be compressed much better by themselves than as part of the complete database.

As example application, this approach was applied to tag data obtained for different media types [36]. It was shown that using only tag information, coherent groups of media, e.g., photos, can be discovered.

**5.4.5 Outlier detection.** All databases contain outliers, but defining what an outlier exactly is and detecting them are well-known to be challenging tasks. By assuming that the number of outliers is small, and given the intuition of what an outlier is this seems a safe assumption, we know that the largest part of a dataset is ‘normal’. Hence, a model induced on the database should capture primarily what is normal, and not so much what is an outlier. Then, outlier detection can be formalized as a one-class classification problem: all tuples that are compressed well belong to the ‘normal’ distribution, while tuples that get a long encoding may be considered outliers. For transactional data, this approach performs on par with the state-of-the-art of the field [58].

## 5.5 The advantage of pattern-based models

For each and every of these tasks, we have to point out the added benefit of using a pattern-based model. Besides obtaining competitive, state-of-the-art performance, these patterns help to characterize decisions. For example, in the case of outlier detection, we can identify *why* a tuple is identified as an anomaly by pointing out the patterns of the norm it does not comply with, as well as how strongly it is an anomaly—how much effort we have to do in order to make it ‘normal’. Similar advantages hold for the classification task. For the clustering related tasks, we have the added benefit that we can offer specialized code tables, specialized descriptions per subpart of the data; we are not only told which parts of the data should go together, but also why, what patterns make these data points similar.

## 6. Challenges ahead

Above we showed that compression provides a powerful approach to both mining and using patterns in a range of data mining tasks. Here we briefly identify and discuss a number of open research problems.

### 6.1 Toward mining structured data

When compared to other data types, compressing itemset data is relatively simple. The most important reason is that the data is unordered over both rows and columns, and hence tuples can be considered as *sets* of items, and the data as a *bag* of tuples.

For 'spatial' binary data, where the order of rows and columns does matter, many tasks already become more difficult. A good example is the extension of tiling, called geometric tiling [19], which aims at finding a hierarchy of (noisy) tiles that describe the data well. Finding optimal sub-tiles is more difficult than mining itemsets, as we now also have to consider every subset of rows. Stijl efficiently finds the MDL-optimal sub-tile in order to greedily find good tilings [63].

Another possible structural constraint is time: sequences and streams are both series of data points, where sequences consist of events while data streams usually consists of complete tuples, e.g., itemsets. Initial attempts to characterize sequence data with patterns using compression include [64] and [35]. Lam et al. [34] mine sequential patterns from streams, whereas the goal of Van Leeuwen and Siebes [37] is to detect changes in data streams. All these are limited though. For example, none are suited for the high velocity of big data streams, as well as suboptimal for data consisting of shifting mixtures of distributions. Other open issues include allowing overlap between patterns, as well as allowing multiple events per time-stamp.

Adding even more structure, we have trees, graphs, as well as multi-relational data. In this area even fewer results have been published, though arguably these data types are most abundant. For graphs, SlashBurn [26] uses compression to separate communities and hubs. For multi-relational data, two variants of Krimp have been proposed [32, 33], yet their modeling power is limited by their restrictive pattern languages—nor are direct candidate mining strategies available.

Further, so far no pattern set mining approaches have been proposed for continuous data. Moreover, all data is assumed to be 'certain'. However, in bioinformatics, for example, many data is probabilistic in nature, e.g., representing the uncertainty of protein-protein interactions. Bonchi et al. [7] proposed an approach to model uncertain data by itemsets, yet they do so with 'certain' itemsets, i.e., without explicit probabilities. Mining pattern sets from numerical and uncertain data, as well as using them in compression-based models, are important future challenges.

## 6.2 Generalization

While the above challenges concern specialization for structured data types and other data primitives, another challenge concern the other direction: generalization. One of the fundamental problems in data mining is that new models, algorithms, and implementations are needed for every combination of task and data type. Though the literature flourishes, it makes the results very hard to use for non-experts.

In this chapter we have shown that patterns can actually be useful: for summarization and characterization, as well as for other tasks. One of the upcoming challenges will be to generalize compression-based data mining. Can patterns be defined in a very generic way, so that mining them and using them for modeling remains possible? For that, progress with regard to both mining and modeling needs to be made. Both are currently strongly tailored toward specific data and pattern types.

One approach may be to represent everything, both data and patterns, as queries. With such a uniform treatment, recently proposed by Siebes [55], the ideal of exploratory data mining might become reachable. Note that the high-level goal of generalizing data mining and machine learning is also pursued by De Raedt et al. [51, 21], yet with different focus: their aim is to develop declarative modeling languages for data mining, which can use existing solver technology to mine solutions.

## 6.3 Task- and/or user-specific usefulness

While obtaining very good results in practice, MDL is not a magic wand. In existing approaches, the results are primarily dependent on the data and pattern languages. In other situations it may be beneficial to take specific tasks and/or users into account. In other words, one may want to keep the purpose of the patterns in mind.

As an example, the code table classifier described in the previous section works well in practice, yet it is possibly sub-optimal. It works by modeling the class distributions, not by modeling the differences between these. Although classification is hardly typical for *exploratory* data mining, similar arguments exist for other data mining tasks.

In this chapter we ignore any background knowledge the user may have. If one is interested in the optimal model *given* certain background knowledge, this entails finding MDL-optimal models given prior distributions—which reduces to the MML [69] principle. The optimal prior can be identified using the Maximum Entropy principle [25].[3]

De Bie [13] argues that the goal of the data miner in data exploration is to model the user's belief-state, so that we can algorithmically discover those results that will be most informative to the user. At the core, this reduces to compression—with the twist that the decision whether to include a pattern is made by the user.

**6.3.1 The Optimum.** A more global issue is the efficiency of the used encodings. Whereas in Kolmogorov complexity we have access to the ultimate algorithmic compressor, the MDL principle assumes that we have access to the ultimate encoding. In practice, we have to make do with an approximation. While when constructing an encoding we can make principled choices, we often have to simplify matters to allow for fast(er) induction of good models. For instance, in Krimp it would be nice if we could encode transactions using their exact probability given the pattern set. However, calculating frequencies of an itemset given a set of itemsets and frequencies is known to be PP-hard [61]. Hence Krimp uses a (admittedly crude) approximation of this ideal. A more efficient encoding would allow to detect more fine-grained redundancy, and hence lead to smaller and better models. Currently, however, there is very little known on how to construct a good yet practical encoding.

A second global issue we need to point out is that of complexity. Intuitively, optimizing an MDL score is rather complex. However, so far we only have hardness results for a simple encoding in Boolean matrix factorization [45]. It may be that other encodings do exhibit structure that we have not yet identified, but which may be exploited for (more) efficient search. Alternatively, so far we have no theoretical results on the quality of our greedy approximations. It may be possible to construct non-trivial MDL scores that exhibit sub-modularity, which would allow approximating the quality of the greedy strategy.

Third, for now assuming the optimization problem is hard, and there are no (useful) approximation guarantees, we need to develop smart heuristics. We described the two main approaches proposed so far, candidate filtering and direct mining. Naively, the larger part of the search space $\mathcal{M}$ we consider, the better the model $M$ we'll be able to find. However, as the model space is too large, we have to find ways of efficiently considering what is good. The direct mining approach provides a promising direction, but is only as good as the quality estimation it

[3] See Chapter 4 for a more complete discussion on MaxEnt.

employs. Improving this estimation will allow to prune away more candidates, and concentrate our effort there where it matters most.

## 7. Conclusions

We discussed how to apply the MDL principle for mining sets of patterns that can be put to use. In particular, we discussed how pattern-based models can be designed and selected by means of compression, giving us succinct and characteristic descriptions of the data.

Firmly rooted in algorithmic information theory, the approach taken in this chapter states that the best set of patterns is that set that compresses the data best. We formalized this problem using MDL, described model classes that can be used to this end, and briefly discussed algorithmic approaches to inducing good models from data. Last but not least, we described how the obtained models, which are very characteristic for the data, can be used for numerous data mining tasks, making the pattern sets practically useful.

## Acknowledgments

Matthijs van Leeuwen is supported by a Post-doctoral Fellowship of the Research Foundation Flanders (FWO). Jilles Vreeken is supported by the Cluster of Excellence “Multimodal Computing and Interaction” within the Excellence Initiative of the German Federal Government.

# References

[1] P. Adriaans and P. Vitányi. Approximation of the two-part MDL code. *IEEE TIT*, 55(1):444–457, 2009.

[2] H. Akaike. A new look at the statistical model identification. *IEEE TAC*, 19(6):716–723, 1974.

[3] L. Akoglu, H. Tong, J. Vreeken, and C. Faloutsos. CompreX: Compression based anomaly detection. In *CIKM*. ACM, 2012.

[4] L. Akoglu, J. Vreeken, H. Tong, N. Tatti, and C. Faloutsos. Mining connection pathways for marked nodes in large graphs. In *SDM*. SIAM, 2013.

[5] R. Bathoorn, A. Koopman, and A. Siebes. Reducing the frequent pattern set. In *ICDM-Workshop*, pages 1–5, 2006.

[6] C. Böhm, C. Faloutsos, J.-Y. Pan, and C. Plant. Robust information-theoretic clustering. In *KDD*, pages 65–75, 2006.

[7] F. Bonchi, M. van Leeuwen, and A. Ukkonen. Characterizing uncertain data using compression. In *SDM*, pages 534–545, 2011.

[8] D. Chakrabarti, S. Papadimitriou, D. S. Modha, and C. Faloutsos. Fully automatic cross-associations. In *KDD*, pages 79–88, 2004.

[9] S. Chakrabarti, S. Sarawagi, and B. Dom. Mining surprising patterns using temporal description length. In *VLDB*, pages 606–617. Morgan Kaufmann, 1998.

[10] V. Chandola and V. Kumar. Summarization – compressing data into an informative representation. *Knowl. Inf. Sys.*, 12(3):355–378, 2007.

[11] R. Cilibrasi and P. Vitányi. Clustering by compression. *IEEE TIT*, 51(4):1523–1545, 2005.

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

[13] T. De Bie. An information theoretic framework for data mining. In *KDD*, pages 564–572. ACM, 2011.

[14] A. Dempster, N. Laird, and D. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *J. R. Statist. Soc. B*, 39(1):1–38, 1977.

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

[16] U. Fayyad and K. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *UAI*, pages 1022–1027, 1993.

[17] R. A. Fisher. On the interpretation of $\chi^2$ from contingency tables, and the calculation of P. *Journal of the Royal Statistical Society*, 85(1):87–94, 1922.

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

[19] A. Gionis, H. Mannila, and J. K. Seppänen. Geometric and combinatorial tiles in 0-1 data. In *PKDD*, pages 173–184. Springer, 2004.

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

[21] T. Guns, S. Nijssen, and L. D. Raedt. Itemset mining: A constraint programming perspective. *Artif. Intell.*, 175(12-13):1951–1983, 2011.

[22] E. Halperin and R. M. Karp. The minimum-entropy set cover problem. *TCS*, 348(2-3):240–250, 2005.

[23] H. Heikinheimo, J. K. Seppänen, E. Hinkkanen, H. Mannila, and T. Mielikäinen. Finding low-entropy sets and trees from binary data. In *KDD*, pages 350–359, 2007.

[24] H. Heikinheimo, J. Vreeken, A. Siebes, and H. Mannila. Low-entropy set selection. In *SDM*, pages 569–580, 2009.

[25] E. Jaynes. On the rationale of maximum-entropy methods. *Proc. IEEE*, 70(9):939–952, 1982.

[26] U. Kang and C. Faloutsos. Beyond caveman communities: Hubs and spokes for graph compression and mining. In *ICDM*, pages 300–309. IEEE, 2011.

[27] R. M. Karp. Reducibility among combinatorial problems. In *Proc. Compl. Comp. Comput.*, pages 85–103, New York, USA, 1972.

[28] E. Keogh, S. Lonardi, and C. A. Ratanamahatana. Towards parameter-free data mining. In *KDD*, pages 206–215, 2004.

[29] E. Keogh, S. Lonardi, C. A. Ratanamahatana, L. Wei, S.-H. Lee, and J. Handley. Compression-based data mining of sequential data. *Data Min. Knowl. Disc.*, 14(1):99–129, 2007.

[30] P. Kontkanen and P. Myllymäki. A linear-time algorithm for computing the multinomial stochastic complexity. *Inf. Process. Lett.*, 103(6):227–233, 2007.

[31] P. Kontkanen, P. Myllymäki, W. Buntine, J. Rissanen, and H. Tirri. An MDL framework for clustering. Technical report, HIIT, 2004. Technical Report 2004-6.

[32] A. Koopman and A. Siebes. Discovering relational items sets efficiently. In *SDM*, pages 108–119, 2008.

[33] A. Koopman and A. Siebes. Characteristic relational patterns. In *KDD*, pages 437–446, 2009.

[34] H. T. Lam, T. Calders, J. Yang, F. Moerchen, and D. Fradkin. Zips: Mining compressing sequential patterns in streams. In *IDEA*, pages 54–62, 2013.

[35] H. T. Lam, F. Mörchen, D. Fradkin, and T. Calders. Mining compressing sequential patterns. In *SDM*, 2012.

[36] M. van Leeuwen, F. Bonchi, B. Sigurbjörnsson, and A. Siebes. Compressing tags to find interesting media groups. In *CIKM*, pages 1147–1156, 2009.

[37] M. van Leeuwen and A. Siebes. StreamKrimp: Detecting change in data streams. In *ECML PKDD*, pages 672–687, 2008.

[38] M. van Leeuwen, J. Vreeken, and A. Siebes. Compression picks the item sets that matter. In *PKDD*, pages 585–592, 2006.

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

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

[41] M. Li, X. Chen, X. Li, B. Ma, and P. Vitanyi. The similarity metric. *IEEE TIT*, 50(12):3250–3264, 2004.

[42] C. Lucchese, S. Orlando, and R. Perego. Mining top-k patterns from binary datasets in presence of noise. In *SDM*, pages 165–176, 2010.

[43] M. Mampaey and J. Vreeken. Summarising categorical data by clustering attributes. *Data Min. Knowl. Disc.*, 26(1):130–173, 2013.

[44] M. Mampaey, J. Vreeken, and N. Tatti. Summarizing data succinctly with the most informative itemsets. *ACM TKDD*, 6:1–44, 2012.

[45] P. Miettinen and J. Vreeken. MDL4BMF: Minimum description length for Boolean matrix factorization. *ACM TKDD. In Press.*

[46] P. Miettinen and J. Vreeken. Model order selection for Boolean matrix factorization. In *KDD*, pages 51–59. ACM, 2011.

[47] S. Papadimitriou, J. Sun, C. Faloutsos, and P. S. Yu. Hierarchical, parameter-free community discovery. In *ECML PKDD*, pages 170–187, 2008.

[48] B. Pfahringer. Compression-based feature subset selection. In *Proc. IJCAI'95 Workshop on Data Engineering for Inductive Learning*, pages 109–119, 1995.

[49] B. A. Prakash, J. Vreeken, and C. Faloutsos. Spotting culprits in epidemics: How many and which ones? In *ICDM*. IEEE, 2012.

[50] J. Quinlan. *C4.5: Programs for Machine Learning.* Morgan-Kaufmann, Los Altos, California, 1993.

[51] L. D. Raedt. Declarative modeling for machine learning and data mining. In *ECML PKDD*, pages 2–3, 2012.

[52] J. Rissanen. Modeling by shortest data description. *Automatica*, 14(1):465–471, 1978.

[53] G. Schwarz. Estimating the dimension of a model. *Annals Stat.*, 6(2):461–464, 1978.

[54] H. Shao, B. Tong, and E. Suzuki. Extended MDL principle for feature-based inductive transfer learning. *Knowl. Inf. Sys.*, 35(2):365–389, 2013.

[55] A. Siebes. Queries for data analysis. In *IDA*, pages 7–22, 2012.

[56] A. Siebes and R. Kersten. A structure function for transaction data. In *SDM*, pages 558–569. SIAM, 2011.

[57] A. Siebes, J. Vreeken, and M. van Leeuwen. Item sets that compress. In *SDM*, pages 393–404. SIAM, 2006.

[58] K. Smets and J. Vreeken. The odd one out: Identifying and characterising anomalies. In *SDM*, pages 804–815. SIAM, 2011.

[59] K. Smets and J. Vreeken. Slim: Directly mining descriptive patterns. In *SDM*, pages 236–247. SIAM, 2012.

[60] J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu. Graphscope: parameter-free mining of large time-evolving graphs. In *KDD*, pages 687–696, 2007.

[61] N. Tatti. Computational complexity of queries based on itemsets. *Inf. Process. Lett.*, 98(5):183–187, 2006.

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

[63] N. Tatti and J. Vreeken. Discovering descriptive tile trees by fast mining of optimal geometric subtiles. In *ECML PKDD*. Springer, 2012.

[64] N. Tatti and J. Vreeken. The long and the short of it: Summarizing event sequences with serial episodes. In *KDD*. ACM, 2012.

[65] J. Vreeken and A. Siebes. Filling in the blanks: Krimp minimisation for missing data. In *ICDM*, pages 1067–1072. IEEE, 2008.

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

[67] J. Vreeken, M. van Leeuwen, and A. Siebes. Preserving privacy through data generation. In *ICDM*, pages 685–690. IEEE, 2007.

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

[69] C. Wallace. *Statistical and inductive inference by minimum message length.* Springer-Verlag, 2005.

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

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