Patterns that Matter

Matthijs van Leeuwen

Netherlands
Bioinformatics
Centre

The research reported in this thesis has been supported by NBIC, the Netherlands Bioinformatics Centre (BioRange project SP4.3.1).

SIKS Dissertation Series No. 2010-01
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Cover photo taken in Hongkong, while visiting for CIKM'09.
Printed by Ridderprint, the Netherlands.

ISBN: 978-90-393-5235-9

Patterns that Matter

Pakkende Patronen
(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de rector magnificus, prof.dr. J.C. Stoof, ingevolge het besluit van het college voor promoties in het openbaar te verdedigen op dinsdag 9 februari 2010 des middags te 2.30 uur

door

Matthijs van Leeuwen

geboren op 19 september 1981
te Woerden

Promotor: Prof.dr. A.P.J.M. Siebes

Contents

1 Introduction **1**

2 Krimp – Mining Itemsets that Compress **7**
- 2.1 Introduction 8
- 2.2 Theory 10
- 2.3 Algorithms 18
- 2.4 Interlude 25
- 2.5 Classification by Compression 26
- 2.6 Related Work 29
- 2.7 Experiments 31
- 2.8 Discussion 44
- 2.9 Conclusions 46

3 Characterising the Difference **53**
- 3.1 Introduction 54
- 3.2 Preliminaries 55
- 3.3 Database Dissimilarity 56
- 3.4 Characterising Differences 65
- 3.5 Related Work 70
- 3.6 Conclusions 71

4 Data Generation for Privacy Preservation **73**
- 4.1 Introduction 74
- 4.2 The Problem 75
- 4.3 Preliminaries 78
- 4.4 Krimp Categorical Data Generator 78
- 4.5 Experiments 81
- 4.6 Discussion 88
- 4.7 Conclusions 90

5 StreamKrimp – Detecting Change in Data Streams 91
5.1 Introduction 92
5.2 The Problem Assuming Unbounded Storage 93
5.3 The Problem Assuming Bounded Storage 95
5.4 The Algorithm 97
5.5 Experiments 101
5.6 Related Work 106
5.7 Discussion 108
5.8 Conclusions 108

6 Identifying the Components 109
6.1 Introduction 110
6.2 Problem Statement 111
6.3 Datasets 112
6.4 Model-Driven Component Identification 113
6.5 Data-Driven Component Identification 118
6.6 Discussion 122
6.7 Related Work 123
6.8 Conclusion 124

7 Compressing Tags to Find Interesting Media Groups 125
7.1 Introduction 126
7.2 The Problem 128
7.3 The Algorithm 130
7.4 Data Pre-processing 134
7.5 Experiments 136
7.6 Related Work 142
7.7 Conclusions 143

8 Conclusions 149

Bibliography 153

Index 163

Chapter 1

Introduction

Over the past 30 years, innovations in *Computer Science* and *Information Technology* have radically changed the world. We now live in the 'digital age' and everything and everybody is connected. For this thesis, two phenomena in particular make the difference with, say, 30 years ago. First, with increasing storage capacities becoming available, more and more data is stored and thus becomes available for all kinds of purposes. Second, the Internet has made collecting and organising data world-wide quick, cheap and easy. Companies have established specialised 'data warehouses' with the sole purpose to collect as much company data as possible. Hence, the need arises for smart methods that can be used to analyse large amounts of data and extract useful information from it. Besides, since computers have become much faster, smaller and cheaper, there has been a huge increase in the computational power available, making large-scale data analysis possible.

Bioinformatics

This situation is true for many different businesses, but also for many disciplines in science. As an example, let us consider the field of *Bioinformatics*. This is the interdisciplinary area between *Biology* and *Computer Science*, in which, simply put, computers are used to analyse biological datasets. Everyone is aware of many problems that are investigated in this field. For example, bioinformaticians investigate the causes of diseases and help develop drugs to cure these. Results of such research lines may directly benefit many ill people all over the world. As a second example, the 'tree of life' as we currently know it is a result of computational evolutionary biology.

A typical bioinformatics research area is sequence analysis, well-known for the international Human Genome Project that aimed to map all genes of the human genome. Other major areas are, e.g. analysis of gene expression, protein structure prediction, modelling biological systems, and computational evolutionary biology.

Technological advances have resulted in more and larger datasets in bioinformatics. Especially high–throughput screening methods contributed to this. A prime example of this is microarray technology, which can be used to measure changes in the expression of hundreds of genes at once. Ongoing advances with nanotechnology and 'lab–on–a–chip' technology can be expected to give plenty of new data opportunities in the (near) future.

As a result, databases with different kinds of data are already available in bioinformatics: databases with genome sequences, gene expression data, protein structures, scientific articles, biochemical data, physical data, and so on. For many research problems, it is required to combine information from multiple sources. This can be complicated, as these databases are often large, located in different physical locations and regularly contain missing and/or conflicting data.

With plenty of (complex) data readily available, new data being collected all the time, and so many important open problems, it is obvious that bioinformatics is an important field of research. To be able to solve problems, bioinformatics needs adequate generic methods for data analysis, data exploration, and knowledge discovery.

Data Mining

Finding patterns in data has been the subject of research for a long time. Within *Mathematics*, *Statistics* has a very long history and many well-founded methods have been developed. A related discipline within *Artificial Intelligence* is *Machine Learning*, which aims to allow computers to learn from data. With the upcoming of computers and *Database Technology*, and combining techniques from both disciplines, a new paradigm emerged in the 90s: *Data Mining*, or *Knowledge Discovery in Databases.* Nowadays, this is a flourishing field within *Computer Science.* It aims to develop generic methods that can be used to extract knowledge from large amounts of data using a computer. In other words, *Data Mining* seeks to answer the problems that came along with the two phenomena mentioned at the start of this section.

Broadly speaking, two types of data mining can be distinguished: *Predictive Data Mining* and *Descriptive Data Mining.* The main difference between these two types lies in their goals. The former intends to build a model on the data that can be used to *predict*, e.g. stock prices in the near future, whereas the latter intends to build a model that *describes* what is in the data. Or, in other words, one can either try to use the past to predict the future, or try to describe the past to discover useful knowledge.

In this thesis, we only consider *Descriptive Data Mining*, as we want to get as much insight in the data as possible. Our goal is to avoid 'black boxes' – we prefer to use methods that allow for human inspection and interpretation, in order to maximise our understanding of both the data and the methods. One of the best-known concepts in *Descriptive Data Mining* that allows for easy interpretation is *pattern mining* [50, 87]. More in particular, we consider *theory mining* [81], which considers patterns that describe subsets of a database. Advantages of these patterns are that they are easy to interpret individually, and that it is always clear which part of the database a pattern describes.

Informally, the task is to find all 'interesting' subgroups, given a database, a pattern language and a selection criterion. Here, the selection criterion is used to determine which patterns are 'interesting' and which are not. Coming up with an adequate selection criterion is more difficult than it may seem, as it is usually extremely hard to specify a priori what is 'interesting'. A famous instance of theory mining is *frequent pattern mining* [7], which uses a minimal frequency constraint as selection criterion: the task is to identify all patterns that occur at least so many times in the database.

The Problem

Since its introduction almost 20 years ago, frequent pattern mining has received ample attention and many algorithms have been designed [45]. However, frequent pattern mining, and pattern mining in general, suffers from a problem that makes using it in practice very difficult. That is, choosing a suitable 'interestingness' constraint is virtually impossible. When a tight (high frequency) constraint is chosen, only very few and unsurprising patterns are selected, while a loose (low frequency) constraint results in humongous amounts of patterns. This is appropriately dubbed the *pattern explosion*.

The severity of this pattern explosion is aptly illustrated by the fact that the set of frequent patterns is often orders of magnitude larger than the database from which they originate. That is, the complete description of the data is much larger than the data itself. Clearly, this does not match the objective of *Descriptive Data Mining*; it is more insightful for a domain expert to manually inspect the entire database than the pattern set.

In this thesis, we will address the pattern explosion in pattern mining. The research problem can be stated as:

Develop methods that find small, human–interpretable, pattern–based descriptive models on large datasets.

Note that this problem statement is rather generic, but this very well matches our objectives. From a data mining perspective, the problem is very fundamental and this asks for fundamental solutions; we are dealing with data mining foundations here. Any methods developed must be applicable to a large variety of situations; to all sorts of problems and different kinds of data, both in bioinformatics and other disciplines.

Challenges

The biggest challenge lies in designing a measure that decides whether a model is good or not. What makes a pattern 'interesting'? What makes a model 'good'? Especially without any a priori knowledge on the data and/or application, this seems hard to define.

Even when a priori knowledge is available, it is often problematic to come up with a suitable measure. In bioinformatics, for example, researchers that have data available for analysis do not always know what is in there and what they are looking for. "Tell me something interesting" is not very helpful when designing an interestingness measure. Hence, we should come up with a very generic quality measure for our models.

In addition to this challenge, there are other difficulties that need attention. To name a few:

i The *entire* database must be described by its model.

ii To keep models as small as possible, redundancy should be avoided.

iii Since there are many patterns, there are also many pattern-based models. We have to ensure that we find a good model within reasonable time.

iv Large datasets should not pose any problems, so efficiency in terms of computation speed and storage requirements is an important issue.

Approach and Contributions

The most important change with respect to 'traditional' pattern mining is that we do not look at individual patterns, but at *pattern sets* that make up our models. We argue that the pattern explosion is caused by determining 'interestingness' for each pattern individually, and that it is far more effective and fruitful to consider a set of patterns as a whole.

To this end, we use compression. That is, we propose the following solution to the problem at hand:

The best set of patterns is that set that compresses the data best.

In Chapter 2, we introduce the theoretical foundations on which our models and algorithms are built. For this, we use the *Minimum Description Length principle* [48], which is a practical version of *Kolmogorov Complexity* [73]. MDL says that, given data and a set of models, the best model is that model that compresses the data best. The Kolmogorov Complexity of given data is the length of the shortest computer program that outputs this data. Contrary to MDL though, Kolmogorov Complexity is uncomputable.

We introduce pattern-based models that compress the data and heuristic methods that find such models. The main algorithm, called Krimp, will be used as groundwork for all research carried out throughout this thesis. Each subsequent chapter shows how compression can be applied to a different *Knowledge Discovery* problem. The following gives a brief overview.

Characterising differences In Chapter 3, we show how compression can be used to both quantify and characterise differences between databases. We introduce a generic database dissimilarity measure and show that the pattern-based Krimp models allow for insightful inspection and visualisations.

Preserving privacy Chapter 4 deals with a very contemporary subject: privacy in data. We propose to preserve people's privacy by replacing the original data with synthetic, generated data. We introduce an algorithm that generates data from a Krimp model that is virtually indiscernible from the original data, while preserving privacy.

Detecting changes In many cases, one would like to know immediately when change occurs. Consider, e.g. stock markets or very ill patients in a hospital. In Chapter 5, we present a compression-based method that detects sudden changes over time.

Identifying database components Databases usually contain mixtures of entity types that each have their own characteristic behaviour. For supermarket data, for example, we expect retired people to buy different products than students. It can be very beneficial to identify and characterise these database components. For this purpose we introduce two methods, again based on compression, in Chapter 6.

Finding interesting groups In Chapter 7, we use compression to find coherent groups in a database and apply this to tag data, obtained from Flickr[1], to find interesting groups of photos. The main difference with

[1] `www.flickr.com`

the previous problem is that we do not attempt to characterise the entire database. Instead, we focus on small groups that are homogeneous within, but different from the rest of the database.

Each chapter contains an extensive experimental evaluation, to show that the proposed generic methods perform well under a variety of realistic conditions. Throughout this thesis, we consider different kinds of data and databases from different sources. The most widely used data type is standard ‘market basket data’, or itemset data, which basically means that we have a matrix consisting of binary values.

We consider both real and synthetic data. Obviously, synthetic data especially plays a major role in Chapter 4. In Chapter 5, we consider data streams for our change detection scheme and we use large tag datasets from Flickr in Chapter 7.

Chapter 2

Krimp – Mining Itemsets that Compress

One of the major problems in pattern mining is the explosion of the number of results[1]. Tight constraints reveal only common knowledge, while loosening these leads to an explosion in the number of returned patterns. This is caused by large groups of patterns essentially describing the same set of transactions. In this chapter we approach this problem using the MDL principle: the best set of patterns is that set that compresses the database best.

For this task we introduce the KRIMP algorithm. Experimental evaluation shows that typically only hundreds of itemsets are returned; a dramatic reduction, up to 7 orders of magnitude, in the number of frequent item sets. These selections, called code tables, are of high quality. This is shown with compression ratios, swap-randomisation, and the accuracies of the code table-based KRIMP classifier, all obtained on a wide range of datasets. Further, we extensively evaluate the heuristic choices made in the design of the algorithm.

[1]This chapter has been accepted for publication as [112]: J. Vreeken, M. van Leeuwen & A. Siebes. Krimp – Mining Itemsets that Compress. In *Data Mining and Knowledge Discovery*, Springer.

It is an extended version of work published at SDM'06 as A. Siebes, J. Vreeken & M. van Leeuwen [101] and at ECML PKDD'06 as M. van Leeuwen, J. Vreeken & A. Siebes [70].

2.1 Introduction

Patterns

Without a doubt, *pattern mining* is one of the most important concepts in data mining. In contrast to *models*, patterns describe only part of the data [50, 87]. In this thesis, we consider one class of pattern mining problems, viz., *theory mining* [81]. In this case, the patterns describe interesting subsets of the database.

Formally, this task has been described by Mannila & Toivonen [80] as follows. Given a database $\mathcal{D}$, a language $\mathcal{L}$ defining subsets of the data and a selection predicate q that determines whether an element $\phi \in \mathcal{L}$ describes an interesting subset of $\mathcal{D}$ or not, the task is to find

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

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

The best known instance of theory mining is *frequent set mining* [7]; this is the problem we will consider throughout this chapter. The standard example for this is the analysis of shopping baskets in a supermarket. Let $\mathcal{I}$ be the set of items the store sells. The database $\mathcal{D}$ consists of a set of *transactions* in which each transaction t is a subset of $\mathcal{I}$. The pattern language $\mathcal{L}$ consists of *itemsets*, i.e. again sets of items. The *support* of an itemset X in $\mathcal{D}$ is defined as the number of transactions that contain X, i.e.

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

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

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

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

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

a simple level-wise search algorithm suffices to compute all the frequent itemsets. Many efficient algorithms for this task are known, see, e.g. Goethals & Zaki [45]. Note, however, that since the size of the output can be exponential in the number of items, the term efficient is used w.r.t. the size of the output. Moreover, note that whenever $\mathcal{L}$ and q satisfy an A Priori like property, similarly efficient algorithms exist [80].

Sets of Patterns

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

This pattern explosion is caused by the locality of the minimal support constraint; each individual itemset that satisfies the constraint is added to the result set, independent of the already returned sets. Hence, we end up with a rather redundant set of patterns, in which many patterns essentially describe the same part of the database. One could impose additional constraints on the individual itemsets to reduce their number, such as closed frequent itemsets [91]. While this somewhat alleviates the problem, redundancy remains an issue.

We take a different approach: rather than focusing on the individual frequent itemsets, we focus on the resulting set of itemsets. That is, we want to find the *best set* of (frequent) itemsets. The question is, of course, what is the best set? Clearly, there is no single answer to this question. For example, one could search for small sets of patterns that yield good classifiers, or show maximal variety [60, 61].

We view finding the best set of itemsets as an *induction problem*. That is, we want to find the set of patterns that *describe the database best*. There are many different induction principles. So, again the question is which one to take?

The classical statistical approach [104] would be to basically test hypotheses, meaning that we would have to test every possible set of itemsets. Given the typically huge number of frequent itemsets and the exponentially larger number of sets of itemsets, testing all these pattern sets individually does not seem a computationally attractive approach.

Alternatively, the Bayesian approach boils down to updating the *a priori distribution* with the data [16]. This update is computed using Bayes' rule, which requires the computation of $P(\mathcal{D} \mid M)$. How to define this probability for sets of frequent itemsets is not immediately obvious. Moreover, our primary goal is to find a *descriptive* model, not a *generative* one[2]. For the same reason, principles that are geared towards predictive models, such as *Statistical Learning Theory* [109], are not suitable. Again, we are primarily interested in a descriptive model, not a predictive one.

[2] Although, in our recent research, we have built generative models from the small number of selected itemsets that generates data virtually indiscernible from the original [111].

The *Minimal Description Length Principle* (MDL) [48, 49, 97] on the other hand, is geared towards descriptions of the data. One could summarise this approach by the slogan: *the best model compresses the data best.* By taking this approach we do not try to compress the set of frequent itemsets, rather, we want to find that set of frequent itemsets that yields the best *lossless* compression of the database.

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

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

Although MDL removes the need for user-defined parameters, it comes with its own problems: only heuristics, no guaranteed algorithms. However, our experiments show that these heuristics give a dramatic reduction in the number of itemsets. Moreover, the set of patterns discovered is characteristic of the database as independent experiments verify; see Section 2.7.

We are not the first to address the pattern explosion, nor are we the first to use MDL. We are the first, however, to employ the MDL principle to select the best pattern set. For a discussion of related work, see Section 2.6.

This chapter is organised as follows. First, we cover the theory of using MDL for selecting itemsets, after which we define our problem formally and analyse its complexity. We introduce the heuristic Krimp algorithm for solving the problem in Section 2.3. In a brief interlude we provide a small sample of the results. We continue with theory on using MDL for classification, and introduce the Krimp classifier in Section 2.5. Related work is discussed in Section 2.6. Section 2.7 provides extensive experimental validation of our method, as well as an evaluation of the heuristic choices made in the design of the Krimp algorithm. We round up with discussion in Section 2.8 and conclude in Section 2.9.

2.2 Theory

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

MDL

MDL (Minimum Description Length) [48, 97], like its close cousin MML (Minimum Message Length) [113], is a practical version of Kolmogorov Complexity [73]. All three embrace the slogan *Induction by Compression*. For MDL, this principle can be roughly described as follows.

Given a set of models[3] $\mathcal{H}$, the best model $H \in \mathcal{H}$ is the one that minimises

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

in which

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

This is called two-part MDL, or *crude* MDL. As opposed to *refined* MDL, where model and data are encoded together [49]. We use two-part MDL because we are specifically interested in the compressor: the set of frequent itemsets that yields the best compression. Further, although refined MDL has stronger theoretical foundations, it cannot be computed except in only some special cases.

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

MDL for Itemsets

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

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

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

[3] MDL-theorists tend to talk about *hypothesis* in this context, hence the $\mathcal{H}$; see Grünwald [48] for the details.

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

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

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

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

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

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

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

Note that to ensure that we can decode an encoded database uniquely we assume that $\mathcal{C}$ is a *prefix code*, in which no code is the prefix of another code [34]. (Confusingly, such codes are also known as *prefix-free* codes [73].)

Since MDL is concerned with the best compression, the codes in CT should be chosen such that the most often used code has the shortest length. That is, we should use an optimal prefix code. Note that in MDL we are never interested in materialised codes, but only in the complexities of the model and the data. Therefore, we are only interested in the *lengths* of the codes of itemsets $X \in CT$. As there exists a nice correspondence between code lengths and probability distributions (see, e.g. [73]), we can calculate the optimal code lengths through the Shannon entropy. So, to determine the complexities we do not have to operate an actual prefix coding scheme such as Shannon-Fano or Huffman encoding.

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

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

Moreover, this code is optimal in the sense that it gives the smallest expected code size for data sets drawn according to P. For the proof, please refer to Theorem 5.4.1 in Cover & Thomas [34].

The optimality property means that we introduce no bias using this code length. The probability distribution induced by a cover function is, of course, simply given by the relative usage frequency of each of the item sets in the code table. To determine this, we need to know how often a certain code is used. We define the *usage* count of an itemset $X \in CT$ as the number of transactions t from $\mathcal{D}$ where X is used to cover. Normalised, this frequency represents the probability that that code is used in the encoding of an arbitrary $t \in \mathcal{D}$. The optimal code length [73] then is $-log$ of this probability, and a code table is optimal if all its codes have their optimal length. Note that we use fractional lengths, not integer-valued lengths of materialised codes. This ensures that the length of a code accurately represents its usage probability, and since we are not interested in materialised codes, only relative lengths are of importance. After all, our ultimate goal is to score the optimal code table and not to actually compress the data. More formally, we have the following definition.

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

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

The probability *of $X \in CT$ being used in the cover of an arbitrary transaction $t \in \mathcal{D}$ is thus given by*

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

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

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

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

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

are optimal for $\mathcal{D}$.

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

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

Lemma 2. *Let $\mathcal{D}$ be a transaction database over $\mathcal{I}$, CT be a code table over $\mathcal{I}$ and code-optimal for $\mathcal{D}$,* cover *a cover function, and usage the usage function for* cover*.*

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

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

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

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

With Lemma 2, we can compute $L(\mathcal{D} \mid H)$. To use the MDL principle, we still need to know what $L(H)$ is, i.e. the encoded size of a code table.

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

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

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

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

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

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

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

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

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

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

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

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

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

The Problem

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

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

More formally, we define the problem as follows.

Problem 1 (Minimal Coding Set)**.** *Let $\mathcal{I}$ be a set of items and let $\mathcal{D}$ be a dataset over $\mathcal{I}$,* cover *a cover function, and $\mathcal{F}$ a candidate set. Find the smallest coding set $CS \subseteq \mathcal{F}$ such that for the corresponding code table CT the total compressed size, $L(\mathcal{D}, CT)$, is minimal.*

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

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

How Hard is the Problem?

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

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

In other words, there are

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

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

Lemma 3. *For one transaction over a set of items $\mathcal{I}$, the number of possible ways to cover it is given by $NCP(\mathcal{I})$.*

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

So, even for a rather small set $\mathcal{I}$ and a database of only *one* transaction, the search space we are facing is already huge. Table 2.1 gives an approximation of NCP for the first few sizes of $\mathcal{I}$. Clearly, the search space is far too large to consider exhaustively.

To make matters worse, there is no useable structure that allows us to prune levelwise as the attained compression is not monotone w.r.t. the addition of itemsets. So, without calculating the usage of the itemsets in CT, it is generally impossible to call the effects (improvement or degrading) on the compression when an itemset is added to the code table. This can be seen as follows.

Given a database $\mathcal{D}$, itemsets X and Y such that $X \subset Y$, and a coding set CS, all over $\mathcal{I}$. The addition of X to CS, can lead to a degradation of the compression, first and foremost as X may add more complexity to the code table than is compensated for by using X in encoding $\mathcal{D}$. Second, X may get in 'the way' of itemsets already in CS, as such providing those itemsets with lower usage, longer codes and thus leading to massively worse compression. Instead, let us consider adding Y. While more complex, exactly those items

Table 2.1: The number of cover possibilities for a database of one (1) transaction over $\mathcal{I}$.

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

$Y \setminus X$ may replace the hindered itemsets. As such Y may circumvent getting 'in the way', and thus lead to an improved compression. However, this can just as well be the other way around, as exactly those items can also lead to low usage and/or overlap with other/more existing itemsets in CS.

2.3 Algorithms

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

Basic Heuristic

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

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

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

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

Standard Cover Function

From the problem complexity analysis in the previous section it is quite clear that finding an optimal cover of the database is practically impossible, even

Algorithm 1 Standard Code Table

```
StandardCodeTable($\mathcal{D}$) :
1. $CT \leftarrow \emptyset$
2. for all $X \in \mathcal{I}$ do
3.     insert $X$ into $CT$
4.     $usage_{\mathcal{D}}(X) \leftarrow supp_{\mathcal{D}}(X)$
5.     $code_{CT}(x) \leftarrow$ optimal code for $X$
6. end for
7. return $CT$
```

if we are given the optimal set of itemsets as the code table: examining all $|CT|!$ possible permutations is already virtually impossible for one transaction, let alone expanding this to all possible combinations of permutations for all transactions.

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

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

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

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

We call this the **Standard Cover Order**. The rationale is as follows. To reach a good compression we need to replace as many individual items as possible, by as few and short as possible codes. The above order gives priority to long itemsets, as these can replace as many as possible items by just one code. Further, we prefer those itemsets that occur frequently in the database

Algorithm 2 Standard Cover

```
StandardCover($t, CT$) :
1. $S \leftarrow$ smallest $X \in CT$ in Standard Cover Order for which $X \subseteq t$
2. if $t \setminus S = \emptyset$ then
3.     $Res \leftarrow \{S\}$
4. else
5.     $Res \leftarrow \{S\} \cup$ StandardCover($t \setminus S, CT$)
6. end if
7. return $Res$
```

to be used as often as possible, resulting in high usage values and thus short codes. We rely on MDL not to select overly specific itemsets, as such sets can only be infrequently used and would thus receive relatively long codes.

Standard Candidate Order

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

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

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

A welcome advantage of the standard orders for both the cover function and the candidate order is that we can easily keep the code table sorted. First, the length of an itemset is readily available. Second, with this candidate order we know that any candidate itemset for a particular length will have to be inserted after any already present code table element with the same length. Together, this means that we can insert a candidate itemset at the right position in the code table in O(1) if we store the code table elements in an array (over itemset length) of lists.

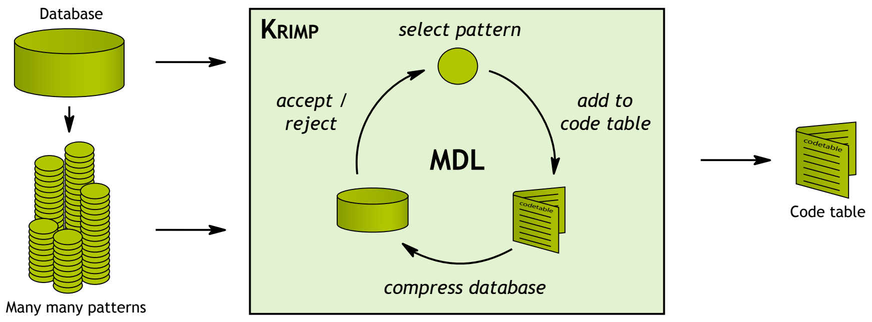


Figure 2.1: Krimp in action.

The Krimp Algorithm

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

– Start with the standard code table ST;

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

This basic scheme is formalised as the Krimp algorithm given as Algorithm 3. For the choice of the name: 'krimp' is Dutch for 'to shrink'. The Krimp pattern selection process is illustrated in Figure 2.1.

Algorithm 3 Krimp

```
Krimp($\mathcal{D}, \mathcal{F}$) :
1. $CT \leftarrow$ StandardCodeTable($\mathcal{D}$)
2. $\mathcal{F}_o \leftarrow \mathcal{F}$ in **Standard Candidate Order**
3. **for all** $F \in \mathcal{F}_o \setminus \mathcal{I}$ **do**
4.    $CT_c \leftarrow (CT \cup F)$ in **Standard Cover Order**
5.    **if** $L(\mathcal{D}, CT_c) < L(\mathcal{D}, CT)$ **then**
6.       $CT \leftarrow CT_c$
7.    **end if**
8. **end for**
9. **return** $CT$
```

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

Now, it may seem that each iteration of Krimp can only lessen the usage of an itemset in CT. For, if $F_1 \cap F_2 \neq \emptyset$ and F_2 is used before F_1 by the standard cover function, the usage of F_1 will go down (provided the support of F_2 does not equal zero). While this is true, it is not the whole story. Because, what happens if we now add an itemset F_3, which is used before F_2 such that:

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

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

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

Pruning

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

$$\begin{aligned}
CT_1 &= \{\{X_1, X_2\}, \{X_1\}, \{X_2\}, \{X_3\}\} \\
CT_2 &= \{\{X_1, X_2, X_3\}, \{X_1, X_2\}, \{X_1\}, \{X_2\}, \{X_3\}\} \\
CT_3 &= \{\{X_1, X_2, X_3\}, \{X_1\}, \{X_2\}, \{X_3\}\}
\end{aligned}$$

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

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

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

To cut the pruning search space further, we do not consider all valid subsets of CT, but iteratively consider itemsets $X \in CT$ of which $usage_{\mathcal{D}}(X)$ has decreased for removal. The rationale is that for these itemsets we know that their code lengths have increased; therefore, it is possible that these sets now harm the compression.

In line with the standard order philosophy, we first consider the itemset with the smallest usage and thus the longest code. If by pruning an itemset the total encoded size decreases, we permanently remove it from the code table. Further, we then update the list of prune candidates with those item sets whose usage consequently decreased. This post-acceptance pruning strategy is formalised in Algorithm 4. We refer to the version of Krimp that employs this pruning strategy (which would be on line 6 of Algorithm 3) as Krimp with pruning. In Section 2.7 we will show that employing pruning improves the performance of Krimp.

Algorithm 4 Code Table Post-Acceptance Pruning

```
CodeTablePostAcceptancePruning($CT_c$, $CT$) :
1.  $PruneSet \leftarrow \{ X \in CT \mid usage_{CT_c}(X) < usage_{CT}(X) \}$
2.  while $PruneSet \neq \emptyset$ do
3.      $PruneCand \leftarrow X \in PruneSet$ with lowest $usage_{CT_c}(X)$
4.      $PruneSet \leftarrow PruneSet \setminus PruneCand$
5.      $CT_p \leftarrow CT_c \setminus PruneCand$
6.      if $L(\mathcal{D}, CT_p) < L(\mathcal{D}, CT_c)$ then
7.          $PruneSet \leftarrow PruneSet \cup \{ X \in CT_p \mid usage_{CT_p}(X) < usage_{CT_c}(X) \}$
8.          $CT_c \leftarrow CT_p$
9.      end if
10. end while
11. return $CT_c$
```

Complexity

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

Given a set of (frequent) itemsets $\mathcal{F}$, we first order this set, requiring $O(|\mathcal{F}| \log |\mathcal{F}|)$ time. Then, every element $F \in \mathcal{F}$ is considered once. Using a hash-table implementation we need only $O(1)$ to insert an element at the right position in CT, keeping CT ordered. To calculate the total encoded size $L(\mathcal{D}, CT)$, the *cover* function is applied to each $t \in \mathcal{D}$. For this, the standard cover function considers each $X \in CT$ once for a t. Checking whether X is an (uncovered) subset of t takes at most $O(|\mathcal{I}|)$. Therefore, covering the full database takes $O(|\mathcal{D}| \times |CT| \times |\mathcal{I}|)$ time. Then, optimal code lengths and the total compressed size can be computed in $O(|CT|)$.

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

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

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

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

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

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

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

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

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

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

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

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

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

2.4 Interlude

Before we continue with more theory, we will first present some results on a small number of datasets to provide the reader with some measure and intuition on the performance of Krimp. To this end, we ran Krimp with post-acceptance pruning on six datasets, using all frequent itemsets mined at *minsup* = 1 as candidates. The results of these experiments are shown in Table 2.2. Per dataset, we show the number of transactions and the number of candidate itemsets. From these latter figures, the problem of the pattern explosion becomes clear: up to 5.5 billion itemsets can be mined from the *Mushroom* database, which consists of only 8124 transactions. It also shows that Krimp successfully battles this explosion, by selecting only hundreds of itemsets from millions up to billions. For example, from the 5.5 billion for *Mushroom*, only 442 itemsets are chosen; a reduction of 7 orders of magnitude.

For the other datasets, we observe the same trend. In each case, fewer than 2000 itemsets are selected, and reductions of many orders of magnitude are attained. The number of selected itemsets depends mainly on the characteristics of the data. These itemsets, or the code tables they form, compress the data to a fraction of its original size. This indicates that very characteristic itemsets are chosen, and that the selections are non-redundant.

Further, the timings for these experiments show that the compression-based selection process, although computationally complex, is a viable approach in practice. The selection of the above mentioned 442 itemsets from 5.5 billion itemsets takes under 4 hours. For the *Adult* database, Krimp considers over 400.000 itemsets per second, and is limited not by the CPUs but by the rate with which the itemsets can be read from the hard disk.

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

Table 2.2: Results of running Krimp on a few datasets.

			Krimp		
Dataset	$\|\mathcal{D}\|$	$\|\mathcal{F}\|$	$\|CT \setminus \mathcal{I}\|$	$\frac{L(\mathcal{D},CT)}{L(\mathcal{D},ST)}\%$	time
Adult	48842	58461763	1303	24.4	2m25
Chess (kr–k)	28056	373421	1684	61.6	13s
Led7	3200	15250	152	28.6	0.05s
Letter recognition	20000	580968767	1780	35.7	52m33
Mushroom	8124	5574930437	442	20.6	3h40
Pen digits	10992	459191636	1247	42.3	31m33

For all datasets the candidate set $\mathcal{F}$ was mined with *minsup* = 1, and Krimp with post-acceptance pruning was used. For Krimp, the size of the resulting code table (minus the singletons), the compression ratio and the runtime is given. The compression ratio is the encoded size of the database with the obtained code table divided by the encoded size with the standard code table. Timings recorded on quad-core 3.0 Ghz Xeon machines.

2.5 Classification by Compression

In this section, we describe a method to verify the quality of the Krimp selection in an independent way. To be more precise, we introduce a simple classification scheme based on code tables. We answer the quality question by answering the question: how well do Krimp code tables classify? For this, classification performance is compared to that of state-of-the-art classifiers in Section 2.7.

Classification through MDL

If we assume that our database of transactions is an independent and identically distributed (i.i.d.) sample from some underlying data distribution, we expect the optimal code table for this database to compress an arbitrary transaction sampled from this distribution well. We make this intuition more formal in Lemma 4.

We say that the itemsets in CT are *independent* if any co-occurrence of two itemsets $X, Y \in CT$ in the cover of a transaction is independent. That is, $P(XY) = P(X)P(Y)$. Clearly, this is a Naïve Bayes [117] like assumption.

Lemma 4. *Let $\mathcal{D}$ be a bag of transactions over $\mathcal{I}$,* cover *a cover function, CT the optimal code table for $\mathcal{D}$ and t an arbitrary transaction over $\mathcal{I}$. Then, if the itemsets $X \in cover(CT, t)$ are independent,*

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

Proof.

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

□

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

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

Lemma 5. *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two bags of transactions over $\mathcal{I}$, sampled from two different distributions,* cover *a cover function, and t an arbitrary transaction over $\mathcal{I}$. Let CT_1 and CT_2 be the optimal code tables for resp. $\mathcal{D}_1$ and $\mathcal{D}_2$. Then, from Lemma 4 it follows that*

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

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

The Krimp Classifier

The previous subsection, with Lemma 5 in particular, suggests a straightforward classification algorithm based on Krimp code tables. This provides

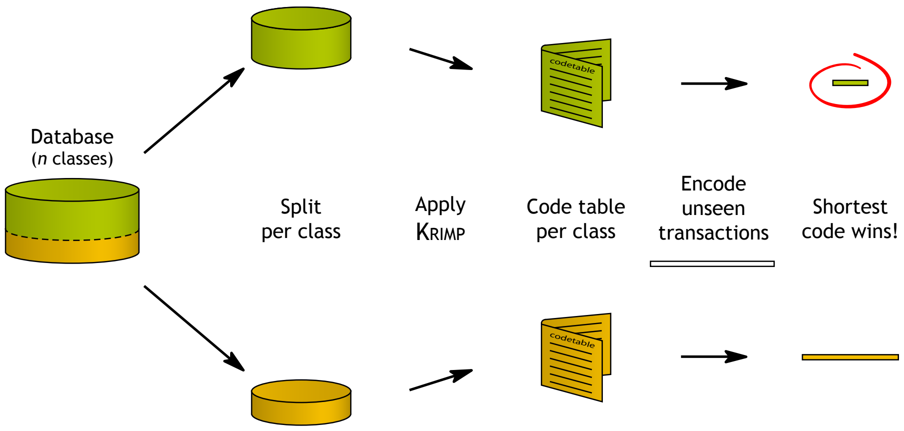


Figure 2.2: The KRIMP Classifier in action.

an independent way to assess the quality of the resulting code tables. The KRIMP Classifier is given in Algorithm 5. The KRIMP classification process is illustrated in Figure 2.2.

The classifier consists of a code table per class. To build it, a database with class labels is needed. This database is split according to class, after which the class labels are removed from all transactions. KRIMP is applied to each of the single-class databases, resulting in a code table per class. At the very end, after all pruning has been done, each code table is Laplace corrected: the usage of each itemset in CT_k is increased by one.

This ensures that all itemsets in CT_k have non-zero usage, therefore have

Algorithm 5 KRIMP Classifier

```
KRIMPCLASSIFIER($\mathcal{D}, t$) :
1. $K \leftarrow \{$ class labels of $\mathcal{D}$ $\}$
2. $\{\mathcal{D}_k\} \leftarrow$ split $\mathcal{D}$ on $K$, remove each $k \in K$ from each $t \in \mathcal{D}$
3. for all $\mathcal{D}_k$ do
4.    $\mathcal{F}_k \leftarrow$ MineCandidates($\mathcal{D}_k$)
5.    $CT_k \leftarrow$ KRIMP($\mathcal{D}_k, \mathcal{F}_k$)
6.    for each $X \in CT_k$ : $usage_{CT_k}(X) \leftarrow usage_{CT_k}(X) + 1$
7. end for
8. return $\arg\min_{k \in K} L(t \mid CT_k)$
```

a code, i.e. their code length can be calculated, and thus, that any arbitrary transaction $t \subseteq \mathcal{I}$ can be encoded. (Recall that we require a code table to always contain all singleton itemsets.)

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

2.6 Related Work

MDL in Data Mining

MDL was introduced by Rissanen [97] as a noise-robust model selection technique. In the limit, refined MDL is asymptotically the same as the Bayes Information Criterion (BIC), but the two may differ (strongly) on finite data samples [49]. We are not the first to use MDL, nor are we the first to use MDL in data mining or machine learning. Many, if not all, data mining problems can be related to Kolmogorov Complexity, which means they can be practically solved through compression [40], e.g. clustering (unsupervised learning), classification (supervised learning), and distance measurement. Other examples include feature selection [93], defining a parameter-free distance measure on sequential data [57, 58], discovering communities in matrices [26], and evolving graphs [105].

Pattern Selection

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

To address this problem, closed [91] and non-derivable [24] itemsets have been proposed, which both provide a concise lossless representation of the original itemsets. However, these methods deteriorate even under small amounts of noise. Similar, but providing a partial (i.e. lossy) representation, are maximal itemsets [14] and δ-free sets [35]. Along these lines, Yan *et al.* [121] proposed a method that selects k representative patterns that together summarize the frequent pattern set.

Recently, the approach of finding small subsets of informative patterns that describe the database has attracted a significant amount of research [21, 60, 85]. First, there are the methods that provide a lossy description of the data. These strive to describe just part of the data, and as such may overlook important interactions. Summarization as proposed by Chandola & Kumar [27] is a compression-based approach that identifies a group of itemsets such that each transaction is summarized by one itemset with as little loss of information as possible. Wang & Karypis [115] find summary sets, sets of itemsets that contain the largest frequent itemset covering each transaction.

Pattern Teams [61] are groups of most-informative length-k itemsets [60]. These are exhaustively selected through an external criterion, e.g. joint entropy or classification accuracy. As this approach is computationally intensive, the number of team members is typically < 10. Bringmann & Zimmerman [21] proposed a similar selection method that can consider larger pattern sets. However, it also requires the user to choose a quality measure to which the pattern set has to be optimized, unlike our parameter-free and lossless method.

Second, in the category of lossless data description, we recently [101] introduced the MDL-based Krimp algorithm. In this chapter we extend the theory, tune the pruning techniques and thoroughly verify the validity of the chosen heuristics, as well as provide extensive experimental evaluation of the quality of the returned code tables.

Tiling [42] is closely related to our approach. A tiling is a cover of the database by a group of (overlapping) item sets. Itemsets with maximal uncovered area are selected, i.e. as few as possible itemsets cover the data. Unlike our approach, model complexity is not explicitly taken into account. Another major difference in the outcome is that Krimp selects more specific (longer) itemsets. Xiang *et al.* [120] proposed a slight reformulation of Tiling that allows tiles to also cover transactions in which not all its items are present.

Two approaches inspired by Krimp are Pack [106] and LESS [53]. Both approaches consider the data 0/1 symmetric, unlike here, where we only regard items that are present (1s). LESS employs a generalised Krimp encoding to select only tens of low-entropy sets [52] as lossless data descriptions, but attains worse compression ratios than Krimp. Pack does provide a significant improvement in that regard. It employs decision trees to succinctly transmit individual attributes, and these models can be built from data or candidate sets. Typically, Pack selects many more itemsets than Krimp.

Our approach seems related to the set cover problem [25], as both try to cover the data with sets. Although NP-complete, fast approximation algorithms exist for set cover. These are not applicable for our setup though, as in set cover the complexity of the model is not taken into account. Another difference is that we do not allow overlap between itemsets. As optimal compression is the goal, it makes intuitively sense that overlapping elements may lead to shorter encodings, but it is not immediately clear how to achieve this in a fast heuristic.

Classification

A lot of classification algorithms have been proposed, many of which fall into either the class of rule-induction-based or that of association-rule-based methods. Because we use classification as a quality measure for the patterns that Krimp picks, we will compare our results with those obtained by some of the best existing classifiers. Such comparison can be done with rule-induction-based methods such as C4.5 [94], FOIL [95] and CPAR [122]. Mehta *et al.* [82] use MDL to prune decision trees for classification. However, we are more interested in the comparison to association-rule-based algorithms like iCAEP [123], HARMONY [114], CBA [75] and LB [83] as these also employ a collection of itemsets for classification. Because we argued that our method is strongly linked to the principle of Naïve Bayes (NB) [39] it's imperative we compare to it. Finally, we compare to Support Vector Machines (SVMs) [100], since it is generally accepted these perform well in many cases. Because, opposed to Krimp, these methods were devised with the goal of classification in mind, we would expect them to (slightly) outperform the Krimp classifier.

2.7 Experiments

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

We first describe our setup and the datasets we use in the experiments. Then, we start our evaluation of Krimp by looking at how many itemsets are selected and what compression ratios are attained. The stability of these results, and whether these rely on specific itemsets is explored. We then test whether the code tables model relevant structure through swap-randomisation. The quality of the code tables is independently validated through classification. At the end of the section, we evaluate the cover and candidate order heuristics of Krimp.

Setup

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

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

wherever $\mathcal{D}$ is clear from context. As candidates, $\mathcal{F}$, we typically use all frequent itemsets mined at *minsup* $= 1$, unless indicated otherwise. We use the AFOPT miner [76], taken from the FIMI repository [45], to mine (closed) frequent itemsets. The reported Krimp timings are of the selection process only and do not include the mining and sorting of the candidate itemset collections. All experiments were conducted on quad-core Xeon 3.0 GHz systems (in black casing) running Windows Server 2003. Reported timings are based on four-threaded runs, again, unless stated otherwise.

Data

For the experimental validation of our methods we use a wide range of freely available datasets. From the LUCS/KDD data set repository [32] we take some of the largest databases. We transformed the *Connect-4* dataset to a slightly less dense format by removing all 'empty-field' items. From the FIMI repository [45] we use the BMS[4] datasets [62]. Further, we use the *Mammals* presence and *DNA Amplification* databases. The former consists of presence records of European mammals[5] within geographical areas of 50×50 kilometers [86]. The latter is data on DNA copy number amplifications. Such copies activate oncogenes and are hallmarks of nearly all advanced tumors [89]. Amplified genes represent attractive targets for therapy, diagnostics and prognostics.

The details for these datasets are depicted in Table 2.3. For each database we show the number of attributes, the number of transactions and the density: the percentage of 'present' attributes. Last, we provide the total compressed size in bits as encoded by the singleton-only standard code tables ST.

Selection

We first evaluate the question whether Krimp provides an answer to the pattern explosion. To this end, we ran Krimp on 27 datasets, and analysed the outcome code tables, with and without post-acceptance pruning. The results of these experiments are shown as Table 2.4. As candidates itemset collections we mined frequent itemsets of the indicated *minsup* thresholds. These were chosen as low as possible, either storage-wise or computationally feasible.

The main result shown in the table is the reduction attained by the selection process: up to 7 orders of magnitude. While the candidate sets contain millions up to billions of itemsets, the resulting code tables typically contain hundreds to thousands of non-singleton itemsets. These selected itemsets compress the data well, typically requiring only a quarter to half of the bits of the independent ST encoding. Dense datasets are compressed very well. For *Adult* and *Mushroom*, ratios of resp. 24% and 21% are noted. Sparse data, on the other hand, typically contains little structure. We see that such datasets (e.g. the *Retail*

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

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

Table 2.3: Statistics of the datasets used in the experiments.

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

Per dataset the number of transactions, the number of attributes, the density (percentage of 1's) and the number of bits required by Krimp to compress the data using the singleton-only standard code table ST.

and *BMS* datasets) indeed prove difficult to compress; relatively many itemsets are required to provide a meagre compression.

Comparing between Krimp with and without post-acceptance pruning, we see that enabling pruning provides the best: fewer itemsets ($\sim$ 1000, on average) are returned, which provide better compression (avg. 2% improvement). For *Accidents* and *BMS-pos* the difference in the number of selected itemsets is a factor of 10. The average length of the itemsets in the code tables is about the same, with resp. 5.9 and 5.7 with and without pruning. However, the average usage of these itemsets differs more, with averages of resp. 80.7 and 48.2.

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

Next, we examine the development in number of selected itemsets w.r.t. the number of candidate itemsets. For the *Mushroom* database, the top graph of Figure 2.3 shows the size of the candidate set and size of the corresponding code table for varying *minsup* thresholds. While we see that the number of candidate itemsets grows exponentially, to 5.5 billion for $minsup = 1$, the number of selected itemsets stabilises at around 400. This stabilisation is typical for all datasets, with the actual number being dependent on the characteristics of the data.

This raises the question whether the total compressed size also stabilises. In the bottom graph of Figure 2.3, we plot the total compressed size of the database for the same range of *minsup*. From the graph it is clear that this is not the case: the compressed size decreases continuously, it does not stabilise. Again, this is typical behaviour; especially for sparse data we have recorded steep descents at low *minsup* values. As the number of itemsets in the code table is stable, we thus know that itemsets are being replaced by better ones. Further, note that the compressed size of the code table is dwarfed by the compressed size of the database. This is especially the case for large datasets.

Back to the top graph of the figure, we see a linear correlation between the runtime and the number of candidate sets. The correlation factor depends heavily on the data characteristics; its density, the number of items and the number of transactions. For this experiment, we observed 200,000 to 400,000 candidates considered per second, the performance being limited by IO.

In the top graph of Figure 2.4 we provide an overview of the differences in the sizes of the candidate sets and code tables, and in the bottom graph the runtimes recorded for these experiments. Those experiments for which the runtime bars are missing finished within one second. The bottom graph shows that the runtimes are low. Letting Krimp consider the largest candidate sets, billions of itemsets, takes up to a couple of hours. The actual speed (candidates per second) mainly depends on the support of the itemsets (the number of transactions that have to be covered). The speed at which our current implementation considers candidate itemsets typically ramps up to thousands, even hundreds of thousands, per second.

Stability

Here, we verify the stability of Krimp w.r.t. different candidate sets. First we investigate whether good results can be attained without the itemsets normally chosen in the code table. Given the large redundancy in the frequent pattern set, one would expect so.

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

As a code table typically consists of about 100 to 1000 elements, a considerable set of experiments is required per database. Therefore, we use five of the datasets. The results of these experiments are presented in Table 2.5. The minute differences in compression ratios show that the performance of Krimp does not rely on just those specific itemsets in the original code tables. Excluding all elements from the original code table results in compression ratios up till only .5% worse than normal. This is expected, as in this setting all structure in the data captured by the original code table has to be described differently. The results of the individual itemset exclusion experiments, on the other hand, are virtually equal to the original. In fact, sometimes shorter (better) descriptions are found this way: the removed itemsets were in the way of ones that offer a better description.

Next, we consider excluding far more itemsets as candidates. That is, we use closed, as opposed to all, frequent itemsets as candidates for Krimp. For datasets with little noise the closed frequent pattern set can be much smaller and faster to mine and process. For the *minsup* thresholds depicted in Table 2.4, we mined both the complete and closed frequent itemset collections, and used these as candidate sets $\mathcal{F}$ for running Krimp. Due to crashes of the closed frequent itemset miner, we have no results for the *Chess (k–k)* dataset. Figure 2.5 shows the comparison between the results of either candidate set in terms of the relative compression ratio. The differences in performance are slight, with an average increase in $L\%$ of about 1%. The only exception seems *Ionosphere*, where a 12% difference is noted. Further, the resulting code tables are of approximately the same size; the 'closed' code tables consist of fewer itemsets: 10 on average. From these experiments we can conclude that closed frequent itemsets are a good alternative to be used as input for Krimp, especially if otherwise too many frequent itemsets are mined.

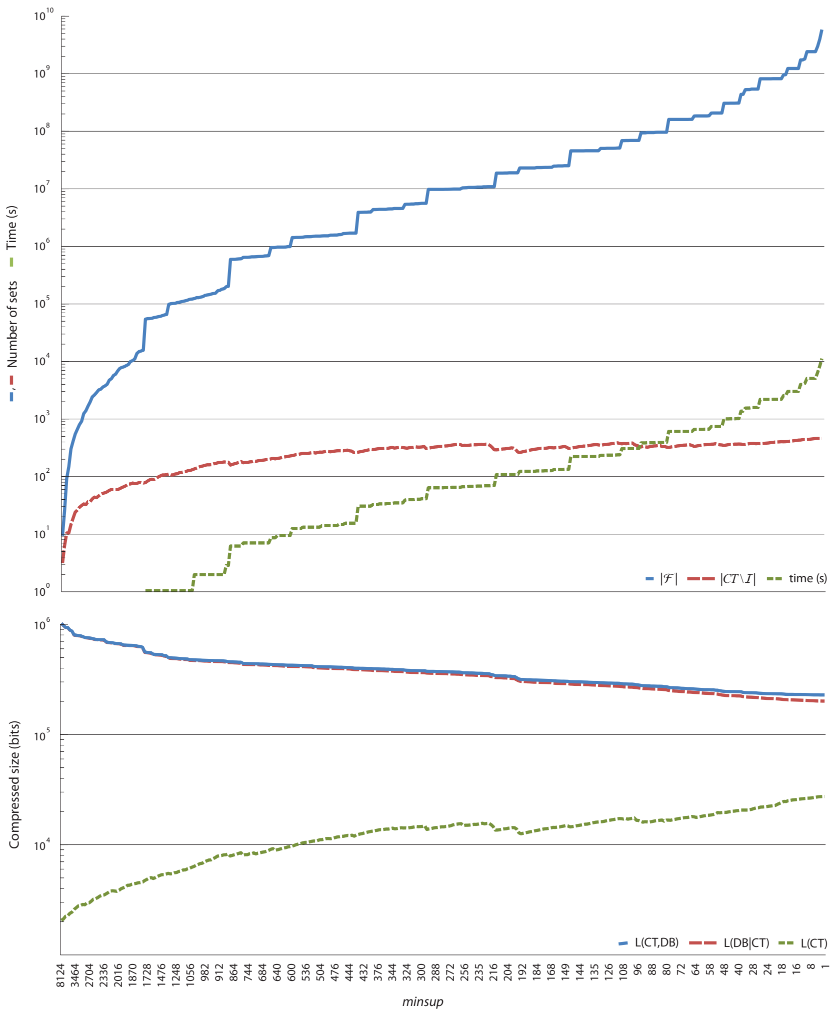


Figure 2.3: Running Krimp with post-acceptance pruning on the *Mushroom* dataset using 5.5 billion frequent itemsets as candidates ($minsup = 1$). (top) Per *minsup*, the size of the candidate set, $|\mathcal{F}|$, the size of the code table, $|CT \setminus \mathcal{I}|$, and the runtime in seconds. (bottom) Per *minsup*, the size in bits of the code table, the data, and the total compressed size (resp. $L(CT)$, $L(\mathcal{D} \mid CT)$ and $L(\mathcal{D}, CT)$).

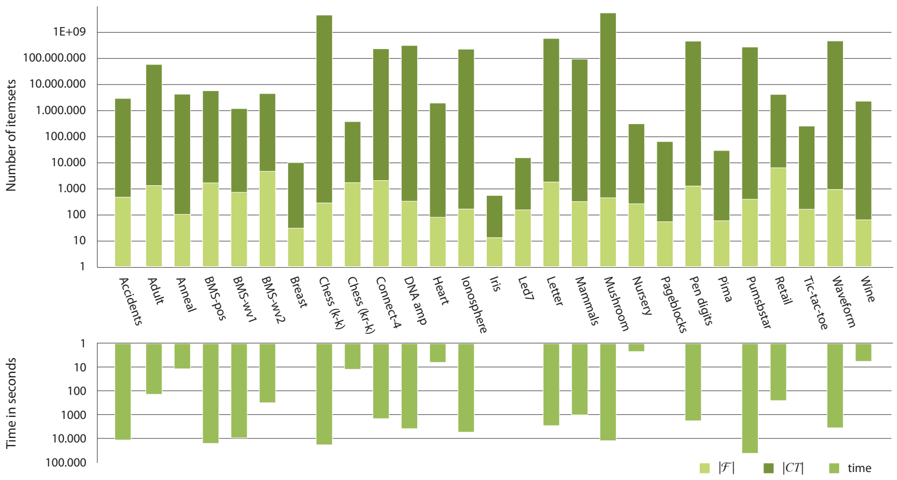


Figure 2.4: The results of Krimp with post-acceptance pruning on the 27 datasets using the *minsup* thresholds of Table 2.4. The dark coloured bars show the number of itemsets in the candidate set, $|\mathcal{F}|$, the lighter coloured bars the number of non-singleton itemsets selected in the code table, $|CT \setminus \mathcal{I}|$, and the bottom graph the associated runtimes.

Relevance

To evaluate whether Krimp code tables model relevant structure, we employ swap randomisation [44]. Swap randomisation is the process of randomising data to obscure the internal dependencies, while preserving the row and column margins of the data. The idea is that if the true structure of the data is captured, there should be significant differences between the models found in the original and randomised datasets.

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

Figure 2.6 shows the histogram of the total compressed sizes of these 1000

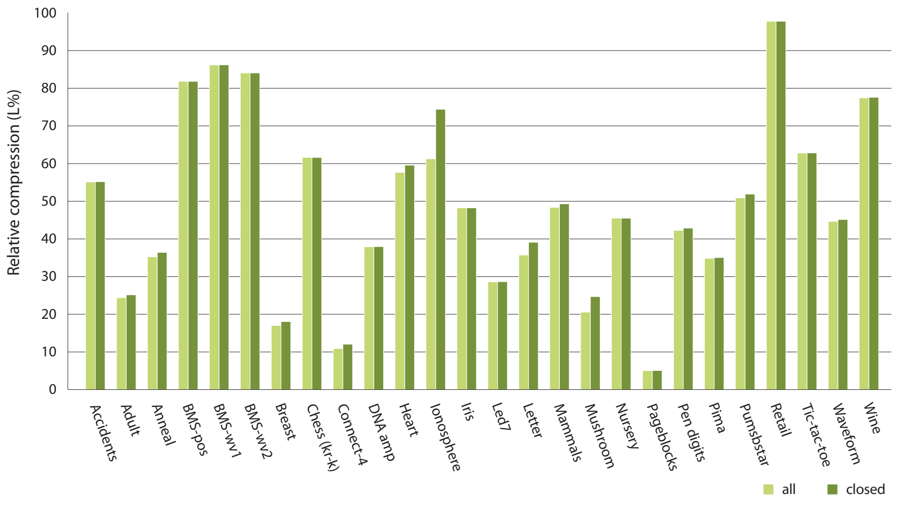


Figure 2.5: The results of Krimp with post-acceptance pruning on 26 datasets using the *minsup* thresholds of Table 2.4. Per dataset, the upward bars indicated the relative total compressed size ($L\%$). As candidates, $\mathcal{F}$, all (left bars), and closed (right bars) frequent itemsets were used.

randomisations. The total compressed sizes of the original databases are indicated by arrows. The standard encoded sizes for these three databases, $L(\mathcal{D}, ST)$, are 1173962 bits, 569042 bits and 14100 bits, respectively. The graphs show that the original data can be compressed significantly better than the randomised datasets (p-value of 0). Further quantitative results are shown in Table 2.6. Besides much better compression, we see that for *Nursery* and *Wine* the code tables induced on the original data contain fewer, but more specific (i.e. longer) itemsets. For *BMS-wv1* the randomised data is virtually incompressible with Krimp ($L\% \approx 98\%$), and as such much fewer itemsets are selected.

Classification

As an independent evaluation of the quality of the itemsets picked by Krimp, we compare the performance of the Krimp classifier to the performance of a wide range of well-known classifiers. Our hypothesis is that, if the code table-based classifier performs on-par, Krimp selects itemsets that are characteristic for the data.

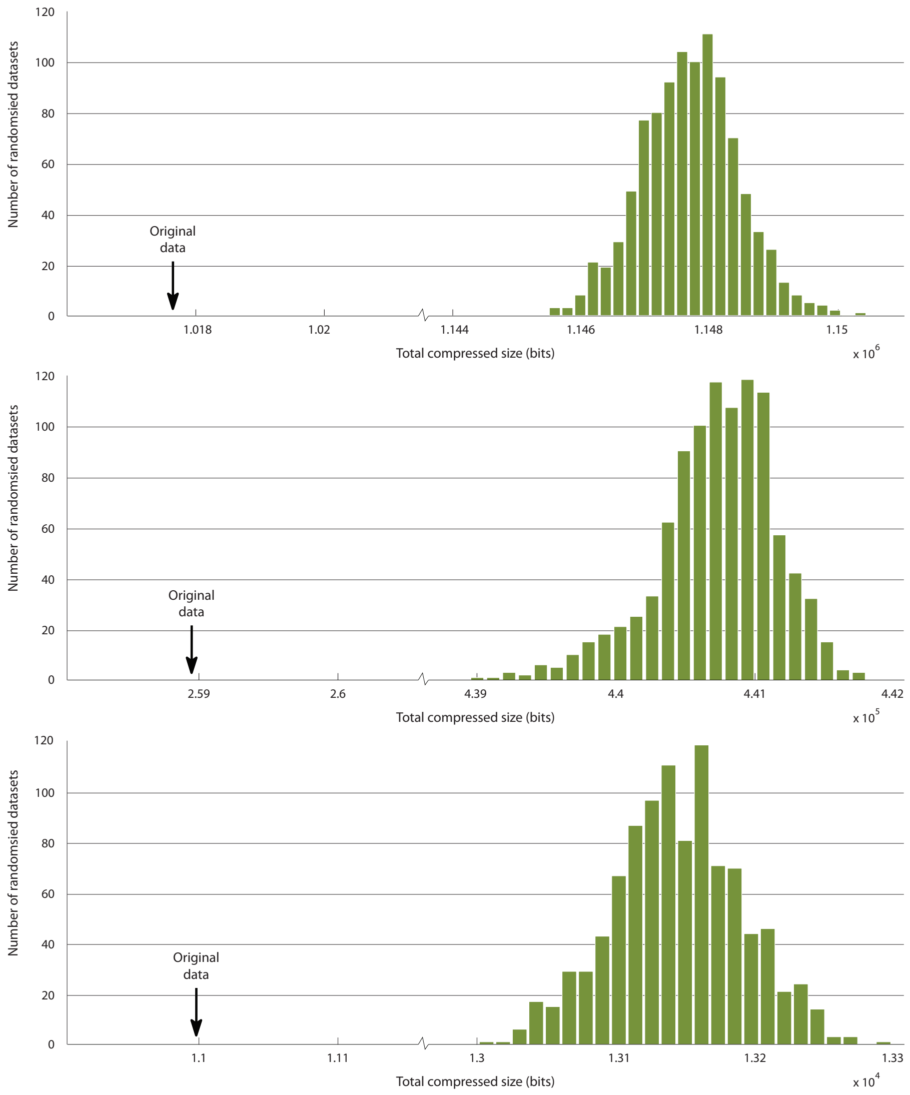


Figure 2.6: Histograms of the total compressed sizes of 1000 swap randomised *BMS-wv1* (top), *Nursery* (middle) and *Wine* (bottom) datasets, using all frequent itemsets as candidates for Krimp with post-acceptance pruning. Total compressed sizes of the original datasets are indicated by the arrows. Note the jumps in compressed size on the x-axes.

We use the same *minsup* thresholds as we used for the compression experiments, listed in Table 2.4. Although the databases are now split on class before they are compressed by Krimp, these thresholds still provide large numbers of candidate itemsets and result in code tables that compress well.

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

All results reported in this section have been obtained using 10-fold cross-validation. As performance measure we use accuracy, the percentage of true positives on the test data. We compare to results obtained with 6 state-of-the-art classifiers. These scores are taken from [19, 114, 123], which all used the same discretised datasets. Missing scores for C4.5, Naïve Bayes and SVM have been acquired using Weka [119]. Note that, in contrast to others, we use the sparser version of *Connect-4*, described before.

Before we compare the Krimp classification performance to other methods, we verify whether there is a qualitative difference between using all or only closed frequent itemsets as candidates and using post-acceptance pruning or not. Table 2.7 shows that the variation in average accuracy is very small, but using all frequent itemsets as candidates with pruning gives the highest accuracy, making it an obvious choice for inspection of more detailed results in the rest of this subsection.

Classification results with all frequent itemsets as candidates and post-acceptance pruning are presented in Table 2.8, together with accuracies obtained with 6 competitive classifiers. Baseline accuracy is the (relative) size of the largest class in a dataset. For about half of the datasets, absolute pairing gives the maximum score, relative pairing gives the best results for the other cases. As expected, relative pairing performs well especially for datasets with small classes or unbalanced class sizes. In general though, the difference in maximum accuracy between the two types of pairings is very small, i.e. $< 1\%$. For a few datasets, the difference is more notable, $2 - 10\%$.

Looking at the scores, we observe that performance on most datasets is very similar for most algorithms. For *Pima*, for example, all accuracies are within a very small range. Because of this, it is important to note that performance may vary up to a few percent depending on the (random) partitioning used for cross-validation, especially for datasets having smaller classes. Since we took the accuracies from different sources, we cannot conclude that a particular classifier is better on a certain dataset if the difference is not larger than $2-3\%$.

The Krimp classifier scores 6 wins, indicated in boldface, which is only beaten by C4.5 with 7 wins. Additionally, the achieved accuracy is close to the winning score in 5 cases, not so good for *Connect–4*, *Heart* and *Tic–tac–toe*, and average for the remaining 5 datasets.

Compared to Naïve Bayes, to which our method is closely related, we observe that the obtained scores are indeed quite similar. On average, however, Krimp achieves 2.4% higher accuracy, with 84.5% for Krimp and 82.1% for Naïve Bayes. The average accuracy for SVM is 83.8%, while C4.5 outperforms all with 86.3%. We also looked at the performance of FOIL, PRM and CPAR [95, 122] reported by Coenen [33]. These classifiers perform sub-par in comparison to those in Table 2.8. A comparison to LB and/or LB-chi2 [83] is problematic, as only few accuracies are available for the (large) datasets we use and for those that are available, the majority of the LB results is based on train/test, not 10-fold cross-validated.

To get more insight in the classification results, confusion matrices for 3 datasets are given in Table 2.9. The confusion matrix for *Heart* shows us why the Krimp classifier is unable to perform well on this dataset: it contains 4 very small classes. For such small databases, the size of the code table is dominant, precluding the discovery of the important frequent itemsets. This is obviously the case for some *Heart* classes. If we consider *Mushroom* and *Iris*, then the bigger the classes, the better the results. In other words, if the classes are big enough, the Krimp classifier performs very well.

We can zoom in even further to show how the classification principle works in practice. Take a look at the effect of coding a transaction with the 'wrong' code table, which is illustrated in Figure 2.7. The rounded boxes in this figure visualise the itemsets that make up the cover of the transaction. Each of the itemsets is linked to its code by the dashed line. The widths of the black and white encodings represent the actual computed code lengths. From this figure, it is clear that code tables can be used to both characterise and recognise data distributions.

Order

Next, we investigate the order heuristics of the Krimp algorithm. Both the standard cover order and standard candidate order are rationally made choices, but choices nevertheless. Here, we consider a number of alternatives for both and evaluate the quality of possible combinations through compression ratios

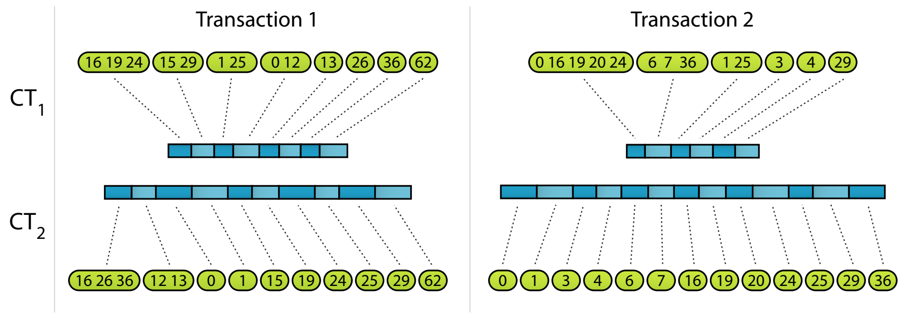


Figure 2.7: *Wine*; two transactions from class 1, $\mathcal{D}_1$, encoded by the code tables for class 1, CT_1 (top), and class 2, CT_2 (bottom).

and classification accuracies. The outcome of these experiments are shown in Table 2.10. Before we cover these results, we discuss the orders. As before, $\downarrow$ indicates the property to be sorted descending, and $\uparrow$ ascending.

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

– **Standard Cover Order**:

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

– **Entry**:

$$index \text{ of } X \text{ in } \mathcal{F} \downarrow$$

– **Area ascending**:

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

We also consider the **Random** cover order, where new itemsets are entered at a random position in the code table. For all the above orders, we sort the singleton itemsets below itemsets of longer length. In Table 2.10 we refer to these orders, respectively, as Standard, Entry, Area and Random. We further experimented with a number of alternatives of the above, but the measured performance did not warrant their inclusion in the overall comparison.

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

– **Standard Candidate Order**:

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

– **Standard, but length ascending**:

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

– **Length ascending, support descending**:

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

– **Area descending, support descending**:

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

A **Random** candidate order is also considered, which is simply a random permutation of the candidate set. In Table 2.10 we refer to these orders as, respectively, Standard, Standard′, Length, Area and Random. Again, we considered a number of variants of the above, for none of which the performance was good enough to be included here.

For all 20 combinations of the above cover and candidate orders we ran compression experiments on 16 datasets and classification experiments for 11 datasets. As candidate itemsets, we ran experiments with both the complete and closed frequent itemset collections. Due to the amount of experiments required by this evaluation we only used single samples for the random orders. Classification scores are 10-fold cross-validated, as usual. Per dataset, the score of the best performing pairing (absolute or relative) was chosen. The details on which datasets and what *minsup* thresholds were used can be found in Table 2.11.

The results of these experiments are depicted in Table 2.10. Shown are, per combination of orders, the total compression ratio $L\%$ and the classification accuracy, both of which are averaged over all databases and both all and closed frequent itemsets as candidate sets.

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

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

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

2.8 Discussion

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

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

The timings reported in this study show that compression is not only a good, but also a realistic approach, even for large databases and huge candidate collections; the single-threaded implementation already considers up to hundreds of thousands of itemsets per second. While highly efficient frequent itemset miners were used, we observed that mining the candidates sometimes takes (much) longer than the actual Krimp selection. Also, the algorithm can be easily parallelised, both in terms of covering parts of the database and of checking the candidate itemsets. The implementation[6] we used for the experiments in this chapter does the latter, as the performance of the former deteriorates rapidly for candidate itemsets with low support.

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

When mining all frequent itemsets for a low *minsup* is infeasible, using closed frequent itemsets as candidate set is a good alternative. For most datasets, results obtained with closed are almost as good as with all frequent itemsets, while for some datasets this makes the candidate set much smaller and thus computationally attractive.

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

Although code tables are made to just compress well, it turns out they can easily be used for classification. Because other classifiers have been designed with classification in mind, we expected these to outperform the Krimp classifier. We have shown this is not the case: Krimp performs on par with the best classifiers available. We draw two conclusions from this observation. Firstly,

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

Krimp selects itemsets that are very characteristic for the data. Secondly, the compression-based classification scheme works very well.

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

2.9 Conclusions

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

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

Also, we verified the heuristic choices we made for the Krimp algorithm. We extensively evaluated different possible orders for both the candidate set and code table. The outcome is that the standard orders are very good: no combination of orders was found that performs significantly better, while the standard orders offer good opportunities for optimisation.

Because we set the frequent pattern explosion, the original problem, in a wide context but discussed only frequent itemsets, the reader might wonder: does this also work for other types of patterns? The answer is affirmative, in Bathoorn *et al.* [13] we have shown that our MDL-based approach also works for pattern-types such as *frequent episodes* for sequence data and *frequent subgraphs* for graph data. In Koopman *et al.* [64,65], we extended the approach to multi-relational databases, i.e. to select patterns over multiple tables. Also, the LESS algorithm [53] (see also Section 2.6) introduces an extension of the encoding such that it can be used to select more generic patterns, e.g. low-entropy sets.

Like detailed in Faloutsos & Megalooikonomou [40], there are many data mining tasks for which compression, and thus the foundations presented in this chapter, can be used. We will show this in subsequent chapters.

Table 2.4: Results of KRIMP with and without post-acceptance pruning

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

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

Table 2.5: Stability of the Krimp given candidate sets with exclusions

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

Per dataset, the *minsup* threshold at which frequent itemsets were mined as candidates $\mathcal{F}$ for Krimp. Further, the relative compression $L\%$ for running Krimp with $\mathcal{F}$, the average relative compression attained by excluding single original code table elements from $\mathcal{F}$, and the relative compression attained by excluding all itemsets normally chosen from $\mathcal{F}$, i.e. using $\mathcal{F} \setminus \{X \in CT \mid X \notin \mathcal{I}\}$ as candidates for Krimp. For *Mushroom* and *Pen digits* the closed frequent itemset collections were used as candidates.

Table 2.6: Swap randomisation experiments

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

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

Table 2.7: Results of KRIMP classification, for all/closed frequent itemsets and without/with pruning

Candidates	w/o pruning	with pruning
all	84.3 ± 14.2	84.5 ± 13.1
closed	84.0 ± 13.9	83.7 ± 14.2

For each combination of candidates and pruning, the average accuracy (%) over the 19 datasets from Table 2.8 is given. Standard deviation is high as a result of the diverse range of datasets used.

Table 2.8: Results of Krimp classification, compared to 6 state-of-the-art classifiers

Dataset	*base*	Krimp	NB	C4.5	CBA	HRM	iCAEP	SVM
Adult	76.1	84.3	83.8	**85.4**	75.2	81.9	80.9	84.1
Anneal	76.2	96.6	97.0	90.4	**98.1**		95.1	97.4
Breast	65.5	94.1	97.1	95.4	95.3		**97.4**	69.6
Chess (k–k)	52.2	90.0	87.9	**99.5**	98.1		94.6	95.4
Chess (kr–k)	16.2	**57.9**	36.1	56.6		44.9		29.8
Connect–4	65.8	69.4	72.1	**81.0**		68.1	69.9	72.5
Heart	54.1	61.7	83.5	78.4	81.9		80.3	**84.2**
Ionosphere	64.1	91.0	89.5	92.0	**92.1**		90.6	88.6
Iris	33.3	**96.0**	93.3	94.7	92.9	94.7	93.3	**96.0**
Led7	11.0	**75.3**	74.0	71.5	71.9	74.6		73.8
Letter	4.1	70.9	64.1	**87.9**		76.8		67.8
Mushroom	51.8	**100**	99.7	**100**		99.9	99.8	99.7
Nursery	33.3	92.3	90.3	**97.1**	88.8	92.8	84.7	91.4
Page blocks	89.8	**92.6**	90.9	92.4	89.0	91.6		91.2
Pen digits	10.4	95.0	85.8	**96.6**		96.2		93.2
Pima	65.1	72.7	74.7	72.5	73.1	73.0	72.3	**77.3**
Tic–tac–toe	65.3	88.7	70.2	86.3	**100**	81.0	92.1	98.3
Waveform	33.9	77.1	80.8	70.4	75.3	80.5	81.7	**83.2**
Wine	39.9	**100**	89.9	87.9	91.6	63.0	98.9	98.3

For each dataset, baseline accuracy and accuracy (%) obtained with Krimp classification is given, as well as accuracies obtained with 6 other classifiers are given. We use HRM as abbreviation for HARMONY. The highest accuracy per dataset is displayed in boldface. For Krimp with post-acceptance pruning, per class, frequent itemsets mined at thresholds found in Table 2.4 were used as candidates. All results are 10-fold cross-validated.

Table 2.9: Confusion matrices

Mushroom

	1	2
1	4208	0
2	0	3916

Iris

	1	2	3
1	47	2	0
2	2	48	1
3	1	0	40

Heart

	1	2	3	4	5
1	142	22	9	6	2
2	17	23	8	9	5
3	3	2	12	4	2
4	0	7	5	10	4
5	2	1	2	6	0

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

Table 2.10: Evaluation of candidate and cover orders

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

Results for 20 combinations of candidate and cover orders for Krimp with post-acceptance pruning. Shown are average relative Krimp compression, $L\%$, and average classification accuracy (%) on a number of datasets. Results for compression and classification are averaged over 16 resp. 11 datasets. Table 2.11 shows which datasets were used and at what *minsup* thresholds the candidate sets, $\mathcal{F}$, were mined for these experiments.

Table 2.11: Datasets and settings used for the candidate and cover order experiments

	minsup		Used for	
Dataset	all	closed	Compression	Classification
Adult	20	1	✓	✓
Anneal	1	1	✓	✓
Breast	1	1	✓	✓
Chess (kr–k)	1	1	✓	
DNA amplification	10	1	✓	
Ionosphere	50	1	✓	✓
Iris	1	1	✓	✓
Led7	1	1	✓	✓
Letter recognition	50	20	✓	
Mammals	545	545	✓	
Mushroom		1	✓	✓
Nursery	1	1	✓	
Pen digits	20	1	✓	✓
Pima	1	1	✓	✓
Waveform	50	5	✓	✓
Wine	1	1	✓	✓

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

Chapter 3

Characterising the Difference

Characterising the differences[1] between two databases is an often occurring problem in data mining. Detection of change over time is a prime example, comparing databases from two branches is another one. The key problem is to discover the patterns that describe the difference. Emerging patterns provide only a partial answer to this question.

In the previous chapter, we showed that the data distribution can be captured in a pattern-based model using compression. Here, we extend this approach to define a generic dissimilarity measure on databases. Moreover, we show that this approach can identify those patterns that characterise the differences between two distributions. Experimental results show that our method provides a well-founded way to independently measure database dissimilarity that allows for thorough inspection of the actual differences. This illustrates the use of our approach in real world data mining.

[1]This chapter has been published as [110]: J. Vreeken, M. van Leeuwen & A. Siebes. Characterising the Difference. In *Proceedings of the KDD'07* (2007).

3.1 Introduction

Comparing databases to find and explain differences is a frequent task in many organisations. The two databases can, e.g. be from different branches of the same organisations, such as sales records from different stores of a chain or the “same” database at different points in time. In the first case, the goal of the analysis could be to understand why one store has a much higher turnover than the other. In the second case, the goal of the analysis could be to detect changes or drift over time.

The problem of this kind of “difference detection” has received ample attention, both in the database and in the data mining community. In the database community, OLAP [31] is the prime example. Using roll-up and drill-down operations, a user can (manually) investigate, e.g. the difference in sales between the two stores. Emerging pattern mining [37] is a good example from the data mining community. It discovers those patterns whose support increase significantly from one database to the other.

Emerging patterns, though, are often redundant, giving many similar patterns. Also, the growth rate that determines the minimal increase in support has a large impact on the number of resulting patterns. Lower growth rates give large amounts of patterns, of which only some are useful. To discover only “interesting” differences would require the data miner to test with multiple growth rate settings and, manually, trace what setting gives the most useful results and filter those from the complete set of emerging patterns.

In this chapter we propose a new approach to ‘difference detection’ that identifies those patterns that characterise the differences between two databases. In fact, the approach just as easily identifies the characteristic differences between multiple databases. As in the previous chapter, we restrict ourselves to frequent itemset mining, although the methodology easily extends to other kinds of patterns and data types, as mentioned in Section 2.9.

In Chapter 2 we attacked the well-known frequent itemset explosion at low support thresholds using MDL. Also, we introduced a simple classification algorithm which scores on-par with state-of-the-art classification algorithms.

The approach towards difference detection introduced in this chapter is again based on compression. First, we use compression to define a dissimilarity measure on databases. Then we introduce three ways to characterise the differences between two (dis)similar databases.

Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be the two databases, with transactions concerning the same sets of items, of which we need to analyse the differences. In Section 3.3, we first consider the difference in compressed length for the transactions in $\mathcal{D}_1$ when compressed by the MDL-compression schemes. The MDL-principle as well as our results in classification imply that the compression scheme induced from $\mathcal{D}_2$ should in general do worse than the scheme induced from $\mathcal{D}_1$. This is verified by some simple experiments.

Next, we aggregate these differences per transaction by summing over all transactions in $\mathcal{D}_1$ and normalising this sum by the optimal code length for $\mathcal{D}_1$. This aggregation measures how different a database is from $\mathcal{D}_1$. This is verified by experiments that show the correlation between this similarity measure and the confusion matrix of our classification algorithm briefly introduced above and in Section 3.2. Finally, this simple measure is turned into a dissimilarity measure for any pair of databases by taking the maximum of how different $\mathcal{D}_1$ is from $\mathcal{D}_2$ and vice versa. Again, the MDL-principle implies that this is a dissimilarity measure. Experiments verify this claim by showing the correlation between this dissimilarity measure and the accuracy of our classification algorithm.

The result of Section 3.3 is a dissimilarity measure for a pair of databases, based on code tables. If the dissimilarity is small, the two databases are more or less the same and a further analysis of the differences will not show anything interesting. The topic of Section 3.4 is on how to proceed if the dissimilarity is large. In that section, we introduce three ways to characterise these differences. The first approach focuses on the usage-patterns of the code table elements, while the second focuses on how (sets of) transactions are compressed by the two different schemes. The third and last approach focuses on differences in the code tables themselves. All three approaches highlight complementary, characteristic, differences between the two databases.

In Section 3.5 we discuss related work and describe the differences with our work. We round up with conclusions in Section 3.6.

3.2 Preliminaries

In this chapter, we build upon the Krimp algorithm and Krimp Classifier we introduced in the previous chapter.

For the sake of readability, we will use some notational shortcuts in the sections that follow:

$$\begin{aligned} CT_i(\mathcal{D}_j) &= L(\mathcal{D}_j \mid CT_i) \\ CT_i(t) &= L(t \mid CT_i) \end{aligned}$$

During the classification experiments reported in the previous chapter, we made some interesting observations in the distributions of the code lengths. Figure 3.1 shows the encoded lengths for transactions of a single class, encoded by code tables constructed for each of the three classes. Not only gives the code table constructed for these transactions shorter encodings, the standard deviation is also much smaller (compare the histogram on the left to the other two). This means that a better fit of the code table to the distribution of the compressed data results in a smaller standard deviation.

Experimental Setup

Although a lot of time series data is being gathered for analysis, no good benchmark datasets with this type of data currently exist. We therefore decided to use a selection from the UCI repository [32], which has been commonly used for emerging patterns [37] and related topics before.

As these are all datasets containing multiple classes, we look at the differences between classes. Hence, we split each dataset on class label C and remove this label from each transaction, resulting in a database $\mathcal{D}_i$ per class C_i. A code table induced from $\mathcal{D}_i$ using Krimp is written as CT_i.

For many steps in Sections 3.3 and 3.4, we show results obtained with the datasets *Heart* and *Wine* because of their properties: they are interesting because they consist of more than 2 classes, but don't have too many classes. Please note this selection is only for the purpose of presentation; results we obtained with other (larger) datasets are similar. In fact, Krimp is better at approximating data distributions of larger databases, providing even more reliable results.

Characteristics of all datasets used are summarised in Table 3.8, together with the minimum support levels we use for mining the frequent itemsets that function as candidates for Krimp. All experiments in this chapter are done with all frequent itemsets and Krimp with pruning.

3.3 Database Dissimilarity

In this section, we introduce a dissimilarity measure for transaction databases. This measure indicates whether or not it is worthwhile to analyse the differences between two such databases. If the dissimilarity is low, the differences between the two databases are small. If the measure is high, it is worthwhile to investigate the differences. Rather than defining the similarity measure upfront followed by a discussion and illustration of its properties, we "develop" the measure in a few steps as that allows us to discuss the intuition that underlies the definition far easier.

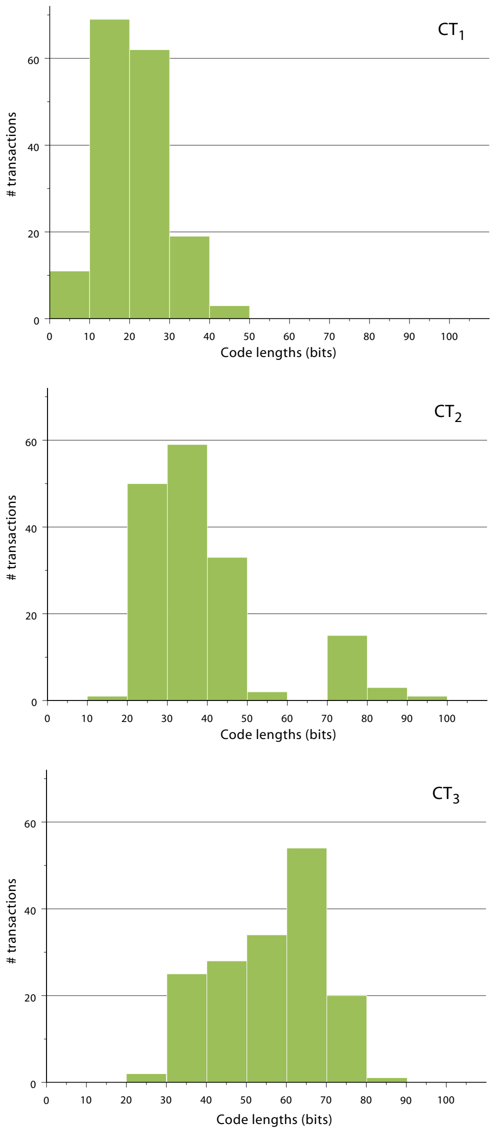


Figure 3.1: *Heart*; encoded transaction lengths for all transactions belonging to one class ($\mathcal{D}_1$), encoded with the code tables constructed for each of the three classes (top to bottom: CT_1, CT_2, CT_3).

Differences in Code Lengths

The MDL principle implies that the optimal compressor induced from a database $\mathcal{D}_1$ will generally provide shorter encodings for its transactions than the optimal compressor induced from another database $\mathcal{D}_2$. Our earlier experiments on classification verify that this is also true for the code table compressors Krimp discovers heuristically; see Section 3.2.

More in particular, denote by H_i the optimal compressor induced from database $\mathcal{D}_i$ and let t be a transaction in $\mathcal{D}_1$. Then, the MDL principle implies that:

$$|H_1(t) - H_2(t)|$$

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

In fact the MDL principle implies that if the code length differences are large (the second case), then on average the smallest code length will be $H_1(t)$. Our classification results suggest that something similar should hold for the code table compressors discovered by Krimp. In other words, we expect that

$$CT_2(t) - CT_1(t)$$

measures how characteristic t is for $\mathcal{D}_1$. That is, we expect that this difference is most often positive and large for those transactions that are characteristic for $\mathcal{D}_1$.

In Figures 3.2 and 3.3, code length differences are shown for two datasets, respectively for transactions of the $Wine_1$ and $Heart_1$ databases. As we expected, virtually all code length differences are positive. This means that in practice the native code table does indeed provide the shortest encoding.

In the case of the $Wine_1$ database depicted in Figure 3.2, we see a whopping average difference of 45bits per transaction. The shapes of the two histograms also show a nice clustering of the differences between the encoded lengths. No negative differences occur, each single transaction is compressed better by its native code table. This confirms that MDL creates code tables that are truly specific for the data.

We see the same general effect with $Heart_1$ in Figure 3.3, as again the peaks of the distribution lay within safe distance from the origin. From the histograms there is little doubt that code tables CT_2 and CT_3 are encoding data from a different distribution than they have been induced from. More importantly,

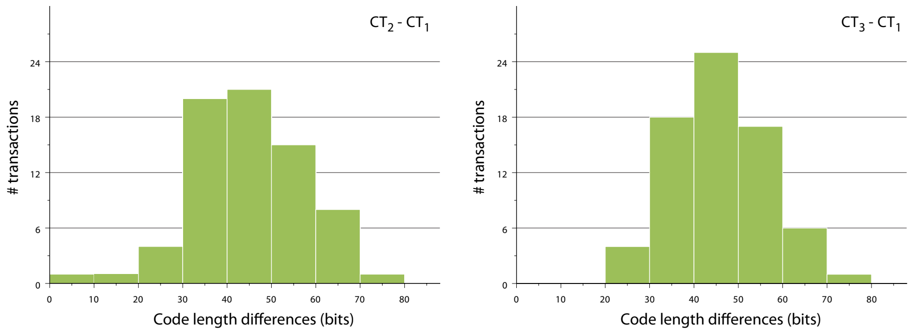


Figure 3.2: *Wine*; code length difference histograms for transactions in $\mathcal{D}_1$: encoded length differences between CT_2 and CT_1 (left) and between CT_3 and CT_1 (right).

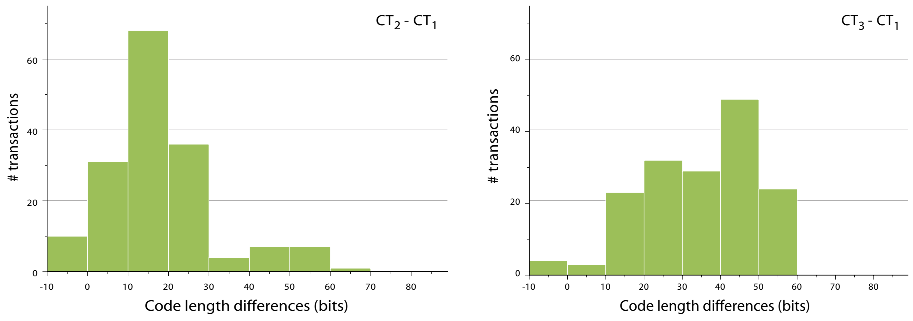


Figure 3.3: *Heart*; code length difference histograms for transactions in $\mathcal{D}_1$: encoded length differences between CT_2 and CT_1 (left) and between CT_3 and CT_1 (right).

comparing these diagrams unambiguously shows that it is possible to use the differences in encoded lengths to measure the amount of change between data. For example, as the differences on the left histogram are clearly smaller than in the situation on the right, this seems to imply that *Heart* classes 1 and 2 are more alike than classes 1 and 3. How to investigate this hypothesis further will be discussed in the next section. First we continue the development of our dissimilarity measure.

Aggregating Code Length Differences

In the previous subsection we have seen that the histograms of code length differences give good insight in the differences between two databases. The next logical step towards the definition of a dissimilarity measure is to aggregate these differences over the database. That is, to sum the individual code length differences over the complete database.

Straightforward aggregation, however, might give misleading results for two reasons:

- code length differences can be negative, so even if $\mathcal{D}_1$ and $\mathcal{D}_2$ are rather different, the aggregated total might be small.
- if $\mathcal{D}_1$ is a large database, the aggregated total might be large even if $\mathcal{D}_2$ is very similar to $\mathcal{D}_1$.

As already mentioned in the previous subsection, the MDL principle implies that for the MDL-optimal compressors H_1 and H_2, the expected average value of $H_2(t) - H_1(t)$ is positive. In other words, negative code length differences will be relatively rare and won't unduly influence the aggregated sum.

Our results in classification and, more importantly, the results of the previous subsection indicate that the same observation holds for the code table compressors CT_1 and CT_2 induced by Krimp. Clearly, only experiments can verify this claim.

The second problem indicated above is, however, already a problem for the MDL-optimal compressors H_1 and H_2. For, the expected value of the sum of the code length differences is simply the number of transactions times the expected average code length difference. Since the latter number is positive according to the MDL principle, the expected value of the sum depends linearly on the number of transactions on the database.

Clearly, the "native" encoded size of the database, $CT_1(\mathcal{D}_1)$, also depends on the size of the database. Therefore, we choose to counterbalance this problem by dividing the sum of code length differences by this size. Doing this, we end up with the *Aggregated Code Length Difference*, denoted by *ACLD* and

Table 3.1: *Heart*; aggregated code length differences.

	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$
CT_1	0.00	0.36	0.71	0.88	1.58
CT_2	0.85	0.00	0.60	0.65	1.03
CT_3	1.65	0.78	0.00	0.60	1.25
CT_4	1.85	0.65	0.61	0.00	1.09
CT_5	2.18	1.07	0.72	0.87	0.00

Table 3.2: *Wine*; aggregated code length differences.

	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$
CT_1	0.00	1.27	1.32
CT_2	1.13	0.00	1.73
CT_3	1.14	1.68	0.00

defined as

$$ACLD(\mathcal{D}_1, CT_2) = \frac{CT_2(\mathcal{D}_1) - CT_1(\mathcal{D}_1)}{CT_1(\mathcal{D}_1)}.$$

Note that $ACLD$ is an asymmetric measure: it measures how different $\mathcal{D}_2$ is from $\mathcal{D}_1$, not vice versa! While one would expect both to be in the same ballpark, this is by no means given. The asymmetry is further addressed in the next subsection. To clearly indicate the asymmetry, the parameters are asymmetric: the first parameter is a database, while the second is a code table.

Given this definition, we can now verify experimentally whether it works or not. That is, do greater dissimilarities imply larger differences and vice versa?

In Table 3.1 we read the aggregated code length differences for all possible combinations of code tables and class databases for the *Heart* dataset. It is immediately clear there are distinct differences between the class distributions, as measurements of 1.00 imply code lengths averaging twice as long as that of the actual class. We also notice that while the data distributions of databases 1 and 5 are quite distinct, the lower measurements between the other three classes indicate that their distributions are more alike.

For the *Wine* database the class distributions are even more adrift than those in the *Heart* database, for all cross-compressions result in encodings more than twice as long as the native ones. This is completely in line with what we have seen before in Figure 3.2, in which we showed there is no uncertainty in keeping transactions of the *Wine* databases apart based on encoded lengths.

Table 3.3: *Heart*: classification confusion matrix.

Classified	Class				
as	1	2	3	4	5
1	137	24	9	6	3
2	12	11	11	7	5
3	6	8	7	8	1
4	8	10	7	9	4
5	1	2	2	5	0

Table 3.4: *Wine*: classification confusion matrix.

Classified	Class		
as	1	2	3
1	65	3	6
2	5	55	0
3	1	1	42

If this technique truly quantifies the likeliness of the distributions belonging to some data, intuition tells us there has to be a close relation with the classification quality based on encoded transaction lengths. We can easily check this by comparing the aggregated code length differences with the confusion matrices for these databases. We therefore ran 10-fold cross validated classification experiments for these databases, as we did in the previous chapter.

The confusion matrix for the *Heart* database, in Table 3.3, clearly shows the intuition to be correct, as the number of misclassified instances drops completely according to ACLD. While 24 transactions of class 2 are misclassified as belonging to class 1, we see in Table 3.1 that these two classes are measured as rather similar. In fact, if we sort the measurements in Table 3.1 per class, we find the same order as when we sort Table 3.3 on the number of misclassifications. The measured difference thus directly relates to the ability to distinguish classes.

In Table 3.4 we see the same pattern with the *Wine* database as with the *Heart* database before: the lowest dissimilarities relate to the most misclassifications. We also observe that while analysis of individual code length differences, like Figure 3.2, suggests there should be no confusion in classification, a number of transactions are misclassified. These can be tracked back as being artefacts of the 10-fold cross-validation on a small database.

The Database Dissimilarity Measure

The experiments presented above verified that the aggregated differences of database encodings provide a reliable means to measure the similarity of one database to another. To make it into a true dissimilarity measure, we would like it to be symmetric. Since the measure should indicate whether or not we should investigate the differences between two databases, we do this by taking the maximum value of two Aggregated Code Length Differences:

$$\max\{ACLD(\mathcal{D}_1, CT_2), ACLD(\mathcal{D}_2, CT_1)\}.$$

This can easily be rewritten in terms of compressed database sizes, without using the ACLD function.

Definition 6. *For all databases x and y, define the* code table dissimilarity *measure DS between x and y as*

$$DS(x, y) = \max\left\{\frac{CT_y(\mathcal{D}_x) - CT_x(\mathcal{D}_x)}{CT_x(\mathcal{D}_x)}, \frac{CT_x(\mathcal{D}_y) - CT_y(\mathcal{D}_y)}{CT_y(\mathcal{D}_y)}\right\}.$$

The databases are deemed very similar (possibly identical) iff the score is 0, higher scores indicate higher levels of dissimilarity. Although at first glance this method comes close to being a distance metric for databases, this is not entirely the case. A distance metric D must be a function with non-negative real values defined on the Cartesian product $K \times K$ of a set K. Furthermore, it must obey the following requirements for every $k, l, m \in K$:

1. $D(k, l) = 0$ iff $k = l$ (identity)
2. $D(k, l) = D(l, k)$ (symmetry)
3. $D(k, l) + D(l, m) \geq D(k, m)$ (triangle inequality)

For the MDL optimal compressors, we can prove that DS will be positive. For our code table compressors, we can not. However, the experiments in the previous two subsections as well as those in this one indicate that DS is unlikely to be negative. As we can not even guarantee that DS is always positive, we can certainly not prove the identity axiom. The second axiom, the symmetry axiom holds, of course, by definition. For the triangle inequality axiom we again have no proof. However, in the experiments reported on this subsection the axioms hold. In other words, for all practical purposes our measure acts as a distance measure. However, to clearly indicate that our measure is not a proven distance metric we call it a dissimilarity measure.

Table 3.5: *Heart*: dissimilarity.

	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$
$\mathcal{D}_2$	0.85			
$\mathcal{D}_3$	1.65	0.78		
$\mathcal{D}_4$	1.85	0.65	0.61	
$\mathcal{D}_5$	2.18	1.07	1.25	1.09

Table 3.6: *Nursery*: dissimilarity.

	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$
$\mathcal{D}_2$	2.62			
$\mathcal{D}_3$	2.83	2.04		
$\mathcal{D}_4$	3.10	1.91	4.05	
$\mathcal{D}_5$	7.38	1.26	10.12	1.54

Table 3.7: *Wine*: dissimilarity.

	$\mathcal{D}_1$	$\mathcal{D}_2$
$\mathcal{D}_1$	1.27	
$\mathcal{D}_2$	1.32	1.73

The dissimilarity measurements for the *Heart*, *Nursery* and *Wine* database are given in respectively Tables 3.5, 3.6 and 3.7. One of the most striking observations is that many of the measurements are greater than 1.0, meaning that the cross-compressed databases are more than twice as large as the natively-compressed databases. The differences between the *Nursery*$_3$ and *Nursery*$_5$ datasets are such that a dissimilarity measurement of 10.12 is the result: a difference of a factor 11 of the average encoded length of a transaction.

In Table 3.8 a summary of datasets, their characteristics and dissimilarity results is given. For each dataset, the lowest and the highest observed dissimilarity is listed. A full results overview would obviously require too much space; datasets with many classes have squared as many database pairs of which the dissimilarity can be measured.

Overall, we see that the dissimilarities between the classes of the UCI datasets vary quite a bit. Some datasets seem to have very little difference

Table 3.8: Database characteristics.

			Krimp		DS	
Dataset	$\|\mathcal{D}\|$	*#classes*	*minsup*	*Acc. (%)*	*min*	*max*
Adult	48842	2	20	84.6	0.60	0.60
Chess (kr–k)	28056	18	10	58.0	0.29	2.69
Connect–4	67557	3	50	69.9	0.18	0.28
Heart	303	5	1	52.5	0.61	2.18
Iris	150	3	1	96.0	2.06	13.00
Led7	3200	10	1	75.3	1.27	11.29
Letter recognition	20000	26	50	68.1	0.43	2.83
Mushroom	8124	2	50	100	8.24	8.24
Nursery	12960	5	1	92.4	1.26	10.12
Pen digits	10992	10	20	88.6	1.33	4.43
Tic–tac–toe	958	2	1	87.1	0.62	0.62
Wine	178	3	1	97.7	1.27	1.73

Candidate *minsup* and class dissimilarity measurements for a range of UCI datasets. As candidates, all frequent itemsets were used up to the given minimum support level.

between classes (*Connect–4*, *Adult*, *Tic–tac–toe*), others contain rather large dissimilarity (*Mushroom*, *Iris*, *Led7*).

Another interesting comparison is between the dissimilarities and the classification results also reported in that table. There is a clear correlation between the two. The larger the dissimilarity, the better the classification results. This pattern is less clear for datasets containing small classes, which is caused by the fact that MDL doesn't work well for small data sets.

This observation is interesting because classification errors are made on individual transactions, whereas *DS* is an aggregated measure. In other words, the observation verifies that this aggregated measure reflects what happens at the level of individual transactions. This is exactly the property our dissimilarity measure should hold.

3.4 Characterising Differences

The first benefit of our dissimilarity measure is that it quantifies the difference between databases, the second advantage is the ability to characterise those differences.

There are three methods available for difference analysis, which zoom in to separate levels of difference between the distributions. First, we can compare the code table covers of the databases. This directly informs us which patterns that are important in one database are either over or under-expressed in another database. The second approach is to zoom in on how specific transactions are covered by the different code tables. This reveals in detail where differences are identified by the code tables. Thirdly, we can extract knowledge about the specific differences and similarities between the distributions from the code tables.

Comparing Database Covers

The most straightforward, but rather informative method for difference analysis is the direct comparison of database covers. Such evaluation immediately identifies which patterns are over and under-expressed, showing us the characteristics of the differences in structure between the two databases.

To run this analysis, we first use Krimp to obtain a code table for database $\mathcal{D}_2$ and use it to cover database $\mathcal{D}_1$. Because the itemsets and their usages in the code table capture the data distribution of database $\mathcal{D}_2$, the usages found by covering database $\mathcal{D}_1$ are expected to be different if the two databases are different.

Identification of these differences is done by finding those patterns in the code table that have a large shift in frequency between the two database covers. The same process can be applied vice versa for even better insight of the differences.

If the distribution is really different, we would expect to see a dramatic increase in use of the singletons caused by a decrease in use of the larger, more specific, sets. Slighter differences will lead to more specific shifts in patterns usage, with less of a shift towards singleton usage.

An example visualisation can be seen in Figure 3.4. A code table for *Wine* $\mathcal{D}_1$ has been constructed and used to cover all three databases. A quick glance shows that our hypothesis on the use of singletons is correct: $\mathcal{D}_1$ is covered by quite some sets of 2 or more items, but both $\mathcal{D}_2$ and $\mathcal{D}_3$ are covered largely by singletons.

Of special interest is the contrast in peaks between the plots, indicating (strong) shifts in pattern usage. A rather strong difference in pattern usage is visible for the lower indexes in the code table, corresponding to the longest, most specific, patterns. However, in this figure the high peaks are also indicative; we marked the peaks of an interesting case A1 and A2. These peaks are at exactly the same code table element, meaning that this pattern is used quite often in the covers of both $\mathcal{D}_1$ and $\mathcal{D}_2$. Note that it is not used at all in the

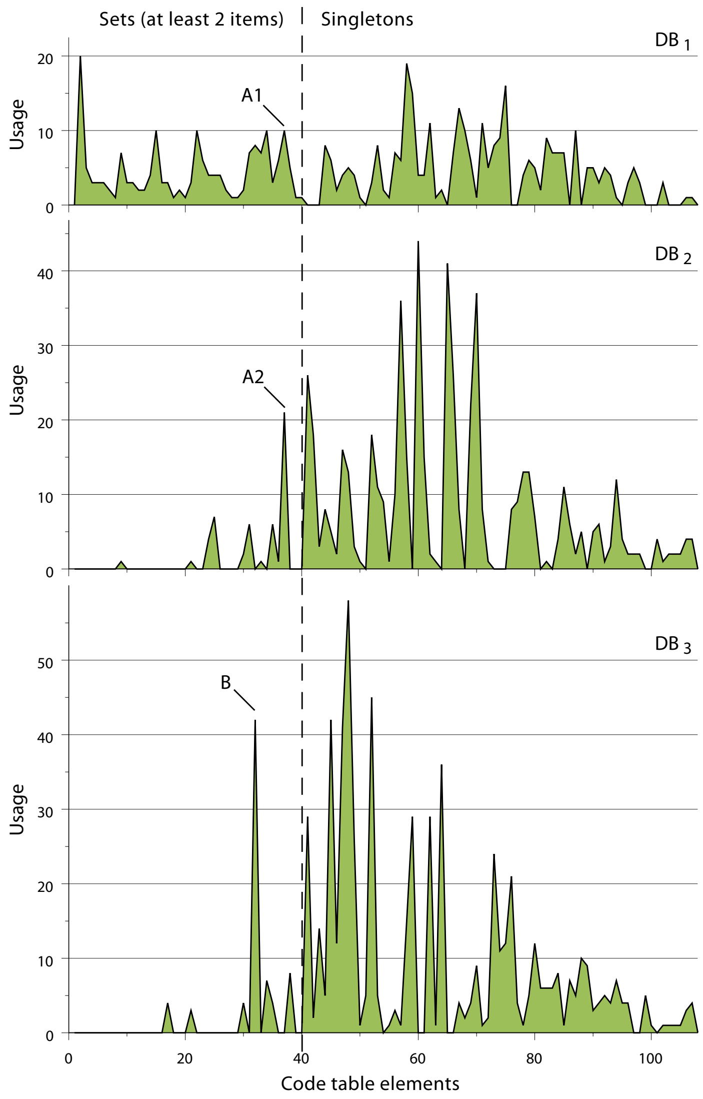


Figure 3.4: Comparing database covers. Each database of *Wine* has been covered by code table CT_1. Visualised is the absolute usage for each of the code table elements.

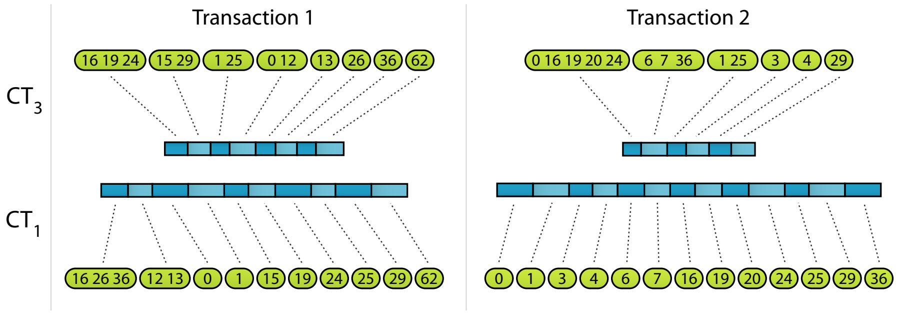


Figure 3.5: *Wine*; two transactions from $\mathcal{D}_3$ encoded by CT_3 (above) and CT_1 (below). The green rounded boxes visualise the itemsets making up the cover of the transaction. Each of the itemsets is linked to its code by the dashed line. The widths of the blue boxes, the codes, represent the actual computed code lengths.

cover of $\mathcal{D}_3$; hence this pattern could really give us a clue as to what differentiates $\mathcal{D}_1$ and $\mathcal{D}_2$ from $\mathcal{D}_3$. Another interesting peak is the one indicated with B: although it is also applied in the other covers, this pattern is clearly used much more often to cover $\mathcal{D}_3$.

Comparing Transaction Covers

A second approach for difference characterisation zooms in on individual rows in a database, and is thus especially useful when you are interested in specific transactions: why does a certain transaction belong to one database and not to another? Again, we use our code tables to inspect this.

Suppose we have two databases and their respective code tables. After computing the individual code length differences, it is easy to pick out those transactions that fit well in one database and not in another. After selecting a transaction, we can cover it with both code tables separately and visualise which patterns are used for this. In general, it will be covered by longer patterns with higher usage if it belongs to a certain distribution than if it does not. Manual inspection of the individual transaction covers can reveal valuable knowledge.

As an example, have a look at another *Wine* example in Figure 3.5. The encodings by CT_1 and CT_3 of two sets from $\mathcal{D}_3$ are shown. Left and right show the same transactions, but they are covered by different itemsets (depicted by the rounded boxes). The itemsets are linked to their codes with the dashed lines. The width of each black or white code represents the length of that particular code; together the sum of these widths makes up the total length of the encoded transaction.

Looking at the upper transaction, we observe that both code tables cover the transaction with itemsets of intermediate length. However, CT_3 uses less and different patterns in its cover than CT_1. Moreover, the code lengths are obviously shorter, relating to high occurrence in the distribution from which CT_3 was induced. For further inspection of how important such patterns are, we zoom in to the pattern level in the third approach.

The covers of the second transaction give an even larger contrast than the previous one. The native code table covers the transaction with few and large patterns, while the other one uses only singletons. We may therefore conclude this transaction fits very well in its native distribution and very bad in the other. This also shows in the lengths of the encodings. Both examples show again that more singletons are used in a cover when data doesn't belong to a distribution.

Comparing Code Tables

The final third method for difference inspection focuses on the individual patterns in a data distribution. In order to pinpoint the differences in this respect, we have to directly compare the patterns in two code tables.

The weight and importance of patterns in the code tables cannot be compared naively, as for many of the patterns in a code table there does not have to be a direct equivalent in the other code table. However, the set of patterns in a code table can also be regarded as a database; in that fashion we can actually apply code tables to each other to find out what the alternative encoded length for each pattern is.

For each pattern in a code table we can compare its own encoded length to that of the alternative provided by the other code table, similarly to what we did for transactions in the previous subsection. Likewise, if the distributions are similar, we expect the encoded lengths to be comparable; even if the code tables use rather different patterns to encode it. In contrast, exactly those patterns for which the encoded lengths differ significantly mark the difference between the distributions.

We analysed the CT_2 and CT_3 code tables of the *Wine* dataset, and found further evidence for what puts these databases apart. The first peak in the topmost plot of Figure 3.4 corresponds to the pattern (0 16 19 20 24) from CT_3, which, due to its high relative usage, is encoded natively using only 1.4bits. From the same figure we already know this pattern is not used when covering the other databases; suggesting that perhaps neither this pattern, nor anything like it exists in the other code tables. Confirmation comes from an encoded length of 12.6bits that CT_2 assigns to this pattern; making it one of the patterns for which the encoded lengths differ most. As CT_2 cannot use any of the more often used code table patterns, it has to resort to low-frequency singleton encoding; arguably the least efficient method for encoding a pattern.

From the definition of the *Wine* database and analysis above we conclude that the main difference between the two classes lies in the combination of certain levels of malic acid (element 0) and a corresponding colour intensity (16). While CT_3 has a number of patterns that give these short encodings, CT_2 has virtually none: this pattern does not occur in this data distribution.

The above example evidently shows that the differences between the data distributions can be directly analysed, and that through comparison of the code table encodings key differences can be extracted. Similarities as well as the differences between distributions are pinpointed.

3.5 Related Work

Our dissimilarity measure DS is clearly related to the Normalised Information Distance (NID) and its compression-based instantiation Normalized Compression Distance (NCD) [72]. With the NCD, general compressors like gzip are used as Kolmogorov complexity approximators and as such compressed sizes are used to measure distance between strings. As a generic distance, the NID has been successfully applied in a plethora of clustering tasks including small snippet based language and evolutionary tree rebuilding [30]. An adaptation was developed that has some practical data mining applications, among which compression-based anomaly detection [57].

However, the aim of the NID is different from ours: compression is only used as a means to quantify differences, not to qualitatively find what these differences are. In contrast, this is the main goal of our line of research. This is illustrated by the results of both earlier chapters and this chapter. By considering transactional databases instead of individual strings and building code tables that can be analysed, Krimp provides a very natural way to gain insight in the differences between data distributions.

Our dissimilarity measure is also related to Emerging Patterns [37], although there are major differences. First of all, here we only consider patterns that are MDL-wise important with respect to the data distribution of a single database. The code table built allows to investigate other data sets (or transactions) from that particular database's perspective. This in contrast to Emerging Patterns, which are by definition identified as differences between pairs of databases, without regarding individual data distributions. Although we here focus on identifying differences, Krimp also reveals similarities between databases; arguably equally important when inspecting two databases. Also, when a large number n of databases is to be compared, constructing n code tables is computationally less intensive than mining n^2 sets of Emerging Patterns.

Secondly, Emerging Patterns are defined as patterns having a large difference in support (growth rate) between two databases. However, the frequencies used in our approach depend on the database cover, thus taking into account other patterns (and their order) in the code table. Through these dependencies, important changes in the structure of the data are enlarged and therefore easier to spot.

Thirdly, Krimp only selects small numbers of patterns. This allows for manual inspection at all stages, from data distribution approximation to difference detection and characterisation. Emerging Patterns suffer from the same combinatory explosion problem as frequent patterns: in order to capture all differences, a low (zero) growth rate has to be used, resulting in obstructively many patterns. Shorter descriptions have been defined for EPs, for example using borders [38], but as these only give a shorter description for the same set of patterns, manual inspection remains impossible. The set of Emerging Patterns cannot straightforwardly be reduced by Krimp. First, because it operates on individual databases, not on pairs. Second, to satisfy the MDL assumption, the candidate pattern set should enable the algorithm to grasp full data distributions, not just differences. This is guaranteed by the frequent pattern set, but not by a set solely consisting of EPs.

3.6 Conclusions

In the previous chapter, the MDL-principle and its implementation in the Krimp algorithm have proven themselves to be a reliable way for approximating the data distributions of databases. Here, we used this principle to develop a database dissimilarity measure with which characteristic differences between databases can be discovered.

Histograms for encoded transaction lengths, and the differences thereof, show differences between data distributions straightforwardly. From the MDL principle, code tables with a good fit on the distribution of some data provide shorter codes and smaller standard deviations than code tables less suited for the data at hand. The code length difference is shown to be a good indication to how well a transaction fits a distribution.

We show the informative quality of the aggregation of the code length differences. The measured likenesses show close relation to the confusion matrices of earlier classification experiments; the number of misclassified instances drops according to this measure.

We define a generic dissimilarity measure on databases as the maximum of two mirrored aggregated code length difference measurements; it is symmetric and well suited to detect and characterise the differences between two databases. While we cannot prove it to fulfil the distance metric axioms, we argued that these hold for all practical purposes.

A large advantage of our method is that it allows for thorough inspection of the actual differences between data distributions. Based on the dissimilarity, three methods for detailed inspection are proposed. The most detailed method zooms in onto and compares the patterns that describe the data distribution in the code tables. Individual transactions that do not fit the current distribution well can be identified. Further, it can be analysed why they do not fit that distribution well. Last but not least is the possibility to take a more global stance and pinpoint under or over expressed patterns in the respective databases.

Dissimilarity measures are key to many different data mining algorithms. It is possible to apply our measure in a number of bioinformatics applications using these algorithms. For example, in those cases where classification appears to be hard; deeper insight in the causes of these problems may suggest new promising research directions.

Chapter **4**

Data Generation for Privacy Preservation

Many databases will not or cannot be disclosed without strong guarantees that no sensitive information can be extracted[1]. To address this concern several data perturbation techniques have been proposed. However, it has been shown that either sensitive information can still be extracted from the perturbed data with little prior knowledge, or that many patterns are lost.

In this chapter we show that generating new data is an inherently safer alternative. We present a data generator based on the models obtained by the KRIMP algorithm. These are accurate representations of the data distributions and can thus be used to generate data with the same characteristics as the original data.

Experimental results show a very large pattern-similarity between the generated and the original data, ensuring that viable conclusions can be drawn from the anonymised data. Furthermore, anonymity is guaranteed for suited databases and the quality-privacy trade-off can be balanced explicitly.

[1]This chapter has been published as [111]: J. Vreeken, M. van Leeuwen & A. Siebes. Data Generation for Privacy Preservation. In *Proceedings of the ICDM'07* (2007).

4.1 Introduction

Many databases will not or can not be disclosed without strong guarantees that no sensitive information can be extracted from it. The rationale for this ranges from keeping competitors from obtaining vital business information to the legally required protection of privacy of individuals in census data. However, it is often desirable or even required to publish data, leaving the question how to do this without disclosing information that would compromise privacy.

To address these valid concerns, the field of privacy-preserving data mining (PPDM) has rapidly become a major research topic. In recent years ample attention is being given to both defender and attacker stances, leading to a multitude of methods for keeping sensitive information from prying eyes. Most of these techniques rely on perturbation of the original data: altering it in such a way that given some external information it should be impossible to recover individual records within certainty bounds.

Data perturbation comes in a variety of forms, of which adding noise [9], data transformation [5] and rotation [28] are the most commonly used. At the heart of the PPDM problem is the balance between the quality of the released data and the amount of privacy it provides. While privacy is easily ensured by strongly perturbing the data, the quality of conclusions that can be drawn from it diminishes quickly. This is inherent of perturbation techniques: sensitive information cannot be fully masked without destroying non-sensitive information as well [56]. This is especially so if no special attention is given to correlations within the data by means of multidimensional perturbation [55], something which has hardly been investigated so far [77].

An alternative approach to the PPDM problem is to generate new data instead of perturbing the original. This has the advantage that the original data can be kept safe as the generated data is published instead, which renders data recovery attacks useless. To achieve this, the expectation that a data point in the generated database identifies a data point in the original database should be very low, whilst all generated data adhere to the characteristics of the original database. Data generation as a means to cover-up sensitivities has been explored in the context of statistical databases [74], but that method ignores correlations as each dimension is sampled separately.

We propose a novel method that uses data generation to guarantee privacy while taking important correlations into account. For this we use the Krimp algorithm from Chapter 2. Using the patterns picked by MDL, we can construct a model that generates data very similar (but not equal) to the original data. Experiments show that the generative model is well suited for producing data that conserves the characteristics of the original data while preserving privacy.

Using our generative method, it is easy to ensure that generated data points cannot reliably be traced to individual data points in the original data. We can thus easily obtain data that is in accordance with the well-known privacy measure k-anonymity [99]. Also, we can mimic the effects that can be obtained with l-diversity [78].

Although preserving intrinsic correlations is an important feat, in some applications preservation of particular patterns might be highly undesirable from a privacy point of view. Fortunately, this can easily be taken care of in our scheme by influencing model construction.

4.2 The Problem

Data Perturbation

Since Agrawal & Srikant [9] initiated the privacy-preserving data mining field, researchers have been trying to protect and reconstruct sensitive data. Most techniques use data perturbation and these can be divided into three main approaches, of which we will give an overview here.

The addition of random noise to the original data, obfuscating without completely distorting it, was among the first proposals for PPDM [9]. However, it was quickly shown that additive randomisation is not good enough [6]. The original data can often be reconstructed with little error using noise filtering techniques [56] – in particular when the distortion does not take correlations between dimensions into account [55].

The second class is that of condensation-based perturbation [5]. Here, after clustering the original data, new data points are constructed such that cluster characteristics remain the same. However, it has been observed that the perturbed data is often too close to the original, thereby compromising privacy [28]. A third major data perturbation approach is based on rotation of the data [28]. While this method seemed sturdy, it has recently been shown that with sufficient prior knowledge of the original data the rotation matrix can be recovered, thereby allowing full reconstruction of the original data [77]. In general, perturbation approaches suffer from the fact that the original data is used as starting point. Little perturbation can be undone, while stronger perturbation breaks correlations and non-sensitive information is also destroyed. In other words, there is a privacy-quality trade-off which cannot be balanced well.

In the effort to define measures on privacy, a few models have been proposed that can be used to obtain a definable amount of privacy. An example is the well-known k-anonymity model that ensures that no private information can be related to fewer than k individuals [99]. A lack of diversity in such masses can thwart privacy though and in some situations it is well possible to link private information to individuals. Improving on k-anonymity, the required critical diversity can be ensured using the l-diversity model. However, currently the available method can only ensure diversity for one sensitive attribute [78].

Data Generation

The second category of PPDM solutions consists of methods using data generation, generating new (privacy preserving) data instead of altering the original. This approach is inherently safer then data perturbation, as newly generated data points can not be identified with original data points. However, not much research has been done in this direction yet.

Liew *et al.* [74] sample new data from probability distributions independently for each dimension, to generate data for use in a statistical database. While this ensures high quality point estimates, higher order dependencies are broken - making it unsuited for use in data mining.

The condensation-based perturbation approach [5] could be regarded as a data generation method, as it samples new data points from clusters. However, as mentioned above, it suffers from the same problems as perturbation techniques.

Problem Statement

Reviewing the goals and pitfalls of existing PPDM methods, we conclude that a good technique should not only preserve privacy but also quality. This is formulated in the following problem statement:

Problem 2 (Privacy and Quality Preserving Database)**.** *A database $\mathcal{D}_{priv}$ induced from a database $\mathcal{D}_{orig}$ is privacy and quality preserving iff:*

- *no sensitive information in $\mathcal{D}_{orig}$ can be derived from $\mathcal{D}_{priv}$ given a limited amount of external information* (privacy requirement)*;*
- *models and patterns derived from $\mathcal{D}_{priv}$ by data mining techniques are also valid for $\mathcal{D}_{orig}$* (quality requirement)*.*

From this statement follows a correlated data generation approach to induce a privacy and quality preserving database $\mathcal{D}_{priv}$ from a database $\mathcal{D}_{orig}$, for which the above requirements can be translated into concrete demands.

Using Krimp, construct a model that encapsulates the data distribution of $\mathcal{D}_{orig}$ in the form of a code table consisting of frequent patterns. Subsequently, transform this code table into a pattern-based generator that is used to generate $\mathcal{D}_{priv}$.

It is hard to define an objective measure for the privacy requirement, as all kinds of 'sensitive information' can be present in a database. We guarantee privacy in two ways. Firstly, the probability that a transaction in $\mathcal{D}_{orig}$ is also present in $\mathcal{D}_{priv}$ should be small. Secondly, the more often a transaction occurs in $\mathcal{D}_{orig}$, the less harmful it is if it also occurs in $\mathcal{D}_{priv}$. This is encapsulated in the *Anonymity Score*, in which transactions are grouped by the number of times a transaction occurs in the original database (support):

Definition 7. *For a database $\mathcal{D}_{priv}$ based on $\mathcal{D}_{orig}$, define the* Anonymity Score *(AS) as*

$$AS(\mathcal{D}_{priv}, \mathcal{D}_{orig}) = \sum_{supp \in \mathcal{D}_{orig}} \frac{1}{supp} P(t \in \mathcal{D}_{priv} \mid t \in \mathcal{D}_{orig}^{supp}).$$

In this definition, $\mathcal{D}^{supp}$ is defined as the selection of $\mathcal{D}$ with only those transactions having a support of *supp*. For each support level in $\mathcal{D}_{orig}$, a score is obtained by multiplying a penalty of 1 divided by the support with the probability that a transaction in $\mathcal{D}_{orig}$ with given support also occurs in $\mathcal{D}_{priv}$. These scores are summed to obtain AS. Note that when all transactions in $\mathcal{D}_{orig}$ are unique (i.e. have a support of 1), AS is equal to the probability that a transaction in $\mathcal{D}_{orig}$ also occurs in $\mathcal{D}_{priv}$.

Worst case is when all transactions in $\mathcal{D}_{orig}$ also occur in $\mathcal{D}_{priv}$. In other words, if we choose $\mathcal{D}_{priv}$ equal to $\mathcal{D}_{orig}$, we get the highest possible score for this particular database, which we can use to normalise between 0 (no privacy at all) and 1 (best possible privacy):

Definition 8. *For a database $\mathcal{D}_{priv}$ based on $\mathcal{D}_{orig}$, define the* Normalised Anonymity Score *(NAS) as*

$$NAS(\mathcal{D}_{priv}, \mathcal{D}_{orig}) = 1 - \frac{AS(\mathcal{D}_{priv}, \mathcal{D}_{orig})}{AS(\mathcal{D}_{orig}, \mathcal{D}_{orig})}.$$

To conform to the quality requirement, the frequent pattern set of $\mathcal{D}_{priv}$ should be very similar to that of $\mathcal{D}_{orig}$. We will measure pattern-similarity in two ways: 1) on database level through a database dissimilarity measure (see Section 3.3) and 2) on the individual pattern level by comparing frequent pattern sets. For the second part, pattern-similarity is high iff the patterns in $\mathcal{D}_{orig}$ also occur in $\mathcal{D}_{priv}$ with (almost) the same support. So,

$$P(|supp_{priv} - supp_{orig}| > \delta) < \epsilon. \tag{4.1}$$

The probability that a pattern's support in $\mathcal{D}_{orig}$ differs much from that in $\mathcal{D}_{priv}$ should be very low: the larger δ, the smaller ϵ should be. Note that this second validation implies the first: only if the pattern sets are highly similar, the code tables become similar, which results in low measured dissimilarity. Further, it is computationally much cheaper to measure the dissimilarity than to compare the pattern sets.

4.3 Preliminaries

In this chapter we discuss categorical databases. A database $\mathcal{D}$ is a bag of tuples (or transactions) that all have the same attributes $\{A_1, \dots, A_n\}$. Each attribute A_i has a discrete domain of possible values $V_i \in \mathcal{V}$.

The Krimp algorithm operates on itemset data, as which categorical data can easily be regarded. The union of all domains $\cup V_i$ forms the set of items $\mathcal{I}$. Each transaction t can now also be regarded as a set of items $t \subseteq \mathcal{P}(\mathcal{I})$. Then, as in the previous chapters, an itemset $X \subseteq \mathcal{I}$ occurs in a transaction $t \in \mathcal{D}$ iff $X \subseteq t$. The support of X in $\mathcal{D}$ is the number of transactions in the database in which X occurs. Speaking in market basket terms, this means that each item for sale is represented as an attribute, with the corresponding domain consisting of the values 'bought' and 'not bought'.

In this chapter, Krimp without pruning is applied, since keeping all patterns in the code table causes more diversity during data generation, as will become clear later.

4.4 Krimp Categorical Data Generator

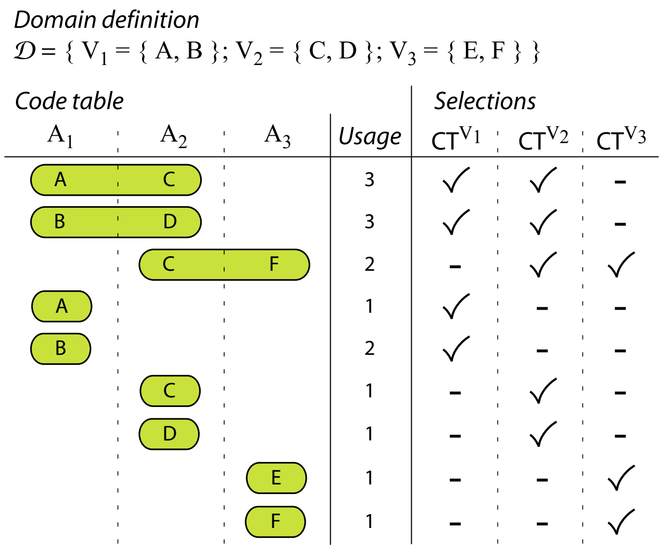
In this section we present our categorical data generation algorithm. We start off with a simple example, sketching how the algorithm works by generating a single transaction. After this we will detail the scheme formally and provide the algorithm in pseudo-code.

Generating a Transaction, an Example

Suppose we need to generate a new transaction for a simple three-column categorical database. To apply our generation scheme, we need a domain definition $\mathcal{V}$ and a Krimp code table CT, both shown in Figure 4.1.

We start off with an empty transaction and fill it by iterating over all domains and picking an itemset from the code table for each domain that has no value yet. We first want to assign a value for the first domain, V_1, so we have to select one pattern from those patterns in the code table that provide a value for this domain. This subset is shown as selection CT^{V_1}.

Using the usages of the code table elements as probabilities, we randomly select an itemset from CT^{V_1}; elements with high usage occur more often in the



Domain definition

$\mathcal{D} = \{ V_1 = \{ A, B \}; V_2 = \{ C, D \}; V_3 = \{ E, F \} \}$

Code table / *Selections*

A_1	A_2	A_3	*Usage*	CT^{V_1}	CT^{V_2}	CT^{V_3}
A	C		3	✓	✓	-
B	D		3	✓	✓	-
	C	F	2	-	✓	✓
A			1	✓	-	-
B			2	✓	-	-
	C		1	-	✓	-
	D		1	-	✓	-
		E	1	-	-	✓
		F	1	-	-	✓

Figure 4.1: Example for 3-column database. Each usage is Laplace corrected by 1.

original database and are thus more likely to be picked. Here we randomly pick 'BD' (probability 3/9). This set selects value 'B' from the first domain, but also assigns a value to the second domain, namely 'D'.

To complete our transaction we only need to choose a value for the third domain. We do not want to change any values once they are assigned, as this might break associations within an itemset previously chosen. So, we do not want to pick any itemset that would re-assign a value to one of the first two domains. Considering the projection for the third domain, CT^{V_3}, we thus have to ignore set CF(2), as it would re-assign the second domain to 'C'. From the remaining sets E(3) and F(3), both with usage 3, we randomly select one - say, 'E'. This completes generation of the transaction: 'BDE'. With different rolls of the dice it could have generated 'BCF' by subsequently choosing CF(2) and B(3), and so on.

Here we will detail our data generator more formally. First, define the projection CT^V as the subset of itemsets in CT that define a value for domain $V \in \mathcal{V}$. To generate a database, our categorical data generator requires four ingredients: the original database, a Laplace correction value, a *minsup* value for mining candidates for the Krimp algorithm and the number of transactions

Algorithm 6 Krimp Categorical Data Generator

```
  GenerateDatabase($\mathcal{D}$, $laplace$, $minsup$, $numtrans$) :
 1. $gdb \leftarrow \emptyset$
 2. $CT \leftarrow$ Krimp($\mathcal{D}$, MineCandidates($\mathcal{D}, minsup$))
 3. for each $X \in CT$ do
 4.     $X$.usage $\leftarrow X$.usage $+ laplace$
 5. end for
 6. $\mathcal{V} \leftarrow \mathcal{D}$.getDomains
 7. while $|gdb| < numtrans$ do
 8.     $gdb \leftarrow gdb +$ GenerateTransaction($CT, \mathcal{V}$)
 9. end while
10. return $gdb$
  GenerateTransaction($CT, \mathcal{V}$) :
11. $t \leftarrow \emptyset$
12. while $\mathcal{V} \neq \emptyset$ do
13.     pick a random $V \in \mathcal{V}$
14.     $X \leftarrow$ PickRandomItemSet($CT^V$)
15.     $t \leftarrow t \cup X$
16. end while
17. for each domain $W$ for which $X$ has a value do
18.     $CT \leftarrow CT \setminus CT^V$
19.     $\mathcal{V} \leftarrow \mathcal{V} \setminus W$
20. end for
21. return $t$
  PickRandomItemSet($CT$) :
22. $weights \leftarrow \{usage_{CT}(X) \mid X \in CT\}$
23. $X \leftarrow$ WeightedSample($weights, CT$)
24. return $X$
```

that is to be generated. We present the full algorithm as Algorithm 6.

Generation starts with an empty database *gdb* (line 1). To obtain a code table CT, the Krimp algorithm is applied to the original database $\mathcal{D}$ (2). A Laplace correction *laplace* is added to all code table elements (3–5). Next, we return the generated database when it contains *numtrans* transactions (7–10).

Generation of a transaction is started with an empty transaction t (11). As long as $\mathcal{V}$ is not empty (12), our transaction is not finished and we continue. First, a domain V is randomly selected (13). From the selection CT^V, one itemset is randomly chosen, with probabilities defined by their relative usages (14). After the chosen set is added to t (15), we filter from CT all sets that would redefine a value - i.e. those sets that intersect with the definitions of the domains for which t already has a value (17–18). Further, to avoid reconsideration we also filter these domains from $\mathcal{V}$ (19). After this the next domain is picked from $\mathcal{V}$ and another itemset is selected; this is repeated until $\mathcal{V}$ is empty (and t thus has a value from each domain).

Note that code table elements are treated fully independently, as long as they do not re-assign values. Correlations between dimensions are stored explicitly in the itemsets and are thus taken into account implicitly.

Besides the original database and the desired number of generated transactions, the database generation algorithm requires two other parameters: *laplace* and *minsup*. Both fulfil an important role in controlling the amount of privacy provided in the generated database, which we will discuss here in more detail.

A desirable parameter for any data generation scheme is one that controls the data diversity and strength of the correlations. In our scheme this parameter is found in the form of a Laplace correction. Before the generation process, a small constant is added to the usage of the code table elements. As code tables always contains all single values, this ensures that all values for all categories have at least a small probability of being chosen. Thus, 1) a complete transaction can always be generated and 2) all possible transactions can be generated. For this purpose the correction needs only be small. However, the strength of the correction influences the chance an otherwise unlikely code table element is used; with larger correction, the influence of the original data distribution is dampened and diversity is increased.

The second parameter to our database generation algorithm, *minsup*, has a strong relation to the k-anonymity blend-in-the-crowd approach. The *minsup* parameter has (almost) the same effect as k: patterns that occur less than *minsup* times in the original database are not taken into account by Krimp. As they cannot get in the code table, they cannot be used for generation either. Particularly, complete transactions have to occur at least *minsup* times in order for them to make it to the code table. In other words, original transactions that occur less often than *minsup* can only be generated if by chance often occurring patterns are combined such that they form an original transaction. As code table elements are regarded independent, it follows that when more patterns have to be combined, it becomes less likely that transactions are generated that also exist in the original database.

4.5 Experiments

In this section we will present empirical evidence of the method's ability to generate data that provides privacy while still allowing for high quality conclusions to be drawn from the generated data.

Table 4.1: Database characteristics and dissimilarities.

			Krimp	Dissimilarity (DS)	
Dataset	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{V}\rvert$	*minsup*	*Gen. vs. orig.*	*Orig. internal*
Chess (kr–k)	28056	7	1	0.037	0.104
Iris	150	5	1	0.047	0.158
Led7	3200	8	1	0.028	0.171
Letter recognition	20000	17	50	0.119	0.129
Mushroom[2]	8124	22	20	0.010	0.139
Nursery	12960	9	1	0.011	0.045
Page blocks	5473	11	1	0.067	0.164
Pen digits	10992	17	50	0.198	0.124
Pima	786	9	1	0.110	0.177
Quest A	4000	8	1	0.016	0.077
Quest B	10000	16	1	0.093	0.223

Database characteristics, candidate *minsup* and dissimilarity measurements (between original and generated datasets) for a range of datasets. As candidates, frequent itemsets up to the given minimum support level were used.

Experimental Setup

In our experiments, we use a selection from the commonly used UCI repository [32]. Also, we use two additional databases that were generated with IBM's Quest basket data generator [8]. To ensure that the Quest data obeys our categorical data definition, we transformed it such that each original item is represented by a domain with two categories, in a binary fashion (present or not). Both Quest datasets were generated with default settings, apart from the number of columns and transactions.

Characteristics of all used datasets are summarised in Table 4.1, together with the minimum support levels we use for mining the frequent itemsets that function as candidates for Krimp.

For all experiments we used a Laplace correction parameter of 0.001, an arbitrarily chosen small value solely to ensure that otherwise zero-usage code table elements can be chosen during generation. All experimental results presented below are averaged over 10 runs and all generated databases have the same number of transactions as the originals, unless indicated otherwise.

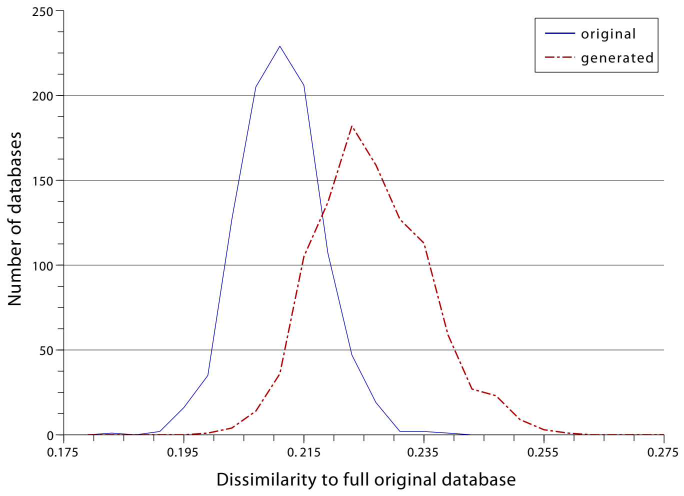


Figure 4.2: Histogram of dissimilarities between samples (original and generated) and the full original database, *Chess (kr–k)*.

Results

To quantify the likeness of the generated databases to their original counterparts, we use the database dissimilarity measure as described in Section 3.3. To judge these measurements, we also provide the dissimilarity between the original database and independent random samples of half the size from the original database.

In Table 4.1 we show both these internal dissimilarity scores and the dissimilarity measurements between the original and generated databases. To put the reported dissimilarities in perspective, note that the dissimilarity measurements between the classes in the original databases range from 0.29 up to 12 (see Chapter 3). The measurements in Table 4.1 thus indicate clearly that the generated databases adhere very closely to the original data distribution; even better than a randomly sampled subset of 50% of the original data captures the full distribution.

To show that the low dissimilarities for the generated databases are not caused by averaging, we provide a histogram in Figure 4.2 for the *Chess (kr–k)* dataset. We generated thousand databases of 7500 transactions, and measured the dissimilarity of these to the original database. Likewise, we also measured dissimilarity to the original database for equally many and equally sized independent random samples. The peaks for the distance histograms lie very near

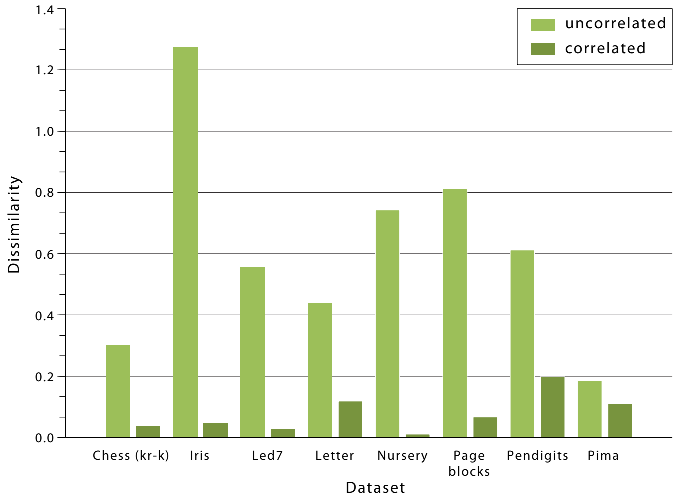


Figure 4.3: Dissimilarity scores between generated (with and without correlations) and original databases.

to each other at 0.21 and 0.22 respectively. This and the very similar shapes of the histograms confirm that our generation method samples databases from the original distribution.

Turning back to Table 4.1, we notice that databases generated at higher values of the *minsup* parameter show slightly larger dissimilarity. The effect of this parameter is further explored in Figures 4.3 and 4.4. First, the bar diagram in Figure 4.3 shows a comparison of the dissimilarity scores between uncorrelated and correlated generation: uncorrelated databases are generated by a code table containing only individual values (and thus no correlations between domains can exist), correlated databases are generated using the *minsup* values depicted in Table 4.1 (at which the correlations in the data are captured in the patterns in the code table). We see that when generation is allowed to take correlations into account, the generated databases are far more similar to the original ones.

Secondly, the graph in Figure 4.4 shows the dissimilarity between the original *Pen digits* database and databases generated with different values for *minsup*. As expected, lower values of *minsup* lead to databases more similar to the original, as the code table can better approximate the data distribution of the original data. For the whole range of generated databases, individual value frequencies are almost identical to those of the original database; the increase

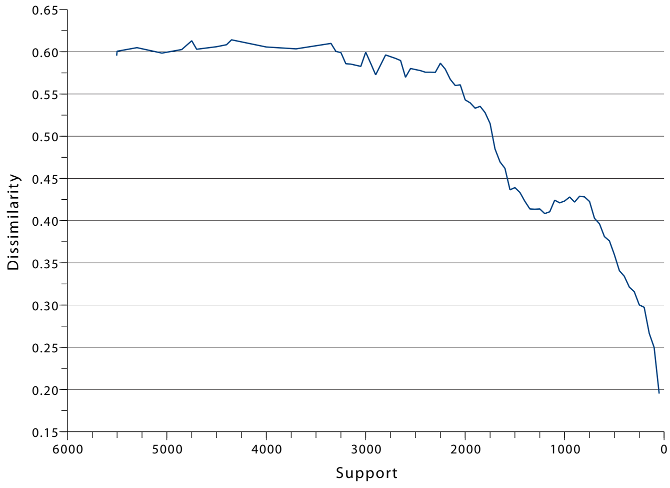


Figure 4.4: Dissimilarity between generated database (at different *minsups*) and the original database for *Pen digits*.

in similarity is therefore solely caused by the incorporation of the right (type and strength of) correlations.

Now that we have shown that the quality of the generated databases is very good on a high level, let us consider quality on the level of individual patterns. For this, we mined frequent itemsets from the generated databases with the same parameters as we did for candidate mining on the original database. A comparison of the resulting sets of patterns is presented in Table 4.2. We report these figures for those databases for which it was feasible to compute the intersection of the frequent pattern collections.

Large parts of the generated and original frequent pattern sets consist of exactly the same items sets, as can be seen from the first column. For example, for *Led7* and *Nursery* about 90% of the mined itemsets is equal. In the generated *Pen digits* database a relatively low 25% of the original patterns are found. This is due to the relatively high *minsup* used: not all correlations have been captured in the code table. However, of those patterns mined from the generated database, more than 90% is also found in the original frequent pattern set.

For itemsets found in both cases, the average difference in support between original and generated is very small, as the second column shows. *Iris* is a bit of an outlier here, but this is due to the very small size of the dataset.

Table 4.2: Frequent pattern set comparison.

Dataset	*% equal itemsets*	*% avg supp diff equal itemsets*	*% avg supp new itemsets*
Chess (kr–k)	71	0.01	0.01
Iris	83	1.69	0.80
Led7	89	0.14	0.06
Nursery	90	0.04	0.03
Page blocks	75	0.06	0.02
Pen digits	25	0.50	0.59
Pima	60	0.30	0.14

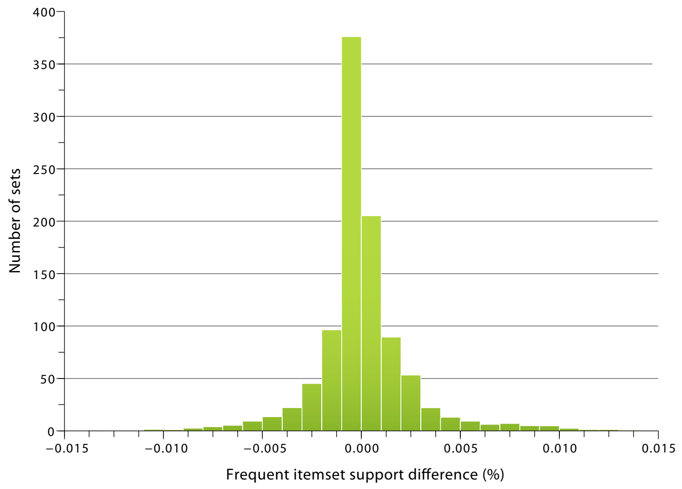


Figure 4.5: Difference in support, supp($\mathcal{D}_{priv}$)−supp($\mathcal{D}_{orig}$), for identical itemsets in generated and original *Led7*.

Not only the average is low, standard deviation is also small: as can be seen from Figure 4.5, almost all sets have a very small support difference. The generated databases thus fulfil the support difference demands we formulated in Equation 4.1.

The third column of Table 4.2 contains the average supports of itemsets that are newly found in the generated databases; these supports are very low. All this together clearly shows that there is a large pattern-similarity, thus

Table 4.3: Normalised Anonymity Scores.

Dataset	*NAS*	*Dataset*	*NAS*
Chess (kr–k)	0.70	Page blocks	0.23
Iris	0.28	Pen digits	0.78
Led7	0.34	Pima	0.36
Letter recognition	0.69	Quest A	0.16
Mushroom	0.91	Quest B	0.19
Nursery	0.51		

showing a high quality according to our problem statement.

However, this quality is of no worth if the generated data does not also preserve privacy. To measure the level of provided anonymity, we calculate the *Normalised Anonymity Score* as given by Definition 8. These scores are presented in Table 4.3.

As higher scores indicate better privacy, some datasets (e.g. *Mushroom*, *Pen digits*) are anonymised very well. On the other hand, other datasets (*Page blocks*, *Quest*) do not seem to provide good privacy. As discussed in Section 4.4, the *minsup* parameter of our generation method doubles as a k-anonymity provider.

This explains that higher values for *minsup* result in better privacy, as the measurements for *Letter recognition*, *Mushroom* and *Pen digits* indeed show. Analogously, the (very) low *minsup* values used for the other databases result in lower privacy (aside from data characteristics to which we'll return shortly).

To show the effect of *minsup* in action, as an example we increase the *minsup* for the *Chess* database to 50. While the so-generated database is still very similar to the original (dissimilarity of 0.19), privacy is considerably increased - which is reflected by a *Normalised Anonymity Score* of 0.85. For further evidence of the k-anonymity obtained, we take a closer look at *Pen digits*, for which we use a *minsup* of 50. Of all transactions with support < 50 in the generated database, only 3% is also found in the original database with support < 50. It is thus highly unlikely that one picks a 'real' transaction from the generated database with support lower than *minsup*.

Although not all generated databases preserve privacy very well, the results indicate that privacy can be obtained. This raises the question when privacy can be guaranteed. This not only depends on the algorithm's parameters, but also on the characteristics of the data. It is difficult to determine the structure of the data and how the parameters should be set in advance, but during the generation process it is easy to check whether privacy is going to be good.

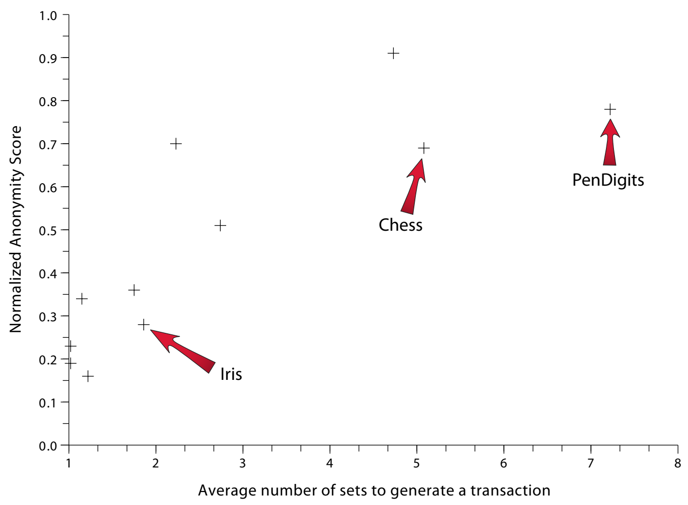


Figure 4.6: Average number of patterns used to generate a transaction versus Normalised Anonymity Score, for all datasets in Table 4.1.

The key issue is whether transactions are generated by only very few or many code table elements. In Figure 4.6 we show this relation: for each dataset in Table 4.1, a cross marks the average number of itemsets used to generate a single transaction and the *Normalised Anonymity Score*. In the top-right corner we find the generated databases that preserve privacy well, including *Pen digits* and *Letter recognition*. At the bottom-left reside those databases for which too few elements per transaction are used during generation, leading to bad privacy; *Quest*, *Page blocks* and *Led7* are the main culprits. Thus, by altering the *minsup*, this relation allows for explicit balancing of privacy and quality of the generated data.

4.6 Discussion

The experimental results in the previous section show that the databases generated by our Krimp Categorical Data Generator are of very high quality; pattern similarity on both database level and individual pattern level is very high. Furthermore, we have shown that it is possible to generate high quality databases while privacy is preserved. The *Normalised Anonymity Scores* for some datasets are pretty high, indicating that hardly any transactions that occur few times in the original database also occur in the generated database. As expected, increasing *minsup* leads to better privacy, but dissimilarity remains good and thus the trade-off between quality and privacy can be balanced explicitly.

A natural link between our method and k-anonymity is provided by the *minsup* parameter, of which we have shown that it works in practice. While we haven't explored this parameter in this work, it is also possible to mimic l-diversity, as in our method the *laplace* parameter acts as diversity control. The higher the Laplace correction, the less strong the characteristics of the original data are taken into account (thus degrading quality, but increasing diversity). Note that one could also increase the Laplace correction for specific domains or values, thereby dampening specific (sensitive) correlations - precisely the effect l-diversity aims at.

To obtain even better privacy, one can also directly influence model construction: for example, by filtering the Krimp candidates prior to building the code table. Correlations between specific values and/or categories can be completely filtered. If correlations between values A and B are sensitive, then by removing all patterns containing both A and B from the candidate set, no such pattern can be used for generation.

From Figure 4.6 followed that the number of patterns used to generate a transaction greatly influences privacy: more elements leads to higher anonymity. In the same line of thought, the candidate set can be filtered on pattern length; imposing a maximum length directly influences the number of patterns needed in generation, and can thus increase the provided anonymity.

The average number of patterns needed to generate a transaction is a good indication of the amount of anonymity. We can use this property to check whether parameters are chosen correctly and to give a clue on the characteristics of the data. If already at high *minsup* few patterns are needed to encode a transaction, and thus hardly any 'sensitive' transactions occur, the database is not 'suited' for anonymisation through generation.

Reconsidering our problem statement in Section 4.2, the Krimp generator does a good job as solution for this PPDM problem. The concrete demands we posed for both the quality and privacy requirements are met, meaning that databases generated by our method are privacy and quality preserving as we interpreted this in our problem statement. Generating new data is therefore a good alternative to perturbing the original data.

Our privacy-preserving data generation method could be well put to practice in the distributed system Merugu & Ghosh [84] proposed: to cluster privacy-preserving data in a central place without moving all the data there, a privacy-preserving data generator for each separate location is to be built. This is exactly what our method can do and this would therefore be an interesting application. Because the quality of the generated data is very high, the method could also be used in limited bandwidth distributed systems where privacy is not an issue. For each database that needs to be transported, construct a code table and communicate this instead of the database. If precision on the individual transaction level is not important, new highly similar data with the same characteristics can be generated.

In this chapter, we generated databases of the same size as the original, but the number of generated transactions can of course be varied. Therefore, the method could also be used for up-sampling. Furthermore, it could be used to induce probabilities that certain transactions or databases are sampled from the distribution represented by a particular code table.

4.7 Conclusions

We introduce a pattern-based data generation technique as a solution to the privacy-preserving data mining problem in which data needs to be anonymised. Using the MDL-based Krimp algorithm we obtain accurate approximations of the data distribution, which we transform into high-quality data generators with a simple yet effective algorithm.

Experiments show that the generated data meets the criteria we posed in the problem statement, as privacy can be preserved while the high quality ensures that viable conclusions can still be drawn from it. The quality follows from the high similarity to the original data on both the database and individual pattern level. Anonymity scores show that original transactions occurring few times only show up in the generated databases with very low probability, giving good privacy.

Preserving privacy through data generation does not suffer from the same weaknesses as data perturbation. By definition, it is impossible to reconstruct the original database from the generated data, with or without prior knowledge. The privacy provided by the generator can be regulated and balanced with the quality of the conclusions drawn from the generated data. For suited databases, the probability of finding a 'real' transaction in the generated data is extremely low.

Chapter **5**

StreamKrimp – Detecting Change in Data Streams

Data streams are ubiquitous[1]. Examples range from sensor networks to financial transactions and website logs. In fact, even market basket data can be seen as a stream of sales. Detecting changes in the distribution a stream is sampled from is one of the most challenging problems in stream mining, as only limited storage can be used. In this chapter we analyse this problem for streams of transaction data from an MDL perspective.

Based on this analysis we introduce the StreamKrimp algorithm, which uses the Krimp algorithm to characterise probability distributions with code tables. With these code tables, StreamKrimp partitions the stream into a sequence of substreams. Each switch of code table indicates a change in the underlying distribution. Experiments on both real and artificial streams show that StreamKrimp detects the changes while using only a very limited amount of data storage.

[1]This chapter has been published as [69]: M. van Leeuwen & A. Siebes. StreamKrimp: Detecting Change in Data Streams. In *Proceedings of the ECML PKDD'08* (2008).

5.1 Introduction

Data streams are rapidly becoming a dominant data type or data source. Common examples of streams are sensor data, financial transactions, network traffic and website logs. Actually, it would also be appropriate to regard supermarket basket data as a stream, as this is often a - seemingly never-ending - flow of transactions.

Detecting change in streams has traditionally attracted a lot of attention [1, 36, 59, 90], both because it has many possible applications and because it is a hard problem. In the financial world, for example, quick reactions to changes in the market are paramount. Supermarkets and on-line stores have to respond quickly to changing interests of their customers. As a final example, web hosts have to respond to changes in the way users use their websites.

The unbounded growth of a stream causes the biggest challenges in stream mining: after all, only limited storage and computational capacity is available. To address this, many existing algorithms use a sliding window [1, 36, 59]. The problem with this approach is that often a fixed window size has to be set in advance, which strongly influences the results. Some algorithms avoid this by using an adaptive window size [118]. Many current methods focus on single-dimensional item streams or multi-dimensional real-valued streams [1, 2, 59, 88, 90]. In this chapter, we address the problem of detecting change in what we call data streams, that is, streams of transactions. A change in such a stream of transactions is a change in the distribution the transactions are sampled from. So, given a data stream, we would like to identify, online, a series of consecutive substreams that have different sampling distributions.

Our approach to the problem is again based on the MDL principle. We use code tables to compress the data stream, as this allows to characterise the sampling distributions and to detect shifts between such distributions. If we would have unbounded data storage, the MDL-optimal partitioning of the data stream would be that one that minimises the total compressed length. However, unbounded storage is clearly not realistic and we will have to resort to a solution that is at best locally optimal.

With bounded storage, the best approach is to first identify distribution P_1 at the beginning of the stream and then look for a shift to distribution P_2. When P_2 has been identified, we look for the next shift, etc. We give MDL-based solutions for both of these sub-problems.

We turn this MDL-based analysis of the problem into algorithms using our Krimp algorithm (see Chapter 2). Here, we use the code tables given by this algorithm as foundation for the change detection algorithm StreamKrimp, which characterises streams on-the-fly.

We empirically test StreamKrimp on a variety of data streams. Results on two types of artificial streams show that changes in distribution are detected at the right moment. Furthermore, the experiment on a real stream shows that large data streams pose no problem and changes are accurately spotted while noise is neglected.

5.2 The Problem Assuming Unbounded Storage

Preliminaries

We assume that our data consists of a *stream* of *transactions* over a fixed set of items $\mathcal{I}$. That is, each transaction is simply a subset of $\mathcal{I}$ and a stream S is an unbounded ordered sequence of transactions, i.e. $S = s_1 s_2 s_3 \cdots$ in which $s_i \subseteq \mathcal{I}$. The individual transactions in a stream are identified by an integer; without loss of generality we assume that this index is consecutive.

A finite stream T is a substream of S if T consists of consecutive elements of S. In particular $S(i, j)$ is the substream that starts at the i-th element and stops at the j-th.

The Problem in Distribution Terms

We assume that S consists of a, possibly infinite, set of consecutive non-overlapping subsequences $S = s_1 s_2 s_3 \cdots$ such that

– s_i is drawn i.i.d. from distribution P_i on $\mathcal{P}(\mathcal{I})$, and

– $\forall i \in \mathbb{N} : P_i \neq P_{i+1}$.

So, informally, the problem is:

Given such a sequence S, identify the subsequences S_i.

The, somewhat loosely formulated, assumption underlying this problem is that the S_i are big enough to identify. If the source of the stream would change the sample distribution at every transaction it emits, identification would be impossible.

From Distributions to MDL

The two main ingredients of the problem are:

1. How do we identify the distributions P_i?
2. How do we recognise the shift from P_i to P_{i+1}?

If both these problems are solved, the identification of the S_i is trivial.

If the P_i would belong to some well-known family of distributions, the first problem would be solvable by parameter estimation. Unfortunately, this is not a reasonable assumption.

Rather than trying to estimate the underlying distributions directly, we resort, again, to MDL. In this chapter, we employ MDL both to identify the P_i and to identify the shifts from P_i to P_{i+1}.

Streams of transactions are subtly different from transaction databases. The most important difference is that streams are unbounded. This means, e.g. that some care has to be taken to define the support of an itemset in a stream.

Itemsets in Streams

An itemset X is, as usual, a set of items. That is, $X \subseteq \mathcal{I}$. An itemset X occurs in a transaction s_i in stream S, iff $X \subseteq s_i$. While streams may be infinite, at any point in time we will only have seen a finite substream. In other words, we only have to consider the support of itemsets on finite streams. The support of an itemset X on a finite stream S is defined as usual: the number of transactions in S in which X occurs.

Coding Finite Data Streams

As in the previous chapters, we use code tables to compress data streams. Since we only consider finite data streams, we can treat a finite data stream S exactly the same as a 'regular' dataset $\mathcal{D}$. Hence, we have the following definition.

Definition 9. *Let S be a finite data stream over $\mathcal{I}$ and let CT be a code table that is code-optimal for S. The total size of the encoded data stream, denoted by $L(S, CT)$, is given by*

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

An optimal code table is one that minimises the total size.

Definition 10. *Let S be a finite data stream over $\mathcal{I}$ and let $\mathcal{CT}$ be the set of code tables that are code-optimal for S. $CT^{opt} \in \mathcal{CT}$ is called* optimal *if*

$$CT^{opt} = \underset{CT \in \mathcal{CT}}{\operatorname{argmin}}\, L(S, CT).$$

The total size of the stream S encoded with an optimal code table CT^{opt} is called its optimal size and is denoted by $\mathcal{L}(S)$:

$$\mathcal{L}(S) = L(S, CT^{opt}).$$

The Problem in MDL Terms

Now that we know how to code finite data streams, we can formalise our problem in MDL terminology:

Problem 3 (Data Stream Partitioning with Unbounded Storage). *Let S be a finite data stream, partition S into consecutive substreams $S_1, \ldots, S_k$, such that*

$$\sum_{i=1}^{k} \mathcal{L}(S_i) \text{ is minimised.}$$

5.3 The Problem Assuming Bounded Storage

The Problem of Streams

Let S, T and U be finite data streams, such that U is the concatenation of S and T. There is no guarantee that the optimal partition of U coincides with the optimal partition of S on the S-part of U. This observation points out two disadvantages of the problem as stated above.

1. It assumes knowledge of the complete stream; this is a flagrant contradiction to the main idea of data streams: they are too big to store.
2. It disregards the dynamic nature of data streams. Changes in the underlying distribution can only be detected after the whole stream has been observed. Clearly, such a posteriori results are not that useful.

In other words, we will have to settle for a partitioning that is at best locally optimal.

Too Large to Store: One Distribution

If the stream S is sampled i.i.d. from one distribution only, the estimates of $P(X \mid S(1,n))$ get closer and closer to their true value. That is, we have the following lemma.

Lemma 6. *Let data stream S be drawn i.i.d from distribution Q on $\mathcal{P}(\mathcal{I})$, then*

$$\forall X \in \mathcal{P}(\mathcal{I}) : \lim_{n \to \infty} P(X \mid S(1,n)) = Q(X).$$

This well-known statistical result has an interesting result for code tables: code tables converge! To make this more precise, denote by CT_n an optimal code table on S_n. Moreover, let $CT(S(1,j))$ be a shorthand for $L_{CT}(S(1,j))$.

Theorem 7. *Let data stream S be drawn i.i.d from distribution Q on $\mathcal{P}(\mathcal{I})$, then*

$$\forall k \in \mathbb{N} : \lim_{n \to \infty} |CT_n(S(1,n)) - CT_{n+k}(S(1,n))| = 0.$$

Proof. Let FCT be a code table in which only the left-hand column is specified. Lemma 6 implies that

$$\forall X \in \mathcal{P}(\mathcal{I}) \; \forall k \in \mathbb{N} : \lim_{n \to \infty} |P(X \mid S(1,n)) - P(X \mid S(1,n+k))| = 0.$$

In other words, the optimal codes we assign to the itemsets in FCT become the same in the limit. But this implies that an optimal code table on $S(1,n+k)$ is, in the limit, also an optimal code table on $S(1,n)$. $\square$

That is, if our stream comes from one distribution only, we do not need to store the complete stream to induce the optimal code table. A large enough sample suffices. Denote by $CT^{app}(S)$ the optimal code table induced from a large enough "head" of the stream, i.e. after convergence has set in. This head of the stream is denoted by $H(S)$. Note that Theorem 7 also suggests a way to check that the sample is large enough. If for some reasonable sized k,

$$|L(S(1,n), CT_n) - L(S(1,n), CT_{n+k})|$$

gets small, we may conclude convergence. Small is, of course, a relative notion: if $L(S(1,n), CT_n)$ is millions of bits, a difference of a few thousand bits can already be considered as small. Hence, it is better to look at a weighted version; which is our improvement rate, defined as follows.

Definition 11. *With the notations from above, the* Improvement Rate *IR is given by*

$$\frac{|L(S(1,n), CT_n) - L(S(1,n), CT_{n+k})|}{L(S(1,n), CT_n)}.$$

When IR becomes small in an absolute sense, we may conclude convergence. We return to this observation later.

Too Large to Store: Detecting Change

So, for a data stream that comes from one distribution, the problem is easy. The optimal code table can be computed from a large enough head of the stream. After this code table has been computed, no further data storage is necessary. The problem is, however, that after a while the distribution changes. How can we detect that?

Let the, finite, stream $S = S_1 S_2$ such that S_i is sampled from distribution P_i. Moreover, let CT_i denote the optimal code table on S_i. To detect the change in distribution, we need that

$$L(S_1, CT_1) + L(S_2, CT_2) < L(S, CT).$$

This inequality translates to

$$L(S_1, CT_1^{app}) + L(S_2, CT_2^{app}) < L(S_1, CT_1^{app}) + L(S_2 \mid CT_1^{app}).$$

Note that $L(S, CT)$ translates to the sum of the two heads encoded with CT_1^{app} because CT_1^{app} has converged. That is, if there is *no* change in the underlying distribution, CT_1^{app} is still the correct code table. The second summand has the bar $|$, since we count $L(CT_1^{app})$ only once.

Because S may be too big to store, we store $H(S)$. To weigh both code tables equally, we approximate the inequality as follows in the definition of a split.

Definition 12. *Let the, finite, stream $S = S_1 S_2$ such that S_i is sampled from distribution P_i. Moreover, let CT_i^{app} denote the approximated optimal code table for S_i. The pair $(S1, S2)$ is called a* split *of S if*

$$L(H(S_2), CT_2^{app}) < L(H(S_2), CT_1^{app}).$$

A split is called minimal *if there is no other split (T_1, T_2) of S such that T_1 is a substream of S_1.*

Note that this definition implies that we do not have to store $H(S_1)$ to detect a change in the underlying definition. CT_1^{app} provides sufficient information.

The Problem for Data Streams with Bounded Storage

We can now formalise our problem for data streams with bounded storage.

Problem 4 (Data Stream Minimal Split Partitioning)**.** *Let S be a data stream, partition S into consecutive substreams $S_1, \ldots, S_k, \ldots$, such that*

$$\forall S_i : (S_i, S_{i+1}) \textit{ is the } \text{minimal split} \textit{ of } S_i S_{i+1}.$$

5.4 The Algorithm

Finding the Right Code Table on a Stream

We can now translate the formal scheme presented in Section 5.3 to a practical implementation: assume that the stream S is sampled i.i.d. from one distribution only and find CT^{app} using Krimp.

Algorithm 7 Find Code Table on Stream

```
FindCodeTableOnStream($S, offset, blocksize, minsup, maxIR$) :
1.  $numTransactions \leftarrow blockSize$
2.  $CT \leftarrow$ Krimp($S(offset, offset + numTransactions), minsup$)
3.  $ir \leftarrow \infty$
4.  while $ir > maxIR$ do
5.      $numTransactions \leftarrow numTransactions + blockSize$
6.      $newCT \leftarrow$ Krimp($S(offset, offset + numTransactions), minsup$)
7.      $ir \leftarrow$ ImprovementRate($CT, newCT$)
8.      $CT \leftarrow newCT$
9.  end while
10. return $CT$
```

The general idea of the algorithm presented in Algorithm 7 is simple: run Krimp on the growing head of a stream S until the resulting code tables converge. As we know that individual transactions don't make a large difference, we work with blocks of *blockSize* transactions. Start with one block and obtain a code table. For each block added to the head, a new code table is induced and the Improvement Rate is computed. Whenever the IR drops below *maxIR*, the code table is good enough and returned.

The other parameters are an offset that makes it possible to start anywhere within the stream and the *minsup* used for mining Krimp candidates.

Moreover, a Laplace correction is applied to each code table returned by Krimp; this to ensure that each code table can encode each possible transaction.

Detecting Change in a Stream

Given a code table induced on the head of a data stream, we would now like to detect change in the sampling distribution of the rest of the stream. More formally, we would like to detect the minimal split given CT_1^{app}.

The minimal split can be found by inducing code tables on consecutive heads of the stream until a split is encountered. We would rather avoid building a code table for each and every consecutive head, but luckily we can speed things up in two different ways. First of all, change does not come in a single transaction, so again we iterate over blocks instead. Secondly, we can skip each block that obviously belongs to CT_1^{app}.

For this second optimisation, we apply a statistical test that tests whether the encoded size of the current block deviates from the expected size. If it does not, discard it and skip to the next block. Before discarding the head of a converged code table, this data is used to randomly sample encoded block sizes. Both the lower and upper *leaveOut* percent samples are removed. If the encoded size of a new block falls within the range of the remaining samples, the block is considered to belong to the distribution of CT_1^{app} and skipped.

For each block that is not skipped, we have to test whether it marks a split or not. For this, we have to induce a code table CT_2^{app}. To be able to reject a code table that is only just better than the previous one, we introduce the *Code Table Difference*:

Definition 13. *Given a code table* CT_1^{app} *and a code table* CT_2^{app} *induced on* $H(S_2)$*, the* Code Table Difference CTD *is given by*

$$\frac{L(H(S_2), CT_1^{app}) - L(H(S_2), CT_2^{app})}{L(H(S_2), CT_2^{app})}.$$

Normalised the same way as the *Improvement Rate*, the CTD tells us how many percent CT_2^{app} compresses the new head better than CT_1^{app}. We can now define a minimum CTD in the overall algorithm, which is presented next.

StreamKrimp

Putting together the algorithms of the previous subsections, we are able to partition a stream into consecutive substreams with minimal splits. The complete algorithm is shown in Algorithm 8.

It starts with finding the code table on the head of the stream (line 2) and then iterates over the rest of the stream. Each iteration starts with skipping as many blocks as possible (5). When a block cannot be skipped straightaway, it is used as starting position for a new candidate code table (6). The Code Table Difference of this candidate to the current code table is computed (7) and the code table is either accepted (8–10) or rejected (12). When finished, the complete set of code tables is returned (13). Naturally, these could be inspected and used while the algorithm is still running as well.

How Large is Zero?

Or: How should we set our parameters? We will here motivate the default values we suggest for the algorithm, which we will use throughout the rest of the chapter.

minsup Lower *minsup* levels result in more candidate patterns and therefore better compression and better quality code tables. Sensitivity of the change detection scheme is influenced through this parameter: lower values result in a higher sensitivity. To avoid overfitting on very small data

Algorithm 8 StreamKrimp

```
StreamKrimp($S, minsup, blockSize, maxIR, leaveOut, minCTD$) :
1.  $i \leftarrow 1$
2.  $CTS_i \leftarrow$ FindCodeTableOnStream($S, 0, blockSize, minsup, maxIR$)
3.  $pos \leftarrow CTS_i$.endPos
4.  while $pos <$ sizeof($S$) do
5.    $pos \leftarrow$ SkipBlocks($S, CTS_i, pos, blockSize, leaveOut$)
6.    $candCT \leftarrow$ FindCTOnStream($S, pos, blockSize, minsup, maxIR$)
7.    if CTD($S, CTs_i, candCT$) $>= minCTD$ then
8.      $i \leftarrow i + 1$
9.      $CTS_i \leftarrow candCT$
10.     $pos \leftarrow candCT$.endPos
11.   else
12.     $pos \leftarrow blockSize$
13.   end if
14. end while
15. return $CTS$
```

segments, we use a *minsup* of 20 in the experiments presented in this chapter.

blockSize The resolution at which StreamKrimp works should always be high enough. This will be the case if we choose the size of a block such that, on average, every possible item occurs once in every block. Therefore, we choose *blockSize* to be equal to $|\mathcal{I}|$.

leaveOut Set to 0.01: both the lower 1% and the upper 1% of the randomly sampled blocksizes are discarded by SkipBlocks.

maxIR Set to 0.02: if a new code table compresses less than 2% better than its predecessor, we decide it has converged.

minCTD Set to 0.10: a new code table is accepted only if it compresses at least 10% better than the previous code table.

The choices for *maxIR* and *minCTD* may seem arbitrary, but this is not the case. They are actually comparable to the dissimilarity values we reported before (see Chapter 3). Dissimilarity values between random samples from a single dataset range from 0.045 to 0.177 on UCI datasets (also reported on in the next section).

Table 5.1: Properties of 7 UCI datasets.

Dataset	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{C}\rvert$	$\lvert\mathcal{I}\rvert$
Adult	48842	2	97
Chess (kr–k)	28056	18	58
Led7	3200	10	24
Letter recognition	20000	26	102
Mushroom	8124	2	119
Nursery	12960	5	32
Pen digits	10992	10	86

For each dataset, given are the number of transactions, classes and items.

Therefore, 0.02 and 0.10 are very conservative and may be considered zero for all practical purposes: with these thresholds, code tables converge and significant changes are detected.

5.5 Experiments

Artificial Streams – UCI Datasets

The first series of experiments is done on a selection of the largest datasets from the well-known UCI repository [32], as shown in Table 5.1. These datasets are transaction databases and not streams, but they have the advantage that each of them consists of multiple known classes. This allows for easy validation of the identified splits.

To transform a UCI dataset into a stream, each dataset is split on class label and the class labels are removed from all transactions. This results in a transaction database per class. The transactions within each database are ordered (randomly) to form a stream. The resulting streams are concatenated into one single stream (in random order). Because of this randomisation, each dataset is turned into a stream 10 times.

The main results are summarised in Table 5.2. If we compare the number of code tables found to the actual number of classes in Table 5.1, we see that StreamKrimp finds the right number of distributions in the stream. Only for *Chess*, the algorithm doesn't find enough splits, but this is not surprising as there are quite many classes and some of them are rather small. Analysing the splits reveals that indeed the larger classes are identified and only the smallest ones go undetected.

Table 5.2: Results for 7 UCI datasets.

		Blocks	*#CTs*	*Blocks*	Purity	
Dataset	$\lvert CTS \rvert$	*per CT*	*rejected*	*skipped*	*Base*	*Actual*
Adult	3.4	4.7	118	367	76.1%	99.7%
Chess (kr–k)	13	4.2	165	264	17.8%	80.1%
Led7	12	3.9	3.5	82	10.9%	95.2%
Letter recognition	27	5.9	0.2	32	4.1%	80.1%
Mushroom[2]	2.7	6.2	6.2	44	51.7%	96.5%
Nursery	6.2	5.7	140	228	33.3%	98.5%
Pen digits	15	6.0	2.3	34	10.4%	87.2%

For each dataset, the following is listed: the number of code tables found, the average number of blocks used for construction per CT, the number of code tables rejected, the number of blocks skipped and finally base and obtained purity. Averages over 10 randomisations (class order, transaction order within classes).

The next column tells us that approximately 4 to 6 blocks are enough for code table construction on these datasets. For some datasets, such as *Adult*, *Chess* and *Nursery*, quite some code tables are rejected, as is shown under '#CTs rejected'. However, also quite some blocks are skipped by SkipBlocks. These values vary quite a bit for the different datasets, telling us that the conservative statistical skip test seems to work better for one dataset than for another.

The last columns show baseline and obtained purity values. Purity is the size of the majority class relative to the entire segment, averaged over all identified segments. Baseline purity is the size of the largest class. Although the transactions of the classes are not interleaved and the task is therefore easier than clustering, the attained purity values are very high. This indicates that the algorithm correctly identifies the boundaries of the classes in the streams. This is supported by Figure 5.1, which depicts actual and found splits for three datasets. No expected splits are missed by StreamKrimp and the shift moments are accurately detected. (For each dataset, a single run with average performance is shown.)

Artificial Streams – Synthetic

The experiments in the previous subsection show that the proposed algorithm accurately detects changes in a stream. The objective of the following experi-

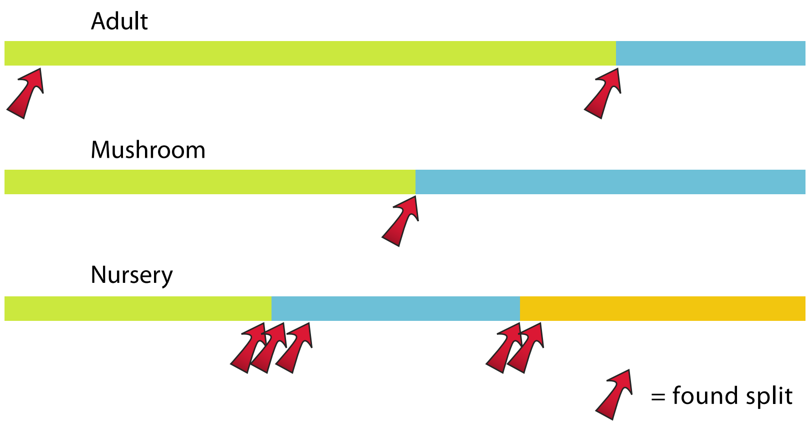


Figure 5.1: Actual and found (marked with arrows) splits for three datasets.

ments is simple: which elementary distribution changes are detected?

We manually create simple code tables and use the KRIMP data generator (see Chapter 4) to generate a series of synthetic datasets. Each generated stream consists of two parts: 5000 rows generated with one code table, followed by 5000 rows generated by a variation on this code table. In these experiments, $|\mathcal{I}|$ is 10 and each item is in the code table with a count of 1 (i.e. all individual items are generated with equal probability). The databases have 5 two-valued attributes (resulting in a total of 10 possible items). So, each transaction consists of 5 items.

Because the number of different items is very small (only 10) we manually set the *blockSize* for these experiments to 200. This way, we ensure that KRIMP gets enough data and candidate patterns to learn the structure that is in the data.

The basic distribution changes made halfway in the synthetic datasets (through changing the code tables) are depicted in Figure 5.2. Each rounded box represents an itemset, with the individual items given as numbers. All usage counts are set to 5, except for those itemsets where a multiplier is shown ($\times 4$ means $5 \times 4 = 20$). As data generation is a stochastic process, 10 streams were generated for each change and the main results shown in Table 5.3 are averaged over these.

The last column indicates the number of times the algorithm found the optimal solution; two code tables and 100% purity (i.e. a split after 5000 rows). From the results it is clear that STREAMKRIMP is very capable at detecting 4

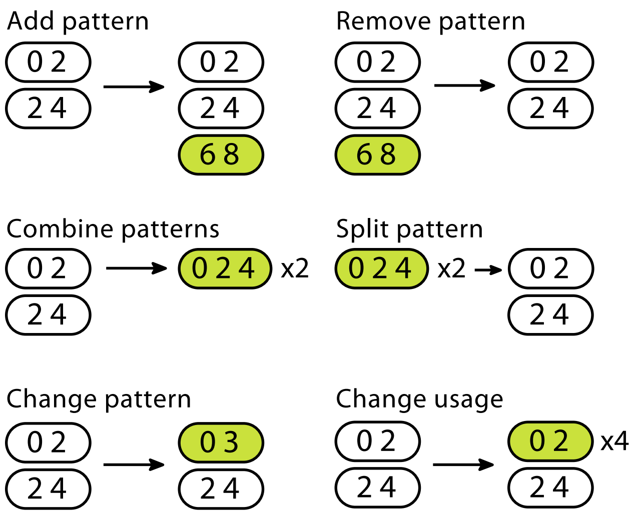


Figure 5.2: Changes inflicted in the code tables used for stream generation.

Table 5.3: Results for 6 synthetic streams.

Change	$\|CTS\|$	*#CTs rejected*	*Purity*	$\|Optimal\|$
Add pattern	1.0	1.8	79.8	0
Remove pattern	1.3	3.8	97.0	6
Combine patterns	1.3	2.8	98.6	6
Split pattern	1.1	2.6	98.8	8
Change pattern	1.1	1.8	98.6	7
Change usage	1.9	15.6	75.2	1

For each dataset, the following is listed: the number of code tables found, the number of code tables rejected, obtained purity and the number of optimal solutions found. Averages over 10 generated streams (except for the last column).

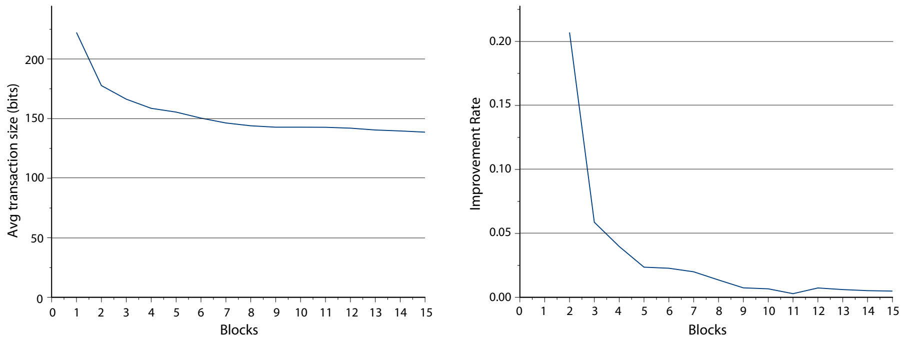


Figure 5.3: Average encoded length per transaction (left) and improvement rates (right) for the code tables built on 15 consecutive blocks from the *Accidents* dataset.

out of 6 of the tested types of change. Only detection of the subtle addition of a single (small) pattern and changing the frequency of an existing pattern turns out to be difficult occasionally. In these cases, change is often detected but this takes some blocks, resulting in lower purity values.

A Real Stream - Accidents Dataset

A more realistic dataset is taken from Geurts *et al.* [43]. It contains data obtained from the National Institute of Statistics (NIS) for the region of Flanders (Belgium) for the period 1991-2000. More specifically, the data are obtained from the Belgian 'Analysis Form for Traffic Accidents' that should be filled out by a police officer for each traffic accident that occurs with injured or deadly wounded casualties on a public road in Belgium. In total, 340,184 traffic accident records are included in the data set.

No timestamps are available, but accidents are ordered on time and it is an interesting question whether structural changes can be detected. With over 340,000 transactions over a large number of items ($|\mathcal{I}| = 468$), running any regular pattern mining algorithm on the entire dataset is a challenge. Therefore, it is a perfect target for finding 'good enough' code tables and detecting change. As Krimp candidates we use closed frequent itemsets with minimum support 500 and we rounded the block size to 500.

An important question we have not yet addressed with the (much smaller) artificial streams is how well the FindCodeTableOnStream algorithm approximates the best possible code table on a stream. To assess this, the average encoded size per transaction is plotted for a series of code tables in Figure 5.3 on the left. On the right, Figure 5.3 shows the computed Improvement Rates for the same set of code tables. Each code table is built on a head of x blocks, where x is the number of blocks indicated on the x-axis. Average encoded size is computed on all transactions the code table is induced from. The graphs clearly show that the most gain in compression is obtained in the first few blocks. After that, the average size per transaction decreases only slowly and this is also reflected in the Improvement Rate. With *maxIR* set to 0.02, StreamKrimp would pick the code table built on 8 blocks, which seems a good choice: after that, improvement is marginal.

Running StreamKrimp on the entire dataset resulted in only 14 code tables that characterise the entire stream of over 340,000 transactions. 140 blocks were skipped, 429 code tables were built but rejected. On average, 7.43 blocks of data were required for a code table to converge and the average Code Table Difference of accepted code tables was 0.14. This means that each consecutive distribution differs about 14% from its predecessor in terms of compression!

To further illustrate the differences between the identified substreams, Figure 5.4 shows compressed block sizes over the entire dataset for three consecutive code tables. Substreams clearly consist of blocks that are equally well compressed. The split marking the end of the last substream shown seems to be a tad late, the rest is spot on. In other words, the change detection is both quick and accurate, also on large datasets.

5.6 Related Work

Stream mining has attracted a lot of attention lately, which is nicely illustrated by the recent book by Aggarwal *et al.* [3]. Here, we focus on change detection.

In many cases, streams are considered to be sequences of single (real-valued) numbers or items. Kifer *et al.* [59] use two sliding windows to obtain two samples of which it is statistically determined whether they belong to different distributions. Papadimitriou *et al.* [90] use autoregressive modelling and wavelets, Muthukrishnan *et al.* [88] avoid a fixed window size by introducing a method based on sequential hypothesis testing.

A second class of stream mining algorithms considers multi-dimensional, real-valued streams. Aggarwal *et al.* [1] visualise evolving streams using velocity density estimation. The visualisation is inherently 2-dimensional and it is not possible to accurately estimate densities with increasing dimensionality. Aggarwal *et al.* [2] use a polynomial regression technique to compute statistically expected values.

Dasu *et al.* [36] take an information-theoretic approach by using the Kullback-

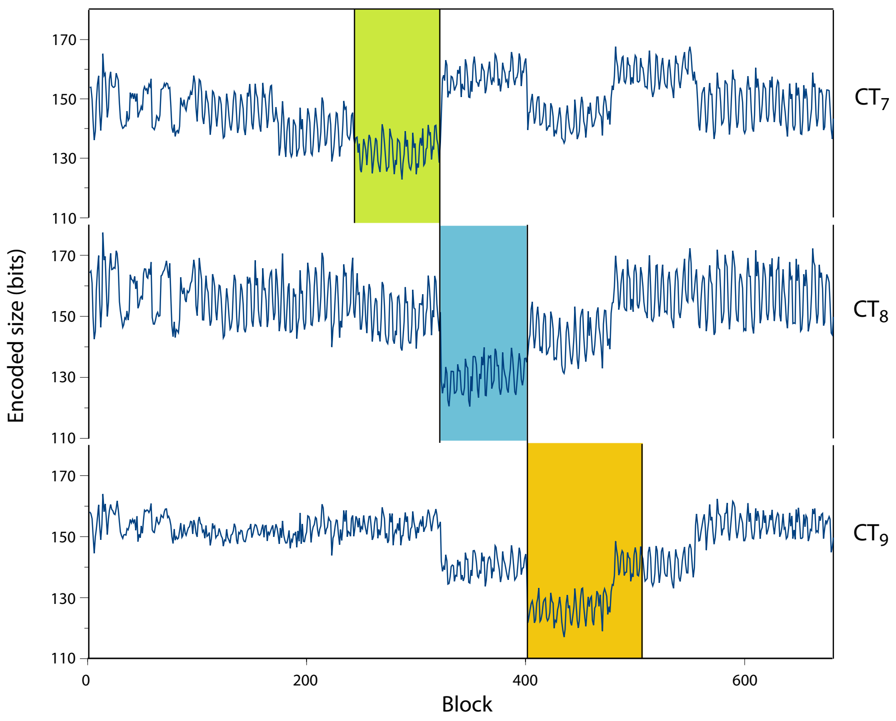


Figure 5.4: Encoded length per block for three consecutive substreams on *Accidents*. The blocks belonging to each of the code tables are indicated with the coloured blocks.

Leibler distance to measure the difference between distributions. They experiment on multi-dimensional real-valued data, but claim the method can also be applied to categorical data. However, a fixed window size strongly influences the changes that can be detected and the method seems better suited for relatively few dimensions (< 10).

Widmer & Kubat [118] use an adaptive window size to do online learning in domains with concept drift. Predictive accuracy is used to detect drift and adjust the window size heuristically. This does require (known) binary class labels though.

The final class of algorithms considers streams of categorical transactions, as we do in this chapter. Chen *et al.* [29] propose a method to visualise changes in the clustering structure of such streams. A disadvantage is that snapshots of these visualisations have to be manually analysed. More recently, Calders *et al.* [23] proposed an alternative 'minimum support' measure for patterns in streams called *max-frequency*. This measure uses flexible windows to maintain the max-frequency on patterns in the stream.

5.7 Discussion

The results on both the artificial and realistic streams show that StreamKrimp is very capable at detecting changes in large data streams. No actual splits are missed and the results on the synthetic streams show that even small modifications in the distribution can be detected.

The algorithm satisfies the general requirements for stream mining, as only very limited data storage is required and online mining is possible. Also, the resulting code tables are much smaller than the data itself and can therefore be stored for a much longer time. This means that it is possible to store a full characterisation of the entire stream.

In many stream mining algorithms, a window size has to be defined. This window size determines what changes can and can not be found; nothing outside the window is seen. Contrary, the block size of our algorithm is only the resolution which determines how quickly a distribution is detected and characterised.

5.8 Conclusions

We introduce StreamKrimp, an algorithm that detects changes in the sampling distribution of a stream of transactions. Based on an analysis from MDL perspective, it partitions a stream into a sequence of substreams. For each substream, it uses Krimp to characterise its distribution with a code table and each subsequent code table indicates a change in the underlying distribution. Only a very limited amount of data storage is required and StreamKrimp facilitates online mining of streams.

The results of experiments on both artificial and realistic streams show that StreamKrimp detects the changes that make a difference, no relevant changes are missed and noise is neglected. Finally, large streams with many attributes pose no problems.

Chapter 6

Identifying the Components

Most, if not all, databases are mixtures of samples from different distributions[1]. Transactional data is no exception. For the prototypical example, supermarket basket analysis, one also expects a mixture of different buying patterns. Households of retired people buy different collections of items than households with young children.

Models that take such underlying distributions into account are in general superior to those that do not. In this chapter we introduce two MDL-based algorithms that follow orthogonal approaches to identify the components in a transaction database. The first follows a model-based approach, while the second is data-driven. Both are parameter-free: the number of components and the components themselves are chosen such that the combined complexity of data and models is minimised. Further, neither prior knowledge on the distributions nor a distance metric on the data is required. Experiments with both methods show that highly characteristic components are identified.

[1]This chapter has been published as [71]: M. van Leeuwen, J. Vreeken & A. Siebes. Identifying the Components. In *Data Mining and Knowledge Discovery* 19(2), Springer. ECML PKDD'09 special issue, Best Student Paper.

6.1 Introduction

Most, if not all, databases are mixtures of samples from different distributions. In many cases, nothing is known about the source components of these mixtures and therefore many methods that induce models regard a database as sampled from a single data distribution. While this greatly simplifies matters, it has the disadvantage that it results in suboptimal models.

Models that do take into account that databases actually are sampled from mixtures of distributions are often superior to those that do not, independent of whether this is modelled explicitly or implicitly. A well-known example of explicit modelling is mixture modelling [108]. This statistical approach models data by finding combinations of several well-known distributions (e.g. Gaussian or Bernoulli) that are assumed to underlie the data.

Boosting algorithms [41] are a good example of implicit modelling of multiple underlying distributions. The principle is that a set of weak learners can together form a single strong learner. Each learner can adapt to a specific part of the data, implicitly allowing for modelling of multiple distributions.

Transaction databases are no different with regard to data distribution. As an illustrative example, consider supermarket basket analysis. One also expects a mixture of different buying patterns: different groups of people buy different collections of items, although overlap may exist. By extracting both the groups of people and their corresponding buying patterns, a company can learn a lot about its customers.

Part of this problem is addressed by clustering algorithms, as these group data points together, often based on some distance metric. However, since we do not know upfront what distinguishes the different groups, it is hard to define the appropriate distance metric. Furthermore, clustering algorithms such as k-means [79] only find the groups of people and do not give any insight in their corresponding buying behaviours. The only exception is bi-clustering [92], however this approach imposes strong restrictions on the possible clusters.

Frequent itemset mining, on the other hand, does give insight into what items customers tend to buy together, e.g. the (in)famous Beer and Nappies example. But, not all customers tend to buy Nappies with their Beer and standard frequent set mining does not distinguish groups of people. Clearly, knowing several groups that collectively form the client base as well as their buying patterns would provide more insight. So, the question is: can we find these groups and their buying behaviour? That is, we want to partition the database $\mathcal{D}$ in sub-databases $\mathcal{D}_1, \cdots, \mathcal{D}_k$ such that

- the buying behaviour of each $\mathcal{D}_i$ is different from all $\mathcal{D}_j$ (with $j \neq i$), and
- each $\mathcal{D}_i$ itself is homogeneous with regard to buying behaviour.

But, what does 'different buying behaviour' mean? It certainly does not mean that the different groups should buy completely different sets of items. Also, it does not mean that these groups cannot have common frequent itemsets. Rather, it means that the characteristics of the sampled distributions are different. This may seem like a play of words, but it is not. Sampled distributions of transaction data can be characterised precisely through the use of code tables.

We use Krimp code tables and MDL to formalise our problem statement. That is: find a partitioning of the database such that the total compressed size of the components is minimised. This problem statement agrees with the compression perspective on data mining as recently positioned by Faloutsos *et al.* [40].

We propose two orthogonal methods to solve the problem at hand. The first method is based on the assumption that Krimp implicitly models all underlying distributions in a single code table. If this is the case, it should be possible to extract these and model them explicitly. We propose an algorithm to this end that optimises the compressed total size by specialising copies of the original compressor to the different partial distributions of the data.

Throughout our research we observed that by partitioning a database, e.g. on class label, compressors are obtained that encode transactions from their 'native' distribution shortest. This observation suggests an iterative algorithm that resembles k-means: without any prior knowledge, randomly split the data in a fixed number of parts, induce a compressor for each and re-assign each transaction to the compressor that encodes it shortest, etcetera. This scheme is very generic and could also be easily used with other types of data and compressors. Here, as we are concerned with transaction data, we employ Krimp as compressor.

Both algorithms are implemented and evaluated on the basis of total compressed sizes and component purity, but we also look at (dis)similarities between the components and the code tables. The results show that both our orthogonal methods identify the components of the database, without parameters: the optimal number of components is determined by MDL. Visual inspection confirms that characteristic decompositions are identified.

6.2 Problem Statement

Our 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.

As discussed in previous chapters, code tables characterise the sampled distributions of the data. Hence, we want a partitioning for which the different components have different characteristics, and thus code tables. In terms of MDL, this is stated as follows.

Problem 5 (Identifying Database Components)**.** *Let $\mathcal{I}$ be a set of items and let $\mathcal{D}$ be a bag of transactions over $\mathcal{I}$. Find a partitioning $\mathcal{D}_1, \cdots, \mathcal{D}_k$ of $\mathcal{D}$ and a set of associated code tables $CT_1, \cdots, CT_k$, such that* the total compressed size *of $\mathcal{D}$,*

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

is minimised.

There are a few things one should note about this statement. First, we let MDL determine the optimal number of components for us, by picking the smallest encoded size over every possible partitioning and all possible code tables. Note that this also ensures that we will not end up with two components that have the same or highly similar code tables. It would be far cheaper to combine these.

Secondly, asking for both the database partitioning and the code tables is in a sense redundant. For any partitioning, the best associated code tables are, of course, the optimal code tables. The other way around, given a set of code tables, a database partitions naturally. Each transaction goes to the code table that compresses it best. This suggests two ways to design an algorithm to solve the problem. Either one tries to find an optimal partitioning or one tries to find an optimal set of code tables.

Thirdly, the size of the search space is gigantic. The number of possible partitions of a set of n elements is the well-known Bell number B_n [98]. Similarly, the number of possible code tables is enormous. It consists of the set of all sets of subsets of that contain at least the singletons. Further, for each of these code tables we would have to consider all possible combinations of covers per transaction. Moreover, there is no structure in this search space that we can use to prune. Hence, we will have to introduce heuristic algorithms to solve our problem.

6.3 Datasets

We use a number of UCI datasets [32], the *Retail* dataset [20] and the *Mammals* dataset [51][2] for experimental validation of our methods. The latter consists of presence/absence records of 121 European mammals in geographical areas of 50×50 kilometres. The properties of these datasets are listed in Table 6.1.

[2]The full dataset [86] is available for research purposes from the Societas Europaea Mammalogica, `http://www.european-mammals.org`

Table 6.1: Basic statistics and KRIMP statistics of the datasets used in the experiments.

	Basic Statistics				KRIMP			
Dataset	$\|\mathcal{D}\|$	$\|\mathcal{I}\|$	$\|C\|$	*Pur*	*Cand*	*ms*	$L(CT, \mathcal{D})$	*DS*
Adult	48842	95	2	76.1	Cls	20	841604	0.8
Anneal	898	65	6	76.2	All	1	21559	6.3
Chess (kr–k)	28056	40	18	16.2	All	10	516623	1.3
Mammals	2183	121	-	-	Cls	150	164912	-
Mushroom	8124	117	2	51.8	Cls	1	272600	8.4
Nursery	12960	21	2	33.3	Cls	1	240537	4.2
Page blocks	5473	33	11	89.8	All	1	7160	10.6
Retail	88162	16470	-	-	All	16	10101135	-

Statistics on the datasets used in the experiments. As basic statistics, provided are the number of transactions, number of items, number of classes and purity (the accuracy by majority class voting). KRIMP-specific are the candidates used (all or closed frequent itemsets), the *minsup* at which the candidates are mined, the total compressed size in bits and the average code table dissimilarity between the classes.

For fair comparison between the found components and the original classes, the class labels are removed from the databases in our experiments. In addition, KRIMP candidates (all or closed frequent itemsets up to *minsup*), the regular (single-component) KRIMP compressed size of the database and class dissimilarities are given. Average class dissimilarities are computed by splitting the database on class label, computing pair-wise code table dissimilarities as defined above and taking the average over these. The *Retail* and *Mammals* datasets do not contain class labels.

6.4 Model-Driven Component Identification

In this section we present an algorithm that identifies components by finding an optimal set of code tables.

Motivation

A code table induced for a complete database captures the entire distribution, so the multiple underlying component distributions are modelled implicitly. This suggests that we should be able to extract code tables for specific components from the code table induced on the whole database, the original code table.

If the data in a database is a mixture of several underlying data distributions, the original code table can be regarded as a mixture of code tables. This implies that all patterns required for the components are in the code table and we only need to 'extract' the individual components. In this context, each possible subset of the original code table is a candidate component and thus each set of subsets a possible decomposition. So, given an original code table CT induced for a database $\mathcal{D}$, the optimal model driven decomposition is the set of subsets of CT that minimises the total encoded size of $\mathcal{D}$.

Obviously, the optimal decomposition could be found by simply trying all possible decompositions. However, although code tables consist of only few patterns (typically hundreds), the search space for such approach is enormous. Allowing only partitions of the code table would strongly reduce the size of the search space. But, as different distributions may have overlapping common elements, this is not a good idea. The solution is therefore to apply a heuristic.

Algorithm

The algorithm, presented in detail in pseudo code in Algorithm 9, works as follows: first, obtain the overall code table by applying Krimp to the entire database (line 1). Then, for all possible values of k, i.e. $[1, |\mathcal{D}|]$, identify the best k components. We return the solution that minimises the total encoded size (2-3).

To identify k components, start with k copies of the original code table (5). A Laplace correction of 1 is applied to each copy, meaning that the usages of all itemsets in the code table are increased by one. This correction ensures that each code table can encode any transaction, as no itemsets with zero usage (and therefore no code) can occur. Now, iteratively eliminate that code table element that reduces the total compressed size most; until compression cannot be improved any further (6-8). Iteratively, each element in each code table is temporarily removed to determine the best possible elimination (10). To compute the total compressed size, each transaction is assigned to that code table that compresses it best (12). This can be translated as being the Bayes optimal choice (see Section 2.5). After this, re-compute optimal code lengths for each component (usages may have changed) and compute the total encoded size by summing the sizes of the individual components (13-14).

Algorithm 9 Model-Driven Component Identification

```
IdentifyTheComponentsByModel($\mathcal{D}$, $minsup$) :
1.  $CT_{orig} \leftarrow$ Krimp($\mathcal{D}$, MineFreqSets($\mathcal{D}, minsup$))
2.  $b \leftarrow \operatorname{argmin}_{k \in [1,|\mathcal{D}|]}$ CalcEncSize(IdKComponentsByModel($\mathcal{D}, CT_{orig}, k$))
3.  return IdentifyKComponentsByModel($\mathcal{D}, CT_{orig}, b$)

IdKComponentsByModel($\mathcal{D}, CT_{orig}, k$) :
5.  $C \leftarrow \{$ $k$ Laplace corrected copies of $CT_{orig}$ $\}$
6.  do $e \leftarrow$ FindBestElimination($\mathcal{D}, C, k$)
7.     $C \leftarrow$ ApplyElimination($C, e$)
8.  while compressed size decreases
9.  return $C$

FindBestElimination($\mathcal{D}, C, k$) :
10. $(C_b, B) \leftarrow \operatorname{argmin}_{C_i \in C, X \in C_i}$ CalcEncSize($\mathcal{D}, (C \setminus C_i) \cup (C_i \setminus X), k$)
11. return $(C_b, B)$

CalcEncSize($\mathcal{D}, C, k$) :
12. for each transaction $t \in \mathcal{D}$ : assign $t$ to $CT_i \in C$ that gives shortest code
13. for each $CT_i \in C$ : ComputeOptimalCodes($\mathcal{D}_i, CT_i$)
14. return $\sum_i L(C_i, \mathcal{D}_i)$
```

Experimental Results

The results of the experiments with the algorithm just described are summarised in Table 6.2. By running IdentifyTheComponentsByModel for all values of k and choosing the smallest decomposition, MDL identifies the optimal number of components. However, the search space this algorithm considers grows enormously for large values of k, so in our experiments we imposed a maximum value for k.

The compression gain is the reduction in compressed size relative to that attained by the original code table; the regular Krimp compressed size as given in Table 6.1. The compression gains in Table 6.2 show that the algorithm ably finds decompositions that allow for a much better compression than when the data is considered as a single component. By partitioning the data, reductions from 9% up to 46% are obtained. In other words, for all datasets compression improves by using multiple components.

We define purity as the weighted sum of individual component purities, a measure commonly used in clustering. Hence, the baseline purity is the percentage of transactions belonging to the majority class. If we look at the

Table 6.2: Experimental results for Model-Driven Component Identification

	Model-Driven Component Identification				
Dataset	*Gain L%*	*Comp.Purity (%)*	*Avg DS*	*Opt k*	*Max k*
Adult	8.7	76.1	6.44	2	2
Anneal	18.1	80.8	5.96	19	30
Chess (kr–k)	14.5	18.2	2.86	6	7
Mushroom	25.7	88.2	11.32	12	15
Nursery	11.7	45.0	1.77	14	20
Page blocks	46.4	91.5	804.9	6	40

Experimental results for Model-Driven Component Identification. Given are the gain in compression over the single component Krimp compression, component purity by majority class voting, average dissimilarity between the components, the optimal value of k and the maximum value of k.

obtained purities listed in Table 6.2 and compare these to the baseline values in Table 6.1, we notice that these values range from baseline to very good. Especially the classes of the *Mushroom* and *Page blocks* datasets get separated well. The average (pairwise) dissimilarity between the Optimal k components, *Avg DS*, shows how much the components differ from each other. We obtain average dissimilarities ranging from 1.7 to 804.9, which are huge considering the values measured between the actual classes (as given in Table 6.1). Without any prior knowledge the algorithm identifies components that are at least as different as the actual classes. While for the *Adult* database the obtained purity is at baseline, the data is very well partitioned: the dissimilarity between the components measures 6.4, opposed to just 0.8 between the actual classes.

Arguably, the most important part of the results is the code tables that form the end product of the algorithm: these provide the characterisation of the components. Inspection of these provides two main observations. First, a number of the original elements occur in multiple components, others occur only in a single component and some lost their use. Secondly, the code lengths of the elements are adapted to the specific components: codes become shorter, but lengths also change relative to each other. Example original and resulting code tables are depicted in Figure 6.1.

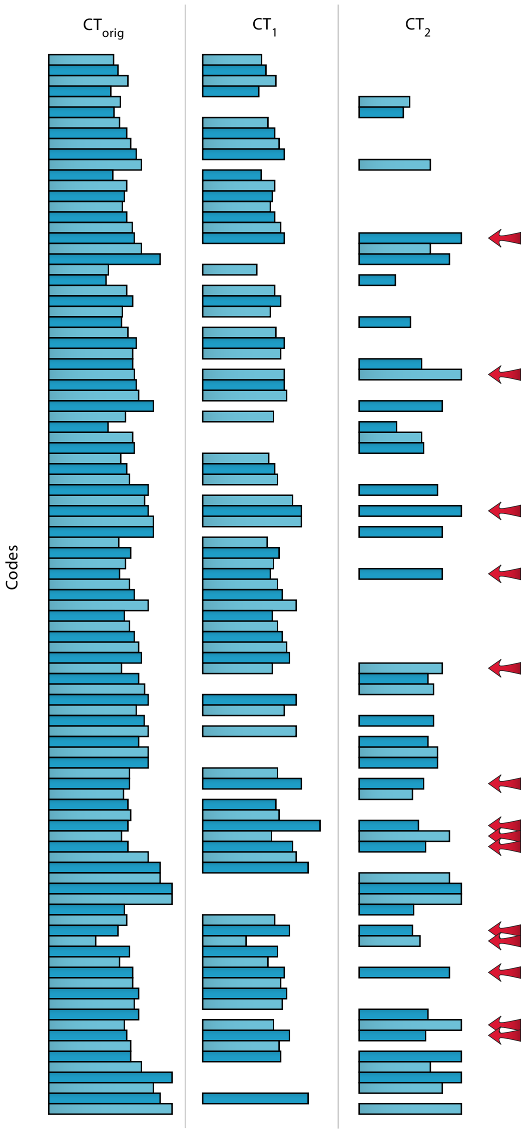


Figure 6.1: The components of *Anneal*. Codes and their lengths (in bits, represented by width) for original and component code tables, $k = 2$. Common elements are marked by arrows.

Discussion

The significantly smaller total encoded sizes after decomposition show that the proposed algorithm extracts different underlying distributions from the mixture. The purities and component dissimilarities show that components are different from each other but homogeneous within themselves.

One of the properties of the method is that the number of (unique) patterns required to define the components is never higher than the number of itemsets in the original code table, which consists of only few patterns. As the total number of patterns actually used in defining the components is often even smaller, the resulting code tables can realistically be manually inspected and interpreted by domain experts.

The runtimes for the reported experiments ranged from a few minutes for *Anneal* up to 80 hours for *Adult*. Obviously, more computation time is needed for very dense and/or very large databases, which is why in this section no results are presented for *Retail* and *Mammals*. While parallelisation of the algorithm is trivial and should provide a significant speedup, there is another approach to handle large datasets: the data driven method presented in the next section is inherently much faster.

6.5 Data-Driven Component Identification

In this section, we present an approach to identify the components of a dataset by finding the MDL-optimal partitioning of the data.

Motivation

Suppose we have two equally sized databases, $\mathcal{D}_1$ and $\mathcal{D}_2$, both drawn from a mixture of distributions D_A and D_B. Further, suppose that $\mathcal{D}_1$ has more transactions from D_A than from D_B and vice versa for $\mathcal{D}_2$. Now, we induce compressors C_1 and C_2 for $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively. Assuming that D_A and D_B are different, C_1 will encode transactions from distribution D_A shorter than C_2, as C_1 has seen more samples from this distribution than C_2. This bias will be stronger if more transactions come from a single distribution - and the strongest when the data consists solely of transactions from one distribution. So, provided a particular source distribution has more transactions in one database than in another, transactions of that distribution will be encoded shorter by a compressor induced on that data.

We can exploit this property to find the components of a database. Given a partitioning of the data, we can induce a code table for each part. Now, we simply reassign each transaction to that part whose corresponding code table encodes it shortest. We thus re-employ the Bayes optimal choice to group

Algorithm 10 Data-Driven Component Identification

```
IdentifyTheComponentsByData($\mathcal{D}$, $minsup$)
1.  for $k = 2$ to $|\mathcal{D}|$ : $r_k \leftarrow$ IdentifyKComponentsByData($\mathcal{D}, k, minsup$)
2.  $best \leftarrow \operatorname{argmin}_{k \in [1,|\mathcal{D}|]}$ CalcEncSize($\mathcal{D}, r_k, k$)
3.  return $r_{best}$

IdentifyKComponentsByData($\mathcal{D}$, $k$, $minsup$)
4.  $parts \leftarrow$ Partition(RandomizeOrder($\mathcal{D}$), $k$)
5.  do
6.     for each $p_i \in parts$ :
7.        $CT_i \leftarrow$ Krimp($p_i$,MineFreqSets($p_i, minsup$))
8.        for each $X \in CT_i$ : $usage_{\mathcal{D}}(X) \leftarrow usage_{\mathcal{D}}(X) + 1$
9.     end for
10.    for each transaction $t \in \mathcal{D}$ : assign $t$ to $CT_i$ that gives the shortest code
11. while transactions were swapped
12. return $parts$
```

those transactions that belong to similar distribution(s) (see Section 2.5). By doing this iteratively until all transactions remain in the same part, we can identify the components of the database. This method can be seen as a form of k-means; without the need for a distance metric and providing insight in the characteristics of the found groupings through the code tables.

Algorithm

From the previous subsection it is clear what the algorithm will look like, but its initialisation remains an open question. Without prior background knowledge, we start with a random initial partitioning. Because of this, it is required to do a series of runs for each experiment and let MDL pick the best result. However, as transactions are reassigned to better fitting components already in the first iteration, it is expected that the algorithm will be robust despite this initial non-deterministic step. The algorithm is presented in pseudo-code as Algorithm 10.

Experiments

Each experiment was repeated 10 times, each run randomly initialised. The different runs resulted in almost the same output, indicating that random initialisation does not harm robustness. In Table 6.3 we report the main results of

Table 6.3: Experimental results for Data-Driven Component Identification

	Data-Driven Component Identification			
Dataset	Gain $L\%$	Comp.Purity (%)	Average DS	Optimal k
Adult	40.3	82.2	31.7	177
Anneal	4.8	76.2	3.7	2
Chess (kr–k)	18.2	17.8	2.9	13
Mammals	46.2	-	1.7	6
Mushroom	14.7	75.6	7.8	20
Nursery	15.0	43.4	3.6	8
Page blocks	70.3	92.5	19.6	30
Retail	25.5	-	0.10	2

Experimental results for Data-Driven Component Identification. Per database, the gain in compression over regular Krimp compression, component purity by majority class voting, average dissimilarity between the components and the optimal value of k are shown.

these experiments. The reported results are those of the run with the shortest global encoded length.

For all datasets decompositions are found that allow for much better compression of the whole. The gains show that very homogeneous blocks of data are identified; otherwise the total encodings would only have become longer, as many code tables have to be included instead of one. The components identified in the *Adult*, *Mammals* and *Page blocks* datasets are so specific that between 40% and 70% fewer bits are required than when the data is compressed as a single block.

In between, however, the components are very heterogeneous. This is shown by the average dissimilarity measurements. For example, the 19.6 measured for *Page blocks* means that on average 1960% more bits would be required for a random transaction, if it were encoded by a 'wrong' component. Also for the other datasets partitions are created that are at least as different as the original classes (see Table 6.1). Further, the component purities are in the same league as those of the model driven algorithm.

The geographic *Mammals* dataset allows us to meaningfully visually inspect the found components. Figure 6.2 shows the best found decomposition. Each of the rows (i.e. each grid location of 50×50 kilometres) has been assigned to one of the six components based on its items (i.e. the mammals that have been recorded at that location). This is a typical example where normally ad-hoc

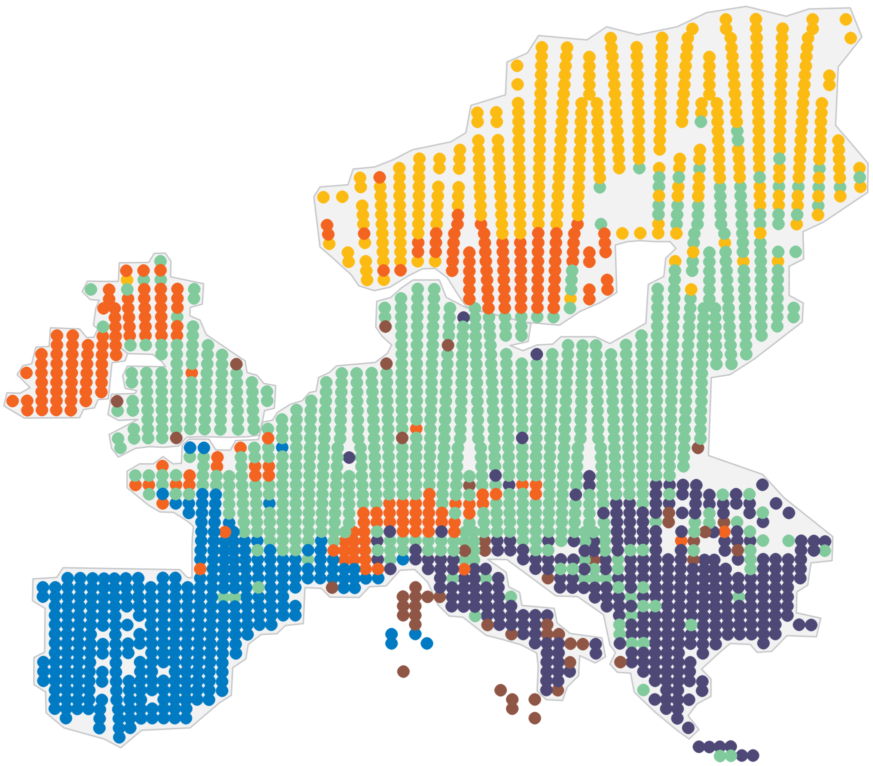


Figure 6.2: The components of *Mammals* ($k = 6$, optimal).

solutions are used [51] as it is difficult to define a meaningful distance measure. Our method only regards the characteristics of the components, the patterns in which the items occur. As can be seen in Figure 6.2, this results in clear continuous and geographically sound groupings - even though the algorithm was completely uninformed in this regard. For example, the yellow, orange and green components cover respectively the 'polar' region, highlands and more temperate areas of Europe.

Discussion

Much shorter data descriptions and high dissimilarities found show that highly specific database components are identified: internally homogeneous and heterogeneous in between. Moreover, the improvements in purity over the baseline show that many components have a strong relation to a particular class. Although mainly a property of the data, the optimal number of components we identify is low, which enables experts to interpret the components by hand.

By definition, each randomly initialised run of the algorithm may result in a different outcome. In all our experiments, however, we found the outcomes to be stable - indicating the robustness of a data compression approach. Nevertheless, to ensure that one finds a very good solution a couple of runs are required. However, as only partitions of the data have to be compressed, the algorithm runs very fast (a run typically takes seconds to minutes) so this poses no practical problems. For example, for the largest dataset, *Retail*, it took only six hours in total to run ten independent runs over all k.

6.6 Discussion

The components of a database can be identified using both model and data driven compression approaches. The experimental results show that both methods return characteristic partitions with much shorter total encoded size than regular single–component Krimp compression. The distributions of the components are shown to be very different from each other using dissimilarity measurements. These dissimilarities show that the code tables, and thus the patterns present in the data, are very unalike - i.e. the characteristics of the data components are different.

The optimal number of components is determined automatically by MDL by either method: no parameter k has to be set by hand. Each of the two proposed component identification methods has its own merit and depending on the data and computational power available one may choose for either the data or model driven algorithm. When dealing with (very) large databases, the data driven method is bound to provide good results quickly. For analysis of reasonable amounts of data the model driven method has the advantage of characterising the components very well with only modest numbers of patterns. Although we here focus on transaction data and the Krimp encoding, note that without much effort the framework can be generalised to a generic solution based on MDL: given a data type and suited encoding scheme, components can be identified by minimising the total compressed size. Especially the data driven algorithm is very generic and can be easily applied to other data types or adapted to use different compressors. This is trivial if a compressor can encode single transactions, otherwise one should assign each transaction to the component of which the encoded size of transaction and component is minimal. The model driven algorithm is more specific to our code table approach, but can also be translated to other data types.

6.7 Related Work

Clustering is clearly related to our work, as it addresses part of the problem we consider. The best-known clustering algorithm is k-means [79], to which our data driven component identifier is related as both iteratively reassign data points to the currently closest component. For this, k-means requires a distance metric, but it is often hard to define one as this requires prior knowledge on what distinguishes the different components. Our method does not require any a priori knowledge. In addition, clustering only aims at finding components, while here we simultaneously model these partitions explicitly with small pattern sets.

Local frequency of items has been used by Wang *et al.* [116] to form clusters, avoiding pair-wise comparisons but ignoring higher order statistics. Aggarwal *et al.* [4] describe a form of k-means clustering with market basket data, but for this a similarity measure on transactions is required. This measure is based on pair-wise item correlations; more complex (dis)similarities between transactions may be missed. Moreover, many parameters (eight) have to be set manually, including the number of clusters k.

Bi-clustering algorithms [92] look for linked clusters of both objects and attribute-value pairs called bi-clusters. These methods impose more restrictions on the possible clusters; for instance, they do not per se allow for overlap of the attribute-value pairs in the clusters. Further, the number of clusters k often has to be fixed in advance.

Koyotürk *et al.* [66] regarded item data as a decomposable matrix, of which the approximation vectors were used to build hierarchical cluster trees. However, rather many vectors are required for decomposition. Cadez *et al.* [22] do probabilistic modelling of transaction data to characterise (profile) what we would call the components within the data, which they assume to be known beforehand.

Recently, a number of information theoretic approaches to clustering have been proposed. The Information Bottleneck [107] can be used to find clusters by minimising information loss. In particular, LIMBO [11] focuses on categorical databases. Main differences with our approach are that the number of clusters k has to be specified in advance, and as a lossy compression scheme is used MDL is not applicable.

Entropy measures allow for non-linearly defined clusters to be found in image segmentation tasks [46]. The MDL principle was used for vector quantization [17], where superfluous vectors were detected via MDL. Böhm *et al.* [18] used MDL to optimise a given partitioning by choosing specific models for each of the parts. These model-classes need to be pre-defined, requiring premonition of the component models in the data. Cilibrasi & Vitányi [30] used pairwise compression to construct hierarchical cluster trees with high accuracy. The method works well for a broad range of data, but performance deteriorates for larger (>40 rows) datasets. Kontkanen *et al.* [63] proposed a theoretical framework for data clustering based on MDL which shares the same global code length criterion with our approach, but does not focus on transaction databases.

6.8 Conclusion

Transaction databases are mixtures of samples from different distributions; identifying the components that characterise the data, without any prior knowledge, is an important problem. We formalise this problem in terms of total compressed size using MDL: the optimal decomposition is that set of code tables and partitioning of the data that minimises the total compressed size. No prior knowledge on the distributions, distance metric or the number of components has to be known or specified.

We presented two approaches to solve the problem and provide parameter-free algorithms for both. The first algorithm provides solutions by finding optimal sets of code tables, while the second does this by finding the optimal data partitioning. Both methods result in significantly improved compression when compared to compression of the database as a whole. Component purity and dissimilarity results confirm our hypothesis that characteristic components can be identified by minimising total compressed size. Visual inspection shows that MDL indeed selects sound groupings.

The approach we present can easily be adopted for other data types and compression schemes. The Krimp-specific instantiation for categorical data in this chapter shows that the MDL principle can be successfully applied to this problem. The data driven method especially is very generic and only requires a compression scheme that approximates the Kolmogorov Complexity of the data.

Chapter **7**

Compressing Tags to Find Interesting Media Groups

On photo sharing websites like Flickr and Zooomr, users are offered the possibility to assign tags to their uploaded pictures[1]. Using these tags to find interesting groups of semantically related pictures in the result set of a given query is a problem with obvious applications. We analyse this problem from a Minimum Description Length (MDL) perspective and develop an algorithm that finds the most interesting groups. The method is based on Krimp, which finds small sets of patterns that characterise the data using compression. These patterns are sets of tags, often assigned together to photos.

The better a database compresses, the more structure it contains and thus the more homogeneous it is. Following this observation we devise a compression-based measure. Our experiments on Flickr data show that the most interesting and homogeneous groups are found. We show extensive examples and compare to clusterings on the Flickr website.

[1]This chapter has been published as [68]: M. van Leeuwen, F. Bonchi, B. Sigurbjörnsson & A. Siebes. Compressing Tags to Find Interesting Media Groups. In *Proceedings of the CIKM'09* (2009). Runner-up Best Student Paper.

7.1 Introduction

Collaborative tagging services have become so popular that they hardly need any introduction. Flickr, del.icio.us, Technorati, Last.fm, or citeulike – just to mention a few – provide their users with a repository of resources (photos, videos, songs, blogs, urls, scientific papers, etc.), and the capability of assigning tags to these resources. Tags are freely chosen keywords and they are a simple yet powerful tool for organising, searching and exploring the resources.

Suppose you're fond of *high dynamic range* (*HDR* for short, a digital photo technique) and you want to see what other photographers are doing in HDR. You type the query *hdr* in Flickr and you get a list of more than one million pictures (all pictures tagged with *hdr*). This is not a very useful representation of the query result, neither for exploring nor for discovery. In a situation like this, it is very useful to have the resulting pictures automatically grouped on the basis of their other tags. For instance, the system might return groups about natural elements, such as {*sea, beach, sunset*} and {*cloud, winter, snow*}, a group about urban landscape {*city, building*}, or it may localise the pictures geographically, e.g. by means of the groups {*rome, italy*} and {*california, unitedstates*}. Grouping allows for a much better explorative experience.

Presenting the results of a query by means of groups may also help discovery. Suppose you search for *pyramid.* Instead of a unique long list of pictures, you might obtain the groups: {*chichenitza, mexico*}, {*giza, cairo*}, {*luxor, lasvegas*}, {*france, museum, louvre, glass*}, {*rome, italy*}, {*glastonbury, festival, stage*}, and {*alberta, edmonton, canada*}. Thus you discover that there are pyramids in Rome, Edmonton, and that there is a pyramid stage at the Glastonbury festival. You would have not discovered this without groupings, as the first photo of Rome's pyramid might appear very low in the result list.

Grouping also helps to deal with ambiguity. E.g. you type *jaguar* and the system returns groups about the animal, the car, the guitar, and the airplane.

In a nutshell, the problem at hand is the following: given the set of photos belonging to a particular tag, find the most interesting groups of photos based only on their other tags. Since we focus only on tags, we may consider every photo simply as a set of tags, or a *tagset.*

But, what makes a group of photos, i.e. a bag of tagsets, "interesting"? The examples just described suggest the following answer: large groups of photos that have many tags in common. Another word for having many tags in common is homogeneous, which results in the following informal problem statement:

> *For a database $\mathcal{D}_Q$ of photos containing the given query Q in their tagsets, find all significantly large and homogeneous groups $G \subseteq \mathcal{D}_Q$.*

Actually, Flickr has already implemented *Flickr clusters*[2], a tag clustering mechanism which returns 5 groups or less. The method proposed in this chapter is rather different: (1) it finds groups of photos, i.e. groups of tagsets, instead of groups of tags, (2) it is based on the idea of *tagset compression*, and (3) it aims at producing a much more fine grained grouping, also allowing the user to fine-tune the grain.

Difficulties of the problem. All collaborative tagging services, even if dealing with completely different kinds of resources, share the same setting: there is a large database of user-created resources linked to a tremendous amount of user-chosen tags. The fact that tags are freely chosen by users means that irrelevant and meaningless tags are present, many synonyms and different languages are used, and so on. Moreover, different users have different intents and different tagging behaviours [10]: there are users that tag their pictures with the aim of classifying them for easy retrieval (maybe using very personal tags), while other users tag for making their pictures visited by as many other users as possible. There are users that assign the same large set of tags to all the pictures of a holiday, even if they visited different places during the holiday; thus creating very strong, but fake, correlations between tags. This all complicates using tags for any data mining/information retrieval task.

Another difficulty relates to the dimensions of the data. Even when querying for a single tag, both the number of different pictures and the number of tags can be prohibitively large. However, as pictures generally only have up to tens of tags, the picture/tag matrix is sparse. This combination makes it difficult to apply many of the common data mining methods. For instance, clustering methods like k-means [79] attempt to cluster all data. Because the data matrices are large and sparse, this won't give satisfactory results. Clustering all data is not the way to go: many pictures simply do not belong to a particular cluster or there is not enough information available to assign it to the right cluster.

Our approach and contribution. We take a very different approach to the problem by using MDL. In the current context, this means that we are looking for groups of pictures that can be compressed well. We will make this more formal in the next section.

We consider search queries Q consisting of a conjunction of tags and we denote by $\mathcal{D}_Q$ the database containing the results of a search query. Each picture in $\mathcal{D}_Q$ is simply represented by the set of tags it contains. Our goal is to find all large and coherent groups of pictures in $\mathcal{D}_Q$. We do not require all pictures and/or tags to be assigned to a group.

The algorithm presented in this chapter uses Krimp to compress the entire

[2]see `www.flickr.com/photos/tags/jaguar/clusters/`

dataset to obtain a small set of descriptive patterns. These patterns act as candidates in an agglomerative grouping scheme. A pattern is added to a group if it contributes to a better compression of the group. This results in a set of groups, from which the group that gives the largest gain in compression is chosen. This group is taken from the database and the algorithm is re-run on the remainder until no more groups are found. The algorithm will be given in more detail in Section 7.3.

We collect data from Flickr for our experiments. To address the problems specific for this type of data (mentioned above), we apply some pre-processing. Among this is a technique based on Wikipedia redirects (see Section 7.4).

Experiments are performed on a large number of queries. To demonstrate the high quality of the results, we show extensive examples of the groups found. For a quantitative evaluation, we introduce a compression-based score which measures how good a database can be compressed. Using this score, we compare our method to the groups that can be found by *Flickr clusters*. The results show that our method finds groups of tagsets that can be compressed better than the clusters on Flickr. Even more important, pictures not grouped by our method cannot be compressed at all, while pictures not grouped in the current Flickr implementation can be compressed and thus contain structure.

7.2 The Problem

Preliminaries: MDL for Tagsets

Given are a query Q and some mechanism that retrieves the corresponding set of pictures $\mathcal{D}_Q$. We represent this set of pictures as a bag of tagsets (each picture represented by its associated tagset). Hence, the situation is identical to that in previous chapters. Since in this case the transactions and patterns of interest consist of tags, we call them *tagsets* instead of *itemsets* and we have a set of tags $\mathcal{T}$ instead of a set of items $\mathcal{I}$. For the rest, we apply the same theory and notation as in Chapter 2. We sometimes slightly abuse notation in this chapter by denoting $L(\mathcal{D} \mid CT)$ with shortcut $CT(\mathcal{D})$.

To measure structure in a database, we here introduce the notion of compressibility.

Definition 14. *Let $\mathcal{D}$ be a database over $\mathcal{T}$ and CT its optimal code table, we define* compressibility *of $\mathcal{D}$ as*

$$compressibility(\mathcal{D}) = 1 - \frac{L(\mathcal{D}, CT)}{L(\mathcal{D}, ST)}.$$

The higher compressibility, the more structure we have in the database. If compressibility is 0, there is no structure discernible by a code table.

Problem Statement

Recall that our goal is to find all significantly large and homogeneous groups in the data. Both *significantly large* and *homogeneous* are vague terms, but luckily both can be made more precise in terms of compression.

Homogeneous means that the group is characterised by a, relatively, small set of tags. That is, a group is homogeneous if it can be compressed well *relative* to the rest of the database. Hence, we should compare the performance of an overall optimal code table with a code table that is optimal on the group only. For this, we define compression gain:

Definition 15. *Let $\mathcal{D}$ be a database over $\mathcal{T}$, $G \subseteq \mathcal{D}$ a group and $CT_{\mathcal{D}}$ and CT_G their respective optimal code tables. We define* Compression Gain *of group G, denoted by $gain(G, CT_{\mathcal{D}})$, as*

$$gain(G, CT_{\mathcal{D}}) = CT_{\mathcal{D}}(G) - CT_G(G).$$

If the gain in compression for a particular group is large, this means that it can be compressed much better on its own than as part of the database. Note that compression gain is not strictly positive: negative gains indicate that a group is compressed better as part of the database. This could e.g. be expected for a random subset of the data.

Compression Gain is influenced by two factors: (1) homogeneity of the group and (2) size of the group. The first we already discussed above. For the second, note that if two groups G_1 and G_2 have the same optimal code table CT and G_1 is a superset of G_2, then $L(G_1, CT)$ will necessarily be bigger than $L(G_2, CT)$. Hence, bigger groups have potentially a larger compression gain. Since we look for large, homogeneous groups, we can now define the best group.

Problem 6 (Maximum Compression Gain Group)**.** *Given a database $\mathcal{D}$ over a set of tags $\mathcal{T}$ and its optimal code table CT, find that group $G \subseteq \mathcal{D}$ that maximises $gain(G, CT)$.*

We do not want to find only one group, but the set of *all* large homogeneous groups. Denote this set by $\mathcal{G} = \{G_1, \ldots, G_n\}$. $\mathcal{G}$ contains all large homogeneous groups if the remainder of the database contains no more such groups. That is, if the remainder of the database has compressibility 0. Since we want our groups to be homogeneous, we require that the G_i are disjoint. We call a set of groups that has both these properties a *grouping*, the formal definition is as follows.

Definition 16. *Let $\mathcal{D}$ be a database over $\mathcal{T}$ and $\mathcal{G} = \{G_1, \ldots, G_n\}$ a set of groups in $\mathcal{D}$. $\mathcal{G}$ is a* grouping *of $\mathcal{D}$ iff*

1. $compressibility\Big(\mathcal{D} \setminus \big(\bigcup_{G_i \in \mathcal{G}} G_i\big)\Big) = 0$
2. $i \neq j \rightarrow G_i \cap G_j = \emptyset$

The grouping we are interested in is the one that maximises the total Compression Gain.

Problem 7 (Interesting Tag Grouping)**.** *Given a database $\mathcal{D}$ over a set of tags $\mathcal{T}$ and its optimal code table CT, find a grouping $\mathcal{G}$ of $\mathcal{D}$ such that*

$$\sum_{G_i \in \mathcal{G}} gain(G_i, CT)$$

is maximal.

7.3 The Algorithm

In this section, we propose a new algorithm for the *Interesting Tag Grouping* problem. Our method is built upon Krimp with pruning.

Code Table-based Groups

For the *Maximum Compression Gain Group* problem, we need to find the group $G \subseteq \mathcal{D}$ that maximises $gain(G, CT)$. Unfortunately, $gain(G, CT)$ is neither a monotone nor an anti-monotone function. If we add a small set to G, the gain can both grow (a few well-chosen elements) or shrink (using random, dissimilar elements). Given that $\mathcal{D}$ has $2^{|\mathcal{D}|}$ subsets, this means that computing the group that gives the maximal compression gain is infeasible. A similar observation holds for the *Interesting Tag Grouping* problem.

Hence, we have to resort to heuristics. Given that the tagsets in the code table CT characterise the database well, it is reasonable to assume that these tagsets will also characterise the maximum compression gain group well. In other words, we only consider groups that are characterised by a set of code table elements.

Each code table element $X \in CT$ is associated with a bag of tagsets, viz., those tagsets which are encoded using X. If we denote this bag by $G(X, \mathcal{D})$, we have

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

For a set g of code table elements we simply take the union of the individual bags, i.e.

$$G(g, \mathcal{D}) = \bigcup_{X \in g} G(X, \mathcal{D}).$$

Although a code table generally doesn't have more than hundreds of tagsets, considering all $2^{|CT|}$ such groups as candidates is still infeasible. In other words, we need further heuristics.

Growing Homogeneous Groups

Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two databases over the same set of tags $\mathcal{T}$, with code tables CT_1 and CT_2, respectively. Moreover, let $CT_1 \approx CT_2$, i.e. they are based on more or less the same tagsets and the code lengths of these tagsets are also more or less the same. Then it is highly likely that the code table $CT_\cup$ of $\mathcal{D}_1 \cup \mathcal{D}_2$ will also be similar to CT_1 and CT_2. In other words, it is highly likely that

$$L(\mathcal{D}_1 \cup \mathcal{D}_2, CT_\cup) < L(\mathcal{D}_1, CT_1) + L(\mathcal{D}_2, CT_2).$$

This insight suggests a heuristic: we *grow* homogeneous groups. That is, we add code table elements one by one to the group, as long as the group stays homogeneous.

This strategy pre-supposes an order on the code table elements: which code table element do we try to add first? Given that the final result will depend on this order, we should try the best candidate first. In the light of our observation above, the more this new tagset has in common with the current set of code table elements the better a candidate it is. To make this precise, we define the notion of coherence.

Definition 17. *Let $\mathcal{D}$ be a database over the tagset $\mathcal{T}$ and let CT be its code table. Moreover, let $X \in CT$ and $g \subset CT$. Finally, let $U(g) = \bigcup_{Y \in g} Y$. Then the* group coherence *of X with g, denoted by $coherence(X, g, \mathcal{D})$, is defined as*

$$coherence(X, g, \mathcal{D}) = \sum_{i \in (X \cap U(g))} usage_{G(g,\mathcal{D})}(\{i\}).$$

Given a set of candidate tagsets $Cands$, the best candidate to try first is the one with the highest coherence with the current group, i.e.

$$bestCand(g, \mathcal{D}, Cands) = \operatorname*{argmax}_{X \in Cands} coherence(X, g, \mathcal{D}).$$

Next to this order, we need a criterion to decide whether or not to accept the candidate. This is, again, based on compression.

Algorithm 11 Grow Homogeneous Group

```
GrowGroup($g, \mathcal{D}, CT, gMinsup$) :
1.  $Cands \leftarrow CT$
2.  while $Cands \neq \emptyset$ do
3.     $best \leftarrow$ bestCand($g, \mathcal{D}, Cands$)
4.     $Cands \leftarrow Cands \setminus best$
5.     if AcceptCandidate($best, g, \mathcal{D}, CT, gMinsup$) then
6.        $g \leftarrow g \cup \{best\}$
7.     end if
8.  end while
9.  return $g$
AcceptCandidate($best, g, \mathcal{D}, CT, gMinsup$) :
10. $G \leftarrow G(g, \mathcal{D})$
11. $G' \leftarrow G(g \cup \{best\}, \mathcal{D})$
12. $\delta \leftarrow G' \setminus G$
13. $CT_G \leftarrow$ Krimp($G$, MineCandidates($G, gMinsup$))
14. $CT_{G'} \leftarrow$ Krimp($G'$, MineCandidates($G', gMinsup$))
15. return $CT_{G'}(G') < CT_G(G) + CT(\delta)$
```

Definition 18. *Let g be the set of tagsets that define the current group $G = G(g, \mathcal{D})$. Consider candidate X, with candidate cluster $G' = G(g \cup \{X\}, \mathcal{D})$ and $\delta = G' \setminus G$. Let $CT_{\mathcal{D}}$, CT_G and $CT_{G'}$ be the optimal code tables for respectively $\mathcal{D}$, G and G'. We now accept candidate X iff*

$$CT_{G'}(G') < CT_G(G) + CT_{\mathcal{D}}(\delta).$$

When a candidate helps to improve compression of all data in the new group, we decide to keep it. Otherwise, we reject it and continue with the next candidate. The algorithm that does this is given in Algorithm 11.

Finding Interesting Tag Groups

The group growing algorithm given in the previous subsection only gives us the best candidate given a non-empty group. In line with its hill-climbing nature, we consider each code table element $X \in CT$ as starting point. That is, we grow a group from each code table element and choose the one with the maximal compression gain as our *Maximum Compression Gain Group.* Note that this implies that we only need to consider $|CT|$ possible groups.

To solve the *Interesting Tag Grouping* problem we use our solution for the *Maximum Compression Gain Group* problem iteratively. That is, we first find

Algorithm 12 Find Interesting Tag Groups

```
FindTagGroups($\mathcal{D}$, $minElems$, $dbMinsup$, $gMinsup$) :
1.  $groups \leftarrow \emptyset$
2.  loop
3.    $CT \leftarrow$ Krimp($\mathcal{D}$, MineCandidates($\mathcal{D}, dbMinsup$))
4.    $bestGain \leftarrow 0$
5.    $best \leftarrow \emptyset$
6.    for all $X \in CT$ do
7.      $cand \leftarrow$ GrowGroup($\{X\}, \mathcal{D}, CT, gMinsup$)
8.      if $gain(G(cand, \mathcal{D}), CT) > bestGain$ and $|cand| >= minElems$ then
9.        $bestGain \leftarrow gain(G(cand, \mathcal{D}), CT)$
10.       $best \leftarrow cand$
11.     end if
12.   end for
13.   if $best \neq \emptyset$ then
14.     $groups \leftarrow groups \cup \{(best, G(best, \mathcal{D}))\}$
15.     $\mathcal{D} \leftarrow \mathcal{D} \setminus G(best, \mathcal{D})$
16.   else
17.     break
18.   end if
19. end loop
20. return $groups$
```

the Maximum Compression Gain Group G on $\mathcal{D}$, then repeat this on $\mathcal{D} \setminus G$, and so on until no group with gain larger than 0 can be found. This simple recursive scheme is used by the FindTagGroups algorithm presented in Algorithm 12.

The algorithm has four parameters, with database $\mathcal{D}$ obviously being the most important one. The second parameter is the minimum number of code table elements a group has to consist of to get accepted. Krimp requires a minimum support threshold to determine which frequent tagsets are used as candidates and we specify these separately for the database (*dbMinsup*) and the groups (*gMinsup*). We will give details on parameter settings in Section 7.5.

The result of the FindTagGroups algorithm is a set of pairs, each pair representing a group of the grouping. Each pair contains (1) the code table elements that were used to construct the group and (2) the transactions belonging to the group. The former can be used to give a group description that can be easily interpreted, as these are the tagsets that characterise the group. E.g. a group description could be the k most frequent tags or a 'tag cloud' with all tags.

Table 7.1: Dataset properties.

Query	$\lvert\mathcal{T}\rvert$	$\lvert db \rvert$	*#tags/photo*
Architecture	13657	541810	8.0
Art	15498	861711	8.5
Bicycle	10126	98304	6.1
Black and white	14042	567554	7.4
Eiffeltower	3994	13087	4.8
Florence	5299	42528	5.3
Fun	9162	5533	14.2
HDR	12788	210389	7.3
Hollywood	9090	51176	7.7
Jaguar	9283	10996	7.0
Manhattan	9328	167087	6.8
Niagara falls	3144	11307	4.8
Night	13366	790277	7.4
Portrait	12982	846530	7.6
Pyramid	7288	15657	6.9
Reflection	13157	437769	7.5
Rock	16305	249790	8.0
Skyscraper	9298	42292	8.5
Spain	9376	324682	6.9
Windmill	8474	23838	5.9

7.4 Data Pre-processing

Our data collection consists of tagsets from Flickr photos. We evaluate the performance of the algorithm on a diverse set of datasets. Each dataset consists of all photos for a certain query, i.e. all photos that have a certain tag assigned. Table 7.1 shows a list of the queries used to evaluate the algorithm, together with some properties of the pre-processed queries. We use a wide range of different topic types, ranging from locations to photography terms, with general and specific, as well as ambiguous and non-ambiguous queries.

To reduce data sparsity we limit our attention to a subset of the Flickr tag vocabulary, consisting of roughly a million tags used by the largest number of Flickr users. Effectively this excludes only tags that are used by very few users.

Another source of data sparsity is that Flickr users use different tags to refer to the same thing. The source of this sparsity can have several roots:

new york city city new york, new york, city of new york, new york skyline, nyc, new yawk, ny city, the city that never sleeps, new york new york

italy italie, italia, perve

eiffel tower eiffle tower, iffel tower, effel tower, tour eiffel, eifel tower, eiffel tour, the eiffel tower, la tour eiffel, altitude 95

skyscraper skyscrapers, office tower, tall buildings, skyskraper, skycrappers

Figure 7.1: Examples of tag transformations using Wikipedia redirects.

(1) singular or plural forms of a concept (e.g. scyscraper, skyscrapers); (2) alternative names of entities (e.g. New York City, NYC, New York NY); (3) multilingual references to the same entity (e.g. Italy, Italia, Italie); or (4) common misspellings of an entity (e.g. Effel Tower, Eifel Tower). We address this problem using Wikipedia redirects. If a user tries to access the 'NYC' Wikipedia page she is redirected to the 'New York City' Wikipedia page. We downloaded the list of redirects used by Wikipedia and mapped them to Flickr tags using exact string matching. This results in a set of rewrite rules we used to normalise the Flickr tags. E.g. all occurrences of the tag 'nyc' were replaced by the tag 'new york city'.

Figure 7.1 shows some examples of the rewrite rules that were gathered using Wikipedia redirects. The figure shows all strings that were transformed to the given normalised string (bold). We can see that using the redirects we address, to some extent, the problem of singular/plural notation, alternative names, multilinguality and common misspellings.

A very common 'problem' in Flickr data is that users assign a large set of tags to entire series of photos. For example, when someone has been on holiday to Paris, *all* photos get the tags *europe, paris, france, eiffeltower, seine, notredame, arcdetriomphe, montmartre*, and so on. These tagsets are misleading and would negatively influence the results. As a workaround, we make all transactions in a dataset unique; after all, we cannot distinguish photos using only tag information if they have exactly the same tags.

Another issue pointed out by this example is that some items are not informative: e.g. if we query for *eiffeltower*, many of the transactions contain the tags *europe*, *paris* and *france*. If one were to find a large group, including these high-frequent tags would clutter the results. Therefore, we remove all items with frequency $\geq 15\%$ from each dataset. One final pre-processing step is to remove all tags containing either a year in the range 1900-2009 or any camera brand, as both kinds of tags introduce a lot of noise.

7.5 Experiments

Experimental Setup

To assess the quality of the groups produced by the FindTagGroups algorithm, a large number of experiments was performed on the datasets for which basic properties are given in Table 7.2. The experiments reported on in this section all use the same parameter settings:

dbMinsup = 0.33% This is the *minsup* parameter used by Krimp to determine the set of candidate frequent tagsets (see Algorithm 12). This parameter should be low enough to enable Krimp to capture the structure in the data, but not too low as this would allow very infrequent tagsets to influence the results. For this application, we empirically found a *dbMinsup* of 0.33% to always give good results.

gMinsup = 20% This parameter is used when running Krimp on a group (see Algorithms 11 and 12). As groups are generally much smaller than the entire database, this parameter can be set higher than *dbMinsup.* We found a setting of 20% to give good results.

minElems = 2 This parameter determines the minimum number of code table elements required for a group. In practice, it acts as a trade-off between fewer groups that are more conservative (low value) and more groups that are more explorative (high value). Unless mentioned otherwise, this parameter is fixed to 2.

An Example Query

To demonstrate how the algorithm works and performs, let us first consider a single dataset: *Bicycle.* As we can see from Table 7.2, it contains 98,304 photos with a total of 10,126 different tags assigned, 6.1 per photo on average.

When FindTagGroups is applied on this dataset, it first constructs a code table using Krimp. Then, a candidate group is built for each tagset in the code table (using the GrowGroup algorithm). One such tagset is {*race, tourofcalifornia*} and the algorithm successfully adds coherent tags to form a group: first the tagset {*race, racing, tourofcalifornia, atoc, toc*}, then {*race, racing, tourofcalifornia, atoc*}, {*race, racing, tourofcalifornia, stage3*}, and so on until 22 code table elements have been selected to form a group.

Compression gain is computed for each of the candidate groups constructed and the group with the maximal gain is chosen. In this case, the group with most frequent tags {*race, racing, tourofcalifornia, atoc, cycling*} has the largest gain (57,939bits) and is selected. All 3,564 transactions belonging to this group

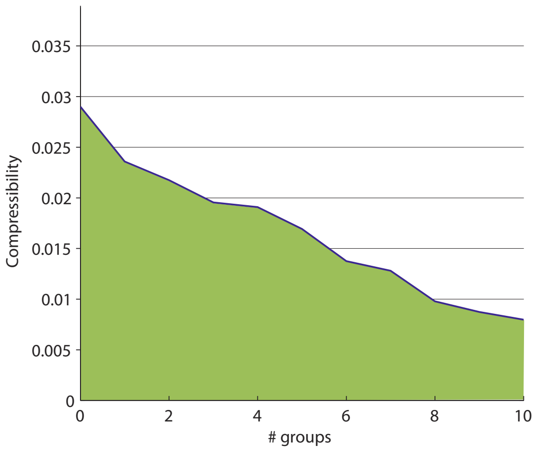


Figure 7.2: Database compressibility for *Bicycle* when applying FindTagGroups ($minElems = 2$).

are removed from the database, a new code table is built on the remainder of the database and the process repeats itself.

As explained in Section 7.2, we can use compressibility to measure how much structure is present in a database. If we do this for the group we just found, we get a value of 0.05, so there is some structure but not too much. However, it is more interesting to see whether there is any structure in the remainder of the database. Compressibility of the original database was 0.029, after removing the group it is 0.024, so we removed some structure. Figure 7.2 shows database compressibility computed each time after a group has been removed. This shows that we remove structure each time we form a group and that there is hardly any structure left when the algorithm ends: compressibility after finding 10 groups is 0.008.

So far we assumed the *minElems* parameter to have its default value 2. However, Figure 7.3 shows the resulting groupings for different settings of *minElems*. Each numbered item represents a group and the groups are given in the order in which they are found. For each group, the union of the code table elements defining the group is taken and the tags are ordered descending on frequency to obtain a group description. For compact representation, only the 5 most frequent tags are given (or less if the group description is smaller). Font size represents frequency relative to the most frequent tag in that particular group description.

Clearly, most groups are found when *minElems* $= 1$. Groups with different topics can be distinguished: sports activities (1, 3, 8, 11), locations (4, 7, 9, 10, 12, 15, 16, 18, 19), 'general circumstances' (6, 14, 17). The second group is due to a linguistic problem which is not solved by the pre-processing.

Increasing *minElems* results in a reduction in the number of groups. This does not mean that only the top-k groups from the *minElems* $= 1$ result set are picked. Instead, a subset of that result set is selected and sometimes even new groups are found (e.g. *minElems* $= 12$, group 3). Unsurprisingly, increasing the *minElems* value results in fewer groups with larger group descriptions. These are often also larger in the number of transactions and can generally be regarded more confident but less surprising. For the examples in Figure 7.3, for *minElems* $= 1$ a group on average contains 1,181 transactions, 1,828 transactions for *minElems* $= 2$, 3,034 transactions for *minElems* $= 6$ and 3,325 transactions for *minElems* $= 12$.

We found *minElems* $= 2$ to give a good balance in the number of groups, the size of the groups and how conservative/surprising these are. We therefore use this as the default parameter setting in the rest of the section.

More Datasets

Table 7.2 shows quantitative results for all datasets. From this table, we see that the algorithm generally finds 10-20 groups, but sometimes more, with *fun* as extreme with 50 groups. This can be explained by the fact that 'fun' is a very general concept, so it is composed of many very different conceptual groups and these are all identified. On average, between 5 and 8 code table elements are used for a group. The average number of transactions (photos) that belong to a group depends quite a lot on the dataset. Average group sizes range from only 73 for *fun* up to 17,899 for *art*. However, the size relative to the original database does not vary much, from 1.3% up to 2.7%. The complete groupings usually give a total coverage between 20 and 40%.

The three rightmost columns show compressibility values: for the groups found (averaged), for the database remaining after the algorithm is finished and for the initial database (called *base*). The compressibility of the initial database can be regarded as a baseline, as it is an indication of how much structure is present. In general, there is not that much structure present in the sparse tag data so values are not far above 0. However, the groups identified have more structure than baseline: for all datasets, average group compressibility is higher than baseline. Even more important is that compressibility of the database remaining after removing all groups is always very close to 0 ($<= 0.01$). Hence, all structure that was in the database has been captured.

To give some more insight in the groups found by the algorithm, Figures 7.4

Table 7.2: Quantitative results obtained with the FindTagGroups algorithm.

		Group average		Compressibility		
Query	$\lvert\mathcal{G}\rvert$	*#elems*	*#transactions*	*group avg*	*rest*	*base*
Architecture	22	5.5	8507 (1.6%)	0.064	0.006	0.047
Art	15	7.9	17899 (2.1%)	0.034	0.002	0.032
Bicycle	10	7.1	1828 (1.9%)	0.061	0.008	0.029
Black & white	17	6.5	10815 (1.9%)	0.056	0.004	0.037
Eiffeltower	12	6.7	309 (2.4%)	0.072	0.001	0.039
Florence	18	6.9	731 (1.7%)	0.078	0.007	0.048
Fun	50	6.3	73 (1.3%)	0.213	0.009	0.137
HDR	14	5.4	3608 (1.7%)	0.043	0.005	0.032
Hollywood	24	8.1	731 (1.4%)	0.112	0.004	0.081
Jaguar	11	9.3	251 (2.3%)	0.132	0.012	0.067
Manhattan	19	8.6	3125 (1.9%)	0.081	0.005	0.063
Niagara falls	18	5.8	227 (2.0%)	0.117	0.006	0.048
Night	19	5.0	13011 (1.6%)	0.043	0.004	0.034
Portrait	19	6.2	15477 (1.8%)	0.047	0.002	0.037
Pyramid	20	11.2	369 (2.4%)	0.115	0.004	0.066
Reflection	17	6.3	8660 (2.0%)	0.036	0.005	0.033
Rock	13	15.9	6635 (2.7%)	0.092	0.005	0.044
Skyscraper	38	8.8	731 (1.7%)	0.059	0.009	0.083
Spain	19	6.3	5670 (1.7%)	0.079	0.008	0.054
Windmill	13	7.6	574 (2.4%)	0.066	0.003	0.036

For each dataset, the number of groups found, the average number of code table elements and transactions per group are given. Compressibility is shown for all groups (averaged), the database remaining after the algorithm is finished and the initial database.

and 7.5 show additional results for 4 datasets. In general, the resulting groups clearly identify specific photo collections that are conceptually different. Sometimes there appears to be some redundancy (from a semantic point of view), but in most cases additional tags reveal subtle differences. Note that not all tags that are part of the group descriptions are shown.

Some groups are surprising. For example, the second group for *black and white*, with *dewolf* as most frequent tag. At first sight, it looks like a single user got chosen as secondmost important concept for this query. However, performing a Flickr search for *blackandwhite* and *dewolf* quickly learns us that there is a Nick DeWolf photo archive project and this archive contains over 13,000 black&white pictures taken by this co-founder of Teradyne in Boston. After all, it is thus by no means strange that this comes up as second group.

Compared to Flickr

As mentioned in the introduction, Flickr offers its visitors the possibility to browse through 'clusters' of photos given a certain tag. However, Flickr clusters are conceptually different from our groupings: (1) Flickr clusters consist of tags, not pictures, (2) Flickr only gives 1 to 5 clusters and (3) clusters are usually quite general and conservative (hardly surprising). An objective semantic comparison of the groups we find to the Flickr clusters is therefore impossible; it would come down to subjective preference of the person asked.

To give an example, Flickr gives the following clusters for *bicycle*[3]:

1. bike street bw cycling city urban cycle red road bikes
2. amsterdam netherlands holland
3. fixie fixed fixedgear gear
4. england uk

It is up to personal taste whether you prefer this or the groupings we presented in Figure 7.3. What we can do though, is to 'simulate' Flickr clusters with the data we have and compute compressibility values for both the clusters and the unclustered photos. To this end, we first retrieve the tagsets that identify the clusters from the Flickr website and pre-process these the same way we pre-processed our data. Next, take the pre-processed datasets also used for the previous experiments and assign each transaction to one of the clusters or to the database with 'remaining' photos. Each cluster is constructed as follows: each photo is assigned to the tagset with which it has the largest tag intersection. (In case of a tie, the transaction is assigned to the first tagset with the

[3]`www.flickr.com/photos/tags/bicycle/clusters/`

largest intersection, clusters ordered as on the website.) Any transactions that do not intersect with any of the cluster's tagsets are assigned to the remainder database.

For all queries used in this chapter, Flickr presents 3.5 clusters on average. Average compressibility for all Flickr clusters we obtained is 0.014. This means that there is hardly any structure in the clusters that can be captured in a code table by Krimp, much less than the groups the FindTagGroups algorithm finds.

More important are the compressibility values computed for the database containing all photos not belonging to any cluster: is there any structure left? The average value we obtained for all datasets is 0.035, clearly indicating that there is structure not yet captured in one of the clusters. Figure 7.6 shows a comparison of remaining database compressibility between the Flickr method and our method. It is easy to see that our method is better at finding all structure in the database than Flickr's method is.

Runtimes

All experiments have been performed on a machine with a 3GHz Intel Xeon CPU and 2Gb of memory. Runtimes depend on a large number of factors, amongst which the number of tags $|\mathcal{T}|$ and transactions $|\mathcal{D}|$. The amount of structure and the number of groups also play an important role. 9 datasets required less than 20 minutes, with *Eiffeltower* being the quickest taking only 3 minutes. The remaining datasets took longer than that, with timings ranging from 1 hour for *Fun* and *HDR*, up to 20 hours for *Architecture* and *Portrait*.

These runtimes indicate that the method should be used offline, not online at query time. For most queries, stability of the resulting groupings should be high, implicating that the algorithm does not need to be re-run often.

Discussion

Using compression, the FindTagGroups algorithm aptly finds significantly large and homogeneous groups in tag data. The groupings usually contain between 20% and 40% of the photos in the database, a fair amount given the sparsity of the data. Compressibility shows that the groups are more homogeneous than the original database, while there is no discernible structure left in the remaining database.

Manual inspection of the groups shows that the quality of the groupings is high; a manageable number of semantically coherent groups is found. As each group is characterised by a small set of tagsets, it is easy to give intuitive group descriptions using tag clouds. The grain of the groups returned by the algorithm can be fine-tuned by the end-user: he may either wish for more, smaller and explorative groups or prefer less, larger and conservative groups.

A qualitative comparison to Flickr clusters is difficult, but by simulating these clusters we showed that Flickr clusters leave more structure in the database than our method.

Data preparation is very important, due to the amount of noise present in user-generated tagsets. The method based on Wikipedia redirects solves many problems with synonyms and other linguistic issues.

In the experiments we focus on pictures, but our algorithm can find groups for any type of tagged resource. Apart from improving usability of tag searching, groupings could also be used for applications like tag recommendation [102, 103].

7.6 Related Work

The main reference for our work is *Flickr clusters*. Even if the technical details are not public, by observing the results we can infer (1) that it clusters tags, as the same tag can not belong to more than one group, and (2) that the maximum number of clusters is 5. As said in the introduction our method finds groups of pictures, i.e. groups of tagsets, instead of groups of tags, and it aims at producing a much more fine grained grouping. A consequence of having a larger number of groups is that they have larger cohesiveness.

Begelman *et al.* [15] propose to first build a graph of tags and then to apply a spectral bisection algorithm in combination with modularity measure optimization. Similar to Flickr clusters, they cluster the tags and not the resources (URLs). Another difference with our work is that they try to cluster the whole tag space, e.g. for creating multiple, more cohesive, *tag clouds* instead of a unique large tag cloud. Instead we focus on the result set of a given query.

Recently, three other papers [47, 96, 124] have studied the use of tags from large-scale social bookmarking sites (such as del.icio.us) for clustering web pages in semantic groups. Ramage *et al.* [96], explore the use of clusterings over vector space models that includes tags and page text. In this vector space k-means is compared to a novel generative clustering algorithm based on latent Dirichlet allocation. Zhou *et al.* [124] investigate the possibility of devising generative models of tags and documents contents in order to improve information retrieval. Their holistic approach combines a language model of the resources and the tags with user domain categorisation. Another work considering resources, tags, and users in a unifying framework is by Grahl *et al.* [47]. They present a conceptual clustering where first tags are clustered by k-means, then the FolkRank [54] algorithm is applied to discover resources and users that are related to each cluster of tags. Yeung *et al.* [12] also focus their analysis on resources-tags-users triplets, in particular for tag disambiguation.

Subspace clustering [67] tries to find clusters in 'subspaces' of the data, where a subspace is usually defined as subsets of both the data points and attributes. There are two important differences with our method. First, tags are not dimensions and the fact that a tag is not present may mean different things – we only look at the tags that are present. Secondly, we do not attempt to group all data, which subspace clustering methods still do.

7.7 Conclusions

We propose an algorithm that addresses the problem of finding homogeneous and significantly large groups in tagged resources, such as photos. The method is based on the MDL principle and uses the Krimp algorithm to characterise data with small sets of patterns. The best group is the group that gives the largest gain in compression, i.e. it can be compressed much better as a group than as part of the entire database. To assess the amount of structure present in a database, we define compressibility: the better a database or group compresses, the more structure it contains.

We perform experiments on a large collection of datasets obtained from Flickr; the data consists of tagsets belonging to photos. A pre-processing technique based on Wikipedia redirects solves many problems typical for this type of data. Experiments show that semantically related groups are identified and no structure is left in the database.

minElems = 1

1. race racing tourofcalifornia atoc cycling
2. bicicleta bici fahrrad cycling
3. freestyle bmx
4. netherlands amsterdam holland nederland fiets
5. fixedgear fixie fixed gear trackbicycle
6. travel trip
7. street tokyo bw japan people
8. cycling cycle mountainbike mtb london
9. newyorkcity city urban china beijing
10. portland oregon bikeportland
11. p12 pro racing race
12. canada bc toronto
13. black white
14. blue sky red cloud
15. sanfrancisco california sf
16. unitedkingdom england
17. winter snow
18. paris france
19. shanghai china
20. child kid

minElems = 2

1. race racing tourofcalifornia atoc cycling
2. bicicleta bici fahrrad cycling
3. netherlands amsterdam holland nederland fiets
4. bmx freestyle oldschool
5. fixedgear fixie fixed gear trackbicycle
6. tokyo china japan street people
7. cycling cycle mountainbike mtb london
8. newyorkcity city urban manhattan street
9. portland oregon bikeportland
10. sky blue cloud red

minElems = 6

1. race racing tourofcalifornia atoc cycling
2. netherlands amsterdam holland fahrrad nederland
3. street tokyo bw japan people
4. cycling city urban cycle street
5. fixedgear fixie fixed gear track

minElems = 12

1. race racing tourofcalifornia atoc cycling
2. netherlands amsterdam holland fahrrad nederland
3. street city urban people bw

Figure 7.3: Groups for *Bicycle* (different settings for *minElems*, showing the 5 most frequent tags per group).

Black and white

1. black art girl kid
2. dewolf blackwhite nick boston
3. bn biancoenero blancoynegro italy portrait
4. woman girl portrait female face
5. film 35mmfilm format '120' ilford
6. noiretblanc france paris nb blancoynegro
7. selfportrait me 365days portrait self
8. portrait people face black man
9. unitedkingdom england london monochrome blackwhite
10. white black blackwhite
11. street urban city newyorkcity manhattan
12. tree cloud nature sky winter
13. cat pet animal dog kitty
14. black white light
15. child kid portrait girl boy
16. beach sand ocean sea water
17. flower macro nature

Night

1. california sanfrancisco sanfranciscobayarea city urban
2. newyorkcity manhattan lights christmas newyork
3. france paris nuit europe
4. japan tokyo light geotagging city
5. thestand sky cloud luna tree
6. london unitedkingdom england riverthames lights
7. party people friends recreation portrait
8. neon sign lights
9. sky cloud sunset star tree
10. italy notte rome roma
11. light darkness street color red
12. city lights urban street light
13. long exposure longexposure light lights
14. lights black darkness white street
15. water bridge reflection river lights
16. australia sydney melbourne
17. germany nacht berlin
18. building lights light architecture tree
19. canada ontario toronto

Figure 7.4: Results for *Black and white* and *Night* (showing ≤ 5 tags per group).

Reflection

1. selfportrait me mirror 365days self
2. beach sunset sea cloud sky
3. lake tree landscape nature mountain
4. nature tree bird pond green
5. night light river lights bridge
6. building architecture glass sky blue
7. tree autumn river leaf pond
8. light shadow color glass mirror
9. unitedkingdom england london
10. sky cloud blue tree
11. mirror portrait self automobile girl
12. macro drop closeup
13. black white bw blackandwhite
14. window glass shop street
15. red blue green yellow color
16. newyorkcity manhattan newyork
17. sunset sun sky

Pyramid

1. itza chichenitza maya mexico chichen
2. france museum glass europe architecture
3. sanfrancisco transamerica california francisco san
4. giza cairo sphinx khufu africa
5. mexico ruins chichenitza mayan maya
6. mexico ruins mexicocity teotihuacan puebla
7. mexico teotihuacan guatemala maya tikal
8. mountain switzerland niesen hurni christopher
9. luxor lasvegas nevada casino hotel
10. sky cloud architecture blue building
11. camel cairo saqqara sand desert
12. france night bw pyramide blackandwhite
13. rome italy piramide roma museo
14. glastonbury festival stage glastonburyfestival
15. mexico travel chichen itza trip
16. alberta edmonton canada muttartconservatory conservatory
17. poodle babyboy royalcanin keops love
18. night light lights
19. galveston texas moodygardens
20. france architecture fountain

Figure 7.5: Results for *Reflection* and *Pyramid* (showing ≤ 5 tags per group).

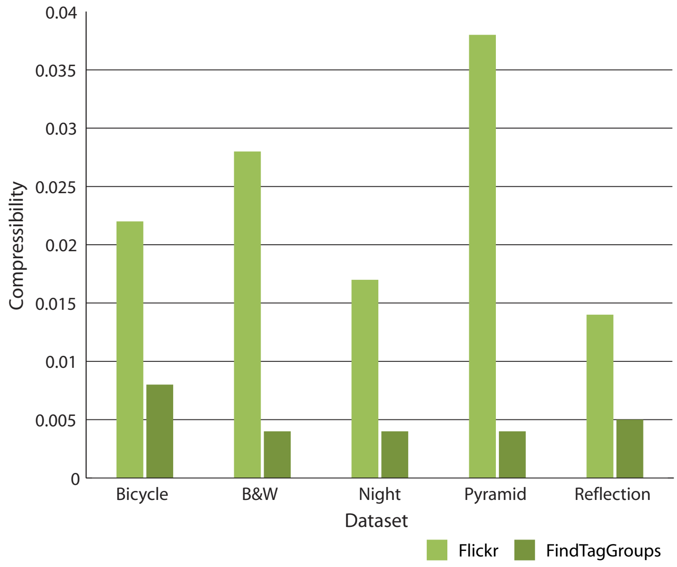


Figure 7.6: Remaining database compressibility compared to Flickr, for 5 datasets.

Chapter **8**

Conclusions

In this thesis we addressed the well-known *pattern explosion* in pattern mining; mining patterns from data is easy, but the result is usually a gigantic amount of patterns, harming both usability and interpretability. We therefore stated the following research objective:

> *Develop methods that find small, human–interpretable, pattern–based descriptive models on large datasets.*

Hence, the goal was to find small pattern sets that together describe the data well. For this task, considering individual patterns is not good enough – a quality criterion for pattern sets was needed. One of the main contributions of this thesis is the approach we took to this problem. We chose to use compression to determine which set of patterns is the best descriptive model:

> *The best set of patterns is that set that compresses the data best.*

In Chapter 2, we presented the theoretical foundations to do this, all based on lossless compression. Or, to be more precise, all based on the Minimum Description Length (MDL) principle. We introduced *code tables*, pattern-based models, and defined how to compute the total compressed size of a code table and a database encoded with this code table. We showed that the search space for finding the best compressing code table is enormous and that we therefore have to resort to heuristics, as is usual with MDL.

The Krimp algorithm approximates the optimal code table for itemset data, by iteratively considering candidate patterns for addition to the code table. If a pattern helps to improve compression, it is kept, otherwise it is permanently discarded. The extensive experimental evaluation showed that this simple scheme results in small code tables that compress the data well. The code tables are non-redundant and insightful, as they allow for human inspection and interpretation. Additionally, they are very characteristic for the data, as shown by the good classification scores obtained with the Krimp Classifier. Finally, the Krimp algorithm runs fast and large datasets pose no problems, showing that this type of pattern selection approach is practically feasible.

In Chapters 3–7, we used Krimp as foundation for solving a series of well-known problems in *Knowledge Discovery.* Each of the proposed solutions fits the main research objective, as each of them uses Krimp code tables as models. These contributions can be summarised as follows.

Characterising differences Krimp code tables can be easily used to both quantify and characterise differences between databases. The database dissimilarity measure quantifies differences, the patterns in the code tables allow for manual inspection and visualisation of differences on three different levels (i.e. on database, transaction and code table level).

Preserving privacy by generating data We have shown that privacy in databases can be preserved by generating new data that replaces the original data. For this, we presented an algorithm that generates categorical data that is virtually indiscernible from the original data.

Detecting changes The StreamKrimp algorithm detects changes and fully characterises data streams, online and with bounded storage. Each sampling distribution is captured in a code table and compression is used to accurately detect (sudden) changes.

Identifying database components We introduced two methods, based on orthogonal approaches, to identify the components that together form a database. One method is model-driven, the other is data-driven, but in both cases each identified component is characterised by a code table.

Finding interesting groups Finally, we presented an algorithm that grows homogeneous groups of transactions within a database, again using compression as quality criterion.

Throughout this thesis, we have focused on developing generic algorithms that can be used for many different applications in different areas. Most algorithms have no or very few parameters and usually no a priori knowledge is required. This makes the algorithms widely applicable in many disciplines.

As is usual in *Data Mining*, both data pre-processing and representation is important. Our Krimp algorithm works on itemset data, and therefore also on categorical data. However, our MDL-based approach can be extended to other data types as well. Actually, this has already been done for sequences & trees [13] and multi-relational data [64, 65].

The main conclusion of this thesis is that mining less, but more characteristic patterns is key to successful *Knowledge Discovery*. Using compression for pattern selection works extremely well; MDL picks the patterns that matter and effectively solves the pattern explosion in pattern mining. The resulting small pattern sets accurately describe the data and can be used for many tasks.

Additionally, this thesis shows that MDL offers a parsimonious approach that can be successfully applied to *Data Mining* problems. Based on solid theory, heuristic algorithms can be designed that perform well in practice.

Not everything is solved though and our approach also introduces new problems. First, combinations of the problems addressed in this thesis are worth investigating, e.g. identifying components in a changing data stream. Also, there are *Knowledge Discovery* problems we have not yet addressed, e.g. outlier detection and resource recommendation. And, although we have shown that our methods perform well on a variety of data, the ultimate goal would be to have these used for large-scale real-world applications.

More fundamental to our approach is that we currently need to mine (all) frequent patterns before doing pattern selection. To speed up the process, it would be very beneficial if we could avoid this and mine the patterns that matter directly from the database. To achieve this, it would be good to get more insight in the process of selecting patterns, for example through visualisation. Naturally, it would be interesting to extend our approach to other types of data, for example to numerical data.

Finally, extending our compression approach to other (pattern-based) *Data Mining* concepts would be a very interesting direction. One could, for example, think of applying compression in Subgroup Discovery to reduce the search space or as subgroup quality criterion.

Bibliography

[1] C.C. Aggarwal. A framework for diagnosing changes in evolving data streams. In *Proceedings of the ACM SIGMOD'03*, pages 575–586. ACM Press, 2003.

[2] C.C. Aggarwal. On abnormality detection in spuriously populated data streams. In *Proceedings of the SDM'05*, pages 80–91, 2005.

[3] C.C. Aggarwal, editor. *Data Streams: Models and Algorithms*. Springer, 2007.

[4] C.C. Aggarwal, C. Procopiuc, and P.S. Yu. Finding localized associations in market basket data. *Trans Knowledge and Data Engineering*, 14(1):51–62, 2002.

[5] C.C. Aggarwal and P.S. Yu. A condensation approach to privacy preserving data mining. In *Proceedings of the EDBT'04*, pages 183–199, 2004.

[6] D. Agrawal and C.C. Aggarwal. On the design and quantification of privacy preserving data mining algorithms. In *Proceedings of the SIGMOD'01*, pages 247–255. ACM, 2001.

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

[8] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the VLDB'94*, pages 487–499, 1994.

[9] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proceedings of the SIGMOD'00*, pages 439–450. ACM, 2000.

[10] M. Ames and M. Naaman. Why we tag: motivations for annotation in mobile and online media. In *CHI 2007: Proc. of the SIGCHI conf. on Human factors in computing systems*, pages 971–980. ACM, 2007.

[11] P. Andritsos, P. Tsaparas, R.J. Miller, and K.C. Sevcik. LIMBO: Scalable clustering of categorical data. In *Proceedings of the EDBT*, pages 124–146, 2004.

[12] C.M. Au Yeung, N. Gibbins, and N. Shadbolt. Understanding the semantics of ambiguous tags in folksonomies. In *Proc. of the First Int. Workshop on Emergent Semantics and Ontology Evolution, ESOE 2007*, 2007.

[13] R. Bathoorn, A. Koopman, and A. Siebes. Reducing the frequent pattern set. In *Proceedings of the ICDM-Workshops'06*, pages 55–59, 2006.

[14] R. Bayardo. Efficiently mining long patterns from databases. In *Proceedings of the SIGMOD'98*, pages 85–93, 1998.

[15] G. Begelman, P. Keller, and F. Smadja. Automated tag clustering improved search and exploration in the tag space. In *Proc. of Collaborative Web Tagging Workshop at WWW 2006*, 2006.

[16] J. Bernardo and A. Smith. *Bayesian Theory.* Wiley Series in Probability and Statistics. John Wiley and Sons, 1994.

[17] H. Bischof, A. Leonardis, and A. Sleb. Mdl principle for robust vector quantization. *Pattern Analysis and Applications*, 2:59–72, 1999.

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

[19] I. Bouzouita, S. Elloumi, and S.B. Yahia. GARC: A new associative classification approach. In *Proceedings of the DaWaK'06*, pages 554–565, 2006.

[20] T. Brijs, G. Swinnen, K. Vanhoof, and G. Wets. The use of association rules for product assortment decisions: a case study. In *Proceedings of the KDD*, pages 254–260, 1999.

[21] B. Bringmann and A. Zimmermann. The chosen few: On identifying valuable patterns. In *Proceedings of the ICDM'07*, pages 63–72, 2007.

[22] I.V. Cadez, P. Smyth, and H. Mannila. Probabilistic modeling of transaction data with applications to profiling, visualization, and prediction. In *Proceedings of the KDD*, pages 37–46, 2001.

[23] T. Calders, N. Dexters, and B. Goethals. Mining frequent itemsets in a stream. In *Proceedings of the IEEE ICDM'07*, pages 83–92, 2007.

[24] T. Calders and B. Goethals. Mining all non-derivable frequent itemsets. In *Proceedings of the ECML PKDD'02*, pages 74–85, 2002.

[25] Richard M. Carp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Proc. of a Symp. on the Complexity of Computer Computations*, pages 85–103, New York, USA, 1972. Plenum Press.

[26] D. Chakrabarti, S. Papadimitriou, D.S. Modha, and C. Faloutsos. Fully automatic cross-associations. In *Proceedings of the KDD'04*, pages 79–88, 2004.

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

[28] K. Chen and L. Liu. Privacy preserving data classification with rotation pertubation. In *Proceedings of the ICDM'05*, pages 589–592, 2005.

[29] K. Chen and L. Liu. Detecting the change of clustering structure in categorical data streams. In *Proceedings of the SDM'06*, 2006.

[30] R. Cilibrasi and P. Vitanyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.

[31] E.F. Codd, S.B. Codd, and C.T. Salley. Providing olap (on-lineanalytical processing) to user analyst: An it mandate. `http://www.arborsoft.com/OLAP.html`, 1994.

[32] F. Coenen. The LUCS-KDD discretised/normalised ARM and CARM data library. `http://www.csc.liv.ac.uk/~frans/KDD/Software/LUCS-KDD-DN/DataSets/dataSets.html`, 2003.

[33] F. Coenen. The LUCS-KDD software library. `http://www.csc.liv.ac.uk/~frans/KDD/Software/`, 2004.

[34] T.M. Cover and J.A. Thomas. *Elements of Information Theory, 2nd ed.* John Wiley and Sons, 2006.

[35] B. Crémilleux and J-F. Boulicaut. Simplest rules characterizing classes generated by δ-free sets. In *Proceedings of the KBSAAI'02*, pages 33–46, 2002.

[36] T. Dasu, S. Krishnan, S. Venkatasubramanian, and K. Yi. An information-theoretic approach to detecting changes in multi-dimensional data streams. In *Proceedings of Interface'06*, 2006.

[37] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *Proceedings of the SIGKDD'99*, pages 43–52, 1999.

[38] G. Dong and J. Li. Mining border descriptions of emerging patterns from dataset pairs. *Knowledge and Information Systems*, 8(2):178–202, 2005.

[39] R. Duda and P. Hart. *Pattern Classification and Scene Analysis.* John Wiley and Sons, New York, 1973.

[40] C. Faloutsos and V. Megalooikonomou. On data mining, compression and Kolmogorov complexity. In *Data Mining and Knowledge Discovery*, volume 15, pages 3–20. Springer-Verlag, 2007.

[41] Y. Freund and R.E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comp. Sys. Sci.*, 55(1):119–139, 1997.

[42] F. Geerts, B. Goethals, and T. Mielikäinen. Tiling databases. In *Proceedings of the DS'04*, pages 278–289, 2004.

[43] K. Geurts, G. Wets, T. Brijs, and K. Vanhoof. Profiling of high-frequency accident locations using association rules. *Transportation research record*, 1840, 2003.

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

[45] B. Goethals and M.J. Zaki. Frequent itemset mining implementations repository (FIMI). `http://fimi.cs.helsinki.fi`, 2003.

[46] E. Gokcay and J.C. Principe. Information theoretic clustering. *Trans. Pattern Analysis and Machine Intelligence*, 24(2):158–171, 2002.

[47] M. Grahl, A. Hotho, and G. Stumme. Conceptual clustering of social bookmarking sites. In *LWA 2007: Lernen - Wissen - Adaption*, 2007.

[48] P.D. Grünwald. Minimum description length tutorial. In P.D. Grünwald and I.J. Myung, editors, *Advances in Minimum Description Length.* MIT Press, 2005.

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

[50] D. Hand, N. Adams, and R. Bolton, editors. *Pattern Detection and Discovery.* Springer-Verlag, 2002.

[51] H. Heikinheimo, M. Fortelius, J. Eronen, and H. Mannila. Biogeography of european land mammals shows environmentally distinct and spatial coherent clusters. *Biogeogr.*, 34(6):1053–1064, 2007.

[52] H. Heikinheimo, E. Hinkkanen, H. Mannila, T. Mielikäinen, and J. K. Seppänen. Finding low-entropy sets and trees from binary data. In *Proceedings of the KDD'07*, pages 350–359, 2007.

[53] H. Heikinheimo, J. Vreeken, A. Siebes, and H. Mannila. Low-entropy set selection. In *Proceedings of the SDM'09*, pages 569–579, 2009.

[54] A. Hotho, R. Jäschke, C. Schmitz, and G. Stumme. Information retrieval in folksonomies: Search and ranking. In *ESWC 2006: Proc. of the 3rd European Semantic Web Conference*, 2006.

[55] Z. Huang, W. Du, and B. Chen. Deriving private information from randomized data. In *Proceedings of the SIGMOD'05.* ACM, 2005.

[56] H. Kargupta, S. Datta, Q. Wang, and K. Sivakumar. Random-data perturbation techniques and privacy-preserving data mining. *Knowledge and Information Systems*, 4(7):387–414, 2005.

[57] E. Keogh, S. Lonardi, and C.A. Ratanamahatana. Towards parameter-free data mining. In *Proceedings of the KDD'04*, pages 206–215, 2004.

[58] 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. Discov.*, 14(1):99–129, 2007.

[59] D. Kifer, S. Ben-David, and J. Gehrke. Detecting change in data streams. In *Proceedings of VLDB'04*, 2004.

[60] A.J. Knobbe and E.K.Y. Ho. Maximally informative k-itemsets and their efficient discovery. In *Proceedings of the KDD'06*, pages 237–244, 2006.

[61] A.J. Knobbe and E.K.Y. Ho. Pattern teams. In *Proceedings of the ECML PKDD'06*, pages 577–584, 2006.

[62] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2(2):86–98, 2000. urlhttp://www.ecn.purdue.edu/KDDCUP.

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

[64] A. Koopman and A. Siebes. Discovering relational item sets efficiently. In *Proceedings of the SDM'08*, 2008.

[65] A. Koopman and A. Siebes. Characteristic relational patterns. In *Proceedings of the KDD'09*, pages 437–446, 2009.

[66] M. Koyotrk, A. Grama, and N. Ramakrishnan. Compression, clustering, and pattern discovery in very high-dimensional discrete-attribute data sets. *Trans Knowledge and Data Engineering*, 17(4):447–461, 2005.

[67] H-P. Kriegel, P. Kröger, and A. Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *TKDD*, 3(1):1–58, 2009.

[68] M. van Leeuwen, F. Bonchi, B. Sigurbjörnsson, and A. Siebes. Compressing tags to find interesting media groups. In *Proceedings of the CIKM'09*, pages 1147–1156, 2009.

[69] M. van Leeuwen and A. Siebes. Streamkrimp: Detecting change in data streams. In *Proceedings of the ECML PKDD'08*, pages 672–687, 2008.

[70] M. van Leeuwen, J. Vreeken, and A. Siebes. Compression picks the item sets that matter. In *Proceedings of the ECML PKDD'06*, pages 585–592, 2006.

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

[72] M. Li, X. Chen, X. Li, B. Ma, and P.M.B. Vitányi. The similarity metric. *IEEE Trans. on Information Theory*, 50(12):3250–3264, 2004.

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

[74] C.K. Liew, U.J. Choi, and C.J. Liew. A data distortion by probability distribution. *ACM Trans. on Database Systems*, 3(10):395–411, 1985.

[75] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *Proceedings of the KDD'98*, pages 80–86, 1998.

[76] G. Liu, H. Lu, J.X. Yu, W. Wei, and X. Xiao. AFOPT: An efficient implementation of pattern growth approach. In *Proceedings of the 2nd workshop on Frequent Itemset Mining Implementations*, 2004.

[77] K. Liu, C. Giannella, and H. Kargupta. An attacker's view of distance preserving maps for privacy preserving data mining. In *Proceedings of the ECMLPKDD'06*, pages 297–308, 2006.

[78] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. In *Proceedings of the ICDE'06*, pages 24–35, 2006.

[79] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Symposium on Mathematical Statistics and Probability*, 1967.

[80] H. Mannila and H. Toivonen. Multiple uses of frequent sets and condensed representations. In *Proceedings of the KDD'96*, pages 189–194, 1996.

[81] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, pages 241–258, 1997.

[82] M. Mehta, R. Agrawal, and J. Rissanen. Sliq: A fast scalable classifier for data mining. In *Advances in database technology*, pages 18–32. Springer, 1996.

[83] D. Meretakis, H. Lu, and B. Wthrich. A study on the performance of large bayes classifier. In *In proceedings of the ECML'00*, pages 271–279, 2000.

[84] S. Merugu and J. Ghosh. Privacy-preserving distributed clustering using generative models. In *Proceedings of the ICDM'03*, pages 211–218, 2003.

[85] T. Mielikäinen and H. Mannila. The pattern ordering problem. In *Proceedings of the ECML PKDD'03*, pages 327–338, 2003.

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

[87] K. Morik, J-F. Boulicaut, and A. Siebes, editors. *Local Pattern Detection*. Springer-Verlag, 2005.

[88] S. Muthukrishnan, E. van den Berg, and Y. Wu. Sequential change detection on data streams. In *Proceedings of the ICDM'07 Workshops*, 2007.

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

[90] S. Papadimitriou, A. Brockwell, and C. Faloutsos. Adaptive, unsupervised stream mining. *VLDB Journal*, 13(3):222–239, 2004.

[91] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proceedings of the ICDT'99*, pages 398–416, 1999.

[92] R. Pensa, C. Robardet, and J-F. Boulicaut. A bi-clustering framework for categorical data. In *Proceedings of the ECML PKDD*, pages 643–650, 2005.

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

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

[95] J.R. Quinlan. FOIL: a midterm report. In *Proceedings of the ECML'93*, 1993.

[96] D. Ramage, P. Heymann, C.D. Manning, and H. Garcia-Molina. Clustering the tagged web. In *WSDM 2009: Proc. of the 2nd ACM Int. Conference on Web Search and Data Mining*, 2009.

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

[98] G.-C. Rota. The number of partitions of a set. *American Mathematical Monthly*, 71(5):498–504, 1964.

[99] P. Samarati. Protecting respondents' identities in microdata release. *IEEE Trans. on Knowledge and Data Engineering*, 13(6):1010–1027, 2001.

[100] J. Shawe-Taylor and N. Cristianini. *Support Vector Machines and other kernel-based learning methods.* Cambridge University Press, 2000.

[101] A. Siebes, J. Vreeken, and M. van Leeuwen. Item sets that compress. In *Proceedings of the SDM'06*, pages 393–404, 2006.

[102] B. Sigurbjörnsson and R. van Zwol. Flickr tag recommendation based on collective knowledge. In *WWW 2008*, 2008.

[103] Y. Song, Z. Zhuang, H. Li, Q. Zhao, J. Li, W-C. Lee, and C. Lee Giles. Real-time automatic tag recommendation. In *ACM SIGIR*, pages 515–522, 2008.

[104] A. Stuart, K. Ord, and S. Arnold. *Classical Inference and the Linear Model*, volume 2A of *Kendall's Advanced Theory of Statistics.* Arnold, 1999.

[105] J. Sun, C. Faloutsos, S. Papadimitriou, and P.S. Yu. Graphscope: parameter-free mining of large time-evolving graphs. In *Proceedings of the KDD'07*, pages 687–696, 2007.

[106] N. Tatti and J. Vreeken. Finding good itemsets by packing data. In *Proceedings of the ICDM'08*, pages 588–597, 2008.

[107] N. Tishby, F.C. Pereira, and W. Bialek. The information bottleneck methods. In *Proceedings of the Allerton Conf. on Communication, Control and Computing*, pages 368–377, 1999.

[108] D. Titterington, A. Smith, and U. Makov. *Statistical Analysis of Finite Mixture Distributions.* John Wiley and Sons, 1985.

[109] V.N. Vapnik. *Statistical Learning Theory.* John Wiley and Sons, 1998.

[110] J. Vreeken, M. van Leeuwen, and A. Siebes. Characterising the difference. In *Proceedings of the KDD'07*, pages 765–774, 2007.

[111] J. Vreeken, M. van Leeuwen, and A. Siebes. Preserving privacy through data generation. In *Proceedings of the ICDM'07*, pages 685–690, 2007.

[112] J. Vreeken, M. van Leeuwen, and A. Siebes. Krimp – mining itemsets that compress. *Data Min. Knowl. Discov.*, 2010.

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

[114] J. Wang and G. Karypis. HARMONY: Efficiently mining the best rules for classification. In *Proceedings of the SDM'05*, pages 205–216, 2005.

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

[116] K. Wang, C. Xu, and B. Liu. Clustering transactions using large items. In *Proceedings of the CIKM*, pages 483–490, 1999.

[117] H.R. Warner, A.F. Toronto, L.R. Veasey, and R. Stephenson. A mathematical model for medical diagnosis, application to congenital heart disease. *Journal of the American Medical Association*, 177:177–184, 1961.

[118] G. Widmer and M. Kubat. Learning in the presence of concept drift and hidden contexts. *Machine Learning*, 23:69–101, 1996.

[119] I.H. Witten and E. Frank. *Data Mining:Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2005.

[120] Y. Xiang, R. Jin, D. Fuhry, and F.F. Dragan. Succinct summarization of transactional databases: an overlapped hyperrectangle scheme. In *Proceedigns of the KDD'08*, pages 758–766, 2008.

[121] X. Yan, H. Cheng, J. Han, and D. Xin. Summarizing itemset patterns: A profile-based approach. In *Proceedings of the KDD'05*, pages 314–323, 2005.

[122] X. Yin and J. Han. CPAR: Classification based on predictive association rules. In *Proceedings of the SDM'03*, pages 331–335, 2003.

[123] X. Zhang, D. Guozhu, and K. Ramamohanarao. Information-based classification by aggregating emerging patterns. In *Proceedings of the IDEAL'00*, pages 48–53, 2000.

[124] D. Zhou, J. Bian, S. Zheng, H. Zha, and C. Lee Giles. Exploring social annotations for information retrieval. In *WWW 2008: Proc. of the 17th int. conf on World Wide Web*, 2008.

Index

A Priori, 8
Aggregated Code Length Difference, 60
algorithm
  Component Identification
    Data-Driven, 119
    Model-Driven, 114
  FindTagGroups, 133
    Grow Homogeneous Group, 132
  Krimp, 21
    Code Table Post-Acceptance Pruning, 23
    Standard Code Table, 18
    Standard Cover, 20
  Krimp Categorical Data Generator, 80
  Krimp Classifier, 28
  StreamKrimp, 99
    Find Code Table on Stream, 97
Anonymity Score, 77

Bayes optimal choice, 27, 114, 118
bioinformatics, 1
buying behaviour, 110

closed itemset, 9, 35
code table, 11
Code Table Difference, 99
Code Table Post-Acceptance Pruning, 23
complexity, 24
compressibility, 128
Compression Gain, 129
cover function, 12

data stream, 93
Data-Driven Component Identification, 119
database, 8
descriptive data mining, 2
Dissimilarity measure, 63

Find Code Table on Stream, 97
FindTagGroups, 133
Flickr, 128
frequent set mining, 8

group coherence, 131
grouping, 130
Grow Homogeneous Group, 132

high dynamic range, 126

Improvement Rate, 96
itemset, 8

k-means, 111, 118, 127
Krimp, 21
Krimp Categorical Data Generator, 80

Krimp Classifier, 28

Laplace correction, 28, 80, 98, 114

MDL, 9, 11, 58
minimal split, 97
Minimum Description Length, *see* MDL
minsup, 8
Model-Driven Component Identification, 114

Naïve Bayes, 26, 41
Normalised Anonymity Score, 77

pattern explosion, 3, 9, 32, 44, 149
pattern mining, 3, 8
pattern sets, 9
pattern-similarity, 77
predictive data mining, 2
problem, 3
  Data Stream Minimal Split Partitioning, 97
  Data Stream Partitioning with Unbounded Storage, 95
  Identifying Database Components, 112
  Interesting Tag Grouping, 130
  Maximum Compression Gain Group, 129
  Minimal Coding Set, 16, 22
  Privacy and Quality Preserving Database, 76
pruning, 22, 34

runtime, 34, 118, 122, 141

size
  code table, 15
  encoded database, 14
  encoded transaction, 14
  total compressed, 15
split, 97
Standard Candidate Order, 20, 42
Standard Code Table, 18
standard code table, 14
Standard Cover, 20
Standard Cover Order, 19, 42
StreamKrimp, 99
support, 8
swap randomisation, 37

tagset, 126

usage count, 13

Wikipedia, 135

zero, 99

Abstract

Pattern mining is one of the best-known concepts in *Data Mining*. A big problem in pattern mining is that humongous amounts of patterns can be mined even from small datasets. This makes it hard for domain experts to discover knowledge using pattern mining, for example in the field of *Bioinformatics*. In this thesis we address the *pattern explosion* using compression.

We argue that the best pattern set is that set of patterns that compresses the data best. Based on an analysis from MDL (Minimum Description Length) perspective, we introduce a heuristic algorithm, called Krimp, that finds the best set of patterns. High compression ratios and good classification scores confirm that Krimp selects patterns that are very characteristic for the data.

After this, we proceed with a series of well-known problems in *Knowledge Discovery*, which we each unravel with our compression approach. We propose a database dissimilarity measure and show how compression can be used to characterise differences between databases. We present an algorithm that generates synthetic data that is virtually indiscernible from the original data, but can also be used to preserve privacy. Changes in data streams are detected by using a Krimp compressor to check whether the data distribution has been changed or not. Finally, compression is used to identify the components of a database and to find interesting groups in a database.

In each chapter, we provide an extensive experimental evaluation to show that the proposed methods perform well on a large variety of datasets. In the end, we conclude that having less, but more characteristic patterns is key to successful *Knowledge Discovery* and that compression is very useful in this respect. Not as goal in itself, but as means to an end: compression picks the patterns that matter.

Samenvatting

'Pattern mining', het geautomatiseerd vinden van interessante patronen in gegevens, is een van de belangrijkste concepten in *Data Mining*. Een groot probleem is echter dat meestal gigantische hoeveelheden patronen worden gevonden, zelfs op kleine databases. Dit maakt het moeilijk om pattern mining te gebruiken voor het ontdekken van waardevolle kennis. In dit proefschrift pakken we deze *patroonexplosie* aan met behulp van compressie (het 'inpakken' van data).

We beargumenteren dat de verzameling patronen die de database het beste comprimeert de beste is. Gebaseerd op een analyse vanuit MDL ('Minimum Description Length') perspectief introduceren we een heuristisch algoritme, genaamd KRIMP, dat de beste verzameling patronen vindt. Goede compressie en hoge classificatiescores bevestigen dat KRIMP patronen selecteert die de data erg goed beschrijven.

Hierna komen een aantal bekende problemen in *Knowledge Discovery* aan de orde. Bij ieder van deze problemen passen we onze op compressie gebaseerde aanpak toe en presenteren we nieuwe methoden om het probleem in kwestie op te lossen. We introduceren een afstandsmaat voor databases and laten zien hoe compressie gebruikt kan worden om verschillen tussen databases te karakteriseren. Tevens presenteren we een algoritme dat synthetische data genereert dat nauwelijks is te onderscheiden van de oorspronkelijke data. Ook laten we zien dat het genereren van data gebruikt kan worden om privacy in databases te beschermen. Veranderingen in gegevensstromen kunnen worden gedetecteerd door te bekijken wanneer de compressie van de data verandert. Tenslotte gebruiken we compressie om componenten en interessante groepen in een database te vinden.

In ieder hoofdstuk geven we een uitgebreide experimentele evaluatie om te laten zien dat de methoden het goed doen. De conclusie is dat het vinden van minder, maar karakteristiekere patronen essentieel is voor succesvolle *Knowledge Discovery*. Dit proefschrift laat zien dat compressie hiervoor uitermate geschikt is. Niet als doel, maar als middel om dit doel te bereiken: compressie kiest de patronen die er toe doen.

Dankwoord

Vier jaar lang aan één project werken lijkt misschien heel lang, maar dankzij een aantal mensen is mijn tijd als promovendus voorbijgevlogen en heb ik een hele leuke en leerzame tijd gehad. Ik wil iedereen die hier betrokken bij is geweest dan ook graag bedanken.

Beste Arno, het begon 4 jaar geleden met een vliegende start: Jilles, jij en ik werkten een ideetje van jou uit en binnen een maand was het eerste paper een feit. Een goed begin is het halve werk, en zo geschiedde. In combinatie met de vrijheid en het vertrouwen dat je me gaf, kon ik vier jaar lang onderzoek doen waar ik plezier in had en bovendien veel van leerde. Als hoofd van het departement moest je soms misschien meer vrijheid geven dan je wenste, maar wanneer nodig was je altijd bereid om mee te denken. Dankjewel!

De leescommissie, bestaande uit Pieter Adriaans, Luc De Raedt, Bart Goethals, Joost Kok en Remco Veltkamp, wil ik graag bedanken voor het lezen en goedkeuren van mijn proefschrift. Ik hoop dat jullie nog een paar interessante vragen voor de verdediging hebben bewaard.

Francesco, thank you for offering me the opportunity to work on the Flickr project and suggesting to send the paper to CIKM'09. Without this, I wouldn't have seen Hongkong! Oh, and yes, I liked the project too. I'm looking forward to visiting your lab in Barcelona.

Zonder geld geen onderzoek. Het NBIC (Netherlands Bioinformatics Centre) heeft mijn aanstelling als promovendus bekostigd, waarvoor hartelijk dank.

Elly, ik wil je bedanken voor het lezen van de introductie van mijn proefschrift, maar meer nog voor je interesse in en positieve kijk op de talloze zaken uit mijn leven (zowel scouting, werk als privé) die we, meest per e-mail, de afgelopen jaren hebben besproken.

Een plezierige werkomgeving is van groot belang en in de Algoritmische Data Analyse groep (voorheen Large Distributed Databases) heb ik me altijd op m'n plek gevoeld. Men zegt wel dat het belangrijk is om ook onder werktijd af en toe te ontspannen en dit advies namen wij dan ook graag ter harte. Hiervoor wil ik alle (oud-)collega's, Ad, Hans, Lennart, Arno, Edwin, Ronnie, Carsten, Rainer, Subianto, Jeroen, Nicola en Diyah, heel erg bedanken. Naast de wisselende activiteiten (van darten via tafeltennissen naar introquizen en what-the-movie-en) was er ook altijd een constante: gezamenlijk koffie drinken.

Beste Arne, al sinds het Robolab vriend en studiegenoot/collega. Maar naast studie & werk was er meer. Behalve vele bezoekjes aan Quignon hebben we vele restaurants in Utrecht bezocht, cabaretvoorstellingen gezien, rare foto's gemaakt, enzovoorts. Alleen naar vastenavend heb je Jilles en mij (nog?) nooit meegekregen. Nu ik klaar ben, ben jij de laatste van A113 die je proefschrift gaat afronden – veel succes met deze laatste loodjes. En dan op naar de comeback van A113 waar Jilles al aan refereerde!

Beste Jilles, in de loop der jaren hebben we samen al veel leuks mee mogen maken, als klasgenoten, studiegenoten en collega's, maar bovenal als vrienden. Van Breukelen naar de Uithof, via Lausanne, Zürich, Spanje (2x), Friesland (2x), Berlijn, Spitsbergen, San Francisco, San José, New York, Omaha, Traben-Trarbach, Antwerpen, Kaapverdië en Bled naar Bisonspoor. Dus, nu we er toch zijn, what's next? Samen fotograferen, reizen, zeilen, eten, drinken, een conferentie bezoeken, een paper schrijven? Wat mij betreft allemaal. En als je in de buurt bent, het bierli staat koud.

Opa en oma, jullie hebben me regelmatig gevraagd wat ik precies voor werk doe. Dit inhoudelijk uitleggen is niet eenvoudig, maar nu kunnen jullie het met eigen ogen zien: dit boekje is het eindresultaat van 4 jaar promoveren! Ik ben blij dat jullie bij de verdediging hiervan aanwezig kunnen zijn.

Zonder mijn ouders zou ik niet zijn wie ik nu ben. Piet en Ida, heel erg bedankt voor jullie interesse, steun en goede zorgen, ieder op jullie eigen manier. Piet, ik hoop dat dit proefschrift een mooi plekje bij de proefschriften van je eigen promovendi krijgt en dat het aantal taalfoutjes dat je vindt niet in de honderden loopt. Ida, fijn dat je altijd voor me klaarstaat, of het nu gaat om het behangen van een huis of om een avondje samen te eten en bij te praten. Karin en David, dank jullie wel voor jullie interesse. Ik zie jullie ook nog wel eens een boek schrijven, al zal het onderwerp heel anders zijn.

Lieve Lieselot, ook al zijn we 'pas' een paar jaar samen, in die tijd hebben we al ontzettend veel meegemaakt. Heel veel leuke dingen, maar helaas ook minder leuke dingen. Dit zorgde voor de nodige verbreding: naast de Uithof heb ik ook het AMC leren kennen en naast informatica heb ik ook veel meer geneeskunde geleerd. Dankzij jouw wilskracht en positieve instelling slaan we ons overal doorheen en kunnen we samen altijd lachen. Ik zou niet meer anders willen. Bwaaaaaaaaah!

Maarssen, december 2009

Curriculum Vitae

Matthijs van Leeuwen was born on 19 September 1981 in Kockengen, the Netherlands. In 1999, he started studying *Computer Science* at the Universiteit Utrecht, specialising in *Technical Artificial Intelligence.* In 2004, he obtained his master's degree after writing his thesis titled 'Spike Timing Dependent Structural Plasticity, in a single model neuron', under supervision of Ora Ohana (ETH, Zürich) and Marco Wiering (UU, Utrecht).

In 2005, Matthijs started his Ph.D. in the Algorithmic Data Analysis group (then called Large Distributed Databases) at the Universiteit Utrecht, under supervision of Prof.dr. Arno Siebes. In 2009, he finished this Ph.D. thesis. After that, he became a postdoctoral researcher in the same group, working on Exceptional Model Mining.

Besides doing research, there are some other activities that Matthijs finds relaxing. Two of these activities, sailing and kiting, both involve wind, which, luckily for him, is usually abundant in the Netherlands. Two other passions are photography and travelling, which he happily combines with either sailing or visiting conferences.

SIKS Dissertation Series

1998	1	J. van den Akker (CWI), DEGAS - An Active, Temporal Database of Autonomous Objects
	2	F. Wiesman (UM), Information Retrieval by Graphically Browsing Meta-Information
	3	A. Steuten (TUD), A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective
	4	D. Breuker (UM), Memory versus Search in Games
	5	E.W. Oskamp (RUL), Computerondersteuning bij Straftopmeting
1999	1	M. Sloof (VU), Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products
	2	R. Potharst (EUR), Classification using decision trees and neural nets
	3	D. Beal (UM), The Nature of Minimax Search
	4	J. Penders (UM), The practical Art of Moving Physical Objects
	5	A. de Moor (KUB), Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems
	6	N.J.E. Wijngaards (VU), Re-design of compositional systems
	7	D. Spelt (UT), Verification support for object database design
	8	J.H.J. Lenting (UM), Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation
2000	1	F. Niessink (VU), Perspectives on Improving Software Maintenance
	2	K. Holtman (TUE), Prototyping of CMS Storage Management
	3	C.M.T. Metselaar (UvA), Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief
	4	G. de Haan (VU), ETAG, A Formal Model of Competence Knowledge for User Interface Design
	5	R. van der Pol (UM), Knowledge-based Query Formulation in Information Retrieval
	6	R. van Eijk (UU), Programming Languages for Agent Communication
	7	N. Peek (UU), Decision-theoretic Planning of Clinical Patient Management
	8	V. Coupé (EUR), Sensitivity Analyis of Decision-Theoretic Networks
	9	F. Waas (CWI), Image database management system design considerations, algorithms and architecture
	11	J. Karlsson (CWI), Scalable Distributed Data Structures for Database Management
2001	1	S. Renooij (UU), Qualitative Approaches to Quantifying Probabilistic Networks
	2	K. Hindriks (UU), Agent Programming Languages: Programming with Mental Models
	3	M. van Someren (UvA), Learning as problem solving
	4	E. Smirnov (UM), Conjunctive and disjunctive version spaces with instance-based boundary sets
	5	J. van Ossenbruggen (VU), Processing Structured Hypermedia: A Matter of Style
	6	M. van Welie (VU), Task-based User Interface Design
	7	B. Schonhage (VU), Diva: Architectural Perspectives on Information Visualization
	8	P. van Eck (VU), A Compositional Semantic Structure for Multi-Agent Systems Dynamics
	9	P.J. 't Hoen (RUL), Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes
	10	M. Sierhuis (UvA), Modeling and Simulating Work Practice BRAHMS: a Multiagent Modeling and Simulation Language for Work Practice Analysis and Design
	11	T.M. van Engers (VU), Knowledge management: The Role of Mental Models in Business Systems Design
2002	1	N. Lassing (VU), Architecture-Level Modifiability Analysis
	2	R. van Zwol (UT), Modelling and Searching Web-based Document Collections
	3	H.E. Blok (UT), Database Optimization Aspects for Information Retrieval
	4	J.R. Castelo Valdueza (UU), The Discrete Acyclic Digraph Markov Model in Data Mining

	5	R. Serban (VU), The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents
	6	L. Mommers (UL), Applied legal epistemology
	7	P. Boncz (CWI), Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications
	8	J. Gordijn (VU), Value-based requirements engineering: exploring innovative e-commerce ideas
	9	W-J. van den Heuvel (KUB), Integrating Modern Business Applications with Objectified Legacy Systems
	10	B. Sheppard (UM), Towards Perfect Play of Scrabble
	11	W.C.A. Wijngaards (VU), Agent based modelling of dynamics: biological and organisational applications
	12	A. Schmidt (UvA), Processing XML in Database Systems
	13	H. Wu (TUE), A Reference Architecture for Adaptive Hypermedia Applications
	14	W. de Vries (UU), Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems
	15	R. Eshuis (UT), Semantics and Verification of UML Activity Diagrams for Workflow Modelling
	16	P. van Langen (VU), The Anatomy of Design: Foundations, Models and Applications
	17	S. Manegold (UvA), Understanding, Modeling, and Improving Main-Memory Database Performance
2003	1	H. Stuckenschmidt (VU), Ontology-based information sharing in weakly structured environments
	2	J. Broersen (VU), Modal Action Logics for Reasoning About Reactive Systems
	3	M. Schuemie (TUD), Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy
	4	M. Petkovic (UT), Content-Based Video Retrieval Supported by Database Technology
	5	J. Lehmann (UvA), Causation in Artificial Intelligence and Law - A modelling approach
	6	B. van Schooten (UT), Development and Specification of Virtual Environments
	7	M. Jansen (UvA), Formal Explorations of Knowledge Intensive Tasks
	8	Y. Ran (UM), Repair Based Scheduling
	9	R. Kortmann (UM), The Resolution of Visually Guided Behaviour
	10	A. Lincke (UT), Some Experimental Studies on the Interaction between Medium, Innovation context and Culture
	11	S. Keizer (UT), Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks
	12	R. Ordelman (UT), Dutch Speech Recognition in Multimedia Information Retrieval
	13	J. Donkers (UM), Nosce Hostem - Searching with Opponent Models
	14	S. Hoppenbrouwers (KUN), Freezing Language: Conceptualisation Processes across ICT-Supported Organisations
	15	M. de Weerdt (TUD), Plan Merging in Multi-Agent Systems
	16	M. Windhouwer (CWI), Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses
	17	D. Jansen (UT), Extensions of Statecharts with Probability, Time, and Stochastic Timing
	18	L. Kocsis (UM), Learning Search Decisions
2004	1	V. Dignum (UU), A Model for Organizational Interaction: Based on Agents, Founded in Logic
	2	L. Xu (UT), Monitoring Multi-party Contracts for E-business
	3	P. Groot (VU), A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving
	4	C. van Aart (UvA), Organizational Principles for Multi-Agent Architectures
	5	V. Popova (EUR), Knowledge Discovery and Monotonicity
	6	B-J. Hommes (TUD), The Evaluation of Business Process Modeling Techniques
	7	E. Boltjes (UM), Voorbeeldig onderwijs
	8	J. Verbeek (UM), Politie en de Nieuwe Internationale Informatiemarkt
	9	M. Caminada (VU), For the Sake of the Argument; Explorations into Argument-based Reasoning
	10	S. Kabel (UvA), Knowledge-rich Indexing of Learning-objects
	11	M. Klein (VU), Change Management for Distributed Ontologies
	12	T. Duy Bui (UT), Creating Emotions and Facial Expressions for Embodied Agents
	13	W. Jamroga (UT), Using Multiple Models of Reality: On Agents who Know how to Play
	14	P. Harrenstein (UU), Logic in Conflict. Logical Explorations in Strategic Equilibrium
	15	A. Knobbe (UU), Multi-Relational Data Mining
	16	F. Divina (VU), Hybrid Genetic Relational Search for Inductive Learning
	17	M. Winands (UM), Informed Search in Complex Games
	18	V. Bessa Machado (UvA), Supporting the Construction of Qualitative Knowledge Models
	19	T. Westerveld (UT), Using Generative Probabilistic Models for Multimedia Retrieval
	20	M. Evers (Nyenrode), Learning from Design: Facilitating Multidisciplinary Design Teams
2005	1	F. Verdenius (UvA), Methodological Aspects of Designing Induction-Based Applications
	2	E. van der Werf (UM), AI techniques for the Game of Go
	3	F. Grootjen (RUN), A Pragmatic Approach to the Conceptualisation of Language

4 N. Meratnia (UT), Towards Database Support for Moving Object data
5 G. Infante-Lopez (UvA), Two-Level Probabilistic Grammars for Natural Language Parsing
6 P. Spronck (UM), Adaptive Game AI
7 F. Frasincar (TUE), Hypermedia presentation generation for semantic web information systems
8 R. Vdovjak (TUE), A Model-driven Approach for Building Distributed Ontology-based Web Applications
9 J. Broekstra (VU), Storage, Querying and Inferencing for Semantic Web Languages
10 A. Bouwer (UvA), Explaining behaviour: using qualitative simulation in interactive learning
11 E. Ogston (VU), Agent Based Matchmaking and Clustering - A Decentralized Approach to Search
12 C. Boer (EUR), Distributed Simulation in Industry
13 F. Hamburg (UL), Een Computermodel voor het ondersteunen van euthanasiebeslissingen
14 B. Omelayenko (VU), Web-Service configuration on the Semantic Web; Exploring how Semantics meets Pragmatics
15 T. Bosse (VU), Analysis of the Dynamics of Cognitive Processes
16 J. Graaumans (UU), Usability of XML Query Languages
17 B. Shishkov (TUD), Software Specification Based on Re-usable Business Components
18 D. Sent (UU), Test-selection Strategies for Probabilistic Networks
19 M. van Dartel (UM), Situated Representation
20 C. Coteanu (UL), Cyber Consumer Law, State of the Art and Perspectives
21 W. Derks (UT), Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006 1 S. Angelov (TUE), Foundations of B2B Electronic Contracting
2 C. Chisalita (VU), Contextual issues in the Design and use of Information Technology in Organizations
3 N. Christoph (UvA), The role of Metacognitive Skills in Learning to Solve Problems
4 M. Sabou (VU), Building Web Service Ontologies
5 C. Pierik (UU), Validation Techniques for Object-Oriented Proof Outlines
6 Z. Baida (VU), Software-aided service bundling - intelligent methods tools for graphical service modeling
7 M. Smiljanic (UT), XML schema matching - balancing efficiency and effectiveness by means of clustering
8 E. Herder (UT), Forward, Back and Home Again - Analyzing User Behavior on the Web
9 M. Wahdan (UM), Automatic Formulation of the Auditor's Opinion
10 R. Siebes (VU), Semantic Routing in Peer-to-Peer Systems
11 J. van Ruth (UT), Flattening Queries over Nested Data Types
12 B. Bongers (VU), Interactivation - Towards an e-cology of people, our t-environment, and the arts
13 H-J. Lebbink (UU), Dialogue and Decision Games for Information Exchanging Agents
14 J. Hoorn (VU), Software requirements: update, upgrade, redesign
15 R. Malik (UU), CONAN: Text Mining in the Biomedical Domain
16 C. Riggelsen (UU), Approximation Methods for Efficient Learning of Bayesian Networks
17 S. Nagata (UU), User Assistance for Multitasking with Interruptions on a Mobile Device
18 V. Zhizhkun (UvA), Graph transformation for Natural Language Processing
19 B. van Riemsdijk (UU), Cognitive Agent Programming: A Semantic Approach
20 M. Velikova (UvT), Monotone models for prediction in data mining
21 B. van Gils (RUN), Aptness on the Web
22 P. de Vrieze (RUN), Fundaments of Adaptive Personalisation
23 I. Juvina (UU), Development of Cognitive Model for Navigating on the Web
24 L. Hollink (VU), Semantic Annotation for Retrieval of Visual Resources
25 M. Drugan (UU), Conditional log-likelihood MDL and Evolutionary MCMC
26 V. Mihajlovic (UT), Score Region Algebra: A Flexible Framework for Structured Information Retrieval
27 S. Bocconi (CWI), Vox Populi: generating video documentaries from semantically annotated media repositories
28 B. Sigurbjörnsson (UvA), Focused Information Access using XML Element Retrieval

2007 1 K. Leune (UvT), Access Control and Service-Oriented Architectures
2 W. Teepe (RUG), Reconciling Information Exchange and Confidentiality: A Formal Approach
3 P. Mika (VU), Social Networks and the Semantic Web
4 J. van Diggelen (UU), Achieving Semantic Interoperability in Multi-agent Systems
5 B. Schermer (UL), Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance
6 G. Mishne (UvA), Applied Text Analytics for Blogs
7 N. Jovanovic' (UT), To whom it may concern - addressee identification in face-to-face meetings
8 M. Hoogendoorn (VU), Modeling of Change in Multi-Agent Organizations
9 D. Mobach (VU), Agent-Based Mediated Service Negotiation
10 H. Aldewereld (UU), Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols

11 N. Stash (TUE), Incorporating cognitive learning styles in a gen-purpose adaptive hypermedia system
12 M. van Gerven (RUN), Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty
13 R. Rienks (UT), Meetings in Smart Environments; Implications of Progressing Technology
14 N. Bergboer (UM), Context-Based Image Analysis
15 J. Lacroix (UM), NIM: a Situated Computational Memory Model
16 D. Grossi (UU), Designing Invisible Handcuffs
17 T. Charitos (UU), Reasoning with Dynamic Networks in Practice
18 B. Orriens (UvT), On the development an management of adaptive business collaborations
19 D. Levy (UM), Intimate relationships with artificial partners
20 S. Jansen (UU), Customer Configuration Updating in a Software Supply Network
21 K. Vermaas (UU), Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005
22 Z. Zlatev (UT), Goal-oriented design of value and process models from patterns
23 P. Barna (TUE), Specification of Application Logic in Web Information Systems
24 G. Ramírez Camps (CWI), Structural Features in XML Retrieval
25 J. Schalken (VU), Empirical Investigations in Software Process Improvement

2008 1 K. Boer-Sorbán (EUR), Agent-Based Simulation of Financial Markets
2 A. Sharpanskykh (VU), On computer-aided methods for modeling and analysis of organizations
3 V. Hollink (UvA), Optimizing hierarchical menus: a usage-based approach
4 A. de Keijzer (UT), Management of Uncertain Data - towards unattended integration
5 B. Mutschler (UT), Modeling and simulating causal dependencies on process-aware information systems from a cost perspective
6 A. Hommersom (RUN), On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective
7 P. van Rosmalen (OU), Supporting the tutor in the design and support of adaptive e-learning
8 J. Bolt (UU), Bayesian Networks: Aspects of Approximate Inference
9 C. van Nimwegen (UU), The paradox of the guided user: assistance can be counter-effective
10 W. Bosma (UT), Discourse oriented summarization
11 V. Kartseva (VU), Designing Controls for Network Organizations: A Value-Based Approach
12 J. Farkas (RUN), A Semiotically Oriented Cognitive Model of Knowledge Representation
13 C. Carraciolo (UvA), Topic Driven Access to Scientific Handbooks
14 A. van Bunningen (UT), Context-Aware Querying; Better Answers with Less Effort
15 M. van Otterlo (UT), The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains
16 H. van Vugt (VU), Embodied agents from a user's perspective
17 M. Op 't Land (TUD), Applying Architecture and Ontology to the Splitting and Allying of Enterprises
18 G. de Croon (UM), Adaptive Active Vision
19 H. Rode (UT), From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search
20 R. Arendsen (UvA), Geen bericht, goed bericht
21 K. Balog (UvA), People Search in the Enterprise
22 H. Koning (UU), Communication of IT-Architecture
23 S. Visscher (UU), Bayesian network models for the management of ventilator-associated pneumonia
24 Z. Aleksovski (VU), Using background knowledge in ontology matching
25 G. Jonker (UU), Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency
26 M. Huijbregts (UT), Segmentation, diarization and speech transcription: surprise data unraveled
27 H. Vogten (OU), Design and Implementation Strategies for IMS Learning Design
28 I. Flesch (RUN), On the Use of Independence Relations in Bayesian Networks
29 D. Reidsma (UT), Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans
30 W. van Atteveldt (VU), Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content
31 L. Braun (UM), Pro-Active Medical Information Retrieval
32 T.H. Bui (UT), Toward Affective Dialogue Management using Partially Observable Markov Decision Processes
33 F. Terpstra (UvA), Scientific Workflow Design; theoretical and practical issues
34 J. De Knijf (UU), Studies in Frequent Tree Mining
35 B.T. Nielsen (UvT), Dendritic morphologies: function shapes structure

2009 1 R. Jurgelenaite (RUN), Symmetric Causal Independence Models
2 W.R. van Hage (VU), Evaluating Ontology-Alignment Techniques
3 H. Stol (UvT), A Framework for Evidence-based Policy Making Using IT

	4	J. Nabukenya (RUN), Improving the Quality of Organisational Policy Making using Collaboration Engineering
	5	S. Overbeek (RUN), Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality
	6	M. Subianto (UU), Understanding Classification
	7	R. Poppe (UT), Discriminative Vision-Based Recovery and Recognition of Human Motion
	8	V. Nannen (VU), Evolutionary Agent-Based Policy Analysis in Dynamic Environments
	9	B. Kanagwa (RUN), Design, Discovery and Construction of Service-oriented Systems
	10	J.A.N. Wielemaker (UvA), Logic programming for knowledge-intensive interactive applications
	11	A. Boer (UvA), Legal Theory, Sources of Law & the Semantic Web
	12	P. Massuthe (TUE, Humboldt-Universitaet zu Berlin), Operating Guidelines for Services
	13	S. de Jong (UM), Fairness in Multi-Agent Systems
	14	M. Korotkiy (VU), From ontology-enabled services to service-enabled ontologies
	15	R. Hoekstra (UvA), Ontology Representation - Design Patterns and Ontologies that Make Sense
	16	F. Reul (UvT), New Architectures in Computer Chess
	17	L. van der Maaten (UvT), Feature Extraction from Visual Data
	18	F. Groffen (CWI), Armada, An Evolving Database System
	19	V. Robu (CWI), Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets
	20	B. van der Vecht (UU), Adjustable Autonomy: Controling Influences on Decision Making
	21	S. Vanderlooy (UM), Ranking and Reliable Classification
	22	P. Serdyukov (UT), Search For Expertise: Going beyond direct evidence
	23	P. Hofgesang (VU), Modelling Web Usage in a Changing Environment
	24	A. Heuvelink (VU), Cognitive Models for Training Simulations
	25	A. van Ballegooij (CWI), RAM: Array Database Management through Relational Mapping
	26	F. Koch (UU), An Agent-Based Model for the Development of Intelligent Mobile Services
	27	C. Glahn (OU), Contextual Support of social Engagement and Reflection on the Web
	28	S. Evers (UT), Sensor Data Management with Probabilistic Models
	29	S. Pokraev (UT), Model-Driven Semantic Integration of Service-Oriented Applications
	30	M. Zukowski (CWI), Balancing vectorized query execution with bandwidth-optimized storage
	31	S. Katrenko (UvA), A Closer Look at Learning Relations from Text
	32	R. Farenhorst and R. de Boer (VU), Architectural Knowledge Management: Supporting Architects and Auditors
	33	K. Truong (UT), How Does Real Affect Affect Affect Recognition In Speech?
	34	I. van de Weerd (UU), Advancing in Software Product Management: An Incremental Method Engineering Approach
	35	W. Koelewijn (UL), Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling
	36	M. Kalz (OUN), Placement Support for Learners in Learning Networks
	37	H. Drachsler (OUN), Navigation Support for Learners in Informal Learning Networks
	38	R. Vuorikari (OU), Tags and self-organisation: a metadata ecology for learning resources in a multilingual context
	39	C. Stahl (TUE, HU Berlin), Service Substitution – A Behavioral Approach Based on Petri Nets
	40	S. Raaijmakers (UvT), Multinomial Language Learning: Investigations into the Geometry of Language
	41	I. Berezhnyy (UvT), Digital Analysis of Paintings
	42	T. Bogers (UvT), Recommender Systems for Social Bookmarking
	43	V. Nunes Leal Franqueira (UT), Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients
	44	R. Santana Tapia (UT), Assessing Business-IT Alignment in Networked Organizations
	45	Jilles Vreeken (UU), Making Pattern Mining Useful
2010	1	Matthijs van Leeuwen (UU), Patterns that Matter