Universiteit Utrecht

The research reported in this thesis has been supported by the Ministry of Communication and Information Technology of the Republic of Indonesia (Depkominfo) and Utrecht University.

Printed by Ridderprint, the Netherlands.

ISBN: 978-90-393-5852-8

Patterns, Models, and Queries

Patronen, Modellen en Vragen
(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. G.J. van der Zwaan, ingevolge het besluit van het college voor promoties in het openbaar te verdedigen op dinsdag 31 oktober 2012 des middags te 2.30 uur

door

Diyah Puspitaningrum

geboren op 5 oktober 1976
te Semarang, Indonesië

Promotor: Prof.dr. A.P.J.M. Siebes
Co-promotor: dr. M. van Leeuwen

Contents

Chapter 1

Introduction

Over the past decade, the field of data mining has grown tremendously. Since the introduction of frequent itemsets and association rules by Agrawal et al. in 1993 [1], hundreds of research papers have been published presenting, amongst others, new algorithms to mine frequent patterns efficiently. The number of applications is also growing very fast. From the original motivation for association rules that came from the need to analyse client's purchasing behaviour in a supermarket, to *Bioinformatics* for gene expression analysis and prediction of protein structure, to *automated recommendation systems* [53].

Next to the advances in algorithms, there are advances in hardware (storage, processors, etc) and networking that make large-scale data analysis feasible. Together they keep the data explosion under control. Sharing huge volumes of data is, however, a problem. First and foremost because of privacy and security, it is not necessarily allowed or preferable to share data with a competitor; even if they want to collaborate. Such problems may be partially alleviated if, rather than sharing data, companies can share models of the data. This will be even more useful if these models allow one to answer queries on the data; without access to the data, of course.

This is exactly the main focus of this thesis: how to generate answers to queries in this case? Both on the standard rectangular tables and on *itemset* databases.

1.1 Pattern Mining and Compression with Patterns

Our work is in the area known as *pattern mining*. We search for interesting regularities in the data. One can find such patterns in a database. A common interpretation for the *interestingness* of a pattern is its *frequency*, viz. if it occurs at least a certain number of times in the database.

By fine-tuning the minimal frequency threshold, one can choose one of two options: mining only very frequent patterns by setting the minimum support to a high value, or, lowering the minimum support which leads to very many patterns. The choice depends on one's goal. If one intends to know only a summary of a database, i.e. what the data is about, one should choose the first. For our goals, however, knowing very detailed relationships is an advantage. However we have to be careful because when a low minimum support is chosen, the result can be a humongous number of patterns; this known as the *pattern explosion*.

In this thesis we use the pattern mining tool Krimp [73] to reduce this very large set of patterns. Krimp is based on the Minimum Description Length (MDL)principle. MDL is related to *Kolmogorov complexity* [45]. The complexity of a sequence is defined as the length of the shortest program that prints the sequence and then halts. The lower the Kolmogorov complexity of a sequence, the *more regular* it is.

A good example of this, taken from [23], is as follows. Consider the following three sequences. Assume each sequence is 10000 bits long, and we just list the beginning and the end of each sequence.

$$0001000010001000 \ldots 0001000100010001000100010001 \quad (1.1)$$
$$0111010011010010 \ldots 1010111010111011000101100010 \quad (1.2)$$
$$0001100000101010000 \ldots 0010001000010000001000110000 \quad (1.3)$$

If we look at the regularity of the above data, the first sequence is a 2500-fold repetition of 0001, thus regular. Any future data based on it will likely also follow this regularity. The second sequence is like one generated by tossing a fair coin, or as 'random as possible'; no regularity is found underlying the data. The third sequence looks less regular and more random than the first, but also less random than the second. There is still some regularity to be found, and it is reasonable to expect that future data derived from it will behave according to the same regularity.

A program that represents the first sequence is:

for i = 1 to 2500; print '0001';

That is, we can compress sequence 1 to $O(\log n)$ bits where n is the length of the given sequence.

Meanwhile for the second sequence, which is an extremely random sequence, the shortest program that prints it has a size about equal to the length of the sequence itself; it cannot be compressed at all.

The third sequence lies between the two, it can be compressed to some length αn, with $0<\alpha<1$. Any type of regularity like of the first and the third sequence can be used to compress a database.

Using similar principles, MDL selects a model that compresses the database best. The rationale behind the choice of MDL for our research is in a sense the rationale for MDL itself: *the better a model compresses a database, the better it captures the regularities of that database*. Moreover, MDL does not model noise because capturing spurious patterns lead to an increase of the complexity of a model and, thus, does not lead to better compression. This topic is further discussed in Chapter 2, where the KRIMP algorithm is introduced.

1.2 Recommender Systems

In popular social media sharing sites such as YouTube and Flickr, there are many billions of images and videos annotated by millions of users. These very big databases of "collective knowledge" contain a lot of useful knowledge. Consider, as in [58], tag recommender systems. There are two tasks for which recommender systems are particularly useful. In the first scenario, a user annotating a photo is recommended tags related to that photo that can be used to extend the existing annotation. In the second scenario, the role of the recommender system is to provide search recommendations. The latter can be done by automatic query expansion, or by an interactive process with search assistants that provide additional query terms. The collective knowledge allows us to build accurate recommendation systems on a broad range of topics.

Frequent association rules induced from such a collective knowledge database can be used to provide accurate recommendations. However, such a recommendation system suffers in efficiency and scalability. One of topics in this thesis is about tag recommendation, in which we show that the patterns induced by the KRIMP algorithm can alleviate these problems i.e. using *only* these patterns we still can generate accurate recommendations. The advantage is that while pattern selection is done offline, online recommendations can be generated in a split second.

Since sites as YouTube and Flickr are not only sharing sites, but also social media sites, their databases contain even more knowledge. In particular, we show that if we take the user's social contacts into account, we can create even better, and personalised recommendations.

1.3 Research Problem

One of the most important features – if not the defining feature – of any database system is that it supports queries. If one wants to know information that is stored in the database, one doesn't have to search and combine stored information manually. Rather a, usually, succinct formulated query will deliver exactly the information you need.

Next to querying, one of the prime uses of databases, and *the* prime use of data warehouses, is *data mining*: searching for patterns in models of (part of) the data. The so-called data explosion means that we do not only get more and more databases, but the databases also gets bigger and bigger. Hence, it becomes more and more difficult to get insight in the data by means of queries only. That is where data mining steps in: the models and patterns provide insight that is hard to get with queries.

Given that both querying and data mining are important usages of a database, it is natural to wonder about their interaction. Part of that interaction is the topic of this thesis. More precisely, the research problem it addresses is as follows.

> What does a model of the data tell you about the results of a query on the data?

That is, suppose we have a model M of – or a set of patterns on – a database D and now we wish to compute a query Q on D, what do we know about $Q(D)$ a priori, i.e., before (or even without) computing $Q(D)$.

To make this problem more specific it is split into three sub-problems as follows:

1. Does having a model of a database help you in building a model of the result of a query on that database?

2. Can you use a model of a database to compute the answer to a descriptive query?

3. Can you use a model of a database to compute the answer to a predictive query?

These problems are addressed in the context of *itemset* databases. However, since for categorical data the usual relational data tables can easily be seen as itemset data, our results also hold for such databases. In fact, for the first two sub-problems we are mainly interested in such categorical databases.

The models we consider for all three sub-problems are the code tables that Krimp computes.

1.4 Approach and Contributions

To answer our three questions above, we divide our chapters into three parts: *Modeling A Query*: Chapter 3; this Chapter is a unification of the research from two earlier papers [61, 62]. *Answering Descriptive Queries* : Chapter 4 *Answering Predictive Queries*: Chapter 5 and 6; Chapter 5 is published in [43],

the results of Chapter 6 will be submitted to a suitable conference or journal. Chapter 2 provides background material on, e.g., the KRIMP algorithm, which is used in the remainder of this thesis.

Modelling A Query

The first (sub-)problem we consider in this thesis is:

> Does having a model of a database help you in building a model of the result of a query on that database?

Clearly, this is still a vague question. Being slightly more concrete, let M_D be the model we induced from database D and let Q be a query on D. Does knowing M_D help in inducing a model M_Q on $Q(D)$. But even this is vague: what do we mean by help? In Chapter 3, we formalized "help" in two different ways.

Firstly, in the sense that we can compute M_Q directly from M_D *without* consulting either D or $Q(D)$. Secondly we interpreted "help", far less ambitiously, as meaning "speeding-up" the computation of M_Q. That is, let $\mathcal{A}lg$ be the algorithm used to induce M_D from D, i.e., $\mathcal{A}lg(D) = M_D$. We want to transform $\mathcal{A}lg$ into an algorithm $\mathcal{A}lg^*$, which takes M_D as extra input such that

$$\mathcal{A}lg^*(Q(D), M_D) \approx \mathcal{A}lg(Q(D))$$

We investigate both approaches for itemset mining. The main result is that KRIMP does well in the second approach.

Answering Descriptive Queries

The second (sub-)problem we consider in this thesis is:

> Can you use a model of a database to compute the answer to a descriptive query?

The term *descriptive* query is used to emphasize that we want to generate the answer to a query on a database. It is not about *predictions*. In Chapter 4, we research this problem using KRIMP on categorical relational databases.

The question is then, informally: can we use CT – computed by KRIMP on D – to generate answers to an arbitrary query Q on D? That is, rather than accessing D to compute $Q(D)$ we want to generate an answer: we want to generate a dataset $A(Q|CT)$ such that

$$A(Q|CT) \approx Q(D)$$

The solution in Chapter 4 is a very simple adaptation of a general method to generate transactions from a given code table CT. The adaptation is basically a sanity check: will the tuple generated adhere to the schema prescribed by the query Q. If so, it is part of the answer, if not it is not.

Answering Predictive Queries

The third and final (sub-)problem we consider in this thesis is:

> Can you use a model of a database to compute the answer to a predictive query?

In contrast to the previous sub-problem, here we want to make predictions. That is, we are querying the underlying data distribution rather than the database itself. This problem is researched in Chapters 5 and 6, both in the context of *tag recommendation*.

Collaborative tagging services allow users to freely assign tags to resources. As the large majority of users enters only very few tags, good tag recommendation can vastly improve the usability of tags for techniques such as searching, indexing, and clustering. Previous research has shown that accurate recommendation can be achieved by using conditional probabilities computed from tag associations. The main problem, however, is that enormous amounts of associations are needed for optimal recommendation.

To overcome this problem, we use the strengths of pattern selection in Chapter 5. Using Krimp as 'off-the-shelf' pattern selection method, we have proposed a new method called FastAR and we demonstrate that our FastAR method gives a very favourable trade-off between runtime and recommendation quality. Because pattern selection is done offline and results in small pattern sets, online recommendation is fast.

Social networking sites such as Flickr and Facebook allow users to share content with family, friends, and interest groups. That is, next to the tag sets we exploit in Chapter 5, there is also a social network. Not taking this information into account probably does not result in optimal recommendations. That is the issue we research in Chapter 6.

To address the problem, we propose two approaches that take a more social view on tag recommendation. As we do not want to change a method that works well, instead of modifying the algorithm we vary the data that is used as source of associations. We dub the first social variant User-centered Knowledge, to contrast Collective Knowledge. It improves tag recommendation by grouping historic tag data according to friend relationships and interests. The second variant is dubbed 'social batched personomies' and attempts to address

both quality and scalability issues by processing queries in batches instead of individually, such as done in a regular personomy approach.

Chapter 2

Pattern Mining

Pattern mining or finding interesting patterns in data is a successful area in data mining for a few decades already. Many scalable algorithms have been proposed to mine frequent itemsets efficiently. Moreover, there is a wide variety of applications. While understanding the patterns itself is interesting, applications such as classification, clustering, and imputation are as challenging as finding an efficient mining method. As a further example, finding the best kind of summarisation for recommender system is a challenging task for *association rules* pattern mining.

In this chapter we shall provide a brief overview of:

- various forms of pattern mining,
- itemset mining (especially algorithms for mining frequent itemsets),
- the minimum description length (MDL) principle, and
- the KRIMP algorithm, a pattern mining algorithm which is based on MDL.

We will conclude with an introduction of pattern mining on relational databases.

2.1 Pattern Mining

Pattern mining is one of the most important concepts in data mining. In contrast to *models*, patterns describe only part of the data [28, 54]. In this section we shall describe *theory mining* [50], in which the patterns describe interesting subsets of the database.

Given a database 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 D or not, the task is to find *all* interesting subsets:

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

Frequent set mining is the most popular instance of *theory mining*. Its standard example is market basket data analysis, i.e., finding items frequently purchased together to increase sales and reduce costs. Other examples for this data mining task are:

- classification: (*"is customer X a diaper buyer?"*),
- clustering: (*"which groups of customers buy similar goods?"*),
- recommendation: (*"if customer X is buying item I_1 and I_2 what items will customer X buy next?"*).

Frequent Itemset Mining

Let $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ be the set of all items. A transaction is a set of items: $t \subseteq \mathcal{I}$. A transaction database D is a bag of transactions, $|D|$ is the number of transactions it contains. See Table 2.1 for an example of such a transaction database.

An itemset pattern, denoted by F, is defined as a set of items: $F \subseteq \mathcal{I}$. The *cardinality* of F is defined as the number of items it contains. For example, singletons have cardinality 1, while all non-singletons have a cardinality of 2 or larger. An itemset F *occurs* in transaction $t \in D$ iff $F \subseteq t$.

Given a database D, we could mine for *all* patterns that occur in it. However, this can lead to humongous pattern sets, i.e., to a *pattern explosion*. To cope with this problem we restrict ourselves to mining *frequent* itemsets only.

The *cover* of an itemset X in D, denoted by $cover_D(X)$, consists of the set of transactions in D in which it occurs:

$$cover_D(X) = \{t \in D \mid X \subseteq t\}.$$

The *support* of an itemset X in D is the number of transactions in the cover of X in D:

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

The *frequency* of an itemset X in D is the probability of X occurring in a transaction $T \in D$:

$$frequency_D(X) = P(X) = \frac{sup_D(X)}{|D|}.$$

Frequent itemsets are now defined as itemsets whose support is larger than or equal to some user-defined minimal support threshold *minsup*, which is sometimes denoted by θ.

Association rules in D are rules of the form $X \Rightarrow Y$, in which both X and Y are itemsets in D. The support of such an association rule is the support of the itemset $X \cup Y$. Its *confidence*, or *accuracy*, is the conditional probability of having Y contained in a transaction, given that X is contained in that transaction:

$$confidence_D(X \rightarrow Y) = P(Y \mid X) = \frac{sup_D(X \cup Y)}{sup_D(X)}.$$

The rule is called *confident* if $P(Y|X)$ exceeds a given *minimal confidence threshold* γ, with $0 \leq \gamma \leq 1$. Tables 2.1, 2.2, 2.3 provide some examples of the above definitions.

Table 2.1: An example transaction database D that consists of 4 tuples over the set of items $\mathcal{I} = \{\text{beer}, \text{chips}, \text{pizza}, \text{wine}\}$.

tid	X
100	{beer,chips,wine}
200	{beer,chips}
300	{pizza,wine}
400	{chips,pizza}

Association rule algorithm

By setting the two thresholds (*minsup* and *minimal confidence threshold*) for a database D, the association rule mining task is specified. In general, association rule mining algorithms take a two-phase strategy. The first phase generates all frequent itemsets, and the second phase than generates all *frequent* and *confident* association rules from this set of frequent itemsets.

To mine the frequent itemsets, we have the *A Priori* property:

$$X \subseteq Y \Rightarrow sup_D(Y) \leq sup_D(X).$$

Hence, a simple level-wise search suffices to compute all the frequent itemsets [49]. The term *efficient* is used here with respect to the output size, which can be exponential in the number of items.

Table 2.2: Itemsets and their support (see database D in Table 1). Setting $minsup$=1.

Itemset	Cover	Support	Frequency
{}	{100,200,300,400}	4	100%
{beer}	{100,200}	2	50%
{chips}	{100,200,400}	3	75%
{pizza}	{300,400}	2	50%
{wine}	{100,300}	2	50%
{beer,chips}	{100,200}	2	50%
{beer,wine}	{100}	1	25%
{chips,pizza}	{400}	1	25%
{chips,wine}	{100}	1	25%
{pizza,wine}	{300}	1	25%
{beer,chips,wine}	{100}	1	25%

Some basic frequent itemset miners

There are many frequent-itemset mining algorithms. Many address aspects such as memory and computational efficiency [27, 76]. We briefly introduce three basic algorithms, viz., *A Priori* [3], *FP-Growth* [20] and *Eclat* [25]. There is also a trend towards algorithms that do not mine for all frequent sets but only some reduced set [25]. This will be discussed briefly in the next section as well as in our introduction of the Krimp algorithm in Section 2.3.

Frequent itemset mining algorithms have to scale to large databases. The first such algorithm was developed by Agrawal and Srikant, who based their *A Priori* algorithm [3] on the Apriori property: *a k-itemset can only be frequent only if all of its sub-itemsets are frequent.* Using this property, frequent itemsets can be mined by first scanning the database to find the frequent 1-itemsets, then using the frequent 1-itemsets to generate candidate frequent 2-itemsets, and check against the database to obtain the frequent 2-itemsets. The process is continued until no more frequent k-itemsets can be generated for some k. Even though this algorithm significantly reduces the size of candidate sets, it suffers two drawbacks. Firstly, it generates a large number of candidate sets. Secondly, it scans the database often to check the candidates by pattern matching.

FP-growth algorithm works with a *divide-and-conquer* approach. FP-growth generates far less candidates than A Priori. In this way it substantially reduces search time. It first scans the database to derive a list of frequent items, in which the items are ordered by frequency descending order. Using this

Table 2.3: Association rules and their support and confidence in D. Setting *minsup*=1 and *minimal confidence value* γ=0.5

Rule	Support	Frequency	Confidence
{beer} ⇒ {chips}	2	50%	100%
{beer} ⇒ {wine}	1	25%	50%
{chips} ⇒ {beer}	2	50%	66%
{pizza} ⇒ {chips}	1	25%	50%
{pizza} ⇒ {wine}	1	25%	50%
{wine} ⇒ {beer}	1	25%	50%
{wine} ⇒ {chips}	1	25%	50%
{wine} ⇒ {pizza}	1	25%	50%
{beer,chips} ⇒ {wine}	1	25%	50%
{beer,wine} ⇒ {chips}	1	25%	100%
{chips,wine} ⇒ {beer}	1	25%	100%
{beer} ⇒ {chips,wine}	1	25%	50%
{wine} ⇒ {beer,chips}	1	25%	50%

frequency-descending list, the database is compressed into a frequent-pattern tree (FP-tree), which retains the itemset association information. The FP-tree is mined by starting from each frequent length-1 pattern as an initial suffix pattern. We construct a *conditional pattern base*, viz. a sub-database that consists of the set of prefix paths in the FP-tree co-occurring with the suffix pattern. Then we construct the conditional FP-tree of this conditional pattern base and mine recursively on a tree. The pattern growth is achieved by the concatenation of the suffix pattern with the frequent patterns generated from a conditional FP-tree.

Eclat, or Equivalence CLASS Transformation [76], takes advantage of the Apriori property while retaining FP-growth like computation times. Different from Apriori and FP-growth that mine frequent patterns in *horizontal data format*, Eclat is an example of a mining algorithm in *vertical data format*. In Eclat, the first scan of the database builds the transaction-ID set of each single item (tids(X), in a vertical layout). Starting initially from all single items (k = 1), the frequent (k+1)-itemsets are generated from previous k-itemsets based on the Apriori property. Because of its depth-first approach, its computation time is similar to FP-growth. The actual computation is done by intersection of the tids(X) of the frequent k-itemsets to compute the tids(X) of the corresponding (k+1)-itemsets; i.e., no additional database scans are necessary. Eclat iterates until no frequent itemsets or no candidate itemsets are found.

Database Issues

With a horizontal data layout, counting the support of a candidate set X requires a complete scan of the database, testing every transaction t whether or not $X \subseteq t$. Scanning a database is I/O intensive, whether reading the transaction from a file or from a database cursor. The major costs of this approach, however, are in updating the supports of all candidate sets contained in a transaction.

The vertical data layout has the major advantage that the support of a set X can be easily computed by intersecting the covers of any two subsets Y, $Z \subseteq X$, such that $Y \cup Z = X$ [30, 60]. To do this efficiently for a given set of candidate itemsets, the covers of many itemsets have to be in main memory, which may not always be possible.

Search Space Issues

Given the set of all items $\mathcal{I}$, the search space of all itemsets contains $2^{|\mathcal{I}|}$ different sets. If $\mathcal{I}$ contains thousands of items, the number of itemsets explodes. Compare it, e.g., to the number of atoms in the universe which is only $\approx 10^{79}$. The trick is to generate as few candidate sets as possible. In the ideal case, one would generate and count frequent itemsets only. Note that most implementations break down when the candidate sets exceed available main memory.

Condensed Representations

As previously stated, if the number of frequent sets for a given database is large, it will be infeasible to generate them all. If a database is dense, or the *minsup* is too low, then one will swamped with lots and lots of frequent sets; for there are 2^k-1 different non-empty frequent subsets of a frequent set of size k. To overcome this problem, several proposals have been made to generate a good representative collection of the frequents sets in a database. The three most popular ones are: *maximal frequent patterns* [6], *closed frequent patterns* [57], and *non-derivable frequent patterns* [9]. In this thesis we will mainly use *closed frequent patterns*.

- *we call a pattern F a* maximal *frequent pattern given a database D iff F is frequent and there exists no pattern F' such that F ⊂ F' and F' is frequent in D.*

- *we call a pattern F a* closed *frequent pattern given a database D iff F is frequent and there exist no pattern F' such that F ⊂ F' and* $\sup(F) = \sup(F')$. *The Maximal frequent pattern set* is lossy: it is more compact than *closed* but does not contain the complete frequency information for

the set of all frequent patterns. The *closed frequent pattern* set is a lossless representation of the complete frequent pattern set.

- *For* non-derivable frequent itemsets *[20], the idea is to generate only those itemsets whose support cannot be derived from others, since their derivable counterparts essentially give no new information about a database.*

  NDI [9] is the name of an *A Priori* style algorithm that is adapted to generate only the non-derivable frequent itemsets by implementing the inclusion-exclusion formulas. More in particular, if $A_1, \ldots, A_n$ are finite sets, the inclusion-exclusion formula states that:

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{1\leq j\leq n} |A_i| - \sum_{1\leq i<j\leq n} \left|A_i \bigcap A_j\right| + \cdots - (-1)^n \left|\bigcap_{i=1}^{n} A_i\right|$$

  Given the support of some item sets, this formula allows us to compute an upperbound and a lowerbound of other item sets. Whenever, the upper and lowerbound coincide, an item set is called derivable.

2.2 Minimum Description Length

The Minimum Description Length (MDL) principle is a method for inductive inference, or better, for the model selection problem. The basic idea of MDL is trying to find regularity in the data. 'Regularity' here may be identified with the 'ability to compress' (*viewing learning as data compression*). For a given set of hypotheses $\mathcal{H}$ and dataset D, one should try to find the hypothesis in $\mathcal{H}$ that compresses D most.

MDL (Minimum Description Length), first introduced in [59], see also [23], is closely related to MML (Minimum Message Length) [74] and also to Kolmogorov Complexity [45]; in fact, one could see MML as fully *Bayesian* MDL. All three embrace the slogan *Induction by Compression.* Below is a brief description of MDL principle.

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

$$L(H) + L(D|H)$$

in which

- $L(H)$ is the length, in bits, of the description of H, and
- $L(D|H)$ 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 [24]. Refined MDL has a major weakness for our purposes, although it has stronger theoretical foundations: it cannot be computed except for some special cases [24]. Hence, we use *crude* MDL. For modelling a database we use *crude* MDL, that is for compression purpose. MDL finds the set of frequent itemsets that yields the best compression.

Central in our approach is the notion of a *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. Using such a table we can encode and decode databases. This is were MDL comes in: we search for the code table that compresses the data best.

More formally, code tables are introduced as follows; see also [73].

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.

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*, denoted CS. For the number of itemsets in the code table we write $|CT|$, i.e. we define $|CT| = |\{X \in CT\}|$. Likewise, $|CT \backslash \mathcal{I}|$ indicates the number of non-singleton itemsets in the code table. See Figure 2.1 for an illustration. To encode a transaction t from database 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.

Definition 2 Let D be a database over a set of items $\mathcal{I}$, t a transaction drawn from 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, $cover : \mathcal{CT} \times \mathcal{P}(\mathcal{I}) \longmapsto \mathcal{P}(\mathcal{P}(\mathcal{I}))$ 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. That is, if $X \in cover(CT,t)$ then $X \in CT$

2. if $X,\ Y \in cover(CT,t)$, then either $X = Y$ or $X \cap Y = \emptyset$

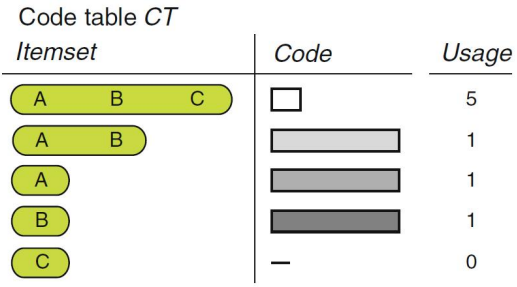

Figure 2.1: Example code table [73]. The widths of the codes represent their length. $\mathcal{I} = \{A,B,C\}$, $|CT|$=5, $|CT \setminus \mathcal{I}| = 2$. Note that the *usage* column is not part of the code table, but shown here as illustration: for optimal compression, codes should be shorter the more often they are used

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 $\mathcal{I}$. Figure 2.2 provides an example of a *cover*. There are multiple reasons for not allowing the itemsets in the cover in a transaction to overlap. It ensures an unambiguous interpretation and it reduces the number of possible covers.

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

$$t \rightarrow \{code_{CT}(X) | X \in cover(CT,t)\}.$$

Since code tables contain the singletons by definition, each code table can encode any database. To ensure that encoded databases can be decoded unambiguously, $\mathcal{C}$ should be a *prefix code*, i.e., no code is the prefix of another code [14] (see Figure 2.3).

There is a well-known correspondence between code lengths and probability distributions for prefix codes [45]. Let P be any probability distribution on a finite set D than there exists a prefix code $\mathcal{C}$ for D such that for $d \in D$:

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

This code is optimal in the sense that on average "messages" drawn according to D get the shortest encoding with $\mathcal{C}$, [14].

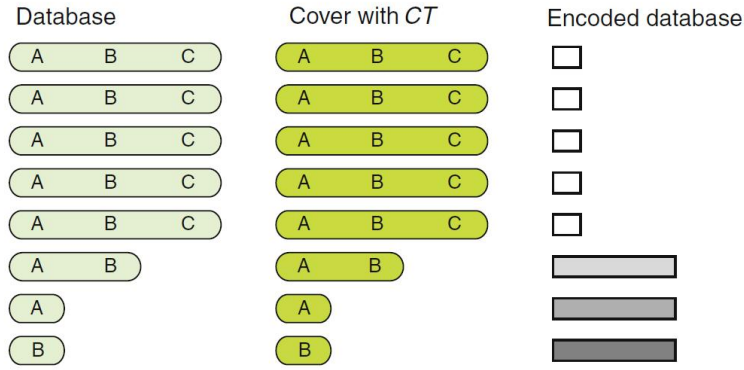

Figure 2.2: Example database [73], and the cover and encoded database obtained by using the code table shown in Fig. 2.1. $\mathcal{I} = \{A,B,C\}$. See that there are 4 codes occuring in the database: $[\{A,B,C\},\{A,B\},\{A\},\{B\}]$ with frequencies = [5,1,1,1]. The more frequent an itemset the shorter its code is for optimal compression.

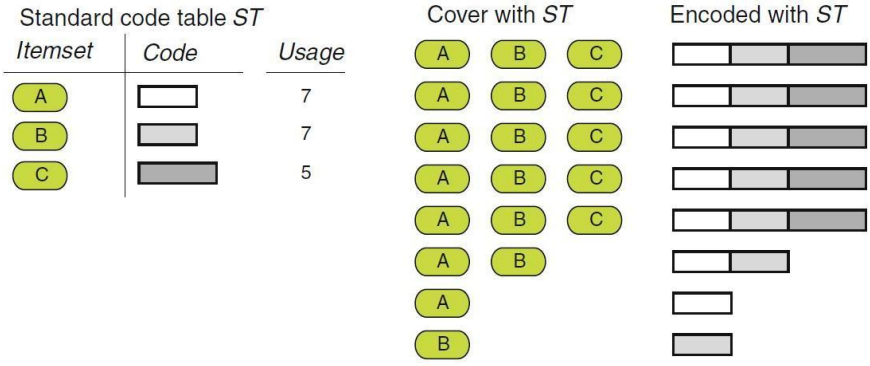

Figure 2.3: Example standard code table for the database in Fig. 2.2, with associated cover and encoded database

In our case, our "messages" are the transactions in D. To be more specific, our messages are the patterns in the cover set over all transactions. This optimality property thus tells us that if we encode with such a Shannon code, we introduce no bias in our model selection process. The probability distribution we use is induced by the cover function. It is defined as the relative usage frequency of the elements of the code table. In more detail it is defined as follows.

Definition 3 Let 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

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

This implies a *probability* distribution on $X \in CT$ given D, by

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

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

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

A code table CT is *code-optimal* for D iff all its codes are optimal for D:

$$\{code_{CT}(X) | X \in CT\}.$$

From now on, all code tables we consider are code-optimal unless stated differently.

Lemma 1 *Let D be a transaction database over $\mathcal{I}$, CT be a code table over $\mathcal{I}$ and code-optimal for D,* cover *a cover function, and* usage *the usage function for cover.*

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

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

2. *The encoded size of D, in bits, when encoded by CT, denoted by $L(D|CT)$, is*

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

So, we know how to compute the size of the database and the size of the right-hand side of the code table is also no problem. But, what about the size of the left-hand column? For this we use the *standard code table* for D, denoted by ST. This is the simplest code table, i.e., the one that contains the singleton sets only.

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

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

Note that itemsets with zero usage are ignored. Thus the total size of our encoded database is simply the size of encoded database plus the size of the code table.

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

$$L(D, CT) = L(D|CT) + L(CT|D).$$

Our goal is now to find the code table that minimises $L(D, CT)$, in line with the MDL principle. Unfortunately, computing the optimal code table is infeasible. In the next section we introduce a heuristic algorithm that finds good, rather than optimal, code tables.

2.3 The Krimp Algorithm

The Krimp algorithm [73] employs four heuristics to find good code tables, viz., it uses a greedy search, considering each candidate only once starting from the *Standard Code Table*, using the *Standard Candidate Order*, and the *Standard Cover Order* function. It only accepts patterns that give a better compressed size. In this section we outline the main components of Krimp.

Standard Code Table

The standard code table is computed by Algorithm 1. It simply adds all the singletons, computes their support and optimal code.

Algorithm 1 The Standard Code Table Algorithm

Input: A transaction database D over a set of items $\mathcal{I}$.
Output: The standard code table CT for D.

1. $CT \leftarrow \emptyset$
2. **for all** $X \in \mathcal{I}$ **do**
3. insert X into CT
4. **end for**
5. **for all** $X \in CT$ **do**
6. $usage_D(X) \leftarrow sup_D(X)$
7. $code_{CT}(X) \leftarrow$ optimal code for X
8. **end for**
9. **return** CT

Algorithm 2 The Standard Cover Algorithm

Input: Transaction $t \in D$ and code table CT, with CT and D over a set of items $\mathcal{I}$.
Output: A cover of t using non-overlapping elements of CT.

1. $S \leftarrow$ smallest element X of 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

Standard Cover Function

Since it is infeasible to examine all possible covers to find an optimal cover, Krimp employs a heuristic standard cover function given in 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. $t \setminus X$. This heuristic algorithm, viz. Standard Cover, drastically reduces the complexity. The order of the $X \in CT$ is first by cardinality, then by support in D, and then lexicographically. The first two are descending, the last is ascending. This order gives priority to long itemsets as these can replace as many items as possible by just one code. By $sup_D(X)$ we prefer those itemsets that occur frequently in database as their expected usage is higher. The lexicographic order only serves to break ties.

Algorithm 3 The Krimp Algorithm

Input: A transaction database D and a candidate set $\mathcal{F}$, both over a set of items $\mathcal{I}$.

Output: A heuristic solution to the optimal code table problem.

1. $CT \leftarrow$ **Standard Code Table**(D)
2. $\mathcal{F}_0 \leftarrow \mathcal{F}$ in **Standard Candidate Order**
3. **for all** $F \in \mathcal{F}_0 \setminus \mathcal{I}$ **do**
4. $\quad CT_C \leftarrow (CT \cup F)$ in **Standard Cover Order**
5. $\quad$ **if** $L(D, CT_C) < L(D, CT)$ **then**
6. $\quad\quad CT \leftarrow CT_C$
7. $\quad$ **end if**
8. **end for**
9. **return** CT

Standard Candidate Order

The order of candidate itemsets is similar to the one used by the cover function above:

$$sup_D(X) \downarrow |X| \downarrow lexicographically \uparrow .$$

Krimp

Krimp uses a greedy search strategy with these heuristics, see Algorithm 3. It starts with the standard code table ST and adds the candidate itemsets from $\mathcal{F}$ one by one. Each time, it chooses the itemset that is maximal with respect to the standard candidate order. It encodes the database with the resulting new code table. If the resulting encoding yields a smaller compressed size, the itemset is added to the code table. Otherwise, it is discarded permanently. Fig. 2.4 illustrates the algorithm.

Each iteration, of Krimp, can both lessen and heighten the usage of an itemset in CT. For example, if F_1 is already accepted in the code table, $F_1 \cap F_2 \neq \varnothing$ and F_2 is used before F_1 by the standard cover function, the usage of F_1 will go down if F_2 is added to the code table (given the support of F_2 does not equal zero). The reverse situation happens when we now add an itemset F_3 which is used before F_2 such that:

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

The usage of F_2 will decrease while the usage of F_1 will increase by the same amount.

Thus, one cannot remove code table elements with zero usage without consequences. For, if such an element is removed, it will never be considered again.

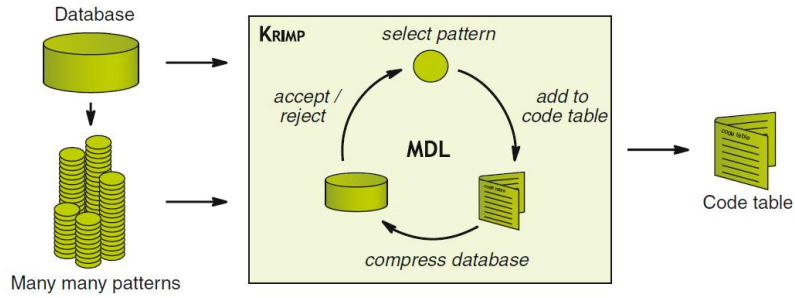

Figure 2.4: KRIMP in action

Algorithm 4 Code Table Post-Acceptance Pruning

Input: Codetables CT_c and CT, for a transaction database D over a set of items $\mathcal{I}$, where $\{X \in CT\} \subset \{Y \in CT_c\}$ and $L(D,CT_c) < L(D,CT)$.

Output: Pruned code table CT_p, such that $L(D,CT_p) \leq L(D,CT_c)$ and $CT_p \subseteq CT_c$.

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

Even if it would be useful in later encodings. Fortunately, the elements with zero usage 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 they can be safely removed. Since they do not code, it means they are not part of the optimal answer for smallest coding set. Singletons with zero usage have to remain in the code table, of course, they have to by definition.

Pruning

Not withstanding the previous discussion, itemsets with a very low usage count in the code table are not always desirable. Since they have a very small prob-

ability, their 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. Consider the following three code tables:

$$CT_1 = \{\{X_1,X_2\},\{X_2,X_3\}\}$$
$$CT_2 = \{\{X_1,X_2,X_3\},\{X_1,X_2\},\{X_1\},\{X_2\},\{X_3\}\}$$
$$CT_3 = \{\{X_1,X_2,X_3\},\{X_1\},\{X_2\},\{X_3\}\}$$

Assume that $sup_D(\{X_1,X_2,X_3\}) = sup_D(\{X_1,X_2\})$ - 1. Given this assumption, Krimp will never consider CT_3, but it is very well possible that $L(D,CT_3) < L(D,CT_2)$ and that CT_3 provides access to a branch of the search space that is otherwise left unvisited. To allow searching that branch, we can prune the code table that Krimp is currently considering.

For pruning, Krimp uses *post-acceptance pruning*. That is, Krimp only prunes when a candidate F is accepted. F is accepted when the candidate code table $CT_c = CT \cup F$ is better than CT, i.e. $L(D,CT_c) < L(D,CT)$. For pruning, we consider all its valid subsets. This effectively reduces pruning costs, as only few candidate itemsets will be accepted.

To reduce pruning costs further, Krimp does not consider all valid subsets of CT, but iteratively consider itemsets $X \in CT$ for which $usage_D(X)$ has decreased for removal. The rationale is that for these itemsets we know that their code lengths have increased and these now harm the compression.

First Krimp considers the itemset with the smallest usage and thus the longest code. If by pruning an itemset the total encoded length decreases, we permanently remove it from the code table. Further, we then update the list of prune candidates with those itemsets whose usage consequently decreased. This post-acceptance pruning is formalised in Algorithm 4.

Complexity

In the worst case, all candidate patterns are added to the code table. However, due to MDL, the number of elements in the code table is usually very small: $|CT| \ll |D| \ll |\mathcal{F}|$, in particular when pruning is enabled. Here $\mathcal{F}$ denotes the set of all candidate itemsets.

Using techniques such as hash tables, bitmaps, and more sophisticated cover algorithms, we can restrain the time complexity of Krimp with or without pruning to:

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

As for the memory requirements of the Krimp algorithm, its worst case estimate is:

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

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. Since it is iterated in order, it can be handled from the hard disk without much performance loss. Preferably the database is kept resident as it is covered many many times. Table 2.4 shows some results of running KRIMP with post-acceptance pruning.

Table 2.4: 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 length of the database with the obtained code table divided by the encoded length with the standard code table.

				KRIMP	
Dataset	$\|D\|$	$\|\mathcal{F}\|$	$\|\mathcal{I}\|$	$\|CT \backslash \mathcal{I}\|$	$\frac{L(D,CT)}{L(D,ST)}\%$
Adult	48842	58461763	468	1303	24.4
Chess (kr-k)	28056	373421	58	1684	61.6
Led7	3200	15250	24	152	28.6
Letter	20000	580968767	102	1780	35.7
Mushroom	8124	5574930437	119	442	20.6
Pen	10992	459191636	86	1247	42.3

As an aside, note that the name 'KRIMP' is Dutch for 'to shrink'. For more details on KRIMP see [64, 73]. For further details on its complexity [41, 70].

2.4 Code Tables and Relational Databases

Part of our research involves pattern mining in relational databases [12], hence, we briefly discuss this topic here.

The strength of relational databases is that tables can relate to each other via *associations*, either *one-to-one*, *one-to-many*, or *many-to-many*. An attribute that is used for relating tables is known as a *key*. For example, we have two tables PERSON and CHILD with keys *pid* and (*pid, cid*). The attribute *cid* is the primary key of CHILD, while *pid* is the primary key of PERSON. Moreover, *pid* also has the role of *foreign key*, i.e., it is the key that relates the two tables. By using this foreign key *pid* in CHILD one can determine the tuples in the PERSON that are related to a given tuple in the CHILD table.

By defining an item in a relational table as an attribute-value pair, frequent itemset mining is simply generalised to relational tables. To generalise this to patterns over multiple tables, there are various options. An important solution to mine patterns directly from relational databases comes from the field of Inductive Logic Programming (ILP). In ILP, patterns are given by first order logic clauses such as:

$$\textsc{Person}(pid = X) \leftarrow \textsc{Person}(pid = Y) \wedge \textsc{Child}(pid = X, cid = Y).$$

This example denotes a list of all children for each parent X.

WARMR, developed by Dehaspe and Toivonen [15], is a well-known ILP algorithm to find frequent queries within a relational database. Similar to *A Priori*, WARMR finds all frequent queries given a minimal support threshold. It starts with patterns of cardinality l, i.e., patterns denoting one attribute value pair in a single table. In subsequent iterations these patterns are extended as long as their supports exceeds the minimal support threshold.

WARMR only finds patterns with one child per parent node. FARMER [56] is not only more efficient than WARMR, it also allows patterns with multiple children per parent. SMuRFIG [21, 22], finally, mines for a special class of queries, simple conjunctive queries (select-project-join queries), very efficiently.

A second approach to mining queries on multiple tables is by *propositionalisation*. The information stored in related tables is transferred to the so-called target table using aggregation functions such as *count*, *max*, and *min* [35,36,39]. This is far more efficient than the ILP approaches, but it looses some fine grained information due to the aggregation.

Note that in both approaches, after the frequent patterns have been determined, one can again use Krimp to reduce the resulting set to something more manageable.

A third approach would be to join all the tables into one huge table and mine that table directly. In general, this leads to gigantic numbers of frequent patterns. If the goal is to reduce that set with a Krimp-like approach, Koopman and Siebes [38] have shown that one can achieve this far more efficiently with the R-Krimp algorithm. They first compute the code tables of each of the relations separately and then combine these into a code table for the whole database. This reduces the search space dramatically, while resulting in a code table that is approximately the same as one would get by starting with computing the join of all tables. Some of the work reported on in this thesis is related to this approach.

We conclude this section with some further examples of relational itemsets in Figure 2.5. The database consists of a collection of tables, one contains transactions of book sales, the other table contains record sales. Each transaction consists of the titles of the work that are sold during that transaction. Two

Id = {*Person Id*}
I_1 = {abbey road, thriller, queen}

T_1:Records	
Person Id	Tr_1
1	{abbey road}
1	{queen , thriller}
2	{abbey road , queen}

Id = {*Person Id*}
I_2 = {colour of magic, good omens}

T_2:Books	
Person Id	Tr_2
1	{colour of magic , good omens}
1	{colour of magic}
2	{good omens}

Join(T_1,T_2) - target:Records		
{{abbey road , queen , thriller}	,	{good omens}}
{{abbey road , queen}	,	{colour of magic}}
{{abbey road , queen}	,	{good omens}}
F_1 = {{abbey road, queen}, {Ø}}		supp(F_1) = 3

Join(T_1,T_2) - target:Books		
{{abbey road}	,	{colour of magic , good omens}}
{{queen , thriller}	,	{colour of magic , good omens}}
{{abbey road , queen}	,	{colour of magic , good omens}}
F_1 = {{Ø}, {colour of magic, good omens}}		supp(F_1) = 3

Join(T_1,T_2)		
{{abbey road}	,	{colour of magic , good omens}}
{{queen , thriller}	,	{colour of magic , good omens}}
{{abbey road , queen}	,	{colour of magic}}
{{abbey road , queen}	,	{good omens}}
F_1 = {{abbey road, queen}, {Ø}}		supp(F_1) = 2
F_1 = {{Ø}, {colour of magic, good omens}}		supp(F_2) = 2

Figure 2.5: Example of joined relational database with records and books taken from [37]. The join is performed via *Person Id*. Depending on the focus of interest their algorithm has frequent Record patterns, frequent Book patterns, patterns that describe the relation. Note that only the largest patterns are depicted for clarity.

transactions are related when the same customer has bought two product sets, one from each table. For example *Person 1* who bought books titled "Colour of Magic" and "Good Omens" has two related transactions in the *Records* table, viz, "Abbey Road" and "Queen, Thriller".

From the database one can generate different join results depending on what one is interested in. For example: "*What record titles are sold to a person together with a given frequent book title?*", or "*What book titles are sold to a person together with a given frequent record title?*". The figure shows the results together with a pattern found in each join-table. Another question such as "*What record titles or book titles are frequently sold to a person?*". Each such query leaks to a different join and to different relational patterns. More on this can be found in [37].

2.5 Code Table and Probabilities

A code table CT on a database D implicitly defines a probability distribution on the set of all possible tuples in the domain Dom of D. Let t be such an arbitrary tuple, then we can compress it with CT:

$$\begin{aligned} L(t|CT) &= \sum_{I \in cover(t)} L(I|CT) = \sum_{I \in cover(t)} -\log(P(I|D)) \\ &= -\log\left(\prod_{I \in cover(t)} P(I|D)\right) = -\log(P(t|D)) \stackrel{def}{=} -\log(P(t|CT)) \end{aligned}$$

The one but last equation rests on the Naive Bayes like assumption that the itemsets in a cover are independent. They are not(!), but in previous work this distribution has shown to characterise the data distribution on D very well. Since we use this distribution in this thesis as well, we illustrate with previous work on *imputation*, i.e., filling in missing data, here.

One of the first known methods for imputation, hot deck imputation, [4], was employed by the US census bureau in the 1950s. Hot deck imputation replaces missing records by random draws from complete records from the same local area. It may be viewed as a crude form of k nearest-neighbour imputation which also uses a distance function on the data [75].

Imputations using regression, mean substitution and mean-mode [26] are greedy, which harms the variance in the completed data [4]. A better approach, and state of the art, is using Expectation Maximisation EM [16]. EM maximises the likelihood of the data given a distribution. It adapts the model to the data iteratively and re-imputes it. This leads EM to provide very accurate probability estimations.

Algorithm 5 Simple Completion

$SC(D)$

1. $CT := \textsc{Krimp}(D_{comp})$
2. **for each** $t \in D_{inc}$ **do**
3. $\quad t := \text{argmin}_{s \in C(t)}\ L_{CT}(s)$
4. **end for**
5. **return** $D_{comp} \cup D_{inc}$

Vreeken and Siebes [71] proposed an idea of doing imputation with EM-like algorithm based on the distribution we introduced above. They iteratively optimise the compressed size of the database. In fact they offer 3 approaches: simple completion, Krimp completion, and Krimp Minimisation.

In simple completion, missing values are imputed by their maximal likelihood estimation. Let $t \in D$ be a transaction with missing values and denote by $C(t)$ the set of all its possible completions, the algorithm then replaces t by that element of $C(t)$ that has the shortest encoded length. This algorithm is given in Algorithm 5. It works as follows.

Let $D = D_{comp} \cup D_{inc}$ such that all transactions in D_{comp} are complete, while all transactions in D_{inc} are incomplete. First, Krimp is used to induce a code table CT from D_{comp}. Since it is possible that one or more of the singletons have zero usage, we apply a Laplace correction on the usages such that any tuple in the domain can be encoded [44]. Each incomplete transaction in D_{inc} is then replaced by the completed transaction with the shortest encoding.

Clearly, simple completion overestimates shorter codes and underestimates the longer ones. The Krimp Completion algorithm (Algorithm 6) remedies this by choosing an element of $C(t)$ with a chance proportional to its encoded length. Again, Krimp first computes the code table of D_{comp}. Then we define a probability distribution on the set of all possible completions $C(t)$ of a given incomplete tuple t by:

$$\frac{2^{-L_{CT}(s)}}{\sum_{u \in C(t)} 2^{-L_{CT}(u)}}$$

The completion of t is then chosen from $C(t)$ according to this distribution. The Krimp completion approach avoids the bias of the simple completion algorithm, but the quality of its result depends on the amount of complete data in the database. This final weakness is addressed using an EM-like algorithm [16].

Krimp Minimisation starts with a random completion of the incomplete database D. It then iterates through a number of Krimp and Krimp completion steps. In the Krimp step it compresses the current *complete* database. In the Krimp completion step it completes the *incomplete* database D using the

Algorithm 6 Krimp Completion

$KC(D)$

1. $CT :=$ Krimp(D_{comp})
2. **for each** $t \in D_{inc}$ **do**
3. $\quad t :=$ Choice$(C(t), CT)$
4. **end for**
5. **return** $D_{comp} \cup D_{inc}$

Algorithm 7 Krimp Minimisation

$KM(D)$

1. $D_c :=$ random completion of D
2. **while** not converged **do**
3. $\quad CT :=$ Krimp(D_c)
4. $\quad D_c :=$ KC(D)
5. **end while**
6. **return** D_c

code table computed in the previous Krimp step. This is continued as long as the total encoded length of the completed database shrinks. The algorithm returns the final completed database, see Algorithm 7. This rather simple algorithm performs on par – often better – than state-of-the-art algorithms such as Structural EM (SEM) [19]; see [71] for further details.

2.6 Conclusions

In this chapter we briefly explained aspects of pattern mining and the Minimum Description Length (MDL) principle applied to pattern mining. This chapter also introduced how to mine *interesting* patterns in a database by computing a *code table* (CT); both for itemset databases and for normal relational databases. Finally we briefly discussed the probability distribution induced by such code tables. Together these are the basic ingredients we use in this thesis to answer our research question:

> What does a model of the data tell you about the results of a query on the data?

Chapter 3

Patterns on Queries

Inductive databases, introduced in a seminal paper by Imielinski and Mannila [33], are databases in which models and patterns are first class citizens. One of the most important features of any database system is that it supports *queries*. For example, in relational databases one can construct new tables from the stored tables using relational algebra. For an inductive database, it is reasonable to assume that the stored tables have been modelled. The problem we study in this chapter is: do the models available on the stored tables help to model the table constructed by a query? To focus the discussion, we concentrate on one type of modelling, viz., computing frequent item sets. This chapter is published in [63] based on results reported in two earlier papers, viz., [61, 62]. The approaches advocated in those papers are unified and compared.

3.1 Introduction

By far the most successful type of DBMS is relational. In a relational database, the data is stored in tables and a query constructs a new table from these stored tables using, e.g., the relational algebra [12]. While querying an inductive relational database, the user will, in general, not only be interested in the table that the query yields, but also – if not more – in a particular model induced from that result-table. Since inductive databases have models as first-class citizens – meaning they can be stored and queried – it is reasonable to assume that the original, stored, tables are already modelled. Hence, a natural question is: does knowing a model on the original tables help in inducing a model on the result of a query?

Slightly more formal, let H_D be the model we induced from database D and let Q be a query on D. Does knowing H_D help in inducing a model H_Q on $Q(D)$, i.e., on the result of Q when applied to D. For example, if H_D is a

classifier and Q selects a subset of D, does knowing H_D help the induction of a new classifier H_Q on the subset $Q(D)$?

This formulation is only slightly more formal as the term "help" is a non-technical and, thus, ill-defined concept. In this chapter we will formalise "help" in two different ways. Firstly, in the sense that we can compute H_Q directly from H_D *without* consulting either D or $Q(D)$. While this is clearly the most elegant way to formalise "help", it puts such stringent requirements on the class of models we consider that the answer to our question becomes *no* for many interesting model-classes; we'll exhibit one in this chapter.

Hence, secondly, we interpret "help", far less ambitiously, as meaning "speeding-up" the computation of H_Q. That is, let $\mathcal{A}lg$ be the algorithm used to induce H_D from D, i.e., $\mathcal{A}lg(D) = H_D$. We want to transform $\mathcal{A}lg$ into an algorithm $\mathcal{A}lg^*$, which takes H_D as extra input such that

$$\mathcal{A}lg^*(Q(D), H_D) \approx \mathcal{A}lg(Q(D))$$

Note that we do not ask for exactly the same model, approximately the same answer is acceptable if the speed-up is considerable. In fact, for many application areas, such as marketing, a *good enough model* rather than the *best model* is all that is required.

The problem as stated is not only relevant in the context of inductive databases, but also in existing data mining practice. In the data mining literature, the usual assumption is that we are given some database that has to be mined. In practice, however, this assumption is usually not met. Rather, the construction of the mining database is often one of the hardest parts of the KDD process [18]. The data often resides in a data warehouse or in multiple databases, and the mining database is constructed from these underlying databases.

From most perspectives, it is not very interesting to know whether one mines a specially constructed database or an original database. For example, if the goal is to build the best possible classifier on that dataset, the origins of the database are of no importance whatsoever.

It makes a difference, however, if the underlying databases have already been modelled. Then, like with inductive databases, one would hope that knowing such models would help in modelling the specially constructed 'mining database'. For example, if we have constructed a classifier on a database of customers, one would hope that this would help in developing a classifier for the female customers only.

In other words, the problem occurs both in the context of inductive databases and in the everyday practice of data miners. Hence, it is a relevant problem, but isn't it trivial? After all, if H_D is a good model on D, it is almost always also a good model on a random subset of D; almost always, because a random

subset may be highly atypical. The problem is, however, *not* trivial because queries in general do *not* compute a random subset. Rather, queries construct a very specific result.

For the usual "project-select-join" queries, there is not even a natural way in which the query-result can be seen as subset of the original database. Even if Q is just a "select"-query, the result is usually not random and H_D can even be highly misleading on $Q(D)$. This is nicely illustrated by the well-known example of *Simpson's Paradox*, viz., Berkeley's admission data [8]. Overall, 44% of the male applicants were admitted, while only 35% of the females were admitted. Four of the six departments, however, have a bias that is in favour of female applicants. While the overall model may be adequate for certain purposes, it is woefully inadequate for a query that selects a single department.

In other words, we do address a relevant and non-trivial problem. Addressing the problem, in either sense of "help", for all possible model classes and/or algorithms is, unfortunately, too daunting a task for this chapter. In the sense of "direct construction" it would require a discussion of all possible model classes, which is too large a set to consider (and would result in a rather boring discussion). In the "speed-up and approximation" sense it would require either a transformation of all possible induction algorithms or a generic transformation that would transform any such algorithm to one with the required properties. The former would, again, be far too long, while a generic transformation is unlikely to exist.

Therefore we restrict ourselves to one type of model, viz., frequent itemsets [2] and one induction algorithm, viz., our own Krimp algorithm [64]. The structure of this chapter is as follows. In Section 3.2 we investigate the "direct computation" interpretation of "help" in the context of frequent itemset mining. This is followed in Section 3.3 by the introduction of a transformed variant of Krimp for the "speed-up" interpretation of "help". In Section 3.4 we discuss and compare these two approaches. The chapter ends with conclusions and prospects for further research.

3.2 Frequent Itemset Mining

The goal of this section is to investigate whether we can determine the set of frequent itemsets on $Q(D)$ without consulting $Q(D)$. In other words, we are given the frequent itemsets on D and a query Q and from that information only we want to determine the frequent itemsets on $Q(D)$. That is, we want to *lift* the relational operators to sets of frequent itemsets.

Selection

The relational algebra operator σ (select) is a mapping:

$$\sigma : \mathcal{B}(D) \rightarrow \mathcal{B}(D)$$

in which $\mathcal{B}(D)$ denotes all possible bags over domain D.

Lifting means that we are looking for an operator $\sigma_{(D,\mathcal{A}lg)}$ that makes the diagram in Figure 3.1 commute.

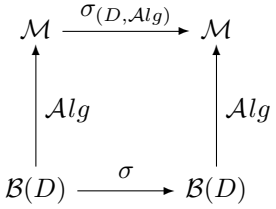

Figure 3.1: Lifting the selection operator

Such diagrams are well-known in, e.g., category theory [5] and the standard interpretation is:

$$\mathcal{A}lg \circ \sigma = \sigma_{(D,\mathcal{A}lg)} \circ \mathcal{A}lg$$

In other words, first inducing the model using algorithm $\mathcal{A}lg$ followed by the application of the *lifted* selection operator $\sigma_{(D,\mathcal{A}lg)}$ yields the same result as first applying the *standard* selection operator σ followed by induction with algorithm $\mathcal{A}lg$.

In fact, we are willing to settle for commutation of the diagram in a loose sense of the way. That is, if we are able to give reasonable support bounds for those itemsets whose support we can not determine exactly, we are satisfied.

For frequent itemsets the three basic selections are $\sigma_{I=0}$, $\sigma_{I=1}$, and $\sigma_{I_1=I_2}$. More complicated selections can be made by conjunctions of these basic comparisons. We look at the different basic selections in turn.

First consider $\sigma_{I=0}$. If it is applied to a table, all transactions in which I occurs are removed from that table. Hence, all itemsets that contain I get a support of zero in the resulting table. For those itemsets in which I doesn't occur, we have to compute which part of their support consists of transactions in which I does occur and subtract that number. Hence, for support for itemsets J, we have:

$$sup_{\sigma_{I=0}(T)}(J) = \begin{cases} 0 & \text{if } I \in J, \\ sup_T(J) - sup_T(J \cup \{I\}) & \text{else.} \end{cases}$$

If we apply $\sigma_{I=1}$ to the table, all transactions in which I doesn't occur are removed from the table. In other words, the support of itemsets that contain I doesn't change. For those itemsets that do not contain I, the support is given by those transactions that *also* contained I. Hence, we have:

$$sup_{\sigma_{I=1}(T)}(J) = \begin{cases} sup_T(J) & \text{if } I \in J, \\ sup_T(J \cup \{I\}) & \text{else.} \end{cases}$$

If we apply $\sigma_{I_1=I_2}$ to the table, the only transactions that remain are those that either contain both I_1 and I_2 or neither. In other words, for frequent itemsets that contain both the support remains the same. For all others, the support changes. For those itemsets J that contain just one of the I_i the support will be the support of $J \cup \{I_1, I_2\}$. For those that contain neither of the I_i, we have to correct for those transactions that contain one of the I_i in their support. If we denote this by $sup_T(J \neg I_1 \neg I_2)$ (a support that can be easily computed) We have:

$$sup_{\sigma_{I_1=I_2}(T)}(J) = \begin{cases} sup_T(J \cup \{I_1, I_2\}) & \text{if } \{I_1, I_2\} \cap J \neq \emptyset, \\ sup_T(J \neg I_1 \neg I_2) & \text{else.} \end{cases}$$

Clearly, we can also "lift" conjunctions of the basic selections, simply process one at the time. So, in principle, we can lift all selections for frequent itemsets. But only in principle, because we need the support of itemsets that are *not necessarily frequent.* Frequent itemsets are a lossy model (not all aspects of the data distribution are modelled) and that can have its repercussions: in general the lifting will *not* be commutative. In our loose sense of "commutativity", the situation is slightly better. For, we can give reasonable bounds for the resulting supports; for those supports we do not know are bounded (from above) by *min-sup.*

We haven't mentioned constraints [55] so far. Constraints in frequent item set mining are the pre-dominant way to select a subset of the frequent itemsets. In general the constraints studied do not correspond to selections on the database. The exception is the class of *succinct anti-monotone constraints* introduced in [55]. For these constraints there is such a selection (that is what succinct means) and the constraint can be pushed into the algorithm. This means we get the commutative diagram in Figure 3.2.

Note that in this case we know that the diagonal arrow makes the bottom right triangle commute in the strict sense of the word. For the upper left triangle, as well as the square, our previous analysis remains true.

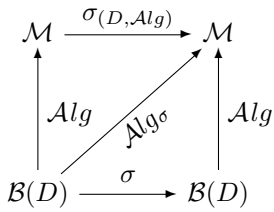

Figure 3.2: Lifting selections for succinct constraints

Projection

For the projection operator π we have a new domain D_1 such that $D = D_1 \times D_2$. Projection on D_1 has thus as signature:

$$\pi_{D_1} : \mathcal{B}(D) \rightarrow \mathcal{B}(D_1)$$

Hence, we try to find an operator $\pi_{D_1}^{\mathcal{A}lg}$ that makes the diagram in Figure 3.3 commute. Note that D_1 is spanned by the set of variables (or items) we project

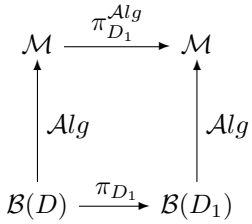

Figure 3.3: Lifting projections

on.

We project on a set of items $\mathcal{J} \subseteq \mathcal{I}$, let $J \subseteq \mathcal{I}$ be a frequent itemset. There are three cases to consider:

1. if $J \subseteq \mathcal{J}$, then all transactions in the support of J will simply remain in the table, hence J will remain frequent.
2. if $J \cap \mathcal{J} \neq \emptyset$, then $J \cap \mathcal{J}$ is also frequent and will remain in the set of frequent itemsets.
3. if $J \cap \mathcal{J} = \emptyset$, then its support will vanish.

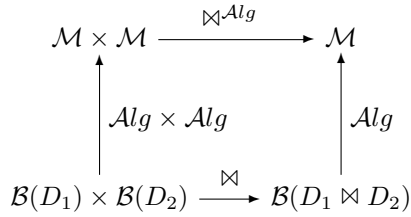

Figure 3.4: Lifting the equijoin

In other words, if $\mathcal{F}$ denotes the set of all frequent itemsets, then:

$$\pi_{\mathcal{J}}(\mathcal{F}) = \{J \in \mathcal{F} | J \subseteq \mathcal{J}\}$$

Clearly, this method of lifting will make the diagram commute in the strict sense if we use an absolute minimal frequency. In other words, for projections, frequent itemsets do capture enough of the underlying data distribution to allow lifting.

EquiJoin

The equijoin has as signature:

$$\bowtie: \mathcal{B}(D_1) \times \mathcal{B}(D_2) \to \mathcal{B}(D_1 \bowtie D_2)$$

Hence, the diagram we want to make commute is given in Figure 3.4. The join can be computed, though not very efficiently, starting with the Cartesian product of the two tables. Moreover in extreme cases the equijoin equals the Cartesian product, hence we discuss that operator first.

Let J_1 be a frequent itemset for the first table and J_2 for the second. The frequency of the pair on the Cartesian product of the two tables is simply given by:

$$sup_{T_1 \times T_2}(J_1, J_2) = sup_{T_1}(J_1) \times sup_{T_2}(J_2)$$

While this is easy to compute, it means again that in general we will not be able to compute all frequent itemsets on the Cartesian product without consulting the database. Even if we set the minimal frequency to the product of the two minimal frequencies, the combination of an infrequent itemset on one database with a frequent one on the other may turn out to be frequent.

In other words, we cannot even make the diagram commute in the approximate sense of the word. For, the bound is given by $\max\{|T_1| \times (minsup - 1), |T_2| \times (minsup - 1)\}$, which is hardly a reasonable bound.

Given that the number of joins possible in a database is limited and known beforehand, we may make our lives slightly easier. That is, we may allow ourselves to do some pre-computations.

Assume that we compute the tables $T_1^2 = \pi_{T_1}(T_1 \bowtie T_2)$ and $T_2^1 = \pi_{T_2}(T_1 \bowtie T_2)$ and their frequent itemsets, say $\mathcal{F}_1^2$ and $\mathcal{F}_2^1$, off-line. Are those sets enough to lift the join? For the extreme case, the Cartesian product, the answer is clearly: *yes*. By "blowing" up the original tables we add enough information to compute the support of any itemset in the join *iff* that itemset exceeds the minimal support.

Unfortunately, the same is not true for the join in general. Since we cannot see from either $\mathcal{F}_1^2$ or $\mathcal{F}_2^1$ which combinations of frequent itemsets will actually occur in $(T_1 \bowtie T_2)$. That is, we can only compute a superset of the frequent itemsets on the join.

Hence, the only way to lift the join is to compute and store the frequent itemsets on all possible joins. While this is doable given the limited number of possible joins, this can hardly count as lifting.

Discussion

The fact that lifting the relational algebra operators to sets of frequent itemsets is only partially possible should hardly come as a surprise: the *min-sup* constraint makes this into an inherently lossy model. For models that do try to capture the complete distribution, such as Baysian networks, one would expect far better results; see [61] for a discussion of lifting for such networks.

3.3 Transforming Krimp

Recall from the Introduction that the problem we investigate in this section is that we want to transform an induction algorithm $\mathcal{A}lg$ into an algorithm $\mathcal{A}lg^*$ that takes at least two inputs, i.e, both Q and $\mathcal{M}_D$, such that:

1. $\mathcal{A}lg^*$ gives a reasonable approximation of $\mathcal{A}lg$ when applied to Q, i.e.,

$$\mathcal{A}lg^*(Q, \mathcal{M}_D) \approx \mathcal{M}_Q$$

2. $\mathcal{A}lg^*(Q, \mathcal{M}_D)$ is simpler to compute than $\mathcal{M}_Q$.

The second criterion is easy to formalise: the runtime of $\mathcal{A}lg^*$ should be shorter than that of $\mathcal{A}lg$. The first one is harder. What do we mean that one model is an approximation of another? Moreover, what does it mean that it is a *reasonable* approximation?

Before we discuss how KRIMP can be transformed and provide experimental evidence that our approach works, we first formalise this notion of approximation.

Model Approximation

The answer to the question how to formalise that one model approximates another depends very much on the goal. If $\mathcal{A}lg$ induces classifiers, approximation should probably be defined in terms of prediction accuracy, e.g., on the Area Under the ROC-curve (AUC).

KRIMP computes code tables. Hence, the quick approximating algorithm we are looking for, KRIMP* in the notation used above, also has to compute code tables. So, one way to define the notion of approximation is by comparing the resulting code tables. Let CT_{KRIMP} be the code table computed by KRIMP and, similarly, let CT_{KRIMP^*} denote the code table computed by KRIMP* on the same data set. The more similar CT_{KRIMP^*} is to CT_{KRIMP}, the better KRIMP* approximates KRIMP.

While this is intuitively a good way to proceed, it is far from obvious how to compare two code tables. Fortunately, we do not need to compare code tables directly. KRIMP is based on MDL and MDL offers another way to compare models, viz., by their *compression rates*. Note that using MDL to define "approximation" has the advantage that we can formalise our problem for a larger class of algorithms than just KRIMP. It is formalised for all algorithms that are based on MDL. MDL is quickly becoming a popular formalism in data mining research, see, e.g., [17] for an overview of other applications of MDL in data mining.

What we are interested in is comparing two algorithms on the same dataset, viz., on $Q(D)$. Slightly abusing notation, we will write $\mathcal{L}(\mathcal{A}lg(Q))$ for $[L(\mathcal{A}lg(Q)) + L(Q(D)|\mathcal{A}lg(Q))]$, similarly, we will write $\mathcal{L}(\mathcal{A}lg^*(Q, H_D))$. Then, we are interested in comparing $\mathcal{L}(\mathcal{A}lg^*(Q, H_D))$ to $\mathcal{L}(\mathcal{A}lg(Q))$. The closer the former is to the latter, the better the approximation is.

Just taking the difference of the two, however, can be quite misleading. Take, e.g., two databases D_1 and D_2 sampled from the same underlying distribution, such that D_1 is far bigger than D_2. Moreover, fix a model H. Then necessarily $L(D_1|H)$ is bigger than $L(D_2|H)$. In other words, big absolute numbers do not necessarily mean very much. We have to *normalise* the difference to get a feeling for how good the approximation is. Therefore we define the asymmetric dissimilarity measure (ADM) as follows [72].

Definition 1 Let H_1 and H_2 be two models for a dataset D. The asym-

metric dissimilarity measure $ADM(H_1, H_2)$ is defined by:

$$ADM(H_1, H_2) = \frac{|\mathcal{L}(H_1) - \mathcal{L}(H_2)|}{\mathcal{L}(H_2)}$$

Note that this dissimilarity measure is related to the Normalised Compression Distance [11]. The reason why we use this asymmetric version is that we have a "gold standard". We want to know how far our approximate result $\mathcal{A}lg^*(Q, H_D)$ deviates from the optimal result $\mathcal{A}lg(Q)$.

The remaining question is, of course, what ADM scores indicate a good approximation? In a previous paper [72], we took two random samples from datasets, say D_1 and D_2. Code tables CT_1 and CT_2 were induced from D_1 and D_2 respectively. Next we tested how well CT_i compressed D_j. For the four datasets also used in this paper, *Iris, Led7, Pima* and, *PageBlocks*, the "other" code table compressed 16% to 18% worse than the "own" code table; the figures for other datasets are in the same ball-park. In other words, an ADM score of 0.2 is in line with the "natural variation" in a dataset. If it gets much higher, it shows that the two code tables are rather different.

Clearly, $ADM(\mathcal{A}lg^*(Q, H_D), \mathcal{A}lg(Q))$ does not only depend on $\mathcal{A}lg^*$ and on $\mathcal{A}lg$, but also very much on Q. We do not seek a low ADM on one particular Q, rather we want to have a reasonable approximation on all possible queries. Requiring that the ADM is equally small on all possible queries seems too strong a requirement. Some queries might result in a very atypical subset of D, the ADM is probably higher on the result of such queries than it is on queries that result in more typical subsets. Hence, it is more reasonable to require that the ADM is small most of the time. This is formalised through the notion of an (ϵ, δ)-approximation.

Definition 2 Let D be a database and let Q be a random query on D. Moreover, let $\mathcal{A}lg_1$ and $\mathcal{A}lg_2$ be two data mining algorithms on D. Let $\epsilon \in \mathbb{R}$ be the threshold for the maximal acceptable ADM score and $\delta \in \mathbb{R}$ be the error tolerance for this maximum. $\mathcal{A}lg_1$ is an (ϵ, δ)-approximation of $\mathcal{A}lg_2$ iff

$$P(ADM(\mathcal{A}lg_1(Q), \mathcal{A}lg_2(Q)) > \epsilon) < \delta$$

Transforming Krimp

Given that Krimp results in a code table, there is only one sensible way in which Krimp(D) can be re-used to compute Krimp(Q): provide Krimp only with the itemsets in CT_D as candidates. While we change nothing to the algorithm, we'll use the notation Krimp* to indicate that Krimp got only code table elements as candidates. So, Krimp$^*(Q)$ is the code table that Krimp induces from $Q(D)$ using the itemsets in CT_D only.

Given our general problem statement, we now have to show that KRIMP* satisfies our two requirements for a transformed algorithm. That is, we have to show for a random database D:

- For reasonable values for ϵ and δ, KRIMP* is an (ϵ, δ)-approximation of KRIMP, i.e, for a random query Q on D:

$$P(ADM(\text{KRIMP}^*(Q), \text{KRIMP}(Q)) > \epsilon) < \delta$$

  Or in MDL-terminology:

$$P\left(\frac{|\mathcal{L}(\text{KRIMP}^*(Q)) - \mathcal{L}(\text{KRIMP}(Q))|}{\mathcal{L}(\text{KRIMP}(Q))} > \epsilon\right) < \delta$$

- Moreover, we have to show that it is faster to compute KRIMP*(Q) than it is to compute KRIMP(Q).

The first property cannot be formally proven, if only because KRIMP and thus KRIMP* are both heuristic algorithms. Rather, we report on extensive tests of this requirement. The second property is obviously true, since we have less candidates.

The Experiments

In this subsection we describe our experimental set-up. First we briefly describe the datasets we use. Next we discuss the queries used for testing. Finally we describe how the tests are performed.

The Datasets

To test our hypothesis that KRIMP* is a good and fast approximation of KRIMP, we have performed extensive tests mostly on 6 well-known UCI [13] datasets and one dataset from the KDDcup 2004.

More in particular, we have used the datasets *connect, adult, Chess (kr vs k), letRecog, PenDigits* and *mushroom* from UCI. These datasets were chosen because they are well suited for KRIMP. Some of the other datasets in the UCI repository are simply too small for KRIMP to perform well. MDL needs a reasonable amount of data to be able to function. Some other datasets are very dense. While KRIMP performs well on these very dense datasets, choosing them would have turned our extensive testing prohibitively time-consuming.

Since all these datasets are single table datasets, they do not allow testing with queries involving joins. To test such queries, we used tables from

the "Hepatitis Medical Analysis"[1] of the KDDcup 2004. From this relational database we selected the tables *bio* and *hemat*. The former contains biopsy results, while the latter contains results on hematological analysis. The original tables have been converted to itemset data and rows with missing data have been removed.

The Queries

To test our hypothesis, we need to consider randomly generated queries. On first sight this appears a daunting task. Firstly, because the set of all possible queries is very large. How do we determine a representative set of queries? Secondly, many of the generated queries will have no or very few results. If the query has no results, the hypothesis is vacuously true. If the result is very small, MDL (and thus Krimp) doesn't perform very well.

To overcome these problems, we restrict ourselves to queries that are build using selections (σ), projections (π), and joins ($\bowtie$) only. The rationale for this choice is twofold. Firstly, simple queries will have, in general, larger results than more complex queries. Secondly, we have seen in Section 3.2 that lifting these operators is already a problem.

The Experiments

In all the experiments, we used Krimp and Krimp* in combination with post-acceptance pruning. For Krimp, we used closed frequent itemset as candidates, with the *minsup* tuned per dataset, as low as possible.

The experiments performed for each of the queries on each of the datasets were generated as follows.

Projection: The projection queries were generated by randomly choosing a set X of n items, for $n \in \{1, 3, 5, 7, 9\}$. The generated query is then $\pi_{\overline{X}}$. That is, the elements of X are projected out of each of the transactions. For example, $\pi_{\overline{\{I_1, I_3\}}}(\{I_1, I_2, I_3\}) = \{I_2\}$. For this case, the code table elements generated on the complete dataset were projected in the same way. For each value of n, 10 random sets X were generated on each dataset.

As an aside, note that the rationale for limiting X to maximally 9 elements is that for larger values too many result sets became too small for meaningful results.

Selection: The random selection queries were again generated by randomly choosing a set X of n items, with $n \in \{1, 2, 3, 4\}$. Next for each random

[1] http://lisp.vse.cz/challenge/

item I_i a random value v_i (uniform from {0,1}) in its domain D_i was chosen. Finally, for each I_i in X a random $\theta_i \in \{=, \neq\}$ was chosen (note that this does not influence the results in our case, since we only consider binary domains). The generated query is thus $\sigma(\bigwedge_{I_i \in X} I_i \theta_i v_i)$. As in the previous case, we performed 10 random experiments on each of the datasets for each of the values of n.

Project-Select: The random project-select queries generated, are essentially combinations of the simple projection and selection queries as explained above. The only difference is that we used $n \in \{1, 3\}$ for the projection and $n \in \{1, 2\}$ for the selections. That is we select on 1 or 2 items and we project away either 1 or 3 items. The size of the results is, of course, again the rationale for this choice. For each of the four combinations, we performed 100 random experiments on each of the datasets: first we chose randomly the selection (10 times for each selection), for each such selection we performed 10 random projections.

Project-Select-Join: Since we only use one "multi-relational" dataset and there is only one possible way to join the *bio* and *hemat* tables, we could not do random tests for the join operator. However, in combination with projections and selections, we can perform random tests. These tests consist of randomly generated project-select queries on the join of *bio* and *hemat*. In this two-table case, KRIMP* got as input all pairs $(\mathcal{I}_1, \mathcal{I}_2)$ in which $\mathcal{I}_1$ is an itemset in the code table of the "blown-up" version of *bio*, and $\mathcal{I}_2$ is an itemset in the code table of the "blown-up" version of *hemat*. Again we select on 1 or 2 items and we project away either 1 or 3 items. And, again, we performed again 100 random experiments on the database for each of the four combinations; as above.

The Results

In this subsection we give an overview of the results of the experiments described in the previous section. Each test query is briefly discussed in its own subsubsection.

Projection Queries

In Figure 3.5 the results of the random projection queries on the *letRecog* dataset are visualised. The marks in the picture denote the averages over the 10 experiments, while the error bars denote the standard deviation. Note that, while not statistically significant, the average ADM grows with the number of attributes projected away. This makes sense, since the more attributes are projected away, the smaller the result set becomes.

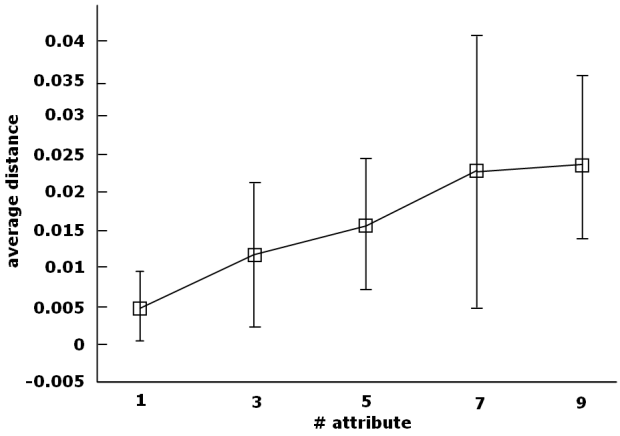

Figure 3.5: Projection results on *letRecog*. The x-axis depicts the number of attributes projected out by each random query. The y-axis depicts the ADM between the code tables obtained with Krimp and Krimp*.

On the other datasets, Krimp* performs similarly. Since this is also clear from the project-select query results, we do not provide all details here.

Selection Queries

The results of the random selection queries on the *penDigits* dataset are visualised in Figure 3.6. For the same reason as above, it makes sense that the average ADM grows with the number of attributes selected on. Note, however, that the ADM averages for selection queries seem much larger than those for projection queries.

Although these ADM averages are still reasonable, it turned out that *penDigits* is actually too small and sparse to test Krimp* on combined queries. In the remainder of our results section, we therefore do not report further results on *penDigits*. The reason why we report on it here is to illustrate that even on rather small and sparse datasets Krimp* still performs reasonably well with simple queries.

Project-Select Queries

The results of the project-select queries are given in Table 3.1. All numbers are the average ADM score ± the standard deviation for the 100 random experiments. All the ADM numbers are rather small, only for mushroom do they get above 0.2.

Table 3.1: The results of Project-Select Queries

ADM ± STD		connect	adult	Chess	letRecog	mushroom
Select 1	**Project out 1**	0.1 ± 0.01	0.1 ± 0.01	0.04 ± 0.01	0.1 ± 0.01	0.3 ± 0.02
	Project out 3	0.1 ± 0.02	0.1 ± 0.01	0.04 ± 0.03	0.1 ± 0.01	0.3 ± 0.16
Select 2	**Project out 1**	0.2 ± 0.01	0.1 ± 0.01	0.1 ± 0.03	0.04 ± 0.01	0.2 ± 0.04
	Project out 3	0.2 ± 0.02	0.1 ± 0.01	0.1 ± 0.03	0.04 ± 0.01	0.2 ± 0.05

Two important observations can be made from this table. Firstly, as for the projection and selection queries reported on above, the ADM scores get only slightly worse when the query results get smaller: "Select 2, Project out 3" has slightly worse ADM scores than "Select 1, Project out 1". Secondly, even more importantly, combining algebra operators only degrades the ADM scores slightly. This can be seen if we compare the results for "Project out 3" on *letRecog* in Figure 3.5 with the "Select 1, Project out 3" and "Select 2, Project out 3" queries in Table 3.1 on the same dataset. These results are very comparable, the combination effect is small and mostly due to the smaller result sets. While not shown here, the same observation holds for the other datasets.

To give insight in the distribution of the ADM scores, we present those obtained with the "Select 2, Project out 3" queries on the *connect* dataset are given in Figure 3.7. From this figure we see that KRIMP* is an (ϵ, δ)-approximation if we choose $\epsilon = 0.2$, $\delta = 0.08$. In other words, KRIMP* is a

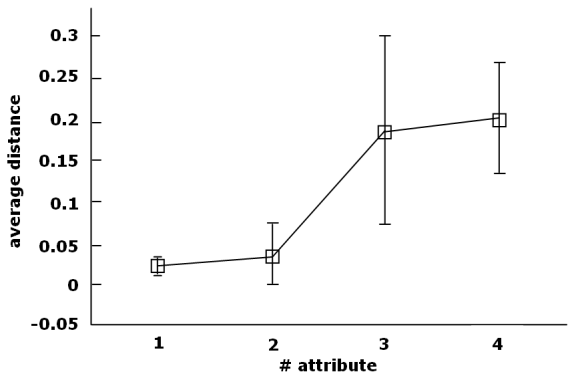

Figure 3.6: Selection results on *penDigits*.

pretty good approximation of Krimp. Almost always the approximation is less than 20% worse than the optimal result. The remaining question is, of course, how much faster is Krimp*? This is illustrated in Table 3.2.

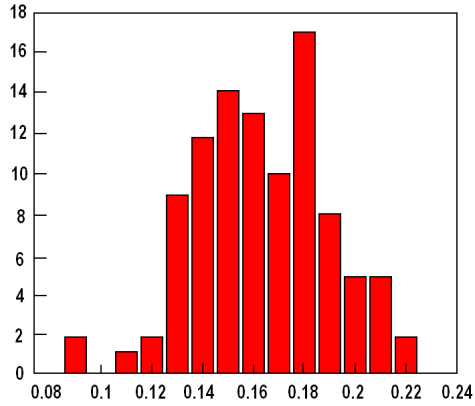

Figure 3.7: Histogram of 100 Project-Select Queries on *connect*

Table 3.2: Relative number of candidates for Krimp*

Relative #candidates		connect	adult	Chess	letRecog	mushroom
Select 1	**Project out 1**	0.01 ± 0.001	0.01 ± 0.002	0.21 ± 0.012	0.01 ± 0.001	0.01 ± 0.001
	Project out 3	0.01 ± 0.001	0.01 ± 0.004	0.26 ± 0.031	0.02 ± 0.004	0.01 ± 0.001
Select 2	**Project out 1**	0.01 ± 0.001	0.03 ± 0.003	0.76 ± 0.056	0.02 ± 0.002	0.03 ± 0.002
	Project out 3	0.01 ± 0.002	0.03 ± 0.008	0.96 ± 0.125	0.02 ± 0.004	0.03 ± 0.003

Table 3.2 gives the average number of candidates Krimp* has to consider relative to those that the full Krimp run has to consider. Since, both Krimp* and Krimp are linear in the number of candidates, this table shows that the speed-up is considerable; a factor of 100 is often attained; except for *Chess* where the query results get small and, thus, have few frequent itemsets. The experiments are those that are reported on in Table 3.1.

Project-Select-Join Queries

The results for the project-select-join queries are very much in line with the results reported on above. In fact, they are even better. Since the join leads

to rather large results, the ADM score is almost always zero: in only 15 of the 400 experiments the score is non-zero (average of non-zero values is 1%). The speed-up is also in line with the numbers reported above, a factor of 100 is again often attained.

Discussion

As noted in the previous section, the speed-up of KRIMP* is easily seen. The number of candidates that KRIMP* has to consider is often a factor 100 smaller than those that the full KRIMP run has to consider. Given that the algorithm is linear in the number of candidates, this means a speed-up by a factor 100. In fact, one should also note that for KRIMP*, we do not have to run a frequent itemset miner. In other words, in practice, using KRIMP* is even faster than suggested by the speed-up scores.

But, how about the other goal: how good is the approximation? That is, how should one interpret ADM scores? Except for some outliers, ADM scores are equal to or below 0.2. That is, a full-fledged KRIMP run compresses the dataset 20% better than KRIMP*. As noted when we introduced the ADM score, this about as good as one can expect, since such a percentage shows the natural variation in the data. Hence, given that the average ADM scores are often lower we conclude that the approximation by KRIMP* is good.

In other words, the experiments verify our hypothesis: KRIMP* gives a fast and good approximation of KRIMP. The experiments show this for simple "project-select-join" queries, but as noticed with the results of the "project-select" queries, the effect of combining algebra operators is small. If the result set is large enough, the approximation is good.

3.4 Comparing the two Approaches

In this chapter we introduced two ways in which the models present in an inductive database D help in computing the models on the results of a query Q on the data in that database. The first, if applicable, gives results without consulting $Q(D)$. The result is computed directly from the models H_T induced on the tables used by Q. For the relational algebra we formalised this by lifting the relational algebra operators to the set of all models.

The second approach does allow access to $Q(D)$. The induction algorithm $\mathcal{A}lg$ is transformed into an algorithm $\mathcal{A}lg^*$ that takes at least two inputs, i.e, both Q and $\mathcal{M}_D$, such that:

1. $\mathcal{A}lg^*$ gives a reasonable approximation of $\mathcal{A}lg$ when applied to Q, i.e.,

$$\mathcal{A}lg^*(Q, \mathcal{M}_D) \approx \mathcal{M}_Q$$

2. $\mathcal{A}lg^*(Q, \mathcal{M}_D)$ is simpler to compute than $\mathcal{M}_Q$.

The first requirement was formalised using MDL into the requirement:

$$P\left(\frac{|\mathcal{L}(\mathcal{A}lg^*(Q)) - \mathcal{L}(\mathcal{A}lg(Q))|}{\mathcal{L}(\mathcal{A}lg(Q))} > \epsilon\right) < \delta$$

for reasonably small ϵ and δ. The second requirement was simply interpreted as a significant speed-up in computation.

Clearly, when applicable, the first approach is to be preferred above the second approach. Firstly because it doesn't even require the computation of $Q(D)$, and is, hence, likely to be much faster. Secondly, because an algebraic structure on the set of all models opens up many more possible applications.

In this chapter, we investigated both approaches on itemsets. More precisely, we investigated lifting the relational algebra operators to sets of frequent itemsets. Moreover, we transformed our Krimp algorithm to investigate the second approach.

As noted already in Section 3.2, lifting the relational algebra operators to sets of frequent itemsets has its problems. Only for the projection it works well. For the selection operator we get a reasonable approximation. Reasonable in the sense that we can put a bound on the error of the approximated support; an upperbound that is determined by the minimal support threshold. Since this bound is an upperbound, this means that we may declare too many itemsets to be frequent. If we declare an itemset to be infrequent, it is infrequent on the result of the selection.

The join operator, unfortunately, can not be lifted at all. Not even if we provide extra information by giving access to the frequent item sets on the "blown-up" version of the underlying tables. In that case, we again only have an upperbound on the support. That is, again, we declare too many itemsets to be frequent. In the case of the join, however, there is no bound on the error. For, if I_1 has a high support on $T_1^2 = \pi_{T_1}(T_1 \bowtie T_2)$, say n_1, while I_2 has a high support on $T_2^1 = \pi_{T_2}(T_1 \bowtie T_2)$, say n_2, then the computed upperbound on the support of (I_1, I_2) on $T_1 \bowtie T_2$ will be $n_1 \times n_2$, while there may be no transaction in $T_1 \bowtie T_2$ which actually supports this pair! Again, if we declare an itemset to be infrequent on the join, it is infrequent.

As noted before, the reason for this failure is that sets of frequent itemsets are an inherently lossy model. As our analysis above shows, this loss of information makes us overestimate the support of itemsets on $Q(D)$, in the case of the join with an unbounded error.

The transformation of Krimp proved to be far more successful. The algorithm Krimp*, which is simply Krimp with a restricted set of candidates, proved in the experiments to be much faster and provide models which approx-

imate the true model very well. Given the lack of success for frequent itemsets, this is a surprising result

For, from earlier research [72] we know that the code tables produced by KRIMP determine the support of all itemsets rather accurately. More precisely, in that paper we showed that these code tables can be used to generate a new code table. The support of an arbitrary frequent item set in this generated database, say D_{gen}, is almost always almost equal to the support of that itemset in the original database, say D_{orig}. As usual, this sentence is probably more clear in its mathematical formulation:

$$P\left(|sup_{D_{orig}}(I) - sup_{D_{gen}}(I)| > \epsilon\right) < \delta$$

This surprise raises two immediate questions:

1. why does transforming KRIMP work and
2. can we transform frequent itemset mining?

The reason that transforming KRIMP work is firstly exactly the fact that it determines the support of all itemsets so well. Given a code table, we know the support of these itemsets. Clearly, as for the set of frequent itemsets, this means that we will overestimate the support of itemsets on the result of a query. However, different from the lifting approach, we do allow access to the query result and, hence, the overestimation can be corrected. This is the second reason why transforming KRIMP works.

This reasoning makes the question "can we transform itemset mining?" all the more relevant. Unfortunately, the answer to this question is *probably not*. This can be easily seen from the join. The input for the transformed itemset miner would be the joined tables as well as the Cartesian product of the sets of frequent itemsets on the "blown-up" individual tables. This set of candidate frequent itemsets will be **prohibitively large**, far larger than the final set of itemsets that is frequent on the join. Hence, checking all these candidates will be more expensive than computing only the frequent ones efficiently.

Pruning the set of candidates while searching for the frequent ones requires a data structure that stores all candidates. Whenever we can prune, a set of candidates has to be physically deleted from this data structure. The normal itemset miners do not even generate most of these pruned candidates. In this approach we would first generate and then delete them. In other words, it is highly unlikely that this approach will have a performance similar to the best itemset miners. Let alone that it will be significantly more efficient than these algorithms, as is required by the transformation approach.

In turn, this reasoning points to the third reason why transforming KRIMP works. The code tables KRIMP produces are small, far smaller than the set of

frequent itemsets. Hence, checking the support of all candidates suggested by Krimp is not detrimental for the efficiency of Krimp*.

From this discussion we can derive the following succinct all-encompassing reason why transforming Krimp works. Krimp produces, relatively, small code tables that capture the support of all itemsets rather well, such that checking the set of all suggested candidates is rather cheap.

Note that the comparison of the two approaches for a single case, viz., that of itemsets does not imply at all that the second approach is inherently superior to the first one. In fact, we already argued at the start of this section that the first approach, if applicable, is to be preferred above the second one. Moreover, in [61] it has been shown that the first approach is applicable for the discovery of Bayesian networks from data. In other words, the first approach is a viable approach.

A conclusion we can, tentatively, draw from the discussion in this section is that for either approach to work, the models should capture the data distribution well.

3.5 Conclusions and Prospects for Further Research

In this chapter we introduced a problem that has received little attention in the literature on inductive databases or in the literature on data mining in general. This question is: does knowing models on the database help in inducing models on the result of a query on that database?

We gave two approaches to solve this problem, induced by two interpretations of "help". The first, more elegant, one produces results without access to the result of the query. The second one does allow access to this result.

We investigated both approaches for itemset mining. It turned out that the first approach is not applicable to frequent itemset mining. While the second one produced good experimental results for our Krimp algorithm. In Section 3.4 we discussed this failure and success. The final tentative conclusion of this discussion is: for either approach to work, the models should capture the data distribution well.

This conclusion points directly to other classes of models that may be good candidates for either approach, viz., those models that capture a detailed picture of the data distribution. One example are Bayesian networks already discussed in [61]. Just as interesting, if not even more, are models based on bagging or boosting or similar approaches. Such models do not concentrate all effort on the overall data distribution, but also take small selections with their own distribution into account. Hence, for such models one would expect that, e.g., lifting the selection operator should be relatively straight forward.

This is an example for a much broader research agenda: for which classes

of models and algorithms do the approaches work? Clearly, we have only scratched the top of this topic. Another, similarly broad, area for further research is: are there other, better, ways to formalise "help"?

Chapter 4

Generating Answers to Queries

In the previous chapter we have seen that the code table KRIMP computes on a dataset D are close to the code table it computes on the result $Q(D)$ of an arbitrary query Q on D. Close in the sense that we essentially only have to do a recount of the usages of the elements of the code table; after all, that is what using KRIMP with the code table elements as only candidates more or less is.

In [72] it is shown how code tables can be used to generate transactions and complete databases. Moreover, these generated databases are similar to the original dataset. That is, if we start with a dataset D, compute its code table CT with KRIMP, and then use CT to generate a dataset D', then

$$D' \approx D.$$

If we combine this observation with the results of the previous chapter, a natural question to ask is: can we use CT to generate answers to an arbitrary query Q on D? That is, rather than accessing D to compute $Q(D)$ we want to generate an answer. In the light above, we want to generate a dataset $A(Q|CT)$ such that

$$A(Q|CT) \approx Q(D)$$

That is the problem we solve in this chapter. Clearly there is a solution to this problem. Use CT to generate a dataset D' and compute $Q(D')$. But that is not a very satisfying solution, in this Chapter we show that $A(Q|CT)$ can be generated directly from CT.

4.1 Formalizing the Problem

When Are Datasets Similar?

To formalize the problem, we first have to clarify what we mean by $A(Q|CT) \approx Q(D)$. Both are datasets, when are two datasets similar? Having a distance measure between datasets – over the same domain, of course, – would make it easy to formalize this similarity.

Such distance measures do exist. Moreover, at least one is not model-based, i.e., it does not assume that the data is distributed according to some specified model. This distance function is the *Normalised Information Distance* [69], which is defined for two strings x and y over $\{0, 1\}$ by:

$$d(x, y) = \frac{\max\{K(x|y), K(y|x)\}}{\max\{K(x), K(y)\}}$$

where $K(x)$ denotes the Kolmogorov complexity of x and $K(x|y)$ denotes the Kolmogorov complexity of x given y.

There is one problem in using this distance, however: the Kolmogorov complexity of a string is not computable. Fortunately, there is a computable distance function that approximates the Normalised Information Distance. This is the Normalised Compression Distance (NCD) [10]. Let C be an arbitrary compressor (e.g., gzip) and x and y objects than:

$$NCD_C(x, y) = \frac{C(xy) - \min\{C(x), C(y)\}}{\max\{C(x), Cy)\}}$$

in which $C(x)$ denotes the size of x when compressed with C.

We use, of course, the code tables produced by Krimp as compressor. Rather than the NCD itself, we use an asymmetric version here. The reason is that we have a gold standard, i.e., $Q(D)$ is the "true" answer, its code table is the "true" code table. Let D and D' be two datasets with the same number of transactions and let CT be the code table of D. The asymmetric dissimilarity score is defined by:

$$ADMD(D, D') = \frac{|\mathcal{L}(D'|CT) - \mathcal{L}(D|CT)|}{\mathcal{L}(D|CT)}$$

$ADMD$ is, obviously related to the ADM score of the previous chapter. While for ADM the data is "constant" and the models vary, for $ADMD$ the model (code table) is constant, while the datasets are different.

From research in classification with code tables, we know that compressing a dataset with the "wrong" code table gives very bad results [44]. Hence, a low ADMD score means that D and D' share their structure. Again from these experiments one can derive that an ADMD score of 0.2 or lower is good.

Probably, Mostly the Same

In the previous chapter we have seen that on average we get a good code table on $Q(D)$ by just using CT, but there is some variance. It works better for some queries than for others. So, requiring an absolute measure on all queries in this chapter is too stringent a requirement. Therefore we look at it probabilistically.

More in particular, let $\mathcal{Q}$ be a set of queries, we measure

$$P(ADMD(Q(D), A(Q|CT)) \geq \delta)$$

for $Q \in \mathcal{Q}$. The smaller this probability is for the smaller δ, the better our generated datasets are. That is, we are interested in the function $\mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, given by:

$$\delta \to \min\{\epsilon \in \mathbb{R}_{\geq 0} \mid P(ADMD(Q(D), A(Q|CT)) \geq \delta) \leq \epsilon\}$$

again for the queries $Q \in \mathcal{Q}$. If both δ and ϵ can be made small, our computed query results are probably, mostly the same as the query evaluated on the data.

Having an objective function with two variables is not as bad as it seems. For, in the previous subsection we already noted that having $\delta = 0.2$ is a reasonable requirement. Hence, we are mostly interested in having a small ϵ for this δ.

Problem Statement

With these definitions we can state the problem of this chapter as follows.

> Devise a data generator $A(Q|CT)$ such that for a given set of queries Q on D:
>
> $$\delta \to \min\{\epsilon \in \mathbb{R}_{\geq 0} \mid P(ADMD(Q(D), A(Q|CT)) \geq \delta) \leq \epsilon\}$$
>
> has solutions with small values both for ϵ and δ.

4.2 Generating Answers

Generating Transactions with Krimp

In [72] it is shown how code tables can be used to generate transactions. Since this is the starting point of our data generation here, we briefly review this method first.

We apply a Laplace correction to the code table. That is, we add 1 to all the usages to ensure that all alphabet elements in the code table have a usage of at least 1. In Chapter 2, we introduced the probability distribution $P(I|D)$ for

the $I \in CT$. This probability distribution can easily be reconstructed from the lengths of the codes in CT. To generate transactions, we simply pick itemsets from CT according to this probability distribution.

There is one subtle point in this generation. For, we are dealing with normal relational databases that are transformed to transaction databases. That is, multiple items are related to the same attribute A; one for each value of A. Denote by $\mathcal{I}_A$ the itemsets in $\mathcal{I}$ that are derived from attribute A in this transformation.

If a table has attributes $A_1, \ldots, A_n$, then the transactions in the transformed database will have contain exactly one item from each of the $\mathcal{I}_{A_i}$. The same should be true for generated transactions.

The first step is that we choose 1 itemset from CT, the probability that $I \in CT$ is chosen is simply $P(I|D)$. If I contains an item from each of the $\mathcal{I}_{A_i}$, we are done. That is I is a transaction in the generated database.

If not, we have to extend I. Let $\mathcal{A}_I$ denote the attributes that are already covered by I, i.e.,

$$\mathcal{A}_I = \{A_i \mid \exists J \in \mathcal{I}_{A_i} : J \subseteq I\}$$

Moreover, denote by $\mathcal{I}_{bar(I)}$ those items that can not be used to extend I, i.e.,

$$\mathcal{I}_{bar(I)} = \bigcup_{A \in \mathcal{A}_I} \mathcal{I}_A$$

To extend I we may only use itemsets $J \in CT$ for which $J \cap \mathcal{I}_{bar(I)} = \emptyset$. Because only then the resulting transaction will have exactly one item from each of the $\mathcal{I}_{A_i}$. Denote the set of these itemsets by $CT_{not-bar(I)}$. To extend I choose an element $J \in CT_{not-bar(I)}$, the probability that a given J is chosen is proportional to $P(J|D)$; proportional because the $P(J|D)$ will not sum to 1 over $CT_{not-bar(I)}$. If $I \cup J$ contains an itemset from each of the $\mathcal{I}_{A_i}$, we have generated a new transaction. If not we extend $I \cup J$ as just described.

Generating Query Answers

Given that we can generate datasets from code tables, there is a simple way to generate answers to queries. First generate a dataset and then compute the query result on this generated table. However, as noted at the start of this chapter, we want to generate the answer to the query directly. Here we discuss how this is done for the relational algebra operators projection π, selection σ, and the join $\bowtie$.

Project

For convenience, we use π as a "project out" operator; as we did in the previous Chapter. That is π_{A_1,A_2} means that we project the attributes A_1 and A_2 out

of the table, all other attributes remain. That is we should use items from each of the $\mathcal{I}_{A_i}$ except from $\mathcal{I}_{A_1}$ and $\mathcal{I}_{A_2}$.

To make this precise, let α denote the set of attributes that is projected out by π_α. Moreover, let

$$\mathcal{I}_\alpha = \bigcup_{A \in \alpha} \mathcal{I}_A$$

We generate transactions for π_α by first projecting out α from all the $I \in CT$. That is, every $I \in CT$ is replaced by $I \setminus \mathcal{I}_\alpha$, The probabilities of the itemsets in CT is not changed. That is, the probability that a modified I is chosen in the generation of the result is, again, simply $P(I|D)$.[1]

Select

The selections we consider in this chapter have a disjunction as a selection predicate. The reason for this is that conjunctive selection predicates quickly lead to very small query results and MDL (and, thus, Krimp) does not work very well for small datasets.

That is, we consider selections σ_ϕ in which $\phi = i_1 \vee \cdots \vee i_k$ for items $i_j \in \mathcal{I}$. Let $\mathcal{I}_\phi = \{i_1, \ldots, i_k\}$. To ensure that we generate transactions that satisfy σ_ϕ we simply start our generation by choosing an itemset $I \in CT$ such that $I \cap \mathcal{I}_\phi \neq \emptyset$. The probability of choosing such an I is, again, proportional to $P(I|D)$. After this start we continue with all itemsets of CT, except for those barred by I, of course. That is, we may choose itemsets which do have an empty intersection with $\mathcal{I}_\phi$ provided they are not barred by I.[2]

Join

The join is slightly more complex, because it involves multiple tables and, more importantly, the keys and foreign keys that are not encoded by the code table. Fortunately, there is a simple way around this key-structure: we simply "blow up" the tables involved. If the query is $T_1 \bowtie T_2$, we use the code table of $\pi_{T_1}(T_1 \bowtie T_2)$ instead of the code table of T_1 itself; note that only here π_{T_1} denotes the projection *on* the attributes of T_1. We blow up T_2 in the same way.

To generate a transaction for $\pi_{T_1}(T_1 \bowtie T_2)$, we simply generate a transaction for T_1 and T_2 using these blown-up code tables and pair these two transactions.

Complex Queries

To generate transactions for more complex "project-select-join" queries, we simply combine the generation procedures outlined above. For example, for a "project-select" query we first do the projection on the code table as above and

[1] This is actually too simple, as it allows combinations that are not possible with the full code table, hence we will overestimate some probabilities. However, the experiments show that this simple scheme already works well.

[2] Again this is too simple, but again it works reasonably well.

then we generate the results using the scheme for selection queries as, again, outlined above.

4.3 Experiments

For the experiments we have used three University of California, Irvine (UCI) datasets, viz., pen digits, letter, and chess (kr-k). These datasets are on the one hand big enough to allow Krimp to find good code tables. On the other hand, they are not too dense to preclude finding good code tables fast.

Since these datasets are single table datasets, they can not be used to test the join. For that we used the tables "Biopsy" and "Hematology" from the "Hepatitis Medical Analysis" of KDDCup 2004. We removed all tuples with missing data and converted all data in those tables into itemset data using LUCS-KDD ARM Discretisation/ Normalisation (DN) software from Liverpool University [13].

Finally, note that in each generated query answer was generated to the same size as the true answer to the query.

Simple Queries

For the projection and selection, we used a minimal support of 20 on letter and 50 for both pen digits and chess(kr-k). For the projection we projected out random sets of 1, 2, and 3 attributes. Each experiment was performed 30 times. The results are given in table 4.1. If we set δ to 0.2, which was indicated

Dataset	**Dissimilarity Scores ± Standard Deviation**		
	1 attr	**2 attr**	**3 attr**
chess(kr-k)	0.03 ± 0.01	0.03 ± 0.01	0.03 ± 0.01
letter	0.09 ± 0.01	0.15 ± 0.01	0.12 ± 0.01
pen digits	0.03 ± 0.01	0.04 ± 0.02	0.05 ± 0.02

Table 4.1: The results of projection experiments

as a reasonable choice earlier, ϵ is 0. Slightly more realistic is $\delta = 0.14$ with $\epsilon = 0.11$.

For the selection experiments, we used random sets of 1, 2, 3, 4, 7, and 9 elements. The results are given in table 4.2. First note that larger result sets give better scores, this is due to the fact that Krimp computes better code tables for bigger datasets. Both chess(kr -k) and letter do very well for $\delta = 0.2$. This is not true for pen digits. Here the query results remained too small for MDL to its work, only for the rightmost columns the results are OK.

Dataset	Dissimilarity Scores ± Standard Deviation					
	1 attr	**2 attr**	**3 attr**	**4 attr**	**7 attr**	**9 attr**
chess(kr-k)	0.07 ± 0.12	0.22 ± 0.17	0.17 ± 0.11	0.15 ± 0.04	0.14 ± 0.04	0.09 ± 0.05
letter	0.35 ± 0.25	0.27 ± 0.13	0.16 ± 0.09	0.11 ± 0.15	0.14 ± 0.07	0.16 ± 0.16
pen digits	0.42 ± 0.11	0.16 ± 0.02	0.21 ± 0.06	0.28 ± 0.04	0.31 ± 0.04	0.31 ± 0.03

Table 4.2: The results of selection experiments

For the join, we had only one dataset, we used a minimal support of 10 on these datasets. For 10 random trials we got an average dissimilarity of 0.06 with a standard deviation of 0.01. That is, even if we set $\delta = 0.1$, we still have a very small ϵ.

Complex Queries

First we did random selection and projection queries on the join. The results are given in table 4.3. Combining operators clearly increases the error; to a

Type	Dissimilarity Scores ± Standard Deviation		
	1 attr	**2 attr**	**3 attr**
Selection on join	0.22 ± 0.02	0.11 ± 0.01	0.09 ± 0.01
Projection on join	0.21 ± 0.01	0.20 ± 0.01	0.18 ± 0.01

Table 4.3: Selections and Projections on the Join

large extent that is due to the size of the query results. However, most of the results are still in the ball-park of 0.2. For example, $\delta = 0.25$ still gives an excellent ϵ.

Finally, we experimented with "project-select-join" queries. The results are summarised in table 4.4. Given that these queries result in the smallest tables, these results are not too bad. Clearly $\delta = 0.3$ is a necessary choice for a reasonable ϵ. But, then again, $\delta = 0.3$ is not that much worse than $\delta = 0.2$.

4.4 Related Work

The generation of answers to queries is, of course, related to the data generation for privacy protection reported on in [72]; we built upon the ideas in that paper. The novelty in this respect is that we have now shown that one can generate the answer to a query directly from the code table.

It is also interesting to contrast the results of the experiments in this chapter with the previous chapter. There we have shown that the itemsets in the code

# Attributes		
Project	**Select**	**DS ± STD**
1	*1*	0.28 ± 0.02
	2	0.46 ± 0.04
	3	0.51 ± 0.04
2	*1*	0.09 ± 0.06
	2	0.25 ± 0.04
	3	0.33 ± 0.04
3	*1*	0.18 ± 0.1
	2	0.17 ± 0.09
	3	0.25 ± 0.06

Table 4.4: Project-Select-Join Queries

table of the complete database are sufficient to determine the code table for the result of a query. Here we have shown that the probabilities of the code table of the complete database give a lot of information about the probabilities for the query result. Combining these two observations, it may be possible to compute a good code table for the result of a query directly from the code table of the complete database without accessing the data at all.

4.5 Conclusions

In this chapter we have shown how one can generate the answer to a query from the code table Krimp computes from the complete dataset. The experiments show that the generated answers are probably mostly the same to the results from applying the query to the complete dataset. This result can be paraphrased as: a code table is a good summary of the dataset; especially when combined with the results of the previous chapter. Anything you want to know about the data, you can learn from its code table.

Chapter 5

Improving Tag Recommendation: From Pairwise to Many and to Few Associations

Collaborative tagging services allow users to freely assign tags to resources. As the large majority of users enters only very few tags, good tag recommendation can vastly improve the usability of tags for techniques such as searching, indexing, and clustering. Previous research has shown that accurate recommendation can be achieved by using conditional probabilities computed from tag associations. The main problem, however, is that enormous amounts of associations are needed for optimal recommendation.

We argue and demonstrate that pattern selection techniques can improve tag recommendation by giving a very favourable balance between accuracy and computational demand. That is, few associations are chosen to act as information source for recommendation, providing high-quality recommendation and good scalability at the same time.

We provide a proof-of-concept of this idea using KRIMP as an off-the-shelf pattern selection method. Experiments on data from Delicious, LastFM and YouTube show that our proposed methodology works well: applying pattern selection gives a very favourable trade-off between runtime and recommendation quality.

5.1 Introduction

Online collaborative tagging platforms allow users to store, share and discover resources to which tags can be freely assigned. Well-known examples include Delicious (bookmarks), Flickr (photos), LastFM (music), and YouTube (videos). Tags can be chosen completely freely, allowing both generic and very specific tags to be assigned. As a result, tags enable very powerful tools for e.g. clustering [7, 42] and indexing [46] of resources.

Tagging systems also have their downsides though. Apart from linguistic problems such as ambiguity and the use of different languages, the most apparent problem is that the large majority of users assign only very few tags to resources. Sigurbjörnsson and Van Zwol [65] did an analysis of tag usage on photo sharing platform Flickr, and showed that 64% of all tagged photos are annotated with 3 or less tags. Obviously, this severely limits the usability of tags for the large majority of resources.

As a solution to this problem, methods for a wide range of tag recommendation tasks have been proposed. In this paper we consider the most generic tag recommendation task possible. That is, we only assume a binary relation between resources and tags. Because it is so generic, it can be applied in many instances, e.g. on platforms where heterogeneous resource types co-exist, making it impossible to use resource-specific methods. Also, many collaborative tagging platforms maintain only a single set of tags per resource, irrespective of which user assigned which tag.

Sigurbjörnsson and Van Zwol [65] were the first to propose a recommendation method for this setting, based on pairwise tag co-occurrences in the tagsets previously assigned to resources. Using these existing tag assignments, referred to as collective knowledge, candidate tags are generated and subsequently ranked by (aggregated) conditional probabilities. LATRE, introduced by Menezes et al. [52], builds upon this by using larger sets of co-occurring tags, in the form of association rules, instead of only pairwise co-occurrences. To avoid mining and caching enormous amounts of associations, LATRE mines the needed rules for each query individually. The empirical evaluation showed that it is indeed beneficial to use tag associations consisting of more than just two tags.

The big problem, however, is that although 'on-demand' mining circumvents the need to cache millions of associations, the *online* mining of association rules is computationally very demanding. As such, LATRE provides better recommendations at the cost of scalability.

Aims and contributions

Although recommendation using pairwise associations [65] is both accurate and very fast, exploiting associations with more than two tags can improve accuracy even further [52]. Unfortunately, this comes at the cost of memory space and computational complexity. In this chapter, we demonstrate that the balance between accuracy and computational complexity can be improved by using a carefully selected *small set* of associations. To be more precise, we will show how pattern selection can positively contribute to association-based recommendation.

In Section 5.2, we will first provide the notation that we will use throughout the chapter, and formally state the recommendation problem that we consider. After this, Section 5.3 will explain association-based recommendation in more detail. First, the above mentioned method using pairwise associations will be detailed. Second, both our own generalisation to larger associations and LATRE will be discussed. Third, we will introduce FastAR, for Fast Association-based Recommendation, which needs only few associations to achieve high accuracies, making it much faster than its competitors. For this, FastAR uses associations selected by the Krimp algorithm [64], the pattern selection method based on the Minimum Description Length principle we used before. Given a database and a large set of patterns, it returns a set of patterns that together compresses the database well. These so-called *code tables* have been shown to provide very accurate descriptions of the data, which can be used for e.g. tag grouping [42].

After all methods have been introduced, they will be empirically compared in Section 5.4. Finally, we round up with related work and conclusions in Sections 5.5 and 5.6.

5.2 Tag Recommendation

We consider binary relations between a set of resources $\mathcal{S}$ and a set of tags $\mathcal{T}$, i.e. $\mathcal{S} \times \mathcal{T}$. That is, each tag can be assigned to any number of resources, and each resource can have any number of tags assigned. In the following, each resource is represented solely by its set of associated tags, simply dubbed a *transaction*; our terminology is mostly equivalent to what we have used before for itemset mining, but now translated to tags.

Let database D be a bag of transactions over $\mathcal{T}$. A transaction $t \in D$ is a subset $t \subseteq \mathcal{T}$, and $|D|$ denotes the number of transactions in D. A tagset X is a set of tags, i.e. $X \subseteq \mathcal{T}$, X occurs in a transaction t iff $X \subseteq t$, and the length of X is the number of tags it contains, denoted by $|X|$. Maximum length parameter *maxlen* restricts the number of tags a tagset X may contain,

i.e. $|X| \leq maxlen$. The support of a tagset X in database D, denoted by $sup_D(X)$, is the number of transactions in D in which X occurs. That is, $sup_D(X) = |\{t \in D \mid X \subseteq t\}|$. A tagset X is *frequent* if its support exceeds a given minimum support threshold $minsup$, i.e. $sup_D(X) \geq minsup$.

The problem

Informally, we consider the following tag recommendation task. Assume given a database D consisting of tagsets that have previously been assigned to resources. When a user assigns a new tagset I to a resource, the recommendation algorithm is invoked. Using source database D and query I as input, it recommends a tagset $R(D, I) \subseteq (\mathcal{T} \setminus I)$ to the user. Unlike the use of the word 'set' may suggest, order is important; $R(D, I)$ has to be ranked according to relevance to the user.

In practice, query I consists of very few tags in the large majority of cases, since most users do not manually specify many tags. Therefore, high-quality tag recommendation is of utmost importance for resources annotated with 3 or fewer tags. All the more so, since these input sizes potentially benefit most from recommendation: when more than 3 tags are given, recommendation is probably less necessary.

Apart from the quality of the recommendations, performance from a computational point of view is also important. Tag recommendation is often implemented in online web services and thus needs to be fast. Although we do not formalise this in the problem statement, we will evaluate this in the Experiment section. The tag recommendation problem can be stated as follows.

Problem 1 (Tag Recommendation)**.** *Given a source database D over tag vocabulary $\mathcal{T}$, and an input tagset I, recommend a set of tags $R(D, I) \subseteq (\mathcal{T} \setminus I)$ ranked by relevance.*

5.3 Association-based Tag Recommendation

In this section we explain and motivate association-based tag recommendation.

Pairwise Conditional Probabilities

Sigurbjörnsson and Van Zwol [65] were the first to propose a method for tag recommendation that uses only the tagsets previously assigned to resources, which they referred to as collective knowledge. The method is based on pairwise tag co-occurrences in this collective knowledge. In short, given a query I, all tags co-occurring with an $i \in I$ are ranked based on their conditional probabilities.

In more detail, it works as follows. Given an input tagset I and source database D, a candidate list C_i of the top-m most frequently co-occurring tags (with i) is constructed for each $i \in I$ (where m is a parameter). For each candidate tag $c \in C_i$, its empirical conditional probability is then given by

$$P(c \mid i) = \frac{sup_D(\{c, i\})}{sup_D(\{i\})}. \tag{5.1}$$

For any singleton input tagset $I = \{i\}$, the candidate tags in C_i are simply ranked according to $P(c \mid i)$ and then returned as recommendation. For any longer I, however, the rankings obtained for each individual input tag will have to be aggregated. The authors proposed and tested two different strategies for this, i.e. *voting* and *summing*, but we will focus on the latter as this was shown to perform better. First, a set C containing all candidate tags is constructed: $C = \bigcup_{i \in I} C_i$. Next, individual conditional probabilities are simply summed to obtain a score $s(c)$ for each candidate tag $c \in C$. This is given by

$$s(c) = \sum_{i \in I} P(c \mid i). \tag{5.2}$$

Finally, all candidate tags are ranked according to score function s, providing the final recommendation. In the original chapter [65], the concept of *promotion* was introduced to weigh certain tags, which slightly improved recommendation quality. However, this requires parameter-tuning and to avoid unfair comparisons, we do not consider any tag weighing schemes in this chapter.

The principle of using conditional probabilities is very strong, and maybe it is for that reason that very few improvements on this technique have been proposed since (considering only methods that assume exactly the same task). We will compare to this baseline method in Section 5.4. For this purpose, we will dub it PairAR (or PAR for short), for Pairwise Association-based Recommendation.

Many Associations

An obvious generalisation of PAR is to use not only *pairwise* associations between tags, but associations of *any length*. This may lead to better recommendation whenever the input tagset contains more than one tag, because we can compute more accurate empirical conditional probabilities. Suppose for example that we have an input tagset $I = \{x, y\}$ and a candidate tag z. With PAR we would compute $s(c) = P(z \mid x) + P(z \mid y)$ and use this for ranking, but with longer associations we could also compute $P(z \mid xy)$ and exploit this information.

The most naïve approach to do this would be to compute and use *all* associations of length $|I| + 1$, which would be sufficient to compute all needed conditional probabilities. Unfortunately, were we to cache all associations, this would come down to mining all frequent tagsets up to maximum length $|I|+1$, with a minimum support threshold of 1. In practice, this is infeasible due to the so-called pattern explosion, and larger *minsup*s are also problematic for realistically sized datasets. Even if it is at all possible to mine the desired associations, using all of them for recommendation is likely to be slow.

Nevertheless, we adopt this approach to see how it performs in practice and dub it NaiveAR (or NAR for short). First, it mines all frequent tagsets $\mathcal{F}$, with $maxlen = |I|+1$ (hence a different set of associations for each input length is required) and a specified *minsup*. Additionally, the top-m co-occurrences per tag are computed as is done by the PairAR baseline. This to ensure that there are always candidate tags, even for rare input tags.

As candidates, all tags that co-occur with any of the input tags in any $F \in \mathcal{F}$ are considered, augmented with the top-m tags for each individual input tag. Conditional probabilities and scores $s(c)$, as defined in Equation 5.2, are computed as before, except that $\mathcal{F}$ is consulted for any conditional probabilities that cannot be obtained from the top-m's. There is a good reason for using the summing strategy of $s(c)$ despite having access to the 'exact' conditional probability $P(z \mid xy)$. That is, this exact probability is not always available due to the *minsup* parameter, and for those cases the summing strategy has the same effect as with PAR. Preliminary experiments showed this to be a good choice.

As discussed, LATRE, introduced by Menezes et al. [52], is a very similar method that also uses longer associations. To overcome the pattern explosion, LATRE uses on-demand mining of association rules for each individual query. Three parameters can be used to reduce the number of rules: 1) a maximum number of tags per rule, 2) minimum support, and 3) minimum confidence. Note that the main difference with NAR is that NAR uses a higher *minsup* to mine associations once beforehand and augments these with the top-m co-occurring tags for each input tag. We will empirically compare to LATRE in Section 5.4.

From Many to Few Associations

Although experiments will show that NaiveAR and LATRE can improve on PairAR in terms of precision, the downside is that both methods are rather slow. Unfortunately, most users would rather skip recommendations than wait for them. The question is therefore whether a small set of associations could already improve precision, such that recommendation is still (almost) instantly.

That is, we are looking for a pattern set that provides a complete description of (the associations contained in) source database D. In previous chapters we have shown that *code tables* produced by KRIMP provide such a description.

Analogue to NAIVEAR, the set of tagsets provided by the first column of a code table is augmented with the top-m pairwise co-occurrences for each input tag. Also, the set of candidate tags is obtained in exactly the same way: each tag that co-occurs with any of the input tags in the code table is considered a candidate tag, as are the top-m tags for each input tag.

Although the code table does not store tagset supports, we can infer information from the lengths of its codes. As discussed in Section 2.2, the code for each $X \in CT$ is chosen such that its length is

$$-\log\left(\frac{usage_D(X)}{\sum_{Y \in CT} usage_D(Y)}\right).$$

Given an optimal code table CT, one can compute a tagset's support by summing the usages of all code table elements that are a superset of the tagset. As we have seen before, computing the optimal code table is infeasible. However, we know that the KRIMP code tables are rather accurate. Hence, we estimate the support of any tagset X by

$$\hat{sup}_D(X) = \sum_{Y \in CT, X \subseteq Y} usage_D(Y),$$

which can be easily computed from the code table.

Again, computing the scores for the candidate tags is done as by PAR and NAR, except that code table CT is consulted for any conditional probabilities that cannot be obtained from the top-m's. In these cases, the support of a tagset X is estimated by $\hat{sup}_D(X)$.

Note that the code table itself can be computed offline. While computing recommendations, we only need to estimate supports. Given that a code table is significantly smaller than the set of all (frequent) tagsets, this algorithm should be much faster than both NAR and LATRE. We therefore dub it FASTAR (or FAR for short), for Fast Association-based Recommendation.

5.4 Experiments

In this section we evaluate PAIRAR, NAIVEAR, FASTAR, and LATRE on three datasets collected from online collaborative tagging services.

Evaluation criteria To assess recommendation quality, we resort to the most commonly taken approach, i.e. 5-fold cross-validation. The dataset is partitioned into 5 equal-sized parts and for each of 5 folds, four parts are concatenated into a training database D^{train} and the remaining part D^{test} is used for

Table 5.1: Datasets. The number of transactions and distinct tags, as well as average and maximum transaction lengths are given. $|D^k|$ represents the number of transactions used for testing after removing transactions that are too short, for $k \in \{2, 3\}$.

Dataset	*Properties*				*Effective test data*	
	$\|D\|$	$\|\mathcal{T}\|$	avg($\|t\|$)	max($\|t\|$)	$\|D^2\|$	$\|D^3\|$
Delicious	526,490	19,037	9.6	30	338,284	308,832
LastFM	91,325	7,380	20.3	152	61,701	56,904
YouTube	169,290	7,785	8.3	74	97,591	83,450

testing. All results are averaged over all 5 folds (except for runtime, which is summed).

D^{train} is used as source database D for all methods. From each tagset $X \in D^{\text{test}}$, a query $I \subset X$ is randomly selected. The remaining tags in the tagset, $X \setminus I$, are used for validation. The size of query I is parametrised by k and fixed for a given experiment, s.t. $k = |I|$. Since small queries are most prominent in real world situations, we consider $k \in \{2, 3\}$. Note that we do not consider $k = 1$, because the proposed methods are the same as PairAR for this setting. To ensure that there are always at least five validation tags that can be used to measure precision, we exclude all transactions that contain less than $k + 5$ tags from D^{test}.

As evaluation criteria we use precision, mean reciprocal rank and runtime, denoted by P@x, MRR and time. P@x denotes precision of the x highest ranked tags, with $x \in \{1, 3, 5\}$, and is defined as the average percentage of the first x recommended tags that occur in validation tags $X \setminus I$. Mean reciprocal rank represents the average rank of the first correctly recommended tag. It is defined as $MRR = \frac{1}{|D^{\text{test}}|} \sum_{X \in D^{\text{test}}} \frac{1}{first(X)}$, where $first(X)$ is the rank of the first correctly recommended tag for test case X.

To quantify computational demand we measure runtime, on average per query. Note that this excludes the time needed for generating the associations needed by PAR/NAR/FAR, as this can be done *offline* beforehand.

Datasets We use the pre-processed datasets crawled from Delicious[1], YouTube[2], and LastFM[3] by Menezes et al. [52]. Some basic properties are presented in Table 5.1; see their paper for more details on the collection and pre-processing. Note that we experiment on the complete datasets, which contrasts their setup.

[1] `www.delicious.com`
[2] `www.youtube.com`
[3] `www.last.fm`

Delicious is a social bookmarking service that allows users to store, share and discover bookmarks, i.e. URLs. Tag assignment is collaborative, meaning that all users together assign a single tagset to each resource. The dataset consists of tagsets crawled from the 'Recent Bookmarks' page in October 2009.

LastFM is a large online music catalog that allows users to keep track of the tracks they play. Further, users collaboratively maintain tags for artists. The dataset consists of tagsets corresponding to artists crawled in October 2008.

YouTube is the largest video sharing platform on the Web. It is not collaborative; only users who uploaded a video are allowed to assign tags to it. This dataset was crawled in July 2008.

PAR, NAR and FAR In all cases, $m = 50$, i.e. the top-50 most frequent co-occurring tags for each tag are collected. For NAR, tagset collection $\mathcal{F}$ is obtained by mining all closed frequent tagsets with *minsup* as given in Table 5.2 and $maxlen = k + 1$ (as was already specified in the previous section). For FAR, KRIMP is used to obtain a code table CT from all frequent closed tagsets for the specified *minsup* (no *maxlen*).

LATRE For comparison to LATRE, we use the original implementation and the same settings as in the original chapter: *minsup* = 1, *minimum confidence* = 0.00001, and $\alpha_{max} = 3$ (implying that association rules with antecedent up to the length 3 are allowed). Note that strictly speaking we compare to LATNC, a variant of LATRE without 'calibration', a tag weighing scheme like PAR's promotion. We decided to refrain from such schemes to keep comparison fair. Promotion and/or calibration could be applied to all methods described in Section 5.3, but require fine-tuning.

Implementation Prototypes of PAR, NAR and FAR were implemented in C++. Each experiment was run single-threaded on machine with a quad-core Intel Xeon 3.0GHz CPU and 8Gb RAM running Windows Server 2003.

Results

A complete overview of the results is presented in Table 5.2. PAR performs quite well in terms of precision and it is extremely fast, requiring up to 5 seconds for the online recommendation phase. We chose the *minsup*s for NAR such that the number of resulting tagsets in $\mathcal{F}$ was in the order of the $100,000s$; these amounts of tagsets could be kept in memory and recommendation was computationally feasible. Looking at the precision of the highest ranked recommended tags, we observe that NAR provides improvements up to 2.71% over PAR. Only for LastFM with $k = 3$ precision slightly decreases. Due to the large numbers of tagsets to be considered, runtimes increase up to 1229.32 milliseconds per query.

To demonstrate the effect of pattern selection, we ran FAR with the same *minsup*s as NAR. This shows that Krimp significantly reduces the number of tagsets. For LastFM with $k = 2$, for example, $\mathcal{F}$ contains $540,697$ tagsets, whereas the code table used by FAR contains only $9,513$ tagsets. With this decrease in the number of tagsets, runtime is reduced from 1229.32 to only 4.2 milliseconds per query, while an –admittedly small– increase in precision with respect to PAR is still attained. When we consider YouTube with $k = 2$ (or similarly $k = 3$), running NAR with a *minsup* lower than 13 was not feasible, while FAR could easily be applied with a *minsup* of 1. This resulted in a code table consisting of only $14,981$ tagsets. Despite the modest number of tagsets, this gave an increase in P@1 of 4.73%, which is substantially better than the 2.71% improvement obtained by NAR.

The relative performance of LATNC varies from setting to setting: precisions and MRR are subpar for Delicious, for LastFM with $k = 2$ and for YouTube with $k = 2$. For LastFM with $k = 3$, the scores are better than those of any other method, and LATNC is only beaten by FAR for YouTube with $k = 3$. LATNC is always much slower than FAR though.

Analysis

The overall good performance of PairAR shows that the idea of summing pairwise conditional probabilities is hard to beat. It very much depends on the data at hand whether associations consisting of multiple tags can be exploited to improve recommendation quality. This is not the case for the Delicious dataset that we used, which is probably due to the relative large number of tags, of which many occur only very rarely. For the YouTube data, however, recommendation can be substantially improved using FastAR. That is, for this dataset our method based on pattern selection beats the more 'exhaustive' methods with respect to both precision and runtime.

Whenever NAR and/or LATNC achieve higher precisions than PAR, FAR provides a much better trade-off between precision and runtime. The downside is that it does not always give the highest possible precision. Mining all association rules on demand, such as LATNC does, does not seem to be a good idea for realistically sized datasets, as this comes at the cost of long runtimes.

Using pattern selection, on the other hand, comes at the cost of longer offline pre-processing times. Krimp requires up to 15 minutes to obtain code tables for Delicious and YouTube, and up to 10 hours for LastFM. However, this is mostly due to the fact that it needs to generate and test a large set of candidates, which could be avoided by using different heuristics to construct code tables [66]. In the end, performing pattern selection once beforehand is well worth the effort if this results in much faster yet high-quality recommendation.

Table 5.2: Results. For each combination of dataset, query size k, method, and *minsup*, the obtained precisions, MRR and runtime are given. For NaiveAR and FastAR, the number of tagsets in $\mathcal{F}$ resp. CT are given. Runtime is given per query, on average in milliseconds.

Dataset	k	method	*minsup*	#tagsets	P@1	P@3	P@5	MRR	time (ms)
Delicious	2	PAR	-		45.93	35.19	29.20	56.95	0.01
		NAR	37	142,185	46.80	35.79	29.64	55.68	59.07
		FAR	37	25,736	46.28	35.42	29.36	55.26	6.89
		LATNC	1		39.39	29.82	24.47	50.34	129.88
Delicious	3	PAR	-		51.13	39.27	32.53	62.35	0.02
		NAR	37	219,832	51.33	39.35	32.60	60.59	428.91
		FAR	37	25,736	50.86	39.18	32.54	60.43	11.83
		LATNC	1		49.14	37.98	31.60	59.77	606.06
LastFM	2	PAR	-		52.18	41.38	35.11	63.02	0.03
		NAR	51	540,697	53.49	42.60	36.18	62.41	1229.32
		FAR	51	9,513	52.70	42.04	35.78	61.63	2.16
		LATNC	1		46.36	37.75	32.17	58.43	156.41
LastFM	3	PAR	-		52.59	42.40	36.31	64.22	0.05
		NAR	146	132,103	52.16	41.97	36.11	61.88	415.44
		FAR	146	5,111	52.46	42.84	36.86	62.59	1.02
		FAR	51	9,513	52.91	43.27	37.41	62.90	4.20
		LATNC	1		55.42	45.15	38.83	66.49	596.18
YouTube	2	PAR	-		50.10	39.86	34.05	58.69	0.04
		NAR	13	324,696	52.81	42.48	36.29	59.03	300.26
		FAR	13	10,159	52.12	41.81	35.73	58.34	3.30
		FAR	1	14,981	54.84	44.86	38.78	60.46	19.08
		LATNC	1		38.86	31.00	26.54	48.29	63.50
YouTube	3	PAR	-		54.90	43.83	37.42	63.53	0.06
		NAR	40	183,228	55.73	44.32	37.66	62.17	562.79
		FAR	40	4,933	55.63	44.37	37.72	62.29	1.69
		FAR	1	14,981	59.20	48.44	41.83	64.87	45.37
		LATNC	1		55.81	46.20	40.15	63.71	150.68

Finally, note that we here focused on relatively small query sizes (k) because these are both realistic and the most useful. However, exploiting longer associations can clearly be even more beneficial when k grows. This explains why LATRE performed much better than PAR in Menezes et al. [52].

5.5 Related work

Many variants of tag recommendation have been studied in recent years [51]. Based on the used information, we split existing methods into three categories.

The first category is the most generic, and the one we consider in this chapter: methods that only assume a binary relation between resources and tags. We already introduced and experimented with the methods in this category [52, 65].

The second category aims at tagging systems that allow users to individually assign their own tagset to each resource. The resulting ternary relation is often called a *folksonomy*. One of the first methods for folksonomies was Folkrank [31], based on the same principles as PageRank. It outperforms baseline approaches like collaborative filtering [34]. Also, a system composed of several recommenders that adapts to new posts and tunes its own parameters was proposed [48].

The third category is characterised by its use of resource-specific information. These systems use the resources themselves to improve recommendation. Examples include methods specifically designed for documents [46, 67], web pages [29], YouTube videos [68], and songs [40]. Finally, Rae et al. proposed to use social network information to improve tag recommendation [58].

Note that we cannot compare to methods from the latter two categories, as they all make additional assumptions about the recommendation task.

5.6 Conclusions

Conditional probabilities based on pairwise associations allow for high-quality tag recommendation, and this can be further improved by exploiting associations between more than two tags. Unfortunately, doing this naïvely is infeasible in realistic settings, due to the enormous amounts of associations in tag data.

To overcome this problem, we propose to use the strengths of pattern selection. Using Krimp as 'off-the-shelf' pattern selection method, we have demonstrated that our FastAR method gives a very favourable trade-off between runtime and recommendation quality. Because pattern selection is done offline and results in small pattern sets, online recommendation is fast.

FastAR uses Krimp to pick a small set of tagsets. However, the used coding scheme is not specifically designed for selecting associations that are useful for recommendation. Despite this, the presented results are encouraging. By modifying the encoding to better reflect the needs of tag recommendation, e.g. by allowing overlapping tagsets, we believe that the results could be further improved.

Acknowledgments

We would like to thank Adriano Veloso for kindly providing the datasets and LATNC implementation.

Chapter 6

The Impact of Social Networks on Tag Recommendation

Social networking sites such as Flickr and Facebook allow users to share content with family, friends, and interest groups. In addition, tags can often be freely assigned to resources. In the previous chapter we have seen that high-quality and efficient association-based tag recommendation is possible, but the set-up that we considered was very generic and did not take social information into account. FastAR in particular, the method that we proposed, exhibited a favourable trade-off between recommendation quality and runtime. Unfortunately, recommendation quality is unlikely to be optimal because the algorithms are not aware of any social information that may be available.

To address this issue, we propose two approaches that take a more social view on tag recommendation. As we do not want to change a method that works well, instead of modifying the algorithm we vary the data that is used as source of associations. We dub the first social variant User-centered Knowledge, to contrast Collective Knowledge. It improves tag recommendation by grouping historic tag data according to friend relationships and interests. The second variant is dubbed 'social batched personomies' and attempts to address both quality and scalability issues by processing queries in batches instead of individually, such as done in a regular personomy approach.

The empirical evaluation on data crawled from Flickr shows that we can improve tag recommendation by taking social information into account. Furthermore, the experiments provide an additional, independent confirmation of the conclusions that we drew in the previous chapter.

6.1 Introduction

Two kinds of recommender systems currently dominate the literature. The first kind consists of recommender systems based on "collective knowledge", i.e. large collections of historic tagsets assigned to resources, by any users. When each user can assign his/her own tagset to a resource, contrary to the set-up that we considered in the previous chapter, this is usually called a 'folksonomy'. The second kind consists of "personalised" recommender systems that only use annotations from the history of a given user. Such 'personomies' generally perform well if the user is rather active and thus has sufficient history.

Hence, existing systems either assume that available tag data should be used, or only personal tag data. However, in the case of social networks such as Flickr and Facebook, users can also maintain 'friendship' relations with other users. Since it seems safe to assume that people are influenced by their friends and family, this suggests a third type of recommender system, i.e. one that exploits 'social' information. Hence, the question that we address in this chapter is the following: *Can we improve tag recommendation by exploiting social information?*

In the previous chapter, we demonstrated the performance of three association-based tag recommendation algorithms in the situation where no user information is available. That is, all tagsets previously assigned to resources are together considered as collective knowledge. Now that we assume that both user and social information is available, the situation is different. However, directly integrating user and social information in the association-based algorithms is far from straightforward. Furthermore, PAR, NAR, and FAR have been proven to perform well; why would we change them?

The other possible approach, and the one that we will consider, is that of modifying the data that the algorithms use for mining associations and computing the corresponding conditional probabilities. Instead of using the complete set of collective knowledge as source database, we will introduce 'social' variants that use only a subset of the available data. By doing this, we hope to improve performance in terms of both recommendation quality and speed.

In the next section, we will first discuss existing work related to social tag recommendation. After that, in Section 6.3, we will introduce our notation and formally define the problem of Social Tag Recommendation. It is mostly the same as the Tag Recommendation problem that we defined in Section 5.2, except that we allow the system to exploit user information and the social network that the user is in.

Section 6.4 presents our two approaches to social tag recommendation. The first is dubbed *User-centered Knowledge* and is based on the intuition that people are influenced by nearby people that share the same interest. Whereas

one would normally use all historic tag data, we propose to use only tagsets assigned by a user and his/her friends (and possibly friends of friends) that all share the same interest. If this results in better recommendation, this is a strong indication that social information can be used to improve tag recommendation.

The second approach is dubbed (*Social*) *Batched Personomies* and is applicable on platforms where tens or hundreds of queries are issued every second. Inspired by personomies, where only an individual's tag history is used, we start from personal tag histories. However, by processing queries in batches, we both avoid huge computational loads and problems with users that have no history. And as with user-centered knowledge, we can augment these batched personomies with tag data obtained from friends.

We evaluate our approaches with PAR, NAR, and FAR, on data crawled from the social photo sharing platform Flickr. The experiments are presented in Section 6.5. Finally, we round up with a brief analysis and conclusions.

6.2 Related Work

We have already discussed the generic association-based recommendation methods by Sigurbjörnsson and Van Zwol [65] and Menezes et al. [52] in the previous chapter. In this section, we will therefore only discuss relevant work in the context of social tag recommendation.

Rae, Sigurbjörnsson and Van Zwol [58] proposed to combine different 'contexts' to improve tag recommendation in social networking environments. They investigated whether the Social Group context (a user's interest group) and Social Contact Context (a user's friends) can contribute to the performance of the system, in particular when used in combination with the more conventional Personal and Collective context. They do this with a straightforward voting approach. As baseline they use both the Personal Context and the Collective Context. According to their paper, Social Contact Context (i.e. using a user's friends' tag data) perform so badly that it is harmful to overall performance when used in combination with other contexts. In other words, the tagging behaviour of a user's contacts is unhelpful for tag recommendation.

Different from their approach [58], we explore the combination of different information sources by physically selecting/appending tag data, and then performing recommendation based on the newly created knowledge source. This is obviously different from a voting approach, as data instead of associations are combined and the resulting associations are thus appropriately weighted.

Hu, Wang, Liu and Li [32] investigated social influence by Gaussian-type topological potential, in which they rank user influence in social contact networks and choose the top-N contacts to form a 'preference community'. Tagging information from friends-of-friends is used to make the recommendation more

diverse, then through measurement of user social influence they find those who really have an impact on the target user [32].

6.3 Social Tag Recommendation

We consider ternary relations between a set of resources $\mathcal{S}$, a set of tags $\mathcal{T}$, and a set of users $\mathcal{U}$, i.e. $\mathcal{S} \times \mathcal{T} \times \mathcal{U}$. That is, each tag can be assigned to any combination of resource and user, each resource can have any number of tags assigned by any user, and each user can have zero or one tagset for each resource. For ease of presentation, in the following we consider assigned tagsets to be *transactions*, associated to a resource $s \in \mathcal{S}$ and a user $u \in \mathcal{U}$.

Let database D be a bag of transactions over $\mathcal{T}$. A transaction $t \in D$ is a subset $t \subseteq \mathcal{T}$ that is associated to a user $u \in \mathcal{U}$ and $s \in \mathcal{S}$. The remainder of the transaction-related notation is equal to that used in the previous chapter.

Next to the transaction data as just discussed, we assume that we have access to a social network for users $\mathcal{U}$. Let a social network be an undirected graph $G = (V, E)$, in which V is a set of vertices and E a set of edges. Each vertex $v \in V$ corresponds to one user $u \in \mathcal{U}$, and each edge $e \in E$ represents a 'friendship' relation between two users. We define the distance between two users u and v, denoted by $d(u, v)$, as the geodesic distance between the corresponding vertices in the graph.

Given a user u in a social network G, we define the nth-*degree* friends of u as the set of users that have distance n to u in G. For example, all *direct* friends of user u are 1st-degree friends. When starting from a user u, a single *hop* in the social graph brings us to the users one degree further away from u. A group of users $U \subseteq \mathcal{U}$ is *connected* if there exist a path between each pair of users in graph G. We will often omit G because there is only one social graph per dataset and G is thus clear from the context.

The Problem

The social tag recommendation task that we consider in this chapter is different from the one that we considered in the previous chapter, as we take the users and social network into account. Informally, assume given a database D consisting of tagsets that have previously been assigned to resources, and a social network G corresponding to the set of users $\mathcal{U}$. For each tagset, we know which user assigned it to the resource. When a user u assigns a new tagset I to a resource, the recommendation algorithm is invoked. Using a source database D, social network G, query I and user u as inputs, it recommends a tagset $R(D, G, I, u) \subseteq (\mathcal{T} \setminus I)$. That is, apart from the collective knowledge D and query I, the social network and user are provided as additional input. The rec-

ommended tags, $R(D, G, I, u)$, must be ranked according to relevance to the user.

Obviously, the high-level goals the we aim for are the same as in the previous chapter: we strive for high-quality association-based tag recommendation in as little time as possible. And, as before, we again stress the importance of good performance for small queries I ($k = \{1, 2, 3\}$), since these situations are both most realistic and benefit the most from recommendation.

The social tag recommendation problem can now be stated as follows.

Problem 2 (Social Tag Recommendation)**.** *Given a source database D over tag vocabulary $\mathcal{T}$, users $\mathcal{U}$ and corresponding social network G, and an input tagset I provided by user u, recommend a set of tags $R(D, G, I, u) \subseteq (\mathcal{T} \setminus I)$ ranked by relevance.*

The main question is now whether social tag recommendation can provide higher-quality recommendation than the 'regular' tag recommendation task formalised in Section 5.2, i.e. when we do not use any user and/or social information.

6.4 Algorithms

In the previous chapter we investigated the performance of the state-of-the-art when it comes to association-based tag recommendation, and proposed additional methods ourselves. In this chapter, we will use the three main variants PairAR, NaiveAR, and FastAR as base methods, but we will vary the source database D that is used as input for the recommendation methods. After a very quick recap of the base methods, we will explain the variants of the source data that we use.

PairAR (or PAR) is the name that we use for the method proposed by Sigurbjörnsson and Van Zwol [65]. Next, NaiveAR (or NAR) is a generalisation that employs tag associations of *any length*, that is, up to length $|I| + 1$ (which is sufficient to compute all needed conditional probabilities). We saw in the previous chapter that this results in the highest precisions, but also requires most runtime. Finally, we introduced FastAR (or FAR), which augments the pairwise associations from PAR with associations selected with Krimp. Compared to NAR, this results in slightly lower precisions but much shorter runtimes.

In the previous chapter, we used *all* available historic tag data as source of information for recommendation, i.e. as D. We will refer to this choice of source database D as *collective knowledge* (CK), a term that has been used before [65]. When user (but no social) information is available, this is sometimes called a *folksonomy* [34].

At the other extreme, we could consider users one by one. To be more precise, all historic tag assignments of user u, denoted by D^u, can be used as source database for recommendation. This case is usually called a *personomy* [47]. Although such a set-up leads to fully personalised recommendations, this also has some disadvantages: 1) this approach cannot be used when no historic assignments are available for a user, and 2) doing full-fledged personalised association-based recommendation can be computationally intensive, as associations need to be mined for each user individually.

Also, observe that neither of the two approaches takes the social graph into account, which we will do in the following.

User-centered Knowledge

The first alternative that we consider in this chapter is centered around the user, in terms of both a user's social network and his or her interests. In terms of the amount of tag data used, it lies somewhere between collective knowledge and a personomy.

As has been argued before [58], several social factors play an important role in improving recommendation quality. Firstly, friends influence a user, hence using tags assigned by a user's friends can be beneficial. Obviously, to a lesser extent the same can be said for friends of friends, friends of friends of friends, and so on. However, friendship alone is not enough, as users may have very different friend groups (e.g. family, soccer friends, college friends, ...). This brings us to the second social factor: shared interests. Friends that share the same interests are more likely to use the same tags.

We propose to use *user-centered knowledge* (UK), consisting of tag data that is grouped by both friendship relations and user interests. That is, when an association-based tag recommendation algorithm is invoked for a query I and user u, a source database D should be used that consists of tag data 1) from all users within the nth-degree of u, 2) that concerns the same high-level interest as query I.

Although 'high-level interest' can be interpreted in many different ways, we will not go into the details here. In the experiment section we will give a proof-of-concept based on queries, but other ways to cluster interests or identify topics could be explored as well.

Social Batched Personomies

We already briefly mentioned the personomy, being the historic tag data assigned by an individual user. To avoid the problem that no (or very little) historic data is available for many users, commonly known as the *cold-start* problem, one could consider exploiting the social graph by adding historic data

from a user's friends. Unfortunately, this only resolves part of the problem, as the approach still needs to process each query/user individually. In an online environment, this may be undesirable or even infeasible. Also, this clearly does not help for new users, who do not have friends yet.

Hence, we introduce the concept of *batched personomies*, which is applicable in very large scale systems where 10s to 100s or even 1000s of queries per second are given. Instead of processing each query individually, we propose to process them in *batches*, as follows.

1. The system does not process each individual query, but waits until either a maximum number of queries or a maximum waiting time is reached.
2. The cached queries form a batch B.
3. For a batch B, construct a database D_B that contains previous tag assignments done by all users that have a query in B.
4. If desired, add to D_B historic tag data up to the nth-degree friends for each individual user.
5. Recommend tags for each query I in B using D_B as source database.

It is clear that such an approach is only possible when many queries come in at the same time, but considering the scale of current online social networks this is completely realistic. Given the right circumstances, the approach has quite some potential advantages.

First of all, by batching queries associations have to be mined only once for each batch. This is much better than having to mine associations for each query, as is the case with personomies. Also, the source databases will be much smaller than when (all) collective knowledge is used, which may again result in faster recommendation. Second, recommendation quality is likely to be better, on average. Better than with personomies, because problems such as lack of historic data are smoothed out by forming batches. Better than with collective knowledge, because recommendation is more personalised.

The main difference with the user-centered knowledge approach is that groups are formed on-the-fly, rather than beforehand. Also, social information can be used by enabling step 4 in the above procedure, but this is not required. When this step is conducted, we refer to this approach as *social batched personomies*. How this method performs relative to other approaches can only be determined with an empirical evaluation.

6.5 Experiments

In this section we evaluate the different source database set-ups in combination with PAR, NAR and FAR, on data crawled from an online collaborative tagging service.

Evaluation Criteria We assess recommendation quality in mostly the same way as we did in the previous chapter, i.e. using train/test data and by splitting each test tagset into a query and a validation set. The evaluation measures also remain the same: we use precision (P@x), success (S@y), mean reciprocal rank (MRR) and runtime. S@y denotes the success of the y highest ranked tags. For more details we refer to Section 5.4.

Datasets We initially crawled 2 'user-centered' datasets from the Flickr photo sharing platform using the Flickr API[1]. To ensure that each dataset satisfies the requirements of user-centered knowledge, we did the following. First, we picked a query for each of the two datasets, viz. *'london'* and *'paris'*. Second, we randomly selected a user from the results of these queries. This gave us one user with 'interest' London, and one with interest Paris. From these two seed users, we did breadth-first search to crawl tag data for all users within 4 degrees (or hops), constrained to tag data that contains the query. By strictly enforcing these queries, we obtain a shared high-level interest among the crawled groups of friends. The datasets were crawled from photos added between January 31, 2010 and June 6, 2012.

To clean the data, some pre-processing was needed. We first deleted the respective queries, i.e. the tag 'london' from London, as well as 'paris' from Paris. We further deleted tags containing camera brands, photography terms, days, month, year, number and copyright. We only keep a single instance of each transaction per user, enforcing uniqueness per user. We then filtered the data by keeping only users that have at least 10 transactions with more than 7 tags. However, for users that meet this requirement, we allow all transactions consisting of at least 2 tags. From the quarter million users that we started with, after pre-processing we ended up with 1571 users, over 109680 photos, and a total of $|\mathcal{T}| = 58930$ different tags. London contains tag data for 54620 photos, Paris for 55060 photos. The largest tagset contains 73 tags.

Finally, we combined the two datasets London and Paris into a third dataset that we will refer to as Combined. Table 6.1 shows some basic properties of the three datasets.

PairAR, NaiveAR and FastAR In all cases, we use the top $m = 50$ most frequent co-occurring tags. For NAR, to obtain tagset collection $\mathcal{F}$ we mine all closed frequent tagsets with $minsup = 0.0007$ and $maxlen = 3$. We use a uniform $maxlen$ for simplicity. For FAR, we use a code table CT induced

[1] `http://www.flickr.com/services/api/`

Table 6.1: Dataset properties. '$|t|$ (%)' denotes the percentage of transactions that have < 6, $6 - 8$, resp. > 8 tags. '#friends' denotes the number of users that have X friends.

Dataset	$\|\mathcal{U}\|$	$\|\mathcal{T}\|$	$\|t\|$ (%)			#friends					
			<6	6-8	>8	0	1-2	3-10	11-50	51-250	≥251
London	749	40106	21	24	55	655	0	0	2	30	62
Paris	874	33172	8	28	64	626	0	0	8	74	166
Combined	1571	58930	14	26	60	1243	0	0	10	102	216

by KRIMP from closed itemsets with $minsup = 0.00007$ and $maxlen = 3$.

Implementation The base tag recommender algorithms PAR, NAR and FAR were implemented in C++. The social source database variants were constructed in Python, calling the base algorithms for recommendation. Experiments were run on a machine with an Intel Core 2, 2.66 GHz CPU, and 1.99 GB of RAM running Windows XP.

Results

The first series of experiments concerns the comparison of the user-centered and collective knowledge approaches. Table 6.2 shows the results obtained on the Combined Flickr dataset. 5-fold cross-validation was used to create the train/test datasets. Each tagset in the Combined dataset is used once for testing and results are averaged over all these results, but which data is used as source database D depends on the specific approach. In the case of User-centered Knowledge (UK), training data is restricted to either the London or Paris data, depending on the origin of the test tagset. This simulates using tag data that is within the nth degree of a user corresponding to the same interest.

For $k = 1$, the results of NAR and FAR are equivalent to that of PAR, hence we only present the latter. The first –and most important– observation is that independent of the used method, the UK approach *always* provides better results than CK. This implies that exploiting social and interest information can contribute to higher quality association-based recommendation.

The second observation is that given a k and D, the three methods, PAR, NAR and FAR, perform similar to what we have seen before. That is, NAR gives high precisions while the runtimes are 2.25 to 20.71 times slower than PAR. FAR achieves similar albeit slightly less accurate recommendation, but is as about as fast as PAR. Again, we conclude that FAR gives a favourable trade-off between runtime and recommendation quality. These experiments show that this is also the case in the context of social tag recommendation, on a

Table 6.2: Recommendation performance: user-centered versus collective knowledge (UK resp. CK), on the Combined dataset. For each query size k, method, and source database D, the obtained precisions, successes, MRR and runtime are given. Runtime is averaged per query, in milliseconds. For NAR and FAR, the number of tagsets used for recommendation is given.

k	Method	D	#tagsets	P@1	P@3	P@5	S@3	S@5	MRR	time
1	PAR	UK		52.93	41.71	35.54	67.13	72.13	60.46	426
	PAR	CK		47.35	38.14	32.87	62.98	69.39	55.76	568
2	PAR	UK		59.77	50.66	44.18	75.72	80.18	69.22	314
	PAR	CK		55.83	46.93	41.00	72.24	77.60	65.79	527
	NAR	UK	250822	60.73	51.39	44.84	76.05	80.42	68.67	962
	NAR	CK	194516	56.75	47.61	41.59	72.55	77.85	65.04	881
	FAR	UK	10036	60.06	50.87	44.33	76.51	80.88	68.53	305
	FAR	CK	15636	56.45	47.25	41.24	73.05	78.34	65.13	514
3	PAR	UK		61.38	54.75	48.74	79.57	83.95	71.76	288
	PAR	CK		58.89	51.07	45.40	76.35	81.48	69.19	589
	NAR	UK	250822	62.92	55.33	49.19	79.57	83.86	71.51	4040
	NAR	CK	194516	59.91	51.55	45.78	76.50	81.49	68.53	2169
	FAR	UK	10036	62.20	54.76	48.81	80.67	84.93	71.60	258
	FAR	CK	15636	59.92	51.42	45.68	77.73	82.67	69.09	463

different type of tag data.

In the remaining experiments, an alternative method was used to generate the train/test data. The reason for this is that 5-fold cross-validation gives test sets that are too small for the more personalised approaches that we will evaluate now. Consequently, the results in Table 6.2 can *not* be compared to those in Tables 6.3, 6.4, and 6.5. In the remaining experiments, train/test generation was done as follows: for each of 5 'folds', randomly select 40% of the Combined dataset as test data, and use the remainder for training. All results are averaged over all 5 'folds'.

In the second series of experiments, we evaluate the batched personomies approach that we proposed. For each fold, we treat the entire test set as a single batch. For each of the users that have a query in the test set, all corresponding data from the training set is selected. This gives us the basic 'non-social' batched source database used for recommendation, which we denote by D^u_{batched}. Then, we construct social variants by including up to nth-degree friends for each user u that has a query in the batch, denoted D^n_{batched}, for $n = \{1, 2, 3\}$. For example, D^1_{batched} is equal to D^u_{batched} augmented with

Table 6.3: Batched personomy results on the Combined dataset. We used FAR for all experiments, except for those that indicate '+ NAR'. D^x_{batched} indicates a batched set-up (see text for more details), UK/CK and other terms as before. The last two columns show the average percentage of transactions that were selected as source data, relative to either UK or CK.

k	D	#tagsets	P@1	P@3	P@5	MRR	time	$\frac{\lvert D\rvert}{\lvert UK\rvert}$%	$\frac{\lvert D\rvert}{\lvert CK\rvert}$%
1	D^u_{batched}	10097	36.45	26.22	20.92	43.93	77	23	12
	D^1_{batched}	8674	37.61	26.87	21.32	45.34	154	54	27
	D^2_{batched}	10138	39.08	28.16	22.38	47.09	251	85	43
	D^3_{batched}	11494	39.27	28.31	22.48	47.30	234	99	49
	UK	11085	39.14	28.26	22.39	47.20	256	100	50
	CK	15362	35.16	26.76	21.91	44.21	622	200	100
	UK + NAR	225060	40.39	29.09	23.08	48.44	465	100	50
	CK + NAR	181116	35.16	26.76	21.91	44.21	302	200	100
2	D^u_{batched}	10097	42.02	31.38	25.65	50.93	75	23	12
	D^1_{batched}	8674	42.78	31.96	25.92	51.90	183	54	27
	D^2_{batched}	10138	44.04	33.37	27.2	53.50	187	85	43
	D^3_{batched}	11494	44.15	33.46	27.33	53.64	268	99	49
	UK	11085	44.16	33.43	27.29	53.62	238	100	50
	CK	15362	40.68	31.83	26.6	50.90	637	200	100
	UK + NAR	225060	45.69	34.47	28.15	54.89	1201	100	50
	CK + NAR	181116	41.24	32.02	26.71	51.15	451	200	100
3	D^u_{batched}	10097	43.81	33.13	27.63	53.12	71	23	12
	D^1_{batched}	8674	44.14	33.65	27.96	53.84	138	54	27
	D^2_{batched}	10138	45.67	35.24	29.35	55.61	179	85	43
	D^3_{batched}	11494	45.63	35.32	29.45	55.68	265	99	49
	UK	11085	45.64	35.29	29.41	55.68	262	100	50
	CK	15362	42.44	33.87	28.79	53.25	661	200	100
	UK + NAR	225060	47	36.45	30.28	56.76	5796	100	50
	CK + NAR	181116	42.96	34.07	28.84	53.39	906	200	100

tag data supplied by the direct friends of all users in the batch. When D is empty after construction as just described, the method falls back to using the collective knowledge, to ensure that a recommendation can be given. FAR is used as recommendation algorithm (except when indicated otherwise), because it performed well in the previous experiment series. For reference, we also provide results obtained on UK and CK with NAR.

We can make several interesting observations from the results, which are presented in Table 6.3. First of all, the basic (non-social) batched personomy,

Table 6.4: Batched personomy results for London with $k = 3$ and FAR. Columns as in Table 6.3.

D	#tagsets	P@1	P@3	P@5	MRR	time	$\frac{\lvert D\rvert}{\lvert UK\rvert}\%$	$\frac{\lvert D\rvert}{\lvert CK\rvert}\%$
D^u_{batched}	10893	34.98	27.47	23.19	44.52	85	22.65	11.28
D^1_{batched}	8368	34.71	27.12	22.82	44.31	123	34.49	17.17
D^2_{batched}	9527	37.53	29.91	25.27	47.45	181	71.02	35.37
D^3_{batched}	12138	37.54	30.07	25.46	47.63	316	97.75	48.68
UK	11328	37.51	30.02	25.39	47.59	265	100	49.8
CK	15362	33.45	28.5	25.18	44.79	609	200.81	100

D^u_{batched}, consistently results in a higher P@1 than the collective knowledge approach. The opposite is true for P@3 and P@5, indicating that a personalised approach improves recommendation of very specific tags, but this may come at the cost of diversity. The second observation, however, is that exploiting social information can compensate for this: using user-centered knowledge always give the best performance in terms of precision. The third observation is that we can trade-off runtime with precision by using social batched personomies, with either only direct friends, or friends and friends-of-friends. In the two rightmost columns, we can see how large a percentage of the data is used on average for a batch, and we can see that using less of the data leads to faster runtimes and slightly lower precisions.

Table 6.4 shows the results for a subset of the same experiments in more detail, i.e. only for the London data with $k = 3$. We can immediately see that the London dataset is much tougher than the Paris one, as the datasets are more or less of the same size and the averages in Table 6.3 are much higher. The relative differences between the different approaches are pretty much the same though. The reason that we provide these results is that they allow for comparison to the next experiment.

Given that our batched personomies are strongly inspired by personomies, the question that naturally arises is how these two approaches compare. To assess this, we performed an experiment in which queries were processed individually. Because this is computationally rather intensive, we only did this for a single setting: for dataset London, with $k = 3$ and NAR. The results are presented in Table 6.5. For D^u, only training data provided by the user giving u is used, unless u has no such history. In that case, the complete training set is used to ensure that a recommendation can be given. D^1, D^2, and D^3 are social personomies, completely analogue to their batched counterparts.

The most important observation to be made concerns the total number of tagsets that have been mined to be able to provide the requested recommen-

Table 6.5: Personomy results for London with $k = 3$ and NAR. Again, columns as in Table 6.3.

D	#tagsets	P@1	P@3	P@5	MRR	time	$\frac{\lvert D\rvert}{\lvert UK\rvert}\%$	$\frac{\lvert D\rvert}{\lvert CK\rvert}\%$
D^u	22787756	43.58	38.02	34.23	48.53	30102	27.6	13.75
D^1	25817746	42.98	37.25	33.51	47.84	30092	26.51	13.2
D^2	27347812	42.84	37.17	33.47	47.75	31176	26.94	13.42
D^3	26304926	43.06	37.25	33.52	47.96	30965	26.5	13.2
UK	204114	39.3	31.43	26.49	49.12	2142	100	49.80
CK	181116	33.69	28.85	25.32	44.83	922	200.81	100

dations. In this query-by-query set-up, the number of mined tagsets by large exceeds all numbers we have seen so far. As a logical side-effect, the average runtime per query also increases, to about 30 seconds per query. This is way too long for application in an online environment. On the upside, this non-batched personomy approach leads to much better recommendation than its batched counterpart, as can be observed by comparing their obtained precisions in Tables 6.4 resp. 6.5.

Analysis

Whereas friendship relations between users in a social network are readily available and therefore straightforward to exploit, this is on itself not enough to guarantee high-quality recommendation [58]. We argued that friendships should always be considered in perspective of shared interests. That is, friendship relations are only useful when they are driven by a shared interest that matches the current query. This lead us to propose the User-centered Knowledge approach, which combines these two requirements.

Translating the high-level requirements for such a user-centered knowledge approach to a principled methodology is, unfortunately, far from trivial. Using tag data for friends up to the nth degree is easy, but ensuring that only friends corresponding to the current 'interest' are selected is not. We leave this to future work. Instead, we provided a proof-of-concept by simulating two user-centered datasets by crawling datasets based on queries and friendship relations. With the uprising of community detection methods that take user interests into account, more principled and practical solutions will probably be available in the near future. This is promising, because the experiments clearly showed that taking friendship and interest information into account leads to better recommendation. Both in terms of recommendation quality and runtime.

As for the personomy approaches, the query-by-query approach is clearly infeasible for association-based recommendation. Recommendation accuracy is high and the method might be useful for offline situations, but runtimes are extremely high. Even if we would optimise all algorithms that we used, the difference between the CK/UK/batched personomy approach and the personomy approach would remain large. The social batched personomy approach appears to be a viable alternative. The friendship degree can be varied to trade-off runtime with recommendation quality. Further experiments are needed to investigate how this approach performs in the more general case, when the data is not composed of two clearly coherent parts.

The performance of the three base methods, PAR, NAR, and FAR, has proven consistent in the social context. As before, FAR provides a desirable trade-off between runtime and recommendation quality, and remains our algorithm of choice when it comes to association-based tag recommendation. It is flexible enough to take different source databases as input and still provide good recommendations. It therefore seems worth the effort to further explore this direction in the future.

6.6 Conclusions

We proposed to improve tag recommendation in social networks by exploiting the information that is available in the social graph. Many of the existing recommender systems do not take this social information into account.

The two approaches that we proposed take a more social view on tag recommendation. Instead of modifying the base algorithms, we varied the data that is used as association source. We dubbed the first social variant User-centered Knowledge, to contrast Collective Knowledge. It aims to group historic tag data according to friend relationships and interests. The second variant is dubbed Social Batched Personomies and processes queries in batches instead of individually.

The empirical evaluation on data crawled from Flickr showed that, as we hypothesised, tag recommendation can be improved by taking social information into account. For future work, it would be interesting to investigate how user-centered knowledge can be constructed in online environments. Whether to do this online or offline is also an important open question. We believe that community detection methods will play an important role in further enhancing and improving social tag recommendation.

Chapter 7

Conclusions

To answer the question what we achieved in this thesis, we first recapitulate both its research problem as well as its motivation.

One of the most important features – if not the defining feature – of any database system is that it supports queries. If one wants to know information that is stored in the database, one doesn't have to search and combine stored information manually. Rather a, usually, succinct formulated query will deliver you exactly the information you need.

Next to querying, one of the prime uses of databases, and *the* prime use of data warehouses, is data mining: searching for patterns in and models of (part of) the data. The so-called data explosion means that we do not only get more and more databases, but each of these databases also gets bigger and bigger. Hence, it becomes more and more difficult to get insight in the data by means of queries only. That is were data mining steps in: the models and patterns provide insight that is hard to get with queries.

Given that both querying and data mining are important usages of a database, it is natural to wonder about their interaction. Part of that interaction is the topic of this thesis. More precisely, the research problem it addresses is as follows.

> What does a model of the data tell you about the results of a query on the data?

That is, we have a model M of – or a set of patterns on – a database D and now we compute a query Q on D, what do we know about $Q(D)$ a priori, i.e., before (or even without) computing $Q(D)$.

To make this problem more specific it is split into three sub-problems as follows:

1. Does having a model of a database help you in building a model of the result of a query on that database?

2. Can you use a model of a database to compute the answer to a descriptive query?

3. Can you use a model of a database to compute the answer to a predictive query?

To make these problems even more concrete, each of these is researched with the code tables Krimp produces as models.

Before we formulate our general conclusions with regard to our research question, we first discuss the three sub-problems and the results we achieved for each of them.

Modelling a Query

The first (sub-)problem we considered in this thesis was:

> Does having a model of a database help you in building a model of the result of a query on that database?

Clearly, this is still a vague question. Slightly more concrete, let M_{DB} be the model we induced from database DB and let Q be a query on DB. Does knowing M_{DB} help in inducing a model M_Q on $Q(DB)$. But even this is vague: what do we mean by help? In Chapter 3, we formalized "help" in two different ways.

Firstly, in the sense that we can compute M_Q directly from M_{DB} *without* consulting either DB or $Q(DB)$. Secondly we interpreted "help", far less ambitiously, as meaning "speeding-up" the computation of M_Q. That is, let $\mathcal{A}lg$ be the algorithm used to induce M_{DB} from DB, i.e., $\mathcal{A}lg(DB) = M_{DB}$. We want to transform $\mathcal{A}lg$ into an algorithm $\mathcal{A}lg^*$, which takes M_{DB} as extra input such that

$$\mathcal{A}lg^*(Q(DB), M_{DB}) \approx \mathcal{A}lg(Q(DB))$$

We investigated both approaches for itemset mining. It turned out that the first approach is not applicable to frequent itemset mining. While the second one produced good experimental results for our Krimp algorithm.

The Krimp* algorithm is simply Krimp itself, but with a reduced candidate set: only the elements of the code table induced from the complete database are considered. In other words, we basically do only a recount of the relative usages of these itemsets on $Q(D)$.

Answering Descriptive Queries

The second (sub-)problem we considered in this thesis was:

> Can you use a model of a database to compute the answer to a descriptive query?

The term *descriptive* query was used to emphasize that we want to generate the answer to a query on a database. It is not about *predictions*. In Chapter 4, we researched this problem using KRIMP on categorical relational databases.

The question is then, informally, can we use CT – computed by KRIMP on D – to generate answers to an arbitrary query Q on D? That is, rather than accessing D to compute $Q(D)$ we want to generate an answer: we want to generate a dataset $A(Q|CT)$ such that

$$A(Q|CT) \approx Q(D).$$

Clearly there is a solution to this problem. Use CT to generate a dataset D' and compute $Q(D')$. But that is not a very satisfying solution, we want to generate $A(Q|CT)$ directly from CT.

The solution in Chapter 4 is a very simple adaptation of a general method to generate transactions from a given code table CT. The adaptation is basically a sanity check: will the tuple generated adhere to the schema prescribed by the query Q. If so, it is part of the answer, if not it is not.

The experiments showed that the generated answers are probably mostly the same as the results from applying the query to the complete dataset. This result can be paraphrased as: a code table is a good summary of the dataset; especially when combined with the results of Chapter 3. Anything you want to know about the data, you can learn from its code table.

Answering Predictive Queries

The third and final (sub-)problem we considered in this thesis was:

> Can you use a model of a database to compute the answer to a predictive query?

In contrast to the previous sub-problem, here we wanted to make predictions. That is, we are querying the underlying data distribution rather than the database itself. This problem is researched in Chapters 5 and 6. Both in the context of *tag recommendation*.

Collaborative tagging services allow users to freely assign tags to resources. As the large majority of users enters only very few tags, good tag recommendation can vastly improve the usability of tags for techniques such as searching,

indexing, and clustering. Previous research has shown that accurate recommendation can be achieved by using conditional probabilities computed from tag associations. The main problem, however, is that enormous amounts of associations are needed for optimal recommendation.

To overcome this problem, we used the strengths of pattern selection in Chapter 5. Using Krimp as 'off-the-shelf' pattern selection method, we have demonstrated that our FastAR method gives a very favourable trade-off between runtime and recommendation quality. Because pattern selection is done offline and results in small pattern sets, online recommendation is fast.

FastAR uses Krimp, an existing pattern selection technique, to pick a small set of tagsets. However, the used coding scheme is not specifically designed for selecting associations that are useful for recommendation. Despite this, the presented results are encouraging. By modifying the encoding to better reflect the needs of tag recommendation, e.g. by allowing overlapping tagsets, we believe that the results could be further improved.

Social networking sites such as Flickr and Facebook allow users to share content with family, friends, and interest groups. That is, next to the tag sets we exploited in Chapter 5, there is also a social network. Not taking this information into account probably does not result in optimal recommendations. That is the issue we researched in Chapter 6.

To address the problem, we proposed two approaches that take a more social view on tag recommendation. As we did not want to change a method that works well, instead of modifying the algorithm we vary the data that is used as source of associations. We dubbed the first social variant User-centered Knowledge, to contrast Collective Knowledge. It improves tag recommendation by grouping historic tag data according to friend relationships and interests. The second variant is dubbed 'social batched personomies' and attempts to address both quality and scalability issues by processing queries in batches instead of individually, such as done in a regular personomy approach.

The empirical evaluation on data crawled from Flickr showed that we can improve tag recommendation by taking social information into account. Furthermore, the experiments provided an additional, independent confirmation of the conclusions that we drew in Chapter 5: pattern selection allows for fast, scalable, but accurate tag recommendation.

7.1 Overall Conclusion

The overall research question we posed at the beginning of this thesis was:

> What does a model of the data tell you about the results of a query on the data?

In the course of our research and in this thesis we have seen that if the model is a code table as computed by KRIMP, the answer to this question is: a lot!

In slightly more detail, we have seen that, given a code table CT computed by KRIMP on a dataset D, we can:

- compute a code table CT_Q on the result of a query Q on D very efficiently;
- we can also generate an answer to Q on D from CT without having access to D. This generated answer is almost always approximately the correct answer;
- do state-of-the-art, fast, scalable, and accurate tag recommendation for collaborative tagging. Moreover, if we also have a social network available, our recommendations only become better.

Note that the first two items relate to the database itself, while the third item relates to its underlying data distribution.

In all cases, the answers are approximately correct, most of the time. But this shouldn't come as a surprise. After all, the database is a sample from some larger, potentially infinite, population. Hence, one cannot expect models to be absolutely correct.

If one would want to summarize this thesis in just one sentence, it would be:

> A code table is a good summary of the dataset; anything you want to know about the data, you can learn from its code table.

It is also an apt conclusion.

Bibliography

[1] R. Agrawal, T. Imielinksi, and A. Swami. Mining association rules between sets of items in large databases. In *Proc of the SIGMOD'93*, pages 207–216. ACM, 1993.

[2] 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.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc of the 1994 International Conference on Very Large Databases (VLDB)*, pages 487–499. Morgan Kaufmann, 1994.

[4] P. Allison. *Missing Data - Quantitative Applications in the Social Science.* Sage Publishing, 2001.

[5] A. Asperti and G. Longo. *Categories, Types, and Structures.* MIT Press, 1991.

[6] R.J. Bayardo, Jr. Efficiently mining long patterns from databases. In *Proc of the SIGMOD'98*, pages 85–93, 1998.

[7] G. Begelman, P. Keller, and F. Smadja. Automated tag clustering: Improving search and exploration in the tag space. In *Proc of the WWW'06*, 2006.

[8] P.J. Bickel, E.A. Hammel, and J.W. O'Connell. Sex bias in graduate admissions: Data from berkeley. *Science*, 187(4175):398–404, 1975.

[9] T. Calders and B. Goethals. Mining all non-derivable frequent itemsets. In *Proc of the ECML PKDD'02*, pages 74–85, 2002.

[10] R. Cilibrasi. *Statistical Inference Through Data Compression.* PhD thesis, Universiteit van Amsterdam, 2007.

[11] R. Cilibrasi and P. Vitanyi. The google similarity distance (automatic meaning discovery using google). In *IEEE Transactions on Knowledge and Data Engineering*, volume 19, pages 370–383. IEEE, 2007.

[12] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[13] 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.

[14] T.M. Cover and J.A. Thomas. *Elements of Information Theory, 2nd ed.* John Wiley and Sons, 2006.

[15] L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36, 1999.

[16] A.P. Dempster, N.M. Laird, and D.B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society, series B*, 39(1):1–38, 1977.

[17] 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.

[18] U.M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery: An overview. *AI Magazine*, pages 1–34, 1996.

[19] N. Friedman. Learning bayesian networks in the presence of missing values and hidden variables. In *Proc. of ICML*, pages 125–133, 1997.

[20] B. Goethals. *Frequent Set Mining*, chapter 17, pages 377–397. Springer, 2005.

[21] B. Goethals, W. Le Page, and M. Mampaey. Mining interesting sets and rules in relational databases. In *Proc. of the SAC '10*. ACM, 2010.

[22] B. Goethals, W. Le Page, and H. Mannila. Mining association rules of simple conjunctive queries. In *Proc. of the SDM '08*, pages 96–107. SIAM, 2008.

[23] P.D. Grünwald. Minimum description length tutorial. In P. D. Grünwald and I. J. Myung, editors, *Advances in Minimum Description Length*. MIT Press, 2005.

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

[25] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: Current status and future directions. *Data Mining and Knowledge Discovery*, 15:55–86, 2007.

[26] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2001.

[27] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2000.

[28] D. Hand, N. Adams, and R. Bolton, editors. *Pattern Detection and Discovery*. Springer-Verlag, 2002.

[29] P. Heymann, D. Ramage, and H. Garcia-Molina. Social tag prediction. In *Proc of the SIGIR'08*, pages 531–538, 2008.

[30] M. Holsheimer, M. Kersten, H. Mannila, and H. Toivonen. A perspective on databases and data mining. In *Proc of the First International Conference on Knowledge Discovery and Data Mining*, pages 150–155. AAAI Press, 1995.

[31] A. Hotho, R. Jäschke, C. Schmitz, and G. Stumme. Information retrieval in folksonomies: Search and ranking. In *Proc of the ESWC'06*, pages 411–426, 2006.

[32] J. Hu, B. Wang, Y. Liu, and D-Y. Li. Personalized tag recommendation using social influence. In *J. of Computer Science and Technology*, volume 27, pages 527–540, 2012.

[33] T. Imielinski and H. Mannila. A database perspective on knowledge discovery. *Communications of the ACM*, 39(11):58–64, 1996.

[34] R. Jäschke, L.B. Marinho, A. Hotho, L. Schmidt-Thieme, and G. Stumme. Tag recommendations in folksonomies. In *Proc of the PKDD'07*, pages 506–514, 2007.

[35] A.J. Knobbe, M. de Haas, and A. Siebes. Propositionalisation and aggregates. In *Proc. of the PKDD '01*, pages 277–288. Springer, 2001.

[36] A.J. Knobbe, A. Siebes, and B. Marseille. Involving aggregate functions in multi-relational search. In *Proc. of the PKDD '02*, pages 287–298. Springer, 2002.

[37] A. Koopman. *Characteristic Relational Patterns*. PhD thesis, Universiteit Utrecht, the Netherlands, 2010.

[38] A. Koopman and A. Siebes. Discovering relational items sets efficiently. In M. Zaki and K. Wang, editors, *Proc of the SDM'08*, pages 108–119. SIAM, 2008.

[39] M.A. Krogel and S. Wrobel. Feature selection for propositionalization. In *Proc. of the DS '02*, pages 430–434. Springer, 2002.

[40] E. Law, B. Settles, and T.M. Mitchell. Learning to tag from open vocabulary labels. In *Proc of the ECML/PKDD'10*, pages 211–226, 2010.

[41] M. van Leeuwen. *Patterns that Matter*. PhD thesis, Universiteit Utrecht, the Netherlands, 2010.

[42] M. van Leeuwen, F. Bonchi, B. Sigurbjörnsson, and A. Siebes. Compressing tags to find interesting media groups. In *Proc of the CIKM'09*, pages 1147–1156, 2009.

[43] M. van Leeuwen and D. Puspitaningrum. Improving tag recommendation using few associations. In *Proceedings of the 11th International Symposium on Intelligent Data Analysis (IDA 2012)*, 2012.

[44] M. van Leeuwen, J. Vreeken, and A. Siebes. Compression picks the item sets that matter. In *Proc of the ECML PKDD'06*, pages 585–592, 2006.

[45] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications (Second Edition)*. Springer-Verlag, 1997.

[46] X. Li, L. Guo, and Y.E. Zhao. Tag-based social interest discovery. In *Proc of the WWW'08*, pages 675–684, 2008.

[47] M. Lipczak. Tag recommendation for folksonomies oriented towards individual users. In *ECML PKDD Discovery Challenge*, 2008.

[48] M. Lipczak and E.E. Milios. Learning in efficient tag recommendation. In *Proc of the RecSys'10*, pages 167–174, 2010.

[49] H. Mannila and H. Toivonen. Multiple uses of frequent sets and condensed representations. In *Proc of the KDD'96*, pages 189–194, 1996.

[50] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, pages 241–258, 1997.

[51] L.B. Marinho, A. Hotho, R. Jäschke, A. Nanopoulos, S. Rendle, L. Schmidt-Thieme, G. Stumme, and P. Symeonidis. *Recommender Systems for Social Tagging Systems*. Springer, February 2012.

[52] G.V. Menezes, J.M. Almeida, F. Belém, M. A. Gonçalves, A. Lacerda, E.S. de Moura, G.L. Pappa, A. Veloso, and N. Ziviani. Demand-driven tag recommendation. In *ECML/PKDD (2)*, pages 402–417, 2010.

[53] F. Meyer. *Recommender Systems in Industrial Contexts*. PhD thesis, Université de Grenoble, France, 2012.

[54] K. Morik, J-F. Boulicaut, and A. Siebes, editors. *Local Pattern Detection*. Springer-Verlag, 2005.

[55] R.T. Ng, L.V.S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. ACM SIGMOD conference*, pages 13–24, 1998.

[56] S. Nijssen and J.N. Kok. Faster association rules for multiple relations. In *Proc. of the IJCAI '01*, pages 891–896. Morgan Kaufmann, 2001.

[57] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc of the ICDT'99*, pages 398–416, 1999.

[58] A. Rae, B. Sigurbjörnsson, and R. van Zwol. Improving tag recommendation using social networks. In *Proc of the RIAO'10*, pages 92–99, 2010.

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

[60] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc of the 21st VLDB Conference*, pages 432–444. Morgan Kaufmann, 1995.

[61] A. Siebes. Data mining in inductive databases. In F. Bonchi and J. Boulicaut, editors, *Knowledge Discovery in Inductive Databases, 4th International Workshop, KDID 2005, Revised Selected and Invited Papers*, volume 3933 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2005.

[62] A. Siebes and D. Puspitaningrum. Mining databases to mine queries faster. In W. L. Buntine, M. Grobelnik, D. Mladenic, and J. Shawe-Taylor, editors, *Proceedings ECML PKDD 2009, Part II*, volume 5782 of *Lecture Notes in Computer Science*, pages 382–397. Springer, 2009.

[63] A. Siebes and D. Puspitaningrum. Patterns on queries. In S. Džeroski, B. Goethals, and P. Panov, editors, *Inductive Databases and Constraint-Based Data Mining*, chapter 13, pages 311–334. Springer, 2010.

[64] A. Siebes, J. Vreeken, and M. van Leeuwen. Item sets that compress. In *Proc of the SDM'06*, pages 393–404, 2006.

[65] B. Sigurbjörnsson and R. van Zwol. Flickr tag recommendation based on collective knowledge. In *WWW*, pages 327–336, 2008.

[66] K. Smets and J. Vreeken. Slim: Directly mining descriptive patterns. In *Proc of the SDM'12*, pages 236–247, 2012.

[67] Y. Song, Z. Zhuang, H. Li, Q. Zhao, J. Li, W.-C. Lee, and C. Lee Giles. Real-time automatic tag recommendation. In *Proc of the SIGIR'08*, pages 515–522, 2008.

[68] G. Toderici, H. Aradhye, M. Pasca, L. Sbaiz, and J. Yagnik. Finding meaning on youtube: Tag recommendation and category discovery. In *CVPR*, pages 3447–3454, 2010.

[69] P.M.B. Vitányi, F.J. Balbach, R.L. Cilibrasi, and M. Li. Normalized information distance. In Frank Emmert-Streib and Matthias Dehmer, editors, *Information Theory and Statistical Learning*, pages 39 – 71. Springer, 2009.

[70] J. Vreeken. *Making Pattern Mining Useful*. PhD thesis, Universiteit Utrecht, the Netherlands, 2009.

[71] J. Vreeken and A. Siebes. Filling in the blanks - krimp minimisation for missing data. In *Proc of the ICDM'08*, pages 1067–1072, 2008.

[72] J. Vreeken, M. van Leeuwen, and A. Siebes. Preserving privacy through data generation. In *Proc of the ICDM'07*, pages 685–690, 2007.

[73] J. Vreeken, M. van Leeuwen, and A. Siebes. Krimp: mining itemsets that compress. *Data Mining and Knowledge Discovery*, 23(1):169–214, 2011.

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

[75] I. Wasito and B. Mirkin. Nearest neighbour approach in the least-squares data imputation algorithms. *Information Sciences*, 169:1–25, 2005.

[76] M.J. Zaki. Scalable algorithms for association mining. *IEEE Trans. Knowledge Data Engineering*, 12(3):372–390, 2000.

Index

Abstract

Data mining provides methods that help to acquire insight in a dataset automatically. One of its problem areas is to select a small set of useful patterns from the huge collection of patterns that can be found in a dataset. This thesis presents our results in this area. We show that such a small set of patterns, if well-chosen, allows one to answer queries on the dataset without referring to the data itself. Moreover, we show how these pattern sets enable one to build fast and scalable recommender systems.

To choose such a small set of patterns, we rely on the Minimum Description Length (MDL) principle: the best model compresses the data best. More precisely, we use the code tables that the heuristic Krimp algorithm induces from the data. Our results show that these code tables are highly characteristic of the dataset. Anything one wants to know about the data can be inferred from its code table.

In more detail, we show how such a code table can be used to compute the answer to a query on the dataset. These answers are almost always very close to the answer one gets by actually computing the query on the data itself. This similarity is verified experimentally and quantified using an asymmetric dissimilarity score which is derived from the Normalised Compression Distance.

Next we show how the code tables can be used for the – predictive – task of tag recommendation. In particular it is shown that the proposed algorithms show a good trade-off between accuracy and time-efficiency; using the full set of patterns yields only slightly better results but requires infeasible amounts of time. In a social networking context we show how to personalize – and thus improve – our tag recommendations. This is achieved by using user-centered knowledge in contrast to the collective knowledge used for the general task. For quality and scalability reasons, we use 'social batched personomies' by processing queries in batches, instead of individually, as is done in the standard personomy approach.

In each chapter we provide extensive experimental evaluation to show that our methods perform well on a variety of datasets. From these experiments one cannot but conclude that code tables are highly characteristic of the data.

Samenvatting

Data mining onderzoekt geautomatiseerde methoden om inzicht in data te krijgen. Een van haar onderzoeksgebieden is om een klein aantal representatieve patronen te kiezen uit de gigantische berg patronen die in data te vinden zijn. Dit proefschrift presenteert onze resultaten op dit gebied. Wij laten zien dat zo'n kleine verzameling, mits goed gekozen, het toestaat het antwoord op een vraag te berekenen zonder de data te raadplegen. Bovendien laten we zien hoe zulke patroonverzamelingen het mogelijk maken om goede *recommender systems* te bouwen.

Voor de keuze van patronen vertrouwen wij op het *Minimum Description Length principle*: het beste model comprimeert de data het best. Voor die compressie gebruiken we de codetabellen die het heuristische Krimp algoritme uit de data induceert. Onze resultaten laten zien dat die tabellen zeer karakteristiek zijn voor de data. Alles wat men van de data wil weten kan men uit de codetabel afleiden.

Tevens laten we zien hoe de codetabellen gebruikt kunnen worden voor de voorspellingstaak *tag-recommendation*. De algoritmen die wij voorstellen vinden een goede balans tussen nauwkeurigheid en efficiëntie; als we alle patronen gebruiken zijn de resultaten iets beter, maar duurt de berekening onwerkbaar lang. In de context van sociale netwerken kunnen wij nóg nauwkeuriger voorspellen door gebruik te maken van *user-centered knowledge* in plaats van de *collective knowledge* die we in het algemene geval gebruiken.

Elk hoofdstuk bevat een experimentele validatie van onze methoden om aan te tonen dat zij goed werken op een grote variëteit van dataverzamelingen. Uit deze experimenten kan men niet anders dan concluderen dat de codetabellen bijzonder karakteristiek zijn voor de data.

Acknowledgements

This dissertation would not have been possible without guidance, help, and support of several individuals and institutions who in one way or another contributed and extended their valuable assistance in the preparation and completion of my PhD thesis.

First and foremost, this thesis would not have been possible without the help, support and patience of my principal supervisor and promotor, Prof. dr. A.P.J.M. Siebes, not to mention his advice and unsurpassed knowledge of data mining. Thank you for your friendship, fresh research topic suggestions, and financial support during my final year at Utrecht University.

Dr. M. van Leeuwen, as my co-promotor, thank you for the good advice, support and friendship. Thank you for many technical ideas in difficult times during my research and thesis writing.

I would like to thank the committee members, prof.dr. J. van Leeuwen (UU), prof.dr.ir. L.C. van der Gaag (UU), prof.dr. J.N. Kok (LIACS), prof.dr. H. Blockeel (KU Leuven), and dr. J. Vreeken (Universiteit Antwerpen), for reading my thesis and for their helpful suggestions and discussions.

I would like to acknowledge the financial support of the Ministry of Communication and Information Technology of the Republic of Indonesia for my 4-year PhD fellowship. My gratitude goes especially to Bapak Ir. Aizirman Djusan, MSc,Econ and all of his cooperative staff at the human resource department (TPSDM) of Depkominfo.

I also would like to thank Utrecht University for academic, financial, and facility support, and also the staff, particularly Marinus Veldhorst for assigning financial support during the final year of my study, and Edith Stap. I cannot imagine how I could have finished my PhD study without funding.

Thank you to all my colleagues in the Algorithmic Data Analysis (ADA) group for a pleasurable working environment. Thanks to Arne Koopman and Jilles Vreeken for discussion during the first year. Also thanks to Wouter Duivesteijn, Roel Bertens, and Michael Mampaey, who shared the room in Buys Ballot Laboratory 5.61 with me. I am grateful also to Nicola Barile, Ad Feelders, Hans Philippi, Lennart Herlaar, Jeroen De Knijf and Arno Knobbe

for their support.

Finally, I would like to express my gratitude to all members of Annisaa-Utrecht, from whom I learned much about the philosophy of life, how to do something in *kaffah*, i.e. a totality for perfection. To my parents, my sisters, and my lovely nieces and nephews (Salsabiila, Ayu, Rasendriya, Maitsaa, and Rama), who supported and encouraged me, I dedicate this thesis to you.

Utrecht, September 2012

Curriculum Vitae

Diyah was born on the 5th of October 1976 in Semarang, Central Java, Indonesia. She obtained her BSc. in Informatics Engineering at Mandala College of Technology in Bandung, Indonesia in 1999. After this she obtained her MSc. in Computer Science with cum laude at Gadjah Mada University in 2003. Her thesis was in the area of Artificial Neural Networks, under supervision of Drs. Widodo Priyodiprojo, MSc.EE. During and after her master study, she worked in several universities as a junior lecturer.

Diyah started her Ph.D. in the Algorithmic Data Analysis group (Data Mining group) at Universiteit Utrecht, under supervision of Prof.dr. A.P.J.M. Siebes, in 2008. In 2012, she finished her Ph.D. thesis.