Mining Databases to Mine Queries Faster

Arno Siebes and Diyah Puspitaningrum

Department Of Information and Computing Sciences
Universiteit Utrecht, Utrecht, The Netherlands
{arno,diyah}@cs.uu.nl

Abstract. Inductive databases are databases in which models and patterns are first class citizens. Having models and patterns in the database raises the question: do the models and patterns that are stored help in computing new models and patterns? For example, let C be a classifier on database DB and let Q be a query. Does knowing C speed up the induction of a new classifier on the result of Q?

In this paper we answer this problem positively for the code tables induced by our Krimp algorithm. More in particular, assume we have the code tables for all tables in the database. Then we can approximate the code table induced by Krimp on the result of a query, using only these global code tables as candidates. That is, we do not have to mine for frequent item sets on the query result.

1 Introduction

The problem investigated in this paper can informally be phrased as follows. 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 $Q(DB)$, i.e., on the result of Q when applied to DB. For example, if M_{DB} is a classifier and Q selects a subset of DB, does knowing M_{DB} speed-up the induction of a new classifier M_Q on the subset $Q(DB)$?

There are at least two contexts in which this question is relevant. Firstly in the context of inductive databases. Ever since their introduction in the seminal paper by Imielinski and Mannila [11], they have been a constant theme in data mining research. There is no formal definition of an inductive database, in fact, it may be too early for such a definition [14]. However, consensus is that models and patterns should be first class citizens in such a database. That is, e.g., one should be able to query for patterns. Having models and patterns in the database naturally raises the question: do the models and patterns that are stored help in computing new models and patterns?

The second context in which the problem is relevant is in every day 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. The data often resides in a data warehouse or in multiple databases, and the mining database is constructed from these underlying databases.

W. Buntine et al. (Eds.): ECML PKDD 2009, Part II, LNAI 5782, pp. 382–397, 2009.

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 data set, 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.

So, the problem is relevant, but isn't it trivial? After all, if M_{DB} is a good model on DB, it is almost always also a good model on a random subset of DB; almost always, because a random subset may be highly untypical. 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 M_{DB} can even be highly misleading on $Q(DB)$. This is nicely illustrated by the well-known example of *Simpson's Paradox*, viz., Berkeley's admission data [2]. 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.

Solving the problem for all model classes and algorithms is a rather daunting task. Rather, in this paper we study the problem for one specific class of models, viz., the code tables induced by our Krimp algorithm [15]. Given all frequent item sets on a table, Krimp selects a small subset of these frequent item sets. The reason why we focus on Krimp is threefold. Firstly, because together the selected item sets describe the underlying data distribution of the complete database very well, see, e.g., [16,17]. Secondly, because the code table consists of *local* patterns. Such a local pattern can be seen as a selection query on the database (for the transactions in its support), hence, one would expect Krimp to do well on selection queries. Thirdly, from earlier research on Krimp in a multi-relational setting, we noticed as a side-result that Krimp is probably easily transformed for joins [13]; this is investigated further in this paper.

More in particular, we show that if we know the code tables for all tables in the database, then we can approximate the code table induced by Krimp on the result of a query, using *only* the item sets in these global code tables as candidates. Since Krimp is linear in the number of candidates and Krimp reduces the set of frequent item sets by many orders of magnitude, this means that we can now speed up the induction of code tables on query results by many orders of magnitude.

This speed-up results in a slightly less good code table, but it approximates the optimal solution within a few percent. We will formalise "approximation"

in terms of MDL [10]. Hence, the data miner has a choice: either a quick, good approximation, or the optimal result taking longer time to compute.

2 Problem Statement

This section starts with some preliminaries and assumptions. Then we introduce the problem informally. To formalise it we use MDL, which is briefly discussed.

2.1 Preliminaries and Assumptions

We assume that our data resides in relational databases. In fact, note that the union of two relational databases is, again, a relational database. Hence, we assume, without loss of generality, that our data resides in *one* relational database DB. As query language we will use the standard relational algebra. More precisely, we focus on the usual "select-project-join" queries. That is, on the selection operator σ, the projection operator π, and the (equi-)join operator $\bowtie$; see [5]. Note that, as usual in the database literature, we use *bag* semantics. That is, we do allow duplicates tuples in tables and query results.

As mentioned in the introduction, the mining database is constructed from DB using queries. Given the compositionality of the relational algebra, we may assume, again without loss of generality, that the analysis database is constructed using one query Q. That is, the analysis database is $Q(DB)$, for some relational algebra expression Q. Since DB is fixed, we will often simply write Q for $Q(DB)$; that is, we will use Q to denote both the query and its result.

2.2 The Problem Informally

In the introduction we stated that knowing a model on DB should help in inducing a model on Q. To make this more precise, let $\mathcal{A}$ be our data mining algorithm. At this point, $\mathcal{A}$ can be any algorithm, it may, e.g., compute a decision tree, all frequent item sets or a neural network.

Let $\mathcal{M}_{DB}$ denote the model induced by $\mathcal{A}$ from DB, i.e, $\mathcal{M}_{DB} = \mathcal{A}(DB)$. Similarly, let $\mathcal{M}_Q = \mathcal{A}(Q)$. We want to transform $\mathcal{A}$ into an algorithm $\mathcal{A}^*$ that takes at least two inputs, i.e, both Q and $\mathcal{M}_{DB}$, such that:

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

$$\mathcal{A}^*(Q, \mathcal{M}_{DB}) \approx \mathcal{M}_Q$$

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

The second criterion is easy to formalise: the runtime of $\mathcal{A}^*$ should be shorter than that of $\mathcal{A}$. 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?

2.3 Model Approximation

The answer to the question how to formalise that one model approximates another depends very much on the goal. If $\mathcal{A}$ 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 the Minimum Description Length principle (MDL) [10], and MDL offers another way to compare models, viz., by their *compression-rate*. 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., [8] for an overview of other applications of MDL in data mining.

2.4 Minimum Description Length

MDL embraces the slogan *Induction by Compression.* It can be roughly described as follows.

Given a set of models[1] $\mathcal{H}$, 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.

One can paraphrase this by: the smaller $L(H) + L(D|H)$, the better H models D.

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

[1] MDL-theorists tend to talk about *hypothesis* in this context, hence the $\mathcal{H}$; see [10] for the details.

Just taking the difference of the two, however, can be quite misleading. Take, e.g., two databases db_1 and db_2 sampled from the same underlying distribution, such that db_1 is far bigger than db_2. Moreover, fix a model H. Then necessarily $L(db_1|H)$ is bigger than $L(db_2|H)$. 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.

Definition 1. *Let H_1 and H_2 be two models for a dataset D. The asymmetric 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 [4]. The reason why we use this asymmetric version is that we have a "gold standard". We want to know how far our approximate result $\mathcal{A}^*(Q, \mathcal{M}_{DB})$ deviates from the optimal result $\mathcal{A}(Q)$.

Clearly, $ADM(\mathcal{A}^*(Q, \mathcal{M}_{DB}), \mathcal{A}(Q))$ does not only depend on $\mathcal{A}^*$ and on $\mathcal{A}$, 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 to strong a requirement. Some queries might result in a very untypical subset of DB, 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 DB be a database and let Q be a random query on DB. Moreover, let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two data mining algorithms on DB. $\mathcal{A}_1$ is an (ϵ, δ)-approximation of $\mathcal{A}_2$ iff*

$$P(ADM(\mathcal{A}_1(Q), \mathcal{A}_2(Q)) > \epsilon) < \delta$$

2.5 The Problem

Using the notation introduced above, we formalise the problem as follows.

Problem Statement
For a given data mining algorithm $\mathcal{A}$, devise an algorithm $\mathcal{A}^*$, such that for a random database DB:

1. $\mathcal{A}^*$ is an (ϵ, δ)-approximation of $\mathcal{A}$ for reasonable values for ϵ and δ.
2. Computing $\mathcal{A}^*(Q, \mathcal{M}_{DB})$ is faster than computing $\mathcal{A}(Q)$ for a random query Q on DB.

What reasonable values for ϵ and δ are depends very much on the application. While $\epsilon = 0.5$ for $\delta = 0.9$ might be acceptable for one application, these values may be unacceptable for others.

The ultimate solution to the problem as stated here would be an algorithm that transforms any data mining algorithm $\mathcal{A}$ in an algorithm $\mathcal{A}^*$ with the requested properties. This is a rather ambitious, ill-defined (what is the class of all data mining algorithms?), and, probably, not attainable goal. Hence, in this paper we take a more modest approach: we transform one algorithm only, our KRIMP algorithm.

3 Introducing KRIMP

For the convenience of the reader we provide a brief introduction to KRIMP in this section, it was originally introduced in [15] (although not by that name) and the reader is referred to that paper for more details.

Since KRIMP selects a small set of representative item sets from the set of all frequent item sets, we first recall the basic notions of frequent item set mining [1].

3.1 Preliminaries

Let $\mathcal{I} = \{I_1, \ldots, I_n\}$ be a set of binary (0/1 valued) attributes. That is, the domain D_i of item I_i is $\{0, 1\}$. A transaction (or tuple) over $\mathcal{I}$ is an element of $\prod_{i \in \{1,\ldots,n\}} D_i$. A database DB over $\mathcal{I}$ is a bag of tuples over $\mathcal{I}$. This bag is indexed in the sense that we can talk about the i-th transaction.

An item set J is, as usual, a subset of $\mathcal{I}$, i.e., $J \subseteq \mathcal{I}$. The item set J *occurs* in a transaction $t \in DB$ if $\forall I \in J : \pi_I(t) = 1$. The *support* of item set J in database DB is the number of transactions in DB in which J occurs. That is, $supp_{DB}(J) = |\{t \in DB |\ J \text{ occurs in } t\}|$. An item set is called *frequent* if its support is larger than some user-defined threshold called the *minimal support* or *min-sup*. Given the A Priori property,

$$\forall I, J \in \mathcal{P}(\mathcal{I}) : I \subset J \rightarrow supp_{DB}(J) \leq supp_{DB}(I)$$

frequent item sets can be mined efficiently level wise, see [1] for more details.

Note that while we restrict ourself to binary databases in the description of our problem and algorithms, there is a trivial generalisation to categorical databases. In the experiments, we use such categorical databases.

3.2 KRIMP

The key idea of the KRIMP algorithm is the code table. A code table is a two-column table that has item sets on the left-hand side and a code for each item set on its right-hand side. The item sets in the code table are ordered descending on 1) item set length and 2) support size and 3) lexicographically. The actual codes on the right-hand side are of no importance but their lengths are. To explain how these lengths are computed, the coding algorithm needs to be introduced.

A transaction t is encoded by KRIMP by searching for the first item set I in the code table for which $I \subseteq t$. The code for I becomes part of the encoding of t. If $t \setminus I \neq \emptyset$, the algorithm continues to encode $t \setminus I$. Since it is insisted that

each code table contains at least all singleton item sets, this algorithm gives a unique encoding to each (possible) transaction over $\mathcal{I}$.

The set of item sets used to encode a transaction is called its *cover*. Note that the coding algorithm implies that a cover consists of non-overlapping item sets.

The length of the code of an item in a code table CT depends on the database we want to compress; the more often a code is used, the shorter it should be. To compute this code length, we encode each transaction in the database DB. The *frequency* of an item set $I \in CT$, denoted by $freq(I)$ is the number of transactions $t \in DB$ which have I in their cover. That is,

$$freq(I) = |\{t \in DB | I \in cover(t)\}|$$

The relative frequency of $I \in CT$ is the probability that I is used to encode an arbitrary $t \in DB$, i.e.

$$P(I|DB) = \frac{freq(I)}{\sum_{J \in CT} freq(J)}$$

For optimal compression of DB, the higher P(c), the shorter its code should be. Given that we also need a prefix code for unambiguous decoding, we use the well-known optimal Shannon code [7]:

$$l(I|CT) = -\log(P(I|DB)) = -\log\left(\frac{freq(I)}{\sum_{J \in CT} freq(J)}\right)$$

The length of the encoding of a transaction is now simply the sum of the code lengths of the item sets in its cover. Therefore the encoded size of a transaction $t \in DB$ compressed using a specified code table CT is calculated as follows:

$$L(t|CT) = \sum_{I \in cover(t,CT)} l(I|CT)$$

The size of the encoded database is the sum of the sizes of the encoded transactions, but can also be computed from the frequencies of each of the elements in the code table:

$$\begin{aligned} L(DB|CT) &= \sum_{t \in DB} L(t|CT) \\ &= -\sum_{I \in CT} freq(I) \log\left(\frac{freq(I)}{\sum_{J \in CT} freq(J)}\right) \end{aligned}$$

To find the optimal code table using MDL, we need to take into account both the compressed database size, as described above, as well as the size of the code table. For the size of the code table, we only count those item sets that have a non-zero frequency. The size of the right-hand side column is obvious; it is simply the sum of all the different code lengths. For the size of the left-hand side column, note that the simplest valid code table consists only of the singleton item sets. This is the *standard encoding (ST)*, of which we use the codes to

compute the size of the item sets in the left-hand side column. Hence, the size of code table CT is given by:

$$L(CT|DB) = \sum_{I \in CT : freq(I) \neq 0} l(I|ST) + l(I|CT)$$

In [15] we defined the optimal set of (frequent) item sets as that one whose associated code table minimises the total compressed size:

$$L(CT, DB) = L(CT|DB) + L(DB|CT)$$

As before, this minimal compressed size of DB is denoted by $\mathcal{L}(DB)$. KRIMP starts with a valid code table (only the collection of singletons) and a sorted list of candidates (frequent item sets). These candidates are assumed to be sorted descending on 1) support size, 2) item set length and 3) lexicographically. Each candidate item set is considered by inserting it at the right position in CT and calculating the new total compressed size. A candidate is only kept in the code table iff the resulting total size is smaller than it was before adding the candidate. If it is kept, all other elements of CT are reconsidered to see if they still positively contribute to compression. The whole process is illustrated in Figure 1; see [15].

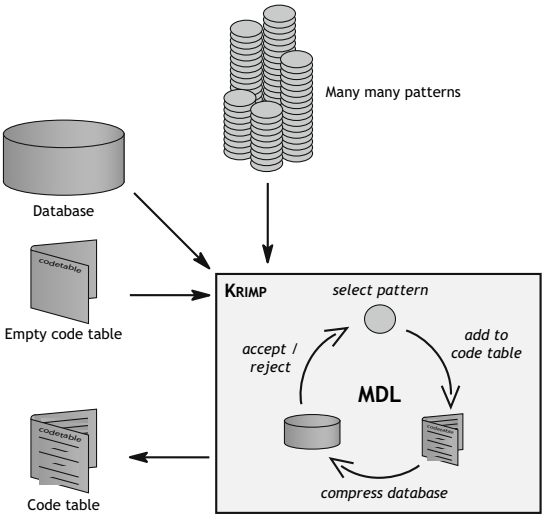

Fig. 1. KRIMP in action

4 The Problem for KRIMP

If we assume a fixed minimum support threshold for a database, KRIMP has only one essential parameter: the database. For, given the database and the

(fixed) minimum support threshold, the candidate list is also specified. Hence, we will simply write CT_{DB} and KRIMP(DB), to denote the code table induced by KRIMP from DB. Similarly CT_Q and KRIMP(Q) denote the code table induced by KRIMP from the result of applying query Q to DB.

Given that KRIMP results in a code table, there is only one sensible way in which KRIMP(DB) can be re-used to compute KRIMP(Q): provide KRIMP only with the item sets in CT_{DB} 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, e.g., KRIMP$^*(Q)$ is the code table that KRIMP induces from $Q(DB)$ using the item sets in CT_{DB} 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 DB:

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

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

  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).

Neither of these two properties can be formally proven, if only because KRIMP and thus KRIMP* are both heuristic algorithms. Rather, we report on extensive tests of these two requirements.

5 The Experiments

In this section we describe our experimental set-up. First we briefly describe the data sets we used. Next we discuss the queries used for testing. Finally we describe how the tests were performed.

5.1 The Data Sets

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 [6] data sets and one data set from the KDDcup 2004.

More in particular, we have used the data sets *connect, adult, chessBig, letRecog, PenDigits* and *mushroom* from UCI. These data sets were chosen because they are well suited for KRIMP. Some of the other data sets 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 data sets are very dense. While

KRIMP performs well on these data sets, choosing them would have turned our extensive testing prohibitively time-consuming.

Since all these data sets are single table data sets, they do not allow testing with queries involving joins. To test such queries, we used tables from the "Hepatitis Medical Analysis"[2] 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 item set data and rows with missing data have been removed.

5.2 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 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, the well-known "project-select-join" queries are among the most used queries in practice. This is witnessed by the important role they play in benchmarks for DBMSs such as the TPC family of benchmarks. Secondly, simple queries will have, in general, larger results than more complex queries.

5.3 The Experiments

The experiments preformed for each of the queries on each of the data sets were generated as follows.

Projection: The projection queries were generated by randomly choosing a set X of n attributes, 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 data set were projected in the same way. For each value of n, 10 random sets X were generated on each data set.

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 attributes, with $n \in \{1, 2, 3, 4\}$. Next for each random attribute A_i a random value v_i in its domain D_i was chosen. Finally, for each A_i in X a random $\theta_i \in \{=, \neq\}$ was chosen The generated query is thus $\sigma(\bigwedge_{A_i \in X} A_i \theta_i v_i)$. As in the previous case, we performed 10 random experiments on each of the data sets for each of the values of n.

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

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 attributes and we project away either 1 or 3 attributes. 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 data sets: 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" data set 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 item set in the code table of *bio*, and $\mathcal{I}_2$ is an item set in the code table of *hemat*. Again we select on 1 or 2 attributes and we project away either 1 or 3 attributes. And, again, we performed again 100 random experiments on the database for each of the four combinations; as above.

6 The Results

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

6.1 Projection Queries

In Figure 2 the results of the random projection queries on the *letRecog* data set 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. On the other data sets, KRIMP* performs similarly. Since this is also clear from the project-select query results, we do not provide all details here. This will become clear when we report on the project-select queries.

6.2 Selection Queries

The results of the random selection queries on the *penDigits* data set are visualised in figure 3. 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. These numbers are, however, not representative for the results on the other data sets. It turned out that *penDigits* is actually too small and sparse to

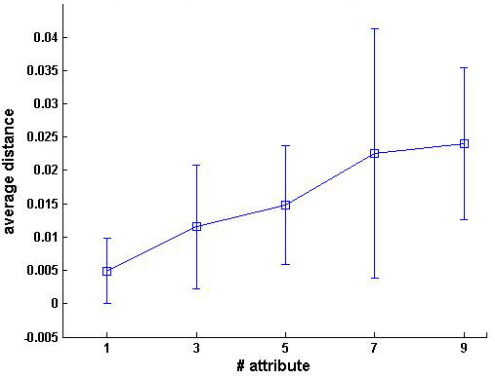

Fig. 2. Projection results on *letRecog*

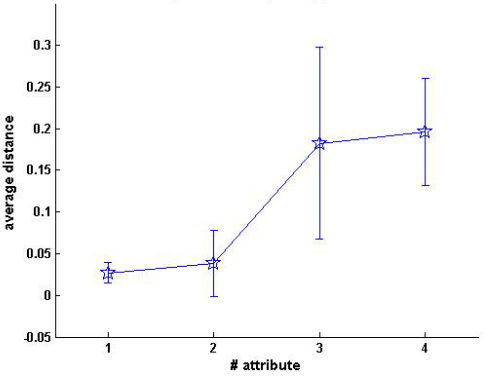

Fig. 3. Selection results on *penDigits*

test KRIMP* seriously. In the remainder of our results section, we 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 data sets KRIMP* still performs reasonably well. On all other data sets KRIMP* performs far better, as will become clear when we report on the project-select queries.

6.3 Project-Select Queries

The results of the projection-select queries are given in the table in Figure 4. 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. 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,

ADM±STD		connect	adult	chessBig	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

Fig. 4. Results of Project-Select Queries

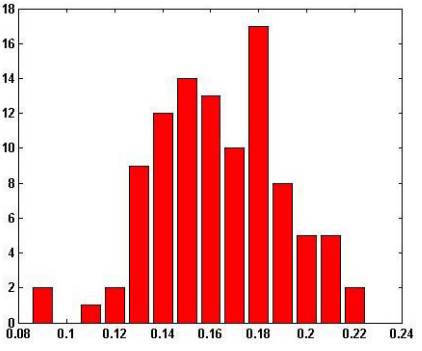

Fig. 5. Histogram of 100 Project-Select Queries on *connect*

Relative #candidates		connect	adult	chessBig	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

Fig. 6. Relative number of candidates for KRIMP*

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 2 with the "Select 1, Project out 3" and "Select 2, Project out 3" queries in Figure 4 on the same data set. 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 data sets.

To give insight in the distribution of the ADM scores of the "Select 2, Project out 3" queries on the *connect* data set are given in Figure 5. From this figure we see that if we choose $\epsilon = 0.2$, $\delta = 0.08$. In other words, KRIMP* is a 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 the table in Figure 6 This table 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 *chessBig* were the query results get

small and, thus, have few frequent item sets. The experiments are those that are reported on in Figure 4.

6.4 Select-Project-Join Queries

The results for the select-project-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.

7 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 item set 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 below 0.2. That is, a full-fledged KRIMP run compresses the data set 20% better than KRIMP*. Is that good?

In a previous paper [17], we took two random samples from data sets, 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 data sets 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 data sets are in the same ball-park. In other words, an ADM score on these data sets below 0.2 is on the level of "natural variations" of the data distribution. Hence, given that the average ADM scores are often much 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.

8 Related Work

While there are, as far as the authors know, no other papers that study the same problem, the topic of this paper falls in the broad class of data mining with background knowledge. For, the model on the database, $\mathcal{M}_{DB}$, is used as

background knowledge in computing $\mathcal{M}_Q$. While a survey of this area is beyond the scope of this paper, we point out some papers that are related to one of the two aspects we are interested in, viz., speed-up and approximation.

A popular area of research in using background knowledge is that of constraints. Rather than trying to speed up the mining, the goal is often to produce models that adhere to the background knowledge. Examples are the use of constraints in frequent pattern mining, e.g. [3], and monotonicity constraints [9]. Note, however, that for frequent pattern mining the computation can be speeded up considerably if the the constraints can be pushed into the mining algorithm [3]. So, speed-up is certainly a concern in this area. However, as far as we know approximation plays no role. The goal is still to find all patterns that satisfy the constraints.

Another use of background knowledge is to find unexpected patterns. In [12], e.g., Bayesian Networks of the data are used to estimate how surprising a frequent pattern is. In other words, the (automatically induced) background knowledge is used filter the output. In other words, speed-up is of no concern in this approach. Approximation clearly is, albeit in the opposite direction of ours: the more a pattern deviates from the global model, the more interesting it becomes. Whereas we would like that all patterns in the query result are covered by our approximate answer.

9 Conclusions

In this paper we introduce a new problem: given that we have a model induced from a database DB, does that help us in inducing a model on the result of a query Q on DB? We formalise the problem for algorithms based on MDL and solve it for a particular algorithm, viz., our Krimp algorithm. More in particular we introduce Krimp*. This is actually the same as Krimp, but it gets a restricted input. The code tables computed by Krimp on DB are used as input, and thus as background knowledge, for Krimp* on $Q(DB)$.

Extensive experiments with select-project-join queries show that Krimp* approximates the results of Krimp very well while it computes these results upto hundreds of times faster. Hence, the data analyst has a real choice: either get good result fast, or get optimal results slower.

References

1. Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., Inkeri Verkamo, A.: Fast discovery of association rules. In: Advances in Knowledge Discovery and Data Mining, pp. 307–328. AAAI, Menlo Park (1996)
2. Bickel, P.J., Hammel, E.A., O'Connell, J.W.: Sex bias in graduate admissions: Data from berkeley. Science 187(4175), 398–404 (1975)
3. Boulicaut, J.-F., Bykowski, A.: Frequent closures as a concise representation for binary data mining. In: Terano, T., Chen, A.L.P. (eds.) PAKDD 2000. LNCS, vol. 1805, pp. 62–73. Springer, Heidelberg (2000)

4. Cilibrasi, R., Vitanyi, P.: Automatic meaning discovery using google. IEEE Transactions on Knowledge and Data Engineering 19, 370–383 (2007)
5. Codd, E.F.: A relational model of data for large shared data banks. Communications of the ACM 13(6), 377–387 (1970)
6. Coenen, F.: The LUCS-KDD discretised/normalised ARM and CARM data library (2003), http://www.csc.liv.ac.uk/~frans/KDD/Software/LUCS_KDD_DN/
7. Cover, T.M., Thomas, J.A.: Elements of Information Theory, 2nd edn. John Wiley and Sons, Chichester (2006)
8. Faloutsos, C., Megalooikonomou, V.: On data mining, compression and kolmogorov complexity. In: Data Mining and Knowledge Discovery, vol. 15, pp. 3–20. Springer, Heidelberg (2007)
9. Feelders, A.J., van der Gaag, L.C.: Learning bayesian network parameters under order constraints. Int. J. Approx. Reasoning 42(1-2), 37–53 (2006)
10. Grünwald, P.D.: Minimum description length tutorial. In: Grünwald, P.D., Myung, I.J. (eds.) Advances in Minimum Description Length. MIT Press, Cambridge (2005)
11. Imielinski, T., Mannila, H.: A database perspective on knowledge discovery. Communications of the ACM 39(11), 58–64 (1996)
12. Jaroszewicz, S., Simovici, D.A.: Interestingness of frequent itemsets using bayesian networks as background knowledge. In: Proceedings KDD, pp. 178–186 (2004)
13. Koopman, A., Siebes, A.: Discovering relational item sets efficiently. In: Proceedings SDM 2008, pp. 585–592 (2008)
14. De Raedt, L.: A perspective on inductive databases. SIGKDD Explorations 4(2), 69–77 (2000)
15. Siebes, A., Vreeken, J., van Leeuwen, M.: Item sets that compress. In: Proceedings of the SIAM Conference on Data Mining, pp. 393–404 (2006)
16. van Leeuwen, M., Vreeken, J., Siebes, A.: Compression picks item sets that matter. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) PKDD 2006. LNCS (LNAI), vol. 4213, pp. 585–592. Springer, Heidelberg (2006)
17. Vreeken, J., van Leeuwen, M., Siebes, A.: Preserving privacy through data generation. In: Proceedings of the IEEE International Conference on Data Mining, pp. 685–690 (2007)