Characterising the Difference

Jilles Vreeken, Matthijs van Leeuwen & Arno Siebes

Department of Information and Computing Sciences
Utrecht University
Technical Report UU-CS-2007-014
www.cs.uu.nl
ISSN: 0924-3275

Characterising the Difference[1]

Jilles Vreeken Matthijs van Leeuwen Arno Siebes
Computer Science Department
Universiteit Utrecht, the Netherlands
+31 30 253 90 49
{jillesv, mleeuwen, arno}@cs.uu.nl

ABSTRACT

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

In previous work, we showed that the data distribution can be captured in a pattern-based model using compression [11]. Here, we extend this approach to define a generic dissimilarity measure on databases. Moreover, we show that this approach can identify those patterns that characterise the differences between two distributions.

Experimental results show that our method provides a well-founded way to independently measure database dissimilarity that allows for thorough inspection of the actual differences. This illustrates the use of our approach in real world data mining.

Categories and Subject Descriptors
H.2.8. Data Mining; I.5.4. Similarity Measures.

General Terms
Algorithms, Experimentation, Theory.

Keywords
Database dissimilarity, temporal data mining, compression.

1. INTRODUCTION

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

The problem of this kind of "difference detection" has received ample attention, both in the database and in the data mining community. In the database community, OLAP [2] is the prime example. Using roll-up and drill-down operations, a user can, manually, investigate, e.g., the difference in sales between the two stores. Emerging pattern mining [5] is a good example from the data mining community. It discovers those patterns whose support increase significantly from one database to the other.

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

In this paper we propose a new approach to "difference detection" that identifies those patterns that characterise the differences between the two databases. In fact, the approach just as easily identifies the characteristic differences between multiple databases. The approach extends our earlier work employing Minimum Description Length (MDL) for frequent pattern mining. As in our previous papers, we restrict ourselves to frequent item set mining, although the methodology easily extends to other kinds of patterns and data types, see [1].

In our first paper [11] we attacked the well-known frequent item set explosion at low support thresholds using MDL. We introduced the KRIMP algorithm that selects that subset of all frequent item sets that gives the best, lossless, compression of the database. KRIMP exploits a compression algorithm that uses a set of frequent item sets as a code table to compress a database. The set of frequent item sets which induces the best compression algorithm is the set of frequent item sets selected by KRIMP. Section 2 gives a brief overview.

The MDL philosophy is that the selected subset gives the best approximation of the underlying data distribution. In our second paper [9] we independently verified this claim by using the compression schemes for classification. Say, we have two classes, C_1 and C_2. Select the MDL-best set F_1 of frequent item sets for the sub-database for class C_1 and F_2 for class C_2. As explained above, this gives us two compression algorithms, configured by code table CT_1 based on F_1 and code table CT_2 based on F_2. A new, unseen, example t can now be compressed by both CT_1 and CT_2. In the paper we argued that the Bayes optimal choice is to assign t to the class whose compressor compresses t best. This simple classification algorithm scores on-par with state-of-the-art classification algorithms; Section 2 gives some more details.

[1] This is a pre-print version of a paper accepted to ACM SIGKDD, 12-15 August 2007, San Jose, USA.

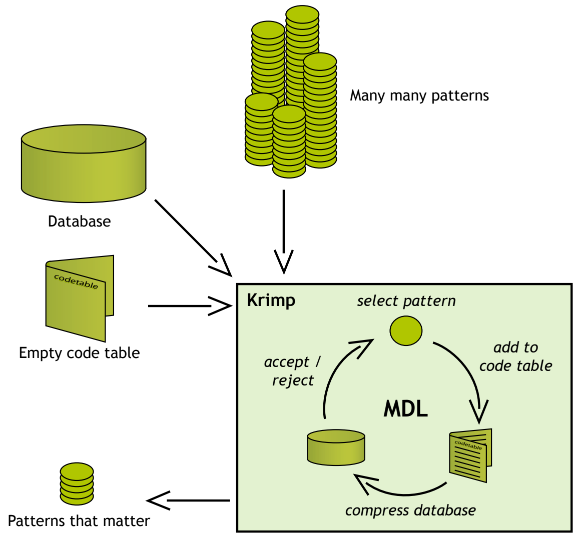


Figure 1. Finding the right code table with KRIMP. From a set of candidates, MDL picks those patterns that compress the database best.

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

Let db_1 and db_2 be the two databases, with transactions concerning the same sets of items, of which we need to analyse the differences. In Section 3, we first consider the difference in compressed length for the transactions in db_1 when compressed by the MDL-compression schemes. The MDL-principle as well as our results in classification imply that the compression scheme induced from db_2 should in general do worse than the scheme induced from db_1. This is verified by some simple experiments.

Next, in Section 3.2, we aggregate these differences per transaction by summing over all transactions in db_1 and normalising this sum by the optimal code length for db_1. This aggregation measures how different a database is from db_1. This is verified by experiments that show the correlation between this similarity measure and the confusion matrix of our classification algorithm briefly introduced above and in Section 2.

Finally, in Section 3.3, this simple measure is turned into a dissimilarity measure for any pair of databases by taking the maximum of how different db_1 is from db_2 and vice versa. Again, the MDL-principle implies that this is a dissimilarity measure. Experiments verify this claim by showing the correlation between this dissimilarity measure and the accuracy of our classification algorithm.

The result of Section 3 is a dissimilarity measure for a pair of databases, based on code tables. If the dissimilarity is small, the two databases are more or less the same and a further analysis of the differences will not show anything interesting. But how to proceed if the dissimilarity is large? That is the topic of Section 4. In that section, we introduce three ways to characterise these differences. The first approach focuses on the usage-patterns of the code table elements, while the second focuses on how (sets of) transactions are compressed by the two different schemes. The third and last approach focuses on differences in the code tables themselves. All three approaches highlight complementary, characteristic, differences between the two databases.

In Section 5 we discuss related work and describe the differences with our work. We round up with conclusions and future research in Section 6.

2. PRELIMINARIES

Foundation of all data discussed in this paper is a set of items $\mathcal{I}$, e.g., the items for sale in a shop. A transaction $t \in \mathcal{P}(\mathcal{I})$ is a set of items, e.g., representing the items a client bought at that store. A database db over $\mathcal{I}$ is a bag of transactions, e.g., the different sale transactions on a given day. An item set $I \in \mathcal{I}$ occurs in a transaction $t \in db$ iff $I \subseteq t$. The support of I in db is the number of transactions in the database in which I occurs.

2.1 Compression with Krimp

In previous work [11] we introduced the KRIMP algorithm; we will now give a quick summary. As mentioned, we restrict ourselves to item set data, although the method can be easily extended to structured data [1].

In KRIMP, a code table 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. The actual codes on the right-hand side are of no importance: their lengths are. To explain how these lengths are computed we first have to introduce the coding algorithm.

A transaction t is encoded by KRIMP by searching for the first item set c in the code table for which $c \subseteq t$. The code for c becomes part of the encoding of t. If $t \setminus c \neq \emptyset$, the algorithm continues to encode $t \setminus c$. Since we insist that each code table contains at least all singleton item sets, this algorithm gives a unique encoding to each (possible) transaction. 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 $c \in CT$ is the number of transactions $t \in db$ which have c in their cover.

The relative frequency of $c \in CT$ is the probability that c is used to encode an arbitrary $t \in db$. For optimal compression of db, the higher P(c), the shorter its code should be. In fact, from information theory [7] we have the optimal code length for c as:

$$l_{CT}(c) = -\log(P(c \mid db)) = -\log\left(\frac{freq(c)}{\sum_{d \in CT} freq(d)}\right) \qquad \textbf{(1)}$$

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:

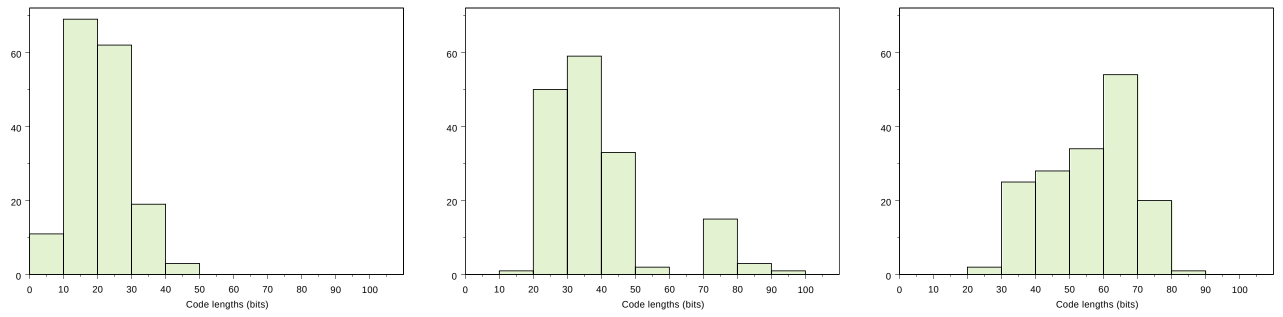


Figure 2. Heart; encoded transaction lengths for all transactions belonging to one class (DB_{10}), encoded with the code tables constructed for each of the three classes (left to right: CT_{10}, CT_{22}, CT_{26}).

$$L_{CT}(t) = \sum_{c \in cover(t,CT)} l_{CT}(c) \qquad \textbf{(2)}$$

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:

$$L_{CT}(db) = \sum_{t \in db} L_{CT}(t) = -\sum_{c \in CT} freq(c) \cdot \log\left(\frac{freq(c)}{\sum_{d \in CT} freq(d)} \right) \qquad \textbf{(3)}$$

2.2 Finding the Right Code Table

The above defines the database compression scheme and tells us how to compute encoded data lengths, but we haven't yet described the method that finds the optimal code table. For starters, in order to find a code table using MDL, we need to take into account both the compressed database size and the size of the code table. (Otherwise, the code table could grow without limits and become even larger than the original database!)

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*) which we use to compute the size of the item sets in the left-hand side column. Hence, the size of the code table is given by:

$$L(CT) = \sum_{c \in CT : freq(c) \neq 0} l_{st}(c) + l_{CT}(c) \qquad \textbf{(4)}$$

In [11] we defined the optimal set of (frequent) item sets as that one whose associated code table minimizes the total compressed size:

$$L(CT) + L_{CT}(db) \qquad \textbf{(5)}$$

The algorithm starts with a valid code table (generally only the collection of singletons) and a sorted list of candidates. These candidates are assumed to be sorted descending on 1) support and 2) item set length. 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.

Moreover, each existing code table element is considered for pruning when a new candidate has been added: when deleting an existing element does not reduce the compressed size it is put back, otherwise it is permanently pruned. The process of constructing code tables is illustrated in Figure 1. For more details, please see [11].

2.3 Classification

In [9] we tested the quality of our code tables as data distribution approximations using classification. Next to the compression ratios, this provided a second measure of how well the code tables describe the data.

The intuition behind the classifier is straightforward and implied by MDL. Suppose two databases db_1 and db_2, from which their respective code tables CT_1 and CT_2 are induced with KRIMP. If CT_1 gives a shorter encoding for an unseen transaction *t* than CT_2, the probability that *t* origins from the distribution of db_1 is larger than that it comes from db_2. This intuition can be written down as:

$$l_{CT_1}(t) < l_{CT_2}(t) \rightarrow \mathrm{P}(t \mid db_1) > \mathrm{P}(t \mid db_2) \qquad \textbf{(6)}$$

So, the Bayes optimal choice is to assign an unseen transaction to that code table that leads to the shortest code length.

The construction of the KRIMP classifier works as follows:

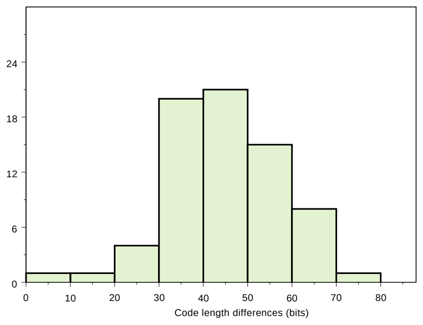
1. Split the training database according to class,
2. Remove the item(s) indicating class from each transaction,
3. Apply KRIMP to each of the databases. This yields a code table CT_i for each class C_i.

Then, to classify an unseen transaction *t*:

1. Compute $l_{CT_i}(t)$ for all classes C_i,
2. Assign *t* to the class that minimizes $l_{CT_i}(t)$.

Note that we have to do a Laplace correction on each CT_i to make sure each possible transaction can be covered by each code table. We showed that classification accuracies obtained with this classifier are on par with those obtained by the best known classifiers. We therefore concluded that KRIMP picks those item sets that together describe the data. See [9] for more detail.

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



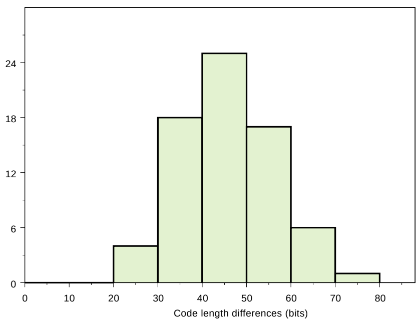


Figure 3. Wine; code length difference histograms for DB_9 with CT_{14} (left) and CT_{22} (right)

$$\begin{aligned} CT_i(db_j) &= L_{CT_i}(db_j) \\ CT_i(t) &= L_{CT_i}(t) \end{aligned} \quad \textbf{(7)}$$

During the classification experiments, we made some interesting observations in the distributions of the code lengths (not shown previously). Figure 2 shows the encoded lengths for transactions of a single class, encoded by code tables constructed for each of the three classes. Not only gives the code table constructed for these transactions shorter encodings, the standard deviation is also much smaller (compare the histogram on the left to the other two). This means that a better fit of the code table to the distribution of the compressed data results in a smaller standard deviation.

2.4 Experimental Setup

Although a lot of time series data is being gathered for analysis, no good benchmark datasets with this type of data currently exist. We therefore decided to use a selection from the UCI repository [3], which has been commonly used for emerging patterns [5] and related topics before.

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

For many steps in Sections 3 and 4, we show results obtained with the datasets Heart and Wine because of their properties: they are interesting because they consist of more than 2 classes, but don't have too many classes. Please note this selection is only for purpose of presentation; results we obtained with other (larger) datasets are similar. In fact, KRIMP is better at approximating data distributions with larger databases, giving even more reliable results.

Characteristics of all datasets used are summarized in Table 8, together with the minimum support levels we use for mining the frequent item sets that function as candidates for KRIMP. All experiments in this paper are done with all frequent item sets.

3. DATABASE DISSIMILARITY

In this Section, we introduce a dissimilarity measure for transaction databases. This measure indicates whether or not it is worthwhile to analyse the differences between two such databases. If the dissimilarity is low, the differences between the two databases are small. If the measure is high, it is worthwhile to investigate the differences.

Rather than defining the similarity measure upfront followed by a discussion and illustration of its properties, we "develop" the measure in a few steps as that allows us to discuss the intuition that underlies the definition far easier.

3.1 Differences in Code Lengths

The MDL principle implies that the optimal compressor induced from a database db_1 will generally provide shorter encodings for its transactions than the optimal compressor induced from another database db_2. Our earlier experiments on classification verify that this is also true for the code table compressors KRIMP discovers heuristically; see Section 2.

More in particular, denote by MDL_i the optimal compressor induced from database db_i and let t be a transaction in db_1. Then, the MDL principle implies that:

$$| MDL_1(t) - MDL_2(t) | \quad \textbf{(8)}$$

- is small if t is equally likely generated by the underlying distributions of db_1 and db_2.
- is large if t is more likely generated by the distribution underlying one database than that it is generated by the distribution underlying the other.

In fact the MDL principle implies that if the code length differences are large (the second case) than on average, the smallest code length will be $MDL_1(t)$.

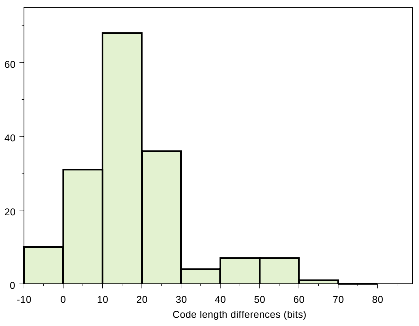
Our classification results suggest that something similar should hold for the code table compressors discovered by KRIMP. In other words, we expect that

$$CT_2(t) - CT_1(t) \quad \textbf{(9)}$$

measures how characteristic t is for db_1. That is, we expect that this difference is most often positive and large for those transactions that are characteristic for db_1.

In Figures 3 and 4 code length differences are shown for two datasets, respectively for transactions of the Wine_9 and Heart_{10} databases. As we expected, virtually all code length differences are positive. This means that in practice the native code table does indeed provide the shortest encoding.

In the case of the Wine_9 database depicted in Figure 3, we see a whopping average difference of 45 bits per transaction. The



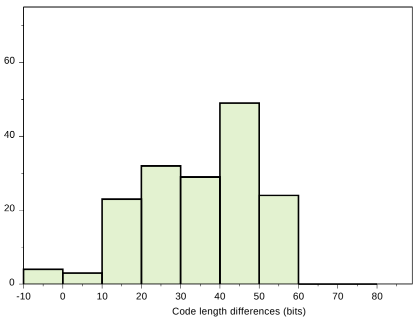


Figure 4. Heart; code length difference histograms for DB_{10} encoded by CT_{22} (left) and CT_{26} (right)

shapes of the two histograms also show a nice clustering of the differences between the encoded lengths. No negative differences occur, each single transaction is compressed better by its native code table. This confirms that MDL creates code tables that are truly specific for the data.

We see the same general effect with $Heart_{10}$ in Figure 4, as again the peaks of the distribution lay within safe distance from the origin. From the histograms there is little doubt that code tables CT_{22} and CT_{26} are encoding data from a different distribution than they've been induced from. More importantly, comparing these diagrams unambiguously shows that it is possible to use the differences in encoded lengths to measure the amount of change between data. For example, as the differences on the left histogram are clearly smaller than in the situation on the right, this seems to imply that Heart classes 10 and 22 are more alike than classes 10 and 26. How to investigate this hypothesis further will be discussed in the next Section. First we continue the development of our dissimilarity measure.

3.2 Aggregating Code Length Differences

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

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

- code length differences can be negative, so even if db_1 and db_2 are rather different, the aggregated total might be small.
- if db_1 is a large database, the aggregated total might be large even if db_2 is very similar to db_1.

As already mentioned in the previous subsection, the MDL principle implies that for the MDL-optimal compressors MDL_1 and MDL_2, the expected average value of $MDL_2(t) - MDL_1(t)$ is positive. In other words, negative code length differences will be relatively rare and won't unduly influence the aggregated sum.

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

The second problem indicated above is, however, already a problem for the MDL-optimal compressors MDL_1 and MDL_2. For, the expected value of the sum of the code length differences is simply the number of transactions times the expected average code length difference. Since the latter number is positive according to the MDL principle, the expected value of the sum depends linearly on the number of transactions on the database.

Clearly, the "native" encoded size of the database, $CT_1(db_1)$, also depends on the size of the database. Therefore, we choose to counterbalance this problem by dividing the sum of code length differences by this size. Doing this, we end up with the Aggregated Code Length Difference:

$$ACLD(db_1, CT_2) = \frac{CT_2(db_1) - CT_1(db_1)}{CT_1(db_1)} \qquad \textbf{(10)}$$

Note that ACLD is an *asymmetric* measure: it measures how different db_2 is from db_1, not vice versa! While one would expect both to be in the same ballpark, this is by no means given. The asymmetry is further addressed in the next subsection. To clearly indicate the asymmetry, the parameters are asymmetric: the first parameter is a database, while the second is a code table.

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

Table 1. Heart: aggregated code length differences for each database/code table combination.

	DB 10	**DB 22**	**DB 26**	**DB 27**	**DB 36**
CT 10	0.00	0.36	0.71	0.88	1.58
CT 22	0.85	0.00	0.60	0.65	1.03
CT 26	1.65	0.78	0.00	0.60	1.25
CT 27	1.85	0.65	0.61	0.00	1.09
CT 36	2.18	1.07	0.72	0.87	0.00

In Table 1 we read the aggregated code length differences for all possible combinations of code tables and class databases for the Heart dataset. It is immediately clear there are distinct differences between the class distributions, as measurements of 1.00 imply code lengths averaging twice as long as that of the actual class. We also notice that while the data distributions of databases 10 and 36 are quite distinct, the lower measurements between the other three classes indicate that their distributions are more alike.

Table 2. Wine: aggregated code length differences for each database/code table combination.

	DB 9	DB 14	DB 22
CT 9	0.00	1.27	1.32
CT 14	1.13	0.00	1.73
CT 22	1.14	1.68	0.00

For the Wine database the class distributions are even more adrift than those in the Heart database, for all cross-compressions result in encodings more than twice as long as the native ones. This is completely in line of what we've seen before in Figure 3, in which we showed there is no uncertainty in keeping transactions of the Wine databases apart based on encoded lengths.

If this technique truly quantifies the likeliness of the distributions belonging to some data, intuition tells us there has to be a close relation with the classification quality based on encoded transaction lengths. We can easily check this by comparing the aggregated code length differences with the confusion matrices for these databases. We therefore ran 10-fold cross validated classification experiments for these databases, as we did for previous work [9].

Table 3. Heart: classification confusion matrix.

Classified as:	Class 10	Class 22	Class 26	Class 27	Class 36
10	137	24	9	6	3
22	12	11	11	7	5
26	6	8	7	8	1
27	8	10	7	9	4
36	1	2	2	5	0

The confusion matrix for the Heart database, in Table 3, clearly shows the intuition to be correct, as the number of misclassified instances drops completely according to ACLD. While the code table for class 10 incorrectly misclassifies 24 transactions of class 22, we see in Table 1 these two classes are measured as rather similar. In fact, if we sort the measurements in Table 1 per class, we find the same order as when we sort Table 3 on the number of misclassifications. The measured difference thus directly relates to the ability to distinguish classes.

Table 4. Wine: classification confusion matrix.

Classified as:	Class 9	Class 14	Class 22
9	65	3	6
14	5	55	0
22	1	1	42

In Table 4 we see the same pattern with the Wine database as with the Heart database before: the lowest differences relate to the most misclassifications. We also observe that while analysis of individual code length differences, like Figure 3, suggests there should be no confusion in classification, a number of transactions are misclassified. These can be tracked back as being artefacts of the 10-fold cross validation on a small database.

3.3 The Database Dissimilarity Measure

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

$$\max\{ACLD(db_a, CT_b), ACLD(db_b, CT_a)\} \qquad \textbf{(11)}$$

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

Definition 1: for all databases x and y, define the code table dissimilarity measure DS between x and y as:

$$DS(x,y) = \max\left\{\frac{\mathrm{CT}_y(x) - CT_x(x)}{CT_x(x)}, \frac{CT_x(y) - CT_y(y)}{CT_y(y)}\right\} \qquad \textbf{(12)}$$

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

1) D(x,y) = 0 iff $x = y$ (identity)
2) D(x,y) = D(y,z) (symmetry)
3) D(x,y) + D(y,z) ≥ D(x,z) (triangle inequality)

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

The dissimilarity measurements for the Heart, Nursery and Wine database are given in respectively Tables 5, 6 and 7. One of the most striking observations is that many of the measurements are greater than 1.0, meaning that the cross-compressed databases are more than twice as large as the natively-compressed databases. The differences between the Nursery$_{16}$ and Nursery$_{31}$ datasets are so huge that a dissimilarity measurement of 10.12 is the result: a staggering difference of a factor 11 times in average encoded length.

Table 5. Heart: dissimilarity.

	DB 10	DB 22	DB 26	DB 27
DB 22	0.85			
DB 26	1.65	0.78		
DB 27	1.85	0.65	0.61	
DB 36	2.18	1.07	1.25	1.09

Table 6. Nursery: dissimilarity.

	DB 2	DB 15	DB 16	DB 30
DB 15	2.62			
DB 16	2.83	2.04		
DB 30	3.10	1.91	4.05	
DB 31	7.38	1.26	10.12	1.54

Table 7. Wine: dissimilarity.

	DB 9	DB 14
DB 14	1.27	
DB 22	1.32	1.73

In Table 8 a summary of datasets, their characteristics and dissimilarity results is given. For each dataset, the lowest and the highest observed dissimilarity is listed. A full results overview would obviously require too much space; datasets with many classes have squared as many database pairs of which the dissimilarity can be measured.

Overall, we see that the dissimilarities between the classes of the UCI datasets vary quite a bit. Some datasets seem to have very little difference between classes (Connect-4, Adult, TicTacToe), others contain rather large dissimilarity (Mushroom, Iris, Led7).

Another interesting comparison is between the dissimilarities and the classification results also reported in that table, taken from [9]. There is a clear correlation between the two. The larger the dissimilarity, the better the classification results. This pattern is less clear for datasets containing small classes, which is caused by the fact that MDL doesn't work well for small data sets.

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

4. CHARACTERISING DIFFERENCES

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

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

4.1 Comparing database covers

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

To run this analysis, we first use KRIMP to obtain a code table for database db_2 and use it to cover database db_1. Because the item sets and their frequencies in the code table capture the data distribution of database db_2, the frequencies found by covering database db_1 are expected to be different if the two databases are different. Identification of these differences is done by finding those patterns in the code table that have a large shift in frequency between the two database covers. The same process can be applied vice versa for even better insight of the differences.

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

An example visualisation can be seen in Figure 5. A code table

Table 8. Database characteristics, candidate min sup and dissimilarity measurements for a range of UCI datasets. As candidates, all frequent item sets were used up to the given minimum support level.

Dataset			**KRIMP**		**Dissimilarity (*DS*)**	
Name	*#rows*	*#classes*	*Min sup*	*Accuracy*	*Minimum*	*Maximum*
Adult	48842	2	20	84.6	0.60	0.60
Chess (kr-k)	28056	18	10	58.0	0.29	2.69
Connect-4	67557	3	50	69.9	0.18	0.28
Heart	303	5	1	52.5	0.61	2.18
Iris	150	3	1	96.0	2.06	13.00
Led7	3200	10	1	75.3	1.27	11.29
LetterRecog	20000	26	50	68.1	0.43	2.83
Mushroom	8124	2	50	100	7.08	8.24
Nursery	12960	5	1	92.4	1.26	10.12
PenDigits	10992	10	20	88.6	1.33	4.43
TicTacToe	958	2	1	87.1	0.62	0.62
Wine	178	3	1	97.7	1.27	1.73

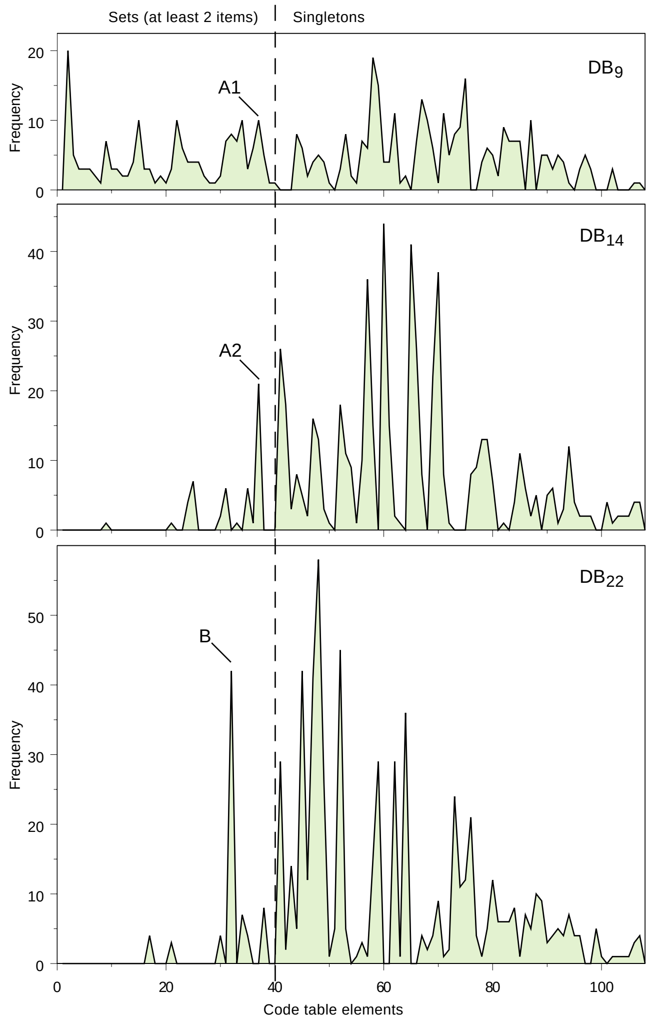


Figure 5. Comparing database covers. Each database of Wine has been covered by code table CT$_9$. Visualised is the absolute frequency for each of the code table elements.

for Wine DB$_9$ has been constructed and used to cover all three databases. A quick glance shows that our hypothesis on the use of singletons is correct: DB$_9$ is covered by quite some sets of 2 or more items, but both DB$_{14}$ and DB$_{22}$ are covered largely by singletons.

Of special interest is the contrast in peaks between the plots, indicating (strong) shifts in pattern usage. A rather strong difference in pattern usage is visible for the lower indexes in the code table, corresponding to the longest, most specific, patterns. However, in this figure the high peaks are also indicative; we marked the peaks of an interesting case A1 and A2. These peaks are at exactly the same code table element, meaning that this pattern is used quite often in the covers of both DB$_9$ and DB$_{14}$. Note that it is not used at all in the cover of DB$_{22}$; hence this pattern could really give us a clue as to what differentiates DB$_9$ and DB$_{14}$ from DB$_{22}$. Another interesting peak is the one indicated with B: although it is also applied in the other covers, this pattern is clearly used much more often to cover DB$_{22}$.

4.2 Comparing transaction covers

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

Suppose we have two databases and their respective code tables. After computing the individual code length differences (as described in section 3.1), it is easy to pick out those transactions that fit well in one database and not in another. After selecting a transaction, we can cover it with both code tables separately and visualise which patterns are used for this. In general, it will be covered by longer and more frequent patterns if it belongs to a certain distribution than if it does not. Manual inspection of the individual transaction covers can reveal valuable knowledge.

As an example, have a look at another Wine example in Figure 6. The encodings by CT$_9$ and CT$_{22}$ of two sets from DB$_{22}$ are shown. Left and right show the same transactions, but they are covered by different item sets (depicted by the rounded boxes). The item sets are linked to their codes with the dashed lines. The width of each black or white code represents the length of that particular code; together the sum of these widths makes up the total length of the encoded transaction.

Looking at the upper transaction, we observe that both code tables cover the transaction with item sets of intermediate length. However, CT$_{22}$ uses less and different patterns in its cover than CT$_9$. Moreover, the code lengths are obviously shorter, relating to high occurrence in the distribution from which CT$_{22}$ was induced. For further inspection of how important such patterns are, we zoom in to the pattern level in the third approach.

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

4.3 Comparing code tables

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

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

For each pattern in a code table we can compare its own encoded length to that of the alternative provided by the other code table, similarly to what we did for transactions in Section 3.1. Likewise, if the distributions are similar, we expect the encoded lengths to be comparable; even if the code tables use rather different patterns to encode it. In contrast, exactly those patterns for which the encoded lengths differ significantly mark the difference between the distributions.

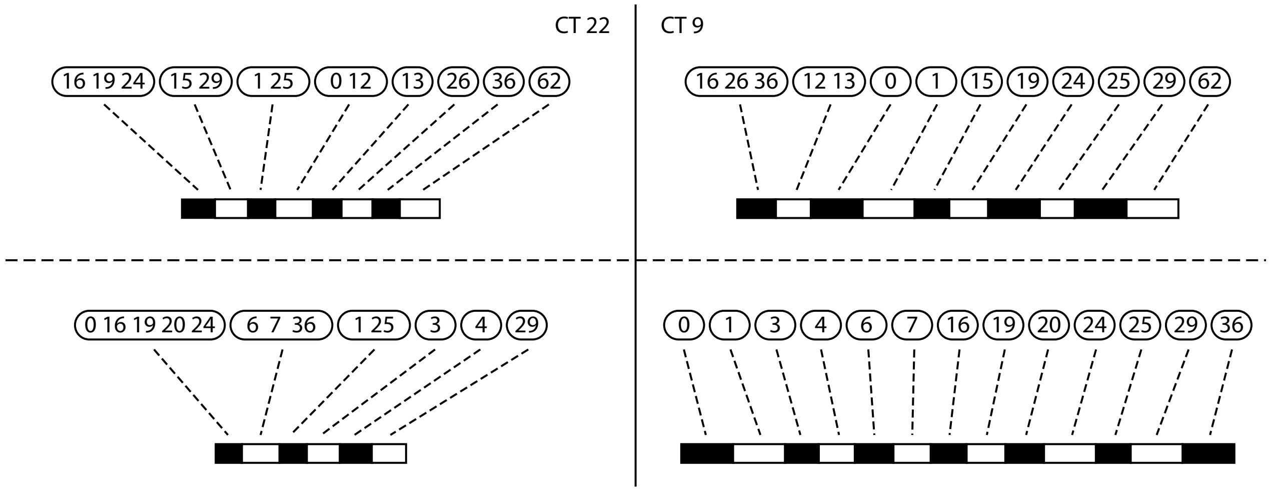


Figure 6. Wine; two transactions from DB_{22} encoded by CT_{22} (left) and CT_9 (right). The rounded boxes visualise the item sets making up the cover of the transaction. Each of the item sets is linked to its code by the dashed line. The widths of the black and white encodings represent the actual computed code lengths.

We analysed the CT_{14} and CT_{22} code tables of the Wine database, and found further evidence for what puts these databases apart. The first peak in the topmost plot of Figure 5 corresponds to the pattern (0 16 19 20 24) from CT_{22}, which due to its high relative usage is encoded natively using only 1.4bits. From the same figure we already know this pattern is not used when covering the other databases; suggesting that perhaps neither this pattern, nor anything like it exists in the other code tables. Confirmation comes from an encoded length of 12.6bits CT_{14} assigns to this pattern; making it one of the patterns for which the encoded lengths differ most. As CT_{14} cannot use any of the more frequently occurring code table patterns, it has to resort to low-frequency singleton encoding; arguably the least efficient method for encoding a pattern.

From the definition of the Wine database and analysis above we conclude that the main difference between the two classes lies in the combination of certain levels of malic acid (element 0) and a corresponding colour intensity (16). While CT_{22} has a number of patterns that give these short encodings, CT_{14} has virtually none: this pattern does not occur in this data distribution.

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

5. RELATED WORK

Our dissimilarity measure DS is clearly related to the Normalized Information Distance (NID) and its compression-based instantiation NCD [10]. With the NCD, general compressors like gzip are used as Kolmogorov complexity approximators and as such compressed sizes are used to measure distance between strings. As a generic distance, the NID has been successfully applied in a plethora of clustering tasks including small snippet based language and evolutionary tree rebuilding [4]. An adaptation was developed that has some practical data mining applications, among which compression-based anomaly detection [8]. However, the aim of the NID is different from ours: compression is only used as a means to quantify differences, not to qualitatively find what these differences are. In contrast, this is the main goal of our line of research. This is illustrated by the results of both our earlier papers and in this paper. By considering transaction databases instead of individual strings and building code tables that can be analysed, KRIMP provides a very natural way to gain insight in the differences between data distributions.

Our dissimilarity measure is also clearly related to Emerging Patterns [5], although there are two major differences. First of all, here we only consider patterns that are important in the data distribution of the database; MDL has selected these for participation in the code table. This results in only a small set of patterns describing the chance, making in-depth analysis of the differences feasible and informative. Secondly, Emerging Patterns are defined as patterns having a large difference in their individual support between two databases, thus irrespective of other patterns in the database. Our approach works with frequencies that depend on the database cover and therefore on the patterns above in the code table. Because of such dependencies, our method automatically identifies the most important changes in the structure of the data.

6. CONCLUSIONS & FUTURE WORK

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

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

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

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

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

Dissimilarity measures are key to many different data mining algorithms. In the near future we expect to apply our measure in a number of bio-informatics applications using these algorithms. For example, in those cases where classification appears to be hard; deeper insight in the causes of these problems might suggest promising research directions.

7. ACKNOWLEDGMENTS

J.Vreeken is supported by the NWO Computational Life Sciences programme. M. van Leeuwen is supported by the NBIC BioRange programme.

8. REFERENCES

[1] Bathoorn, R., Koopman, A., and Siebes, A. Reducing the Frequent Pattern Set. In *Proceedings IEEE Conf. on Data Mining – Workshops* (*ICDMW* '06) 2006, 55-59.

[2] Codd, E.F., S.B. Codd, C.T. Salley, *Providing OLAP (On-LineAnalytical Processing) to User Analyst: An IT Mandate*. http://www.arborsoft.com/OLAP.html. 1994.

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

[4] Cilibrasi, R., and Vitanyi, P. Clustering by Compression. In IEEE Transactions on Information Theory. 51, 4 (Apr 2005), 1523-1545.

[5] Dong, G., and Li, J. Efficient mining of emerging patterns: Discovering trends and differences. In *Proc. SIGKDD conference on Knowledge Discovery in Data. (KDD '99)*, 1999, 43-52.

[6] Goethals, B. et al. *Frequent Itemset Mining Implementations Repository,* http://fimi.cs.helsinki.fi/

[7] Grünwald, P.D. Minimum description length tutorial. In *Advances in Minimum Description Length* (Grünwald, P.D., Myung, I.J. & Pitt, M.A., editors). MIT Press, 2005.

[8] Keogh, E., Lonardi, S., and Ratanamahatana, C.A. Towards Parameter-Free Data Mining. In *Proc. SIGKDD conference on Knowledge Discovery in Data. (KDD'04)*, 2004, 206-215.

[9] Van Leeuwen, M., Vreeken, J., and Siebes, A. Compression Picks Item Sets That Matter. In *Proc. Knowledge Discovery in Databases (PKDD'06)*, 2006, 585-592.

[10] Li, M., Chen, X., Li, X., Ma, B., and Vitányi, P.M.B. The Similarity Metric. In *IEEE Transactions on Information Theory*. 50, 12 (Aug 2004), 3250-3264.

[11] Siebes, A., Vreeken, J., and Van Leeuwen, M. Item Sets That Compress. In *Proceedings of the 2006 SIAM Conference on Data Mining*. 2006, 393-404.

[12] Zhang, X., Guozhu, D., and Ramamohanarao, K. Information-based Classification by Aggregating Emerging Patterns. In *Proc. Intelligent Data Engineering and Automated Learning, Data Mining, Financial Engineering, and Intelligent Agents (IDEAL)*. 2000, 48-53.