Privacy Preservation through Data Generation

Jilles Vreeken, Matthijs van Leeuwen & Arno Siebes

Department of Information and Computing Sciences
Utrecht University
Technical Report UU-CS-2007-020
www.cs.uu.nl
ISSN: 0924-3275

Privacy Preservation through Data Generation [1]

Jilles Vreeken, Matthijs van Leeuwen and Arno Siebes
Universiteit Utrecht
{jillesv,mleeuwen,arno}@cs.uu.nl

Abstract

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

In this paper we show that generating new data is an inherently safer alternative. We present a data generator based on the models obtained by the MDL-based KRIMP [18] algorithm. These are accurate representations of the data distributions and can thus be used to generate data with the same characteristics as the original data.

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

1. Introduction

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

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

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

An alternative approach to the PPDM problem is to generate new data instead of perturbing the original. This has the advantage that the original data can be kept safely private as the generated data is published instead, which should render data recovery attacks useless. To achieve this, the expectation that a data point in the generated database identifies a data point in the original database should be very low, while all generated data together should adhere to the characteristics of the original database. Data generation as a means to cover-up sensitivities has been explored in the context of statistical databases [13], but that method ignores correlations as each dimension is sampled separately.

We propose a novel method that uses data generation to guarantee privacy while taking important correlations into account. For this we use the MDL-based KRIMP algorithm [18] that has been shown to provide accurate pattern-based approximations of data distributions. The high quality of the approximations was verified through classification [12], and consequently put to use for determining and characterising dissimilarities between datasets [19]. Using the patterns picked by MDL, we can construct a model that generates data very similar (but not equal) to the data the patterns were derived from. Experiments show that the generative model is well suited for producing data that con-

[1] This is an extended version of work published at ICDM 2007 [20].

serves the characteristics of the original data while preserving privacy.

Using our generative method, it is easy to ensure that generated data points cannot reliably be traced to individual data points in the original data. We can thus easily obtain data that is in accordance with the well-known privacy measure *k*-anonymity [17]. Also, we can mimic the effects that can be obtained with *l*-diversity [15].

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

2. The Problem

2.1 Data Perturbation

Since Agrawal & Srikant [3] initiated the privacy-preserving data mining field, researchers have been trying to protect and reconstruct sensitive data. Most techniques use data perturbation and these can be divided into three main approaches, of which we will give an overview here.

The addition of random noise to the original data, obfuscating without completely distorting it, was among the first proposals for PPDM [3]. However, it was quickly shown that additive randomization is not good enough [4]. The original data can often be reconstructed with little error using noise filtering techniques [11] - in particular when the distortion does not take correlations between dimensions into account [10].

The second class is that of condensation-based perturbation [1]. Here, after clustering the original data, new data points are constructed such that cluster characteristics remain the same. However, it has been observed that the perturbed data is often *too* close to the original, thereby compromising privacy [6].

A third major data perturbation approach is based on rotation of the data [6]. While this method seemed sturdy, it has recently been shown that with sufficient prior knowledge of the original data the rotation matrix can be recovered, thereby allowing full reconstruction of the original data [14].

In general, perturbation approaches suffer from the fact that the original data is used as starting point. Little perturbation can be undone, while stronger perturbation breaks correlations and non-sensitive information is also destroyed. In other words, there is a privacy-quality trade-off which cannot be balanced well.

In the effort to define measures on privacy, a few models have been proposed that can be used to obtain a definable amount of privacy. An example is the well-known *k*-anonymity model that ensures that no private information can be related to fewer than *k* individuals [17]. A lack of diversity in such masses can thwart privacy though and in some situations it is well possible to link private information to individuals. Improving on *k*-anonymity, the required critical diversity can be ensured using the *l*-diversity model. However, currently the available method can only ensure diversity for one sensitive attribute [15].

2.2 Data Generation

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

Liew et al. [13] sample new data from probability distributions independently for each dimension, to generate data for use in a statistical database. While this ensures high quality point estimates, higher order dependencies are broken – making it unsuited for use in data mining.

The condensation-based perturbation approach [1] could be regarded as a data generation method, as it samples new data points from clusters. However, as mentioned above, it suffers from the same problems as perturbation techniques.

2.3 Problem Statement

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

A database db_{priv} induced from a database db_{orig} is privacy and quality preserving *iff*:

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

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

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

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

Definition 1: for a database db_p based on db_o, define the Anonymity Score (AS) as:

$$AS(db_p, db_o) = \sum_{supp \in db_o} \frac{1}{supp} P(t \in db_p \mid t \in db_o^{supp}) \quad (1)$$

In this definition, db^{supp} is defined as the selection of db with only those transactions having a support of $supp$. For each support level in db_o, a score is obtained by multiplying a penalty of 1 divided by the support with the probability that a transaction in db_o with given support also occurs in db_p. All these scores are summed to obtain AS. Note that when all transactions in db_o are unique (e.g., have a support of 1), AS is equal to the probability that a transaction in db_o also occurs in db_p.

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

Definition 2: for a database db_{priv} based on db_{orig}, define the Normalised Anonymity Score (NAS) as:

$$NAS(db_{priv}, db_{orig}) = \frac{AS(db_{priv}, db_{orig})}{AS(db_{orig}, db_{orig})} \quad (2)$$

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

$$P(|\text{supp}_{priv} - \text{supp}_{orig}| > \delta) < \varepsilon \quad (3)$$

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

3. Preliminaries

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

The KRIMP algorithm operates on item set data, as which categorical data can easily be regarded. The union of all domains $\cup D_i$ forms the set of items $\mathcal{I}$. Each transaction t can now also be regarded as a set of items $t \in \mathcal{P}(\mathcal{I})$. 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. Speaking in market basket terms, this means that each item for sale is represented as an attribute, with the corresponding domain consisting of the values 'bought' and 'not bought'.

3.1 Compression with KRIMP

Siebes et al [18] introduced the KRIMP algorithm, which finds a small set of patterns that together capture the distribution of the data. This approximation will be used as basis for our data generator and we will therefore give a quick summary of the method.

In KRIMP, we have a code table that has item sets on the left-hand side and codes 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.

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.

To compute the length of a code that belongs to an item set, 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 [9] we have the optimal code length for c as:

$$l_{CT}(c) = -\log(\mathrm{P}(c \mid db)) = -\log\left(\frac{freq(c)}{\sum_{d \in CT} freq(d)}\right) \quad (4)$$

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

$$L_{CT}(t) = \sum_{c \in cover(t,CT)} l_{CT}(c) \quad (5)$$

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:

$$L_{CT}(db) = \sum_{t \in db} L_{CT}(t) = -\sum_{c \in CT} freq(c) \cdot \log\left(\frac{freq(c)}{\sum_{d \in CT} freq(d)}\right) \quad (6)$$

3.2 Finding the Right Code Table

Now that we defined the database compression scheme, we can describe the actual algorithm that finds the optimal code table using MDL. For this, we need to take into account both the compressed database size and 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*) 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:

$$L(CT) = \sum_{c \in CT : freq(c) \neq 0} l_{st}(c) + l_{CT}(c) \quad (7)$$

In [18] Siebes et al defined the optimal set of (frequent) item sets as that one whose associated code table minimizes the total compressed size:

$$L(CT) + L_{CT}(db) \quad (8)$$

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. For more details on the algorithm, please see [18].

No pruning strategy is applied in this paper, since keeping all patterns in the code table causes more diversity during data generation, as will become clear later.

3.3 The Database Dissimilarity Measure

In [19] Vreeken et al introduced a database dissimilarity measure based on KRIMP code table compressed database sizes. First define $CT_x(y)$ as the total compressed size of database y as compressed with the code table obtained by applying KRIMP on database x.

Definition 3: for all databases x and y, define the code table dissimilarity measure DS between x and y as:

$$DS(x,y) = \max\left\{\frac{CT_y(x) - CT_x(x)}{CT_x(x)}, \frac{CT_x(y) - CT_y(y)}{CT_y(y)}\right\} \quad (9)$$

Two databases are deemed very similar (possibly identical) iff the score is 0, higher scores indicate higher levels of dissimilarity. As the code tables consist of frequent patterns, it is especially good at measuring the pattern similarity on a database level, as experiments confirmed [19]. We will therefore use it in our experimental section to quantify the differences between original and generated data, helping to verify the quality requirement of the problem statement.

4. KRIMP Categorical Data Generator

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

4.1 Generating a transaction, an example

Suppose we need to generate a new transaction for a simple three-column categorical database. To apply our generation scheme, we need a domain definition $\mathcal{D}$ and a KRIMP code table CT, both shown in Figure 1.

We start off with an empty transaction and fill it by iterating over all domains and picking an item set from the code table for each domain that has no value yet. We first want to assign a value for the first domain, D_1, so we have to select one pattern from those patterns in the code table that provide a value for this domain. This subset is shown as selection CT^{D_1}.

Using the frequencies of the code table elements as probabilities, we randomly select an item set from CT^{D_1}; elements with high frequency occur more often in the original database and are thus more likely to be picked. Here we randomly pick ‘BD’ (probability 3/9).

Domain definition

$\mathcal{D}$ = { D_1 = { A, B }; D_2 = { C, D }; D_3 = { E, F } }

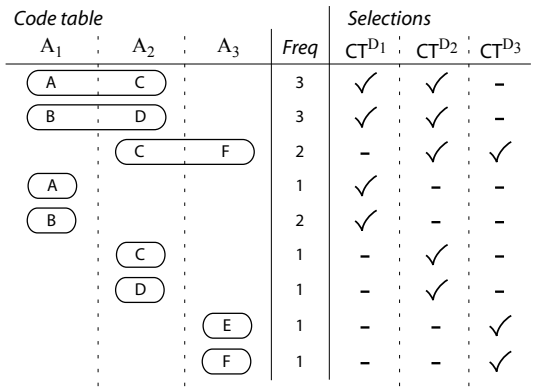

Figure 1. Example for 3-column database. Each frequency is Laplace corrected by 1.

This set selects value 'B' from the first domain, but also assigns a value to the second domain, namely 'D'.

To complete our transaction we only need to choose a value for the third domain. We do not want to change any values once they are assigned, as this might break associations within an item set previously chosen. So, we do not want to pick any item set that would re-assign a value to one of the first two domains. Considering the projection for the third domain, CT^{D_3}, we thus have to ignore set CF(2), as it would re-assign the second domain to 'C'. From the remaining sets E(3) and F(3), both with frequency 3, we randomly select one – say, 'E'. This completes generation of the transaction: 'BDE'. With different rolls of the dice it could have generated 'BCF' by subsequently choosing CF(2) and B(3), and so on.

4.2 Definition of the generator

Here we will detail our data generator more formally. First, define the projection CT^D as the subset of item sets in CT that define a value for domain $D \in \mathcal{D}$. To generate a database, our categorical data generator requires four ingredients: the original database, a Laplace correction value, a *min-sup* value for mining candidates for the KRIMP algorithm and the number of transactions that is to be generated. We present the full algorithm in pseudo-code below, and describe it in detail here.

Generation starts with an empty database *gdb* (line 2). To obtain a code table *CT*, the KRIMP algorithm is applied to the original database *db* (3). A Laplace correction *laplace* is added to all elements in the code table (lines 4 and 5). Next, we return the generated database when it contains *num-trans* transactions (lines 7 to 9).

Generation of a transaction is started with an empty transaction t (line 11). As long as $\mathcal{D}$ is not empty (12), our transaction is not finished and we continue. First, a domain D is randomly selected (13). From the selection CT^D, one item set is randomly chosen, with probabilities defined by their relative frequencies (14). After the chosen set is added to t (15), we filter from CT all sets that would redefine a value – i.e. those sets that intersect with the definitions of the domains for which t already has a value (lines 16 and 17). Further, to avoid reconsideration we also filter these domains from $\mathcal{D}$ (18). After this the next domain is picked from $\mathcal{D}$ and another item set is selected; this scheme is repeated until $\mathcal{D}$ is empty (and t thus has a value from each domain).

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

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

A desirable parameter for any data generation

ALGORITHM KRIMPGENERATOR

```
1  GenerateDatabase(db, laplace, min-sup, num-trans)
2      gdb = ∅
3      CT = KRIMP(db, MineCandidates(db, min-sup))
4      for each item set e in CT
5          e.frequency += laplace
6      𝒟 = db.getDomains
7      while(|gdb| < num-trans)
8          gdb = gdb + GenerateTransaction(CT, 𝒟)
9      return gdb

10 GenerateTransaction(CT, 𝒟)
11     t = ∅
12     while 𝒟 ≠ ∅
13         pick a random D ∈ 𝒟
14         is = PickRandomItemSet(CT^D)
15         t = t ∪ is
16         for each domain C for which is has a value
17             CT = CT \ CT^C
18             𝒟 = 𝒟 \ C
19     return t

20 PickRandomItemSet(CT)
21     weights = { e.frequency | e ∈ CT }
22     is = WeightedSample(weights, CT)
23     return is
```

Table 1. Database characteristics, candidate min-sup and dissimilarity measurements (between original and generated datasets) for a range of datasets. As candidates, frequent item sets up to the given minimum support level were used.

Dataset			**KRIMP**	**Dissimilarity**	
Name	*#rows*	*#domains*	*Min-sup*	*Gen. vs. orig.*	*Orig. internal*
Chess (kr-k)	28056	7	1	0.037	0.104
Iris	150	5	1	0.047	0.158
Led7	3200	8	1	0.028	0.171
LetterRecog	20000	17	50	0.119	0.129
Mushroom*	8124	22	20	0.010	0.139
Nursery	12960	9	1	0.011	0.045
PageBlocks	5473	11	1	0.067	0.164
PenDigits	10992	17	50	0.198	0.124
Pima	786	9	1	0.110	0.177
Quest A	4000	8	1	0.016	0.077
Quest B	10000	16	1	0.093	0.223

* Only closed item sets used as candidates.

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

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

5. Experiments

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

5.1 Experimental Setup

In our experiments, we use a selection from the commonly used UCI repository [7]. Also, we use two additional databases that were generated with IBM's Quest basket data generator [2]. To ensure that the Quest data obeys our categorical data definition, we transformed it such that each original item is represented by a domain with two categories, in a binary fashion (present or not). Both Quest datasets were generated with default settings, apart from the number of columns and transactions.

Characteristics of all used datasets are summarized in Table 1, together with the minimum support levels we use for mining the frequent item sets that function as candidates for KRIMP.

For all experiments we used a Laplace correction parameter of 0.001, an arbitrarily chosen small value solely to ensure that otherwise zero-frequency code table elements can be chosen during generation.

All experimental results presented below are averaged over 10 runs and all generated databases have the same number of transactions as the originals, unless indicated otherwise.

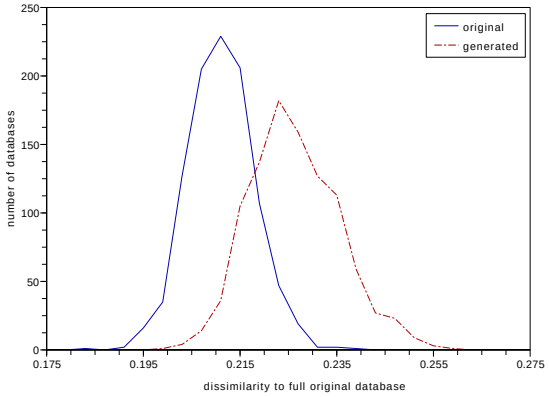

Figure 2. Histogram of dissimilarities between samples (original and generated) and the full original db, Chess (kr-k).

5.2 Results

To quantify the likeness of the generated databases to their original counterparts, we use the database dissimilarity measure as described in Section 3.3. In judging these measurements, a comparison with the diversity within the original data distribution is a valuable reference. We therefore measured the dissimilarity between the original database and independent random samples of half the size from the original database.

In Table 1 we show both these internal dissimilarity scores and the dissimilarity measurements between the original and generated databases. To put the reported dissimilarities in perspective, note that the dissimilarity measurements between the classes in the original databases range from 0.29 up to 12 [19]. The measurements in Table 1 thus indicate clearly that the generated databases adhere very closely to the original data distribution; even better than a randomly sampled subset of 50% of the original data captures the full distribution.

To show that the low dissimilarities for the generated databases are not caused by averaging, we provide a histogram in Figure 2 for the Chess (kr-k) database. We generated thousand databases of 7500 transactions, and measured the dissimilarity of these to the original database. Likewise, we also measured dissimilarity to the original database for equally many and equally sized independent random samples. The peaks for the distance histograms lie very near to each other at 0.21 and 0.22 respectively. This and the very similar shapes of the histograms confirm that our generation method samples databases from the original distribution.

Turning back to Table 1, we notice that databases generated at higher values of the *min-sup* parameter show slightly larger dissimilarity. The effect of this parameter is further explored in Figure 3. First, the bar diagram on the left shows a comparison of the dissimilarity scores between uncorrelated and correlated generation: uncorrelated databases are generated by a code table containing only individual values (and thus no correlations between domains can exist), correlated databases are generated using the *min-sup* values depicted in Table 1 (at which the correlations in the data are captured in the patterns in the code table). We see that when generation is allowed to take correlations into account, the generated databases are far more similar to the original ones.

Secondly, the graph on the right of Figure 3 shows the dissimilarity between the original PenDigits database and databases generated with different values for *min-sup*. As expected, lower values of *min-sup* lead to databases more similar to the original, as the code table can better approximate the data distribution of the original data. For the whole range of generated databases, individual value frequencies are almost identical to those of the original database; the increase in similarity is therefore solely caused by the incorporation of the right (type and strength of) correlations.

Now that we've shown that the quality of the generated databases is very good on a high level, let us con-

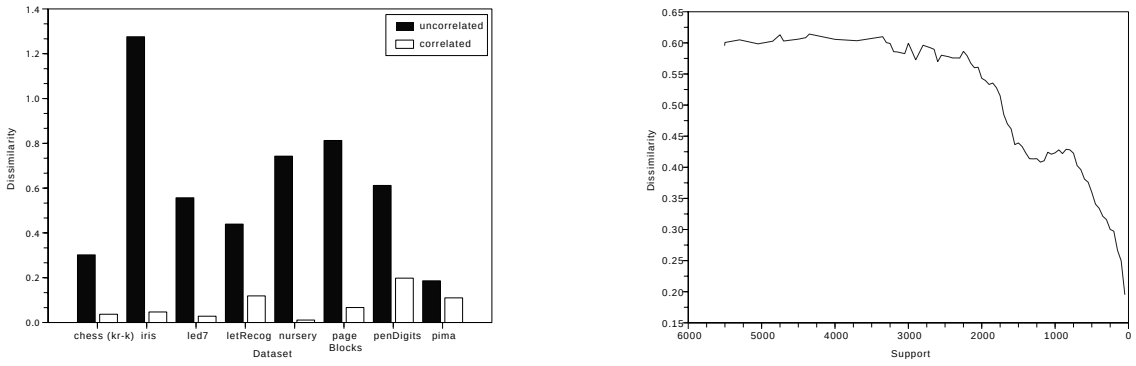

Figure 3. left) Dissimilarity scores between generated (with and without correlations) and original databases. right) Dissimilarity between generated database (at different *min-sups*) and the original database for PenDigits.

Table 2. Frequent pattern set comparison.

Name	*% equal item sets*	*% avg sup diff equal item sets*	*% avg sup new item sets*
Chess (kr-k)	71	0.01	0.01
Iris	83	1.69	0.80
Led7	89	0.14	0.06
Nursery	90	0.04	0.03
PageBlocks	75	0.06	0.02
PenDigits	25	0.50	0.59
Pima	60	0.30	0.14

sider quality on the level of individual patterns. For this, we mined frequent item sets from the generated databases with the same parameters as we did for candidate mining on the original database. A comparison of the resulting sets of patterns is presented in Table 2. We report these figures for those databases for which it was feasible to compute the intersection of the frequent pattern collections.

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

For item sets found in both cases, the average difference in support between original and generated is very small, as the second column shows. Iris is a bit of an outlier here, but this is due to the very small size of the dataset. Not only the average is low, standard deviation is also small: as can be seen from Figure 4, almost all sets have a very small support difference. The generated databases thus fulfil the support difference demands we formulated in Equation 3.

The third column of Table 2 contains the average supports of item sets that are newly found in the generated databases; these supports are very low. All this together clearly shows that there is a large pattern-similarity, thus showing a high quality according to our problem statement.

However, this quality is of no worth if the generated data does not also preserve privacy. To measure the level of provided anonymity, we calculate the Normalised Anonymity Score as given by Definition 2. These scores are presented in Table 3.

As lower scores indicate better privacy, some datasets (e.g. Mushroom, PenDigits) are anonymised very well. On the other hand, other datasets (PageBlocks, Quest) do not seem to provide good privacy. As discussed in Section 4, the *min-sup* parameter of our generation methods doubles as a *k*-anonymity provider.

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

To show the effect of *min-sup* in action, as an example we increase the *min-sup* for the Chess database to 50. While the so-generated database is still very similar to the original (dissimilarity of 0.19), privacy is considerably increased - which is reflected by a Normalised Anonymity Score of 0.15. For further evidence of the *k*-anonymity obtained, we take a closer look at PenDigits, for which we use a *min-sup* of 50. Of all transactions with support < 50 in the generated database, only 3% is also found in the original database with support < 50. It is thus highly unlikely that one picks a 'real' transaction from the generated database with support lower than *min-sup*.

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

The key issue is whether transactions are generated by only very few or many code table elements. In Figure 5 we show this relation: for each dataset in Table 1, a cross marks the average number of item sets used to generate a single transaction and the Normalised Anonymity Score. In the bottom-right corner we find the

Table 3. Normalised Anonymity Scores

Name	*Normalised Anonymity Score*
Chess (kr-k)	0.30
Iris	0.72
Led7	0.66
LetterRecog	0.31
Mushroom	0.09
Nursery	0.49
PageBlocks	0.77
PenDigits	0.22
Pima	0.64
Quest A	0.84
Quest B	0.81

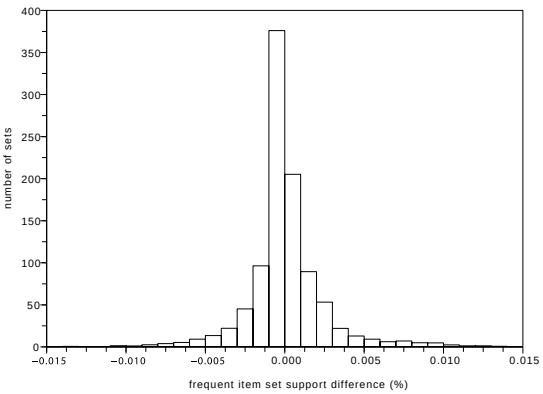

Figure 4. Difference in support, $supp(db_{priv}) - supp(db_{orig})$, for identical item sets in generated and original Led7.

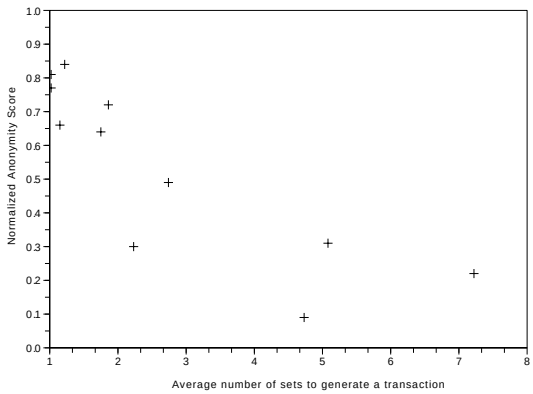

Figure 5. Average number of patterns used to generate a transaction versus Normalised Anonymity Score, for all datasets in Table 1.

generated databases that preserve privacy well, including PenDigits and LetterRecog. At the top-left reside those databases for which too few elements per transaction are used during generation, leading to bad privacy; Quest, PageBlocks and Led7 are the main culprits. Thus, by altering the *min-sup*, this relation allows for explicit balancing of privacy and quality of the generated data.

6. Discussion

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

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

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

From Figure 5 followed that the number of patterns used to generate a transaction greatly influences privacy: more elements leads to higher anonymity. In the same line of thought, the candidate set can be filtered on pattern length; imposing a maximum length directly influences the number of patterns needed in generation, and can thus increase the provided anonymity.

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

Reconsidering our problem statement in Section 2, the KRIMP generator does a good job as solution for this PPDM problem. The concrete demands we posed for both the quality and privacy requirements are met, meaning that databases generated by our method are privacy and quality preserving as we interpreted this in our problem statement. Generating new data is therefore a good alternative to perturbing the original data.

Our privacy-preserving data generation method could be well put to practice in the distributed system Merugu and Ghosh [16] proposed: to cluster privacy-preserving data in a central place without moving all the data there, a privacy-preserving data generator for each separate location is to be built. This is exactly what our method can do and this would therefore be an interesting application.

Because the quality of the generated data is very high, the method could also be used in limited bandwidth distributed systems where privacy is not an issue. For each database that needs to be transported, construct a code table and communicate this instead of the database. If precision on the individual transaction level is not important, new highly similar data with the same characteristics can be generated.

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

8. Conclusions

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

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

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

11. References

[1] Aggarwal, C. C., and Yu. P. S. "A condensation approach to privacy preserving data mining", *Proc. EDBT*, 2004, pp.183-199.

[2] Agrawal, R., and Srikant, R. "Fast Algorithms for Mining Association Rules", *Proc. VLDB,* 1994, pp.487-499.

[3] Agrawal, R., and Srikant, R. "Privacy-preserving data mining", *Proc. SIGMOD,* 2000, pp.439-450.

[4] Agrawal, D., and Aggarwal, C.C. "On the design and quantification of privacy preserving data mining algorithms", *Proc. SIGMOD*, 2001, pp.247-255.

[5] Arik, F., Assaf, S., and Ran, W. "K-Anonymous Decision Tree Induction", *Proc. PKDD*, 2006, pp.151-162.

[6] Chen, K., and Liu, L. "Privacy Preserving Data Classification with Rotation Pertubation", *Proc. ICDM,* 2005, pp.589-592.

[7] Coenen, F. "The LUCS-KDD Discretised/normalised ARM and CARM Data Library", http://www.csc.liv.ac.uk/~frans/KDD/Software/, 2003.

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

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

[10] Huang, Z., Du, W., and Chen, B. "Deriving private information from randomized data", *Proc. SIGMOD,* 2005.

[11] Kargupta, H., Datta, S., Wang, Q., and Sivakumar, K. "Random-data perturbation techniques and privacy-preserving data mining", *Knowledge and Information Systems* 4(7), 2005, pp.387-414.

[12] Van Leeuwen, M., Vreeken, J., and Siebes, A. "Compression Picks Item Sets That Matter", *Proc. PKDD*, 2006, pp.585-592.

[13] Liew, C.K., Choi, U.J., and Liew, C.J. "A data distortion by probability distribution", *ACM Trans. Database Systems* 3(10), 1985, pp.395-411.

[14] Liu, K., Giannella, C., and Kargupta, H. "An Attacker's View of Distance Preserving Maps for Privacy Preserving Data Mining", *Proc. PKDD*, 2006, pp.297-308.

[15] Machanavajjhala, A., Gehrke, J., Kifer, D., and Venkitasubramaniam, M. "*l*-Diversity: Privacy Beyond *k*-Anonymity", *Proc. ICDE,* 2006, pp.24-35.

[16] Merugu, S., and Ghosh, J. "Privacy-preserving Distributed Clustering using Generative Models", *Proc. ICDM,* 2003, pp.211-218.

[17] Samarati, P. "Protecting respondents' identities in microdata release", *IEEE Trans. Knowledge and Data Engineering*, 2001, pp.1010-1027.

[18] Siebes, A., Vreeken, J., and Van Leeuwen, M. "Item Sets That Compress", *Proc. SIAM SDM,* 2006, pp.393-404.

[19] Vreeken, J., Van Leeuwen, M., and Siebes, A. "Characterising the Difference", *Proc. SIGKDD,* 2007.

[20] Vreeken, J., Van Leeuwen, M., and Siebes, A. "Preserving Privacy through Data Generation", *Proc. ICDM 2007*.