Smoothing Categorical Data

Arno Siebes and René Kersten

Universiteit Utrecht, The Netherlands
arno@cs.uu.nl, renegaa@hotmail.com

Abstract. Global models of a dataset reflect not only the large scale structure of the data distribution, they also reflect small(er) scale structure. Hence, if one wants to see the large scale structure, one should somehow subtract this smaller scale structure from the model.

While for some kinds of model – such as boosted classifiers – it is easy to see the "important" components, for many kind of models this is far harder, if at all possible. In such cases one might try an implicit approach: simplify the data distribution without changing the large scale structure. That is, one might first smooth the local structure out of the dataset. Then induce a new model from this smoothed dataset. This new model should now reflect the large scale structure of the original dataset. In this paper we propose such a smoothing for categorical data and for one particular type of models, viz., code tables.

By experiments we show that our approach preserves the large scale structure of a dataset well. That is, the smoothed dataset is simpler while the original and smoothed datasets share the same large scale structure.

1 Introduction

Most often data has structure across multiple scales. It is relatively easy to see fine-grained structure, e.g., through pattern mining. The lower the "support" of a pattern the more detailed structure it conveys. Since the large scale structure of data is convoluted with small scale structure it is, unfortunately, less easy to see its large scale structure. Simply focussing on "high support" patterns might miss large scale structure; we will show examples of this later.

Global models of the data, on the other hand, attempt to capture all, relevant, aspects of the data distribution. This often encompasses both global structure as well as local structure. If the type of model used is additive, such as a boosted classifier [5], the large scale structure may be approximated by disregarding small weight components. For non-additive types of models, syntactic operations that reveal the large scale structure are far less obvious.

This is unfortunate, since this large scale structure conveys important insight both in the data distribution and in the model. Hence, the problem we research in this paper: how to get insight in the large scale structure of a dataset?

If manipulating the model is difficult, manipulating the data might be easier. That is, *smoothing* the local structure – as described by the model – from the data while retaining the global structure – as described by the model. Inducing a new

P. Flach et al. (Eds.): ECML PKDD 2012, Part I, LNCS 7523, pp. 42–57, 2012.
© Springer-Verlag Berlin Heidelberg 2012

model from the smoothed dataset should then reveal the large scale structure of the data distribution as described by the original model. This is our approach.

Note that we should consistently write "structure of the data as described by the model", as we did above, but we simplify this to "structure of the data".

Smoothing is a well-known statistical technique [14] mostly for numerical data and to a lesser extend for ordered data. In this paper we focus on *categorical* data, for which as far as we are aware, no previous smoothing methods exist. Succinctly, our goal is to smooth out local structure from standard categorical data tables, while preserving large scale structure.

The structure of a categorical dataset is given by *item sets* [1]; in the context of categorical data, an item is defined as an attribute-value pair. The large scale structure is mostly – but not exclusively, as noted above – given by more frequent item sets. The small scale structure is given by less frequent item sets.

Informally, two equally sized datasets are indistinguishable if the support of all item sets is the same on both datasets. In the same vein, two equally sized datasets are similar if the support of most item sets is more or less the same on the two datasets. Again informally, we say that two equally sized datasets have the same large scale structure if they are similar as far as *frequent* item sets are concerned. This is made precise in Section 3.

To smooth the data while retaining the large scale structure, we need to use a model. For it is this model that describes the large scale structure we want to preserve. Hence, the way to smooth depends on the type of model. In this paper we focus on one particular class of models that consists of itemsets, viz. *code tables* as introduced in [11].

A code table consists of item sets associated with codes and can be used to encode a database; see Section 2. What is important here is that a code table implicitly encodes a probability distribution on all possible tuples in the database. By replacing less likely tuples with more likely tuples, the small scale structure is smoothed from the dataset while the large scale structure is maintained. That is, the support of most frequent item sets is not changed too much by these replacements; see Section 3.

Note that the reason why we don't simply use the set of all frequent item sets to do the smoothing is not only that this set is far too large. It also doesn't imply a straight forward distribution on all possible tuples; if only because of the interdependencies between the supports of different item sets.

In Figure 1 the effect of this smoothing is illustrated on the *mammals* dataset. This dataset consists of presence/absence records of European mammals[1] within geographical areas of 50x50 kilometres. In the left map, we depict the occurrence of the "item set" {European Hare, European Polecat, European Mole, Wild Boar}, with dark blue dots. The other dots denote the occurrence of variants of this set; variants that differ in one mammal. No dot means that neither the item set, nor any of these variants occur in that place.

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

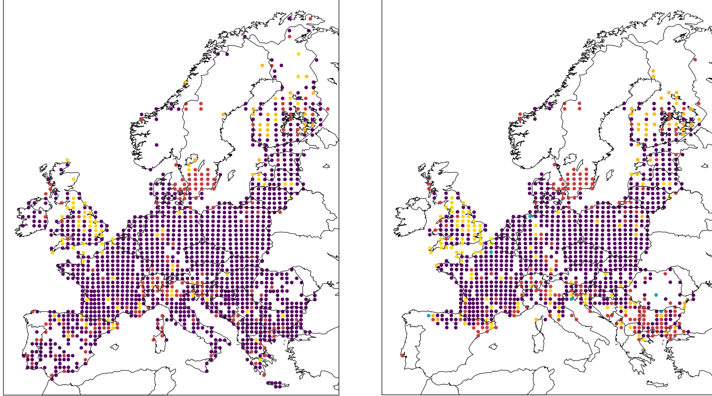

Fig. 1. The smoothed mammals dataset depicted on the right is far more homogeneous than the original mammals dataset depicted on the left

The right map shows the effect of smoothing. The map looks far more homogeneous: the Balkans changed from a variety of colours to almost uniformly dark blue. This does not happen in England and Sweden, but not because the map of the latter two looks more homogeneous than the map of the Balkans. Rather, this happens because the total set of mammals that occur in the Balkans resemble the set of mammals that occur in the rest of mainland Europe far more than the sets of mammals that occur in England and south Sweden do.

2 Preliminaries

2.1 Data and Patterns

We assume that the dataset is a standard rectangular table, well known from relational databases.That is we have a finite set of attributes $\mathcal{A} = \{A_1, \ldots, A_m\}$. Moreover, we assume that each attribute A_i has a finite, categorical, domain Dom_i. A tuple t over $\mathcal{A}$ is an element of $Dom = \prod_{i=1}^{m} Dom_i$. A database D over $\mathcal{A}$ is a bag of tuples over $\mathcal{A}$.

Since our work is rooted in item set mining [1], we will also use the terminology from that area. This means firstly that we will talk about transactions rather than tuples. Secondly we have a set of items $\mathcal{I} = \{I_1, \ldots I_n\}$. Each item I_j corresponds to an attribute-value pair (A_i, v_i^k), where $v_i^k \in Dom_i$. Note that this implies that all transactions have the same number of items.

A transaction t supports an item $I = (A_i, v_i^k)$ iff $t.A_i = v_i^k$. As usual, the support of an item set $J \subset \mathcal{I}$ in D, denoted by $sup_D(J)$, is defined as the number of transactions in D that support all items in J. Given a user defined threshold for support, denoted by *min-sup* or θ, an item set J is called *frequent* on D iff $sup_D(J) \geq \theta$. All frequent item sets can be found relatively efficiently [1].

2.2 Introducing Krimp

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

A transaction t is encoded by Krimp by searching for the first item set I in the code table for which $I \subseteq t$. The code for I becomes part of the encoding of t. If $t \setminus I \neq \emptyset$, the algorithm continues to encode $t \setminus I$. Since it is insisted that each code table contains at least all singleton item sets, this algorithm gives a unique encoding to each (possible) transaction over $\mathcal{I}$. The set of item sets used to encode a transaction is called its *cover.*

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 D. The *usage* of an item set $I \in CT$, denoted by $usage(I)$ is the number of transactions $t \in D$ which have I in their cover. That is, $usage(I) = |\{t \in D \mid I \in cover(t)\}|$.

For an $I \in CT$, the probability that I is used to encode an arbitrary $t \in D$, is simply the fraction of its usage, i.e.,

$$P(I \mid D) = \frac{usage(I)}{\sum_{J \in CT} usage(J)}$$

For optimal compression of D, the higher $P(I)$, the shorter its code should be. Given that a prefix code is necessary for unambiguous decoding, the well-known optimal Shannon code [4] is used. We now have the length of an item set I encoded with CT defined as $L(I \mid CT) = -\log(P(I \mid D))$.

The length of the encoding of a transaction is now simply the sum of the code lengths of the item sets in its cover:

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

The size of the encoded database is the sum of the sizes of the encoded transactions:

$$L(D \mid CT) = \sum_{t \in D} L(t \mid CT) = -\sum_{I \in CT} usage(I) \log\left(\frac{usage(I)}{\sum_{J \in CT} usage(J)}\right)$$

To find the best code table for a dataset, the Minimum Description Length (MDL) principle [6] is used. Which can be roughly described as follows.

Given a set of models $\mathcal{H}$, the best model $H \in \mathcal{H}$ is the one that minimises $L(H) + L(D \mid H)$, in which $L(H)$ is the length, in bits, of the description of H, and $L(D \mid H)$ is the length, in bits, of D encoded with H. One can paraphrase this by: the smaller $L(H) + L(D|H)$, the better H models D.

To apply MDL to code tables, we still need to define the size of a code table, as we previously did in [11]. We only count those item sets that have a non-zero usage. The size of the right-hand side column is obvious; it is 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 code table CT is given by:

$$L(CT \mid D) = \sum_{I \in CT : usage(I) \neq 0} L(I \mid ST) + L(I \mid CT)$$

An optimal code table is a code table which minimises:

$$L(D, CT) = L(CT \mid D) + L(D \mid CT)$$

Finally, $\mathcal{L}(D) = L(D, CT)$ for an optimal code table CT for D.

Unfortunately, computing an optimal code table is intractable [11], hence we introduced the heuristic algorithm Krimp. Krimp starts with a valid code table (only the collection of singletons) and a sorted list of candidates (frequent item sets). These candidates are assumed to be sorted descending on 1) support size, 2) item set length and 3) lexicographically. Each candidate item set is considered by inserting it at the right position in CT and calculating the new total compressed size. A candidate is only kept in the code table iff the resulting total size is smaller than it was before adding the candidate. If it is kept, all other elements of CT are reconsidered to see if they still positively contribute to compression; see [11].

3 The Problem

As stated in the Introduction, our goal is to make the large scale structure of a data distribution more visible by smoothing out small(er) scale structure. The formalisation of that goal rests on four considerations.

Consideration 1: A code table models the structure in the data by a very small subset of all (frequent) item sets, chosen because together they compress the database well. The usages of these item sets say something about their importance with respect to large scale structure, but not everything. High usage clearly means large scale structure, but low usage does *not* necessarily mean small scale structure. The reason for this is the order of the code table which is used to compute covers. In other words, to "see" the large scale structure of the dataset, it is not enough to simply focus on item sets with a high usage. We'll return to this observation later in this paper when we discuss our experiments.

Consideration 2: We claim that the structure of a categorical dataset is given by the support of all item sets; independent from the type of model used. The motivation for this claim is based on two observations.

The first is that if two equally sized datasets D_1 and D_2 over $\mathcal{I}$ have the same support for all item sets over $\mathcal{I}$, they are row permutations of each other. That is,

there exists a permutation π of the rows such that $\pi(D_1) = D_2$. This is obvious from the fact that the transactions in D_1 and D_2 are item sets themselves.

The second observation is that D_1 and D_2 will be indistinguishable by any type of statistical analysis [2]. For, all such analysis boils down to computing aggregates computed on subtables of a dataset. Given that these subtables correspond to item sets, equal support means equal results of the analysis.

Consideration 3: Statistical computations are, in general, robust. That is, small changes in the input yield small changes in the output. One reason for this is that two samples from the same distribution will invariably be subtly different; to be useful, statistical analysis should "smooth out" such differences.

In other words, if D_1 and D_2 have almost the same support for all item sets, statistical analysis will mostly imply that D_1 and D_2 are indistinguishable. For our purposes we can be even more tolerant, for we are willing – indeed aiming – to lose some (local) structure. That is, we are satisfied if for most item sets the support is almost the same. We can loosely formalise this by requiring that the support of a random item set is almost the same on both data sets.

Consideration 4: There are two weak points in this formulation. Firstly, that it is a statement about all item sets. Surely, if it is large scale structure we are interested in, item sets with (very) low support are unimportant. Secondly, what is almost the same?

Fortunately, these two weak points can be resolved in one step. The fact that we are not interested in low support item sets simply means that we are interested in *frequent* item sets. That is, item sets whose support is at least θ. If structure that is described by item sets with a support smaller than θ is deemed not interesting, we should also not worry too much about differences in support smaller than θ.

3.1 Formalising the Problem

Given all the previous, we have the following definition.

Definition 1. *Let D_1 and D_2 be two datasets over $\mathcal{I}$ and let $\epsilon, \delta \in \mathbb{R}_{\geq 0}$. D_2 is (ϵ, δ)-similar to D_1 if for a random item set I with $sup_{D_1}(I) \geq \delta$*

$$P(|sup_{D_1}(I) - sup_{D_2}(I)| \geq \delta) \leq \epsilon$$

If D_2 is to be a smoothed version of D_1, then we want it to be (ϵ, δ)-similar to D_1 for some $\epsilon, \delta \in \mathbb{R}_{\geq 0}$. But we also want D_2 to be simpler than D_1, i.e., we want D_2 to have a simpler code table than D_1.

In our MDL approach, it is easy to formalise what it means that D_1 is simpler than D_2. If D_1 and D_2 have the same size (the same number of transactions and, thus, the same number of items) and $\mathcal{L}(D_2) < \mathcal{L}(D_1)$, then D_2 is simpler than D_1. The reason is that, in this case, D_2 has less local structure that D_1. Local structure next to global structure makes a dataset harder to compress. Hence, we have the following definition.

Definition 2. *Let D_1 and D_2 be two equal sized (categorical) datasets over $\mathcal{I}$. D_2 is simpler than D_1 iff*

$$\mathcal{L}(D_2) < \mathcal{L}(D_1)$$

That a dataset D_2 which is both simpler than D_1 in this sense and (ϵ, δ)-similar to D_1, also has a simpler code table than D_1 is something only experiments can show.

Data Smoothing Problem

Let D be a dataset over $\mathcal{I}$ and let $\epsilon, \delta \in \mathbb{R}_{\geq 0}$. Moreover, let $\mathcal{D}_D^{(\epsilon,\delta)}$ be the set of all datasets over $\mathcal{I}$ that have the same number of transactions as D and are (ϵ, δ)-similar to D. Find a $D' \in \mathcal{D}_D^{(\epsilon,\delta)}$ that minimises $\mathcal{L}(D')$.

4 Introducing Smooth

Given that both the set of datasets with the same number of transactions as D and the set of code tables are finite, our problem is clearly decidable. However, given that finding an optimal code table for a dataset is already intractable [11], our current problem is also intractable. Hence we have to resort to heuristics.

For this heuristic we use an observation first made in [8]: a code table CT on a database D implicitly defines a probability distribution on the set of all possible tuples in the domain Dom of D. Let t be such an arbitrary tuple, then we can compress it with CT:

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

The one but last equation rests on the Naive Bayes like assumption that the item sets in a cover are independent. They are not(!), but in previous work this distribution has shown to characterise the data distribution on D very well [8,13]. Hence, we decided to use it here as well.

The main idea of the Smooth algorithm is to replace less likely tuples in D with more likely tuples, both according to this probability distribution. This strategy is based on the following observation: if D is changed into D' by replacing one $t \in D$ by a $t' \in Dom$ such that $L(t' \mid CT) < L(t \mid CT)$, then $L(D', CT) < L(D, CT)$. That is D' is simpler than D, according to CT.

However, if we replace an arbitrary $t \in D$ by an equally arbitrary $t' \in Dom$ which compresses better, there is no guarantee that D' will be (ϵ, δ)-similar to D for the given parameters ϵ and δ. Drastic changes could influence the support of many (frequent) item sets drastically, effectively disturbing the data distribution captured by the code table, not only on small scales, but also on large scales. To maintain the large scale balance we take two precautions:

Algorithm 1. SMOOTH(D, CT, ϵ, δ)

```
D' := D
F := frequent item sets, min-sup is δ
while D' is (ε, δ)-similar to D for F do
    choose t ∈ D' according to P_sel(t | CT)
    choose t' ∈ Var(t) according to P_var(t' | CT)
    D' := (D' \ {t}) ∪ {t'}
end while
return D'
```

- In one step we only consider modifications with edit distance one. That is, only one attribute-value is changed in one tuple. This set of variants of t is denoted by $Var(t)$.
- Both the selection of tuples to modify and their modification is random, where the choice is guided by the code table.
  - For the tuple selection, the probability of selecting $t \in D$, denoted by $P_{sel}(t \mid CT)$, is defined by

$$P_{sel}(t \mid CT) = \frac{(P(t \mid CT))^{-1}}{\sum_{t' \in D}(P(t \mid CT))^{-1}}$$

  - The probability of selecting an alternative $t' \in Var(t)$, denoted by $P_{var}(t')$, is defined by

$$P_{var}(t' \mid CT) = \frac{P(t' \mid CT)}{\sum_{t'' \in Var(t)} P(t'' \mid CT)}$$

We choose the tuple to replace at random using P_{sel} for two reasons. First, because in this way we "disturb" the data distribution as little as possible. Second, if we only attempt to replace tuples with a large encoded size, we more easily get stuck at a local optimum. We use P_{var} to select a variant at random, again because this disturbs the original data distribution as little as possible.

Neither of these two precautions guarantees that D' will be (ϵ, δ)-similar to D. They only heighten the probability that *one* replacement will result in a (ϵ, δ)-similar dataset. If we would let this replacement scheme run long enough, the resulting database would in most cases *not* be (ϵ, δ)-similar to D. For example, if $t \in D$ is unique in having the shortest encoded length, the replacement scheme, if left to run unbounded, would converge on a dataset that only contains copies of t. Therefore SMOOTH also checks whether D' is still (ϵ, δ)-similar to D.

SMOOTH is listed in algorithm 1. Apart from a dataset D and parameters ϵ and δ, it also takes a code table CT as input.

5 Experiments

In this section we report on two sets of experiments. The first set of experiments, on some well-known UCI datasets[2] (transformed using [3]), shows that SMOOTH

[2] http://archive.ics.uci.edu/ml/

achieves its goal. That is, it results in simpler datasets and, more importantly, simpler models that are still good models of the original dataset.

While these experiments shed some light on how SMOOTH achieves these goals, the second set of experiments is especially designed to do that. We take an artificial dataset which we corrupt by noise, and by examining how SMOOTH alters the noisy dataset we gain a deeper understanding of its doing.

In all experiments, we use KRIMP to approximate the optimal code table for dataset D and use that as input for SMOOTH.

5.1 UCI Data

For all experiments on the UCI datasets Iris, Pageblocks, Pima, Wine, Led7, and TicTacToe we used $\epsilon = 0.01$ and a minimal support, and thus δ, of 5. The experiments were also performed with $\delta = 1$ and $\delta = 10$, but given that the results are very similar we do not report on them here. For all experiments - except for the classification experiments - we report the averages of 50 repeats.

Table 1. Compressed sizes for $\epsilon = 0.01$, averaged over 50 repeats; standard deviation < 2%

Dataset	L(D, CT)	L(D', CT')	L(D', CT)
Iris	1685	1500	1572
Pageblocks	11404	10883	11031
Pima	9331	8652	8864
Wine	11038	10090	9980
Led7	30867	30664	30085
TicTacToe	29004	28575	27956

The results in Table 1 show that D' is indeed simpler than D. In all cases $L(D', CT')$ is significantly smaller than $L(D, CT)$, Moreover, CT is still a good code table for D', which can be seen from the fact that $L(D', CT') < L(D', CT) < L(D, CT)$.

Table 2 shows how many changes SMOOTH made to the original dataset before $\hat{\epsilon}$ (the measured value for ϵ) exceeded 0.01 for the first time. Also, it shows the value of $\hat{\epsilon}$ at that time. Only few changes lead to the simpler datasets shown in Table 1.

Table 2. The number of changes made and $\hat{\epsilon}$ when it first exceeded 0.01, averaged over 50 repeats

Dataset	# changes	$\hat{\epsilon}$
Iris	9.7 ± 1.1	0.014 ± 0.002
Pageblocks	81.2 ± 13.7	0.016 ± 0.005
Pima	34.8 ± 5.0	0.014 ± 0.003
Wine	162.9 ± 17.7	0.011 ± 0.001
Led7	52.8 ± 7.0	0.012 ± 0.001
TicTacToe	279.0 ± 23.8	0.011 ± 0.000

The effects of these changes is shown in Table 3. In that table we compare the number of non-singleton patterns in the code tables, the length of these patterns, and the percentage of the datasets covered by these patterns. In all cases, the number of non-singleton item sets in CT' is smaller than the number of of non-singleton item sets in CT. At the same time, the average size of these patterns goes up. That is, the code table has become simpler. The database has also become simpler, as the number of unique rows goes down. The joint effect of these two simplifications can be seen in the last two columns: the percentage of items in

Table 3. Comparing the number of non-singleton patterns in the code tables, the length of these patterns, the number of unique rows in the data sets, and the percentage of the datasets covered by these patterns; averaged over 50 repeats

	# of patterns		avg pattern length		unique rows		% of covered items	
Dataset	CT	CT'	CT	CT'	D	D'	CT on D	CT' on D'
Iris	14	13.3 ± 0.6	3.4	3.7 ± 0.1	42	35.6 ± 0.9	91	95
Pageblocks	43	32.1 ± 1.4	8.3	10.3 ± 0.2	72	50.5 ± 2.4	100	100
Pima	56	48.9 ± 2.2	4.9	5.7 ± 0.2	170	157.7 ± 2.1	96	97
Wine	60	53.8 ± 3.0	3.5	3.8 ± 0.2	177	176.9 ± 0.2	67	72
Led7	153	148.3 ± 5.8	6.6	6.9 ± 0.1	326	298.8 ± 4.2	98	99
TicTacToe	159	155.0 ± 6.7	4.0	4.1 ± 0.1	958	905.2 ± 6.3	90	91

the database that is covered by non-singleton item sets from the code table goes up, although not by very much.

Both the compression results in Table 1 and the measured ϵ value for $\delta = 5$ in Table 2 indicate that D' has a data distribution very similar to D. For an independent verification of this claim we also performed classification experiments with these code tables. The basic set-up is the same as in [8], with 25-fold cross-validation. Each class-database was individually smoothed with SMOOTH ($\epsilon = 0.01$); the test data was, of course, *not smoothed*. The results are in Table 4.

The first observation is that for PageBlocks, Pima, Led7, and TicTacToe, the classification results of the simplified code table are on par with those of the original code table. Moreover, for all but Pima, these results are way above the baseline scores (assign the tuple to the largest class). The somewhat disappointing result on Pima is caused by the fact that it has small classes that are too small to learn well using MDL.

The second observation is that the degradation in classification performance is far larger for Iris and Wine. Again this is caused by dataset size, both Iris and Wine are small datasets. Moreover, even the degraded scores are way above baseline.

Table 4. Classification accuracy on independent original data, 25-fold cross-validation.

	Classification accuracy		
Dataset	Baseline	CT	CT'
Iris	33.3	94.7 ± 9.3	90.0 ± 15.2
Pageblocks	69.8	92.5 ± 1.5	92.3 ± 1.4
Pima	65.1	69.5 ± 6.9	69.5 ± 9.1
Wine	39.3	91.6 ± 12.0	79.9 ± 21.0
Led7	11.0	73.8 ± 3.5	74.1 ± 3.2
TicTacToe	65.3	87.8 ± 7.3	80.4 ± 8.4

Note that these results by no means imply that SMOOTH induces a state-of-the-art classifier. Rather, the results reconfirm that SMOOTH yields characteristic code tables of the *original* dataset, i.e., structure is preserved.

A classification scheme based on code tables might be biased towards SMOOTH. After all, one of the design goals of SMOOTH was not to change code tables too much. To verify that the original data distribution is not changed too much, we also performed these classification experiments with some well-known algorithms as implemented in Weka [7]. The set-up is the same as with the SMOOTH

Table 5. Classification accuracy before and after smoothing. In all cases, $\epsilon = 0.01$, $\delta \in \{1, 5, 10\}$, and 'orig' denotes the original (non-smoothed) dataset

Dataset - δ	C4.5	Ripper	LR	NB	SVM
Iris-orig	92.67 ± 9.66	92.67 ± 9.66	90.00 ± 13.05	94.00 ± 7.98	92.00 ± 10.80
Iris-1	92.67 ± 9.66	94.00 ± 9.66	92.00 ± 9.84	94.67 ± 6.89	92.00 ± 10.80
Iris-5	98.00 ± 6.32	98.00 ± 6.32	96.00 ± 6.44	94.67 ± 6.89	94.67 ± 7.57
Iris-10	95.33 ± 7.06	95.33 ± 7.06	94.67 ± 6.13	94.67 ± 6.13	94.67 ± 6.13
Led7-orig	75.19 ± 2.90	72.09 ± 3.82	75.63 ± 3.31	75.59 ± 2.81	75.91 ± 2.94
Led7-1	75.16 ± 2.83	71.88 ± 3.14	75.63 ± 3.32	75.50 ± 2.90	75.91 ± 2.96
Led7-5	75.28 ± 4.07	72.31 ± 3.56	75.59 ± 3.92	75.47 ± 3.83	75.75 ± 4.01
Led7-10	74.94 ± 3.66	72.78 ± 4.06	74.78 ± 4.11	75.28 ± 4.05	75.38 ± 4.11
Pageblocks-orig	92.64 ± 1.42	92.57 ± 1.36	92.75 ± 1.49	92.68 ± 1.45	91.91 ± 1.67
Pageblocks-1	92.66 ± 1.40	92.51 ± 1.32	92.75 ± 1.43	92.66 ± 1.47	91.91 ± 1.68
Pageblocks-5	92.64 ± 2.14	92.51 ± 2.17	92.69 ± 2.14	92.60 ± 2.14	91.80 ± 2.30
Pageblocks-10	92.68 ± 1.58	92.57 ± 1.64	92.60 ± 1.65	92.66 ± 1.66	91.82 ± 1.60
Pima-orig	74.58 ± 3.76	73.29 ± 5.14	72.52 ± 4.85	74.32 ± 4.70	73.55 ± 5.82
Pima-1	74.58 ± 3.76	73.81 ± 4.69	72.39 ± 4.93	74.19 ± 4.66	73.68 ± 5.79
Pima-5	74.32 ± 6.00	74.06 ± 6.42	72.77 ± 6.97	74.32 ± 6.68	73.68 ± 6.22
Pima-10	74.84 ± 7.57	74.06 ± 6.81	73.55 ± 8.43	74.58 ± 7.17	73.16 ± 7.32
TicTacToe-orig	85.21 ± 3.92	97.92 ± 1.96	98.02 ± 2.11	70.21 ± 4.23	87.71 ± 4.39
TicTacToe-1	85.00 ± 6.04	97.92 ± 1.70	98.12 ± 1.61	70.21 ± 4.00	87.08 ± 3.78
TicTacToe-5	84.17 ± 3.64	98.33 ± 1.22	97.50 ± 1.32	70.94 ± 3.31	84.90 ± 4.34
TicTacToe-10	81.15 ± 3.91	97.08 ± 2.29	96.88 ± 1.90	71.04 ± 3.50	82.92 ± 3.71
Wine-orig	90.56 ± 6.44	88.89 ± 6.93	92.78 ± 6.95	95.00 ± 4.86	87.78 ± 5.74
Wine-1	85.56 ± 9.15	87.22 ± 5.89	93.89 ± 8.47	95.00 ± 4.86	91.11 ± 5.97
Wine-5	86.11 ± 5.40	88.89 ± 11.11	91.11 ± 7.94	93.89 ± 4.10	88.33 ± 7.15
Wine-10	82.22 ± 9.37	84.44 ± 8.61	91.67 ± 6.00	93.89 ± 8.05	87.78 ± 10.08

based classification, that is we did 25-fold cross-validation, running SMOOTH on each train set, while, again, *the test set was not smoothed*. Table 5 shows the results for C4.5, Ripper, Logistic Regression, Naive Bayes, and Support Vector Machines, each with their default settings in Weka.

The important observation is that the accuracy doesn't change significantly when the dataset is smoothed. Whether δ is set to 1, 5, or 10, the difference in accuracy is not significantly different from the baseline; where the baseline is the accuracy of the algorithm on the original (non-smoothed) dataset. This is true for the rule-based classifiers C4.5 and Ripper, for the instance-based classifier NB, for the traditional statistical classifier Logistic Regression, and for the SVM classifier. None of these classifiers detects a significant difference between the original data distribution and the smoothed data distribution.

There is one notable exception to this observation: C4.5 on Wine. There we see a notable, significant, drop in accuracy. While inspecting the resulting trees, we noticed serious overfitting. Using the default settings only takes you so far.

A natural question with these results is: do the other classifiers also become simpler on the smoothed datasets? While this was not a design goal for SMOOTH - after all, it is a *model-driven* approach - we inspected this for the Ripper results.

The average number of rules goes down while the average length of the rules goes up slightly. This is very similar to the changes we encountered earlier for the code table elements, which also became fewer and longer.

5.2 Artificial Data

The results of the previous subsection already show some of SMOOTH's effects, but even more insight can be gained when we know the ideal result beforehand.

For this, we created an artificial dataset as follows. We generated 50 unique tuples randomly (over 7 attributes, each with a domain from 4 to 18 values). Each of these 50 tuples was duplicated between 10 and 30 times randomly, such that the end result was a dataset with 1000 tuples. This is the Clean dataset. We then ran KRIMP on Clean (with min-sup=1), and the result was - unsurprisingly - a code table with 50 item sets, one for each unique tuple in Clean.

For the experiments, we randomly replace items in the dataset with others (of course, while still adhering to the attributes' domains). The items to be replaced are chosen uniformly. We vary the amount of error to investigate the smoothing capability of SMOOTH in comparison to the amount of noise in the data.

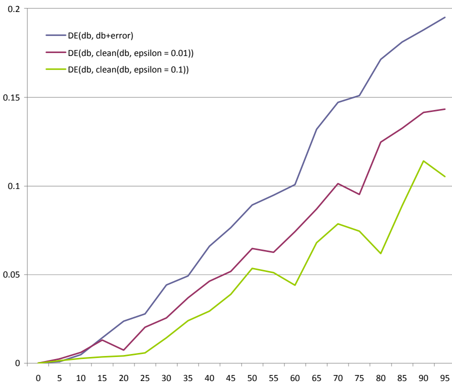

Fig. 2. The x-axis depicts the number of errors made on Clean, the y-axis gives $\hat{\epsilon}$, i.e, the probability of an error larger than δ

Note that we do *not* perform this experiment to show that SMOOTH is good at removing noise. After all, data points that are considered noise in one setting may be considered perfectly valid data in another setting. However, in our artificial setting noise and local structure happen to coincide and we know exactly what the noise is. Hence, we here use SMOOTH as a noise remover, because it allows us to investigate how it removes local structure.

With this proviso, SMOOTH is run to try to "clean up" the data. The parameters are: $\delta = 1$, and $\epsilon = 0.01$ and 0.1 respectively. The algorithm is automatically stopped after 1000 iterations, because for low amounts of error added, epsilon can never even attain 0.01; for larger amounts of noise it can, of course. Performance is measured by comparing the noisy and smoothed datasets to the original Clean dataset in three different ways.

The first measure is simply the measured epsilon, i.e. $\hat{\epsilon}$, for $\delta = 1$. This to make sure we can see the 'noise' being added and subsequently removed. The obtained measurements are graphed in Figure 2. Results are depicted for three more or less noisy datasets, the first being the noisy dataset itself. Obviously, this has consistently the highest $\hat{\epsilon}$. The other two datasets are smoothed versions

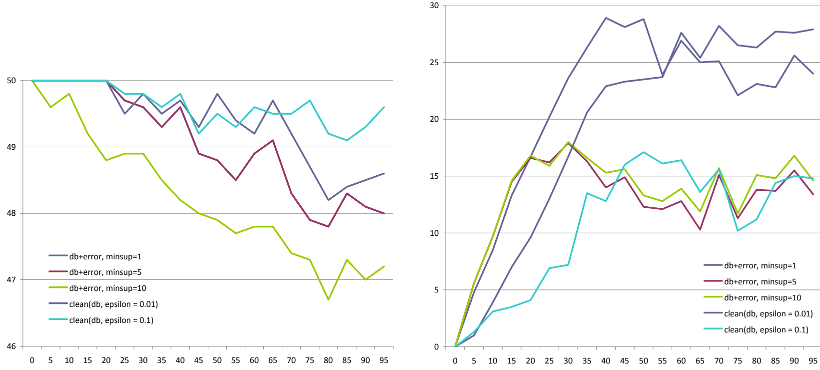

Fig. 3. In the figure on the left $|CT' \cap CT|$ is graphed for various datasets. In the figure on the right $|CT' - CT|$ is graphed for the same datasets. In both, the x-axis depicts the number of errors made on Clean.

of the noisy dataset, with ϵ set to 0.01 resp. 0.1. Note that the ϵ parameter is set with regard to the *noisy* dataset and thus determines how far SMOOTH can go away from the noisy dataset. The two graphs show that SMOOTH uses this extra wriggle-room to get closer to the Clean dataset: the graph for $\epsilon = 0.1$ is consistently better than the graph for $\epsilon = 0.01$.

Note, however, that the algorithm is not able to drop $\hat{\epsilon}$ on the cleaned dataset to 0, even if $\hat{\epsilon}$ between the clean and the cluttered dataset is less than the ϵ parameter. This is because there is always a chance that the algorithm will select and adjust a row with no noise on it. However, the difference between the low chance of adjusting a clean row in a bad way and the high chance of adjusting a noisy row in a good way ensures that $\hat{\epsilon}$ will still drop significantly; in Figure 2 the error is halved!

In Figure 3 the other two measures are graphed, both versus the number of errors on Clean across different datasets. The left figure depicts the number of the original patterns left (50 is ideal). This is the part of the original structure that can still be seen in CT' and we denote this by $|CT' \cap CT|$. The right figure depicts the number of wrongfully added patterns in CT'. These patterns are present in CT' but not in CT; they clutter up the code table and therefore obscure the true structure in the data. We denote this measure by $|CT' - CT|$.

Firstly note that adding more noise to the data will cause more of the good patterns to be 'broken up' into smaller patterns. Therefore $|CT' \cap CT|$ drops and $|CT' - CT|$ rises. Secondly note that increasing the minimal support, i.e, δ, will filter those smaller patterns out. These patterns are simply not frequent enough and thus they cannot end-up in the code table. But for this reason, we also cannot recover the patterns in the Clean dataset that were lost, as can be seen in the left figure. As in Figure 2, $\epsilon = 0.1$ outperforms $\epsilon = 0.01$ for both measures and for the same reason: SMOOTH needs some leeway to remove the unwanted, local structure.

6 Discussion

The reason to introduce the SMOOTH algorithm was to smooth the local structure from the data while retaining the global structure, such that a simpler, but still characteristic, code table could be mined from this smoothed dataset. The results from the previous section show that that goal has been reached.

Firstly, both the code table and the database get simpler when SMOOTH is applied. This is, e.g., clear from Table 3. The new code table has fewer, larger, item sets and the dataset has fewer unique tuples. Moreover, the new code table describes the structure of the new database better for a larger number of items is covered by non-singleton item sets. This latter fact is also witnessed by Table 1, as CT' compresses D' better than CT compresses D.

Secondly, D' - and thus CT' - retains important structure from D. This is, again, witnessed by Table 1: CT compresses D' better than D. Moreover, Table 2 shows that SMOOTH achieves these goals for a modest epsilon, implying that D and D' are almost indistinguishable. Further witness that D and CT' retain the important structure in D and CT is given by Table 4 and Table 5. Both show that training on D and D' leads to classifiers statistically indistinguishable on an independent test from D, for a wide variety of classification algorithms! Moreover, Table 4 shows that CT' retains the important structure in D, for CT' is almost as good as CT in classifying tuples from the original distribution D.

Finally, SMOOTH achieves these results by removing local structure. Figure 2 shows this very well. For a dataset with 10% noise, SMOOTH with $\epsilon = 0.1$ (measured against the noisy dataset) achieves $\hat{\epsilon} = 0.1$ with regard to the Clean dataset. The number of errors with regard to the Clean dataset is halved!

An even stronger witness for the claim that large scale structure is retained while local structure is removed is given by Figure 3. These two graphs show firstly that SMOOTH not only retains global structure, it even recovers structure that is present in the original dataset, but not visible in the corrupted dataset. Secondly, these graphs show that local structure is indeed removed, as the number of wrongfully added patterns goes down.

So, SMOOTH achieves its results very well. The reader might, however, wonder if these results couldn't be achieved easier. One easier way is to simply mine with a larger minimal support, after all the large scale structure is mostly given by (very) frequent item sets. Unfortunately, this doesn't work. Figure 3 shows that a higher minimal support means that fewer original patterns are recovered; we may miss large scale structure that is obfuscated by local structure.

If concentrating on (very) frequent item sets doesn't work, a second simpler strategy might seem to concentrate on item sets with a high usage. That is, simply smooth out the low usage patterns from the code table by modifying only tuples with a large code size. In fact, this strategy suffers from the same problem as the first alternative; Figure 3 shows that original large scale structure may be obfuscated by low usage item sets.

Note that the latter observations also illustrate the importance of the problem solved in this paper: if we do not smooth the data, there may be large scale structure in the data that is simply not apparent from the code table.

7 Related Work

Data smoothing is a research area with a rich history in areas such as statistics and image processing, but also in e.g. signal processing where it is known as filtering. Giving an overview of this vast field is far outside the scope of this paper. A good introductory book from the viewpoint of Statistics is [14].

Smoothing usually refers to continues operations. If the data is real valued, smoothing is often performed by convoluting the data with some distribution. Even if the data is discrete (but still numerical), convolution is often the weapon of choice. For ordered categorical data, a Poisson regression model with log-likelihood may be used [12]. We are not aware of any papers that address general categorical data and/or take an approach similar to ours. The big difference is that convolution – and regression – always involve all data, where there may be many transactions in a dataset that SMOOTH doesn't even touch.

Within the field of pattern mining, our approach is related to fault tolerant patterns [9]. Roughly speaking, these are patterns which with some small modifications to the data would get a larger support. We do discover such patterns, that is what the modifications that SMOOTH makes to the database imply. There is, however, a large difference in aim. Fault tolerant pattern miners want to find all fault tolerant patterns. We want to discover the global structure of the data and discover some fault tolerant patterns as a by product.

In our own research, our paper on missing data [13] is related to this paper. In there we design three algorithms that complete a database with missing data. Like SMOOTH, these algorithms use a probability distribution on variants. There is one major difference, though. With missing data, one knows exactly where the problem is. Thus, the imputation algorithms introduced in [13] do not have to select "which tuples to modify where" and can run until convergence. In the current case, unfortunately, we do not know where the problem is. Only by slowly massaging the data do we discover its, sometimes hidden, large scale structure.

Related in goal, but not algorithmically is our introduction of a structure function in [10]. In that paper we introduce a series of models that capture ever finer details of the data distribution. The first model looks at a very coarse scale, while the final model looks at a very fine scale. The major difference with SMOOTH is that in [10] the dataset is not changed. This means that the structure function approach will not uncover structure which is hidden by noise. That is, while SMOOTH is – to a certain level – able to reconstruct the original dataset from a corrupted variant, the structure function of [10] is not able to do that.

8 Conclusions

The observation that triggered the research reported on in this paper is that it is often hard to understand the large scale structure of the data from a model. The approach we take is that we smooth the local structure from the dataset – guided by the model – while retaining the large scale structure. A model induced from this smoothed dataset reflects the large scale structure of the original data.

While smoothing is a well known for numerical and ordered data, this paper introduces a smoothing algorithm, called Smooth, for categorical data.

Smooth uses code tables such as, e.g., generated by Krimp to smooth the data. It smoothes the data by gradually modifying tuples in the database. While smoothing it ensures that the large scale structure is maintained, i.e., that the support of most frequent item sets remains the same. At the same time it ensures that the data set becomes simpler, i.e., that it compresses better.

The experiments show that Smooth works well. Both the smoothed data set and its code table are simpler than the originals. Moreover, both datasets give more or less the same support to most item sets. Hence, both datasets have more or less the same structure. This is further corroborated by the fact that the original dataset and the smoothed dataset lead to equally good classifiers for an independent test set of the original(!) dataset. This observation was shown to hold for a large variety of classification algorithms.

Finally, experiments on artificial data, for which we know the ideal outcome, show that Smooth does what it is supposed to do. The large scale structure is retained while local structure is removed. Even if the large scale structure is hidden by local structure it may be recovered by Smooth.

References

1. Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., Verkamo, A.I.: Fast discovery of association rules. In: Advances in Knowledge Discovery and Data Mining, pp. 307–328. AAAI (1996)
2. Agresti, A.: Categorical Data Analysis, 2nd edn. Wiley (2002)
3. Coenen, F.: The LUCS-KDD discretised/normalised (C)ARM data library (2003)
4. Cover, T., Thomas, J.: Elements of Information Theory, 2nd edn. Wiley (2006)
5. Freund, Y., Schapire, R.E.: A decision-theoretic generalization of on-line learning and an application to boosting. Journal of Computer and System Sciences 55(1), 119–139 (1997)
6. Grünwald, P.D.: Minimum description length tutorial. In: Grünwald, P., Myung, I. (eds.) Advances in Minimum Description Length. MIT Press (2005)
7. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: Weka data mining software: An update. SIGKDD Explorations 11 (2009)
8. van Leeuwen, M., Vreeken, J., Siebes, A.: Compression Picks Item Sets That Matter. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) PKDD 2006. LNCS (LNAI), vol. 4213, pp. 585–592. Springer, Heidelberg (2006)
9. Pei, J., Tung, A.K.H., Han, J.: Fault tolerant pattern mining: Problems and challenges. In: DMKD (2001)
10. Siebes, A., Kersten, R.: A structure function for transaction data. In: Proc. SIAM Conf. on Data Mining (2011)
11. Siebes, A., Vreeken, J., van Leeuwen, M.: Item sets that compress. In: Proc. SIAM Conf. Data Mining, pp. 393–404 (2006)
12. Simonoff, J.S.: Three sides of smoothing: Categorical data smoothing, nonparametric regression, and density estimation. International Statistical Reviews /Revue Internationale de Statistique 66(2), 137–156 (1998)
13. Vreeken, J., Siebes, A.: Filling in the blanks - krimp minimization for missing data. In: Proceedings of the IEEE International Conference on Data Mining (2008)
14. Wand, M., Jones, M.: Kernel Smoothing. Chapman & Hall (1994)