Fast and Reliable Anomaly Detection in Categorical Data

Leman Akoglu
CMU, SCS
5000 Forbes Ave
Pittsburgh PA 15213
lakoglu@cs.cmu.edu

Hanghang Tong
IBM T. J. Watson
19 Skyline Drive
Hawthorne, NY 10532
htong@us.ibm.com

Jilles Vreeken
University of Antwerp
Middelheimlaan 1
B2020 Antwerp, Belgium
jilles.vreeken@ua.ac.be

Christos Faloutsos
CMU, SCS
5000 Forbes Ave
Pittsburgh PA 15213
christos@cs.cmu.edu

ABSTRACT

Spotting anomalies in large multi-dimensional databases is a crucial task with many applications in finance, health care, security, etc. We introduce COMPREX, a new approach for identifying anomalies using pattern-based compression. Informally, our method finds a collection of dictionaries that describe the *norm* of a database succinctly, and subsequently flags those points dissimilar to the norm—with high compression cost—as anomalies.

Our approach exhibits four key features: 1) it is *parameter-free*; it builds dictionaries directly from data, and requires no user-specified parameters such as distance functions or density and similarity thresholds, 2) it is *general*; we show it works for a broad range of complex databases, including graph, image and relational databases that may contain both categorical and numerical features, 3) it is *scalable*; its running time grows linearly with respect to both database size as well as number of dimensions, and 4) it is *effective*; experiments on a broad range of datasets show large improvements in both compression, as well as precision in anomaly detection, outperforming its state-of-the-art competitors.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data mining; E.4 [**Coding and Information Theory**]: Data compaction and compression

Keywords

anomaly detection, categorical data, data encoding

1. INTRODUCTION

Detecting anomalies and irregularities in data is an important task, and numerous applications exist where anomaly detection is vital, e.g. detecting network intrusion, credit card fraud, insurance claim fraud, and so on. In addition to revealing suspicious behavior, anomaly detection is useful for spotting rare events such as rare diseases in medicine, in addition to data cleaning and filtering. Finding anomalies, however, is a difficult task, in particular for complex multi-dimensional databases.

In this work, we address the problem of anomaly detection in multi-dimensional categorical databases using pattern-based compression. Compression-based techniques have been explored mostly in communications theory, for reduced transmission cost and increased throughput and in databases, for reduced storage cost and increased query performance. Here, we improve over recent work that identified compression as a natural tool for spotting anomalies [22]. Simply put, we define the *norm* by the patterns that compress the data well. Then, any data point that can not be compressed well is said not to comply with the norm, and thus is *abnormal*.

The heart of our method, COMPREX, is to use a set of dictionaries to encode a database. We exploit correlations between the features in the database, grouping those with high information gain, and build dictionaries (also look-up tables or code tables [25]) for each group of strongly interacting features. Informally, these dictionaries capture the data distribution in terms of patterns; the more often a pattern occurs, the shorter its encoded length. The goal is to find the optimal set of dictionaries that yield the minimal lossless compression, and then spot tuples with long encoded lengths.

Dictionary based compression has been shown to be highly effective for anomaly detection in [22], which employs the KRIMP itemset-based compressor introduced by [21]. Besides high performance, it allows for characterization: one can easily inspect how tuples are encoded, and hence why one is deemed an anomaly. In this paper we show that our COMPREX can *do better*, yielding *both* better compression (and relatedly, higher detection accuracy) *as well as* lower running time.

The intuition behind our method achieving better compression is that it uses *multiple* code tables to describe the data, whereas KRIMP builds a *single* code table. As such, our method can better exploit correlations between *groups* of features, as by doing away with uncorrelated features it can locally assign codes more effectively. Moreover, we build code tables directly from data in a bottom-up fashion, instead of filtering very large collections of pre-mined candidate patterns, which becomes exponentially costly with increasing database dimension, i.e. number of features.

Furthermore, by not requiring the user to provide a collection of candidate itemsets, nor a minimal support threshold, COMPREX is parameter free in both theory and practice. We employ the Minimum Description Length principle to automatically decide the number of feature groups, which features to group, what patterns to include in the code tables, as well as to point out anomalies. In contrast, most existing anomaly detection methods have several parameters, such as the choice of a similarity function, density or distance thresholds, number of nearest neighbors, etc.

In a nutshell, we improve over the state of the art by encoding data using multiple code tables, instead of one—allowing us to better grasp strongly interacting features. Moreover, we build our models directly from data—avoiding the expensive step of mining and filtering large collections of candidate patterns.

CIKM'12, October 29–November 2, 2012, Maui, HI, USA.

Experiments show the resulting models obtain high performance in anomaly detection, improving greatly over the state of the art for categorical data. We further show COMPREX is very generally applicable; after discretization it matches the state of the art for numerical data, and correctly identifies anomalies both in translated large graphs and image data.

2. PRELIMINARIES AND BACKGROUND

In this section, we give the preliminaries and notation, and introduce basic concepts used throughout the paper.

2.1 Preliminaries

In this paper we consider categorical databases. A database D is a bag of n tuples over a set of m categorical features $\mathcal{F} = \{f_1, \ldots, f_m\}$. Each feature $f \in \mathcal{F}$ has a domain $dom(f)$ of possible values $\{v_1, v_2, \ldots\}$. The number of values $v \in dom(f)$ is the *arity* of f, i.e. $arity(f) = |dom(f)| \in \mathbb{N}$.

The domains are distinct between features. That is, $dom(f_i) \cap dom(f_j) = \emptyset, \forall i \neq j$. The domain of a feature set $F \subseteq \mathcal{F}$ is the Cartesian product of the domains of the individual features $f \in F$, i.e., $dom(F) = \prod_{f \in F} dom(f)$.

A database D is simply a collection of n tuples, where each tuple t is a vector of length m containing a value for each feature in $\mathcal{F}$. As such, D can also be regarded as a n-by-m matrix, where the possible values in a column i are determined by $dom(f_i)$.

An *item* is a feature-value pair $(f = v)$, with $f \subseteq \mathcal{F}$, and $v \in dom(f)$. A itemset is then a pair $(F = v)$, for a set of features $F \subseteq \mathcal{F}$, and $v \in dom(F)$ is a vector of length $|F|$. We typically refer to an itemset as a *pattern*.

A tuple t is said to contain a pattern $(F = v)$, denoted as $p(F = v) \subseteq t$ (or $p \subseteq t$ for short), if for all features $f \in F$, $t_f = v_f$ holds. The support of a pattern $(F = v)$ is the number of tuples in D that contain it: $supp(F = v) = |\{t \in D \mid (F = v) \subseteq t\}|$. Its frequency is then $freq(F = v) = supp(F = v)/|D|$.

Finally, the entropy of a feature set F is defined as

$$H(F) = -\sum_{v \in dom(F)} freq(F = v) \log freq(F = v) \ .$$

All logarithms are to base 2, and by convention, $0 \log 0 = 0$.

2.2 The MDL principle

The Minimum Description Length (MDL) principle [20] is a practical version of Kolmogorov Complexity [16], and can be regarded as model selection based on lossless compression.

Given a set of models $\mathcal{M}$, MDL identifies the best model $M \in \mathcal{M}$ as the one that minimizes

$$L(M) + L(D \mid M) \ ,$$

in which $L(M)$ is the length in bits of the description of the model M, and $L(D \mid M)$ is the length of the description of the data encoded by M. That is, the MDL-optimal model for a database D encodes D most succinctly among all possible models; it provides the best possible lossless compression.

MDL provides us a systematic approach for selecting the model that best balances the complexity of the model and its fit to the data. While models that are overly complex may provide an arbitrarily good fit to the data and thus have low $L(D \mid M)$, they overfit the data, and are penalized with a relatively high $L(M)$. Overly simple models, on the other hand, have very low $L(M)$, but as they fail to identify important structure in D, their corresponding $L(D \mid M)$ tends to be relatively high. As such, the MDL-optimal model provides the best balance between model complexity and goodness of fit.

2.3 Dictionary based compression

To use MDL, we need to define what a model is, how to encode the data with such a model, and how to encode a model.

2.3.1 Data encoding

As models, we will use code tables. A code table is a simple two column table. The first column contains $pattern$s, which are ordered descending (1) first by length and (2) second by support. The second column contains the code words $code(p)$ corresponding to each pattern p. An illustrative database with 6 tuples and an example code table for the database is illustrated in Table 1.

Table 1: An illustrative database D and an example code table CT for a set of three features, F=$\{f_1, f_2, f_3\}$.

Data	*Code Table*			
$f_1 f_2 f_3$	$p(F = v)$	$code(p)$	$usage(p)$	$L(code(p))$
a b x	a b x	0	4	1 bit
a b x	a c	10	2	2 bits
a b x	x	110	1	3 bits
a b x	y	111	1	3 bits
a c x				
a c y				

The code words in the second column of a code table CT are not important: their lengths are. The length of a code word for a pattern depends on the database we want to compress. Intuitively, the more often a pattern occurs in the database, the shorter its code should be. The *usage* of a pattern $p \in CT$ is the number of tuples $t \in D$ which contain p in their *cover*, i.e. have the code of p in their encoding.

The encoding of a tuple t, given a CT, works as follows: the patterns in the first column are scanned in their predefined order to find the first pattern p for which $p \subseteq t$. p is then said to be *used* in the *cover* of t , and the corresponding code word for p in the second column becomes part of the encoding of t. If $t \setminus p \neq \emptyset$, the encoding continues with $t \setminus p$ until t is completely covered, which yields a unique set of patterns that form the *cover* of t.

Given the usages of the patterns in a CT, we can compute the lengths of the code words for the optimal prefix code [20]. Shannon entropy gives the length for the optimal prefix code for p:

$$L(code(p) \mid CT) = -\log\big(\frac{usage(p)}{\sum_{p' \in CT} usage(p')}\big).$$

The number of bits to encode a tuple t is simply the sum of the code lengths of the patterns in its cover, that is,

$$L(t \mid CT) = \sum_{p \in cover(t)} L(code(p) \mid CT).$$

The total length in bits of the encoded database is then the sum of the lengths of the encoded data tuples,

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

2.3.2 Model encoding

To find the MDL-optimal compressor, we also need to determine the encoded size of the model, the code table in our case. Clearly, the size of the second column in a given code table CT that contains the prefix code words $code(p)$ is trivial; it is simply the sum of their lengths. For the size of the first column, we need to consider all the singleton items $\mathcal{I}$ contained in the patterns, i.e., $\mathcal{I} = \bigcup_{f \in \mathcal{F}} dom(f)$.

For encoding the patterns in the left hand column we again use an optimal prefix code. We first compute the frequency of their appearance in the first column, and then by Shannon entropy calculate the optimal length of these codes. Specifically, the encoding of the first column in a code table requires $cH(P)$ bits, where c is the total count of singleton items in the patterns $p \in CT$, $H(.)$ denotes the Shannon entropy function, and P is a multinomial random variable with the probability $p_i = \frac{r_i}{c}$ in which r_i is the number of occurrences of singleton item $i \in \mathcal{I}$ in the first column (for the actual items, one could add an ASCII table providing the matching from the prefix codes to the original names. Since all such tables are over $\mathcal{I}$, this only adds an additive constant to the total cost). All in all,

$$L(CT) = \sum_{p \in CT} L(code(p) \mid CT) + \sum_{i \in \mathcal{I}} -r_i \log(p_i).$$

3. PROPOSED METHOD

3.1 Compression with Set of Tables: Theory

In the previous section, we showed how to encode a database D using a *single* code table CT. In fact, this is the approach introduced in [21] to compress transaction databases, employing frequent itemset mining to generate the candidate patterns for the code table. Here, we do not regard transaction data, but regular relational data, where tuples are data points in a multi-dimensional categorical feature space. In this space, some groups of features may be highly correlated; and hence may be compressed well together.

As a result we can *improve* by using multiple code tables—as we can then exploit correlations, and build a separate, probably smaller CT for each highly correlated group of features instead of a single, large CT for all, possibly uncorrelated features. As such, using *multiple*, that is a *set* of code tables, and mining these efficiently (bypassing the costly frequent itemset mining), are two of the main contributions of our work.

Next, we formally introduce the concept of feature partitioning, and give our problem statement.

DEFINITION 1. *A feature partitioning $\mathcal{P} = \{F_1, \ldots, F_k\}$ of a set of features $\mathcal{F}$ is a grouping of $\mathcal{F}$, for which (1) each partition contains one or more features: $\forall F_i \in \mathcal{P} : F_i \neq \emptyset$, (2) all partitions are pairwise disjoint: $\forall i \neq j : F_i \cap F_j = \emptyset$, and (3) every feature belongs to a partition: $\bigcup F_i = \mathcal{F}$.*

FORMAL PROBLEM STATEMENT 1. *Given a set of n data tuples in D over a set of m features in $\mathcal{F}$, find a partitioning $\mathcal{P} : \{F_1, F_2, \ldots, F_k\}$ of $\mathcal{F}$ and a set of associated code tables $\mathcal{CT} : \{CT_{F_1}, CT_{F_2}, \ldots, CT_{F_k}\}$, such that the total compression cost in bits given below is minimized.*

$$L(\mathcal{P}, \mathcal{CT}, D) = L(\mathcal{P}) + \sum_{F \in \mathcal{P}} L(\pi_F(D) | CT_F) + \sum_{F \in \mathcal{P}} L(CT_F) \ ,$$

in which $\pi_{F_i}(D)$ is the projection of D on feature subspace F_i.

Note that the number of features m and the number of tuples n are fixed over all models we consider for a D, and hence are a constant additive that we can safely ignore.

3.1.1 Partition encoding

The first term of $L(\mathcal{P}, \mathcal{CT}, D)$ denotes the length of encoding the partitioning, which consists of two parts; encoding (a) the number of partitions and (b) the features per partition.

(a) Encoding the number of partitions: First, we need to encode k, the number of partitions and code tables. For this, we use the MDL-optimal encoding of an integer [12]. The cost for encoding an integer value k is $L^0(k) = \log^\star(k) + \log(c)$, with $c = \sum 2^{-\log(n)} \approx 2.865064$,, and $\log^\star(k) = \log(k) + \log\log(k) + \cdots$ sums over all positive terms. Note that as $\log(c)$ is constant for all models, we ignore it.

(b) Encoding the features per partition: Then, for each feature we have to describe to which partition it belongs. This we do using $m \log(k)$ bits.

In summary, $L(\mathcal{P}) = \log^\star(k) + m \log(k)$.

3.1.2 Data encoding

The second term of $L(\mathcal{P}, \mathcal{CT}, D)$ denotes the cost of encoding the data with the given set of code tables. To do so, each tuple is partitioned according to $\mathcal{P}$, and encoded using the optimal codes in the corresponding code tables following the procedure in §2.3.1.

3.1.3 Model encoding

The last term of $L(\mathcal{P}, \mathcal{CT}, D)$ denotes the model cost, that is the total length of encoding the code tables. Each code table is encoded following the procedure described in §2.3.2.

We note that the number of feature groups k is not a parameter of our method but rather is determined by MDL. In particular, MDL ensures that we will not have two separate code tables for a pair of highly correlated feature groups as it would yield lower data cost to encode them together. On the other hand, combining feature groups may yield larger code tables, that is higher model cost, which may not compensate for the savings from the data cost. In other words, we group features for which the total encoding cost $L(\mathcal{P}, \mathcal{CT}, D)$ is reduced. Basically, we employ MDL to both guide us in finding which features to group, as well as in deciding how many groups we should have.

Also note that compression is a type of statistical method, however, instead of having to choose a prior distribution appropriate for the data (normal, chi-square, etc.) we use a rich model class (code tables) to induce the distribution from the data.

3.2 Mining Set of Code Tables: Algorithm

Having defined the cost function as $L(\mathcal{P}, \mathcal{CT}, D)$, we need an algorithm to search for the best set of code tables $\mathcal{CT}$ for the optimal vertical partitioning $\mathcal{P}$ of the data such that the total encoded size $L(\mathcal{P}, \mathcal{CT}, D)$ is minimized.

The search space for finding the best code table for a given set of features, yet alone for finding the optimal partitioning of features, however, is quite large. Finding the optimal code table for a set of $|F_i|$ features involves finding all the possible patterns with different value combinations up to length $|F_i|$ and choosing a subset of those patterns that would yield the minimum total cost on the database induced on F_i. Furthermore, the number of possible partitioning of a set of m features is the well-known Bell number B_m.

While the search space is prohibitively large, it neither has a structure nor exhibits monotonicity properties which could help us in pruning. As a result, we resort to heuristics. Our approach builds the set of code tables in a greedy bottom-up, iterative fashion. We give the pseudo-code as Algorithm 1, and explain it in more detail in the following.

COMPREX starts with a partitioning $\mathcal{P}$ in which each feature belongs to its own group (1), and separate, elementary code tables CT_i for each feature f_i associated with the feature sets (2).

DEFINITION 2. *An elementary code table CT encodes a database D induced on a single feature $f \in \mathcal{F}$. The patterns $p \in CT$ consist of all length-1 unique items $v_1, \ldots, v_{arity(f)}$ in $dom(f)$. Finally, $usage(p \in CT) = freq(f = v)$.*

Typically, some features of the data will be more strongly correlated than others, e.g., the age of a car and its fuel efficiency, or the weather temperature and flu outbreaks. In such cases, it will be worthwhile to represent features for which the correlation is 'high enough' together within one CT, as we can then exploit correlation to save bits.

More formally, we know from Information Theory that given two (sets of) random variables (in our case feature groups) F_i and F_j, the average number of bits we can save when compressing F_i and F_j together instead of separately, is known as their Information Gain. That is,

$$IG(F_i, F_j) = H(F_i) + H(F_j) - H(F_i, F_j) \geq 0,$$

In fact, the IG of two (sets of) variables is always non-negative (zero when the variables are independent from each other), which implies that the data cost would be the smallest if all the features were represented by a single CT. On the other hand, our objective function also includes the compression cost of the CTs.

Clearly, having one large CT over many (possibly uncorrelated) features will typically require more bits in model cost than it saves in data cost. Therefore, we can use IG to point out good merge candidates, subsequently employing MDL to decide if the total cost is reduced, and hence, whether to approve the merge or not.

The first step then is to compute the IG matrix for all pairs of the current feature sets, which is a non-negative and symmetric matrix (3). Let $|F_i|$ denote the cardinality, i.e. the number of features in the feature set F_i. We sort the pairs of feature sets in decreasing order of IG-per-feature, i.e. normalized by their total cardinality, and start *outer* iterations to go over these pairs as the candidate CTs to be merged, say CT_i and CT_j (5). The starting cost $cost_{init}$ is set to the total cost with the initial set of CTs (6). The construction of the new $CT_{i|j}$ then works as follows: we put all the existing patterns $p_{i,1}, \ldots, p_{i,n_i}$ and $p_{j,1}, \ldots, p_{j,n_j}$ from both CTs into the new CT (7). Following the convention, they are sorted first by length (from longer to shorter) and second by usage (from higher to lower) (8). Candidate partitioning $\hat{\mathcal{P}}$ is built by dropping feature sets F_i and F_j from $\mathcal{P}$ and including the concatenated set $F_{i|j}$ (9). Similarly, we build a temporary code table set $\hat{\mathcal{CT}}$ by dropping the candidate tables CT_i and CT_j and adding the new $\hat{CT}_{i|j}$ (10).

Next, we find all the unique rows of the database induced on $F_{i|j}$ (11). These patterns of length ($|F_i|$+$|F_j|$) are sorted in decreasing order of their occurrence in the database and constitute the candidates to be inserted into the new CT. Let $p_{i|j,1}, \ldots, p_{i|j,n_{i|j}}$ denote these patterns of the combined feature set $F_{i|j}$ in their sorted order of frequency. In our *inner* iterations (12), we insert these one-by-one (13), update (i.e. decrease) the usages of the existing overlapping patterns (14), remove those patterns whose usage drops to zero (15), recompute the code word lengths with updated usages (16) and compute the total cost after each insertion. If total cost is reduced, we store the candidate partitioning $\hat{\mathcal{P}}$ and associated set of code tables $\hat{\mathcal{CT}}$ (18), otherwise we continue insertions (from 12) with the next candidate patterns for possible future cost reduction.

In the outer iterations, if total cost is reduced the IG between the new feature set $F_{i|j}$ and the rest are computed (22). Otherwise the merge is rejected and the candidates $\hat{\mathcal{P}}$ and $\hat{\mathcal{CT}}$ are discarded (24). Next the algorithm continues to search for future merges, and the search terminates when there are no more pairs of feature sets that can be merged for reduced cost. The resulting set of feature sets and their corresponding set of code tables constitute our solution.

Note that the data from which we induce code tables may include outliers. As by MDL we only allow patterns in our code tables that help compression, we are not prone to include spurious patterns. See [22] for a more complete discussion.

Algorithm 1 COMPREX

Input: Database D with n tuples and m (categorical) features
Output: A heuristic solution to Problem Statement 1: a feature partitioning $\mathcal{P} : \{F_1, \ldots, F_k\}$, associated set of code tables $\mathcal{CT} : \{CT_1, \ldots, CT_k\}$, and total encoded size $L(\mathcal{P}, \mathcal{CT}, D)$

1: $\mathcal{P} \leftarrow \{F_1, \ldots, F_m\}$, $F_i = \{f_i\}, 1 \leq i \leq m$
2: $\mathcal{CT} \leftarrow \{CT_1, \ldots, CT_m\}$, where each CT_i is elementary
3: Compute IG between $\forall (F_i, F_j) \in \mathcal{P}, i > j$
4: **repeat**
5: **for** each $(F_i, F_j) \in \mathcal{P}$ in decreasing normalized IG **do**
6: Compute $cost_{init} \leftarrow L(\mathcal{P}, \mathcal{CT}, D)$
7: Put patterns $p \in CT_i$ and $p \in CT_j$ into a new $\hat{CT}_{i|j}$
8: Sort $p \in \hat{CT}_{i|j}$, (1) by length and (2) by usage
9: $\hat{\mathcal{P}} \leftarrow \mathcal{P} \backslash (F_i \cup F_j) \cup F_{i|j}$
10: $\hat{\mathcal{CT}} \leftarrow \mathcal{CT} \backslash (CT_i \cup CT_j) \cup \hat{CT}_{i|j}$
11: Find unique rows (candidate patterns) $p_{i|j}$ in $D_{F_{i|j}}$
12: **for** each unique row $p_{i|j,x}$ in decreasing frequency **do**
13: Insert $p_{i|j,x}$ to new $\hat{CT}_{i|j}$
14: Decrease usages of overlapping patterns $p \in \hat{CT}_{i|j}$ and $p \in cover(p_{i|j,x})$ by $freq(p_{i|j,x})$
15: Remove patterns $p \in \hat{CT}_{i|j}$ with $usage(p)$=0
16: Recompute $L(code(p \in \hat{CT}_{i|j}))$ with new $usage$s
17: **if** $L(\hat{\mathcal{P}}, \hat{\mathcal{CT}}, D) < L(\mathcal{P}, \mathcal{CT}, D)$ **then**
18: $\mathcal{P} \leftarrow \hat{\mathcal{P}}$ and $\mathcal{CT} \leftarrow \hat{\mathcal{CT}}$
19: **end if**
20: **end for**
21: **if** $L(\mathcal{P}, \mathcal{CT}, D) < cost_{init}$ **then**
22: Compute IG between $F_{i|j}$ and $\forall F_x \in \mathcal{P}, F_x \neq F_{i|j}$
23: **else**
24: Discard $\hat{\mathcal{P}}$ and $\hat{\mathcal{CT}}$
25: **end if**
26: **end for**
27: **until** convergence, i.e. no more merges

3.3 Computational Speedup and Complexity

In Algorithm 1, computationally most demanding steps are (1) finding all the unique rows in the database under a particular feature subspace when two feature sets are to be merged (11) and (2) after each insertion of a new pattern to the code table, finding the existing overlapping patterns the usages of which to be decreased (14).

With a naive implementation, step (1) is performed on the fly scanning the entire database once and possibly using many linear scans and comparisons over the unique rows found so far in the process. Furthermore, step (2) requires a linear scan over the current patterns in the new code table $\hat{CT}_{i|j}$ for each new insertion. The total computational complexity of these linear searches depends on the database, however, with the outer and inner iteration levels (Lines 5 and 12, respectively), those may become computationally infeasible for very large databases.

We improve with a simple design choice. Instead of an integer vector of $usage$ per pattern in CT, we have a sparse matrix C for patterns versus data points, the binary entries c_{ji} indicating whether data tuple i contains pattern j in its $cover$. Note that the row sum of the C matrix gives the $usage$s of the patterns. In such a setting, step (1) (mining for candidate patterns) works as follows: Let F_i and F_j denote the feature sets to be merged. Let C_i denote the $n_i \times n$ patterns versus data tuples matrix for code table CT_i and similarly C_j denote the $n_j \times n$ matrix for CT_j, in which n_i and n_j respectively denote the number of patterns each table has. We obtain the usages for the new candidate patterns (merged unique

rows) under the merged feature subspace $F_{i|j}$ by multiplying C_i and C_j^T into a $n_i \times n_j$ matrix U, which takes $\mathcal{O}(n_i n n_j)$.

Next, we sort the merged patterns in decreasing order by their usage $U_{x,y}$ and insert them to $\hat{CT}_{i|j}$ one-by-one. Note that we exploit the existing patterns in the code tables to be merged, rather than finding all the unique rows of the database. This way, we quickly identify good frequent candidates and consider only $n_i n_j$ of them. Since $n_i, n_j \ll n$, we practically reduce the number of inner iterations (12) to a constant.

From here, step (2) (insertions) works as follows: Let $U_{x,y}$ denote the highest usage associated with the merged pattern $p_i(x)|p_j(y)$. The insertion of $p_i(x)|p_j(y)$ into the code table simply means the addition of a new row to the $C_{i|j}$ matrix ($C_{i|j}$ is obtained by concatenating the rows of C_i and C_j and reordering its rows (1) by length and (2) by usage of the patterns it initially contains). The new row is then the dot product (i.e. logical AND) of row x in C_i and the row y in C_j (data tuples which contain *both* $p_i(x)$ and $p_j(y)$ in their cover). Moreover, we decrease the usages of the merged patterns $p_i(x)$ and $p_j(y)$ by subtracting the new row from both of their corresponding rows. All these updates are $\mathcal{O}(n)$.

All in all, the inner loop (starting in 12) goes over $n_i n_j$(≈constant) number of candidate patterns and each insertion takes $\mathcal{O}(n)$. Therefore, the inner loop takes $\mathcal{O}(n)$.

Next, we consider the outer loop (starting in 5), which tries to merge pairs of code tables. In the worst case we get $\mathcal{O}(m^2)$ trials when all merges are discarded. As a result the worst case complexity of COMPREX becomes $\mathcal{O}(m^2 n)$. In practice, however, many features are correlated and we obtain a merge at almost every step, yielding about linear complexity in both data size and dimension.

3.4 COMPREX at Work: Anomaly Detection

Compression based techniques are naturally suited for anomaly and rare instance detection. Next we describe how we exploit our dictionary based compression framework for this task.

In a given code table, the patterns with short code words, that is those that have high usage, represent the patterns in the data that can effectively compress the majority of the data points. In other words, they capture the trends summarizing the *norm* in the data. On the other hand, the patterns with longer code words are rarely used and thus encode the sparse regions in the data. Consequently, the data tuples in a database can be scored by their encoding cost for anomalousness.

More formally, given a set of code tables $CT_1, \ldots, CT_k$ found by COMPREX, each data tuple $t \in D$ can be encoded by one or more code words from each $CT_i, i = \{1, \ldots, k\}$. The corresponding patterns constitute the *cover* of t as discussed in §2.3.1. The encoding cost of t is then considered as its *anomalousness* score; the higher the compression cost, the more likely it is "to arouse suspicion that it was generated by a different mechanism" [13].

$$\begin{aligned} \textit{score(t)} &= L(t|CT) = \sum_{F \in \mathcal{P}} L(\pi_F(t)|CT_F) \\ &= \sum_{F \in \mathcal{P}} \sum_{p \in cover(\pi_F(t))} L(code(p)|CT_F) \end{aligned}$$

Having computed the compression costs, one can sort them and report the top k data points with the highest scores as possible anomalies. An alternative way [22] is to determine a decision threshold θ and flag those points with a compression cost greater than θ as anomalies. One can use the Cantelli's Inequality [11] which provides a well-founded way to determine a good value for the threshold θ for a given confidence level, that is, an upper bound for the false positive rate. For generality, in our experiments we show the accuracy at all decision thresholds.

4. EMPIRICAL STUDY

In our study, we explored the general applicability of COMPREX by considering rich data. We experimented with many (publicly available) datasets[1] from diverse domains, as shown in Tables 2 and 3. Other datasets we used include graph and satellite image datasets, which we will discuss in §4.2.3 and §4.2.4, respectively. The source code of COMPREX is available for research purposes[2].

We evaluated our method with respect to four criteria: (1) compression cost in bits, (2) running time, (3) detection accuracy, and (4) scalability. We also compared our results with two state of the art methods, KRIMP [25] and LOF [4] (note that KRIMP was shown [22] to outperform single-class classification methods, like NNDD and SVDD, for the anomaly detection task; therefore for reasons of space we only compare to the winner). KRIMP is also a compression based anomaly detection method, but uses a single code table to encode a dataset. It performs the costly frequent itemset mining as a pre-processing step to generate candidate patterns for the code table. LOF is a density-based outlier detection method that computes the local-outlier-factor of data points with respect to their nearest neighbors. Neither LOF nor KRIMP are parameter-free, they respectively require the minimum support threshold and the number of neighbors to be considered. COMPREX, in contrast, automatically determines the number of necessary code tables, as well as which patterns to include, and as such has no parameters.

4.1 Compression cost and Running time

One goal of our study is to develop a fast method that would model the *norm* of the data and hence give low compression cost in bits. In this section, we use both COMPREX and KRIMP for compressing our datasets and show the total cost in bits and the corresponding running times in Figure 1 (a) and (b), respectively. Note that the results for our largest datasets Enron, Connect-4, and Covertype are given with broken y-axes with values in millions for cost and thousands for time.

We note that COMPREX achieves very nice compression rates, outperforming KRIMP for all of the datasets, and providing up to 96% savings in bits (47% on average).

With respect to running time, we notice that for most of the (smaller) datasets, our running time is only slightly higher than that of KRIMP but still remains under 16 seconds. Even though for small datasets the run time of KRIMP is negligible, for datasets especially with large number of features, its running time increases significantly. The computationally most demanding part of KRIMP is the frequent itemset mining, making it less feasible for large and high dimensional categorical datasets. For example, for the Connect-4 dataset with 42 features and the Chess (king,rook vs. king,pawn) datasets with 36 features, KRIMP cannot finish within reasonable time due to very large candidate sets.

To alleviate this problem, KRIMP accepts a *minsup* parameter, which is the minimum number of occurrences of an itemset in the database to be considered as frequent. The higher the *minsup* is set, the fewer the extracted itemsets are. However, high *minsup* comes with a trade-off; the higher the *minsup*, the smaller the number of candidates, the smaller the search space and the worse the final code table approximates the optimal code table. In contrast,

[1] http://archive.ics.uci.edu/ml/datasets.html
[2] http://dl.dropbox.com/u/17337370/CompreX_12_tbox.tar.gz

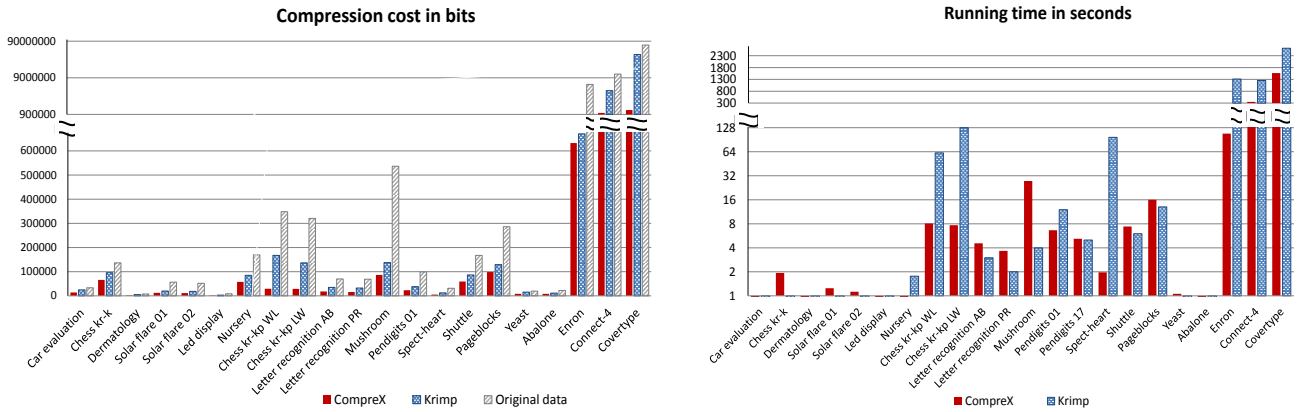


Figure 1: (left) Compression cost (in bits) when encoded by COMPREX vs. KRIMP. (right) Run time (in seconds) of COMPREX vs. KRIMP. For large datasets, extremely many frequent itemsets negatively affect the runtime for KRIMP.

our method does not require any sorts of parameters and frequent itemset mining.

In our experiments, we find *closed* frequent itemsets with *minsup* set to 5000 and 500 for the Connect-4 and Chess (kr-kp) datasets, respectively. Even then, the running time of our method remains lower than that of KRIMP (see Fig. 1). On the other hand, the time required for frequent itemset mining also depends on the dataset characteristics. For example, we observe in Figure 1 that the running time of KRIMP on the (larger) Mushroom dataset is much smaller than that on the (smaller) Spect-heart dataset, with both having the same number of (22) features.

For our largest datasets (in terms of size and number of features) in Figure 1, notice that the running time of KRIMP is quite large (about 20 mins) for Enron and Connect-4, and (45 mins) for Covertype. Moreover, its compression cost is still higher than that our method provides. Therefore, we conclude that our method becomes more advantageous especially for large datasets.

4.2 Detection accuracy

Besides achieving high compression rate, we would also (if not more) want our method to be effective in spotting anomalies. In this section, we experiment with various types of data including relational, graph and image databases.

For measuring detection performance, we use two-class datasets. The number of data points from one class is significantly smaller than that from the other class. We call these classes as minority and the majority classes, respectively. The data points from the minority class are considered to be the "anomalies". The underlying assumption is that different classes have different distributions –some classes may be more similar than others, just like with real anomaly(-classes). Examples to the classes include poisonous vs. edible in Mushroom data, unaccountable vs. very good in Car data, and win vs. loss in Connect-4 data.

As a measure of accuracy, we use *average precision*; the average of the precision values obtained across recall levels. We plot the detection precision, that is the ratio of the number of true positives to the total number of predicted positives, against the recall (=detection rate), that is the fraction of total true anomalies that are detected. A point on the plot is obtained by setting a threshold compression cost—any record with a cost larger than that threshold is flagged as an anomaly. The corresponding precision and recall are then calculated. By varying the threshold, we obtain the curve for the entire range of recalls. The *average precision* then approximates the area under the precision-recall curve.

4.2.1 COMPREX *on categorical data*

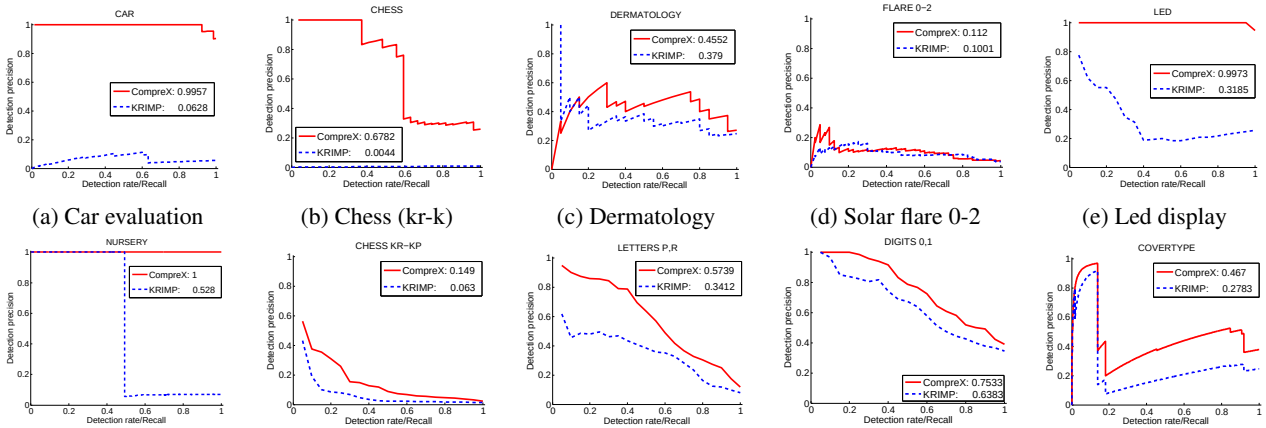
Figure 2 shows the precision-recall plots of COMPREX and KRIMP on several of our categorical datasets. Here, a higher curve denotes better performance, since it corresponds to a higher precision for a given recall. The average precision values are also given in Table 2, for all the categorical datasets. Notice that for most datasets COMPREX achieves higher accuracy than KRIMP. This is obvious especially for the Car, Chess, Led and Nursery datasets. We notice that the performance of the methods also depends on the detection task. For example, both methods perform well on the Mushroom dataset, for which the poisonous ones exhibit quite different features than the edible ones. However, the accuracies of both methods drop for the Connect-4 dataset for which the detection task, i.e. which player is going to win the game given the 8-ply positions, is much harder.

Table 2: Average precision (normalized area under precision-recall curve) for categoric datasets, comparing COMPREX and KRIMP. Further, we give dataset size, number of features and partitions, $minsup$ for KRIMP (star denotes *closed* itemsets).

Dataset	$\|D\|$	$\|\mathcal{F}\|$	$\|\mathcal{P}\|$	$\frac{min}{sup}$	Cp'X	Kr'p
Car evaluation	1275	6	6	1	**0.99**	0.06
Chess (k)	4580	6	4	1	**0.67**	0.01
Dermatology	132	12	12	1	**0.45**	0.37
Solar flare 0–1	1312	10	6	1	**0.19**	0.15
Solar flare 0–2	1211	10	6	1	**0.11**	0.10
Led display	370	7	7	1	**0.99**	0.31
Nursery	4648	8	6	1	**1.00**	0.52
Chess (kp) W-L	1689	36	18	500*	**0.14**	0.06
Chess (kp) L-W	1547	36	18	500*	**0.03**	**0.03**
Letter rec. A–B	809	16	11	1*	0.17	**0.18**
Letter rec. P–R	823	16	11	1*	**0.57**	0.34
Mushroom	4258	22	5	1*	**1.00**	0.93
Digit rec. 0–1	1163	16	12	10	**0.75**	0.63
Digit rec. 1–7	1163	16	9	20	**0.51**	0.34
Spect–heart	267	22	16	1*	**0.12**	**0.12**
Connect–4	44k	42	11	5k*	**0.02**	0.01
Covertype	286k	44	6	280k*	**0.46**	0.27
				average	**0.48**	0.26

4.2.2 COMPREX *on numerical data*

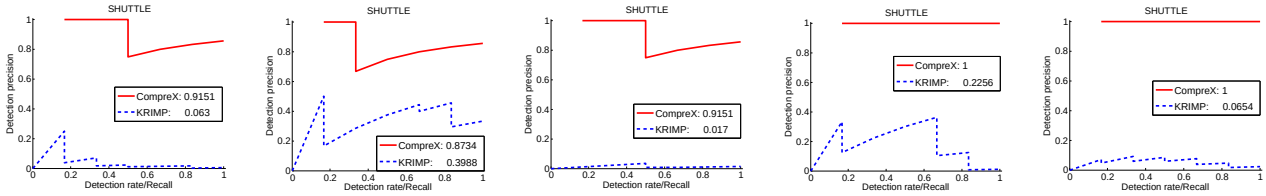
While COMPREX is designed to work with categorical datasets, it can also be used to detect anomalies in datasets with numerical features. For that, we first convert the continuous numerical



(a) Car evaluation (b) Chess (kr-k) (c) Dermatology (d) Solar flare 0-2 (e) Led display

(f) Nursery (g) Chess (kr-kp) (h) Letters P-R (i) Digits 0-1 (j) Covertype

Figure 2: Performance of COMPREX vs KRIMP on two-class categorical datasets. Given are precision vs. recall for various thresholds to flag tuples as anomalous. Note that COMPREX outperforms KRIMP for most detection tasks.



(a) LOG-bins (b) 5-LINbins (c) 10-LINbins (d) 3-SAXbins (e) p0.01-MDLbins

Figure 3: Performance of COMPREX vs KRIMP on the two-class numerical transaction dataset Shuttle. Notice that COMPREX outperforms KRIMP consistently for various discretization methods used.

features to discretized nominal features. There exist various techniques to this end. In our study, we consider several: linear, logarithmic, SAX [17], and MDL-based [14] binning. Linear binning involves dividing the value range of each feature into equal sized intervals. Logarithmic binning first sorts the feature values and assigns the lower b-fraction to the first bin, the next b-fraction of the rest to the second bin, and so on, until all the values are assigned to a bin. SAX has proved to be an effective symbolic representation, especially for time series data. MDL-based binning estimates variable-width histograms with optimal bin count automatically, for a given data precision.

We experiment with these various discretization methods under various parameter settings. In Figure 3, we show the accuracy of COMPREX versus KRIMP on the Shuttle dataset, using logarithmic binning with b=0.5, linear binning with 5 and 10 bins, SAX with 3 bins, and MDL-based binning with precision 0.01. Results are similar for many other settings and for the rest of the numerical datasets, which we omit for brevity. Notice that regardless of the discretization used, COMPREX performs consistently better than its competitor KRIMP.

To this end, we also compare our method with LOF on several numerical datasets. LOF is a widely used outlier detection method based on local density estimation. While it is quite powerful when applied to numeric data, it cannot be directly applied to categorical datasets. In this comparison, both methods require the careful choice of a parameter; number of nearest neighbors k for LOF, and a binning method and its corresponding parameter for COMPREX.

In Figure 4, we show the accuracy of LOF versus COMPREX with their best parameter choices on our numerical two-class datasets.

Table 3: Average precision for the numerical datasets, comparing COMPREX, KRIMP and LOF. Further, dataset size, number of features and partitions, $minsup$ used for KRIMP.

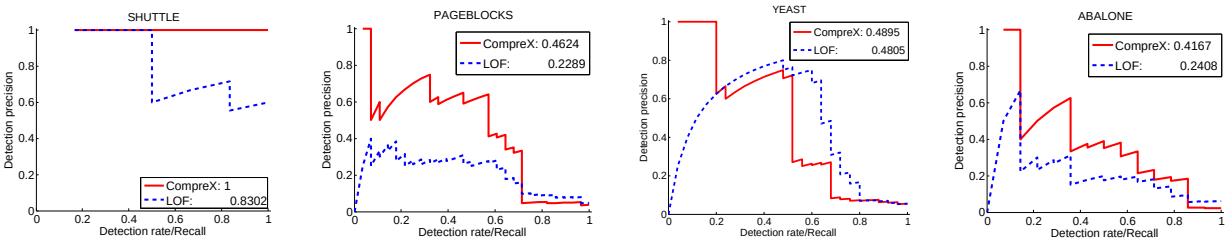
Dataset	$\|D\|$	$\|\mathcal{F}\|$	$\|\mathcal{P}\|$	$\frac{min}{sup}$	Cp'X	Kr'p	LOF
Shuttle	3416	9	5	1	**1.00**	0.22	0.83
Pageblocks	4941	10	6	1	**0.46**	0.37	0.23
Yeast	468	8	8	1	**0.49**	0.17	0.48
Abalone	703	7	3	1	**0.42**	0.15	0.24
				average	**0.59**	0.23	0.45

Table 3 gives the corresponding average precision scores for all three methods. Notice that COMPREX achieves comparable or better performance than LOF, even though it operates on discretized data which loses some information due to this process, and thus is not as optimized as LOF for numeric data. This shows that COMPREX can also be applied to datasets with numerical or with a hybrid of both categorical and numerical features.

4.2.3 COMPREX *on graph data*

Given data points and their features in numerical or categorical space, our method can also be applied to other complex data, including graphs. To this end, we study the Enron graph[3], in which nodes represent individuals and the edges represent email interactions. In our setting the nodes correspond to the data points, and the features to the ego-net features we extract from each node. The

[3] http://www.cs.cmu.edu/~enron/



(a) Shuttle (b) Page blocks (c) Yeast (d) Abalone

Figure 4: Performance of LOF vs COMPREX with the best choice of parameters on the numerical transaction datasets. Notice that COMPREX achieves comparable or better performance than LOF even after discretization.

ego-net of a node (ego) is defined as the subgraph of the node, its neighbors, and all the links between them. We extract 14 numerical ego-net features, such as the number of edges, total weight, ego-net degree (number of edges connecting the ego-net to the rest of the graph), in- and out-degree, etc. We refer the reader to [2] for more on ego-net features. Features are discretized into 10 linear bins.

In Table 4, we show the top 5 email addresses with the highest compression cost found by COMPREX. The dataset does not contain any ground truth anomalies, therefore we provide anecdotal evidence for the discovered "anomalies". Our first observation is the significantly high compression cost of the listed points—103 to 107 bits given a global average of 6.72 bits (median is 4.27). This is due to the rare and high number of patterns used in their cover. Notice that each of them are covered with 7 patterns compared to a global average of 2.05 (median is 2). Moreover, the usages of the cover patterns is quite small—thus longer code words and high total compress-cost. Further inspection justifies our results: for example, 'sally.beck' (employee chief operating officer) contains the highest number of (31k) edges in its ego-net and the highest ego-net degree (of 85k), implying that she is highly connected to the rest of the graph as opposed to many other nodes in the graph.

Table 4: Top-5 anomalies for Enron, with one regular-joe example. Given are, email address, compression cost, size of cover, and average usages of the cover patterns; high usages correspond to short codes. Average cost is 6.72 bits, average number of covering patterns is 2.05.

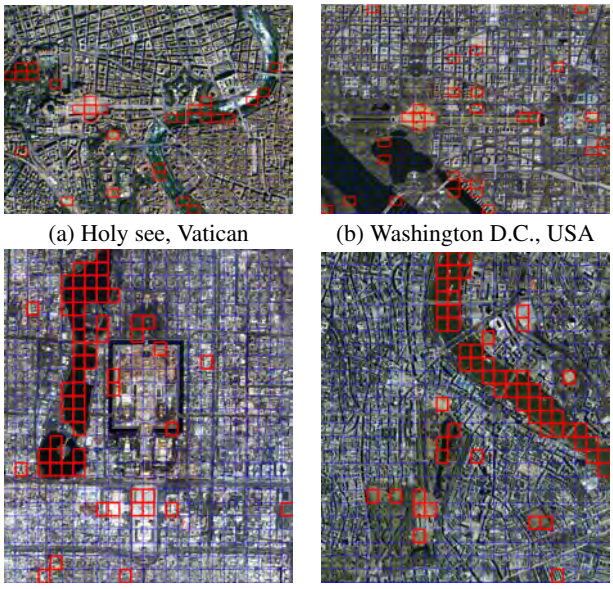
name@enron.com	*cost (bits)*	$\|cover\|$	*avg usage* $\pm$ *std of cover patterns*
sally.beck	107.28	7	3.7 ± 5.4
jeff.dasovich	107.11	7	3.4 ± 3.9
outlook.team	106.70	7	4.8 ± 6.3
david.forster	105.11	7	4.1 ± 5.2
kenneth.lay	103.24	7	5.8 ± 7.5
⋮	⋮	⋮	⋮
robert.badeer	1.53	2	52k $\pm$ 4.3k

4.2.4 COMPREX *on image data*

Next, for our image datasets[4] for which class labels also do not exist, we provide an anecdotal and visual study. The image datasets are the satellite images of four major cities from around the world as shown in Figure 5. Each image is split into 25x25 rectangle tiles, for which we extracted 15 numerical features, and subsequently discretized into 10 linear bins. The first three features denote the mean RGB values for each tile and the rest denote Gabor features.

[4] http://geoeye.com/CorpSite/gallery/

In Figure 5, top tiles with high compression cost are highlighted in red. We observe that COMPREX effectively spots interesting and rare regions. For example, the districts of Roman Catholic Church in Vatican and the Washington Memorial in Washington D.C. that distinctively stand out in the images are captured in top anomalies. In Forbidden city, COMPREX spots the three lakes (Beihai, Zhonghai, Nanhai), the Jingshan Park on its right, as well as the Tiananmen Square on the south. Finally, in London COMPREX marks the part of Thames river, the Buckingham Palace as well as several rare plain fields in the city.



(a) Holy see, Vatican (b) Washington D.C., USA

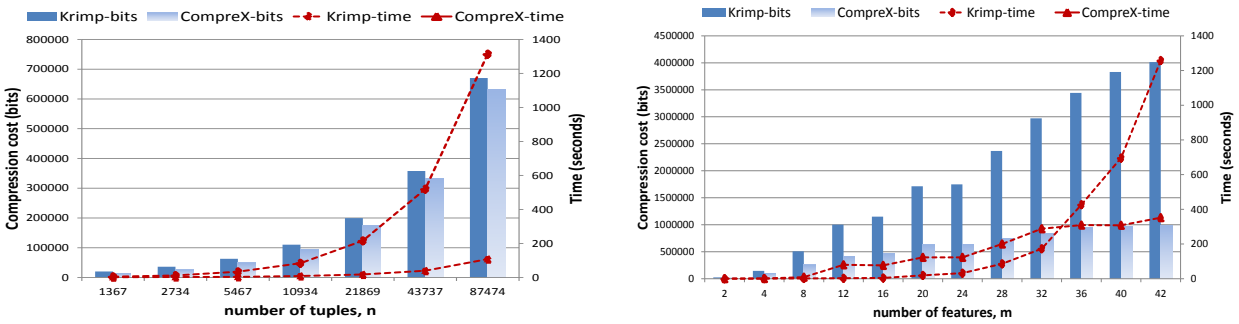
(c) Forbidden city, Beijing (d) London, UK

Figure 5: "Anomalous" tiles—with high compression cost—on the image datasets are highlighted with red borders (figures best viewed in color). Notice that COMPREX successfully spots qualitatively distinct regions that stand out in the images.

4.3 Scalability

KRIMP falling short on large datasets arises the question of scalability, which is maybe even more important than the issue of speed. Therefore, in Figure 6, we also show the running time of both methods for growing dataset and feature sizes on Enron and Connect-4.

We observe that the running time of KRIMP grows significantly with the increasing size in both cases. The difference is evident especially for growing feature size. This is due to frequent itemset mining scaling exponentially with respect to number of features.



(a) time vs. data size

(b) time vs. feature size

Figure 6: Scalability of COMPREX and KRIMP with increasing dataset and feature size. (a) Enron with large n, (b) Connect-4 with large m. Notice that COMPREX achieves better scalability for large databases, following a linear trend in growth.

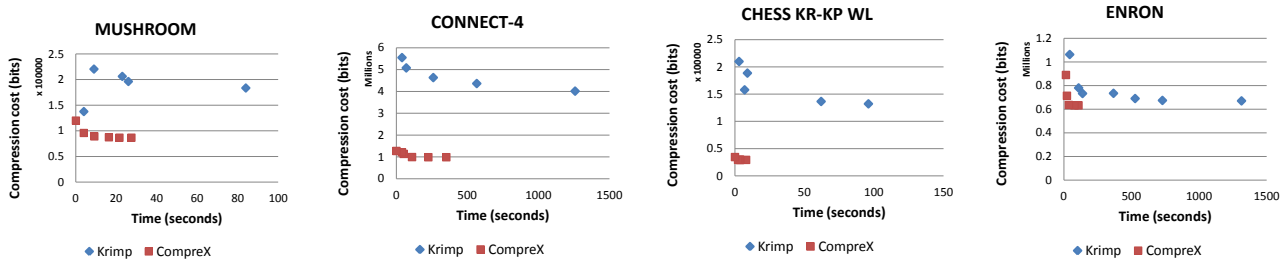


Figure 7: Scatter plots of compression cost versus running time of COMPREX and KRIMP. Notice that COMPREX achieves lower cost at any given time, which KRIMP cannot catch up with even when let to run for longer.

On the other hand, the increase in running time for our method follows a linear trend indicating that COMPREX scales better with increasing database size and dimension.

Another advantage of our method is that it can work as an *anytime* algorithm. Since it is a bottom-up approach which seeks for lower total cost over iterations via more merges, the compression procedure can be stopped at any point given the availability of time. The same goal can be achieved for KRIMP by tuning the *minsup* parameter; the higher the minsup, the lower the running time.

To compare the methods, we show the compression costs achieved at various durations in Figure 7. Notice that for any given run-time duration, COMPREX achieves lower compression cost than its competitor. Moreover, even KRIMP is allowed to run for longer, it still cannot reach as low of a cost as COMPREX can in much less time.

5. RELATED WORK

Our contributions are two-fold: finding descriptive pattern sets, and anomaly detection in categoric data.

5.1 Data Description

COMPREX builds upon the KRIMP algorithm by Siebes et al. [21, 25], and likewise employs the MDL principle [12] to define the best set of patterns as those that compress the data best. KRIMP heuristically approximates the optimum by considering a collection of itemsets in a fixed order, greedily selecting itemsets that contribute to better compression. The code tables KRIMP discovers have been shown to characterize data distributions very well, providing competitive performance on a variety of tasks [22, 25].

Unlike KRIMP, COMPREX can detect and exploit independent groups of attributes by using *sets of* code tables. Whereas KRIMP requires the user to provide a set of candidate itemsets (or a minimal support threshold to mine these), COMPREX is parameter-free in both theory and practice. Furthermore, as the expensive mining and sorting steps are avoided, COMPREX can be used as an anytime algorithm. However, while KRIMP is defined for transaction data in general, we restrict ourselves to categorical data.

The PACK algorithm [24] also follows a bottom-up MDL approach, using a decision tree per attribute to encode binary data 0/1 symmetrically. By translating these trees into downward-closed families of itemsets, patterns can be extracted. Although good compression results are obtained, by the downward-closed requirement typically large groups of itemsets are returned, hindering usability.

Further examples of pattern set mining methods that describe data include Tiling [9], Noisy Tiling [15], and Boolean matrix factorization [19]. As these do not define a score for individual rows, it is not trivial to apply them for anomaly detection.

Attribute clustering [18] is related in that it can be regarded as a crude form of COMPREX. With the goal of only providing a high-level summarization of the data, instead of a detailed characterization, the authors employ MDL to find the optimal grouping of binary attributes. These groups are described using crude code tables, consisting of all attribute-value combinations in the data; unlike in our setting where code tables can consist of itemsets of different lengths, providing better characterization of the local structure. Unlike COMPREX, it can only describe shown attribute-value combinations, and not trivially generalize, nor can it meaningfully capture structure local within the regarded attribute group.

Recently, [23] proposed the SLIM algorithm for mining a *single* code table directly from binary data. The code tables are shown to perform on par with KRIMP for classification and anomaly detection. Here, we consider multiple code tables, obtaining much better anomaly detection results than KRIMP. Moreover, by focusing on categorical data, COMPREX can search more efficiently, returning its code tables in only a fraction of the times reported in [23].

5.2 Anomaly Detection

Identifying outliers in multi-dimensional real-valued data has been studied extensively. Examples of proposals to this end include [1, 3, 4, 7, 10]. These typically exploit the (continuous) ordered domains of the attributes to define meaningful distance functions between tuples, which cannot be straightforwardly applied on nominal (categorical) data. Furthermore, most of these methods require several parameters to be specified by the user. For example, [4] and [10] take the number of nearest neighbors k to be compared to as input. They also require a distance metric for finding the k-nns of the data points, which often suffers from the curse of dimensionality in high dimensions. Lastly, these methods do not build a model, and thus cannot provide anomaly *characterization*. Model-based approaches like COMPREX, on the other hand, capture the *norm* of the data and can pindown the deviations from it, providing better interpretability for the claimed anomalies.

While most work on outlier detection has focused on numerical datasets, there also exist some work on anomaly detection for discrete data. [6] surveys methods for finding the anomalous sequences in a given set of sequences. The main disadvantage of the methods therein (kernel-,window-based, etc.) is that their performance is highly reliant on the choice of their parameters (similarity measure, window size, etc.). Proposals for anomaly detection in categorical data include [5, 8, 26] and recently [22]. [5] learns a structure and the parameters of a Bayesian network, and uses the log-likelihood values as the anomalousness score of each record. [8, 26] address the problem of finding anomaly *pattern*s. They build a Bayes net that represents the baseline distribution, and then score the rules with unusual proportions compared to the baseline. They restrict their method to work with only one and two component rules due to high computation required. COMPREX is most related to [22], which employs KRIMP [21] as its compressor for anomaly detection. The key differences are that KRIMP finds a *single* code table, performs costly frequent itemset mining for candidate generation, and requires a minimum support parameter.

6. CONCLUSIONS

The contributions of this work are two-fold: (1) we achieve fast characterization of data by mining subspace code-tables, i.e. patterns sets, and (2) we apply our descriptive patterns to reliable anomaly detection in categorical data.

We introduce a novel, *parameter-free* method, COMPREX, that builds a data compression model using multiple dictionaries for encoding, and reports the data points with high encoding cost as anomalous. Our method proves *effective* for both tasks: It provides higher compression rates at lower run times especially for large datasets, and it is capable of spotting rare instances effectively, with detection accuracy often higher than its state-of-the-art competitors. Experiments on diverse datasets show that COMPREX successfully *generali*zes to a broad range of datasets including image, graph, and relational databases with both categorical and numerical features. Finally our method is *scalable*, with running time growing linearly with increasing database size and dimension.

Future work can generalize our method to time-evolving data for detecting anomalies over time, where the challenge is to efficiently update the code tables to effectively capture the trending patterns.

Acknowledgement

Research was sponsored by NSF under Grant No. IIS1017415, ARL under Coop. Agreement No. W911NF-09-2-0053, and ADAMS program sponsored by DARPA under Agreements No. W911NF-11-C-0200 and W911NF-11-C-0088. Jilles Vreeken is supported by a Post-Doctoral Fellowship of the Research Foundation – Flanders (FWO).

7. REFERENCES

[1] C. C. Aggarwal and P. S. Yu. Outlier detection for high dimensional data. In *SIGMOD*, 2001.

[2] L. Akoglu, M. McGlohon, and C. Faloutsos. Oddball: Spotting anomalies in weighted graphs. In *PAKDD*, 2010.

[3] A. Arning, R. Agrawal, and P. Raghavan. A linear method for deviation detection in large databases. In *KDD*, 1996.

[4] M. M. Breunig, H. Kriegel, R. T. Ng, and J. Sander. LOF: Identifying density-based local outliers. In *SIGMOD*, 2000.

[5] A. Bronstein, J. Das, M. Duro, R. Friedrich, G. Kleyner, M. Mueller, S. Singhal, and I. Cohen. Using bayes-nets for detecting anomalies in Internet services. In *INM*, 2001.

[6] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection for discrete sequences: A survey. *IEEE Trans. Knowl. Data Eng.*, 24(5):823–839, 2012.

[7] A. Chaudhary, A. S. Szalay, and A. W. Moore. Very fast outlier detection in large multidimensional data sets. In *DMKD*, 2002.

[8] K. Das and J. G. Schneider. Detecting anomalous records in categorical datasets. In *KDD*, 2007.

[9] F. Geerts, B. Goethals, and T. Mielikäinen. Tiling databases. In *DS*, pages 278–289, 2004.

[10] A. Ghoting, S. Parthasarathy, and M. E. Otey. Fast mining of distance-based outliers in high-dimensional datasets. *Data Min. Knowl. Discov.*, 16(3):349–364, 2008.

[11] G. Grimmett and D. Stirzaker. *Probability and Random Processes*. Oxford University Press, 2001.

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

[13] D. Hawkins. *Identification of outliers*. Chapman and Hall, 1980.

[14] P. Kontkanen and P. Myllymäki. MDL histogram density estimation. In *AISTAT*, 2007.

[15] K.-N. Kontonasios and T. De Bie. An information-theoretic approach to finding noisy tiles in binary databases. In *SDM*, 2010.

[16] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer, 1993.

[17] J. Lin, E. J. Keogh, S. Lonardi, and B. Y. Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *DMKD*, pages 2–11, 2003.

[18] M. Mampaey and J. Vreeken. Summarising categorical data by clustering attributes. *Data Min. Knowl. Disc.*, 2012.

[19] P. Miettinen, T. Mielikäinen, A. Gionis, G. Das, and H. Mannila. The discrete basis problem. *IEEE TKDE*, 20(10):1348–1362, 2008.

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

[21] A. Siebes, J. Vreeken, and M. van Leeuwen. Item sets that compress. In *SDM*, pages 393–404. SIAM, 2006.

[22] K. Smets and J. Vreeken. The odd one out: Identifying and characterising anomalies. In *SDM*, pages x–y. SIAM, 2011.

[23] K. Smets and J. Vreeken. SLIM: Directly mining descriptive patterns. In *SDM*, pages 1–12. SIAM, 2012.

[24] N. Tatti and J. Vreeken. Finding good itemsets by packing data. In *ICDM*, pages 588–597, 2008.

[25] J. Vreeken, M. van Leeuwen, and A. Siebes. KRIMP: Mining itemsets that compress. *DAMI*, 23(1):169–214, 2011.

[26] W.-K. Wong, A. W. Moore, G. F. Cooper, and M. M. Wagner. Bayesian network anomaly pattern detection for disease outbreaks. In *ICML*, 2003.