TSum: Fast, Principled Table Summarization

Jieying Chen
Google, Inc.
jieyingchen@google.com

Jia-Yu Pan
Google, Inc.
jypan@google.com

Spiros Papadimitriou
Rutgers University
spapadim@business.rutgers.edu

Christos Faloutsos
Carnegie Mellon University
christos@cs.cmu.edu

ABSTRACT

Given a table where rows correspond to records and columns correspond to attributes, we want to find a *small* number of patterns that succinctly summarize the dataset. For example, given a set of patient records with several attributes each, how can we find (a) that the "most representative" pattern is, say, (*male*, *adult*, $*$), followed by ($*$, *child*, *low-cholesterol*), etc? We propose *TSum*, a method that provides a sequence of patterns ordered by their "representativeness." It can decide both *which* these patterns are, as well as *how many* are necessary to properly summarize the data. Our main contribution is formulating a general framework, *TSum*, using compression principles. *TSum* can easily accommodate different optimization strategies for selecting and refining patterns. The discovered patterns can be used to both represent the data efficiently, as well as interpret it quickly. Extensive experiments demonstrate the effectiveness and intuitiveness of our discovered patterns.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data Mining

General Terms

Algorithms

1. INTRODUCTION

In this paper, we focus on the problem of effectively and succinctly summarizing a table of categorical data. In particular, we are interested in the following questions: How can we find a few patterns, that summarize the majority of the rows? How can we effectively characterize the significance of each pattern, as well as how many patterns are sufficient to summarize the structure of the dataset?

A few examples of applications include: (a) summarizing patient records including symptoms, to find that, e.g., "most patients have in common that they are *male*, *middle-aged*, and have *high cholesterol*, then most patients are *children* with *chickenpox*, and then *female* with *osteoporosis*," and these patterns describe the majority of patients; (b) summarizing a set of click fraud clicks to grasp what these spam clicks have in common (e.g., same IP address or query words, etc.). The patterns cover a big portion of spam clicks and provide useful information to determine the cause/property of the spam attacks. The informal problem definition is **Given** a table with N rows and D categorical attributes, **Find** a few good patterns, that properly summarize the majority of those rows.

To achieve the above, the fundamental question is *how to measure the goodness of a pattern?* We propose to formulate this as a compression problem. Intuitively, a pattern is "good" if it leads to high compression.

The discovered table summary can be used to represent the data more efficiently and to interpret them quickly, by revealing the combinations of features (i.e., patterns) that properly summarize the data. Our main contributions in this paper are the following:

Compression-based formalization: We propose to approach the problem as outlined above. This permits us to discover the patterns and to suggest the appropriate number of patterns without human intervention.
Generality: Our proposed *TSum* framework can accommodate different strategies for searching in the pattern space (e.g., strategies that may incorporate domain knowledge).
Effectiveness: We demonstrate that the discovered patterns are informative and intuitive, by conducting an extensive experimental evaluation.
Scalability: We propose a strategy to discover the patterns, based on our formalization, that is linear on the number of rows N.

1.1 Summarizing Click Fraud Attacks

A click fraud filtering system examines ad clicks and filters clicks that are considered invalid. From time to time, there may be sudden increases of clicks filtered by the system. These sudden increases of invalid clicks may due to (1) new click fraud attacks caught by the system, (2) system misconfigurations at the ad network partner or publisher sites, and so on. These increases of invalid clicks create "spikes" in the ad click traffic.

Our proposed method can automatically generate a summary for the "spikes" in click-fraud invalid clicks, so that an engineer can easily grasp what these spam clicks have in common (e.g., same IP address or query words, etc.). The summary generated by the proposed algorithms consists of the major patterns among the invalid clicks. These patterns provide useful information for determining the cause of an alert and whether there are leakages in the filters.

In the remainder of the paper we first introduce the proposed method, then present experiments, followed by discussion, related work and conclusions.

2. PROPOSED METHOD - OVERVIEW

Our goal is to design a data mining process for finding and ranking patterns that can *properly* summarize a data table. Starting with the informal problem definition given in the introduction, we proceed by formalizing the key concepts of our proposed *TSum* framework. Our formalization is based on the Minimum Description Length (MDL) principle [11, 7]. Furthermore, our proposed framework is modular, allowing users to plug any pattern generation method into the framework. Our framework will subsequently choose those patterns that summarize the table well. Users have the flexibility to incorporate domain knowledge in the process by using a domain-specific pattern generation method.

First, we need to define precisely what we mean by "pattern" and then we define the objective function that determines which are good patterns. Finally, to illustrate the flexibility of *TSum*, we describe two strategies for pattern generation and show that the *TSum* framework can compare and incorporate both.

2.1 Examples and Definitions

The typical application of the proposed *TSum* framework is to summarize a transactional data set with categorical attributes. More specifically, each record in the data set is a tuple (A_1=v_1, ..., A_D=v_D), where A_i are the *attribute names* and v_i are the *attribute values*. In this work we consider *categorical* values, but it is possible to extend to other types. When the order of the attributes is known, we omit their names in a data record (v_1, ..., v_D).

In this work, a *pattern* is a "template" that is matched by a set of records.

Definition 1. (Pattern, matching attributes, and pattern size) *A pattern is a tuple (A_1=u_1, ..., A_D=u_D), where u_i can either be a specific value or a "don't care" symbol (denoted as "$*$"). Attribute A_i is a matching attribute iff $u_i \neq *$. The number of matching attributes in a pattern P is the* size *of the pattern, denoted* size(P).

Following established database terminology [4], the patterns used in this work are *partial match* patterns. A pattern is a conjunction of *conditions*.

For example, consider the patient records in Figure 1, with attributes (*gender*, *age*, *blood-pressure*). Two example patterns are P_1 =(*gender*=*M*, *age*=*adult*, *blood-pressure*= $*$) and P_2 =(*gender*= $*$, *age*=*child*, *blood-pressure*=*low*). To simplify the notation, we either omit the "dont' care" attributes from the pattern representation, i.e., P_1=(*gender*=*M*, *age*=*adult*) and P_2 =(*age*=*child*, *blood-pressure*=*low*), or omit the attribute names, i.e., P_1=(*M*, *adult*, $*$) and P_2=($*$, *child*, *low*).

Furthermore, we say that a data record (v_1, ..., v_D) is *covered* by a pattern P = (u_1, ..., u_D) if either v_i=u_i or u_i=$*$ ("don't care"). A data record is coverable by different patterns, and a pattern can cover more than one record in a data table. In Figure 1, the records in the top of the table are covered by the first pattern, the records in the middle are covered by the second pattern, and the records in the bottom are covered by neither pattern. Intuitively, a human would consider these two patterns as a good summary of the example table: most of these patients are either adult males, or children with low blood pressure.

	gender	age	blood-pressure
ID01	M	adult	normal
ID02	M	adult	low
ID03	M	adult	normal
ID04	M	adult	high
ID05	M	adult	low
ID06	F	child	low
ID07	M	child	low
ID08	F	child	low
ID09	M	teen	high
ID10	F	teen	normal

Figure 1: Example table with patient records. Lines highlight the groupings a human might identify: P_1=(*male*, *adult*, $*$) and P_2=($*$, *child*, *low*).

Definition 2 (Pattern set and list). *A pattern list is an ordered sequence of patterns $\mathcal{P} = [P_1, P_2, \ldots]$, while a pattern set is simply a set of patterns $\mathcal{P} = \{P_1, P_2, \ldots\}$, with no particular order.*

The reason for distinguishing between set and list will become clear later.

A collection of patterns provides a way to *summarize* a data table. For example, in Table 1, the pattern P_1 can be used to summarize the first five records. In particular, the data record ID01 can be summarized as "pattern P_1 with (blood-pressure='normal')". Note that the more a pattern P covers a data record R, the better P can summarize R (i.e., the less additional information we need to give to fully describe the data record R). Similarly, the pattern P_2 can be used to summarize data records ID06, ID07, and ID07. In general, a pattern can be used to summarize all of the data records that it covers. In the example, records ID09 and ID10 are not covered by any pattern, and therefore, cannot be summarized by any pattern.

Our goal is to identify a collection of patterns that *best summarize* a data table.

Problem 1. (Summarizing a table with categorical data) *Given a data set $\mathcal{T}$ that consists of records with categorical attribute values, find patterns $\mathcal{P}$ that summarize the data set $\mathcal{T}$.*

Next, we formalize how we choose $\mathcal{P}$. We propose to use the concept of "encoding cost" to quantify the summarization quality of a given pattern collection. Specifically, we define a cost function that measures the effectiveness of a pattern *list* on compressing the given data table.

Problem 2 (Compression Cost Estimation). *Given a table $\mathcal{T}$ and a list of patterns $\mathcal{P}$ = [$P_1, \ldots, P_m$], estimate the resulting compression cost $CC(\mathcal{T}, \mathcal{P})$.*

Given several competing pattern collections, our proposed framework *TSum* will identify and refine a pattern list that best summarizes the given data set, where the output pattern list achieves the smallest encoding cost.

Problem 3 (Pattern selection). *Given a data table $\mathcal{T}$ and several pattern collections $\mathcal{P}_1, \ldots, \mathcal{P}_m$:* **Identify** *a pattern collection,* **refine** *the patterns, and* **output** *a pattern list that achieves the lowest compression cost for the data table $\mathcal{T}$.*

3. PROPOSED FRAMEWORK

The proposed *TSum* framework uses the compressed size of a data table to choose the most successful patterns. In this section, we first describe how to use a pattern to compress data records, and we introduce our proposed compression encoding scheme. Then, we will describe our "*pattern marshalling*" procedure for ordering and refining an unordered set of patterns $\mathcal{S}$. The output of "pattern marshalling" is an ordered list of patterns $PMA(\mathcal{P})$. We propose a scheme to compute the overall compression cost of a table $\mathcal{T}$ using $PMA(\mathcal{P})$. At the end, *TSum* selects and outputs the pattern list $PMA(\mathcal{P}^*)$ which achieves the best compression cost on the given data table $\mathcal{T}$.

3.1 Pattern Encoding and Compression Cost

We first define a few important concepts. As we mentioned earlier, the *coverage* of a pattern P is the count of records that satisfy P. The *footprint* of a pattern P on a data set $\mathcal{T}$, denoted as $FP(P, \mathcal{T})$, is the number of bits that the pattern can represent and thus save: $FP(P, \mathcal{T}) = Cov(P, \mathcal{T}) * size(P)$. Intuitively, the footprint of a pattern P on a data table $\mathcal{T}$ is correlated to the number of matched attributes covered by the pattern P on $\mathcal{T}$.

We propose an encoding scheme to compress the set of data records covered by a pattern P. The proposed encoding scheme can be illustrated using the example in Figure 1. In the figure, pattern P_1 covers the records ID01, ID02, ID03, ID04, ID05. These five records can be compressed by specifying three components: the pattern P_1, the number of records covered by P_1, and the attribute values that are not covered by P_1.

The *matched* attributes, $matched(P)$, of a pattern P=(u_1, …, u_D), are exactly the attributes that the pattern specifies; the rest we shall refer to as the *don't care* attributes.

Definition 3 (Model cost of a pattern). *The cost of encoding a pattern is:*

$$CC(P) = D + \sum_{i; A_i \in matched(P)} W_i \qquad (1)$$

A pattern P=(u_1, …, u_D) can be encoded in two parts: (1) a D-bit bitmap specifying the matching attributes (bit value 1) and the "don't care" attributes (bit value 0), and (2) the values of the matching attributes. Since W_i is the number of bits for encoding the values of attribute A_i, this completes the definition.

Definition 4 (Compression cost of a pattern). *Given a pattern P and a data table $\mathcal{T}$, the set of records $\mathcal{R}$ covered by P can be compressed in $CC(P, \mathcal{T})$ bits, that is*

$$CC(P, \mathcal{T}) = CC(P) + \log^*(N) + N * \sum_{i, A_i \notin matched(P)} W_i. \quad (2)$$

where $\log^*(N)$ *is the cost in bits to store the self-delimiting version of integer N (see [11]).*

Symbol	Definition
N	Number of records
D	Number of attributes
W_j	Average number of bits for j-th attribute
V_j	Vocabulary = number of distinct values of j-th attribute
$\mathcal{P}$	pattern-list = $(P_1, P_2, \ldots, P_k)$
k	Num. patterns in pattern-list
N_i	Number of records matching the i-th pattern
$\log^*(i)$	Number of bits to encode integer i

Table 1: Symbols and definitions

The compression requires encoding 3 pieces of information: (1) the pattern P, (2) the number of records covered by P, and (3) the attribute values in $\mathcal{R}$ that are not covered by P. Since N is the number of records in $\mathcal{R}$ (the coverage of P on $\mathcal{T}$), the second term follows. In practice, we substitute $\log^*(N)$ with $\log^*(N) = 2 * \lceil \log_2(N+2) \rceil$. The third term is the cost for encoding the attribute values of $\mathcal{R}$ that are not covered by the pattern P. This completes the justification of Definition 4.

Table 1 gives the definitions of symbols that we use.

3.2 Marshalling and Refining a Pattern Set

With our proposed encoding scheme introduced in Definition 4, given a pattern set $\mathcal{P} = \{P_1, \ldots, P_k\}$, we can now compress a data table $\mathcal{T}$ by compressing the records covered by each of the patterns P_i in $\mathcal{P}$. However, in some cases, a data record may be covered by more than one pattern in the given $\mathcal{P}$. How do we resolve such conflict and achieve the best compression? In this section, we introduce a "*pattern marshalling*" procedure to determine a precedence among the patterns in $\mathcal{P}$ and compress a data set using the patterns in their marshalled ordering.

We propose to order the patterns according to the benefit that a pattern can provide in terms of the "*compression savings*."

Definition 5 (Compression saving). *The compression saving of a pattern P on a data table $\mathcal{T}$, denoted as $Saving(P, \mathcal{T})$, is the amount of compression it can achieve. Specifically, it is the difference of bits between the uncompressed and compressed representations of the data records covered by pattern P. Let N be the number of records covered by pattern P and D be the number of attributes in $\mathcal{T}$. Then,*

$$Saving(P, \mathcal{T}) = Benefit(P, \mathcal{T}) - Overhead(P, \mathcal{T}). \quad (3)$$

Intuitively, $Saving(P, \mathcal{T})$ consists of two terms: (1) the benefit of using the pattern to represent data records,

$$Benefit(P, \mathcal{T}) = (N-1) * \sum_{i, A_i \in matched(P)} W_i, \qquad (4)$$

and (2) the overhead of using the pattern in the representation,

$$Overhead(P, \mathcal{T}) = D + \log^*(N) \qquad (5)$$

To resolve the conflict when a data record is covered by more than one pattern, we propose to "marshall" the patterns according to their compression saving. The pattern with the highest compression saving will take precedence over other patterns. Therefore, when there is a coverage conflict, the data record will be assigned to the pattern with the highest compression saving.

More specifically, given a set of patterns $\mathcal{S}$, the "pattern marshalling algorithm (*PMA*)" iteratively picks a pattern from $\mathcal{S}$ which has the highest compression saving. After each iteration, the data records that have been covered by the patterns chosen so far will be removed from consideration. The compression saving of the patterns at the next iteration is considered only on the remaining data records which have not been covered by any chosen pattern. In other words, at each iteration, the pattern chosen is that with the best "*incremental compression saving.*" The *PMA* algorithm repeats the iterations until no pattern has positive incremental compression saving.

DEFINITION 6 (RESIDUE DATA TABLE). *Given a data table $\mathcal{T}$ and a pattern collection $\mathcal{P}$, we defined the "residue data table" of $\mathcal{T}$, with respect to $\mathcal{P}$, as the data records that are not covered by any of the patterns in $\mathcal{P}$. We denote the residue data table as $\mathcal{T} \setminus \mathcal{P}$.*

DEFINITION 7 (INCREMENTAL COMPRESSION SAVING). *Given a list of patterns $\mathcal{P} = [P_1, \ldots, P_l]$ and a data set $\mathcal{T}$, the "incremental compression saving" of a new pattern P, with respect to $\mathcal{P}$, is defined as:*

$$Saving(P, \mathcal{T} \setminus \mathcal{P}) = Benefit(P, \mathcal{T} \setminus \mathcal{P}) - Overhead(P, \mathcal{T} \setminus \mathcal{P}). \quad (6)$$

Intuitively, $Saving(P, \mathcal{T} \setminus \mathcal{P})$ is essentially the compression saving of P on the residue data table $\mathcal{T} \setminus \mathcal{P}$.

Given a set of patterns, the *PMA* algorithm ranks the patterns by their (incremental) compression saving, and outputs a list of patterns that are useful in compressing the data table in question. Algorithm 1 shows the pseudocode of the *PMA* algorithm.

To compress the entire data table $\mathcal{T}$ using a "marshalled" pattern list $\mathcal{P} = [P_1, \ldots, P_k]$, we propose a procedure similar to the PMA algorithm. The procedure starts from the first pattern P_1, compresses the records in $\mathcal{T}$ covered by P_1, and then continues to the next pattern and compresses the records in the residue data set.

In practice, we always include the "don't care" pattern, i.e., `(*, ..., *)` in the pattern list when compressing a data table. The "don't care" pattern can be used in representing the data records that are not covered by any other pattern in the given pattern list.

We now can define the total compression cost of a data table $\mathcal{T}$, using a given set of (marshalled) patterns $\mathcal{P}$:

DEFINITION 8 (COMPRESSION COST OF A DATA TABLE). *Given a data table $\mathcal{T}$ and a "marshalled" pattern list $\mathcal{P} = [P_1, \ldots, P_k]$, the total compression cost of $\mathcal{T}$ using $\mathcal{P}$ is:*

$$CC(\mathcal{P}, \mathcal{T}) = \sum_{i=1}^{k} CC(P_i, \mathcal{T} \setminus \{P_1, \ldots, P_{i-1}\}). \quad (7)$$

In the equation, $CC(P_i, \mathcal{T} \setminus \{P_1, \ldots, P_{i-1}\})$ is the compression cost of P_i on the residue data table $\mathcal{T} \setminus \{P_1, \ldots, P_{i-1}\}$ (see Definition 4 and Definition 6).

DEFINITION 9 (THE *TSum* FRAMEWORK). *Given a data table $\mathcal{T}$, and an input collection of pattern sets $\mathcal{I} = \{\mathcal{P}_1, \ldots, \mathcal{P}_k\}$, the* TSum *framework selects and refines (through marshalling) a pattern set $\mathcal{P}^*$ that best summarizes the table $\mathcal{T}$ in terms of compression saving. In the functional form,*

$$\mathcal{P}^* = TSum(\mathcal{T}, \mathcal{I} = \{\mathcal{P}_1, \ldots, \mathcal{P}_k\}), \quad (8)$$

where

$$\mathcal{P}^* = \arg\min_{\mathcal{P} in \mathcal{I}} CC(\mathcal{T}, PMA(\mathcal{P})). \quad (9)$$

The final output of the TSum *framework is the marshalled pattern list $PMA(\mathcal{P}^*)$.*

4. PATTERN GENERATION STRATEGIES

The *TSum* framework takes in one or more pattern sets and selects a pattern set (after "marshalling") that is best in summarizing a data table. The input pattern sets are therefore important regarding the quality of the summarization. In this work, the best summarization for a data table corresponds to the pattern set that best compresses the data table. Unfortunately, there are exponentially many possible pattern sets that can compress the data table, and it is difficult to find the one that achieves the best compression.

In this section, we introduce two strategies for generating a pattern set that captures the major characteristics of a data table, namely, the "Spike-Hunting" strategy and the "Local-Expansion" strategy. The Spike-Hunting strategy identifies the major multi-dimensional "spikes" in the given data set, where the "spikes" usually correspond to (and, summarize) major groups of data records in the data set. On the other hand, the Local-Expansion strategy tries to "grow" patterns to increase the compression saving on the data records.

4.1 The Spike-Hunting Strategy

Real world data sets usually contain clusters among attribute values. These clusters sometimes capture key properties of the data set and provide a good summary of the data set. The most significant clusters of attribute values also reveal themselves as "spikes" in the multi-dimensional attribute space. In this work, each spike is represented as a set of conditions such as $\{A = a, B = b, \ldots\}$, which is the same representation as that of a pattern (Definition 1).

We propose a pattern generation strategy ("Spike-Hunting") to find these attribute-value spikes (patterns) in a data table. At the high level, the procedure of the "Spike-Hunting" strategy is as follows: First, pick an attribute A that is "spiky" with respect to the marginal distribution of A, and then find the attribute values of A that forms the spike (say, $A = a$). Then, recursively repeat the same steps of picking spiky attributes and finding spike values, from the attributes that have not been considered yet and conditioned on only the data records that satisfy the spiky conditions found so far. Algorithm 2 shows the algorithm details.

Intuitively, the Spike-Hunting algorithm progressively searches for a multi-dimensional spike starting from single attributes. When a spike is found in the marginal distribution of one attribute, the algorithm continues to refine that spike by examining the marginal distributions on other attributes, conditioned on those attributes (and their corresponding spike values) that have already been included in the spike pattern.

In the Spike-Hunting algorithm, we use the entropy of an attribute's (conditional) marginal distribution to determine if there is a spike on that attribute. Then, we determine the values in the conditional marginal distribution that belong

```
Algorithm:(Pattern Marshalling) $PMA$ ($\mathcal{S}$, T)
Input: $\mathcal{T}$: Table, $N$ rows, $D$ attributes
Input: $\mathcal{S}$: Set of $m$ patterns $\{P_1, \ldots, P_m\}$
Output: The best pattern-list $\mathcal{P} = [P_{i_1}, \ldots P_{i_k}]$
$\mathcal{P} = \varnothing$;
$\mathcal{R} = \mathcal{S}$;
while $\mathcal{R} \neq \varnothing$ do
    //Select the pattern with the top "incremental" compression saving.
    Sort $\mathcal{R}$ by the $Saving(P, \mathcal{T} \setminus \mathcal{P})$, w.r.t.  P ; Let the top pattern be $P_{top} = \arg\max_{P \in \mathcal{R}} Saving(P, \mathcal{T} \setminus \mathcal{P})$;
    Let the incremental compression saving be $b_{top} = \max_{P \in \mathcal{R}} Saving(P, \mathcal{T} \setminus \mathcal{P})$
    //$b_{top}$ could be non-positive, if so, ignore the pattern $P_{top}$.
    if $b_{top} > 0$ then
        //Append $P_{top}$ at the end of the output pattern list $\mathcal{P}$.
        $\mathcal{P} \leftarrow [\mathcal{P}, P_{top}]$ ;
    end
    //Update remaining patterns.
    $\mathcal{R} \leftarrow \mathcal{R} \setminus P_{top}$ ;
end
//$\mathcal{P}$ now is a sorted list of patterns.
return $\mathcal{P}$
```

Algorithm 1: Pattern-marshalling algorithm (PMA): Picks patterns with the best incremental compression saving, and skips the ones with zero or negative saving.

to the spike (i.e., the values which are by far more probable than other values). We take the top few values of A that capture the majority of the energy (variance) of the (conditional) distribution as the spike values.

We note that the high-level procedure of Spike-Hunting algorithm looks similar to the CLIQUE algorithm [3], if we viewed each distinct attribute value as an "interval" when using CLIQUE. One difference between the proposed Spike-Hunting algorithm and CLIQUE is the objective function used in selecting the (subspace) dimensions. In particular, Spike-Hunting uses the entropy function, followed by the energy ratio check, whereas CLIQUE uses a user-specified density threshold, to identify dimensions having dense regions. In our experiments (Section 5), we will show that *TSum* can work with any pattern generation strategy, whether it be Spike-Hunting (a strategy with a procedure similar to that of the CLIQUE algorithm) or any other strategy, and compare and select the appropriate patterns from the ones generated by these different strategies.

4.2 The Local-Expansion Strategy

Based on the proposed compression cost function (Definition 4), we also design a pattern generation strategy that directly looks for patterns that could minimize the compression cost. Our approach for finding each such pattern is a "best-first" approach: we start from single attributes first and find a single-condition pattern $P = \{(A = a)\}$ that has the best compression saving, and then expand the pattern by adding other conditions until the compression cost can not be improved. To find the next pattern, the same procedure is repeated, but only on the "residue" data table, the part of the data table not covered by the patterns found so far. Because this strategy generates a pattern by expanding in a "best-first" fashion, we called this strategy the "Local-Expansion" method. Algorithm 3 shows the details of this method.

Figure 2 illustrates the process of pattern expansion (Algorithm 4). A pattern P_1 will be expanded to P_2 by adding an additional condition, if the compression saving (indicated by the colored areas in the figure) increases. According to Definition 5, the compression saving of a pattern is dominated by the term $Benefit(P, \mathcal{T})$, which is closely related to the colored area in the figure.

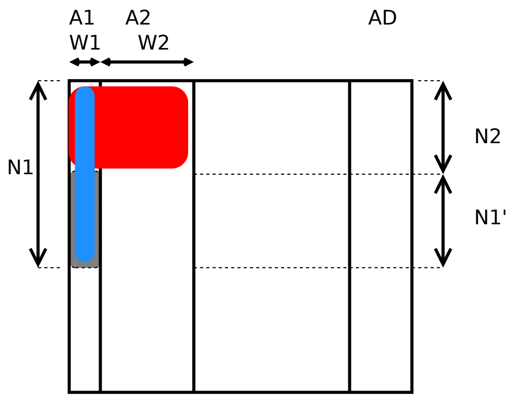

Figure 2: Illustration of pattern expansion: The blue area indicates the saving that a pattern P_1 can achieve. The Local-Expansion algorithm will expand P_1 to P_2 (depicted by the red area) if P_2 has bigger compression saving (that is, roughly speaking, if the red area is bigger than the blue area).

4.3 General Pattern Generation Strategy

The two strategies that we presented above, the Spike-Hunting method and the Local-Expansion method, are just two of the many possible methods for generating pattern sets to summarize a data table. We would like to emphasize that the proposed *TSum* framework can work with any pattern set, regardless of whether it is generated by our proposed strategies or not.

The two strategies we proposed are two special cases of a more general strategy for pattern generation. Conceptually, the Spike-Hunting and Local-Expansion algorithms are both

```
Algorithm:Spike-Hunting ($\mathcal{T}$)
Input: A table $\mathcal{T}$ with $N$ rows and $D$ attributes
Output: A list of patterns $\mathcal{P}$
$\mathcal{P} = \varnothing$;
foreach attribute $A$ in $\mathcal{T}$ do
    //Check whether $A$'s marginal distribution is spiky.
    if $Entropy(A) \leq threshold$ then
        Identify the values forming the spike $\mathcal{SV} = \{a_1, a_2, \ldots\}$;
        foreach value $a$ in $\{a_1, \ldots\}$ do
            //Create a table conditioned on $A = a$.
            Let $\mathcal{T}'$ be the table which contains records with $A = a$, and does not include the attribute $A$;
            //Recusively find the spikes.
            SubPatterns = Spike-Hunting ($\mathcal{T}'$);
            foreach pattern $P$ in SubPatterns do
                $P_{new} = \{A = a\} \cup P$ ;
                $\mathcal{P} = \mathcal{P} \cup P_{new}$ ;
            end
        end
    end
end
return $\mathcal{P}$
```

Algorithm 2: The Spike-Hunting algorithm.

```
Algorithm:Local-Expansion ($\mathcal{T}$)
Input: A table $\mathcal{T}$ with $N$ rows and $D$ attributes
Output: A list of patterns $\mathcal{P}$
$\mathcal{P} = \varnothing$;
while $\mathcal{T}$ is not empty do
    //Expand from an empty pattern (Algorithm 4).
    $P = Expand(\mathcal{T}, \varnothing)$
    //Stop, if we cannot achieve more compression saving.
    if $P == \varnothing$ then
        break;
    end
    //Found a new pattern.
    $\mathcal{P} = \mathcal{P} \cup \{P\}$;
    //Update the table by removing records covered by $P$.
    $\mathcal{T} = \mathcal{T} - \{$ tuples covered by $P\}$;
end
return $\mathcal{P}$
```

Algorithm 3: The Local-Expansion algorithm

doing a traversal on a lattice space of all possible patterns. The two algorithms have different criteria to decide which branch to traverse first and when to stop the traversal.

Figure 3 illustrates the lattice structure of our space, using the 'patient' toy dataset of Figure 1. We start from the empty pattern-list ((*,*,*)), and we proceed with two types of expansions: (a) *condition-expansion*, where we add one more condition to one of the existing patterns of the pattern-list (see thin-lines in Figure 3), and (b) *pattern-expansion*, where we append one more pattern to the pattern list $\mathcal{P}$ (thick line in Figure 3)

The Spike-Hunting and Local-Expansion methods use different criteria when doing condition-expansion. The Spike-Hunting method does the condition-expansion step when it detects additional spiky attribute values, and the expansion to a new attribute can have multiple choices of values (if the spike consists of more than one value). On the other hand, the Local-Expansion method expands with a new condition if adding the new condition can increase the compression saving of the pattern.

Both methods do a series of condition-expansion operations until the expansion of a pattern stops, then a pattern-expansion operation is performed to find the next pattern. Both methods do not revisit a pattern after it is finalized. Regarding the pattern-expansion operation, the Spike-Hunting method starts a new pattern by back-tracking from the pattern finalized before. On the other hand, the Local-Expansion method starts a new pattern from scratch, but considers only the data records on the residue data table, not on the entire table.

5. EXPERIMENTS

In this section, we consider the following data sets from UC-Irvine repository.

- `ADULT50-`: Adults with income less than 50K (24720 records, 7 attributes).
- `ADULT50+`: Adults with income greater than 50K (7841

```
Algorithm:$Expand(\mathcal{T}, P)$
Input: $\mathcal{T}$: A table with $N$ rows and $D$ attributes
Input: $P$: A pattern from which to start the expansion
Output: $P_{expand}$: An expanded pattern with a best compression saving
foreach attribute $A$ not included in $P$ do
    foreach value $a$ of $A$ do
        Let $P'$ be the expanded pattern by appending $(A = a)$ to $P$.;
        Compute the compression saving of $P'$ on $\mathcal{T}$.
    end
end
Let $P_{best}$ be the expanded pattern with the biggest compression saving $Saving(\mathcal{T}, P_{best})$.;
if $Saving(\mathcal{T}, P_{best}) > Saving(\mathcal{T}, P)$ then
    //Continue the expansion recursively.
    return $Expand(\mathcal{T}, P_{best})$
end
else
    return $P$
end
```

Algorithm 4: Expanding a pattern to improve compression saving. The best pattern may be the empty pattern, i.e., no pattern: (*, *, . . . , *).

Dataset	ADULT50-	ADULT50+	CENSUS	NURSERY
Best	Local Expansion	Local Expansion	Spike Hunting	Local Expansion
Bits	447,477	124,252	2,037,398	216,126
Coverage	100%	100%	43.81%	100%

Table 2: Winning heuristic, for each dataset

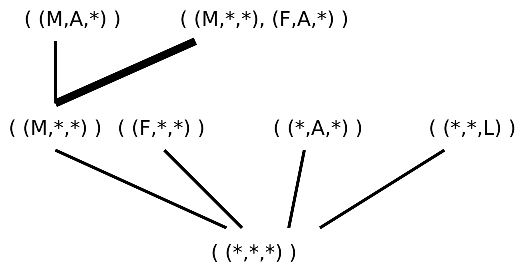

Figure 3: Lattice organization of our search space. Light line: "condition-expansion"; thick line:"pattern-expansion".

records, 7 attributes).

- NURSERY: Nursery data (12960 records, 9 attributes).

We run *TSum* with the two proposed pattern generation strategies, Spike-Hunting and Local-Expansion. Table 2 reports the winning strategy and the corresponding bit cost. Notice that Local-Expansion usually wins, but not always. In general, Local-Expansion tends to choose patterns with much broader coverage, because these typically give larger compression saving. On the other hand, Spike-Hunting tends to produce patterns with much fewer "don't cares" ($*$), which usually correspond to fairly focused spikes and, hence, have lower coverage. In other words, Local-Expansion tries to cover as many rows of the table as possible, whereas Spike-Hunting will mostly cover "hot-spots" in the table. Shorter patterns help reduce the bit cost by covering a larger number of rows, whereas longer patterns cover more attributes and therefore need fewer bits to describe each row they cover. Spike-Hunting will try to explore as many attributes as possible by considering all rows in the table, whereas Local-Expansion progressively excludes rows by considering only the remainder table. If "hot-spots" are fairly spread out, then Spike-Hunting may succeed in finding patterns that achieve a lower bit cost. In general, however, Local-Expansion will tend to produce patterns with lower bit cost.

Finally, we should point out that, our pattern marshalling method will refine any pattern set and rank patterns with respect to how "informative" they are, whether they are produced by Local-Expansion, Spike-Hunting, or another strategy. Next we give more details for each dataset.

ADULT *dataset.* This dataset is from the UCI repository, and is already naturally subdivided into two classes: people above or below income of \$50K (ADULT50+ and ADULT50-, respectively).

For ADULT50-, the Local-Expansion algorithm wins, with a bit cost of $447,477$. The top 3 patterns are listed at Table 3. In the table, "footprint" indicates the number of bits used in encoding the attribute values matched by a pattern.

For comparison, in Table 4 we also show the top patterns selected by Spike-Hunting, even though this is not the winning strategy on this data set (total bit cost 470,406). We clearly see that these patterns have much fewer don't care attributes. Both sets of patterns make sense. Those selected by Local-Expansion show the important broad properties in the dataset (as reflected by the lower total bit cost), whereas those selected by Spike-Hunting are more focused on hot-spots in the data set that are important (i.e., hot spots that

age	workclass	education	marital	occup.	race	sex	Footprint	Benefit	Coverage
∗	private	∗	∗	∗	white	∗	104104	104061	60.16%
∗	∗	∗	∗	∗	white	∗	17481	17442	23.57%
∗	private	∗	∗	∗	black	∗	13657	13614	7.89%

Table 3: Top three patterns of the winning strategy (Local-Expansion) selected by *TSum* for `ADULT50-` data.

age	workclass	education	marital-status	occup.	race	sex	Footprint	Benefit	Coverage
20–30	private	some-college	never-married	∗	white	∗	22192	22137	4.72%
20–30	private	HS-grad	never-married	∗	white	∗	21071	21016	4.49%
30–40	private	HS-grad	married-civ-spouse	∗	white	M	12680	12624	2.56%

Table 4: Top three patterns (ranked by our pattern marshalling algorithm) using Spike-Hunting on `ADULT50-`.

still achieve notable reduction in bit cost).

In summary, both algorithms produce reasonable summaries, with broad characteristics that reflect the design goals of each optimization heuristic. Furthermore, the patterns produced by the winning strategy (Local-Expansion), as chosen by *TSum*, make sense as a good summary of the broad characteristics of the dataset: people at this income bracket are usually younger, there is no dominating marital status, and no dominating occupation. This is naturally reflected by our proposed bit cost, and agrees with intuition.

Before we continue, we note that, for each pattern, the footprint is a good approximation of the benefit, i.e., the actual number of compression bits. Thus, for the remaining datasets we just show footprint.

For `ADULT50+`, *TSum* again chooses Local-Expansion as the winning strategy, with a total bit cost of 124, 252. Table 5 further shows the top three patterns selected by *TSum*. The observations when comparing against Spike-Hunting are very similar to those for `ADULT50-`. Finally, the patterns again make sense and form a good summary of the broad characteristics: people in this income bracket are usually older, and thus married.

`CENSUS` *dataset.* Here Spike-Hunting is chosen as the winning strategy by *TSum*, with a bit cost of 2, 037, 398 (versus 2, 210, 319 bits for Local-Expansion). As noted earlier, we believe this is because the spikes in this dataset are more broad. In more detail, Local-Expansion progressively searches for patterns, favoring those with higher coverage, and subsequenty excluding the covered rows in its search. On the other hand, Spike-Hunting tends to give patterns with more attribute values (and fewer "don't cares") at the cost of typically lower coverage. However, each covered record can be described with fewer bits, since the Spike-Hunting patterns tend to include more attributes. In summary, both algorithms produce reasonable summaries according to the design of their optimization heuristics (and the marshalling process of *TSum* can produce a good ranking of patterns from either strategy) and, depending on the structure of the dataset, one strategy may be overall better than others. Again, the total bit cost reflects the properties of good summaries of the broad characteristics.

Nursery dataset. The nursery dataset is essentially synthetic, with mostly independently uniform random distributions, and thus it is interesting to note that *TSum* finds no meaningful summary because none exists. The results themselves are not interesting, and thus omitted.

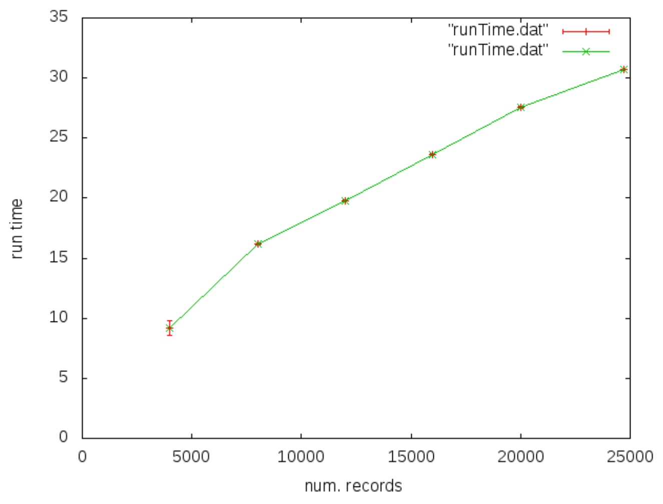

Figure 4: Linear scalability: run time vs number of records.

5.1 Scalability

Figure 4 shows that the running time of our algorithm is linear on N (number of records). The dataset was `ADULT50-` and we sampled records, with sample sizes from 4000 to 24720. We report the average running time over 5 runs, along with the error-bars (red) for one standard deviation.

6. BACKGROUND AND RELATED WORK

Clustering and co-clustering There are numerous such algorithms including methods for numerical attributes (Birch [15], CURE [8], DBSCAN [12], and OPTICS [5]), categorical attributes high dimensional variations (subspace clustering [3] and projected clustering [2]). All of these methods finds cluster centers alone, or cluster centers together with subsets of dimensions. However, they generally do not produce patterns or rules. Furthermore, most of the above approaches need some user-specified parameter or threshold.

Association Rules Association rules typically operate on sets (or, binary attributes), such as "market baskets." However, (a) they need user-defined parameters (e.g., min-support, etc), and (b) they generate too many rules, which defeats the purpose of summarization. Several publications try to address this issue, e.g., [1, 13, 14, 16]. Most of this work needs some user-defined parameter. More importantly, the pattern sets that any of these approaches produce can be

age	workclass	education	marital-status	occup.	race	sex	Footprint	Coverage
$*$	$*$	$*$	married-civ-spouse	$*$	white	male	38108	69.43%
$*$	private	$*$	$*$	$*$	white	$*$	6360	13.52%
$*$	$*$	$*$	married-civ-spouse	$*$	$*$	$*$	2550	10.84%

Table 5: Top three patterns of the winning strategy (Local-Expansion) selected by *TSum* for `ADULT50+` data.

incorporated in our proposed framework, as an alternative optimization strategy, and *TSum* will rank those patterns and choose an appropriate number to form a good summary of the general characteristics of the data set.

SVD, LSI and dimensionality reduction These methods operate on matrices, and have been successful in numerous applications, e.g., [6, 9]. However, in most cases (a) some user-defined parameters are required (e.g., how many singular values/vectors to keep), and (b) the resulting patterns are hard to interpret (vectors in high-dimensional space).

Compression, MDL, and Kolmogorov complexity Lempel-Ziv [10] and LZW are suitable for strings. However, since we would need to linearize the table, an ordering of both rows and columns would be necessary, which is not an easy problem. MDL [11, 7] is a vital concept. At the high level, *TSum* exactly tries to find a pattern language and the corresponding patterns, so that the table (using the patterns) together with the chosen patterns themselves can be compressed as well as possible.

As far as *TSum* is concerned, any method that generates patterns from a given data table can work side-by-side within the *TSum* framework. The pattern generation methods (either a clustering algorithm or a pattern enumeration method) provide "pattern candidates" for *TSum*, and the *TSum* framework will pick the patterns that are suitable for describing the data table.

7. CONCLUSIONS

In this paper we present *TSum*, a method to efficiently discover patterns that properly summarize a data table. Our main contribution is the rigorous formalization, based on the insight that table summarization can be formalized as a compression problem. More specifically, (i) we propose a compression cost objective to effectively discover proper summaries that reveal the global structure of the data table, and (ii) show how different optimzation strategies can be accomodated by our framework, and (iii) propose two strategies, Spike-Hunting and Local-Expansion, each designed with different goals for their optimization heuristics.

8. REFERENCES

[1] F. N. Afrati, A. Gionis, and H. Mannila. Approximating a collection of frequent sets. In *KDD*, 2004.

[2] C. C. Aggarwal and P. S. Yu. Finding generalized projected clusters in high dimensional spaces. In *SIGMOD*, 2000.

[3] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data. *DMKD*, 11(1), 2005.

[4] A. Aho and J. Ullman. Optimal partial match retrieval when fields are independently specified. *ACM TODS*, 4(2):168–179, June 1979.

[5] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. OPTICS: Ordering points to identify the clustering structure. In *SIGMOD Conference*, pages 49–60, 1999.

[6] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, Sept. 1990.

[7] P. D. Grünwald. *Advances in Minimum Description Length: Theory and Applications*, chapter A Tutorial Introduction to the Minimum Description Length Principle. MIT Press, 2005. `http://homepages.cwi.nl/~pdg/ftp/mdlintro.pdf`.

[8] S. Guha, R. Rastogi, and K. Shim. CURE: An e fficient clustering algorithm for large databases. In *SIGMOD Conference*, pages 73–84, Seattle, Washington, 1998.

[9] F. Korn, H. Jagadish, and C. Faloutsos. Efficiently supporting ad hoc queries in large datasets of time sequences. *ACM SIGMOD*, pages 289–300, May 13-15 1997.

[10] A. Lempel and J. Ziv. Compression of two-dimensional images. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, pages 141–154. Springer-Verlag, Malatea, Italy, June 18-22 1984. Published as NATO ASI Series, volume F12.

[11] J. Rissanen. Minimum description length principle. In S. Kotz and N. L. Johnson, editors, *Encyclopedia of Statistical Sciences*, volume V, pages 523–527. John Wiley and Sons, New York, 1985.

[12] J. Sander, M. Ester, H.-P. Kriegel, and X. Xu. Density-based clustering in spatial databases: The algorithm GDBSCAN and its applications. *Data Mining and Knowledge Discovery*, 2(2):169–194, 1998.

[13] G. I. Webb. Discovering significant rules. In *KDD*, 2006.

[14] X. Yan, H. Cheng, J. Han, and D. Xin. Summarizing itemset patterns: a profile-based approach. In *KDD*, 2005.

[15] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An efficient data clustering method for very large databases. *ACM SIGMOD*, pages 103–114, May 1996.

[16] F. Zhu, X. Yan, J. Han, P. S. Yu, and H. Cheng. Mining colossal frequent patterns by core pattern fusion. In *ICDE*, 2007.